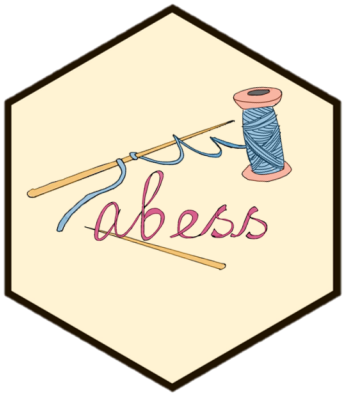
ABESS

Release 0.4.5

abess-team

Jan 14, 2023

USING ABESS


abess

CHAPTER

ONE

OVERVIEW

`abess` (Adaptive BEst Subset Selection) library aims to solve general best subset selection, i.e., find a small subset of predictors such that the resulting model is expected to have the highest accuracy. The selection for best subset shows great value in scientific researches and practical applications. For example, clinicians want to know whether a patient is healthy or not based on the expression level of a few of important genes.

This library implements a generic algorithm framework to find the optimal solution in an extremely fast way[1]. This framework now supports the detection of best subset under: linear regression, (multi-class) classification, censored-response modeling[2], multi-response modeling (a.k.a. multi-tasks learning), etc. It also supports the variants of best subset selection like group best subset selection[3] and nuisance best subset selection[4]. Especially, the time complexity of (group) best subset selection for linear regression is certifiably polynomial[?3].

[1] Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang (2020). A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123.

[2] Pölsterl, S (2020). scikit-survival: A Library for Time-to-Event Analysis Built on Top of scikit-learn. J. Mach. Learn. Res., 21(212), 1-6.

[3] Yanhang Zhang, Junxian Zhu, Jin Zhu, and Xueqin Wang (2022). A Splicing Approach to Best Subset of Groups Selection. INFORMS Journal on Computing (Accepted). doi:10.1287/ijoc.2022.1241.

[4] Qiang Sun and Heping Zhang (2020). Targeted Inference Involving High-Dimensional Data Using Nuisance Penalized Regression, Journal of the American Statistical Association, DOI: 10.1080/01621459.2020.1737079.

CHAPTER

TWO

QUICK START

The `abess` software has both Python and R's interfaces. Here a quick start will be given and for more details, please view: Installation.

2.1 Python package

Install the stable `abess` Python package from PyPI:

```
$ pip install abess
```

or conda-forge

```
$ conda install abess
```

Best subset selection for linear regression on a simulated dataset in Python:

```
from abess.linear import abessLm
from abess.datasets import make_glm_data
sim_dat = make_glm_data(n = 300, p = 1000, k = 10, family = "gaussian")
model = abessLm()
model.fit(sim_dat.x, sim_dat.y)
```

See more examples analyzed with Python in the Python tutorials.

2.2 R package

Install `abess` from R CRAN by running the following command in R:

```
install.packages("abess")
```

Best subset selection for linear regression on a simulated dataset in R:

```
library(abess)
sim_dat <- generate.data(n = 300, p = 1000)
abess(x = sim_dat[["x"]], y = sim_dat[["y"]])
```

See more examples analyzed with R in the R tutorials.

CHAPTER

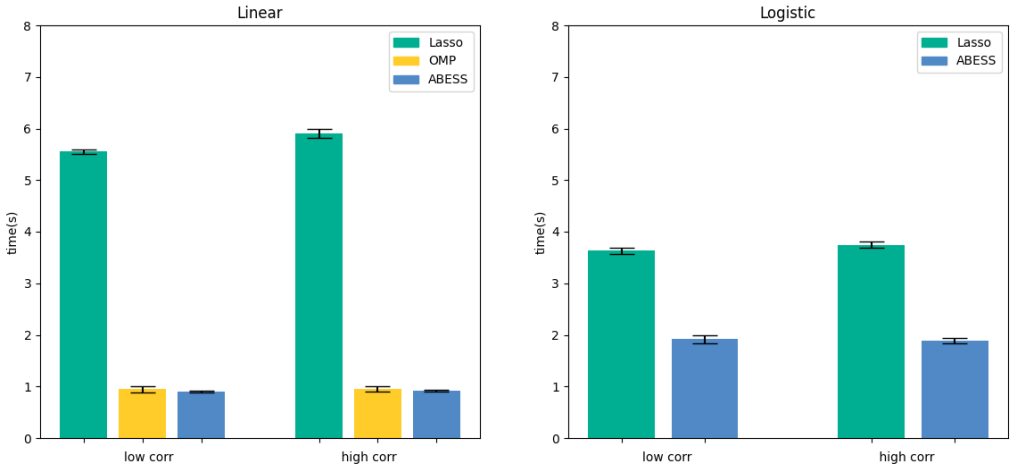
THREE

RUNTIME PERFORMANCE

To show the power of `abess` in computation, we assess its timings of the CPU execution (seconds) on synthetic datasets, and compare them to state-of-the-art variable selection methods. The variable selection and estimation results as well as the details of settings are deferred to Python performance and R performance. All computations are conducted on a Ubuntu platform with Intel(R) Core(TM) i9-9940X CPU @ 3.30GHz and 48 RAM.

3.1 Python package

We compare `abess` Python package with scikit-learn on linear regression and logistic regression. Results are presented in the below figure:

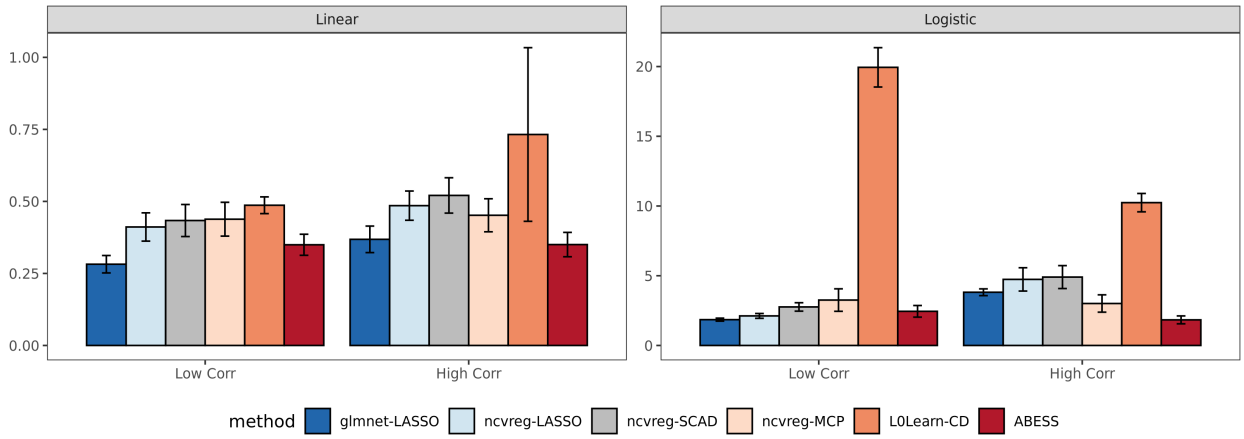


It can be seen that `abess` uses the least runtime to find the solution. The results can be reproduced by running the commands in shell:

```
$ python ./docs/simulation/Python/timings.py
```

3.2 R package

We compare `abess` R package with three widely used R packages: *glmnet*, *ncvreg*, and *L0Learn*. We get the runtime comparison result:



Compared with the other packages, `abess` shows competitive computational efficiency, and achieves the best computational power when variables have a large correlation.

Conducting the following commands in shell can reproduce the above results:

```
$ Rscript ./docs/simulation/R/timings.R
```

CHAPTER

FOUR

OPEN SOURCE SOFTWARE

`abess` is a free software and its source code is publicly available in Github. The core framework is programmed in C++, and user-friendly R and Python interfaces are offered. You can redistribute it and/or modify it under the terms of the GPL-v3 License. We welcome contributions for `abess`, especially stretching `abess` to the other best subset selection problems.

CHAPTER

FIVE

WHAT'S NEW

Version 0.4.5:

- *abess* Python package can be installed via *conda*.
- Easier installation for Python users
- `abess` R package is is highlighted as one of the core packages in CRAN Task View: Machine Learning & Statistical Learning.
- Support predicting survival function in *abess.linear.CoxPHSurvivalAnalysis*.
- Rename estimators in Python. Please check here.

New best subset selection tasks:

- Generalized linear model for ordinal regression (a.k.a rank learning in some machine learning literature).

CHAPTER

SIX

CITATION

If you use `abess` or reference our tutorials in a presentation or publication, we would appreciate citations of our library[5].

Zhu Jin, Xueqin Wang, Liyuan Hu, Junhao Huang, Kangkang Jiang, Yanhang Zhang, Shiyun Lin, and Junxian Zhu. "abess: A Fast Best-Subset Selection Library in Python and R." Journal of Machine Learning Research 23, no. 202 (2022): 1-7.

The corresponding BibteX entry:

```
@article{JMLR:v23:21-1060,
  author  = {Jin Zhu and Xueqin Wang and Liyuan Hu and Junhao Huang and Kangkang Jiang␣
↪and Yanhang Zhang and Shiyun Lin and Junxian Zhu},
  title   = {abess: A Fast Best-Subset Selection Library in Python and R},
  journal = {Journal of Machine Learning Research},
  year    = {2022},
  volume  = {23},
  number  = {202},
  pages   = {1--7},
  url     = {http://jmlr.org/papers/v23/21-1060.html}
}
```

[5] Zhu Jin, Xueqin Wang, Liyuan Hu, Junhao Huang, Kangkang Jiang, Yanhang Zhang, Shiyun Lin, and Junxian Zhu. "abess: A Fast Best-Subset Selection Library in Python and R." Journal of Machine Learning Research 23, no. 202 (2022): 1-7.

CHAPTER

SEVEN

REFERENCES

7.1 Installation

7.1.1 Stable release

Python

To install `abess` on Python, you can simply get the stable version with:

```
$ pip install abess
```

If you don't have pip installed, this Python installation guide can guide you through the process.

R

To install stable version into R environment, run the command:

```
install.packages("abess")
```

7.1.2 Latest version

This page gives instructions on how to build and install `abess` from the source code. If the instructions do not help for you, please feel free to ask questions by opening an issue.

First of all, clone our the latest github project to your device:

```
$ git clone https://github.com/abess-team/abess.git
$ cd abess
```

Next, there have different processing depend on the programming langulage you prefer.

Python

Before installing `abess`, some dependent libraries should be installed first, which may be a little different in different platforms:

- **Linux**: `$ sudo apt install bash` (for Ubuntu, but other Linux systems are similar);
- **Windows**: `$ choco install git` (using Chocolatey), or manually install the software and add them into PATH; Windows user will also need to download Microsoft C++ Build Tools, and then install the "Desktop development with C++" module inside.
- **MacOS**: `$ brew install bash` (using Homebrew).

Then, you need to install `pybind11` via `pip install pybind11` (or other methods). After that, we can manually install `abess` by conducting command:

```
$ cd ./python
$ pip install .
```

or

```
$ cd ./python
$ python setup.py install --user
```

If it finishes with `Finished processing dependencies for abess`, the installation is successful.

Alternatively, if you would like to develop `abess`, install `abess` in editable mode (it is very convenient for development):

```
$ cd ./python
$ pip install -e .
```

or

```
$ cd ./python
$ python setup.py develop --user
```

Note that some may meet "Permission denied" problem like this issue when installing with `pip install -e .`. There are three solutions: 1. run the command as administrator; 2. feel free to use `python setup.py develop --user` instead; 3. try to edit `setup.py` like here (not recommend).

R

To install the development version, some dependencies need to be installed. Before installing `abess`, some dependencies should be installed first, which may be a little different in different platforms:

- **Linux**: `$ sudo apt install autoconf` (for Ubuntu, other Linux systems are similar);
- **Windows**: install Rtools.
- **MacOS**: `$ brew install autoconf`.

Then, you need to install R library dependencies `Rcpp` and `RcppEigen` via conducting `install.packages(c("Rcpp", "RcppEigen"))` in R console.

After installing dependencies, run the following code in terminal/bash:

```
cd R-package
autoreconf
R CMD INSTALL .
```

If it finishes with `* DONE (abess)`, the installation is successful.

7.1.3 Dependencies

C++

Our core C++ code is based on some dependencies:

- Eigen (version 3.3.4): a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
- Spectra (version 1.0.0): a header-only C++ library for large scale eigenvalue problems.

They would be automatically included while installing the `abess` packages.

OpenMP

To support OpenMP parallelism in Cpp, the dependence for OpenMP should be install. Actually, many compliers and tools have supported and you can check here.

> What is more, if you receive a warning like "*Unknown option '-fopenmp'*" while installing abess, it means that OpenMP has not been enabled. Without OpenMP, abess only use a single CPU core, leading to suboptimal learning speed.

To enable OpenMP:

- In Windows, Visual C++ or many other C++ compliers can support OpenMP API, but you may need to enable it manually in additional features (based on the complier you use).
- In Linux, the dependence would be supported if GCC is installed (version 4.2+).
- In MacOS, the dependence can be installed by:

```
$ brew install llvm
$ brew install libomp
```

Python

Some basic Python packages are required for `abess`. Actually, they can be found on `abess/python/setup.py` and automatically installed during the installation.

- pybind11: seamless operability between C++11 and Python
- numpy: the fundamental package for array computing with Python.
- scipy: work with NumPy arrays, and provides many user-friendly and efficient numerical routines.
- scikit-learn: a Python module for machine learning built on top of SciPy.
- pandas: support data manipulation and input.

Furthermore, if you want to develop the Python packages, some additional packages should be installed:

- pytest: simple powerful testing with Python.
  - lifelines: support testing for survival analysis.
- Sphinx: develop the Python documentation.
  - sphinx-rtd-theme: "Read the Docs" theme for Sphinx.

  - sphinix-gallery: develop the gallery of Python examples.

R

The R version should be 3.1.0 and newer in order to support C++11. `abess` R package relies on limited R packages dependencies:

- Rcpp: convert R Matrix/Vector object into C++.
- RcppEigen: linear algebra in C++.

Furthermore, if you would to develop the R package, it would be better to additionally install:

- testthat: conduct unit tests.
- roxygen2: write R documentations.
- knitr and rmarkdown: write tutorials for R package.
- pkgdown: build website for the `abess` R package.

7.2 Tutorial

The Tutorial section aims to provide working code samples demonstrating how to use the `abess` library to solve real world issues. In the following pages, the `abess` Python package is used for illustration. The counterpart for R package is available at here.

7.2.1 Generalized Linear Model

7.2.2 Principal Component Analysis

7.2.3 Advanced Generic Features

When analyzing the real world datasets, we may have the following targets:

1. identifying predictors when group structure are provided (a.k.a., **best group subset selection**);
2. certain variables must be selected when some prior information is given (a.k.a., **nuisance regression**);
3. selecting the weak signal variables when the prediction performance is mainly interested (a.k.a., **regularized best-subset selection**).

These targets are frequently encountered in real world data analysis. Actually, in our methods, the targets can be properly handled by simply change some default arguments in the functions. In the following content, we will illustrate the statistic methods to reach these targets in a one-by-one manner, and give quick examples to show how to perform the statistic methods in `LinearRegression` and the same steps can be implemented in all methods.

Besides, `abess` library is very flexible, i.e., users can flexibly control many internal computational components. Specifically, users can specify: (i) the division of samples in cross validation (a.k.a., **cross validation division**), (ii) specify the initial active set before splicing (a.k.a., **initial active set**), and so on. We will also describe these in the following.

7.2.4 Computational Tips

The generic splicing technique certifiably guarantees the best subset can be selected in a polynomial time. In practice, the computational efficiency can be improved to handle large scale datasets. The tips for computational improvement are applicable for:

1. **ultra-high dimensional data** via
    - feature screening;
    - focus on important variables;
2. **large-sample data** via
    - golden-section searching;
    - early-stop scheme;
3. **sparse inputs** via
    - sparse matrix computation;
4. **specific models** via
    - covariance update for `LinearRegression` and `MultiTaskRegression`;
    - quasi Newton iteration for `LogisticRegression`, `PoissonRegression`, `CoxRegression`, etc.

More importantly, the technique in these tips can be use simultaneously. For example, `abess` allow algorithms to use both feature screening and golden-section searching such that algorithms can handle datasets with large-sample and ultra-high dimension. The following contents illustrate the above tips.

Besides, `abess` efficiently implements warm-start initialization and parallel computing, which are very useful for fast computing. To help use leverage them, we will also describe their implementation details in the following.

7.2.5 Connect to Popular Libraries with `scikit-learn` API

This part is intended to present all possible connection between `abess` and other popular Python libraries via the `scikit-learn` interface. It is keep developing and more examples will come up soon. Contributions for this part is extremely welcome!

Generalized Linear Model

Linear Regression

In this tutorial, we are going to demonstrate how to use the `abess` package to carry out best subset selection in linear regression with both simulated data and real data.

Our package `abess` implements a polynomial algorithm in the following best-subset selection problem:

$$\min_{\beta\in\mathbb{R}^p} \frac{1}{2n}||y - X\beta||_2^2, \quad \text{s.t. } ||\beta||_0 \leq s,$$

where $\|\cdot\|_2$ is the ℓ_2 norm, $\|\beta\|_0 = \sum_{i=1}^p I(\beta_i \neq 0)$ is the ℓ_0 norm of β, and the sparsity level s is an unknown non-negative integer to be determined. Next, we present an example to show the `abess` package can get an optimal estimation.

Toward optimality: adaptive best-subset selection

Synthetic dataset

We generate a design matrix X containing $n = 300$ observations and each observation has $p = 1000$ predictors. The response variable y is linearly related to the first, second, and fifth predictors in X:

$$y = 3X_1 + 1.5X_2 + 2X_5 + \epsilon,$$

where ϵ is a standard normal random variable.

```
import numpy as np
from abess.datasets import make_glm_data
np.random.seed(0)

n = 300
p = 1000
true_support_set=[0, 1, 4]
true_coef = np.array([3, 1.5, 2])
real_coef = np.zeros(p)
real_coef[true_support_set] = true_coef
data1 = make_glm_data(n=n, p=p, k=len(true_coef), family="gaussian", coef_=real_coef)

print(data1.x.shape)
print(data1.y.shape)
```

```
(300, 1000)
(300,)
```

This dataset is high-dimensional and brings large challenge for subset selection. As a typical data examples, it mimics data appeared in real-world for modern scientific researches and data mining, and serves a good quick example for demonstrating the power of the `abess` library.

Optimality

The optimality of subset selection means:

- `true_support_set` (i.e. `[0, 1, 4]`) can be exactly identified;
- the estimated coefficients is ordinary least squares (OLS) estimator under the true subset such that is very closed to `true_coef = np.array([3, 1.5, 2])`.

To understand the second criterion, we take a look on the estimation given by `scikit-learn` library:

```
from sklearn.linear_model import LinearRegression as SKLLinearRegression
sklearn_lr = SKLLinearRegression()
sklearn_lr.fit(data1.x[:, [0, 1, 4]], data1.y)
print("OLS estimator: ", sklearn_lr.coef_)
```

```
OLS estimator:  [3.00085183 1.44388323 2.00633204]
```

The fitted coefficients `sklearn_lr.coef_` is OLS estimator when the true support set is known. It is very closed to the `true_coef`, and is hard to be improve under finite sample size.

Adaptive Best Subset Selection

The adaptive best subset selection (ABESS) algorithm is a very powerful for the selection of the best subset. We will illustrate its power by showing it can reach to the optimality.

The following code shows the simple syntax for using ABESS algorithm via `abess` library.

```
from abess import LinearRegression
model = LinearRegression()
model.fit(data1.x, data1.y)
```

`LinearRegression` functions in `abess` is designed for selecting the best subset under the linear model, which can be imported by: `from abess import LinearRegression`. Following similar syntax like `scikit-learn`, we can fit the data via ABESS algorithm.

Next, we going to see that the above approach can successfully recover the true set `np.array([0, 1, 4])`. The fitted coefficients are stored in `model.coef_`. We use `np.nonzero` function to find the selected subset of `abess`, and we can extract the non-zero entries in `model.coef_` which is the coefficients estimation for the selected predictors.

```
ind = np.nonzero(model.coef_)
print("estimated non-zero: ", ind)
print("estimated coef: ", model.coef_[ind])
```

```
estimated non-zero:  (array([0, 1, 4]),)
estimated coef:  [3.00085183 1.44388323 2.00633204]
```

From the result, we know that `abess` exactly found the true set `np.array([0, 1, 4])` among all 1000 predictors. Besides, the estimated coefficients of them are quite close to the real ones, and is exactly the same as the estimation `sklearn_lr.coef_` given by `scikit-learn`.

Real data example

Hitters Dataset

Now we focus on real data on the Hitters dataset. We hope to use several predictors related to the performance of the baseball athletes last year to predict their salary.

First, let's have a look at this dataset. There are 19 variables except *Salary* and 322 observations.

```
import os
import pandas as pd

data2 = pd.read_csv(os.path.join(os.getcwd(), 'Hitters.csv'))
print(data2.shape)
print(data2.head(5))
```

```
(322, 20)
   AtBat  Hits  HmRun  Runs  RBI  ...  PutOuts  Assists  Errors  Salary  NewLeague
0    293    66      1    30   29  ...      446       33      20     NaN          A
1    315    81      7    24   38  ...      632       43      10   475.0          N
2    479   130     18    66   72  ...      880       82      14   480.0          A
3    496   141     20    65   78  ...      200       11       3   500.0          N
4    321    87     10    39   42  ...      805       40       4    91.5          N
```

(continues on next page)

(continued from previous page)

```
[5 rows x 20 columns]
```

Since the dataset contains some missing values, we simply drop those rows with missing values. Then we have 263 observations remain:

```
data2 = data2.dropna()
print(data2.shape)
```

```
(263, 20)
```

What is more, before fitting, we need to transfer the character variables to dummy variables:

```
data2 = pd.get_dummies(data2)
data2 = data2.drop(['League_A', 'Division_E', 'NewLeague_A'], axis=1)
print(data2.shape)
print(data2.head(5))
```

```
(263, 20)
   AtBat  Hits  HmRun  Runs  ...  Salary  League_N  Division_W  NewLeague_N
1    315    81      7    24  ...   475.0         1           1            1
2    479   130     18    66  ...   480.0         0           1            0
3    496   141     20    65  ...   500.0         1           0            1
4    321    87     10    39  ...    91.5         1           0            1
5    594   169      4    74  ...   750.0         0           1            0

[5 rows x 20 columns]
```

Model Fitting

As what we do in simulated data, an adaptive best subset can be formed easily:

```
x = np.array(data2.drop('Salary', axis=1))
y = np.array(data2['Salary'])

model = LinearRegression(support_size=range(20))
model.fit(x, y)
```

The result can be shown as follows:

```
ind = np.nonzero(model.coef_)
print("non-zero:\n", data2.columns[ind])
print("coef:\n", model.coef_)
```

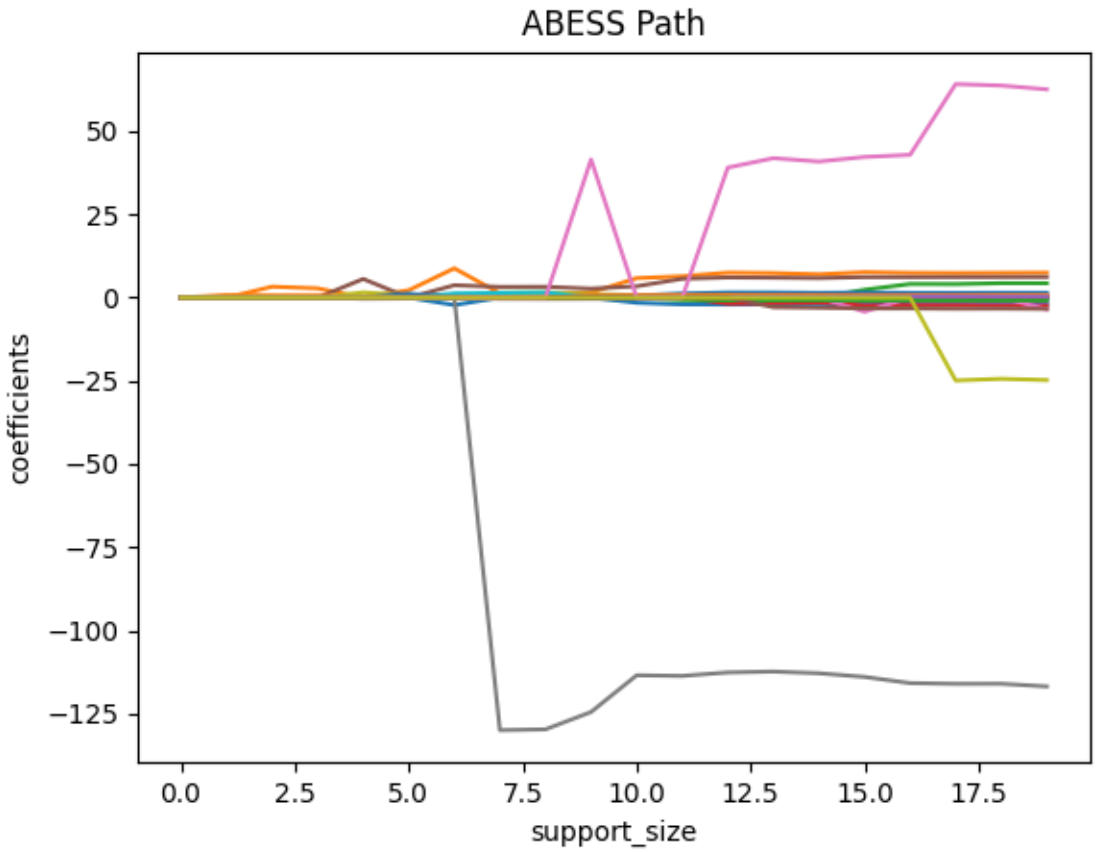
```
non-zero:
 Index(['Hits', 'CRBI', 'PutOuts', 'League_N'], dtype='object')
coef:
 [   0.           2.67579779    0.           0.           0.
    0.           0.           0.           0.           0.
    0.           0.681779     0.           0.27350022    0.
    0.           0.         -139.9538855    0.        ]
```

Automatically, variables *Hits*, *CRBI*, *PutOuts*, *League_N* are chosen in the model (the chosen sparsity level is 4).

More on the results

We can also plot the path of abess process:

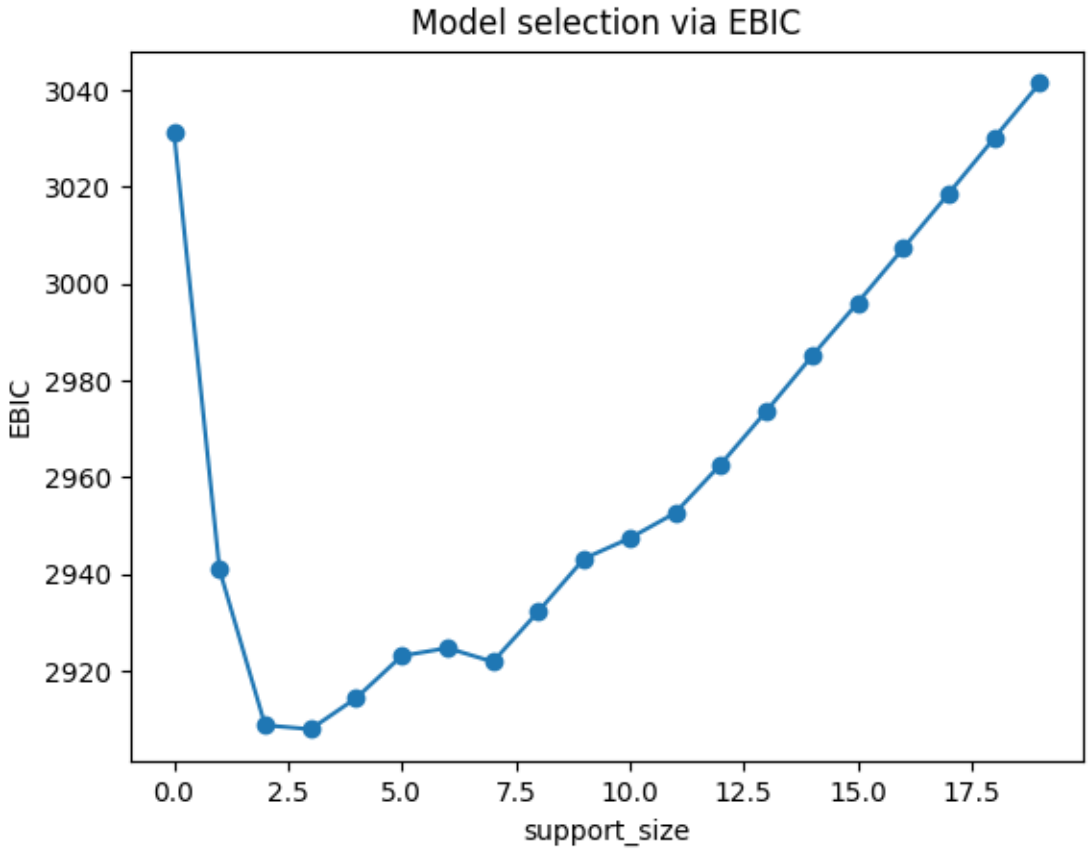
```
import matplotlib.pyplot as plt
coef = np.zeros((20, 19))
ic = np.zeros(20)
for s in range(20):
    model = LinearRegression(support_size=s)
    model.fit(x, y)
    coef[s, :] = model.coef_
    ic[s] = model.eval_loss_

for i in range(19):
    plt.plot(coef[:, i], label=i)

plt.xlabel('support_size')
plt.ylabel('coefficients')
plt.title('ABESS Path')
plt.show()
```



Besides, we can also generate a graph about the tuning parameter. Remember that we used the default EBIC to tune

the support size.

```
plt.plot(ic, 'o-')
plt.xlabel('support_size')
plt.ylabel('EBIC')
plt.title('Model selection via EBIC')
plt.show()
```



In EBIC criterion, a subset with the support size 3 has the lowest value, so the process adaptively chooses 3 variables. Note that under other information criteria, the result may be different.

R tutorial

For R tutorial, please view https://abess-team.github.io/abess/articles/v01-abess-guide.html.

Total running time of the script: (0 minutes 1.643 seconds)

Classification: Logistic Regression and Beyond

We would like to use an example to show how the best subset selection for logistic regression works in our program.

Real Data Example

Titanic Dataset

Consider the Titanic dataset obtained from the Kaggle competition: https://www.kaggle.com/c/titanic/data. The dataset consists of data of 889 passengers, and the goal of the competition is to predict the survival status (yes/no) based on features including the class of service, the sex, the age, etc.

```
import pandas as pd

dt = pd.read_csv("train.csv")
print(dt.head(5))
```

```
   PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0            1         0       3  ...   7.2500   NaN         S
1            2         1       1  ...  71.2833   C85         C
2            3         1       3  ...   7.9250   NaN         S
3            4         1       1  ...  53.1000  C123         S
4            5         0       3  ...   8.0500   NaN         S

[5 rows x 12 columns]
```

We only focus on some numerical or categorical variables:

- predictor variables: *Pclass*, *Sex*, *Age*, *SibSp*, *Parch*, *Fare*, *Embarked*;
- response variable is *Survived*.

```
dt = dt.iloc[:, [1, 2, 4, 5, 6, 7, 9, 11]]  # variables interested
dt['Pclass'] = dt['Pclass'].astype(str)
print(dt.head(5))
```

```
   Survived Pclass     Sex   Age  SibSp  Parch     Fare Embarked
0         0      3    male  22.0      1      0   7.2500        S
1         1      1  female  38.0      1      0  71.2833        C
2         1      3  female  26.0      0      0   7.9250        S
3         1      1  female  35.0      1      0  53.1000        S
4         0      3    male  35.0      0      0   8.0500        S
```

However, some rows contain missing values (NaN) and we need to drop them.

```
dt = dt.dropna()
print('sample size: ', dt.shape)
```

```
sample size:  (712, 8)
```

Then use dummy variables to replace classification variables:

```
dt1 = pd.get_dummies(dt)
print(dt1.head(5))
```

```
   Survived   Age  SibSp  Parch  ...  Sex_male  Embarked_C  Embarked_Q  Embarked_S
0         0  22.0      1      0  ...         1           0           0           1
1         1  38.0      1      0  ...         0           1           0           0
2         1  26.0      0      0  ...         0           0           0           1
3         1  35.0      1      0  ...         0           0           0           1
4         0  35.0      0      0  ...         1           0           0           1

[5 rows x 13 columns]
```

Now we split *dt1* into training set and testing set:

```
from sklearn.model_selection import train_test_split
import numpy as np

X = np.array(dt1.drop('Survived', axis=1))
Y = np.array(dt1.Survived)

train_x, test_x, train_y, test_y = train_test_split(
    X, Y, test_size=0.33, random_state=0)
print('train size: ', train_x.shape[0])
print('test size:', test_x.shape[0])
```

```
train size:  477
test size: 235
```

Here `train_x` contains:

- V0: dummy variable, 1st ticket class (1-yes, 0-no)
- V1: dummy variable, 2nd ticket class (1-yes, 0-no)
- V2: dummy variable, sex (1-male, 0-female)
- V3: Age
- V4: # of siblings / spouses aboard the Titanic
- V5: # of parents / children aboard the Titanic
- V6: Passenger fare
- V7: dummy variable, Cherbourg for embarkation (1-yes, 0-no)
- V8: dummy variable, Queenstown for embarkation (1-yes, 0-no)

And `train_y` indicates whether the passenger survived (1-yes, 0-no).

```
print('train_x:\n', train_x[0:5, :])
print('train_y:\n', train_y[0:5])
```

```
train_x:
 [[54.      1.      0.     59.4     1.      0.      0.      1.      0.
   1.      0.      0.    ]
 [30.      0.      0.      8.6625  0.      0.      1.      1.      0.
   0.      0.      1.    ]
 [47.      0.      0.     38.5     1.      0.      0.      0.      1.
   0.      0.      1.    ]
 [28.      2.      0.      7.925   0.      0.      1.      0.      1.
   0.      0.      1.    ]
 [29.      1.      0.     26.      0.      1.      0.      1.      0.
   0.      0.      1.    ]]
train_y:
 [1 0 0 0 1]
```

Model Fitting

The `LogisticRegression()` function in the `abess.linear` allows us to perform best subset selection in a highly efficient way. For example, in the Titanic sample, if you want to look for a best subset with no more than 5 variables on the logistic model, you can call:

```
from abess import LogisticRegression

s = 5   # max target sparsity
model = LogisticRegression(support_size=range(0, s + 1))
model.fit(train_x, train_y)
```

Now the `model.coef_` contains the coefficients of logistic model with no more than 5 variables. That is, those variables with a coefficient 0 is unused in the model:

```
print(model.coef_)
```

```
[-0.05410776 -0.53642966  0.          0.          1.74091231  0.
 -1.26223831  2.7096497   0.          0.          0.          0.        ]
```

By default, the `LogisticRegression` function set `support_size = range(0, min(p,n/log(n)p)` and the best support size is determined by the Extended Bayesian Information Criteria (EBIC). You can change the tuning criterion by specifying the argument `ic_type`. The available tuning criteria now are `"gic"`, `"aic"`, `"bic"`, `"ebic"`.

For a quicker solution, you can change the tuning strategy to a golden section path which tries to find the elbow point of the tuning criterion over the hyperparameter space. Here we give an example.

```
model_gs = LogisticRegression(path_type="gs", s_min=0, s_max=s)
model_gs.fit(train_x, train_y)
print(model_gs.coef_)
```

```
[-0.05410776 -0.53642966  0.          0.          1.74091231  0.
 -1.26223831  2.7096497   0.          0.          0.          0.        ]
```

where `s_min` and `s_max` bound the support size and this model gives the same answer as before.

More on the Results

After fitting with `model.fit()`, we can further do more exploring work to interpret it. As we show above, `model.coef_` contains the sparse coefficients of variables and those non-zero values indicate "important" variables chosen in the model.

```
print('Intercept: ', model.intercept_)
print('coefficients: \n', model.coef_)
print('Used variables\' index:', np.nonzero(model.coef_ != 0)[0])
```

```
Intercept:  0.5742977507847972
coefficients:
 [-0.05410776 -0.53642966  0.          0.          1.74091231  0.
 -1.26223831  2.7096497   0.          0.          0.          0.        ]
Used variables' index: [0 1 4 6 7]
```

The training loss and the score under information criterion:

```
print('Training Loss: ', model.train_loss_)
print('IC: ', model.eval_loss_)
```

```
Training Loss:  204.35270048059678
IC:  464.3920499135153
```

Prediction is allowed for the estimated model. Just call `model.predict()` function like:

```
fitted_y = model.predict(test_x)
print(fitted_y)
```

```
[0 0 1 0 0 0 1 0 0 1 1 1 1 0 0 1 0 1 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0 1 1 1 1
 1 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 1 1 1
 0 1 0 0 0 0 1 1 0 1 1 0 0 0 1 0 0 0 1 1 1 0 1 1 0 0 0 1 0 0 0 0 1 0 0 0 0
 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0
 0 0 0 0 1 1 1 0 0 1 1 1 1 0 1 1 0 0 0 0 0 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0
 1 0 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 0 1 0 0 0 0 0 0 1 1 1 1 0 0 1 1 0 1 1 0
 0 1 0 0 0 0 0 0 0 0 0 0 0]
```

Besides, we can also call for the survival probability of each observation by `model.predict_proba()`, which returns a matrix with two columns. The first column corresponds to probability for class "0", and the second is for class "1" (survived). Then, we tend to classify the observation into more likely class.

```
fitted_p = model.predict_proba(test_x)
print(fitted_p)
```

```
[[0.50743387 0.49256613]
 [0.74057032 0.25942968]
 [0.15071537 0.84928463]
 [0.79795817 0.20204183]
 [0.96198452 0.03801548]
 [0.95977651 0.04022349]
 [0.27648557 0.72351443]
 [0.76884378 0.23115622]
```

(continues on next page)

(continued from previous page)

```
[0.76884378 0.23115622]
[0.33165327 0.66834673]
[0.03224465 0.96775535]
[0.35094054 0.64905946]
[0.01538079 0.98461921]
[0.84761133 0.15238867]
[0.74995921 0.25004079]
[0.42359788 0.57640212]
[0.73004032 0.26995968]
[0.28735418 0.71264582]
[0.62208165 0.37791835]
[0.8228686  0.1771314 ]
[0.74226703 0.25773297]
[0.24607858 0.75392142]
[0.12025589 0.87974411]
[0.59748431 0.40251569]
[0.43558118 0.56441882]
[0.65942131 0.34057869]
[0.77994844 0.22005156]
[0.932841   0.067159  ]
[0.42119469 0.57880531]
[0.66352233 0.33647767]
[0.84344878 0.15655122]
[0.97317339 0.02682661]
[0.85446957 0.14553043]
[0.30336212 0.69663788]
[0.10921555 0.89078445]
[0.12074848 0.87925152]
[0.08073996 0.91926004]
[0.40918613 0.59081387]
[0.57002721 0.42997279]
[0.54346526 0.45653474]
[0.61153036 0.38846964]
[0.90979818 0.09020182]
[0.94257539 0.05742461]
[0.92226281 0.07773719]
[0.9005148  0.0994852 ]
[0.88993666 0.11006334]
[0.0180426  0.9819574 ]
[0.85780137 0.14219863]
[0.8903911  0.1096089 ]
[0.03059829 0.96940171]
[0.28648812 0.71351188]
[0.30336212 0.69663788]
[0.36336243 0.63663757]
[0.74057032 0.25942968]
[0.45021417 0.54978583]
[0.46690207 0.53309793]
[0.92967528 0.07032472]
[0.9293708  0.0706292 ]
[0.13110112 0.86889888]
[0.62098833 0.37901167]
```

(continues on next page)

(continued from previous page)

```
[0.56123326 0.43876674]
[0.96915459 0.03084541]
[0.85446957 0.14553043]
[0.80006385 0.19993615]
[0.70819044 0.29180956]
[0.88171401 0.11828599]
[0.05413855 0.94586145]
[0.69389487 0.30610513]
[0.01236779 0.98763221]
[0.19088286 0.80911714]
[0.74057032 0.25942968]
[0.06948297 0.93051703]
[0.0902975  0.9097025 ]
[0.48714638 0.51285362]
[0.95075583 0.04924417]
[0.46234646 0.53765354]
[0.51757961 0.48242039]
[0.73959052 0.26040948]
[0.90525825 0.09474175]
[0.6615436  0.3384564 ]
[0.44892685 0.55107315]
[0.11974729 0.88025271]
[0.90941602 0.09058398]
[0.18266554 0.81733446]
[0.13163148 0.86836852]
[0.90525825 0.09474175]
[0.95538456 0.04461544]
[0.71924495 0.28075505]
[0.21109988 0.78890012]
[0.86106974 0.13893026]
[0.97565829 0.02434171]
[0.95302055 0.04697945]
[0.29853147 0.70146853]
[0.08595031 0.91404969]
[0.33767709 0.66232291]
[0.9005148  0.0994852 ]
[0.06280397 0.93719603]
[0.1577817  0.8422183 ]
[0.8903911  0.1096089 ]
[0.84530315 0.15469685]
[0.84761133 0.15238867]
[0.14120978 0.85879022]
[0.77994844 0.22005156]
[0.75908805 0.24091195]
[0.78831956 0.21168044]
[0.84761133 0.15238867]
[0.39506122 0.60493878]
[0.67355065 0.32644935]
[0.73874787 0.26125213]
[0.92482907 0.07517093]
[0.86106974 0.13893026]
[0.25965364 0.74034636]
```

(continues on next page)

(continued from previous page)

```
[0.15253925 0.84746075]
[0.54786818 0.45213182]
[0.9293708  0.0706292 ]
[0.74057032 0.25942968]
[0.77994844 0.22005156]
[0.98164302 0.01835698]
[0.85836737 0.14163263]
[0.79788631 0.20211369]
[0.84761133 0.15238867]
[0.90009763 0.09990237]
[0.76081454 0.23918546]
[0.26927389 0.73072611]
[0.73784984 0.26215016]
[0.96391455 0.03608545]
[0.96129876 0.03870124]
[0.83746312 0.16253688]
[0.25965364 0.74034636]
[0.02006328 0.97993672]
[0.91829389 0.08170611]
[0.35926408 0.64073592]
[0.15966607 0.84033393]
[0.14789964 0.85210036]
[0.19016604 0.80983396]
[0.02742217 0.97257783]
[0.36336243 0.63663757]
[0.98180978 0.01819022]
[0.95478642 0.04521358]
[0.88499785 0.11500215]
[0.64716682 0.35283318]
[0.9395756  0.0604244 ]
[0.19016604 0.80983396]
[0.34572827 0.65427173]
[0.43558118 0.56441882]
[0.78909413 0.21090587]
[0.90979818 0.09020182]
[0.84761133 0.15238867]
[0.90794231 0.09205769]
[0.86741702 0.13258298]
[0.92967528 0.07032472]
[0.89556126 0.10443874]
[0.32670564 0.67329436]
[0.08952309 0.91047691]
[0.12858887 0.87141113]
[0.86741702 0.13258298]
[0.86106974 0.13893026]
[0.30998425 0.69001575]
[0.0145825  0.9854175 ]
[0.25965364 0.74034636]
[0.04842691 0.95157309]
[0.90009763 0.09990237]
[0.02115516 0.97884484]
[0.48933053 0.51066947]
```

(continues on next page)

(continued from previous page)

```
[0.95558225 0.04441775]
[0.95558225 0.04441775]
[0.71638648 0.28361352]
[0.96512977 0.03487023]
[0.50511029 0.49488971]
[0.8821979  0.1178021 ]
[0.35926408 0.64073592]
[0.37487948 0.62512052]
[0.02115516 0.97884484]
[0.9293708  0.0706292 ]
[0.49506961 0.50493039]
[0.37596932 0.62403068]
[0.13163148 0.86836852]
[0.86106974 0.13893026]
[0.82544239 0.17455761]
[0.6968841  0.3031159 ]
[0.92226281 0.07773719]
[0.62098833 0.37901167]
[0.88221559 0.11778441]
[0.5298741  0.4701259 ]
[0.59737712 0.40262288]
[0.0630781  0.9369219 ]
[0.82544239 0.17455761]
[0.83310188 0.16689812]
[0.33359333 0.66640667]
[0.12661189 0.87338811]
[0.75738401 0.24261599]
[0.41474865 0.58525135]
[0.23939759 0.76060241]
[0.90941602 0.09058398]
[0.041657   0.958343  ]
[0.27018941 0.72981059]
[0.69488121 0.30511879]
[0.70819044 0.29180956]
[0.22574405 0.77425595]
[0.03224465 0.96775535]
[0.9141412  0.0858588 ]
[0.13163148 0.86836852]
[0.96915459 0.03084541]
[0.28099043 0.71900957]
[0.91273698 0.08726302]
[0.94704734 0.05295266]
[0.65133737 0.34866263]
[0.67146626 0.32853374]
[0.965596   0.034404  ]
[0.84049023 0.15950977]
[0.08914497 0.91085503]
[0.47466173 0.52533827]
[0.19863876 0.80136124]
[0.44777727 0.55222273]
[0.92605446 0.07394554]
[0.75082977 0.24917023]
```

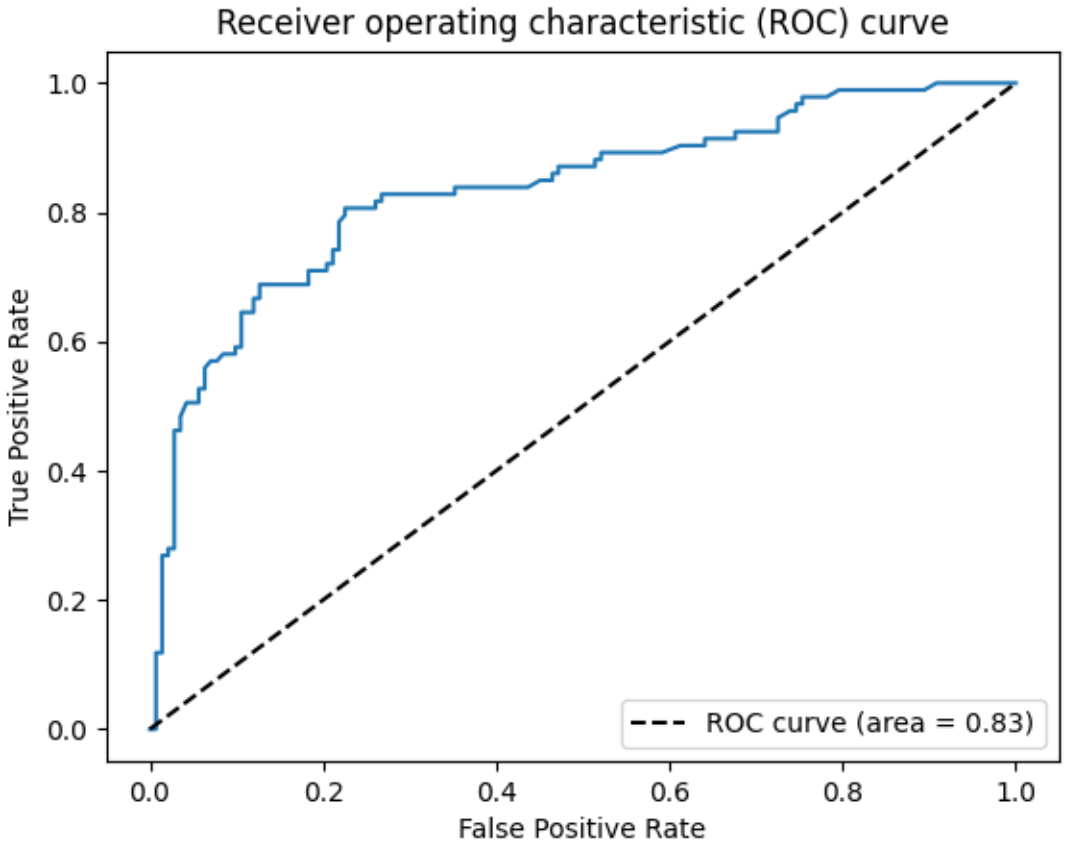
(continues on next page)

(continued from previous page)

```
 [0.23524154 0.76475846]
 [0.26568554 0.73431446]
 [0.72817106 0.27182894]
 [0.1023766  0.8976234 ]
 [0.32670564 0.67329436]
 [0.95558225 0.04441775]
 [0.69875031 0.30124969]
 [0.02351608 0.97648392]
 [0.83746312 0.16253688]
 [0.85107278 0.14892722]
 [0.97930718 0.02069282]
 [0.71732988 0.28267012]
 [0.94257539 0.05742461]
 [0.94987806 0.05012194]
 [0.87351692 0.12648308]
 [0.93254923 0.06745077]
 [0.91724157 0.08275843]
 [0.90979818 0.09020182]
 [0.932841   0.067159  ]]
```

We can also generate an ROC curve and calculate tha AUC value. On this dataset, the AUC is 0.83, which is rather desirable.

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

fpr, tpr, _ = roc_curve(test_y, fitted_p[:, 1])
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], 'k--', label="ROC curve (area = %0.2f)" % auc(fpr, tpr))
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("Receiver operating characteristic (ROC) curve")
plt.legend(loc="lower right")
plt.show()
```



Extension: Multi-class Classification

Multinomial logistic regression

When the number of classes is more than 2, we call it multi-class classification task. Logistic regression can be extended to model several classes of events such as determining whether an image contains a cat, dog, lion, etc. Each object being detected in the image would be assigned a probability between 0 and 1, with a sum of one. The extended model is multinomial logistic regression.

To arrive at the multinomial logistic model, one can imagine, for K possible classes, running $K1$ independent logistic regression models, in which one class is chosen as a "pivot" and then the other $K1$ classes are separately regressed against the pivot outcome. This would proceed as follows, if class K (the last outcome) is chosen as the pivot:

$$\ln(\mathbb{P}(y=1)/\mathbb{P}(y=K)) = x^T\beta^{(1)},$$
$$\cdots$$
$$\ln(\mathbb{P}(y=K-1)/\mathbb{P}(y=K)) = x^T\beta^{(K-1)}.$$

Then, the probability to choose the j-th class can be easily derived to be:

$$\mathbb{P}(y=j) = \frac{\exp(x^T\beta^{(j)})}{1+\sum_{k=1}^{K-1}\exp(x^T\beta^{(k)})},$$

and subsequently, we would that the object belongs to the j^*-th class if the $j^* = \arg\max_j \mathbb{P}(y=j)$. Notice that, for K possible classes case, there are $p \times (K1)$ unknown parameters: $\beta^{(1)}, \ldots, \beta^{(K1)}$ to be estimated. Because the number

of parameters increases as K, it is even more urgent to constrain the model complexity. And the best subset selection for multinomial logistic regression aims to maximize the log-likelihood function and control the model complexity by restricting $B = (\beta^{(1)}, \ldots, \beta^{(K1)})$ with $||B||_{0,2} \leq s$ where $||B||_{0,2} = \sum_{i=1}^{p} I(B_{i\cdot} = 0)$, $B_{i\cdot}$ is the i-th row of coefficient matrix B and $0 \in \mathbb{R}^{K-1}$ is an all zero vector. In other words, each row of B would be either all zero or all non-zero.

Simulated Data Example

We shall conduct Multinomial logistic regression on an artificial dataset for demonstration. The `make_multivariate_glm_data()` provides a simple way to generate suitable dataset for this task.

The assumption behind this model is that the response vector follows a multinomial distribution. The artificial dataset contains 100 observations and 20 predictors but only five predictors have influence on the three possible classes.

```
from abess.datasets import make_multivariate_glm_data

n = 100  # sample size
p = 20  # all predictors
k = 5   # real predictors
M = 3   # number of classes

np.random.seed(0)
dt = make_multivariate_glm_data(n=n, p=p, k=k, family="multinomial", M=M)
print(dt.coef_)
print('real variables\' index:\n', set(np.nonzero(dt.coef_)[0]))
```

```
[[ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 1.09734231  4.03598978  0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 9.91227834 -3.47987303  0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 8.93282229  8.93249765  0.        ]
 [-4.03426165 -2.70336848  0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [-5.53475149 -2.65928982  0.        ]
 [ 0.          0.          0.        ]]
real variables' index:
 {2, 5, 10, 11, 18}
```

To carry out best subset selection for multinomial logistic regression, we can call the `MultinomialRegression()`. Here is an example.

```
from abess import MultinomialRegression

s = 5
model = MultinomialRegression(support_size=range(0, s + 1))
model.fit(dt.x, dt.y)
```

Its use is quite similar to `LogisticRegression`. We can get the coefficients to recognize "in-model" variables.

```
print('intercept:\n', model.intercept_)
print('coefficients:\n', model.coef_)
```

```
intercept:
 [0.8313084  0.17875961 0.        ]
coefficients:
 [[ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.2630968   1.97016805  0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 6.91608854 -2.36539928  0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 7.86005073  7.21979661  0.        ]
 [-3.1454875  -1.63434531  0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [-4.80791468 -1.67622324  0.        ]
 [ 0.          0.          0.        ]]
```

So those variables used in model can be recognized and we can find that they are the same as the data's "real" coefficients we generate.

```
print('used variables\' index:\n', set(np.nonzero(model.coef_)[0]))
```

```
used variables' index:
 {2, 5, 10, 11, 18}
```

The `abess` R package also supports classification tasks. For R tutorial, please view https://abess-team.github.io/abess/articles/v03-classification.html.

Total running time of the script: (0 minutes 0.224 seconds)

Multi-Response Linear Regression

Introduction: model setting

Multi-response linear regression (a.k.a., multi-task learning) aims at predicting multiple responses at the same time, and thus, it is a natural extension for classical linear regression where the response is univariate. Multi-response linear regression (MRLR) is very helpful for the analysis of correlated response such as chemical measurements for soil samples and microRNAs associated with Glioblastoma multiforme cancer. Suppose y is an m-dimensional response variable, x is p-dimensional predictors, $B \in R^{m\times p}$ is the coefficient matrix, the MMLR model for the multivariate response is given by

$$y = Bx + \epsilon,$$

where ϵ is an m-dimensional random noise variable with zero mean.

Due to the Occam's razor principle or the high-dimensionality of predictors, it is meaningful to use a small amount of predictors to conduct multi-task learning. For example, understanding the relationship between gene expression and symptoms of a disease has significant importance in identifying potential markers. Many diseases usually involve multiple manifestations and those manifestations are usually related. In some cases, it makes sense to predict those manifestations using a small but the same set of predictors. The best subset selection problem under the MMLR model is formulated as

$$\frac{1}{2n}\|Y - XB\|_2^2, \text{ subject to: } \|B\|_{0,2} \leq s,$$

where, $Y \in R^{n\times m}$ and $X \in R^{n\times p}$ record n observations` response and predictors, respectively. Here $\|B\|_{0,2} = \sum_{i=1}^{p} I(B_{i\cdot} = \mathbf{0})$, where $B_{i\cdot}$ is the i-th row of coefficient matrix B and $\mathbf{0} \in R^m$ is an all-zero vector.

Simulated Data Example

We use an artificial dataset to demonstrate how to solve best subset selection problem for MMLR with `abess` package. The `make_multivariate_glm_data()` function provides a simple way to generate suitable dataset for this task. The synthetic data have 100 observations with 3-dimensional responses and 20-dimensional predictors. Note that there are three predictors having an impact on the responses.

```
from abess.datasets import make_multivariate_glm_data
import numpy as np
np.random.seed(0)

n = 100
p = 20
M = 3
k = 3

data = make_multivariate_glm_data(n=n, p=p, M=M, k=k, family='multigaussian')
print(data.y[0:5, ])

print(data.coef_)
print("non-zero: ", set(np.nonzero(data.coef_)[0]))
```

```
[[-4.47877355 -2.83660944  9.01903871]
 [ 6.89372511 -2.07996131 -4.30416456]
 [ 0.98334    -1.85903489 -5.4657045 ]
```

(continues on next page)

(continued from previous page)

```
 [ 1.9803706   1.8026655  -6.35946312]
 [ 2.0746701   1.97692084  0.26295969]]
[[ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.8880576   2.35738133  0.33938644]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 2.33460874 -3.0222518  -1.63030259]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [-0.25565796 -0.25578781 -3.82878688]
 [ 0.          0.          0.        ]]
non-zero:  {2, 18, 5}
```

Model Fitting

To carry out sparse mutli-task learning, we can call the `MultiTaskRegression` like:

```
from abess import MultiTaskRegression

model = MultiTaskRegression()
model.fit(data.x, data.y)
```

After fitting, `model.coef_` contains the predicted coefficients:

```
print(model.coef_)
print("non-zero: ", set(np.nonzero(model.coef_)[0]))
```

```
[[ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.82745448  2.38786974  0.32939017]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 2.42521165 -3.12093333 -1.76555086]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
```

(continues on next page)

(continued from previous page)

```
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [-0.2299864  -0.15746641 -3.69082244]
 [ 0.          0.          0.        ]]
non-zero:  {2, 18, 5}
```

The outputs show that the support set is correctly identifying and the parameter estimation approaches to the truth.

More on the results

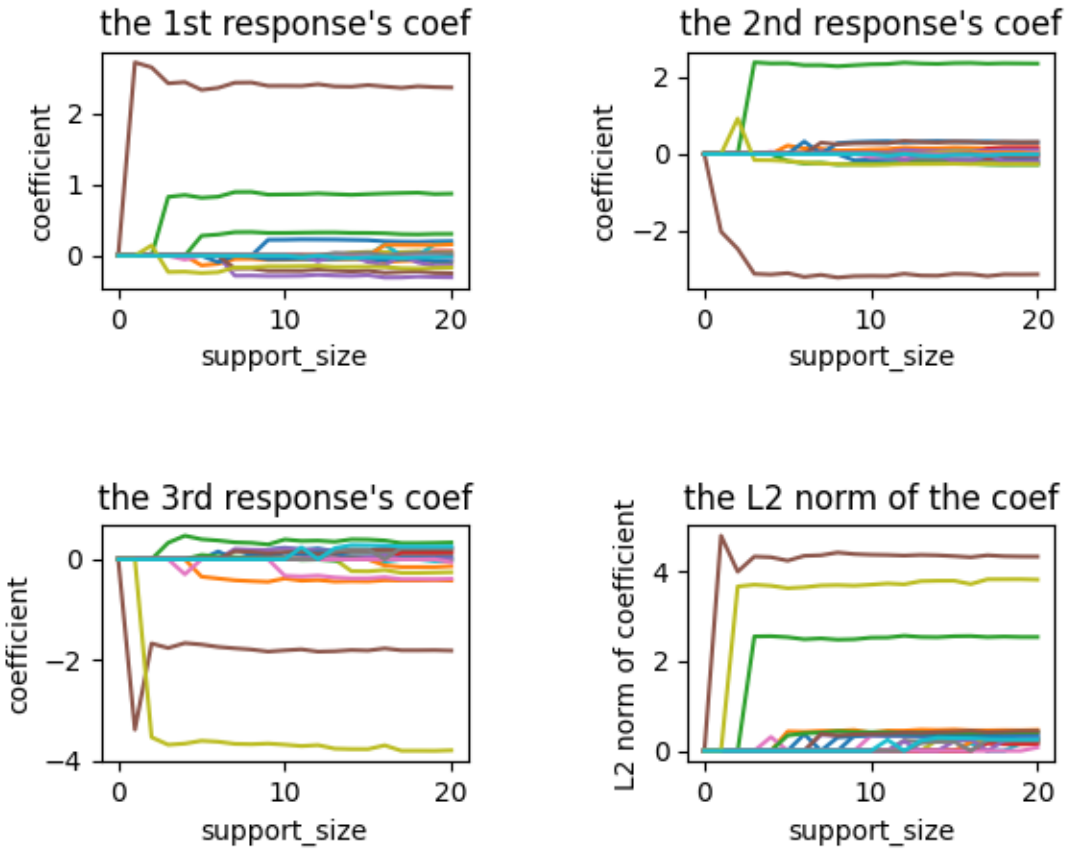
Since there are three responses, we have three solution paths, which correspond to three responses, respectively. To plot the figure, we can fix the `support_size` at different levels:

```
import matplotlib.pyplot as plt

coef = np.zeros((3, 21, 20))
for s in range(21):
    model = MultiTaskRegression(support_size=s)
    model.fit(data.x, data.y)

    for y in range(3):
        coef[y, s, :] = model.coef_[:, y]

plt.subplot(2,2,1)
for i in range(20):
    plt.plot(coef[0, :, i])
plt.xlabel('support_size')
plt.ylabel('coefficient')
plt.title('the 1st response\'s coef')

plt.subplot(2,2,2)
for i in range(20):
    plt.plot(coef[1, :, i])
plt.xlabel('support_size')
plt.ylabel('coefficient')
plt.title('the 2nd response\'s coef')

plt.subplot(2,2,3)
for i in range(20):
    plt.plot(coef[2, :, i])
plt.xlabel('support_size')
plt.ylabel('coefficient')
plt.title('the 3rd response\'s coef')

plt.subplot(2,2,4)
coef_norm =np.sum(coef**2, axis = 0)**0.5
for i in range(20):
    plt.plot(coef_norm[:, i])
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('support_size')
plt.ylabel('L2 norm of coefficient')
plt.title('the L2 norm of the coef')

plt.subplots_adjust(wspace=0.6,hspace=1)
plt.show()
```



The `abess` R package also supports MRLR. For R tutorial, please view https://abess-team.github.io/abess/articles/v06-MultiTaskLearning.html.

Total running time of the script: (0 minutes 0.305 seconds)

Survival Analysis: Cox Regression

Cox Proportional Hazards Regression

Cox Proportional Hazards (CoxPH) regression is to describe the survival according to several corvariates. The difference between CoxPH regression and Kaplan-Meier curves or the logrank tests is that the latter only focus on modeling the survival according to one factor (categorical predictor is best) while the former is able to take into consideration any covariates simultaneously, regardless of whether they're quantitive or categorical. The model is as follows:

$$h(t) = h_0(t) \exp(\eta).$$

where,

- $\eta = x\beta$.
- t is the survival time.
- $h(t)$ is the hazard function which evaluates the risk of dying at time t.
- $h_0(t)$ is called the baseline hazard. It describes value of the hazard if all the predictors are zero.
- $beta$ measures the impact of covariates.

Consider two cases i and i' that have different x values. Their hazard function can be simply written as follow

$$h_i(t) = h_0(t)\exp(\eta_i) = h_0(t)\exp(x_i\beta),$$

and

$$h_{i'}(t) = h_0(t)\exp(\eta_{i'}) = h_0(t)\exp(x_{i'}\beta).$$

The hazard ratio for these two cases is

$$\begin{aligned}\frac{h_i(t)}{h_{i'}(t)} &= \frac{h_0(t)\exp(\eta_i)}{h_0(t)\exp(\eta_{i'})} \\ &= \frac{\exp(\eta_i)}{\exp(\eta_{i'})},\end{aligned}$$

which is independent of time.

Lung Cancer Dataset Analysis

We are going to apply best subset selection to the NCCTG Lung Cancer Dataset from https://www.kaggle.com/ukveteran/ncctg-lung-cancer-data. This dataset consists of survival information of patients with advanced lung cancer from the North Central Cancer Treatment Group. The proportional hazards model allows the analysis of survival data by regression modeling. Linearity is assumed on the log scale of the hazard. The hazard ratio in Cox proportional hazard model is assumed constant.

First, we load the data.

```
import pandas as pd

data = pd.read_csv('cancer.csv')
data = data.drop(data.columns[[0, 1]], axis=1)
print(data.head())
```

```
   time  status  age  sex  ph.ecog  ph.karno  pat.karno  meal.cal  wt.loss
0   306       2   74    1      1.0      90.0      100.0    1175.0      NaN
1   455       2   68    1      0.0      90.0       90.0    1225.0     15.0
2  1010       1   56    1      0.0      90.0       90.0       NaN     15.0
3   210       2   57    1      1.0      90.0       60.0    1150.0     11.0
4   883       2   60    1      0.0     100.0       90.0       NaN      0.0
```

Then we remove the rows containing any missing data. After that, we have a total of 168 observations.

```
data = data.dropna()
print(data.shape)
```

```
(168, 9)
```

Then we change the categorical variable ph.ecog into dummy variables:

```
data['ph.ecog'] = data['ph.ecog'].astype("category")
data = pd.get_dummies(data)
data = data.drop('ph.ecog_0.0', axis=1)
print(data.head())
```

```
   time  status  age  sex  ...  wt.loss  ph.ecog_1.0  ph.ecog_2.0  ph.ecog_3.0
1   455       2   68    1  ...     15.0            0            0            0
3   210       2   57    1  ...     11.0            1            0            0
5  1022       1   74    1  ...      0.0            1            0            0
6   310       2   68    2  ...     10.0            0            1            0
7   361       2   71    2  ...      1.0            0            1            0

[5 rows x 11 columns]
```

We split the dataset into a training set and a test set. The model is going to be built on the training set and later we will test the model performance on the test set.

```
import numpy as np

np.random.seed(0)

ind = np.linspace(1, 168, 168) <= round(168 * 2 / 3)
train = np.array(data[ind])
test = np.array(data[~ind])

print('train size: ', train.shape[0])
print('test size:', test.shape[0])
```

```
train size:  112
test size: 56
```

Model Fitting

The CoxPHSurvivalAnalysis() function in the abess package allows we to perform best subset selection in a highly efficient way.

By default, the function implements the abess algorithm with the support size (sparsity level) changing from 0 to $\min\{p, n/\log(n)p\}$ and the best support size is determined by EBIC. You can change the tuneing criterion by specifying the argument ic_type and the support size by support_size. The available tuning criteria now are "gic", "aic", "bic", "ebic". Here we give an example.

```
from abess import CoxPHSurvivalAnalysis

model = CoxPHSurvivalAnalysis(ic_type='gic')
model.fit(train[:, 2:], train[:, :2])
```

After fitting, the coefficients are stored in model.coef_, and the non-zero values indicate the variables used in our model.

```
print(model.coef_)
```

```
[ 0.         -0.379564    0.02248522  0.          0.          0.
  0.43729712  1.42127851  2.42095755]
```

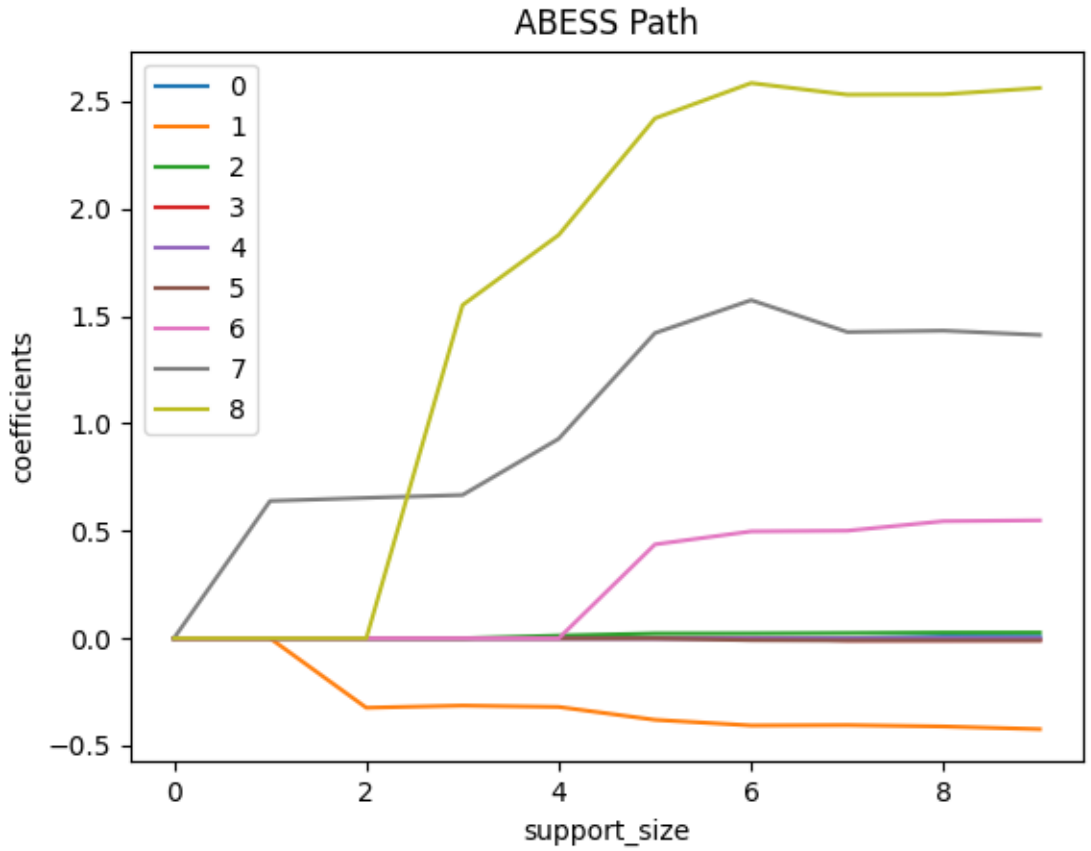
This result shows that 4 variables (the 2nd, 3rd, 7th, 8th, 9th) are chosen into the Cox model. Then a further analysis can be based on them.

More on the results

Hold on, we haven't finished yet. After getting the estimator, we can further do more exploring work. For example, you can use some generic steps to quickly draw some information of those estimators.

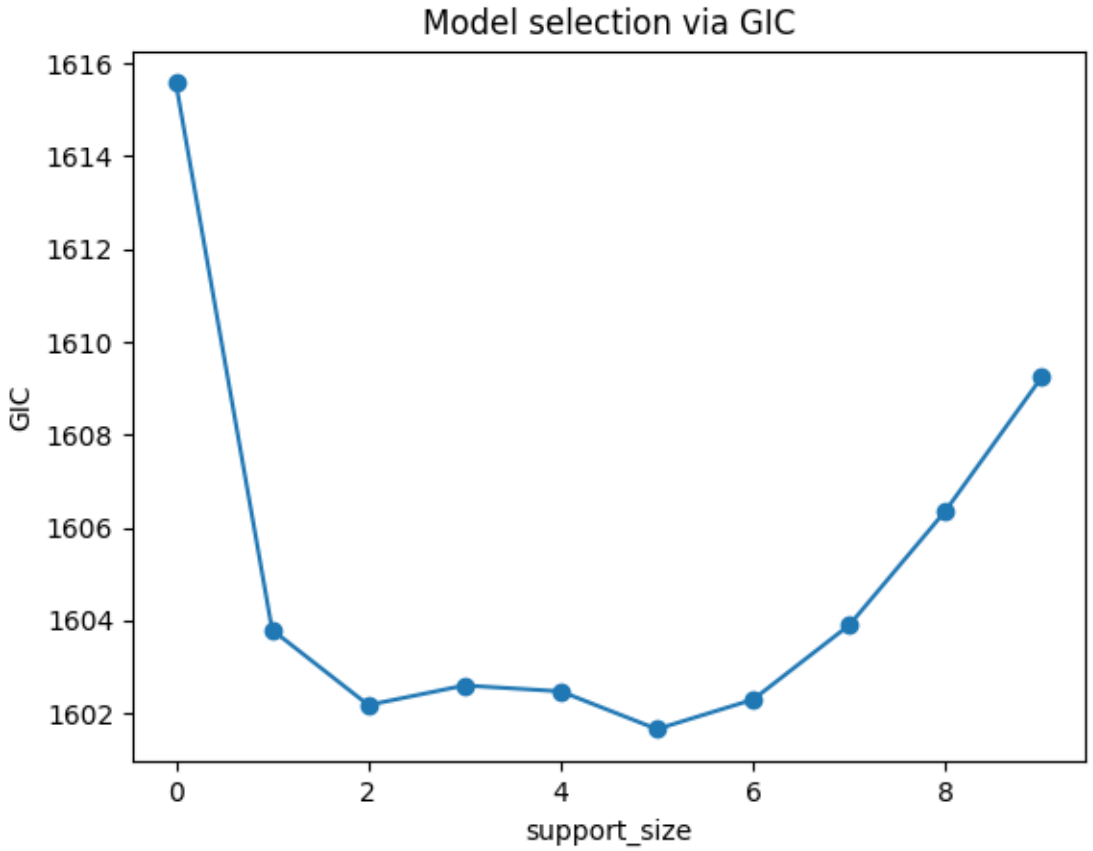
Simply fix the `support_size` in different levels, we can plot a path of coefficients like:

```
import matplotlib.pyplot as plt

coef = np.zeros((10, 9))
ic = np.zeros(10)
for s in range(10):
    model = CoxPHSurvivalAnalysis(support_size=s, ic_type='gic')
    model.fit(train[:, 2:], train[:, :2])
    coef[s, :] = model.coef_
    ic[s] = model.eval_loss_

for i in range(9):
    plt.plot(coef[:, i], label=i)

plt.xlabel('support_size')
plt.ylabel('coefficients')
plt.title('ABESS Path')
plt.legend()
plt.show()
```



Or a view of evolution of information criterion:

```
plt.plot(ic, 'o-')
plt.xlabel('support_size')
plt.ylabel('GIC')
plt.title('Model selection via GIC')
plt.show()
```



Prediction is allowed for all the estimated model. Just call `predict()` function under the model you are interested in. The values it return are $\exp(\eta) = \exp(x\beta)$, which is part of Cox PH hazard function.

Here we give the prediction on the `test` data.

```
pred = model.predict(test[:, 2:])
print(pred)
```

```
[11.0015887  11.97954111  8.11705612  3.32130081  2.9957487   3.23167938
  5.88030263  8.83474265  6.94981468  2.79778448  4.80124013  8.32868839
  6.18472356  7.36597245  2.79540785  7.07729092  3.57284073  6.95551265
  3.59051464  8.73668805  3.51029827  4.28617052  5.21830511  5.11465146
  2.92670651  2.31996184  7.04845409  4.30246362  7.14805341  3.83570919
  6.27832924  6.54442227  8.39353611  5.41713824  4.17823079  4.01469621
  8.99693705  3.98562593  3.9922459   2.79743549  3.47347931  4.40471703
  6.77413094  4.33542254  6.62834299  9.99006885  8.1177072  20.28383502
 14.67346807  2.27915833  5.78151822  4.31221688  3.25950636  6.99318596
  7.4368521   3.86339324]
```

With these predictions, we can compute the hazard ratio between every two observations (by dividing their values). And, we can also compute the C-Index for our model, i.e., the probability that, for a pair of randomly chosen comparable samples, the sample with the higher risk prediction will experience an event before the other sample or belong to a higher binary class.

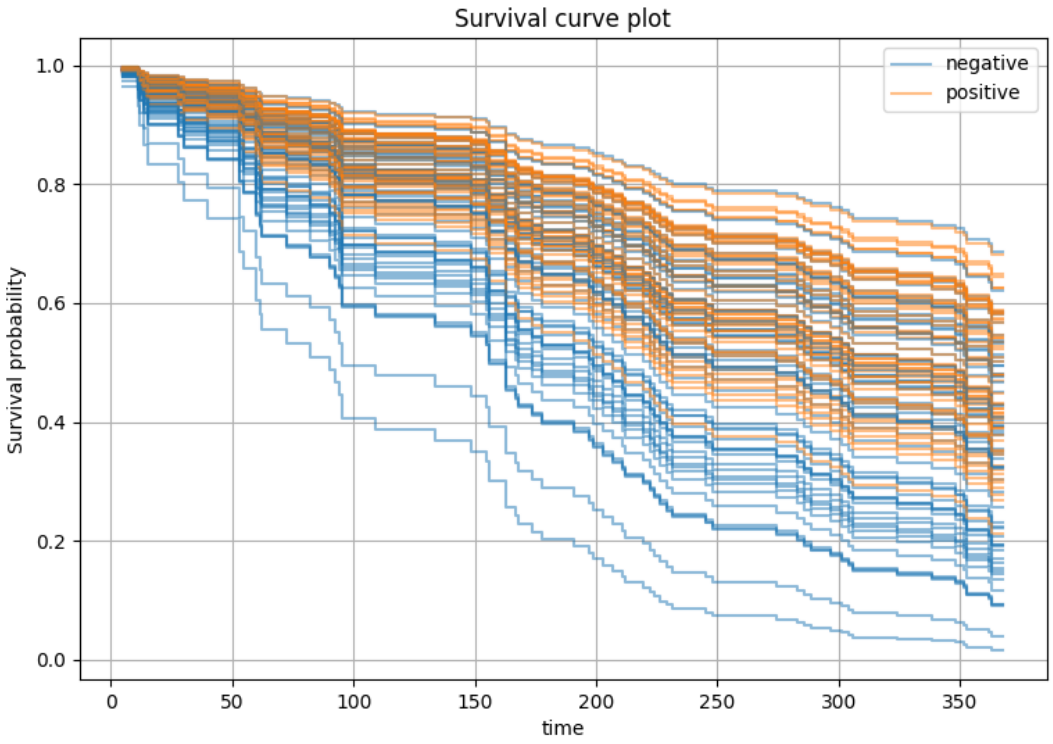
```
test[:, 1] = test[:, 1] == 2
cindex = model.score(test[:, 2:], test[:, :2])
print(cindex)
```

```
0.6839080459770115
```

On this dataset, the C-index is about 0.68.

We can also perform prediction in terms of survival or cumulative hazard function. Here we choose the 9th predictor and we want to see how positive or negative status would affect the survival function for each patient.

```
surv_fns = model.predict_survival_function(train[:, 2:])
time_points = np.quantile(train[:, 0], np.linspace(0, 0.6, 100))
legend_handles = []
legend_labels = []
_, ax = plt.subplots(figsize=(9, 6))
for fn, label in zip(surv_fns, train[:, 8].astype(int)):
    line, = ax.step(time_points, fn(time_points), where="post",
                    color="C{:d}".format(label), alpha=0.5)
    if len(legend_handles) <= label:
        name = "positive" if label == 1 else "negative"
        legend_labels.append(name)
        legend_handles.append(line)

ax.legend(legend_handles, legend_labels)
ax.set_xlabel("time")
ax.set_ylabel("Survival probability")
ax.set_title("Survival curve plot")
ax.grid(True)
```



The graph above shows that patients with positive status on the 9th predictor tend to have better prognosis.

The `abess` R package also supports CoxPH regression. For R tutorial, please view https://abess-team.github.io/abess/articles/v05-coxreg.html. sphinx_gallery_thumbnail_number = 3

Total running time of the script: (0 minutes 0.897 seconds)

Positive Responses: Poisson & Gamma Regressions

Poisson Regression

Poisson Regression involves regression models in which the response variable is in the form of counts. For example, the count of number of car accidents or number of customers in line at a reception desk. The response variables is assumed to follow a Poisson distribution.

The general mathematical equation for Poisson regression is

$$\log(E(y)) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p.$$

Simulated Data Example

We generate some artificial data using this logic. Consider a dataset containing `n=100` observations with `p=6` variables. The `make_glm_data()` function allows uss to generate simulated data. By specifying `k = 3`, we set only 3 of the 6 variables to have effect on the expectation of the response.

```
import numpy as np
from abess.datasets import make_glm_data
np.random.seed(0)

n = 100
p = 6
k = 3
data = make_glm_data(n=n, p=p, k=k, family="poisson")
print("the first 5 x observation:\n", data.x[0:5, ])
print("the first 5 y observation:\n", data.y[0:5])
```

```
the first 5 x observation:
 [[ 0.11025327  0.02500983  0.06117112  0.14005582  0.11672237 -0.06107987]
 [ 0.05938053 -0.00945983 -0.00645118  0.02566241  0.00900272  0.09089209]
 [ 0.04756486  0.00760469  0.02774145  0.02085465  0.09337994 -0.01282239]
 [ 0.01956673 -0.05338098 -0.15956186  0.04085116  0.05402726 -0.04638531]
 [ 0.14185966 -0.09089785  0.00285991 -0.01169899  0.0957987   0.09183492]]
the first 5 y observation:
 [0 0 0 0 0]
```

Notice that, the response have non-negative integer value.

The effective predictors and real coefficients are:

```
print("non-zero:\n", np.nonzero(data.coef_))
print("real coef:\n", data.coef_)
```

```
non-zero:
 (array([0, 2, 5]),)
real coef:
 [-9.28248323  0.         -8.95444082  0.          0.          4.85973384]
```

Model Fitting

The `PoissonRegression()` function in the `abess` package allows you to perform best subset selection in a highly efficient way. We can call the function using formula like:

```
from abess.linear import PoissonRegression

model = PoissonRegression(support_size=range(7))
model.fit(data.x, data.y)
```

where `support_size` contains the level of sparsity we consider, and the program can adaptively choose the "best" one. The result of coefficients can be viewed through `model.coef_`:

```
print(model.coef_)
```

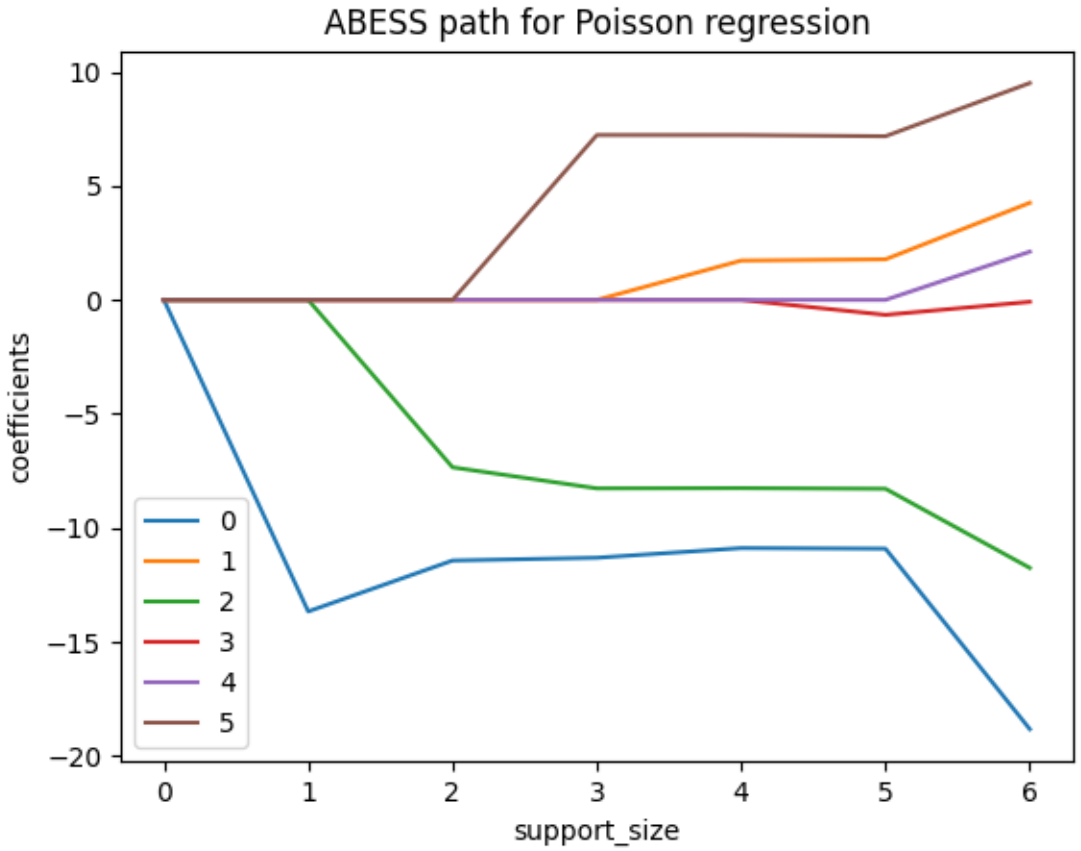
```
[-11.26494228   0.          -8.28071518   0.           0.
   7.2757849 ]
```

So that the first, third and last variables are thought to be useful in the model (the chosen sparsity is 3), which is the same as "real" variables. What's more, the predicted coefficients are also close to the real ones.

More on the Results

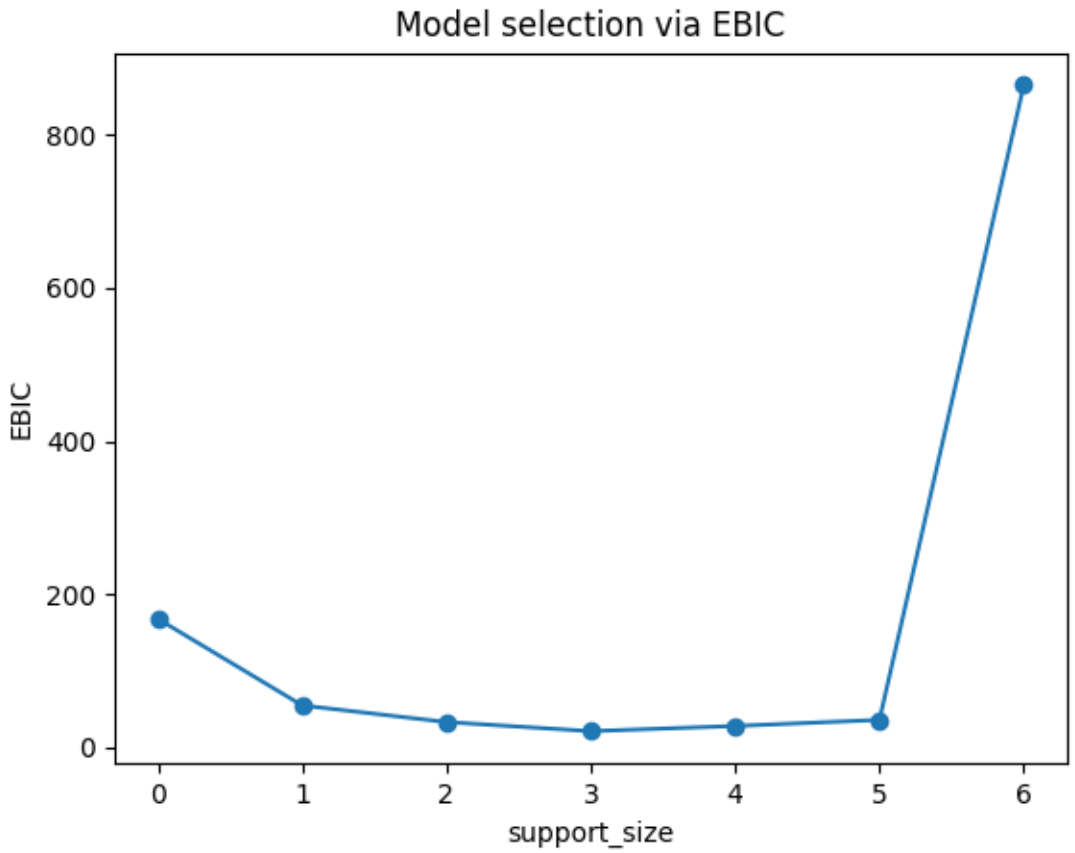
Actually, we can also plot the path of coefficients in `abess` process. This can be computed by fixing the `support_size` as one number from 0 to 6 each time:

```
import matplotlib.pyplot as plt

coef = np.zeros((7, 6))
ic = np.zeros(7)
for s in range(7):
    model = PoissonRegression(support_size=s)
    model.fit(data.x, data.y)
    coef[s, :] = model.coef_
    ic[s] = model.eval_loss_

for i in range(6):
    plt.plot(coef[:, i], label=i)

plt.xlabel('support_size')
plt.ylabel('coefficients')
plt.title('ABESS path for Poisson regression')
plt.legend()
plt.show()
```



And the evolution of information criterion (by default, we use EBIC):

```
plt.plot(ic, 'o-')
plt.xlabel('support_size')
plt.ylabel('EBIC')
plt.title('Model selection via EBIC')
plt.show()
```



The lowest point is shown on `support_size=3` and that's why the program chooses 3 variables as output.

Gamma Regression

Gamma regression can be used when you have positive continuous response variables such as payments for insurance claims, or the lifetime of a redundant system. It is well known that the density of Gamma distribution can be represented as a function of a mean parameter (μ) and a shape parameter (α), respectively,

$$f(y \mid \mu, \alpha) = \frac{1}{y\Gamma(\alpha)} \left(\frac{\alpha y}{\mu}\right)^{\alpha} e^{-\alpha y/\mu} I_{(0,\infty)}(y),$$

where $I(\cdot)$ denotes the indicator function. In the Gamma regression model, response variables are assumed to follow Gamma distributions. Specifically,

$$y_i \sim Gamma(\mu_i, \alpha),$$

where $1/\mu_i = x_i^T \beta$.

Compared with Poisson regression, this time we consider the response variables as (continuous) levels of satisfaction.

Simulated Data Example

Firstly, we also generate data from `make_glm_data()`, but `family = "gamma"` is given this time:

```
np.random.seed(1)

n = 100
p = 6
k = 3
data = make_glm_data(n=n, p=p, k=k, family="gamma")
print("the first 5 x:\n", data.x[0:5, ])
print("the first 5 y:\n", data.y[0:5])
```

```
the first 5 x:
 [[ 0.10152159 -0.03823478 -0.03301073 -0.06706054  0.05408798 -0.14384617]
 [ 0.10905074 -0.04757543  0.01993994 -0.01558565  0.09138175 -0.12875879]
 [-0.02015108 -0.0240034   0.07086059 -0.0687432  -0.01077676 -0.05486615]
 [ 0.00263836  0.03642595 -0.0687887   0.07154523  0.05634942  0.0314059 ]
 [ 0.0563035  -0.04273299 -0.00768064 -0.05848559 -0.016743    0.03314722]]
the first 5 y:
 [0.02229665 0.02624332 0.03562678 0.03518078 0.05029387]
```

Notice that the response y is positive **continuous** value, which is different from the Poisson regression for integer value.

The effective predictors and their effects are presented below:

```
nnz_ind = np.nonzero(data.coef_)
print("non-zero:\n", nnz_ind)
print("non-zero coef:\n", data.coef_[nnz_ind])
```

```
non-zero:
 (array([3, 4, 5]),)
non-zero coef:
 [ 23.56851164  39.0768964  -92.32914775]
```

Model Fitting

We apply the above procedure for gamma regression simply by using `GammaRegression()` in `abess.linear`. It has similar member functions for fitting. We use five fold cross validation (CV) for selecting the model size:

```
from abess.linear import GammaRegression

model = GammaRegression(support_size=range(7), cv=5)
model.fit(data.x, data.y)
```

The fitted coefficients:

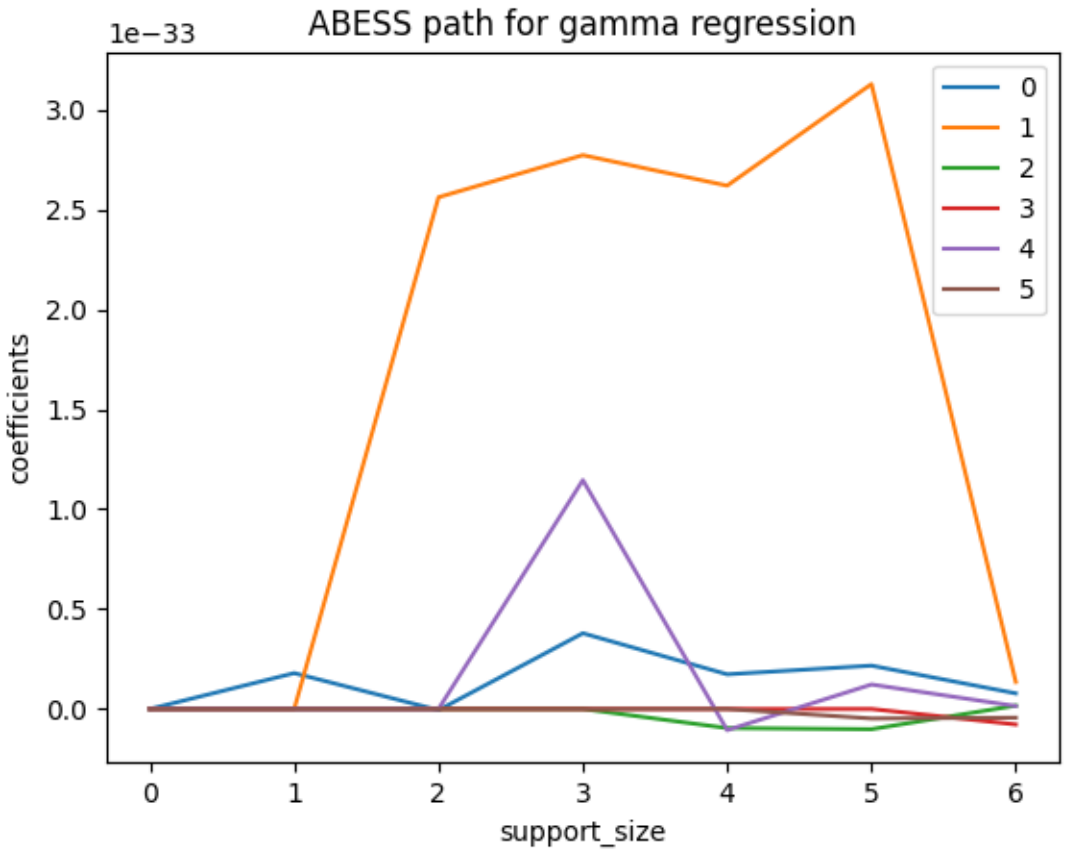
```
print(model.coef_)
```

```
[0. 0. 0. 0. 0. 0.]
```

More on the Results

We can also plot the path of coefficients in abess process.

```
coef = np.zeros((7, 6))
loss = np.zeros(7)
for s in range(7):
    model = GammaRegression(support_size=s)
    model.fit(data.x, data.y)
    coef[s, :] = model.coef_
    loss[s] = model.eval_loss_

for i in range(6):
    plt.plot(coef[:, i], label=i)

plt.xlabel('support_size')
plt.ylabel('coefficients')
plt.title('ABESS path for gamma regression')
plt.legend()
plt.show()
```



The `abess` R package also supports Poisson regression and Gamma regression. For R tutorial, please view https://abess-team.github.io/abess/articles/v04-PoissonGammaReg.html.

Total running time of the script: (0 minutes 0.463 seconds)

Power of abess Library: Empirical Comparison

Introduction

In this part, we are going to explore the power of the `abess` package using simulated data. We compare the abess package with popular Python packages: scikit-learn for linear and logistic regressions in the following section. (Actually, we also compare with python-glmnet, statsmodels and L0bnb, but the `python-glmnet` presents a poor prediction error, the `statsmodels` runs slow and the `L0bnb` cannot adaptively choose sparsity level. So their results are not showed here.)

Simulation Setting

Both packages are compared in three aspects including the prediction performance, the variable selection performance, and the computation efficiency.

- The prediction performance of the linear model is measured by $||y\hat{y}||_2$ on a test set and for logistic regression this is measured by the area under the ROC curve (AUC).
- The coefficient estimation performance are measured by the coefficient error $||\beta - \hat{\beta}||_2$.
- For the variable selection performance, we compute true positive rate (TPR, which is the proportion of variables in the active set that are correctly identified) and the false positive rate (FPR, which is the proportion of the variables in the inactive set that are falsely identified as a signal).
- Timings of the CPU execution are recorded in seconds, and all of methods select the best model among 100 models under different regularization strength or support size.

The simulated data are made by `abess.datasets.make_glm_data()`. The number of predictors is $p = 8000$ and the size of data is $n = 500$. The true coefficient contains $k = 10$ nonzero entries uniformly distributed in $[b, B]$:

- For linear regression (`family = "gaussian"`), we set $b = 5\sqrt{2\ln p/n}$ and $B = 100b$.
- For logistic regression (`family = "binomial"`), we set $b = 10\sqrt{2\ln p/n}$ and $B = 5b$.

In each regression, we test for both low ($\rho = 0.1$) and high correlation ($\rho = 0.7$) scenarios. What's more, a random noise generated from a standard Gaussian distribution is added to the linear predictor $x^\top\beta$ for linear regression.

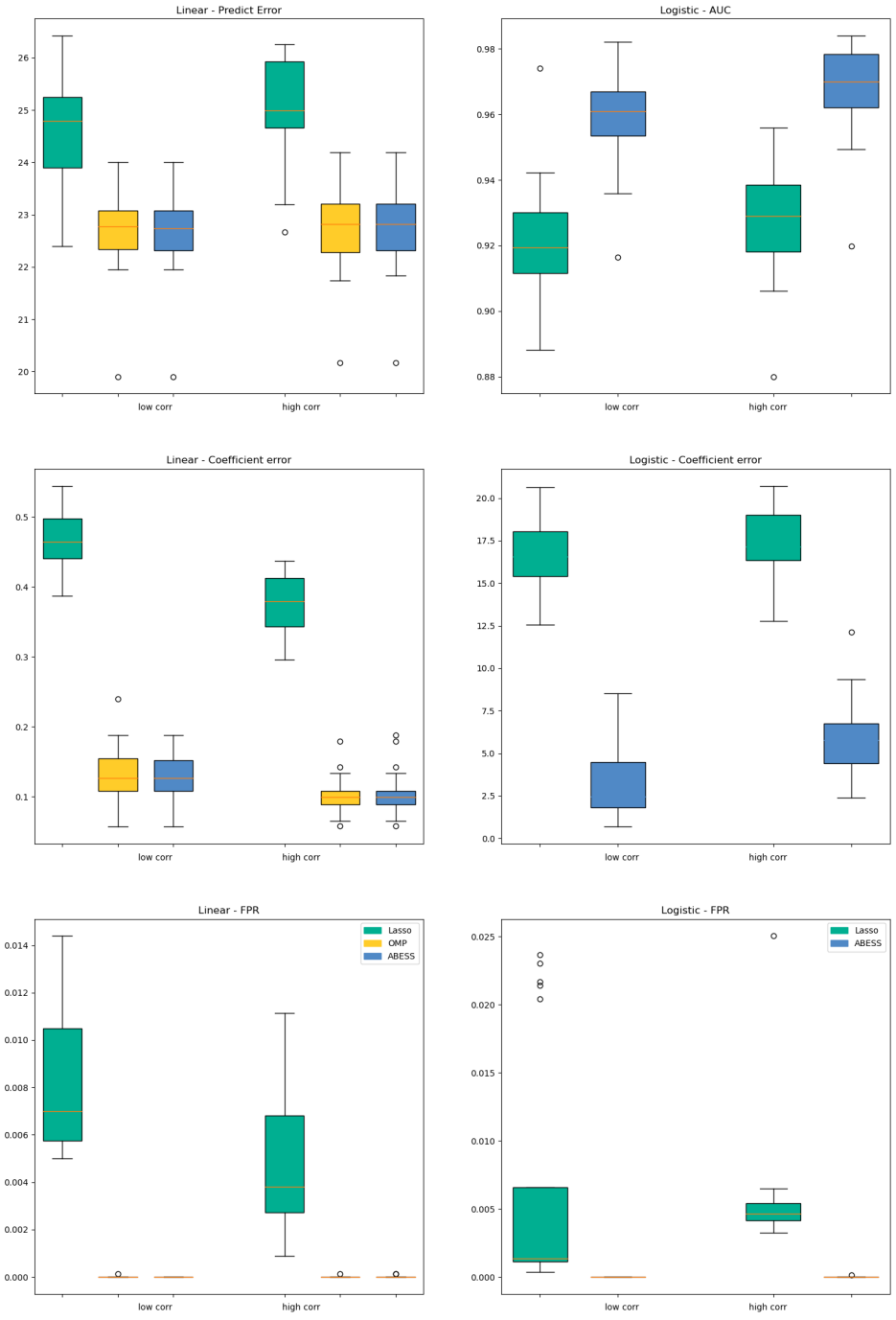
All the performances are averaged over 20 replications. All experiments are evaluated on a Arch Linux platform with Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz and 16 RAM.

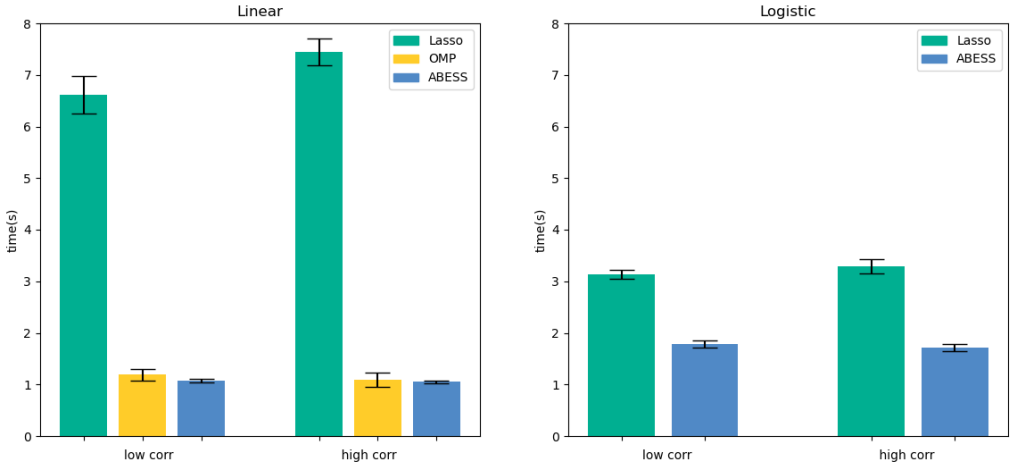
```
$ python abess/docs/simulation/Python/plot_results_figure.py
```

Numerical Results

For linear regression, we compare three methods in the two packages: Lasso, OMP and abess. For logistic regression, we compare two methods: lasso and abess. The results are presented in the following pictures. The first column is the result of linear regression and the second one is of logistic regression.

- Firstly, among all of the methods implemented in different packages, the estimator obtained by the abess package shows both the best prediction performance and the best coefficient error.
- Secondly, the estimator obtained by the abess package can reasonably control FPR in a low level while the TPR stays at 1. (Since all methods' TPR are 1, the figure is not plotted.)
- Furthermore, our abess package is highly efficient compared with other packages, especially in the linear regression.


Linear - Predict Error
Logistic - AUC
Linear - Coefficient error
Logistic - Coefficient error
Linear - FPR
Logistic - FPR
low corr
high corr
Lasso
OMP
ABESS



For `abess` R library's performance, please view https://abess-team.github.io/abess/articles/v11-power-of-abess.html.

sphinx_gallery_thumbnail_path = 'Tutorial/figure/timings.png'

Total running time of the script: (0 minutes 0.000 seconds)

ABESS Algorithm: Details

Introduction

With the `abess` library, users can use the ABESS algorithm to efficiently solve many best subset selection problems. The aim of this page is providing a complete and coherent documentation for ABESS algorithm under linear model such that users can easily understand the ABESS algorithm, thereby facilitating the usage of `abess` software.

linear regression

Sacrifices

Consider the ℓ_0 constraint minimization problem,

$$\min_{\boldsymbol{\beta}} \mathcal{L}_n(\beta), \quad \text{s.t } \|\boldsymbol{\beta}\|_0 \leq \mathrm{s},$$

where $\mathcal{L}_n(\boldsymbol{\beta}) = \frac{1}{2n}\|y - X\beta\|_2^2$. Without loss of generality, we consider $\|\boldsymbol{\beta}\|_0 = \mathrm{s}$. Given any initial set $\mathcal{A} \subset \mathcal{S} = \{1, 2, \ldots, p\}$ with cardinality $|\mathcal{A}| = s$, denote $\mathcal{I} = \mathcal{A}^{\mathrm{c}}$ and compute:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}_{\mathcal{I}}=0} \mathcal{L}_n(\boldsymbol{\beta}).$$

We call $\mathcal{A}$ and $\mathcal{I}$ as the active set and the inactive set, respectively.

Given the active set $\mathcal{A}$ and $\hat{\boldsymbol{\beta}}$, we can define the following two types of sacrifices:

1. Backward sacrifice: For any $j \in \mathcal{A}$, the magnitude of discarding variable j is,

$$\xi_j = \mathcal{L}_n\left(\hat{\boldsymbol{\beta}}^{\mathcal{A}\setminus\{j\}}\right) - \mathcal{L}_n\left(\hat{\boldsymbol{\beta}}^{\mathcal{A}}\right) = \frac{X_j^\top X_j}{2n}\left(\hat{\boldsymbol{\beta}}_j\right)^2,$$

2. Forward sacrifice: For any $j \in \mathcal{I}$, the magnitude of adding variable j is,

$$\zeta_j = \mathcal{L}_n\left(\hat{\boldsymbol{\beta}}^{\mathcal{A}}\right) - \mathcal{L}_n\left(\hat{\boldsymbol{\beta}}^{\mathcal{A}} + \hat{t}^{\{j\}}\right) = \frac{X_j^\top X_j}{2n}\left(\frac{\hat{\boldsymbol{d}}_j}{X_j^\top X_j/n}\right)^2.$$

where $\hat{t} = \arg\min_t \mathcal{L}_n\left(\hat{\boldsymbol{\beta}}^{\mathcal{A}} + t^{\{j\}}\right), \hat{\boldsymbol{d}}_j = X_j^\top(y - X\hat{\boldsymbol{\beta}})/n$. Intuitively, for $j \in \mathcal{A}$ (or $j \in \mathcal{I}$), a large ξ_j (or ζ_j) implies the j th variable is potentially important.

Algorithm

Best-Subset Selection with a Given Support Size

Unfortunately, it is noteworthy that these two sacrifices are incomparable because they have different sizes of support set. However, if we exchange some "irrelevant" variables in $\mathcal{A}$ and some "important" variables in $\mathcal{I}$, it may result in a higher-quality solution. This intuition motivates our splicing method. Specifically, given any splicing size $k \leq s$, define

$$\mathcal{A}_k = \left\{j \in \mathcal{A} : \sum_{i\in\mathcal{A}} \mathrm{I}\left(\xi_j \geq \xi_i\right) \leq k\right\},$$

to represent k least relevant variables in $\mathcal{A}$ and,

$$\mathcal{I}_k = \left\{j \in \mathcal{I} : \sum_{i\in\mathcal{I}} \mid \left(\zeta_j \leq \zeta_i\right) \leq k\right\},$$

to represent k most relevant variables in $\mathcal{I}$.

Then, we splice $\mathcal{A}$ and $\mathcal{I}$ by exchanging $\mathcal{A}_k$ and $\mathcal{I}_k$ and obtain a new active set:$\tilde{\mathcal{A}} = (\mathcal{A}\backslash\mathcal{A}_k) \cup \mathcal{I}_k$. Let $\tilde{\mathcal{I}} = \tilde{\mathcal{A}}^c, \tilde{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}_{\tilde{\mathcal{I}}=0}} \mathcal{L}_n(\boldsymbol{\beta})$, and $\tau_s > 0$ be a threshold. If $\tau_s < \mathcal{L}_n(\hat{\boldsymbol{\beta}}) - \mathcal{L}_n(\tilde{\boldsymbol{\beta}})$, then $\tilde{A}$ is preferable to $\mathcal{A}$. The active set can be updated iteratively until the loss function cannot be improved by splicing. Once the algorithm recovers the true active set, we may splice some irrelevant variables, and then the loss function may decrease slightly. The threshold τ_s can reduce this unnecessary calculation. Typically, τ_s is relatively small, e.g. $\tau_s = 0.01s\log(p)\log(\log n)/n$.

Algorithm 1: BESS.Fix(s): Best-Subset Selection with a given support size s.

1. Input: X, y, a positive integer $k_{\max}$, and a threshold τ_s.
2. Initialize:

$$\mathcal{A}^0 = \left\{j : \sum_{i=1}^{p} \mathrm{I}\left(\left|\frac{X_j^\top y}{\sqrt{X_j^\top X_j}}\right| \leq \left|\frac{X_i^\top y}{\sqrt{X_i^\top X_i}}\right| \leq \mathrm{s}\right\}, \mathcal{I}^0 = \left(\mathcal{A}^0\right)^c$$

and $\left(\boldsymbol{\beta}^0, d^0\right)$:

$$\begin{aligned}
&\boldsymbol{\beta}^0_{\mathcal{I}^0} = 0,\\
&d^0_{\mathcal{A}^0} = 0,\\
&\boldsymbol{\beta}^0_{\mathcal{A}^0} = \left(\boldsymbol{X}^\top_{\mathcal{A}^0}\boldsymbol{X}_{\mathcal{A}^0}\right)^{-1}\boldsymbol{X}^\top_{\mathcal{A}^0}\boldsymbol{y},\\
&d^0_{\mathcal{I}^0} = X^\top_{\mathcal{I}^0}\left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}^0\right).
\end{aligned}$$

3. For $m = 0, 1, \ldots$, do

$$\left(\boldsymbol{\beta}^{m+1}, \boldsymbol{d}^{m+1}, \mathcal{A}^{m+1}, \mathcal{I}^{m+1}\right) = \text{Splicing}\left(\boldsymbol{\beta}^{m}, \boldsymbol{d}^{m}, \mathcal{A}^{m}, \mathcal{I}^{m}, k_{\max}, \tau_s\right).$$

If $\left(\mathcal{A}^{m+1}, \mathcal{I}^{m+1}\right) = \left(\mathcal{A}^{m}, \mathcal{I}^{m}\right)$, then stop.

End For

4. Output $(\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{d}}, \hat{\mathcal{A}}, \hat{\mathcal{I}}) = \left(\boldsymbol{\beta}^{m+1}, \boldsymbol{d}^{m+1}\mathcal{A}^{m+1}, \mathcal{I}^{m+1}\right)$.

Algorithm 2: Splicing $(\beta, d, \mathcal{A}, \mathcal{I}, k_{\max}, \tau_s)$

1. Input: $\boldsymbol{\beta}, \boldsymbol{d}, \mathcal{A}, \mathcal{I}, k_{\max}$, and τ_{s}.
2. Initialize: $L_0 = L = \frac{1}{2n}\|y - X\beta\|_2^2$, and set

$$\xi_j = \frac{X_j^\top X_j}{2n}\left(\beta_j\right)^2, \zeta_j = \frac{X_j^\top X_j}{2n}\left(\frac{d_j}{X_j^\top X_j/n}\right)^2, j = 1, \ldots, p.$$

3. For $k = 1, 2, \ldots, k_{\max}$, do

$$\mathcal{A}_k = \left\{j \in \mathcal{A} : \sum_{i \in \mathcal{A}} \mathrm{I}\left(\xi_j \geq \xi_i\right) \leq k\right\},$$
$$\mathcal{I}_k = \left\{j \in \mathcal{I} : \sum_{i \in \mathcal{I}} \mathrm{I}\left(\zeta_j \leq \zeta_i\right) \leq k\right\}.$$

Let $\tilde{\mathcal{A}}_k = (\mathcal{A} \backslash \mathcal{A}_k) \cup \mathcal{I}_k, \tilde{\mathcal{I}}_k = (\mathcal{I} \backslash \mathcal{I}_k) \cup \mathcal{A}_k$ and solve:

$$\tilde{\boldsymbol{\beta}}_{\mathcal{A}_k} = \left(\boldsymbol{X}_{\mathcal{A}_k}^\top \boldsymbol{X}_{\mathcal{A}_k}\right)^{-1} \boldsymbol{X}_{\mathcal{A}_k}^\top y, \quad \tilde{\boldsymbol{\beta}}_{\mathcal{I}_k} = 0$$
$$\tilde{\boldsymbol{d}}_{\mathcal{I}^k} = X_{\mathcal{I}^k}^\top (y - X\tilde{\beta})/n, \quad \tilde{\boldsymbol{d}}_{\mathcal{A}^k} = 0.$$

Compute: $\mathcal{L}_n(\tilde{\boldsymbol{\beta}}) = \frac{1}{2n}\|y - X\tilde{\boldsymbol{\beta}}\|_2^2$. If $L > \mathcal{L}_n(\tilde{\boldsymbol{\beta}})$, then

$$(\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{d}}, \hat{\mathcal{A}}, \hat{\mathcal{I}}) = \left(\tilde{\boldsymbol{\beta}}, \tilde{\boldsymbol{d}}, \tilde{\mathcal{A}}_k, \tilde{\mathcal{I}}_k\right)$$
$$L = \mathcal{L}_n(\tilde{\boldsymbol{\beta}}).$$

End for

4. If $L_0 - L < \tau_s$, then $(\hat{\boldsymbol{\beta}}, \hat{d}, \hat{A}, \hat{I}) = (\boldsymbol{\beta}, d, \mathcal{A}, \mathcal{I})$.
5. Output $(\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{d}}, \hat{\mathcal{A}}, \hat{\mathcal{I}})$.

Determining the Best Support Size with SIC

In practice, the support size is usually unknown. We use a datadriven procedure to determine s. For any active set $\mathcal{A}$, define an SIC as follows:

$$\mathrm{SIC}(\mathcal{A}) = n \log \mathcal{L}_{\mathcal{A}} + |\mathcal{A}| \log(p) \log\log n,$$

where $\mathcal{L}_{\mathcal{A}} = \min_{\beta_{\mathcal{I}}=0} \mathcal{L}_n(\beta), \mathcal{I} = (\mathcal{A})^c$. To identify the true model, the model complexity penalty is $\log p$ and the slow diverging rate $\log \log n$ is set to prevent underfitting. Theorem 4 states that the following ABESS algorithm selects the true support size via SIC.

Let $s_{\max}$ be the maximum support size. We suggest $s_{\max} = o\left(\frac{n}{\log p}\right)$ as the maximum possible recovery size. Typically, we set $s_{\max} = \left[\frac{n}{\log p \log \log n}\right]$ where $[x]$ denotes the integer part of x.

Algorithm 3: ABESS.

1. Input: X, y, and the maximum support size $s_{\max}$.
2. For $s = 1, 2, \ldots, s_{\max}$, do

   $$\left(\hat{\boldsymbol{\beta}}_s, \hat{\boldsymbol{d}}_s, \hat{\mathcal{A}}_s, \hat{\mathcal{I}}_s\right) = \text{BESS.Fixed}(s).$$

   End for
3. Compute the minimum of SIC:

   $$s_{\min} = \arg\min_s \text{SIC}\left(\hat{\mathcal{A}}_s\right).$$

4. Output $\left(\hat{\boldsymbol{\beta}}_{s_{\min}}, \hat{\boldsymbol{d}}_{s_{\min}}, \hat{A}_{s_{\min}}, \hat{\mathcal{I}}_{s_{\min}}\right)$.

Now, enjoy the data analysis with `abess` library:

```
import abess

# sphinx_gallery_thumbnail_path = 'Tutorial/figure/icon_noborder.png'
```

Total running time of the script: (0 minutes 0.000 seconds)

Principal Component Analysis

Principal Component Analysis

This notebook introduces what is adaptive best subset selection principal component analysis (SparsePCA) and uses a real data example to show how to use it.

PCA

Principal component analysis (PCA) is an important method in the field of data science, which can reduce the dimension of data and simplify our model. It actually solves an optimization problem like:

$$\max_v v^\top \Sigma v, \qquad s.t. \quad v^T v = 1.$$

where $\Sigma = X^T X/(n-1)$ and X is the **centered** sample matrix. We also denote that X is a $n \times p$ matrix, where each row is an observation and each column is a variable.

Then, before further analysis, we can project X to v (thus dimension reduction), without losing too much information.

However, consider that:

- The PC is a linear combination of all primary variables (Xv), but sometimes we may tend to use less variables for clearer interpretation (and less computational complexity);
- It has been proved that if p/n does not converge to 0, the classical PCA is not consistent, but this would happen in some high-dimensional data analysis.

For example, in gene analysis, the dataset may contain plenty of genes (variables) and we would like to find a subset of them, which can explain most information. Compared with using all genes, this small subset may perform better on interpretation, without losing much information. Then we can focus on these variables in further analysis.

When we are trapped by these problems, a classical PCA may not be a best choice, since it uses all variables. One of the alternatives is *SparsePCA*, which is able to seek for principal component with a sparsity limitation:

$$\max_v v^\top \Sigma v, \qquad s.t. \quad v^T v = 1, \ ||v||_0 \leq s.$$

where s is a non-negative integer, which indicates how many primary variables are used in principal component. With *SparsePCA*, we can search for the best subset of variables to form principal component and it retains consistency even under $p >> n$. And we make two remarks:

- Clearly, if s is equal or larger than the number of primary variables, this sparsity limitation is actually useless, so the problem is equivalent to a classical PCA.
- With less variables, the PC must have lower explained variance. However, this decrease is slight if we choose a good s and at this price, we can interpret the PC much better. It is worthy.

In the next section, we will show how to perform *SparsePCA*.

Real Data Example (Communities and Crime Dataset)

Here we will use real data analysis to show how to perform *SparsePCA*. The data we use is from UCI: Communities and Crime Data Set and we pick up its 99 predictive variables as our samples.

Firstly, we read the data and pick up those variables we are interested in.

```
import numpy as np
from abess.decomposition import SparsePCA

X = np.genfromtxt('communities.data', delimiter=',')
X = X[:, 5:127]                          # numeric predictiors
X = X[:, ~np.isnan(X).any(axis=0)]    # drop variables with nan

n, p = X.shape
print(n)
print(p)
```

```
1994
99
```

Model fitting

To build an SparsePCA model, we need to give the target sparisty to its *support_size* argument. Our program supports adaptively finding a best sparisty in a given range.

Fixed sparsity

If we only focus on one fixed sparsity, we can simply give a single integer to fit on this situation. And then the fitted sparse principal component is stored in *SparsePCA.coef_*:

```
model = SparsePCA(support_size=20)
```

Give either X or Σ to *model.fit()*, the fitting process will start. The argument *is_normal = False* here means that the program will not normalize X. Note that if both X and $Sigma$ are given, the program prefers to use X.

```
model.fit(X=X, is_normal=False)
# model.fit(Sigma = np.cov(X.T))
```

After fitting, *model.coef_* returns the sparse principal component and its non-zero positions correspond to variables used.

```
temp = np.nonzero(model.coef_)[0]
print('sparsity: ', temp.size)
print('non-zero position: \n', temp)
print(model.coef_.T)
```

```
sparsity:  20
non-zero position:
 [ 3  4  5 17 28 49 55 56 57 58 59 60 61 62 65 66 67 69 90 96]
[[ 0.          0.          0.         -0.20905321  0.15082783  0.26436836
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.13962306
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.14795039  0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.13366389  0.          0.          0.          0.
   0.          0.28227473  0.28982717  0.29245315  0.29340846 -0.27796975
   0.27817835  0.20793385  0.19020622  0.          0.          0.15462671
  -0.13627653  0.25848049  0.         -0.14433668  0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.27644605  0.          0.          0.          0.          0.
   0.16881151  0.          0.        ]]
```

Adaptive sparsity

What's more, **abess** also support a range of sparsity and adaptively choose the best-explain one. However, usually a higher sparsity level would lead to better explaination.

Now, we need to build an $s_{max} \times 1$ binomial matrix, where s_{max} indicates the max target sparsity and each row indicates one sparsity level (i.e. start from 1, till s_{max}). For each position with 1, **abess** would try to fit the model under that sparsity and finally give the best one.

```
# fit sparsity from 1 to 20
support_size = np.ones((20, 1))
# build model
model = SparsePCA(support_size=support_size)
model.fit(X, is_normal=False)
# results
temp = np.nonzero(model.coef_)[0]
print('chosen sparsity: ', temp.size)
print('non-zero position: \n', temp)
print(model.coef_.T)
```

```
chosen sparsity:  20
non-zero position:
 [11 12 17 19 20 21 27 29 30 35 36 44 76 78 79 80 81 82 83 84]
[[ 0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.         -0.27618663
  -0.23703262  0.          0.          0.          0.          0.18384733
   0.         -0.2275236  -0.21204903 -0.19753942  0.          0.
   0.          0.          0.          0.21358573  0.          0.18270928
  -0.18928695  0.          0.          0.          0.          0.1760962
  -0.17481418  0.          0.          0.          0.          0.
   0.          0.         -0.18581084  0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.23804122  0.
  -0.2415995  -0.24785373 -0.24947283 -0.23938391 -0.23605314 -0.28015859
  -0.23841083  0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.        ]]
```

Because of warm-start, the results here may not be the same as fixed sparsity.

Then, the explained variance can be computed by:

```
Xc = X - X.mean(axis=0)
Xv = Xc @ model.coef_
explained = Xv.T @ Xv                    # explained variance (information)
total = sum(np.diag(Xc.T @ Xc))          # total variance (information)
print('explained ratio: ', explained / total)
```

```
explained ratio:  [[0.16920803]]
```

More on the results

We can give different target sparsity (change *s_begin* and *s_end*) to get different sparse loadings. Interestingly, we can seek for a smaller sparsity which can explain most of the variance.

In this example, if we try sparsities from 0 to p, and calculate the ratio of explained variance:

```
num = 30
i = 0
sparsity = np.linspace(1, p - 1, num, dtype='int')
explain = np.zeros(num)
Xc = X - X.mean(axis=0)
for s in sparsity:
    model = SparsePCA(
        support_size=np.ones((s, 1)),
        exchange_num=int(s),
        max_iter=50
    )
    model.fit(X, is_normal=False)
    Xv = Xc @ model.coef_
    explain[i] = Xv.T @ Xv
    i += 1

print('80%+ : ', sparsity[explain > 0.8 * explain[num - 1]])
print('90%+ : ', sparsity[explain > 0.9 * explain[num - 1]])
```

```
80%+ :  [31 34 37 41 44 47 51 54 57 61 64 67 71 74 77 81 84 87 91 94 98]
90%+ :  [41 44 47 51 54 57 61 64 67 71 74 77 81 84 87 91 94 98]
```

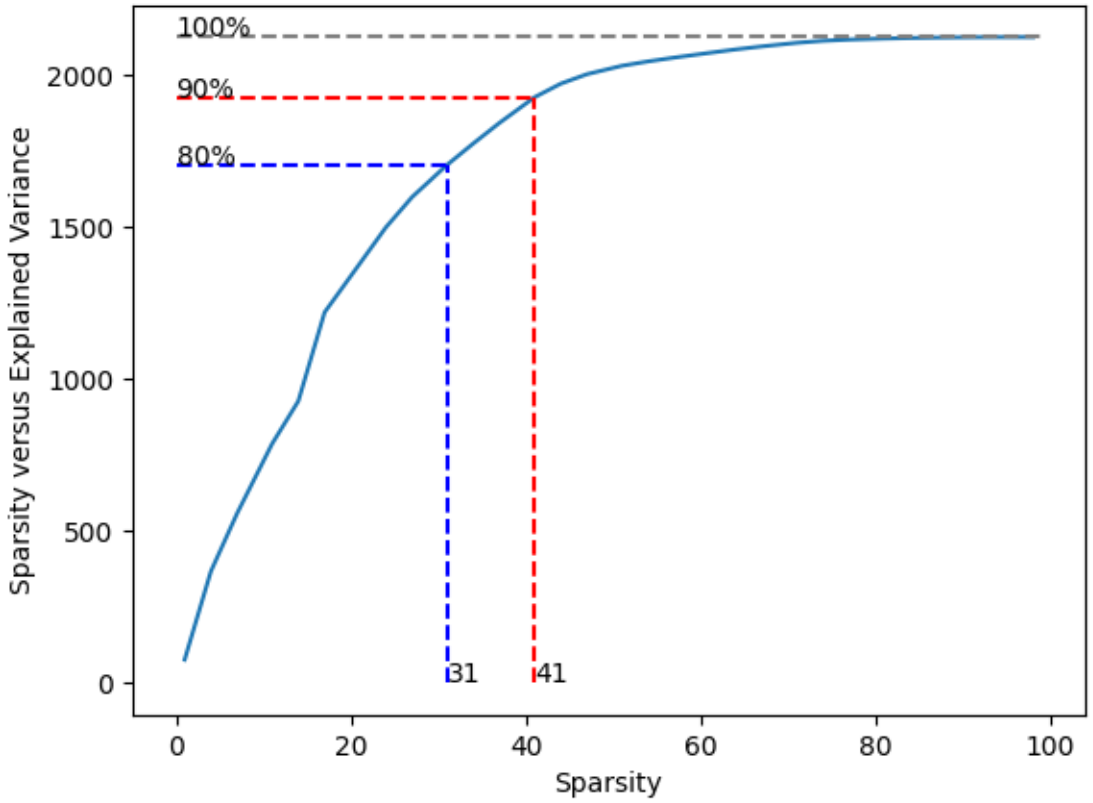
If we denote the explained ratio from all 99 variables as 100%, the curve indicates that at least 31 variables can reach 80% (blue dashed line) and 41 variables can reach 90% (red dashed line).

```
import matplotlib.pyplot as plt

plt.plot(sparsity, explain)
plt.xlabel('Sparsity')
plt.ylabel('Explained variance')
plt.ylabel('Sparsity versus Explained Variance')

ind = np.where(explain > 0.8 * explain[num - 1])[0][0]
plt.plot([0, sparsity[ind]], [explain[ind], explain[ind]], 'b--')
plt.plot([sparsity[ind], sparsity[ind]], [0, explain[ind]], 'b--')
plt.text(sparsity[ind], 0, str(sparsity[ind]))
plt.text(0, explain[ind], '80%')

ind = np.where(explain > 0.9 * explain[num - 1])[0][0]
plt.plot([0, sparsity[ind]], [explain[ind], explain[ind]], 'r--')
plt.plot([sparsity[ind], sparsity[ind]], [0, explain[ind]], 'r--')
plt.text(sparsity[ind], 0, str(sparsity[ind]))
plt.text(0, explain[ind], '90%')

plt.plot([0, p], [explain[num - 1], explain[num - 1]],
         color='gray', linestyle='--')
```

(continues on next page)

(continued from previous page)

```
plt.text(0, explain[num - 1], '100%')

plt.show()
```



This result shows that using less than half of all 99 variables can be close to perfect. For example, if we choose sparsity 31, the used variables are:

```
model = SparsePCA(support_size=31)
model.fit(X, is_normal=False)
temp = np.nonzero(model.coef_)[0]
print('non-zero position: \n', temp)
```

```
non-zero position:
 [ 2 11 12 15 17 19 20 21 22 25 27 28 29 30 31 32 35 36 42 43 44 45 49 76
 78 79 80 81 82 83 84]
```

Extension: Group PCA

Group PCA

Furthermore, in some situations, some variables may need to be considered together, that is, they should be "used" or "unused" for PC at the same time, which we call "group information". The optimization problem becomes:

$$\max_v v^\top \Sigma v, \qquad s.t. \quad v^T v = 1, \sum_{g=1}^{G} I(||v_g|| \neq 0) \leq s.$$

where we suppose there are G groups, and the g-th one correspond to v_g, $v = [v_1^\top, v_2^\top, \cdots, v_G^\top]^\top$. Then we are interested to find s (or less) important groups.

Group problem is extraordinarily important in real data analysis. Still take gene analysis as an example, several sites would be related to one character, and it is meaningless to consider each of them alone.

SparsePCA can also deal with group information. Here we make sure that variables in the same group address close to each other (if not, the data should be sorted first).

Simulated Data Example

Suppose that the data above have group information like:

- Group 0: {the 1st, 2nd, ..., 6th variable};
- Group 1: {the 7th, 8th, ..., 12th variable};
- ...
- Group 15: {the 91st, 92nd, ..., 96th variable};
- Group 16: {the 97th, 98th, 99th variables}.

Denote different groups as different numbers:

```
g_info = np.arange(17)
g_info = g_info.repeat(6)
g_info = g_info[0:99]

print(g_info)
```

```
[ 0  0  0  0  0  0  1  1  1  1  1  1  2  2  2  2  2  2  3  3  3  3  3  3
  4  4  4  4  4  4  5  5  5  5  5  5  6  6  6  6  6  6  7  7  7  7  7  7
  8  8  8  8  8  8  9  9  9  9  9  9 10 10 10 10 10 10 11 11 11 11 11 11
 12 12 12 12 12 12 13 13 13 13 13 13 14 14 14 14 14 14 15 15 15 15 15 15
 16 16 16]
```

And fit a group sparse PCA model with additional argument *group=g_info*:

```
model = SparsePCA(support_size=np.ones((6, 1)), group=g_info)
model.fit(X, is_normal=False)
```

The result comes to:

```
print(model.coef_.T)

temp = np.nonzero(model.coef_)[0]
temp = np.unique(g_info[temp])

print('non-zero group: \n', temp)
print('chosen sparsity: ', temp.size)
```

```
[[-0.04658786 -0.08867352 -0.04812687  0.20196704 -0.15626888 -0.24420959
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
  -0.03874654 -0.12602642 -0.05981463 -0.11432162 -0.13193075 -0.13909102
  -0.14829175 -0.28195208 -0.28747947 -0.28866805 -0.28772941  0.25581611
  -0.25726917 -0.19242565 -0.17526315 -0.08740502 -0.06877806 -0.14679731
   0.1392863  -0.24410922 -0.10815193  0.14329729 -0.03971005  0.00862702
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
  -0.26329492  0.12107731  0.07789811  0.04291127  0.0606106  -0.00599532
   0.          0.          0.        ]]
non-zero group:
 [ 0  8  9 10 11 15]
chosen sparsity:  6
```

Hence we can focus on variables in Group 0, 8, 9, 10, 11, 15.

Extension: Multiple principal components

Multiple principal components

In some cases, we may seek for more than one principal components under sparsity. Actually, we can iteratively solve the largest principal component and then mapping the covariance matrix to its orthogonal space:

$$\Sigma' = (1 - vv^\top)\Sigma(1 - vv^\top)$$

where Σ is the currect covariance matrix and v is its (sparse) principal component. We map it into Σ', which indicates the orthogonal space of v, and then solve the sparse principal component again.

By this iteration process, we can acquire multiple principal components and they are sorted from the largest to the smallest. In our program, there is an additional argument *number*, which indicates the number of principal components we need, defaulted by 1. Now the *support_size* is shaped in $s_{max} \times$ number and each column indicates one principal component.

```
model = SparsePCA(support_size=np.ones((31, 3)))
model.fit(X, is_normal=False, number=3)
model.coef_.shape
```

```
(99, 3)
```

Here, each column of the *model.coef_* is a sparse PC (from the largest to the smallest), for example the second one is that:

```
model.coef_[:, 1]
```

```
array([ 0.        ,  0.        ,  0.        , -0.19036307,  0.1546593 ,
        0.22909258,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.15753326,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.10274482,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.11546037,
        0.        ,  0.        ,  0.11150352,  0.1198269 ,  0.12927627,
        0.27435616,  0.28022643,  0.28220338,  0.28150593, -0.25022706,
        0.24872368,  0.17726069,  0.15920166,  0.        ,  0.        ,
        0.12531301, -0.13462701,  0.22695789,  0.11034058, -0.14302014,
        0.        ,  0.        , -0.11505769,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.10140235,  0.10764364,
        0.10731785,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.1151606 ,  0.        ,  0.        ,  0.        ,
        0.26358121, -0.11368877,  0.        ,  0.        ,  0.        ,
        0.        ,  0.16906762,  0.11129358,  0.        ])
```

If we want to compute the explained variance of them, it is also quite easy:

```
Xv = Xc.dot(model.coef_)
explained = np.sum(np.diag(Xv.T.dot(Xv)))
print('explained ratio: ', explained / total)
```

```
explained ratio:  0.46018459657037164
```

R tutorial

For R tutorial, please view https://abess-team.github.io/abess/articles/v08-sPCA.html.

Total running time of the script: (0 minutes 8.625 seconds)

Robust Principal Component Analysis

This notebook introduces what is adaptive best subset selection robust principal component analysis (RobustPCA) and then we show how it works using **abess** package on an artificial example.

PCA

Principal component analysis (PCA) is an important method in the field of data science, which can reduce the dimension of data and simplify our model. It solves an optimization problem like:

$$\max_{v} v^T \Sigma v, \qquad s.t. \quad v^T v = 1.$$

where $\Sigma = X^T X/(n-1)$ and $X \in \mathbb{R}^{n \times p}$ is the centered sample matrix with each row containing one observation of p variables.

Robust-PCA (RPCA)

However, the original PCA is sensitive to outliers, which may be unavoidable in real data:

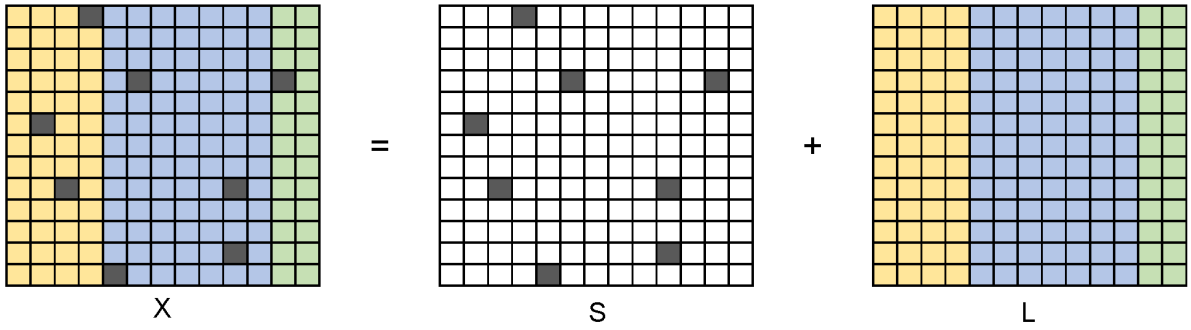
- Object has extreme performance due to fortuity, but he/she shows normal in repeated tests;
- Wrong observation/recording/computing, e.g. missing or dead pixels, X-ray spikes.

In this situation, PCA may spend too much attention on unnecessary variables. That's why Robust-PCA (RPCA) is presented, which can be used to recover the (low-rank) sample for subsequent processing.

In mathematics, RPCA manages to divide the sample matrix X into two parts:

$$X = S + L,$$

where S is the sparse "outlier" matrix and L is the "information" matrix with a low rank. Generally, we also suppose S is not low-rank and L is not sparse, in order to get unique solution.



In Lagrange format,

$$\min_{S,L} \|X - S - L\|_F \leq \varepsilon, s.t. \quad \text{rank}(L) = r, \|S\|_0 \leq s$$

where s is the sparsity of S. After RPCA, the information matrix L can be used in further analysis.

Note that it does NOT deal with "noise", which may stay in L and need further procession.

Hard Impute

To solve its sub-problem, RPCA under known outlier positions, we follow a process called "Hard Impute". The main idea is to estimate the outlier values by precise values with KPCA, where $K = r$.

Here are the steps:

1. Input $X, outliers, M, \varepsilon$, where $outliers$ records the non-zero positions in S;
2. Denote $X_{\text{new}} \leftarrow \mathbf{0}$ with the same shape of X;
3. For $i = 1, 2, \ldots, M$:
   - $X_{\text{old}} = \begin{cases} X_{\text{new}}, & \text{for } outliers \\ X, & \text{for others} \end{cases}$;
   - Form KPCA on X_{old} with $K = r$, and denote v as the eigenvectors;
   - $X_{\text{new}} = X_{\text{old}} \cdot v \cdot v^T$;
   - If $\|X_{\text{new}} - X_{\text{old}}\| < \varepsilon$, break;

   End for;
4. Return X_{new} as L;

where M is the maximum iteration times and ε is the convergence coefficient.

The final X_{new} is supposed to be L under given outlier positions.

RPCA Application

Recently, RPCA is more widely used, for example,

- Video Decomposition: in a surveillance video, the background may be unchanged for a long time while only a few pixels (e.g. people) update. In order to improve the efficiency of storing and analysis, we need to decomposite the video into background and foreground. Since the background is unchanged, it can be stored well in a low-rank matrix, while the foreground, which is usually quite small, can be indicated by a sparse matrix. That is what RPCA does.
- Face recognition: due to complex lighting conditions, a small part of the facial features may be unrecognized (e.g. shadow). In the face recognition, we need to remove the effects of shadows and focus on the face data. Actually, since the face data is almost unchanged (for one person), and the shadows affect only a small part, it is also a suitable situation to use RPCA. Here are some examples:

The shadow effect in face recognition

Simulated Data Example

Fitting model

Now we generate an example with 100 rows and 100 columns with 200 outliers. We are looking forward to recovering it with a low rank 10.

```
from abess.decomposition import RobustPCA
import numpy as np


def gen_data(n, p, s, r, seed=0):
    np.random.seed(seed)
    outlier = np.random.choice(n * p, s, replace=False)
    outlier = np.vstack((outlier // p, outlier % p)).T
    L = np.dot(np.random.rand(n, r), np.random.rand(r, n))
    S = np.zeros((n, p))
    S[outlier[:, 0], outlier[:, 1]] = float(np.random.randn(1)) * 10
    X = L + S
    return X, S


n = 100     # rows
p = 100     # columns
s = 200     # outliers
r = 10      # rank(L)

X, S = gen_data(n, p, s, r)
print(f'X shape: {X.shape}')
# print(f'outlier: \n{outlier}')
```

```
X shape: (100, 100)
```

In order to use our program, users should call `RobustPCA()` and give the outlier number to `support_size`. Note that it can be a specific integer or an integer interval. For the latter case, a support size will be chosen by information criterion (e.g. GIC) adaptively.

```
# support_size can be a interval like `range(s_min, s_max)`
model = RobustPCA(support_size=s)
```

It is quite easy to fit this model, with `RobustPCA.fit` function. Given the original sample matrix X and $rank(L)$ we want, the program will give a result quickly.

```
model.fit(X, r=r)  # r=rank(L)
```

Now the estimated outlier matrix is stored in `model.coef_`.

```
S_est = model.coef_
print(f'estimated sparsity: {np.count_nonzero(S_est)}')
```

```
estimated sparsity: 200
```

More on the result

To check the performance of the program, we use TPR, FPR as the criterion.

```
def TPR(pred, real):
    TP = (pred != 0) & (real != 0)
    P = (real != 0)
    return sum(sum(TP)) / sum(sum(P))


def FPR(pred, real):
    FP = (pred != 0) & (real == 0)
    N = (real == 0)
    return sum(sum(FP)) / sum(sum(N))


def test_model(pred, real):
    tpr = TPR(pred, real)
    fpr = FPR(pred, real)
    return np.array([tpr, fpr])


print(f'[TPR  FPR] = {test_model(S_est, S)}')
```

```
[TPR  FPR] = [0.925      0.00153061]
```

We can also change different random seed to test for more situation:

```
M = 30  # use 30 different seed
res = np.zeros(2)
for seed in range(M):
    X, S = gen_data(n, p, s, r, seed)
    model = RobustPCA(support_size=s).fit(X, r=r)
    res += test_model(model.coef_, S)

print(f'[TPR  FPR] = {res/M}')
```

```
[TPR  FPR] = [0.89866667 0.00206803]
```

Under all of these situations, `RobustPCA` has a good performance.

R tutorial

For R tutorial, please view https://abess-team.github.io/abess/articles/v08-sPCA.html.

sphinx_gallery_thumbnail_path = 'Tutorial/figure/rpca_shadow.png'

Total running time of the script: (0 minutes 1.132 seconds)

Advanced Generic Features

When analyzing the real world datasets, we may have the following targets:

1. identifying predictors when group structure are provided (a.k.a., **best group subset selection**);
2. certain variables must be selected when some prior information is given (a.k.a., **nuisance regression**);
3. selecting the weak signal variables when the prediction performance is mainly interested (a.k.a., **regularized best-subset selection**).

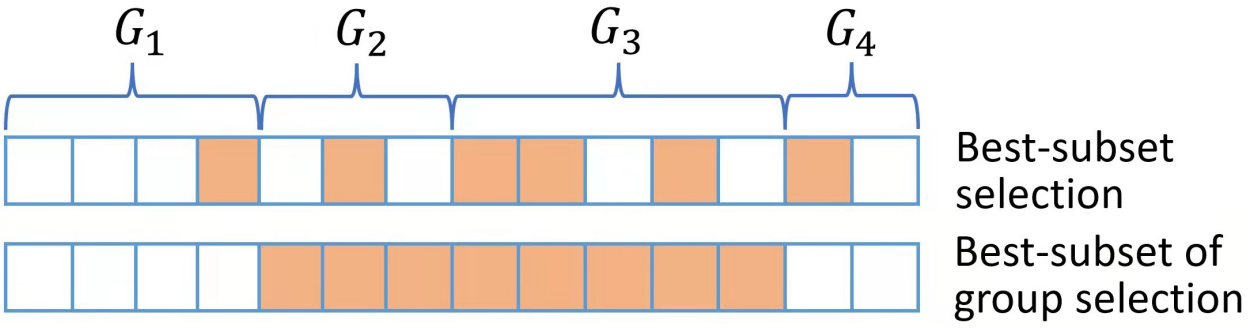
These targets are frequently encountered in real world data analysis. Actually, in our methods, the targets can be properly handled by simply change some default arguments in the functions. In the following content, we will illustrate the statistic methods to reach these targets in a one-by-one manner, and give quick examples to show how to perform the statistic methods in `LinearRegression` and the same steps can be implemented in all methods.

Besides, `abess` library is very flexible, i.e., users can flexibly control many internal computational components. Specifically, users can specify: (i) the division of samples in cross validation (a.k.a., **cross validation division**), (ii) specify the initial active set before splicing (a.k.a., **initial active set**), and so on. We will also describe these in the following.

Best Subset of Group Selection

Introduction

Best subset of group selection (BSGS) aims to choose a small part of non-overlapping groups to achieve the best interpretability on the response variable. BSGS is practically useful for the analysis of ubiquitously existing variables with certain group structures. For instance, a categorical variable with several levels is often represented by a group of dummy variables. Besides, in a nonparametric additive model, a continuous component can be represented by a set of basis functions (e.g., a linear combination of spline basis functions). Finally, specific prior knowledge can impose group structures on variables. A typical example is that the genes belonging to the same biological pathway can be considered as a group in the genomic data analysis. Figure for distinct BSGS and best-subset selection is presented below.



The BSGS can be achieved by solving:

$$\min_{\beta \in \mathbb{R}^p} \frac{1}{2n} ||y - X\beta||_2^2, \text{ s.t. } ||\beta||_{0,2} \leq s.$$

where $||\beta||_{0,2} = \sum_{j=1}^{J} I(||\beta_{G_j}||_2 \neq 0)$ in which $||\cdot||_2$ is the ℓ_2 norm and model size s is a positive integer to be determined from data.

Regardless of the NP-hard of this problem, Zhang et al develop a certifiably polynomial algorithm to solve it. This algorithm is integrated in the `abess` package, and user can handily select best group subset by assigning a proper value to the `group` arguments:

Using best group subset selection

We still use the dataset `data` generated before, which has 100 samples, 5 useful variables and 15 irrelevant variables.

```
import numpy as np
from abess.datasets import make_glm_data
from abess.linear import LinearRegression

np.random.seed(0)

# generate data
n = 100
p = 20
k = 5
coef1 = 0.5*np.ones(5)
coef2 = np.zeros(5)
coef3 = 0.5*np.ones(5)
coef4 = np.zeros(5)
coef = np.hstack((coef1, coef2, coef3, coef4))
data = make_glm_data(n=n, p=p, k=k, family='gaussian', coef_ = coef)
print('real coefficients:\n', data.coef_, '\n')
```

```
real coefficients:
 [0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.5 0.5 0.5 0.5 0.5 0.  0.  0.
 0.  0. ]
```

Support we have some prior information that every 5 variables as a group:

```
group = np.linspace(0, 3, 4).repeat(5)
print('group index:\n', group)
```

```
group index:
 [0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 3. 3. 3. 3. 3.]
```

Then we can set the `group` argument in function. Besides, the `support_size` here indicates the number of groups, instead of the number of variables.

```
model1 = LinearRegression(support_size=range(3), group=group)
model1.fit(data.x, data.y)
print('coefficients:\n', model1.coef_)
```

```
coefficients:
 [0.65915697 0.45713643 0.49044526 0.43927599 0.62863533 0.
 0.         0.         0.         0.         0.575272   0.41249505
 0.37598688 0.59901008 0.58798189 0.         0.         0.
 0.         0.        ]
```

The fitted result suggest that only two groups are selected (since `support_size` is from 0 to 2) and the selected variables are shown above.

Next, we want to compare the result of a given group structure with that without a given group structure.

```
model2 = LinearRegression()
model2.fit(data.x, data.y)
print('coefficients:\n', model2.coef_)
```

```
coefficients:
 [0.61823344 0.54500673 0.59272352 0.42754021 0.65843857 0.
 0.         0.         0.         0.         0.         0.
 0.         0.66978731 0.55137187 0.         0.         0.
 0.         0.        ]
```

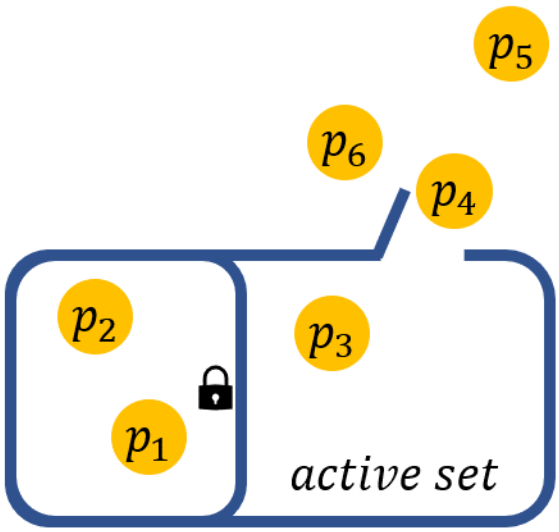
The result from a model without a given group structure omits two predictors belonging to the active set. The `abess` R package also supports best group subset selection.

For R tutorial, please view https://abess-team.github.io/abess/articles/v07-advancedFeatures.html.

sphinx_gallery_thumbnail_path = 'Tutorial/figure/best-subset-group-selection.png'

Total running time of the script: (0 minutes 0.007 seconds)

Nuisance Regression



Introduction

Nuisance regression refers to best subset selection with some prior information that some variables are required to stay in the active set. For example, if we are interested in a certain gene and want to find out what other genes are associated with the response when this particular gene shows effect. The nuisance selection can be achieved by solving:

$$\min_{(\beta,\gamma)\in\mathbb{R}^p} \frac{1}{2n}||y - X(\beta^\top, \gamma^\top)^\top||_2^2, \text{ s.t. } ||\beta||_0 \leq s.$$

Note that, the sparsity constraint restricts on β and not on γ. The effect of γ corresponds to variables that stay in the active set.

Using: nuisance regression

In the `LinearRegression()` (or other methods), the argument `always_select` is designed to realize this goal. Users can pass a list containing the indexes of the target variables to `always_select`.

Here is an example demonstrating the advantage of nuisance selection. We generate a high-dimensional dataset whose predictors are highly correlated (pairwise correlation: 0.6) and the effect of predictors on response is weaker than noise.

```
import numpy as np
from abess.datasets import make_glm_data

# generate dataset
np.random.seed(12345)
data = make_glm_data(n=100, p=500, k=3, rho=0.6, family='gaussian', snr=0.5)
print('True effective subset: ', np.nonzero(data.coef_))
```

```
True effective subset:  (array([ 87, 271, 449]),)
```

We use the standard `abess` to tackle this dataset:

```
from abess.linear import LinearRegression
model = LinearRegression(support_size=range(10))
model.fit(data.x, data.y)
print('Estimated subset:', np.nonzero(model.coef_))
```

```
Estimated subset: (array([271, 449]),)
```

The result from `model` omits the 449-th predictor belonging to the true effective set. But if we suppose that the 449-th predictor are worthy to be included in the model, we can call:

```
model = LinearRegression(support_size=range(10), always_select=[449])
model.fit(data.x, data.y)
print('Estimated subset:', np.nonzero(model.coef_))
```

```
Estimated subset: (array([271, 449]),)
```

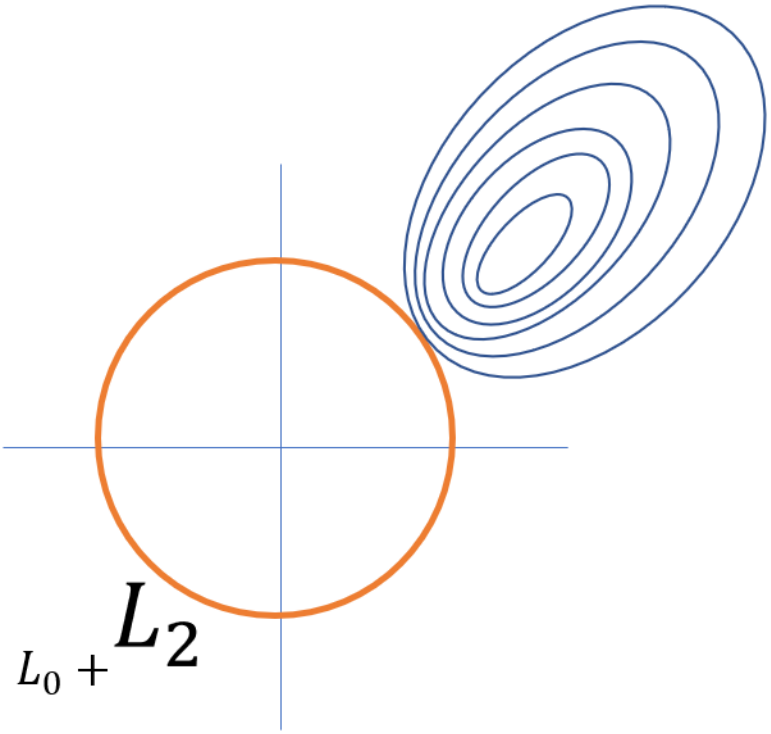
Now the estimated subset is the same as the true effective set. The comparison between nuisance selection and standard ABESS suggests that reasonably leveraging prior information promotes the quality of subset selection.

The `abess` R package also supports nuisance regression. For R tutorial, please view https://abess-team.github.io/abess/articles/v07-advancedFeatures.html.

sphinx_gallery_thumbnail_path = 'Tutorial/figure/nuisance_cover.png'

Total running time of the script: (0 minutes 0.101 seconds)

Regularized Best Subset Selection



In some cases, especially under low signal-to-noise ratio (SNR) setting or predictors are highly correlated, the vallina type of ℓ_0 constrained model may not be satisfying and a more sophisticated trade-off between bias and variance is needed. Under this concern, the `abess` package provides option of best subset selection with ℓ_2 norm regularization called the regularized best-subset selection (RBESS). The model has this following form:

$$\arg\min_{\beta} L(\beta) + \alpha\|\beta\|_2^2, \text{ s.t. } ||\beta||_0 \leq s.$$

To implement the RBESS, user need to specify a value to an additive argument `alpha` in the `LinearRegression()` function (or other methods). This value corresponds to the penalization parameter in the model above.

Let's test the RBESS against the no-regularized one over 100 replicas in terms of prediction performance. With argument `snr` in `make_glm_data()`, we can add white noise into generated data.

```
import numpy as np
from abess.datasets import make_glm_data
from abess.linear import LinearRegression
from sklearn.model_selection import train_test_split

np.random.seed(0)

loss = np.zeros((2, 100))
coef = np.repeat([1, 0], [5, 25])
for i in range(100):
```

(continues on next page)

(continued from previous page)

```
    np.random.seed(i)
    data = make_glm_data(n=200, p=30, k=5, family='gaussian', coef_=coef, snr=0.5, rho=0.
↪5)
    train_x, test_x, train_y, test_y = train_test_split(
        data.x, data.y, test_size=0.5, random_state=i)

    # normal
    model = LinearRegression()
    model.fit(train_x, train_y)
    loss[0, i] = np.linalg.norm(model.predict(test_x) - test_y)
    # regularized
    model = LinearRegression(alpha=0.1)
    model.fit(train_x, train_y)
    loss[1, i] = np.linalg.norm(model.predict(test_x) - test_y)

print("The average predition error under best-subset selection:", np.mean(loss[0, :]))
print("The average predition error under regularized best-subset selection:", np.
↪mean(loss[1, :]))
```

```
The average predition error under best-subset selection: 42.01261325454263
The average predition error under regularized best-subset selection: 41.94262361621864
```

We see that the regularized best subset select ("RABESS") indeed reduces the prediction error.

The `abess` R package also supports regularized best-subset selection. For R tutorial, please view https://abess-team.github.io/abess/articles/v07-advancedFeatures.html.

sphinx_gallery_thumbnail_path = 'Tutorial/figure/regularized_cover.png'

Total running time of the script: (0 minutes 0.460 seconds)

Cross-Validation Division

User-specified cross validation division

Sometimes, especially when running a test, we would like to fix the train and valid data used in cross validation, instead of choosing them randomly. One simple method is to fix a random seed, such as `numpy.random.seed()`. But in some cases, we would also like to specify which samples would be in the same "fold", which has great flexibility.

In our program, an additional argument `cv_fold_id` is for this user-specified cross validation division. An integer `numpy` array with the same size of input samples can be given, and those with same integer would be assigned to the same "fold" in K-fold CV.

```
import numpy as np
from abess.datasets import make_glm_data
from abess.linear import LinearRegression
n = 100
p = 1000
k = 3
np.random.seed(2)

data = make_glm_data(n=n, p=p, k=k, family='gaussian')
```

(continues on next page)

(continued from previous page)

```
# cv_fold_id has a size of `n`
# cv_fold_id has `cv` different integers
cv_fold_id = [1 for i in range(30)] + \
    [2 for i in range(30)] + [3 for i in range(40)]

model = LinearRegression(support_size=range(0, 5), cv=3)
model.fit(data.x, data.y, cv_fold_id=cv_fold_id)
print('fitted coefficients\' indexes:', np.nonzero(model.coef_)[0])
```

```
fitted coefficients' indexes: [243 295 659]
```

The `abess` R package also supports user-defined cross-validation division. For R tutorial, please view https://abess-team.github.io/abess/articles/v07-advancedFeatures.html.

Total running time of the script: (0 minutes 0.575 seconds)

Initial Active Set

User-specified initial active set

We believe that it worth allowing given an initial active set so that the splicing process starts from this set for each sparsity. It might come from prior analysis, whose result is not quite precise but better than random selection, so the algorithm can run more efficiently. Or you just want to give different initial sets to test the stability of the algorithm.

Note that this is **NOT** equivalent to `always_select`, since they can be exchanged to inactive set when splicing.

To specify initial active set, an additive argument `A_init` should be given in `fit()`.

```
import numpy as np
from abess.datasets import make_glm_data
from abess.linear import LinearRegression
n = 100
p = 10
k = 3
np.random.seed(2)

data = make_glm_data(n=n, p=p, k=k, family='gaussian')

model = LinearRegression(support_size=range(0, 5), A_init=[0, 1, 2])
model.fit(data.x, data.y)
```

Some strategies for initial active set are:

- If `sparsity = len(A_init)`, the splicing process would start from `A_init`.
- If `sparsity > len(A_init)`, the initial set includes `A_init` and other variables with larger forward sacrifices chooses.
- If `sparsity < len(A_init)`, the initial set includes part of `A_init`.
- If both `A_init` and `always_select` are given, `always_select` first.
- For warm-start, `A_init` will only affect splicing under the first sparsity in `support_size`.
- For CV, `A_init` will affect each fold but not the re-fitting on full data.

The `abess` R package also supports user-defined initial active set. For R tutorial, please view https://abess-team.github.io/abess/articles/v07-advancedFeatures.html.

Total running time of the script: (0 minutes 0.003 seconds)

Computational Tips

The generic splicing technique certifiably guarantees the best subset can be selected in a polynomial time. In practice, the computational efficiency can be improved to handle large scale datasets. The tips for computational improvement are applicable for:

1. **ultra-high dimensional data** via
    - feature screening;
    - focus on important variables;
2. **large-sample data** via
    - golden-section searching;
    - early-stop scheme;
3. **sparse inputs** via
    - sparse matrix computation;
4. **specific models** via
    - covariance update for `LinearRegression` and `MultiTaskRegression`;
    - quasi Newton iteration for `LogisticRegression`, `PoissonRegression`, `CoxRegression`, etc.

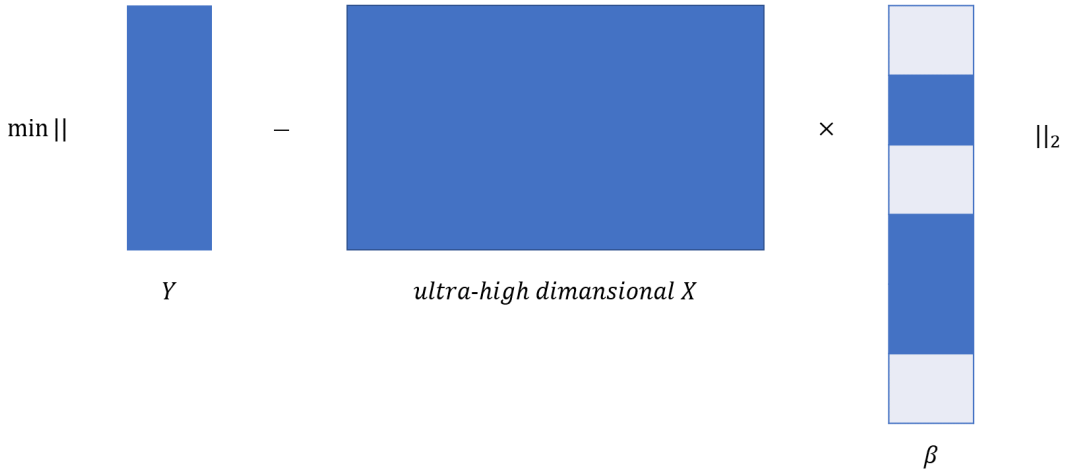
More importantly, the technique in these tips can be use simultaneously. For example, `abess` allow algorithms to use both feature screening and golden-section searching such that algorithms can handle datasets with large-sample and ultra-high dimension. The following contents illustrate the above tips.

Besides, `abess` efficiently implements warm-start initialization and parallel computing, which are very useful for fast computing. To help use leverage them, we will also describe their implementation details in the following.

Ultra-High dimensional data

Introduction

Recent technological advances have made it possible to collect ultra-high dimensional data. A common feature of these data is that the number of variables p is generally much larger than sample sizes n. For instance, the number of gene expression profiles is in the order of tens of thousands while the number of patient samples is in the order of tens or hundreds. Ultra-high dimensional predictors increase computational cost but reduce estimation accuracy for any statistical procedure. We visualize linear regression analysis in the context of ultra-high dimensionality in the following:



`abess` library implements severals features to efficiently analyze the ultra-high dimensional data with a fast speed. In this tutorial, we going to brief describe these helpful features, including: feature screening and importance searching. These features may also improve the statistical accuracy and algorithmic stability.

Feature screening

Feature screening (FS, a.k.a., sure independence screening) is one of the most famous frameworks for tackling the challenges brought by ultra-high dimensional data. The FS can theoretically maintain all effective predictors with a high probability, which is called "the sure screening property". The FS is capable of even exponentially growing dimension.

Practically, FS tries to filtering out the features that have very few marginal contribution on the loss function, hence effectively reducing the dimensionality p to a moderate scale so that performing statistical algorithm is efficient.

In our program, to carrying out the FS, user need to pass an integer smaller than the number of the predictors to the `screening_size`. Then the program will first calculate the marginal likelihood of each predictor and reserve those predictors with the `screening_size` largest marginal likelihood. Then, the ABESS algorithm is conducted only on this screened subset.

Using feature screening

Here is an example under sparse linear model with three variables have impact on the response. This dataset comprise 500 observations, and each observation has 10000 features. We use `LinearRegression` to analyze the synthetic dataset, and set `screening_size = 100` to maintain the 100 features with the largest marginal utilities.

```
from abess.linear import LogisticRegression
from time import time
import numpy as np
from abess.datasets import make_glm_data
from abess.linear import LinearRegression
```

(continues on next page)

(continued from previous page)

```
data = make_glm_data(n=500, p=10000, k=3, family='gaussian')
model = LinearRegression(support_size=range(0, 5), screening_size=100)
model.fit(data.x, data.y)
```

```
print('real coefficients\' indexes:', np.nonzero(data.coef_)[0])
print('fitted coefficients\' indexes:', np.nonzero(model.coef_)[0])
```

```
real coefficients' indexes: [7211 8688 8789]
fitted coefficients' indexes: [7211 8688 8789]
```

It can be seen that the estimated support set is identical to the true support set.

We also study the runtime when the FS is

```
model1 = LinearRegression(support_size=range(0, 20))
model2 = LinearRegression(support_size=range(0, 20), screening_size=100)
t1 = time()
model1.fit(data.x, data.y)
t2 = time()
model2.fit(data.x, data.y)
t3 = time()
print("Runtime (without screening) : ", t2 - t1)
print("Runtime (with screening) : ", t3 - t2)
```

```
Runtime (without screening) :  0.25472068786621094
Runtime (with screening) :  0.22376513481140137
```

The runtime reported above suggests the FS visibly reduce runtimes.

Not all of best subset selection methods support feature screening (e.g., RobustPCA). Please see Python API for more details.

Important searching

Suppose that there are only a few variables are important (i.e. too many noise variables), it may be a vise choice to focus on some important variables in splicing process. This can save a lot of time, especially under a large p.

In `abess` package, an argument called `important_search` is used for it, which means the size of inactive set for each splicing process. By default, this argument is set as 0, and the total inactive variables would be contained in the inactive set. But if an positive integer is given, the splicing process would focus on active set and the most important `important_search` inactive variables. After splicing iteration convergence on this subset, we check if the chosen variables are still the most important ones by recomputing on the full set with the new active set. If not, we update the subset and perform splicing again. From our empirical experience, it would not iterate many time to reach a stable subset. After that, the active set on the stable subset would be treated as that on the full set.

Using important searching

```
# Here, we use a classification task as an example to demonstrate how to use important␣
↪searching.
# This dataset comprise 200 observations, and each observation has 5000
# features.


data = make_glm_data(n=200, p=5000, k=10, family="binomial")
```

We use `LogisticRegression` but only focus on 500 most important variables. The specific code is presented below:

```
model1 = LogisticRegression(important_search=500)
t1 = time()
model1.fit(data.x, data.y)
t2 = time()
print("time : ", t2 - t1)
```

```
time :  0.17272734642028809
```

However, if we turn off the important searching (setting `important_search = 0`), and using `LogisticRegression` as usual:

```
t1 = time()
model2 = LogisticRegression(important_search=0)
model2.fit(data.x, data.y)
t2 = time()
print("time : ", t2 - t1)
```

```
time :  0.3728504180908203
```

It is easily see that the time consumption is much larger than before.

Finally, we investigate the estimated support sets given by `model1` and `model2` as follow:

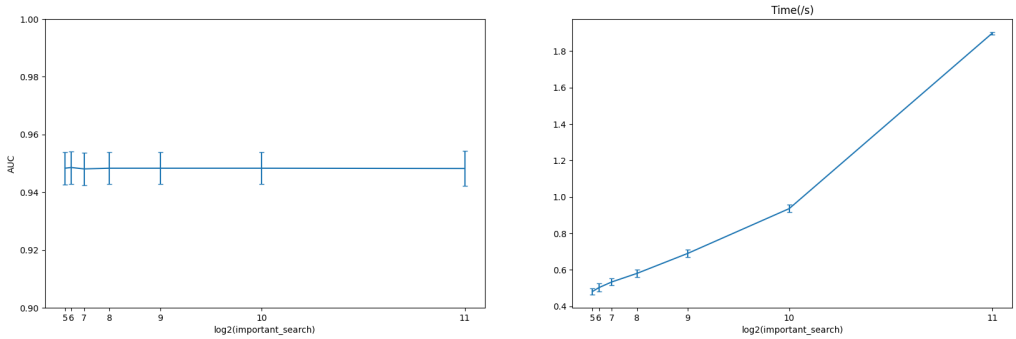
```
print("support set (with important searching):\n", np.nonzero(model1.coef_)[0])
print(
    "support set (without important searching):\n",
    np.nonzero(
        model2.coef_)[0])
```

```
support set (with important searching):
 [  30   34  452 1920 3207 4626]
support set (without important searching):
 [  30   34  452 1920 3207 4626]
```

The estimated support sets are the same. From this example, we can see that important searching uses much less time to reach the same result. Therefore, we recommend use important searching for large p situation.

Experimental evidences: important searching

Here we compare the AUC and runtime for `LogisticRegression` under different `important_search` and the test code can be found here: https://github.com/abess-team/abess/blob/master/docs/simulation/Python/plot_impsearch.py. We present the numerical results under 100 replications below.



At a low level of `important_search`, however, the performance (AUC) has been very good. In this situation, a lower `important_search` can save lots of time and space.

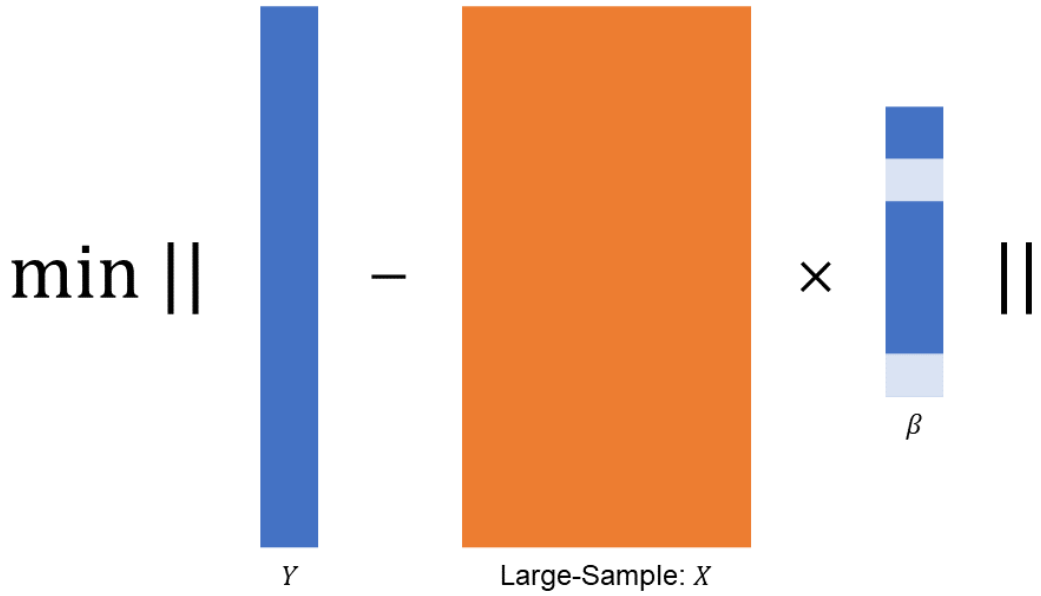
The `abess` R package also supports feature screening and important searching. For R tutorial, please view https://abess-team.github.io/abess/articles/v07-advancedFeatures.html and https://abess-team.github.io/abess/articles/v09-fasterSetting.html.

sphinx_gallery_thumbnail_path = 'Tutorial/figure/highDimension.png'

Total running time of the script: (11 minutes 16.497 seconds)

Large-Sample Data

Introduction



A large sample size leads to a large range of possible support sizes which adds to the computational burdon. The computational tip here is to use the golden-section searching to avoid support size enumeration.

A motivated observation

Here we generate a simple example under linear model via `make_glm_data`.

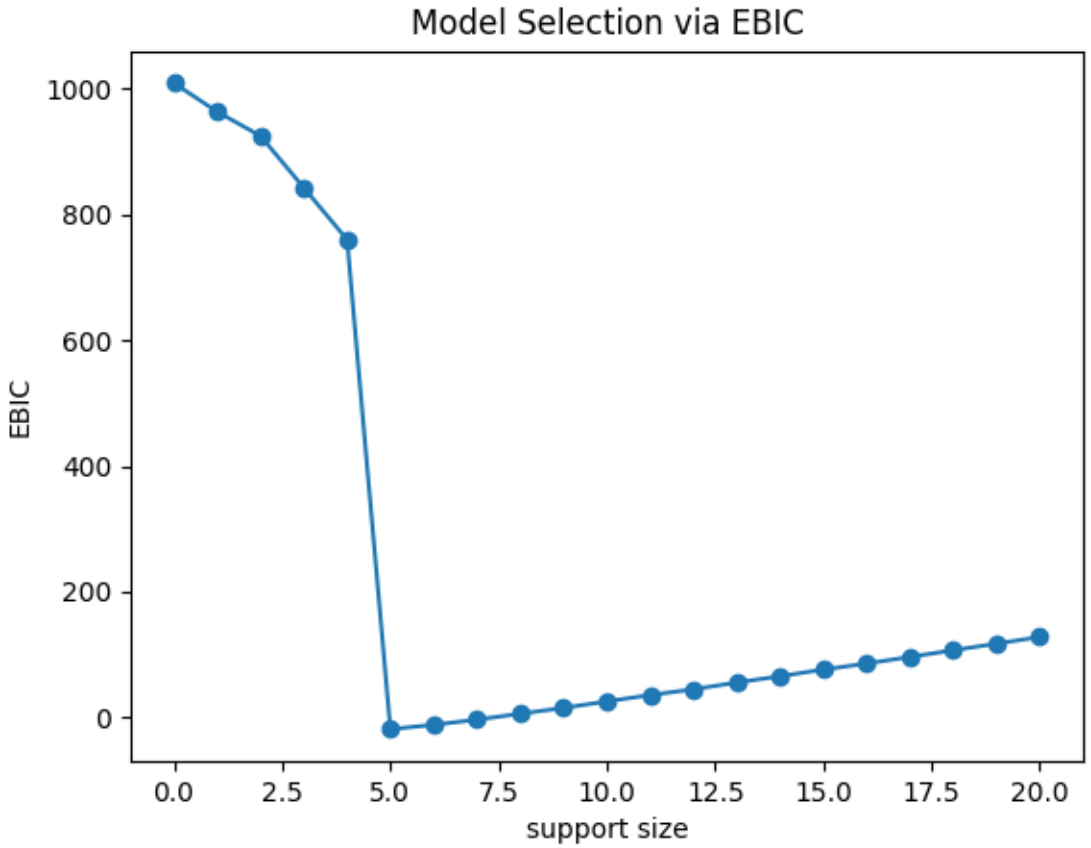
```
from time import time
import numpy as np
import matplotlib.pyplot as plt
from abess.datasets import make_glm_data
from abess.linear import LinearRegression

np.random.seed(0)
data = make_glm_data(n=100, p=20, k=5, family='gaussian')

ic = np.zeros(21)
for sz in range(21):
    model = LinearRegression(support_size=[sz], ic_type='ebic')
    model.fit(data.x, data.y)
    ic[sz] = model.eval_loss_
print("lowest point: ", np.argmin(ic))
```

```
lowest point:  5
```

The generated data contains 100 observations with 20 predictors, while 5 of them are useful (has non-zero coefficients). Uses extended Bayesian information criterion (EBIC), the `abess` successfully detect the true support size.

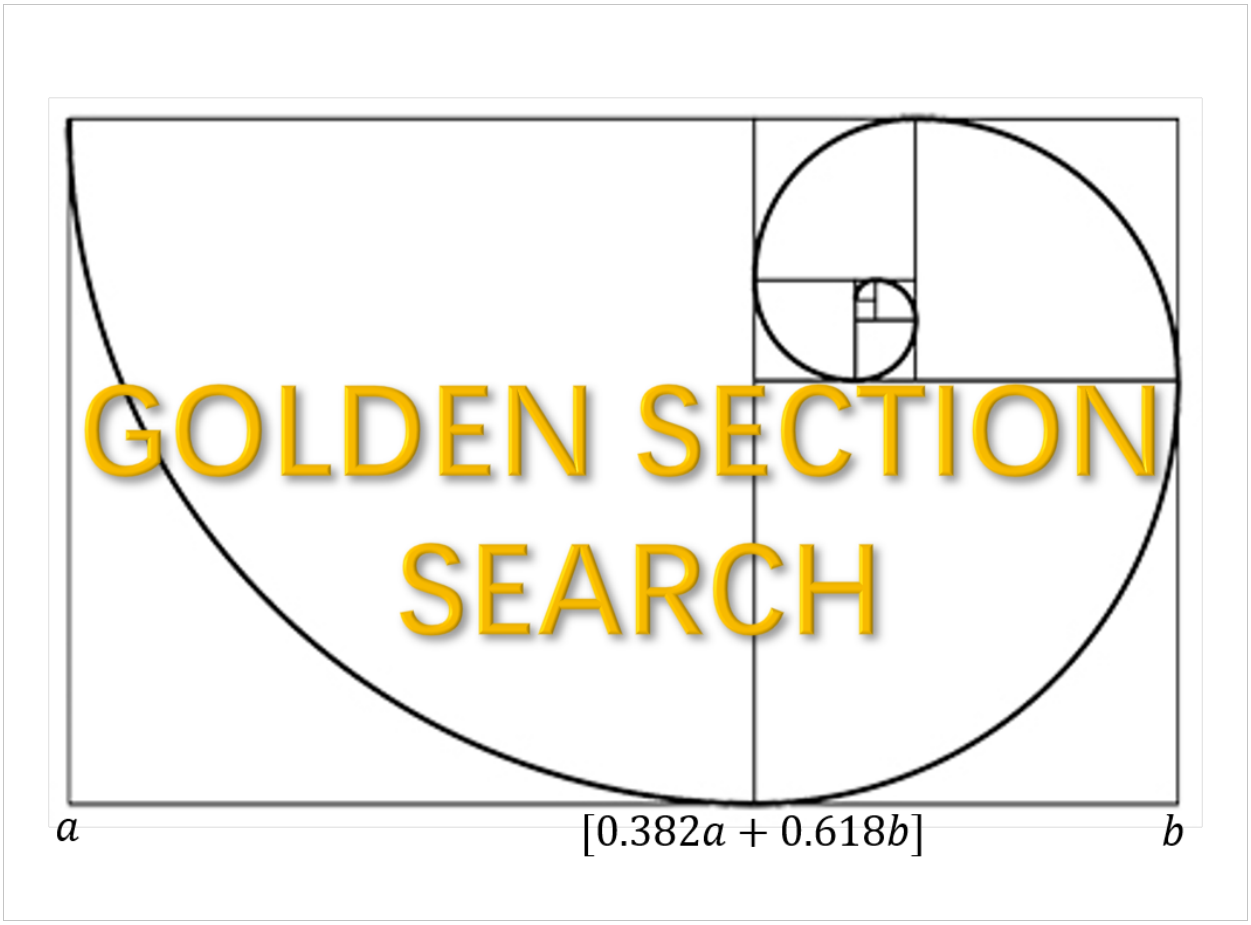
We go further and take a look on the support size versus EBIC returned by `LinearRegression` in `abess.linear`.

```
plt.plot(ic, 'o-')
plt.xlabel('support size')
plt.ylabel('EBIC')
plt.title('Model Selection via EBIC')
plt.show()
```



From the figure, we can find that the curve should is a strictly unimodal function achieving minimum at the true subset size, where `support_size = 5` is the lowest point.

Motivated by this observation, we consider a golden-section search technique to determine the optimal support size associated with the minimum EBIC.



Compare to the sequential searching, the golden section is much faster because it skip some support sizes which are likely to be a non-optimal one. Precisely, searching the optimal support size one by one from a candidate set with $O(s_{max})$ complexity, **golden-section** reduce the time complexity to $O(\ln(s_{max}))$, giving a significant computational improvement.

Usage: golden-section

In `abess` package, golden-section technique can be easily formed like:

```
model = LinearRegression(path_type='gs', s_min=0, s_max=20)
model.fit(data.x, data.y)
print("real coef:\n", np.nonzero(data.coef_)[0])
print("predicted coef:\n", np.nonzero(model.coef_)[0])
```

```
real coef:
 [ 2  5 10 11 18]
predicted coef:
 [ 2  5 10 11 18]
```

where `path_type = 'gs'` means using golden-section rather than search the support size one-by-one. `s_min` and `s_max` indicates the left and right bound of range of the support size. Note that in golden-section searching, we should not give `support_size`, which is only useful for sequential strategy.

The output of golden-section strategy suggests the optimal model size is accurately detected.

Golden-section v.s. Sequential-searching: runtime comparison

In this part, we perform a runtime comparison experiment to demonstrate the speed gain brought by golden-section.

```
t1 = time()
model = LinearRegression(support_size=range(21))
model.fit(data.x, data.y)
print("sequential time: ", time() - t1)

t2 = time()
model = LinearRegression(path_type='gs', s_min=0, s_max=20)
model.fit(data.x, data.y)
print("golden-section time: ", time() - t2)
```

```
sequential time:  0.0013496875762939453
golden-section time:  0.0008890628814697266
```

The golden-section runs much faster than sequential method. The speed gain would be enlarged when the range of support size is larger.

The `abess` R package also supports golden-section. For R tutorial, please view https://abess-team.github.io/abess/articles/v09-fasterSetting.html.

sphinx_gallery_thumbnail_path = 'Tutorial/figure/large-sample.png'

Total running time of the script: (0 minutes 0.150 seconds)

Sparse Inputs

We sometimes meet with problems where the Np input matrix X is extremely sparse, i.e., many entries in X have zero values. A notable example comes from document classification: aiming to assign classes to a document, making it easier to manage for publishers and news sites. The input variables for characterizing documents are generated from a so called "bag-of-words" model. In this model, each variable is scored for the presence of each of the words in the entire dictionary under consideration. Since most words are absent, the input variables for each document is mostly zero, and so the entire matrix is mostly zero.

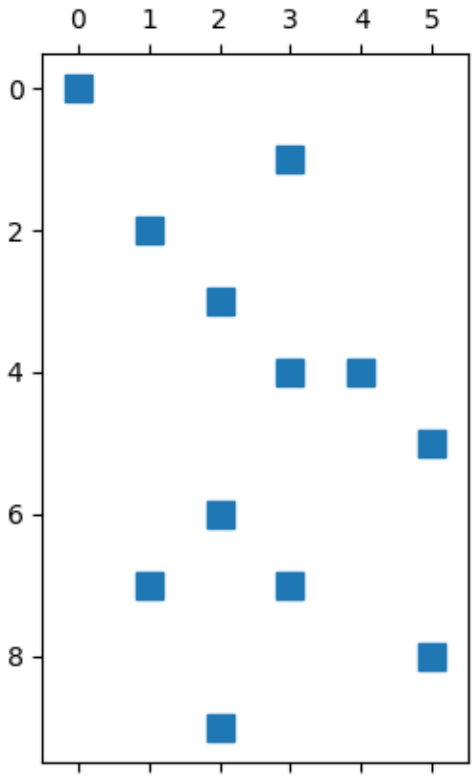
Example

We create a sparse matrix as our example:

```
from time import time
from abess import LinearRegression
from scipy.sparse import coo_matrix
import numpy as np
import matplotlib.pyplot as plt

row = np.array([0, 1, 2, 3, 4, 4, 5, 6, 7, 7, 8, 9])
col = np.array([0, 3, 1, 2, 4, 3, 5, 2, 3, 1, 5, 2])
data = np.array([4, 5, 7, 9, 1, 23, 4, 5, 6, 8, 77, 100])
x = coo_matrix((data, (row, col)))
```

And visualize the sparsity pattern via:

```
plt.spy(x)
plt.show()
```



Usage: sparse matrix

The sparse matrix can be directly used in `abess` pacakges. We just need to set argument `sparse_matrix = True`. Note that if the input matrix is not sparse matrix, the program would automatically transfer it into the sparse one, so this argument can also make some improvement.

```
coef = np.array([1, 1, 1, 0, 0, 0])
y = x.dot(coef)
model = LinearRegression()
model.fit(x, y, sparse_matrix=True)

print("real coef: \n", coef)
print("pred coef: \n", model.coef_)
```

```
real coef:
 [1 1 1 0 0 0]
pred coef:
 [1. 1. 1. 0. 0. 0.]
```

Sparse v.s. Dense: runtime comparsion

We compare the runtime when the input matrix is dense matrix:

```
t = time()
model.fit(x.toarray(), y)
print("dense matrix:  ", time() - t)

t = time()
model.fit(x, y, sparse_matrix=True)
print("sparse matrix:  ", time() - t)
```

```
dense matrix:   0.0005803108215332031
sparse matrix:   0.0009183883666992188
```

From the comparison, we see that the time required by sparse matrix is smaller, and this sould be more visible when the sparse imput matrix is large. Hence, we suggest to assign a sparse matrix to `abess` when the input matrix have a lot of zero entries.

The `abess` R package also supports sparse matrix. For R tutorial, please view https://abess-team.github.io/abess/articles/v09-fasterSetting.html

Total running time of the script: (0 minutes 0.119 seconds)

Specific Models

Introduction

From the algorithm preseneted in "ABESS algorithm: details", one of the bottleneck in algorithm is the computation of forward and backward sacrifices, which requires conducting iterative algorithms or frequently visiting p variables. To improve computational efficiency, we designed specialize strategies for computing forward and backward sacrifices for different models. The specialize strategies is roughly divide into two classes: (i) covariance update for (multivariate) linear model; (ii) quasi Newton iteration for non-linear model (e.g., logistic regression). We going to specify the two strategies as follows.

Covariance update

Under linear model, the core bottleneck is computing sacrifices, e.g. the foreward sacrifices,

$$\zeta_j = \mathcal{L}_n\left(\hat{\boldsymbol{\beta}^{\mathcal{A}}}\right) - \mathcal{L}_n\left(\hat{\boldsymbol{\beta}}^{\mathcal{A}} + \hat{t}^{\{j\}}\right) = \frac{X_j^\top X_j}{2n}\left(\frac{\hat{\boldsymbol{d}}_j}{X_j^\top X_j/n}\right)^2.$$

where $\hat{t} = \arg\min_t \mathcal{L}_n\left(\hat{\boldsymbol{\beta}}^{\mathcal{A}} + t^{\{j\}}\right), \hat{\boldsymbol{d}}_j = X_j^\top(y - X\hat{\boldsymbol{\beta}})/n$. Intuitively, for $j \in \mathcal{A}$ (or $j \in \mathcal{I}$), a large ξ_j (or ζ_j) implies the j th variable is potentially important.

It would take a lot of time on calculating $X_j^T y$, $X_j^T X_j$ and its inverse. To speed up, it is actually no need to recompute these items at each splicing process. Instead, they can be stored when first calculated, which is what we call "covariance update".

It is easy to enable this feature with an additional argument `covariance_update=True` for linear model, for example:

```
import numpy as np
from time import time
from abess.linear import LinearRegression
from abess.datasets import make_glm_data

np.random.seed(1)
data = make_glm_data(n=10000, p=100, k=10, family='gaussian')
model1 = LinearRegression()
model2 = LinearRegression(covariance_update=True)

t1 = time()
model1.fit(data.x, data.y)
t1 = time() - t1

t2 = time()
model2.fit(data.x, data.y)
t2 = time() - t2

print(f"No covariance update: {t1}")
print(f"Covariance update: {t2}")
print(f"Same answer? {(model1.coef_==model2.coef_).all()}")
```

```
No covariance update: 2.10817289352417
Covariance update: 1.9920933246612549
Same answer? True
```

We can see that covariance update improve computation when sample size n is much larger than dimension p.

However, we have to point out that covariance update will cause higher memory usage, especially when p is large. So, we recommend to enable covariance update for fast computation when sample size is much larger than dimension and dimension is moderate ($p \leq 2000$).

Quasi Newton iteration

In the third step in Algorithm 2 , we need to solve a convex optimization problem:

$$\tilde{\beta} = \arg \min_{\text{supp}(\beta)=\tilde{\mathcal{A}}} l_n(\beta).$$

But generally, it has no closed-form solution, and has to be solved via iterative algorithm. A natural method for solving this problem is Netwon method, i.e., conduct the update:

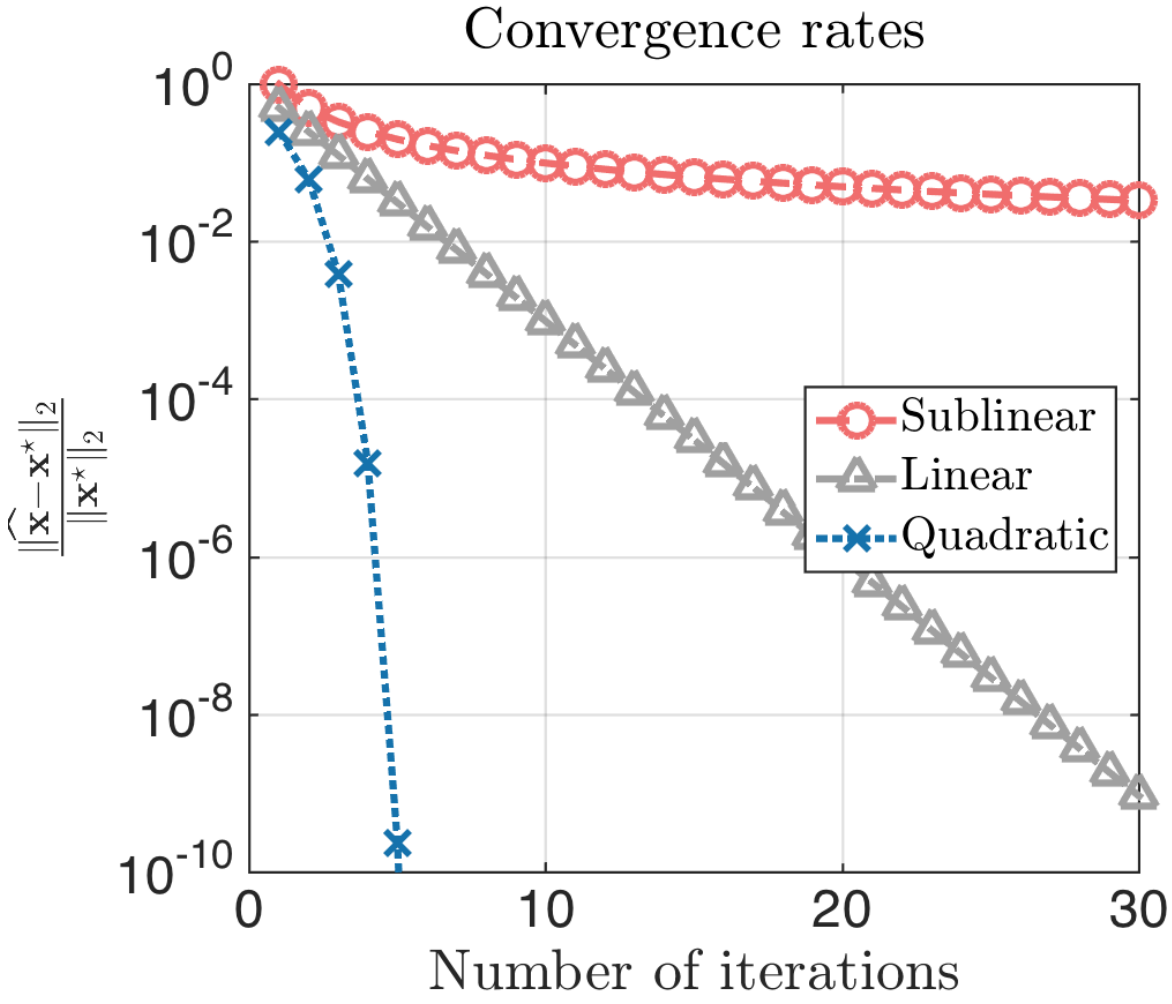
$$\beta_{\tilde{\mathcal{A}}}^{m+1} \leftarrow \boldsymbol{\beta}_{\tilde{\mathcal{A}}}^{m} - \Big(\frac{\partial^2 l_n(\boldsymbol{\beta})}{(\partial \boldsymbol{\beta}_{\tilde{\mathcal{A}}})^2}\Big|_{\boldsymbol{\beta}=\boldsymbol{\beta}^m} \Big)^{-1} \Big(\frac{\partial l_n(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{\tilde{\mathcal{A}}}}\Big|_{\boldsymbol{\beta}=\boldsymbol{\beta}^m} \Big),$$

until $\|\beta_{\tilde{\mathcal{A}}}^{m+1} - \beta_{\tilde{\mathcal{A}}}^{m}\|_2 \leq \epsilon$ or $m \geq k$, where ϵ, k are two user-specific parameters. Generally, setting $\epsilon = 10^{-6}$ and $k = 80$ achieves desirable estimation. Generally, the inverse of second derivative is computationally intensive, and thus, we approximate it with its diagonalized version. Then, the update formulate changes to:

$$\beta_{\tilde{\mathcal{A}}}^{m+1} \leftarrow \boldsymbol{\beta}_{\tilde{\mathcal{A}}}^{m} - \rho D \Big(\frac{\partial l_n(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{\tilde{\mathcal{A}}}}\Big|_{\boldsymbol{\beta}=\boldsymbol{\beta}^m} \Big),$$

where $D = \text{diag}((\frac{\partial^2 l_n(\boldsymbol{\beta})}{(\partial \boldsymbol{\beta}_{\tilde{\mathcal{A}}_1})^2}\Big|_{\boldsymbol{\beta}=\boldsymbol{\beta}^m})^{-1}, \ldots, (\frac{\partial^2 l_n(\boldsymbol{\beta})}{(\partial \boldsymbol{\beta}_{\tilde{\mathcal{A}}_{|A|}})^2}\Big|_{\boldsymbol{\beta}=\boldsymbol{\beta}^m})^{-1})$ and ρ is step size. Although using the approximation may increase the iteration time, it avoids a large computational complexity when computing the ma-

trix inversion. Furthermore, we use a heuristic strategy to reduce the iteration time. Observing that not every new support after exchanging the elements in active set and inactive set may not reduce the loss function, we can early stop the newton iteration on these support. Specifically, support $l_1 = L(\beta^m), l_2 = L(\beta^{m+1})$, if $l_1 - (k - m - 1) \times (l_2 - l_1)) > L - \tau$, then we can expect the new support cannot lead to a better loss after k iteration, and hence, it is no need to conduct the remaining $k - m - 1$ times Newton update. This heuristic strategy is motivated by the convergence rate of Netwon method is linear at least.



To enable this feature, you can simply give an additional argument `approximate_Newton=True`. The ϵ and k we mentioned before, can be set with `primary_model_fit_epsilon` and `primary_model_fit_max_iter`, respectively. For example:

```
import numpy as np
from time import time
from abess.linear import LogisticRegression
from abess.datasets import make_glm_data

np.random.seed(1)
data = make_glm_data(n=1000, p=100, k=10, family='binomial')
model1 = LogisticRegression()
model2 = LogisticRegression(approximate_Newton=True,
                            primary_model_fit_epsilon=1e-6,
                            primary_model_fit_max_iter=10)
```

(continues on next page)

(continued from previous page)

```
t1 = time()
model1.fit(data.x, data.y)
t1 = time() - t1

t2 = time()
model2.fit(data.x, data.y)
t2 = time() - t2

print(f"No newton: {t1}")
print(f"Newton: {t2}")
print(f"Same answer? {(np.nonzero(model1.coef_)[0]==np.nonzero(model2.coef_)[0]).all()}")
```

```
No newton: 6.120475769042969
Newton: 1.9317810535430908
Same answer? True
```

The abess R package also supports covariance update and quasi Newton iteration. For R tutorial, please view https://abess-team.github.io/abess/articles/v09-fasterSetting.html

Total running time of the script: (0 minutes 12.196 seconds)

Connect to Popular Libraries with scikit-learn API

This part is intended to present all possible connection between abess and other popular Python libraries via the scikit-learn interface. It is keep developing and more examples will come up soon. Contributions for this part is extremely welcome!

Work with scikit-learn

abess is very easy to work with the famous package scikit-learn, and here is an example. We going to illustrate the integration of the abess with scikit-learn's pre-processing and model selection modules to build a non-linear model for diagnosing malignant tumors. Let start with importing necessary dependencies:

```
from abess.linear import LogisticRegression
from sklearn.datasets import load_breast_cancer
from sklearn.pipeline import Pipeline
from sklearn.metrics import roc_auc_score, make_scorer, roc_curve, auc
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import GridSearchCV
```

Establish the process

Suppose we would like to extend the original variables to their interactions, and then do `LogisticRegression` on them. This can be record with `Pipeline`:

```
pipe = Pipeline([
    ('poly', PolynomialFeatures(include_bias=False)),   # without intercept
    ('alogistic', LogisticRegression())
])
```

Parameter grid

We can give different parameters to model and let the program choose the best. Here we should give parameters for `PolynomialFeatures`, for example:

```
param_grid = {
    # whether the "self-combination" (e.g. X^2, X^3) exists
    'poly__interaction_only': [True, False],
    'poly__degree': [1, 2, 3]                   # the degree of polynomial
}
```

Note that the program would try all combinations of what we give, which means that there are $2 \times 3 = 6$ combinations of parameters will be tried.

Criterion

After giving a grid of parameters, we should define what is a "better" result. For example, the AUC (area under ROC curve) can be a criterion and the larger, the better.

```
scorer = make_scorer(roc_auc_score, greater_is_better=True)
```

Cross Validation

For more accurate results, cross validation (CV) is often formed. In this example, we use 5-fold CV for parameters searching:

```
grid_search = GridSearchCV(pipe, param_grid, scoring=scorer, cv=5)
```

Model fitting

Eveything is prepared now. We can simply load the data and put it into `grid_search`:

```
X, y = load_breast_cancer(return_X_y=True)
grid_search.fit(X, y)
print([grid_search.best_score_, grid_search.best_params_])
```

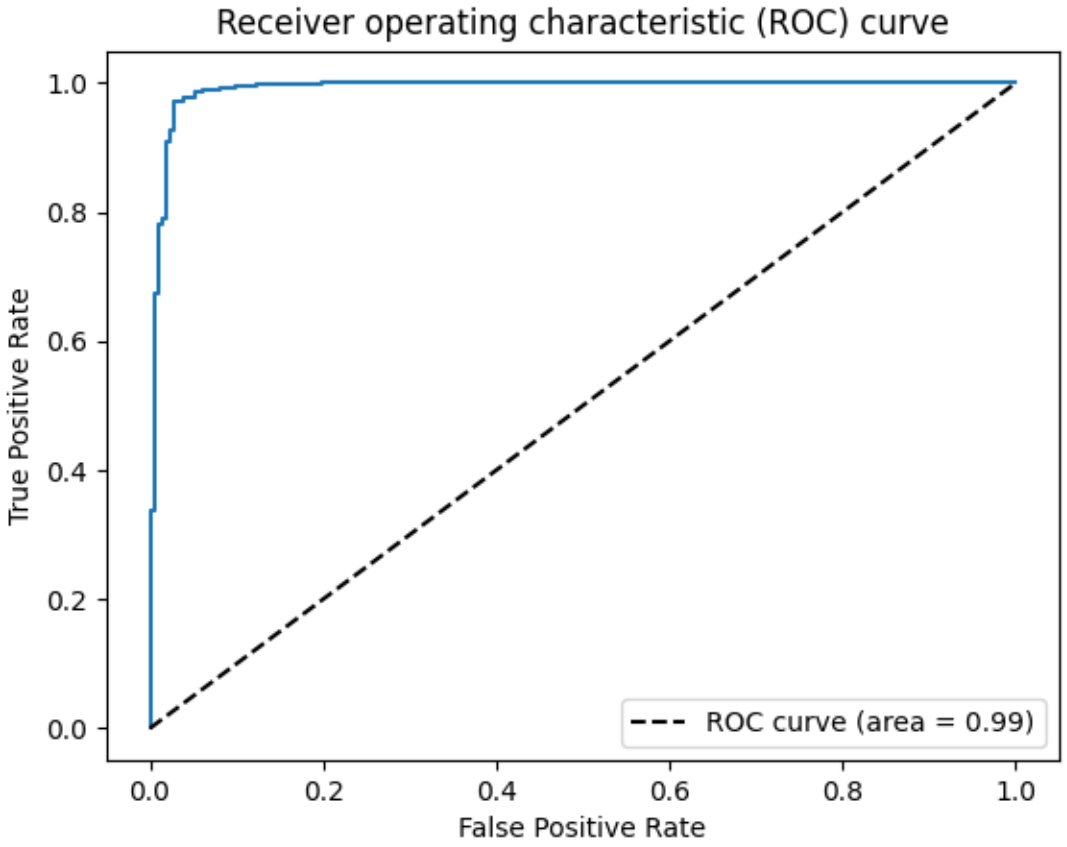
```
[0.9714645755670978, {'poly__degree': 2, 'poly__interaction_only': True}]
```

The output of the code reports the information of the polynomial features for the selected model among candidates, and its corresponding area under the curve (AUC), which is 0.99, indicating the selected model would have an admirable contribution in practice.

Moreover, the best choice of parameter combination is shown above: 2 degree with "self-combination", implying the inclusion of the pairwise interactions between any two features can lead to a better model generalization.

Here is its ROC curve:

```
import matplotlib.pyplot as plt
proba = grid_search.predict_proba(X)
fpr, tpr, _ = roc_curve(y, proba[:, 1])
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], 'k--', label="ROC curve (area = %0.2f)" % auc(fpr, tpr))
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("Receiver operating characteristic (ROC) curve")
plt.legend(loc="lower right")
plt.show()
```



sphinx_gallery_thumbnail_path = 'Tutorial/figure/scikit_learn.png'

Total running time of the script: (0 minutes 10.746 seconds)

Work with geomstats

The package *geomstats* is used for computations and statistics on nonlinear manifolds, such as Hypersphere,Hyperbolic Space, Symmetric-Positive-Definite (SPD) Matrices Space and Skew-Symmetric Matrices Space. *abess* also works well with the package *geomstats*. Here is an example of using *abess* to do logistic regression of samples on Hypersphere, and we will compare the precision score, the recall score and the running time with *abess* and with *scikit-learn*.

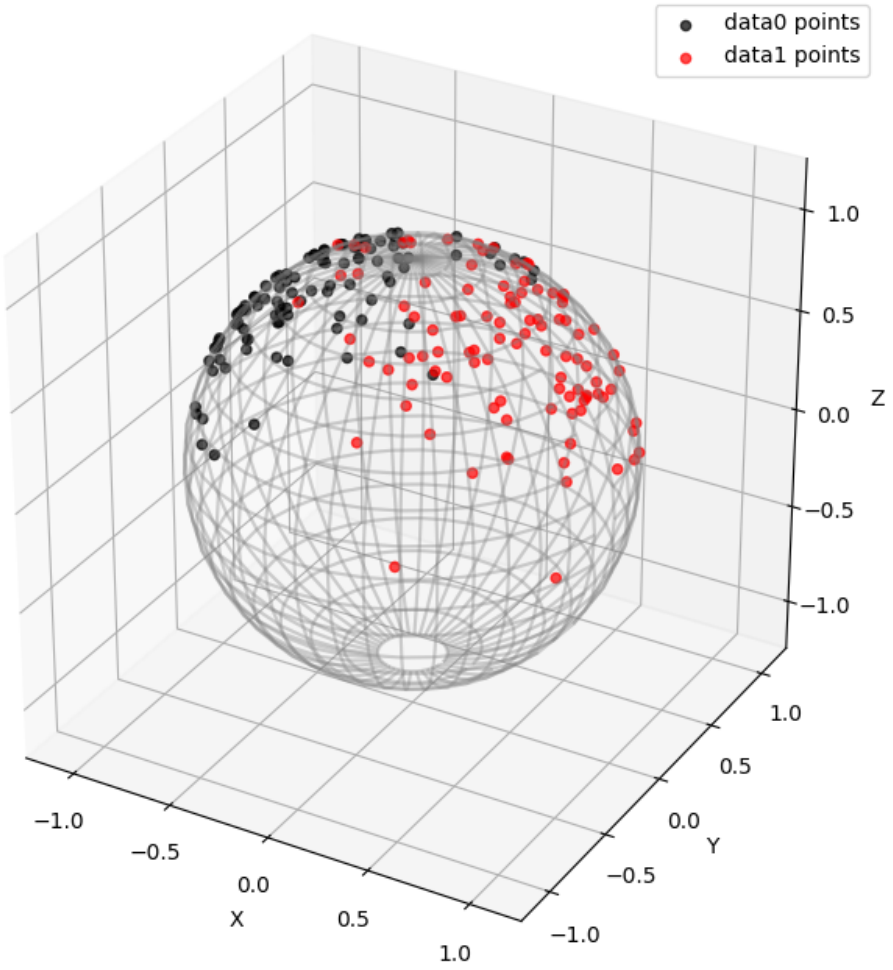
```
import numpy as np
import matplotlib.pyplot as plt
import geomstats.backend as gs
import geomstats.visualization as visualization
from geomstats.learning.frechet_mean import FrechetMean
from geomstats.geometry.hypersphere import Hypersphere
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score
from sklearn.linear_model import LogisticRegression as sklLogisticRegression
from abess import LogisticRegression
import time
import warnings
warnings.filterwarnings("ignore")
gs.random.seed(0)
```

```
INFO: Using numpy backend
```

An Example

Two sets of samples on Hypersphere in 3-dimensional Euclidean Space are created. The sample points in *data0* are distributed around $[-3/5, 0, 4/5]$, and the sample points in *data1* are distributed around $[3/5, 0, 4/5]$. The sample size of both is set to 100, and the precision of both is set to 5. The two sets of samples are shown in the figure below.

```
sphere = Hypersphere(dim=2)
data0 = sphere.random_riemannian_normal(mean=np.array([-3/5, 0, 4/5]), n_samples=100,␣
↪precision=5)
data1 = sphere.random_riemannian_normal(mean=np.array([3/5, 0, 4/5]), n_samples=100,␣
↪precision=5)

fig = plt.figure(figsize=(8, 8))
ax = visualization.plot(data0, space="S2", color="black", alpha=0.7, label="data0 points
↪")
ax = visualization.plot(data1, space="S2", color="red", alpha=0.7, label="data1 points")
ax.set_box_aspect([1, 1, 1])
ax.legend()
plt.show()
```



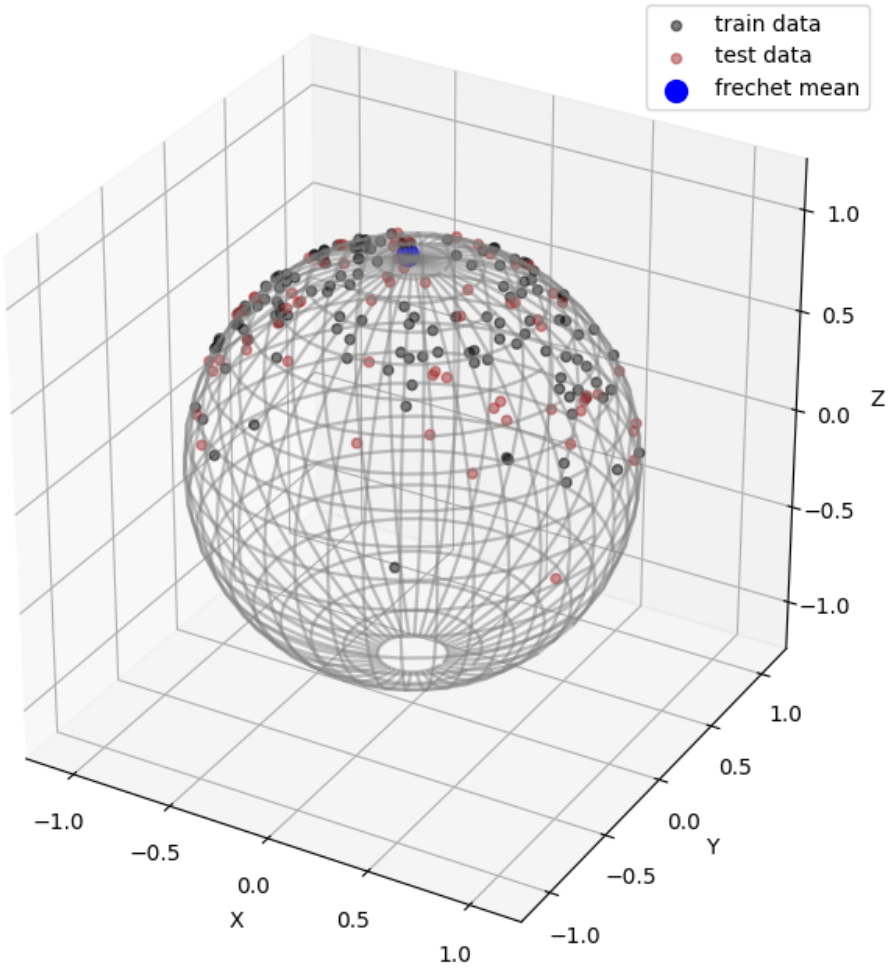
Then, we divide the data into *train_data* and *test_data*, and calculate the frechit mean of *train_data*, which has the minimum sum of the squares of the distances along the geodesic to each sample point in *train_data*. The *test_data*,the *train_data* and the frechit mean are shown in the figure below.

```
labels = np.concatenate((np.zeros(data0.shape[0]),np.ones(data1.shape[0])))
data = np.concatenate((data0,data1))
train_data, test_data, train_labels, test_labels = train_test_split(data, labels, test_
↪size=0.33, random_state=0)

mean = FrechetMean(metric=sphere.metric)
mean.fit(train_data)
mean_estimate = mean.estimate_

fig = plt.figure(figsize=(8, 8))
```

(continues on next page)

(continued from previous page)

```
ax = visualization.plot(train_data, space="S2", color="black", alpha=0.5, label="train␣
↪data")
ax = visualization.plot(test_data, space="S2", color="brown", alpha=0.5, label="test data
↪")
ax = visualization.plot(mean_estimate, space="S2", color="blue", s=100, label="frechet␣
↪mean")
ax.set_box_aspect([1, 1, 1])
ax.legend()
plt.show()
```

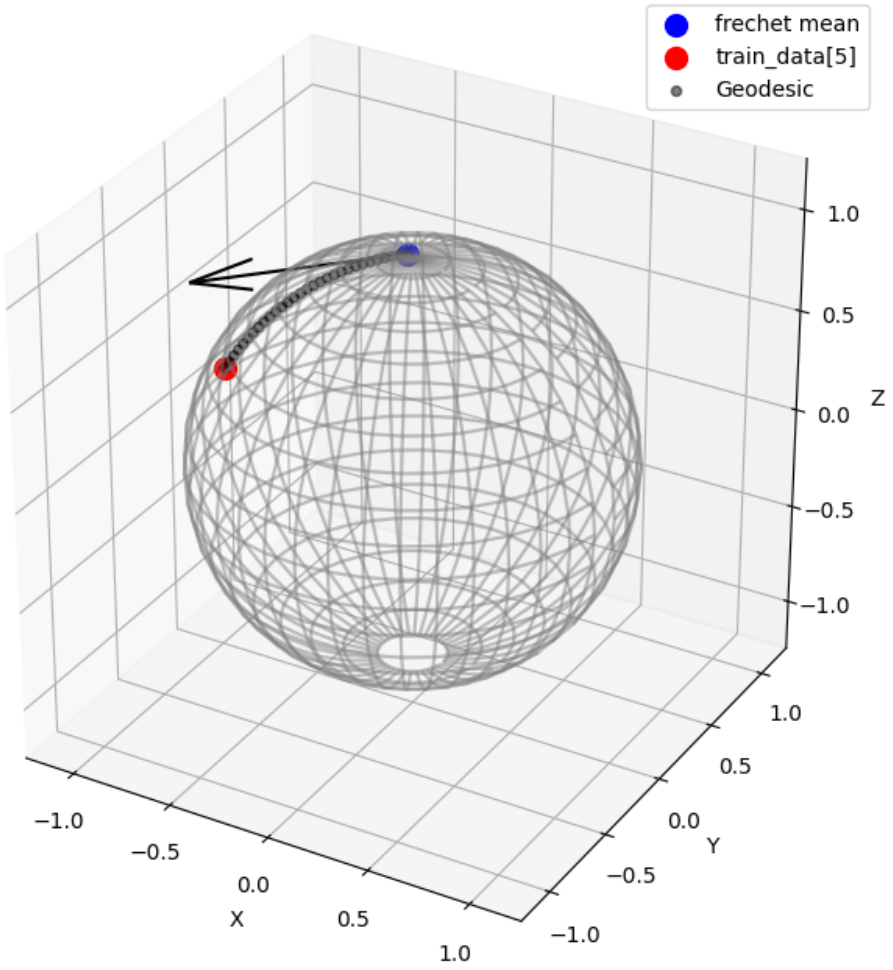


Next, do the logarithm map for all sample points from the frechit mean. That is, map each sample point to which point on the tangential of the geodesic (from the frechit mean to the sample point) at the frechit mean and has the distance to the frechit that equals to the length of the geodesic.

```
log_train_data = sphere.metric.log(train_data, mean_estimate)
log_test_data = sphere.metric.log(test_data, mean_estimate)
```

The following figure shows the logarithm mapping of *train_data[5]* from the frechit mean.

```
geodesic = sphere.metric.geodesic(mean_estimate, end_point=train_data[5])
points_on_geodesic = geodesic(gs.linspace(0.0, 1.0, 30))

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection="3d")
ax = visualization.plot(mean_estimate, space="S2", color="blue", s=100, label="frechet␣
↪mean")
ax = visualization.plot(train_data[5], space="S2", color="red", s=100, label="train_
↪data[5]")
ax = visualization.plot(points_on_geodesic, ax=ax, space="S2", color="black", alpha=0.5,␣
↪label="Geodesic")
arrow = visualization.Arrow3D(mean_estimate, vector=log_train_data[5])
arrow.draw(ax, color="black")
ax.legend();
plt.show()
```



After that, the samples are naturally distributed on a linear area. Then, some common analysis methods can be used to analyze this set of data, such as LogisticRegression from *abess*.

```
model = LogisticRegression(support_size= range(0,4))
model.fit(log_train_data, train_labels)
fitted_labels = model.predict(log_test_data)

print('Used variables\' index:', np.nonzero(model.coef_ != 0)[0])
print('accuracy:',sum((fitted_labels - test_labels + 1) % 2)/test_data.shape[0])
```

```
Used variables' index: [0]
accuracy: 0.9090909090909091
```

The result shows that the only variables' index it used is $[0]$. When constructing the samples, the means of the

two sets are only different in the 0th direction. It shows that *abess* correctly identifies the most relevant variable for classification.

Comparison

Here is the comparison of the precision score and the recall score with *abess* and *scikit-learn*, and the comparison of the running time with *abess* and *scikit-learn*.

We loop 50 times. At each time, two sets of samples on Hypersphere in 10-dimensional Euclidean Space are created. The sample points in *data0* are distributed around $[1 / 3, 0, 2 / 3, 0, 2 / 3, 0, 0, 0, 0, 0]$, and the sample points in *data1* are distributed around $[0, 0, 2 / 3, 0, 2 / 3, 0, 0, 0, 0, 1 / 3]$. The sample size of both is set to 200, and the precision of both is set to 5.

```
m = 50  # cycles
n_sam = 200
s = 10
pre = 5

sphere = Hypersphere(dim=s - 1)
labels = np.concatenate((np.zeros(n_sam), np.ones(n_sam)))
abess_precision_score = np.zeros(m)
skl_precision_score = np.zeros(m)
abess_recall_score = np.zeros(m)
skl_recall_score = np.zeros(m)
abess_geo_time = np.zeros(m)
skl_geo_time = np.zeros(m)

for i in range(m):
    data0 = sphere.random_riemannian_normal(mean=np.array([1 / 3, 0, 2 / 3, 0, 2 / 3, 0,␣
↪0, 0, 0, 0]), n_samples=n_sam,
                                            precision=pre)
    data1 = sphere.random_riemannian_normal(mean=np.array([0, 0, 2 / 3, 0, 2 / 3, 0, 0,␣
↪0, 0, 1 / 3]), n_samples=n_sam,
                                            precision=pre)
    data = np.concatenate((data0, data1))
    train_data, test_data, train_labels, test_labels = train_test_split(data, labels,␣
↪test_size=0.33, random_state=0)
    mean = FrechetMean(metric=sphere.metric)
    mean.fit(train_data)
    mean_estimate = mean.estimate_
    log_train_data = sphere.metric.log(train_data, mean_estimate)
    log_test_data = sphere.metric.log(test_data, mean_estimate)

    start = time.time()
    abess_geo_model = LogisticRegression(support_size=range(0, s + 1)).fit(log_train_
↪data, train_labels)
    abess_geo_fitted_labels = abess_geo_model.predict(log_test_data)
    end = time.time()
    abess_geo_time[i] = end - start
    abess_precision_score[i] = precision_score(test_labels, abess_geo_fitted_labels,␣
↪average='micro')
    abess_recall_score[i] = recall_score(test_labels, abess_geo_fitted_labels, average=
↪'micro')
```

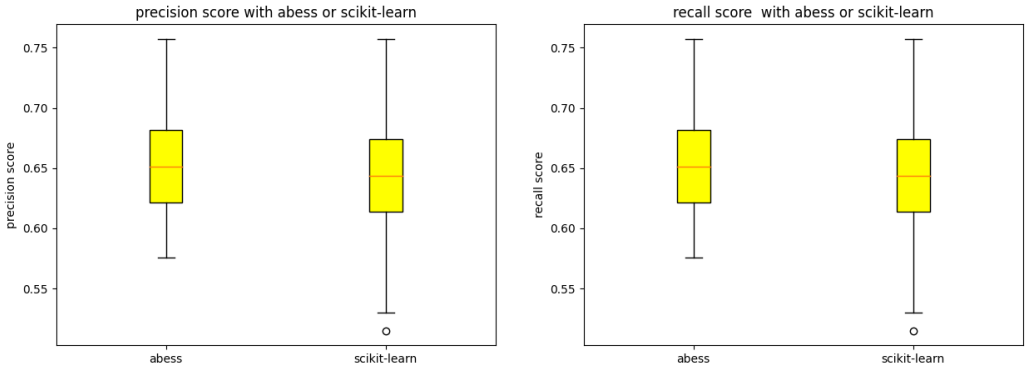
(continues on next page)

(continued from previous page)

```
    start = time.time()
    skl_geo_model = sklLogisticRegression().fit(X=log_train_data, y=train_labels)
    skl_geo_fitted_labels = skl_geo_model.predict(log_test_data)
    end = time.time()
    skl_geo_time[i] = end - start
    skl_precision_score[i] = precision_score(test_labels, skl_geo_fitted_labels, average=
↪'micro')
    skl_recall_score[i] = recall_score(test_labels, skl_geo_fitted_labels, average='micro
↪')
```

The following figures show the precision score and the recall score with *abess* or *scikit-learn*.

```
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(121)
ax1.boxplot([abess_precision_score, skl_precision_score],
            patch_artist='Patch',
            labels = ['abess', 'scikit-learn'],
            boxprops = {'color':'black','facecolor':'yellow'}

           )
ax1.set_title('precision score with abess or scikit-learn')
ax1.set_ylabel('precision score')

ax2 = fig.add_subplot(122)
ax2.boxplot([abess_recall_score, skl_recall_score],
            patch_artist='Patch',
            labels = ['abess', 'scikit-learn'],
            boxprops = {'color':'black','facecolor':'yellow'}

           )
ax2.set_title('recall score  with abess or scikit-learn')
ax2.set_ylabel('recall score')
plt.show()
```



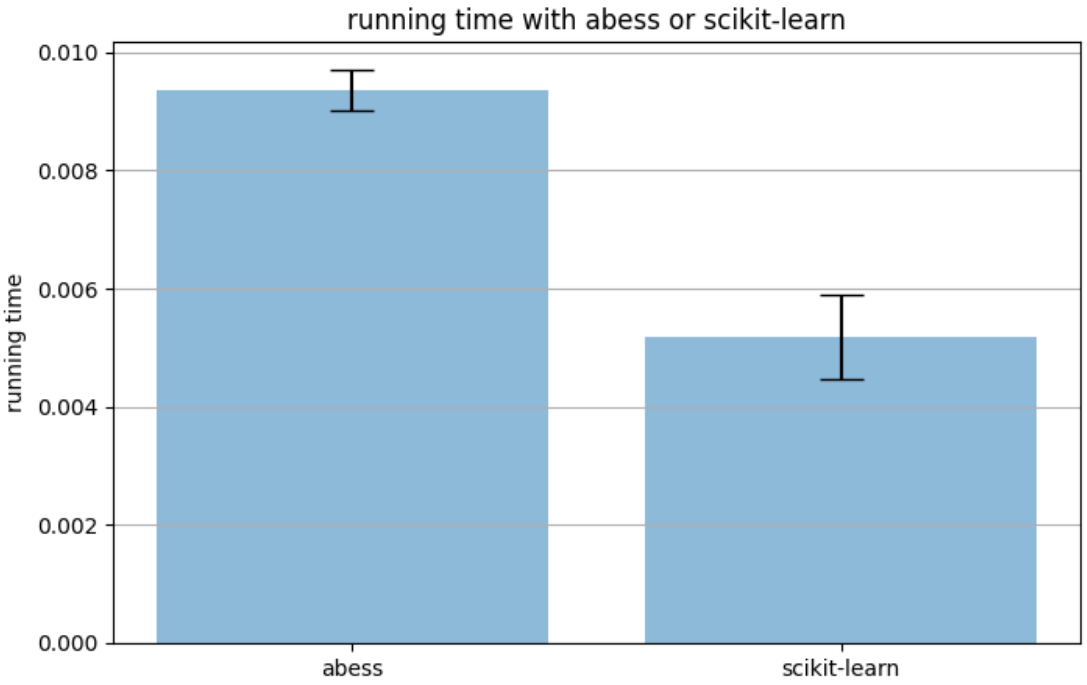
The following figure shows the running time with *abess* or *scikit-learn*.

```
abess_geo_time_mean = np.mean(abess_geo_time)
skl_geo_time_mean = np.mean(skl_geo_time)
```

(continues on next page)

(continued from previous page)

```
abess_geo_time_std = np.std(abess_geo_time)
skl_geo_time_std = np.std(skl_geo_time)
meth = ['abess', 'scikit-learn']
x_pos = np.arange(len(meth))
CTEs = [abess_geo_time_mean, skl_geo_time_mean]
error = [abess_geo_time_std, skl_geo_time_std]

fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111)
ax.bar(x_pos, CTEs, yerr=error, align='center', alpha=0.5, ecolor='black', capsize=10)
ax.set_ylabel('running time')
ax.set_xticks(x_pos)
ax.set_xticklabels(meth)
ax.set_title('running time with abess or scikit-learn')
ax.yaxis.grid(True)
plt.show()
```



We can find that the precision score and the recall score with *abess* are generally higher than those without *abess*. And the running time with *abess* is only slightly slower than that without *abess*.

sphinx_gallery_thumbnail_path = 'Tutorial/figure/geomstats.png'

Total running time of the script: (0 minutes 2.881 seconds)

Work with DoubleML

Double machine learning[1] offer a debiased way for estimating low-dimensional parameter of interest in the presence of high-dimensional nuisance. Many machine learning methods can be used to estimate the nuisance parameters, such as random forests, lasso or post-lasso, neural nets, boosted regression trees, and so on. The Python package `DoubleML`[2] provide an implementation of the double machine learning. It's built on top of scikit-learn and is an excellent package. The object-oriented implementation of `DoubleML` is very flexible, in particular functionalities to estimate double machine learning models and to perform statistical inference via the methods fit, bootstrap, confint, p_adjust and tune.

In fact, `abess`[3] also works well with the package `DoubleML`. Here is an example of using `abess` to solve such a problem, and we will compare it to the lasso regression.

```
import numpy as np
from sklearn.base import clone
from sklearn.linear_model import LassoCV
from abess.linear import LinearRegression
from doubleml import DoubleMLPLR
import matplotlib.pyplot as plt
import warnings             # ignore warnings
warnings.filterwarnings('ignore')
import time
```

Partially linear regression (PLR) model

PLR models take the form

$$Y = D\theta_0 + g_0(X) + U, \quad \mathbb{E}(U \mid X, D) = 0,$$
$$D = m_0(X) + V, \quad \mathbb{E}(V \mid X) = 0,$$

where Y is the outcome variable, D is the policy/treatment variable. θ_0 is the main regression coefficient that we would like to infer, which has the interpretation of the treatment effect parameter. The high-dimensional vector $X = (X_1, \ldots, X_p)$ consists of other confounding covariates, and U and V are stochastic errors. Usually, p is not vanishingly small relative to the sample size, it's difficult to estimate the nuisance parameters $\eta_0 = (m_0, g_0)$. `abess` aims to solve general best subset selection problem. In PLR models, `abess` is applicable when nuisance parameters are sparse. Here, we are going to use `abess` to estimate the nuisance parameters, then combine with `DoubleML` to estimate the treatment effect parameter.

Data

We simulate the data from a PLR model, which both m_0 and g_0 are low-dimensional linear combinations of X, and we save the data as `DoubleMLData` class.

```
from doubleml import DoubleMLData
np.random.seed(1234)
n_obs = 200
n_vars = 600
```

(continues on next page)

[1] Chernozhukov V, Chetverikov D, Demirer M, et al. Double/debiased machine learning for treatment and structural parameters[M]. Oxford University Press Oxford, UK, 2018.

[2] Bach P, Chernozhukov V, Kurz M S, et al. Doubleml-an object-oriented implementation of double machine learning in python[J]. Journal of Machine Learning Research, 2022, 23(53): 1-6.

[3] Zhu J, Hu L, Huang J, et al. abess: A fast best subset selection library in python and r[J]. arXiv preprint arXiv:2110.09697, 2021.

(continued from previous page)

```
theta = 3
X = np.random.normal(size=(n_obs, n_vars))
d = np.dot(X[:, :3], np.array([5]*3)) + np.random.standard_normal(size=(n_obs,))
y = theta * d + np.dot(X[:, :3], np.array([5]*3)) + np.random.standard_normal(size=(n_
↪obs,))
dml_data_sim = DoubleMLData.from_arrays(X, y, d)
```

Model fitting with abess

Based on the simulated data, now we are going to illustrate how to integrate the `abess` with `DoubleML`. To estimate the PLR model with the double machine learning algorithm, first we need to choose a learner to estimate the nuisance parameters $\eta_0 = (m_0, g_0)$. Considering the sparsity of the data, we can use the adaptive best subset selection model. Then fitting the model to learn the average treatment effct parameter θ_0.

```
abess = LinearRegression(cv = 5)       # abess learner
ml_g_abess = clone(abess)
ml_m_abess = clone(abess)

obj_dml_plr_abess = DoubleMLPLR(dml_data_sim, ml_g_abess, ml_m_abess)    # model fitting
obj_dml_plr_abess.fit();
print("thetahat:", obj_dml_plr_abess.coef)
print("sd:", obj_dml_plr_abess.se)
```

```
thetahat: [3.03421218]
sd: [0.07167669]
```

The estimated value is close to the true parameter, and the standard error is very small. `abess` integrates with `DoubleML` easily, and works well for estimating the nuisance parameter.

Comparison with lasso

The lasso regression is a shrinkage and variable selection method for regression models, which can also be used in high-dimensional setting. Here, we compare the abess regression with the lasso regression at different variable dimensions.

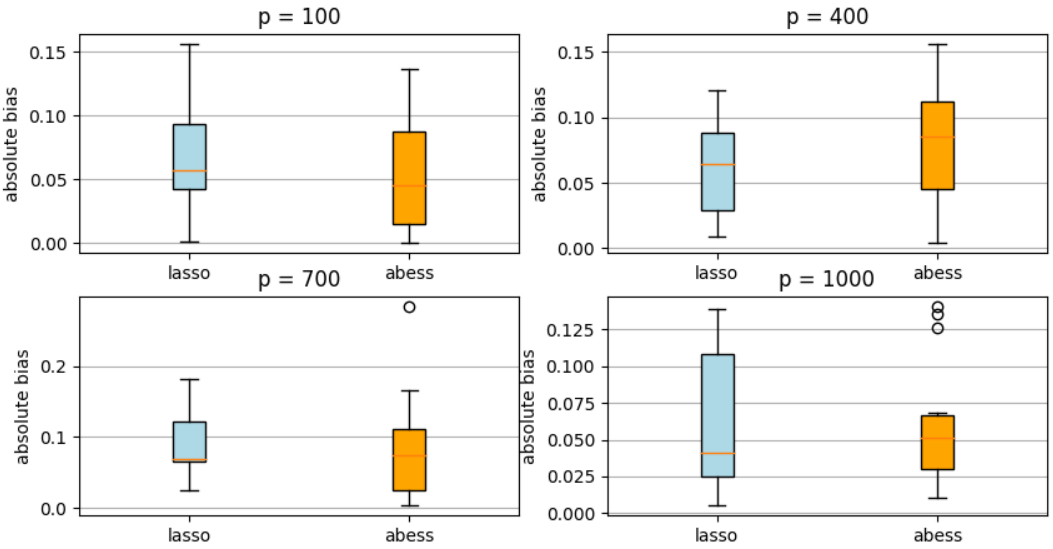
The following figures show the absolute bias of the abess learner and the lasso learner.

```
lasso = LassoCV(cv = 5)      # lasso learner
ml_g_lasso = clone(lasso)
ml_m_lasso = clone(lasso)

M = 15       # repeate times
n_obs = 200
n_vars_range = range(100,1100,300)     # different dimensions of confounding covariates
theta_lasso = np.zeros(len(n_vars_range)*M)
theta_abess = np.zeros(len(n_vars_range)*M)
time_lasso = np.zeros(len(n_vars_range)*M)
time_abess = np.zeros(len(n_vars_range)*M)
j = 0

for n_vars in n_vars_range:
```

(continues on next page)

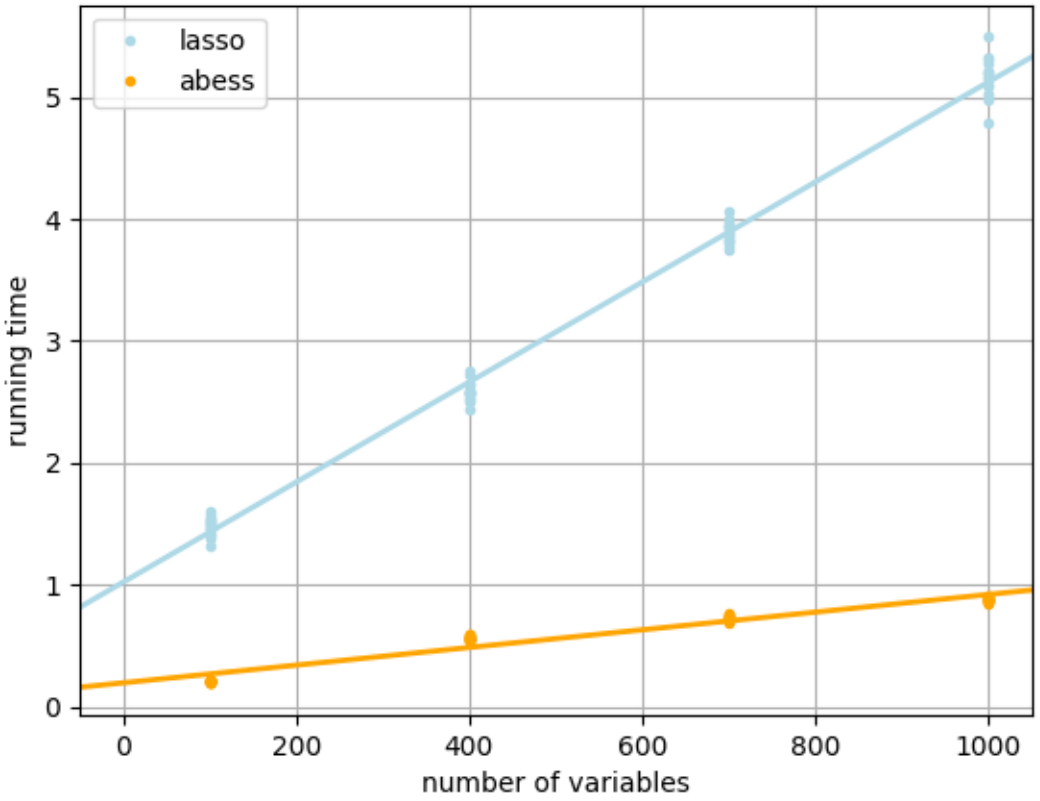
(continued from previous page)

```
    for i in range(M):
        np.random.seed(i)
        # simulated data: three true variables
        X = np.random.normal(size=(n_obs, n_vars))
        d = np.dot(X[:, :3], np.array([5]*3)) + np.random.standard_normal(size=(n_obs,))
        y = theta * d + np.dot(X[:, :3], np.array([5]*3)) + np.random.standard_
↪normal(size=(n_obs,))
        dml_data_sim = DoubleMLData.from_arrays(X, y, d)

        # Estimate double/debiased machine learning models
        starttime = time.time()
        obj_dml_plr_lasso = DoubleMLPLR(dml_data_sim, ml_g_lasso, ml_m_lasso)
        obj_dml_plr_lasso.fit()
        endtime = time.time()
        time_lasso[j*M + i] = endtime - starttime
        theta_lasso[j*M + i] = obj_dml_plr_lasso.coef

        starttime = time.time()
        obj_dml_plr_abess = DoubleMLPLR(dml_data_sim, ml_g_abess, ml_m_abess)
        obj_dml_plr_abess.fit()
        endtime = time.time()
        time_abess[j*M + i] = endtime - starttime
        theta_abess[j*M + i] = obj_dml_plr_abess.coef
    j = j + 1

# absolute bias
abs_bias1 = [abs(theta_lasso-theta)[:M],abs(theta_abess-theta)[:M]]
abs_bias2 = [abs(theta_lasso-theta)[M:2*M],abs(theta_abess-theta)[M:2*M]]
abs_bias3 = [abs(theta_lasso-theta)[2*M:3*M],abs(theta_abess-theta)[2*M:3*M]]
abs_bias4 = [abs(theta_lasso-theta)[3*M:4*M],abs(theta_abess-theta)[3*M:4*M]]
labels = ["lasso", "abess"]

fig, ([ax1, ax2], [ax3, ax4]) = plt.subplots(nrows=2, ncols=2, figsize=(10,5))
bplot1 = ax1.boxplot(abs_bias1, vert=True, patch_artist=True, labels=labels)
ax1.set_title("p = 100")
bplot2 = ax2.boxplot(abs_bias2, vert=True, patch_artist=True, labels=labels)
ax2.set_title("p = 400")
bplot3 = ax3.boxplot(abs_bias3, vert=True, patch_artist=True, labels=labels)
ax3.set_title("p = 700")
bplot4 = ax4.boxplot(abs_bias4, vert=True, patch_artist=True, labels=labels)
ax4.set_title("p = 1000")
colors = ["lightblue", "orange"]

for bplot in (bplot1, bplot2, bplot3, bplot4):
    for patch, color in zip(bplot["boxes"], colors):
        patch.set_facecolor(color)
for ax in [ax1, ax2, ax3, ax4]:
    ax.yaxis.grid(True)
    ax.set_ylabel("absolute bias")
plt.show();
```



The following figure shows the running time of the abess learner and the lasso learner.

```
plt.plot(np.repeat(n_vars_range, M),time_lasso, "o", color = "lightblue", label="lasso",␣
↪markersize=3);
plt.plot(np.repeat(n_vars_range, M),time_abess, "o", color = "orange", label="abess",␣
↪markersize=3);
slope_lasso, intercept_lasso = np.polyfit(np.repeat(n_vars_range, M),time_lasso, 1)
slope_abess, intercept_abess = np.polyfit(np.repeat(n_vars_range, M),time_abess, 1)
plt.axline(xy1=(0,intercept_lasso), slope = slope_lasso, color = "lightblue", lw = 2)
plt.axline(xy1=(0,intercept_abess), slope = slope_abess, color = "orange", lw = 2)
plt.grid()
plt.xlabel("number of variables")
plt.ylabel("running time")
plt.legend(loc="upper left")
```




```
<matplotlib.legend.Legend object at 0x7f0a41b012b0>
```

At each dimension, we repeat the double machine learning procedure 15 times for each of the two learners. As can be seen from the above figures, the parameters estimated by both learners are very close to the true parameter θ_0. But the running time of abess learner is much shorter than lasso. Besides, in high-dimensional situations, the mean absolute bias of abess learner regression is relatively smaller.

References

sphinx_gallery_thumbnail_path = 'Tutorial/figure/doubleml.png'

Total running time of the script: (3 minutes 54.402 seconds)

Work with pyts

`pyts` is a Python package dedicated to time series classification. It aims to make time series classification easily accessible by providing preprocessing and utility tools, and implementations of several time series classification algorithms. In this example, we will mainly focus on the shapelets-based algorithms.

Shapelets learning is a new primitive in time series classification. Shapelets are defined as subsequences of time series that are in some sense maximally representative of a class. Informally, in a binary classification task, a shapelet is discriminant if it is present in most series of one class and absent from series of the other class. `ShapeletTransform` is a powerful method implemented by `pyts` to perform shapelets-based feature transformation. Actually, `abess` also

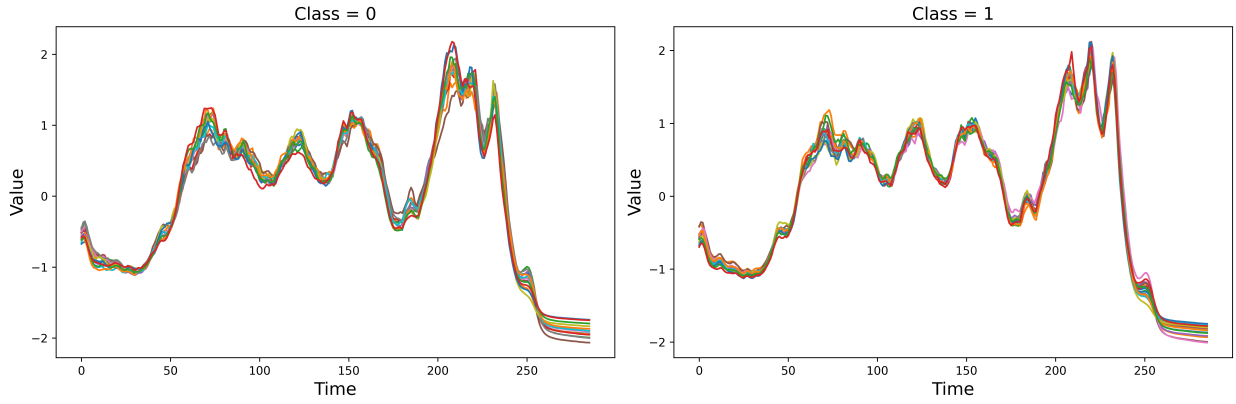
works well with shapelets-based methods. This example shows how to effectively select discriminant shapelets with abess.

```
import numpy as np
import matplotlib.pyplot as plt
import time
import warnings
warnings.filterwarnings('ignore')
from abess.linear import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.pipeline import make_pipeline
from pyts.transformation import ShapeletTransform
from pyts.datasets import load_coffee
```

Data

In this example, we use the buint-in coffee dataset in pyts to perform shapelets learning. It has two classes, 0 and 1. So, this is a binary classification task. Both train dataset and test dataset have 28 time series and the dimension of each time series is 286. We plot the time series in the train dataset.

```
X_train, X_test, y_train, y_test = load_coffee(return_X_y=True)
print("X_train shape: ", X_train.shape)
print("X_test shape: ", X_test.shape)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5), dpi=600)
ax1.plot(X_train[y_train == 0].T)
ax1.set_title("Class = 0", fontsize=15)
ax1.set_xlabel("Time", fontsize=15)
ax1.set_ylabel("Value", fontsize=15)
ax2.plot(X_train[y_train == 1].T)
ax2.set_title("Class = 1", fontsize=15)
ax2.set_xlabel("Time", fontsize=15)
ax2.set_ylabel("Value", fontsize=15)
fig.tight_layout()
plt.show()
```




```
X_train shape:  (28, 286)
X_test shape:  (28, 286)
```

Learning shapelets with abess

To select discriminant shapelets, we first collect all subsequences with predefined length and step as the candidates. Then we transform the original time series by computing the distance between them to each subsequence. Therefore, the original time series are transformed to some ultra high dimensional vectors. Finally, we perform binary classification and shapelets selection simultaneously with `LogisticRegression` implemented by `abess`.

```
class abessShapelet(object):

    def __init__(self, X_train, X_test, y_train, y_test, len_shapelet=None, step=None):
        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.n, self.p = X_train.shape
        if len_shapelet == None:
            len_shapelet = int(self.p / 4)
        if step == None:
            step = int(len_shapelet / 2)
        num_each = 1 + (self.p - len_shapelet) // step
        self.shapelets = []
        for i in range(self.n):
            for j in range(num_each):
                col = j*step
                self.shapelets.append(self.X_train[i, col:(col+len_shapelet)])
        self.shapelets = np.array(self.shapelets)

    def distant(self, x, y):
        assert x.ndim == 1 and y.ndim == 1
        n_x = len(x)
        n_y = len(y)
        if n_x <= n_y:
            dist = np.zeros(n_y-n_x+1)
            for i in range(n_y-n_x+1):
                shapelet = y[i:i+n_x]
                dist[i] = np.sum((x-shapelet)**2)
            return dist.min()
        else:
            dist = np.zeros(n_x-n_y+1)
            for i in range(n_x-n_y+1):
                shapelet = x[i:i+n_y]
                dist[i] = np.sum((y-shapelet)**2)
            return dist.min()

    def featureTransform(self, X, shapelets, index=None):
        if index is None:
            index = np.arange(shapelets.shape[0])
        n, p = X.shape
        num_shapelets, k = shapelets.shape
        new_feature = np.zeros((n, num_shapelets))
        for i in range(n):
            for j in index:
                new_feature[i, j] = self.distant(X[i], shapelets[j])
```

(continues on next page)

(continued from previous page)

```
        return new_feature

    def fit_predict(self, size=None):
        X_train_new = self.featureTransform(self.X_train, self.shapelets)
        model = LogisticRegression(support_size=size)
        model.fit(X_train_new, self.y_train)
        self.index = np.nonzero(model.coef_)[0]
        X_test_new = self.featureTransform(
            self.X_test, self.shapelets, self.index)
        y_pred = model.predict(X_test_new)
        return y_pred
```

In the following, we perform shapelets learning using `abessShapelet`. We print the performance and execution time.

```
t1 = time.time()
aShapelet = abessShapelet(X_train, X_test, y_train, y_test, len_shapelet=75)
y_pred = aShapelet.fit_predict(size=2)
score_abess = (y_pred == y_test).mean()
t2 = time.time()
print("score_abess: ", round(score_abess, 2))
print("time_abess : {}s".format(round(t2 - t1, 2)))
```

```
score_abess:  0.96
time_abess : 6.91s
```

Learning shapelets with `pyts`

We compare our method with the one implemented in `pyts`, which is a two-step procedure. First, it selects discriminant shapelets based on mutual information. Then, a support vector machine is applied to perform binary classification with transformed time series based on those selected shapelets. Analogously, we print the performance and execution time.

```
t3 = time.time()
shapelet = ShapeletTransform(n_shapelets=2, window_sizes=[75], sort=True)
svc = LinearSVC()
clf = make_pipeline(shapelet, svc)
clf.fit(X_train, y_train)
score_pyts = clf.score(X_test, y_test)
t4 = time.time()
print("score_pyts: ", round(score_pyts, 2))
print("time_pyts : {}s".format(round(t4 - t3, 2)))
```

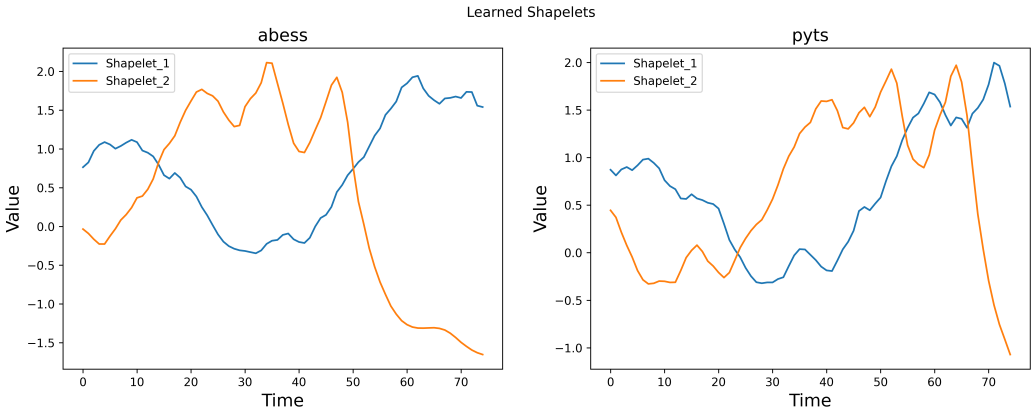
```
score_pyts:  0.96
time_pyts : 27.49s
```

It can be seen from the above results that the linear classifier `abessShapelet` obtains the same performance with the method implemented by `pyts` while the running time is much shorter.

Plot: learned shapelets

The following figure shows the discriminant shapelets selected by these two methods.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5), dpi=600)

ax1.plot(aShapelet.shapelets[aShapelet.index][0], label="Shapelet_1")
ax1.plot(aShapelet.shapelets[aShapelet.index][1], label="Shapelet_2")
ax1.legend()
ax1.set_title("abess", fontsize=15)
ax1.set_xlabel("Time", fontsize=15)
ax1.set_ylabel("Value", fontsize=15)

ax2.plot(shapelet.shapelets_[0], label="Shapelet_1")
ax2.plot(shapelet.shapelets_[1], label="Shapelet_2")
ax2.legend()
ax2.set_title("pyts", fontsize=15)
ax2.set_xlabel("Time", fontsize=15)
ax2.set_ylabel("Value", fontsize=15)

plt.suptitle("Learned Shapelets")
plt.show()
```



sphinx_gallery_thumbnail_path = 'Tutorial/figure/pyts.png'

Total running time of the script: (0 minutes 38.858 seconds)

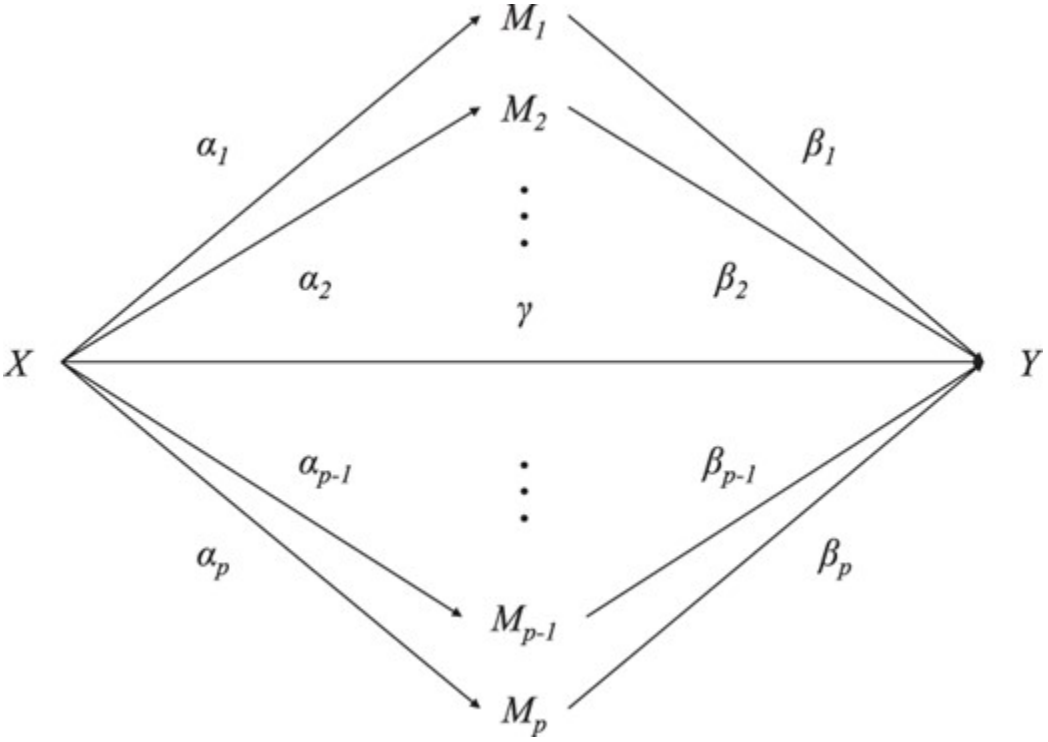
Work with DoWhy

`DoWhy` is a Python library for causal inference that supports explicit modeling and testing of causal assumptions. In this section, we will use `abess` to cope with high-dimensional mediation analysis problem, which is a popular topic in the research of causal inference. High-Dimensional mediation model is crucial in biomedical research studies. People always want to know by what mechanism the genes with differential expression, distinct genotypes or various epigeneticmarkers affect the outcome or phenotype. Such a mechanistic process is what a mediation analysis aims to characterize (Huang and Pan, 2016[2]).

A typical example of the high-dimensional mediators is high-dimensional DNA methylation markers. This model can

[2] Huang YT & Pan WC (2016). "Hypothesis test of mediation effect in causal mediation model with high-dimensional continuous mediators." Biometrics, 72(2), 402-413. (doi.org/10.1111/biom.12421)

be represented by the following figure (Zhang et al. 2016[1]),



where X is treatment (exposure), Y is outcome, and $M_k, k = 1, ..., p$ are (potential) mediators. Moreover, $\alpha = (\alpha_1, \cdots, \alpha_p)^{\mathrm{T}}$ denotes the parameters relating X to mediators, and $\beta = (\beta_1, \cdots, \beta_p)^{\mathrm{T}}$ denotes the parameters relating the mediators to the outcome Y. For the latter relation, we would consider both continuous outcomes (linear regression) and binary outcomes (logistic regression). These two models can be implemented by `abess` directly.

For instance, if the outcome is continuous, then we assume the model take the form

$$M_k = c_k + \alpha_k X + e_k, \quad k = 1, \cdots, p$$
$$Y = c + \gamma X + \beta_1 M_1 + \cdots + \beta_p M_p + \epsilon,$$

Among all the possible M_p (p may be large), only few of them are real mediators, which means that both α_k and β_k should be non-zero. Then, an indirect path $X \to M_k \to Y$ can be built. Next, we will show that by directly using ABESS in a naive form, we can get a good result.

Continuous Outcome

We will follow the simulation settings and the data generating process in the R document of the R package HIMA (Zhang et al. 2016). X is generated from $N(0, 1.5)$, the first 8 elements of $\beta\,(\beta_k, k = 1, \cdots, 8)$ are $(0.55, 0.6, 0.65, 0.7, 0.8, 0.8, 0, 0)^{\mathrm{T}}$, and the first 8 elements of $a\,(\alpha_k, k = 1, \cdots, 8)$ are $(0.45, 0.5, 0.6, 0.7, 0, 0, 0.5, 0.5)^{\mathrm{T}}$. The rest of β and α are all 0 . Let $c = 1, \gamma = 0.5$.c_k is chosen as a random number from $U(0, 2)$.e_k and ϵ are generated from $N(0, 1.2)$ and $N(0, 1)$, respectively.

```
import numpy as np
import pandas as pd
import random
```

(continues on next page)

[1] Zhang H, Zheng Y, Zhang Z, Gao T, Joyce B, Yoon G, Zhang W, Schwartz J, Just A, Colicino E, Vokonas P, Zhao L, Lv J, Baccarelli A, Hou L & Liu L (2016). "Estimating and Testing High-dimensional Mediation Effects in Epigenetic Studies." Bioinformatics, 32(20), 3150-3154. (doi.org/10.1093/bioinformatics/btw351).

(continued from previous page)

```
import abess
import math
from dowhy import CausalModel
import dowhy.datasets
import dowhy.causal_estimators.linear_regression_estimator
import warnings
warnings.filterwarnings('ignore')

# The data-generating function:

def simhima (n,p,alpha,beta,seed,binary=False):
    random.seed(seed)
    ck = np.random.uniform(0,2,size=p)
    M = np.zeros((n,p))
    X = np.random.normal(0,1.5,size=n)
    for i in range(n):
        e = np.random.normal(0,1.2,size=p)
        M[i,:] = ck + X[i]*alpha + e
    X = X.reshape(n,1)
    XM = np.concatenate((X,M),axis=1)
    B = np.concatenate((np.array([0.5]),beta),axis=0)
    E = np.random.normal(0,1,size=n)
    Y = 0.5 + np.matmul(XM,B) + E
    if binary:
        Y = np.random.binomial(1,1/(1+np.exp(Y)),size=n)
    return {"Y":Y, "M":M, "X":X}

n = 300
p = 200
alpha = np.zeros(p)
beta = np.zeros(p)
alpha[0:8] = (0.45,0.5,0.6,0.7,0,0,0.5,0.5)
beta[0:8] = (0.55,0.6,0.65,0.7,0.8,0.8,0,0)
simdat = simhima(n,p,alpha,beta,seed=12345)
```

Now, let's examine again our settings. There are altogether $p = 200$ possible mediators, but only few of them are the true mediators that we really want to find out. A true mediator must have both non-zero α and β, so only the first four mediators satisfy this condition (indices 0,1,2,3). We also set up four false mediators that are easily confused (indices 4,5,6,7), which have either non-zero α or β, and should avoid being selected by our method.

The whole structure can be divided into left and right parts. The left part is about the paths $X \to M_i, i = 1, 2, ...p$, and the right part is about the paths $M_i \to Y, i = 1, 2, ...p$. A natural idea is to apply `abess` to these two subproblems separately. Notice: the right part is in the same form as the problem `abess` wants to solve: one dependent variable and multiple possible independent variables, but the left part is opposite: we have one independent variable and multiple possible dependent variables. In this case, continuing to naively use `abess` may lead to philosophical causal issues and cannot have good theoretical guarantees, since we have to treat X as an "dependent variable" and treat M_i as "independent variables", which is contrary to the interpretation in reality. However, this naive approach performs well in this task of selecting true mediators, and this kind of idea has already been used in some existing methods, such as Coordinate-wise Mediation Filter (Van Kesteren and Oberski, 2019[3]). Therefore, we will still use this kind of idea here, and the main task is to show the power of `abess`.

[3] Van Kesteren, E. J., & Oberski, D. L. (2019). "Exploratory mediation analysis with many potential mediators." Structural Equation Modeling: A Multidisciplinary Journal, 26(5), 710-723. (doi.org/10.1080/10705511.2019.1588124)

We will first apply BESS with a fixed support size to one of these subproblems, conducting a preliminary screening to determine some candidate mediators, and then apply ABESS with an adaptive size to the second subproblem and decide the final selected mediators. If we directly use ABESS in the first subproblem, the candidate set would be too small, and make the ABESS in the second step meaningless (because the candidate set is no longer high-dimensional at this time), which could induce a large drop of TPR (True Positive Rate). The support size used in the first step can be tuned, and its size is preferably 2 to 4 times the number of true mediators.

Now there is a problem of order. Since we have to run `abess` twice, should we do the left half or the right half first? We've found that doing the left half first is almost always a better choice. The reason is as follows: if we do the right half first, those false mediators that only have correlation coefficients with the left half will be easily selected because there is an $M_i \leftarrow X \rightarrow Y$ path (note that from X to Y has not only indirect paths, but also a direct path!), and once these false mediators are selected into the second step, they will be selected eventually because they have non-zero coefficients in the left half, resulting in uncontrollable FDR. But doing the left half first won't have such a problem.

```
model = abess.LinearRegression(support_size=10)
model.fit(simdat["M"],simdat["X"])
ind = np.nonzero(model.coef_)
print("estimated non-zero: ", ind)
print("estimated coef: ", model.coef_[ind])
```

```
estimated non-zero:  (array([  0,   1,   2,   3,   6,   7,  46, 112, 124, 185]),)
estimated coef:  [ 0.19938462  0.14855243  0.27148782  0.23536246  0.15900585  0.23215292
 -0.10422092 -0.11303449  0.09631314 -0.08907528]
```

This the subproblem of left half, and we use a "support size=10" to demonstrate conveniently. These 10 mediators have been selected in the first step and entered our candidate set for the second step. Recall that the true mediators we want to find have index 0,1,2,3. They are all selected in the candidate set.

```
model1 = abess.LinearRegression()
model1.fit(simdat["M"].T[ind].T,simdat["Y"])
ind1 = np.nonzero(model1.coef_)
print("estimated non-zero: ", ind[0][ind1])
recorded_index = ind[0][ind1]
```

```
estimated non-zero:  [0 1 2 3 7]
```

This is the second step, and we use an adaptive support size, which lead to a final selection: index 0,1,2,3. We've perfectly accomplished the task of selecting real mediators. After this step, we can use the `DoWhy` library for our data for further analysis.

```
m_num = len(recorded_index)
df = pd.DataFrame(simdat["M"].T[recorded_index].T, columns=["FD"+str(i) for i in range(m_
↪num)])
df["y"] = simdat["Y"]
df["v0"] = simdat["X"]
df.head()
```

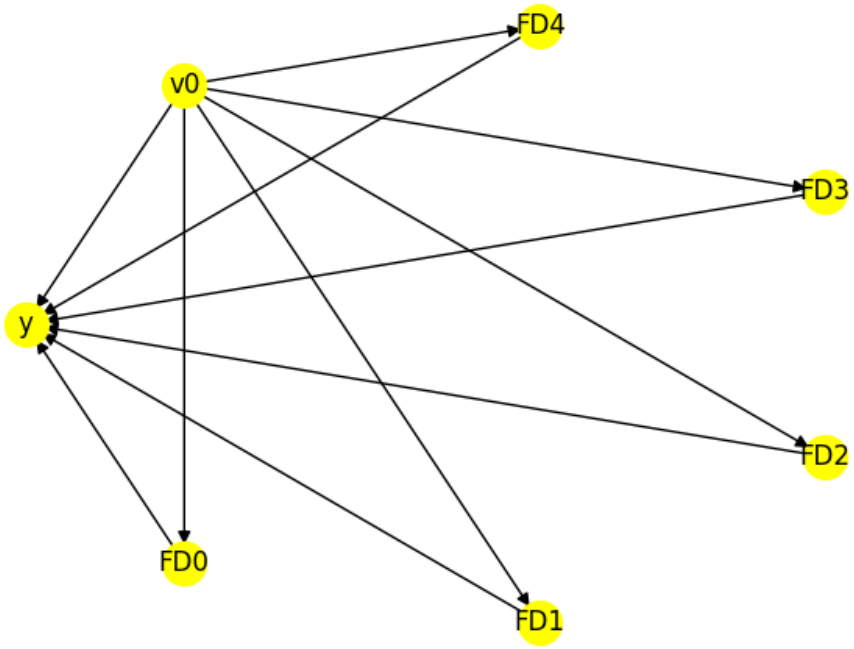
In order to adapt to the naming convention of the `DoWhy` library, we renamed the above variables. v0 is treatment, y is outcome, and FD0 to FD3 (short for Front Door) are mediators.

```
data = dowhy.datasets.linear_dataset(0.5, num_common_causes=0, num_samples=300,
                                     num_instruments=0, num_effect_modifiers=0,
                                     num_treatments=1,
                                     num_frontdoor_variables=m_num,
```

(continues on next page)

(continued from previous page)

```
                                    treatment_is_binary=False,
                                   outcome_is_binary=False)
my_graph = data["gml_graph"][:-1] + ' edge[ source "v0" target "y"]]'
model = CausalModel(df,"v0","y",my_graph,
                    missing_nodes_as_confounders=True)
model.view_model()
```



DoWhy library can directly display the causal graph we built. Now we can do identification and estimation based on this causal graph and the data we simulated with DoWhy. For example, we are going to estimate the natural indirect effect (NIE) of the first mediator M_0 (FD0).

```
identified_estimand_nie = model.identify_effect(estimand_type="nonparametric-nie",
                                                proceed_when_unidentifiable=True)
causal_estimate_nie = model.estimate_effect(identified_estimand_nie,
                                            method_name="mediation.two_stage_regression",
                                           confidence_intervals=False,
                                           test_significance=False,
                                            method_params = {
                                                'first_stage_model': dowhy.causal_estimators.
↪linear_regression_estimator.LinearRegressionEstimator,
                                                'second_stage_model': dowhy.causal_
↪estimators.linear_regression_estimator.LinearRegressionEstimator
```

(continues on next page)

(continued from previous page)

```
                                        }
                                       )
print("The estimate of the natural indirect effect of the first mediator is ",causal_
↪estimate_nie.value)
```

```
two_stage_regression
{'control_value': 0, 'treatment_value': 1, 'test_significance': False, 'evaluate_effect_
↪strength': False, 'confidence_intervals': False, 'target_units': 'ate', 'effect_
↪modifiers': [], 'first_stage_model': <class 'dowhy.causal_estimators.linear_regression_
↪estimator.LinearRegressionEstimator'>, 'second_stage_model': <class 'dowhy.causal_
↪estimators.linear_regression_estimator.LinearRegressionEstimator'>}
{'control_value': 0, 'treatment_value': 1, 'test_significance': False, 'evaluate_effect_
↪strength': False, 'confidence_intervals': False, 'target_units': 'ate', 'effect_
↪modifiers': [], 'first_stage_model': <class 'dowhy.causal_estimators.linear_regression_
↪estimator.LinearRegressionEstimator'>, 'second_stage_model': <class 'dowhy.causal_
↪estimators.linear_regression_estimator.LinearRegressionEstimator'>}
The estimate of the natural indirect effect of the first mediator is  0.512104917915887
```

Recall that we have $\alpha_0 = 0.45$ and $\beta_0 = 0.55$, the true value of the natural indirect effect of the first mediator is $0.45 \times 0.55 = 0.2475$. Similarly, we can also get the estimated value of NIE of other mediator variables, and also the natural direct effect (NDE). Since the linear regression model has a simple and known form in our simulation, it's obvious that the accuracy of our estimates depends only on whether we choose the correct mediators. Next, we would do 1000 replications and see the performance of *abess* on choosing mediators.

```
recorded_index = ind[0][ind1]
for i in range(999):
    simdat = simhima(n,p,alpha,beta,seed=i)
    model = abess.LinearRegression(support_size=10)
    model.fit(simdat["M"],simdat["X"])
    ind = np.nonzero(model.coef_)
    model1 = abess.LinearRegression()
    model1.fit(simdat["M"].T[ind].T,simdat["Y"])
    ind1 = np.nonzero(model1.coef_)
    recorded_index = np.concatenate((recorded_index,ind[0][ind1]),axis=0)
mask = np.unique(recorded_index)
tmp = []
for v in mask:
    tmp.append(np.sum(recorded_index==v))
np.vstack((mask,np.array(tmp))).T
```

```
array([[   0,  985],
       [   1,  998],
       [   2, 1000],
       [   3, 1000],
       [   4,   21],
       [   5,   20],
       [   6,   40],
       [   7,   39],
       [  31,    1],
       [  46,    1],
       [  54,    1],
       [  57,    1],
```

(continues on next page)

(continued from previous page)

```
       [  73,    1],
       [  81,    1],
       [  83,    1],
       [  84,    1],
       [  91,    1],
       [ 102,    1],
       [ 128,    1],
       [ 147,    1],
       [ 159,    1],
       [ 168,    1],
       [ 175,    1],
       [ 177,    1]])
```

After doing 1000 replications of the process mentioned above, we can get this list. The left number in each row is the index, and the right number is the times that this mediator been selected. We can find that:

- The true mediators (indices 0-3) can always be selected during all the 1000 replications.
- The bewildering false mediators (indices 4-7) may be occasionally selected, but FDR can be controlled at a low level.
- It is almost impossible for other mediators (indices 8-199) to be selected by our method.

Now, we can do the confusion matrix analysis, and output some commonly used metrics to measure our selection method.

```
Positive = 4*1000
Negative = 196*1000
TP = np.sum(tmp[:4])
FP = np.sum(tmp[4:])
FN = Positive-TP
TN = Negative-FP
TPR = TP/Positive
TNR = TN/Negative
PPV = TP/(TP+FP)
FDR = 1-PPV
ACC = (TP+TN)/(Positive+Negative)
F1 = 2*TP/(2*TP+FP+FN)
print('TPR:',TPR,'\nTNR:',TNR,'\nFDR:',FDR,'\nPPV:',PPV,'\nACC:',ACC,'\nF1 score:',F1)
```

```
TPR: 0.99575
TNR: 0.9993061224489795
FDR: 0.033017722748239886
PPV: 0.9669822772517601
ACC: 0.999235
F1 score: 0.9811553146939278
```

Binary Outcome

For binary outcome, we still follow the simulation settings of the R documentation of the R package HIMA. We increased the sample size from 300 to 600, which is also a reasonable size.

```
n = 600
p = 200
alpha = np.zeros(p)
beta = np.zeros(p)
alpha[0:8] = (0.45,0.5,0.6,0.7,0,0,0.5,0.5)
beta[0:8] = (1.45,1.5,1.55,1.6,1.7,1.7,0,0)
simdat = simhima(n,p,alpha,beta,seed=12345,binary=True)
```

First step:

```
model = abess.LinearRegression(support_size=10)
model.fit(simdat["M"],simdat["X"])
ind = np.nonzero(model.coef_)
print("estimated non-zero: ", ind)
print("estimated coef: ", model.coef_[ind])
```

```
estimated non-zero:  (array([  0,   1,   2,   3,   6,   7,  52, 154, 168, 197]),)
estimated coef:  [ 0.22098081  0.18630423  0.2115897   0.3276324   0.14303822  0.22073874
 -0.06781737 -0.06798672 -0.06352619  0.05718398]
```

Second step:

```
model1 = abess.LogisticRegression()
model1.fit(simdat["M"].T[ind].T,simdat["Y"])
ind1 = np.nonzero(model1.coef_)
print("estimated non-zero: ", ind[0][ind1])
recorded_index = ind[0][ind1]
```

```
estimated non-zero:  [0 1 2 3]
```

Again, we got a perfect result.

```
recorded_index = ind[0][ind1]
for i in range(999):
    simdat = simhima(n,p,alpha,beta,seed=i,binary=True)
    model = abess.LinearRegression(support_size=10)
    model.fit(simdat["M"],simdat["X"])
    ind = np.nonzero(model.coef_)
    model1 = abess.LogisticRegression()
    model1.fit(simdat["M"].T[ind].T,simdat["Y"])
    ind1 = np.nonzero(model1.coef_)
    recorded_index = np.concatenate((recorded_index,ind[0][ind1]),axis=0)
mask = np.unique(recorded_index)
tmp = []
for v in mask:
    tmp.append(np.sum(recorded_index==v))
np.vstack((mask,np.array(tmp))).T
```

```
array([[  0, 903],
       [  1, 934],
       [  2, 959],
       [  3, 986],
       [  4,  25],
       [  5,  21],
       [  6,   2],
       [  7,   7],
       [ 37,   1],
       [111,   1],
       [136,   1],
       [143,   1],
       [146,   1],
       [188,   2]])
```

TPR has dropped significantly because problems with binary outcomes require a larger sample size than problems with continuous outcomes. But we found that FDR was also well controlled.

```
Positive = 4*1000
Negative = 196*1000
TP = np.sum(tmp[:4])
FP = np.sum(tmp[4:])
FN = Positive-TP
TN = Negative-FP
TPR = TP/Positive
TNR = TN/Negative
PPV = TP/(TP+FP)
FDR = 1-PPV
ACC = (TP+TN)/(Positive+Negative)
F1 = 2*TP/(2*TP+FP+FN)
print('TPR:',TPR,'\nTNR:',TNR,'\nFDR:',FDR,'\nPPV:',PPV,'\nACC:',ACC,'\nF1 score:',F1)
```

```
TPR: 0.9455
TNR: 0.9996836734693878
FDR: 0.016129032258064502
PPV: 0.9838709677419355
ACC: 0.9986
F1 score: 0.9643039265680775
```

In a word, by simply using `abess` in high-dimensional mediation analysis problem, we can get good results both under continuous and binary outcome settings.

References

sphinx_gallery_thumbnail_path = 'Tutorial/figure/dowhy.png'

Total running time of the script: (1 minutes 5.933 seconds)

7.3 abess Python Package

This page contains links to all the related API documents on Python package.

7.3.1 Generalized Linear Models

This page contains links to `abess.linear` module and its functions. It is used for GLM problem.

LinearRegression

Warning: In the old version of abess (before 0.4.0), this model is named *abess.linear.abessLm*. Please note that it will be deprecated in version 0.6.0.

class abess.linear.**LinearRegression**(*path_type='seq'*, *support_size=None*, *s_min=None*, *s_max=None*, *group=None*, *alpha=None*, *ic_type='ebic'*, *ic_coef=1.0*, *cv=1*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=0*, *important_search=128*, *screening_size=- 1*, *covariance_update=False*)

Adaptive Best-Subset Selection(ABESS) algorithm for linear regression.

Parameters

- **path_type** (*{"seq", "gs"}, optional, default="seq"*) -- The method to be used to select the optimal support size.
  - For path_type = "seq", we solve the best subset selection problem for each size in support_size.
  - For path_type = "gs", we solve the best subset selection problem with support size ranged in (s_min, s_max), where the specific support size to be considered is determined by golden section.
- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes. Only used when path_type = "seq".
- **s_min** (*int, optional, default=0*) -- The lower bound of golden-section-search for sparsity searching.
- **s_max** (*int, optional, default=min(n, int(n/(log(log(n))log(p))))*.) -- The higher bound of golden-section-search for sparsity searching.
- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **alpha** (*float, optional, default=0*) -- Constant that multiples the L2 term in loss function, controlling regularization strength. It should be non-negative.

  - If alpha = 0, it indicates ordinary least square.

- **ic_type** (*{'aic', 'bic', 'gic', 'ebic'}, optional, default='ebic'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.
  - If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p)))))$.
- **primary_model_fit_max_iter** (*int, optional, default=10*) -- The maximal number of iteration for primary_model_fit.
- **primary_model_fit_epsilon** (*float, optional, default=1e-08*) -- The epsilon (threshold) of iteration for primary_model_fit.
- **splicing_type** (*{0, 1}, optional, default=0*) -- The type of splicing: "0" for decreasing by half, "1" for decresing by one.
- **important_search** (*int, optional, default=128*) -- The size of inactive set during updating active set when splicing. It should be a non-positive integer and if important_search=0, it would be set as the size of whole inactive set.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.linear import LinearRegression
>>> from abess.datasets import make_glm_data
>>> import numpy as np
>>> np.random.seed(12345)
>>> data = make_glm_data(n = 100, p = 50, k = 10, family = 'gaussian')
>>> model = LinearRegression(support_size = 10)
>>> model.fit(data.x, data.y)
LinearRegression(always_select=[], support_size=10)
>>> model.predict(data.x)
array([   1.42163813,  -43.23929886, -139.79509191,  141.45138403])
```

```
>>> ### Sparsity unknown
>>>
>>> # path_type="seq"
>>> model = LinearRegression(path_type = "seq")
>>> model.fit(data.x, data.y)
LinearRegression(always_select=[])
>>> model.predict(data.x)[1:4]
array([   1.42163813,  -43.23929886, -139.79509191,  141.45138403])
>>>
>>> # path_type="gs"
>>> model = LinearRegression(path_type="gs")
>>> model.fit(data.x, data.y)
LinearRegression(always_select=[], path_type='gs')
>>> model.predict(data.x)[1:4]
array([   1.42163813,  -43.23929886, -139.79509191,  141.45138403])
```

coef_

Estimated coefficients for the best subset selection problem.

Type array-like, shape(p_features,) or (p_features, M_responses)

intercept_

The intercept in the model.

Type float or array-like, shape(M_responses,)

train_loss_

The loss on training data.

Type float

eval_loss_

- If cv=1, it stores the score under chosen information criterion.
- If cv>1, it stores the test loss under cross-validation.

Type float

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

predict(*X*)

Predict on given data.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **y** -- Prediction of the mean on given X.

Return type array-like, shape(n_samples,)

score(*X*, *y*, *sample_weight=None*)

Give data, and it returns the coefficient of determination.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix.
- **y** (*array-like, shape(n_samples, p_features)*) -- Real response for given X.
- **sample_weight** (*array-like, shape(n_samples,), default=None*) -- Sample weights.

Returns **score** -- R^2 score.

Return type float

fit(*X=None*, *y=None*, *is_normal=True*, *sample_weight=None*, *cv_fold_id=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like of shape(n_samples, p_features)*) -- Training data matrix. It should be a numpy array.
- **y** (*array-like of shape(n_samples,) or (n_samples, M_responses)*) -- Training response values. It should be a numpy array.
  - For regression problem, the element of y should be float.
  - For classification problem, the element of y should be either 0 or 1. In multinomial regression, the p features are actually dummy variables.
  - For survival data, y should be a $n \times 2$ array, where the columns indicates "censoring" and "time", respectively.
- **is_normal** (*bool, optional, default=True*) -- whether normalize the variables array before fitting the algorithm.
- **sample_weight** (*array-like, shape (n_samples,), optional*) -- Individual weights for each sample. Only used for is_weight=True. Default=np.ones(n).
- **cv_fold_id** (*array-like, shape (n_samples,), optional, default=None*) -- An array indicates different folds in CV. Samples in the same fold should be given the same number.
- **sparse_matrix**(*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) -- If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** -- Parameter names mapped to their values.

Return type dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) -- Estimator parameters.

Returns **self** -- Estimator instance.

Return type estimator instance

LogisticRegression

Warning: In the old version of abess (before 0.4.0), this model is named *abess.linear.abessLogistic*. Please note that it will be deprecated in version 0.6.0.

class abess.linear.**LogisticRegression**(*path_type='seq'*, *support_size=None*, *s_min=None*, *s_max=None*, *group=None*, *alpha=None*, *ic_type='ebic'*, *ic_coef=1.0*, *cv=1*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=0*, *important_search=128*, *screening_size=- 1*, *primary_model_fit_max_iter=10*, *primary_model_fit_epsilon=1e-08*, *approximate_Newton=False*)

Adaptive Best-Subset Selection (ABESS) algorithm for logistic regression.

Parameters

- **path_type** (*{"seq", "gs"}, optional, default="seq"*) -- The method to be used to select the optimal support size.
  - For path_type = "seq", we solve the best subset selection problem for each size in support_size.
  - For path_type = "gs", we solve the best subset selection problem with support size ranged in (s_min, s_max), where the specific support size to be considered is determined by golden section.
- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes. Only used when path_type = "seq".
- **s_min** (*int, optional, default=0*) -- The lower bound of golden-section-search for sparsity searching.
- **s_max** (*int, optional, default=min(n, int(n/(log(log(n))log(p))))*.) -- The higher bound of golden-section-search for sparsity searching.

- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **alpha** (*float, optional, default=0*) -- Constant that multiples the L2 term in loss function, controlling regularization strength. It should be non-negative.
  - If alpha = 0, it indicates ordinary least square.
- **ic_type** (*{'aic', 'bic', 'gic', 'ebic'}, optional, default='ebic'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.
  - If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p))))$.
- **primary_model_fit_max_iter** (*int, optional, default=10*) -- The maximal number of iteration for primary_model_fit.
- **primary_model_fit_epsilon** (*float, optional, default=1e-08*) -- The epsilon (threshold) of iteration for primary_model_fit.
- **splicing_type** (*{0, 1}, optional, default=0*) -- The type of splicing: "0" for decreasing by half, "1" for decresing by one.
- **important_search** (*int, optional, default=128*) -- The size of inactive set during updating active set when splicing. It should be a non-positive integer and if important_search=0, it would be set as the size of whole inactive set.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.linear import LogisticRegression
>>> from abess.datasets import make_glm_data
>>> import numpy as np
>>> np.random.seed(12345)
>>> data = make_glm_data(n = 100, p = 50, k = 10, family = 'binomial')
>>> model = LogisticRegression(support_size = 10)
>>> model.fit(data.x, data.y)
LogisticRegression(always_select=[], support_size=10)
>>> model.predict(data.x)[1:10]
array([0., 0., 1., 1., 0., 1., 0., 1., 0.])
```

```
>>> ### Sparsity unknown
>>>
>>> # path_type="seq"
>>> model = LogisticRegression(path_type = "seq")
>>> model.fit(data.x, data.y)
LogisticRegression(always_select=[])
>>> model.predict(data.x)[1:10]
array([0., 1., 0., 0., 1., 0., 1., 0., 1., 1.])
>>>
>>> # path_type="gs"
>>> model = LogisticRegression(path_type="gs")
>>> model.fit(data.x, data.y)
LogisticRegression(always_select=[], path_type='gs')
>>> model.predict(data.x)[1:10]
array([0., 0., 0., 1., 1., 0., 1., 0., 1., 0.])
```

coef_

Estimated coefficients for the best subset selection problem.

Type array-like, shape(p_features,) or (p_features, M_responses)

intercept_

The intercept in the model.

Type float or array-like, shape(M_responses,)

train_loss_

The loss on training data.

Type float

eval_loss_

- If cv=1, it stores the score under chosen information criterion.
- If cv>1, it stores the test loss under cross-validation.

Type float

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

`predict_proba`(*X*)

Give the probabilities of new sample being assigned to different classes.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **proba** -- Returns the probabilities for class "0" and "1" on given X.

Return type array-like, shape(n_samples, 2)

`predict`(*X*)

This function predicts class label for given data.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **y** -- Predict class labels (0 or 1) for samples in X.

Return type array-like, shape(n_samples,)

`score`(*X*, *y*, *sample_weight=None*)

Give new data, and it returns the prediction accuracy.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix.
- **y** (*array-like, shape(n_samples,)*) -- Real class labels (0 or 1) for X.
- **sample_weight** (*array-like, shape(n_samples,), default=None*) -- Sample weights.

Returns **score** -- The mean prediction accuracy on the given data.

Return type float

`fit`(*X=None*, *y=None*, *is_normal=True*, *sample_weight=None*, *cv_fold_id=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like of shape(n_samples, p_features)*) -- Training data matrix. It should be a numpy array.
- **y** (*array-like of shape(n_samples,) or (n_samples, M_responses)*) -- Training response values. It should be a numpy array.
  - For regression problem, the element of y should be float.
  - For classification problem, the element of y should be either 0 or 1. In multinomial regression, the p features are actually dummy variables.
  - For survival data, y should be a $n \times 2$ array, where the columns indicates "censoring" and "time", respectively.
- **is_normal** (*bool, optional, default=True*) -- whether normalize the variables array before fitting the algorithm.
- **sample_weight** (*array-like, shape (n_samples,), optional*) -- Individual weights for each sample. Only used for is_weight=True. Default=np.ones(n).

- **cv_fold_id** (*array-like, shape (n_samples,), optional, default=None*) -- An array indicates different folds in CV. Samples in the same fold should be given the same number.
- **sparse_matrix** (*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) -- If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** -- Parameter names mapped to their values.

Return type dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) -- Estimator parameters.

Returns **self** -- Estimator instance.

Return type estimator instance

PoissonRegression

Warning: In the old version of abess (before 0.4.0), this model is named *abess.linear.abessPoisson*. Please note that it will be deprecated in version 0.6.0.

class abess.linear.**PoissonRegression**(*path_type='seq'*, *support_size=None*, *s_min=None*, *s_max=None*, *group=None*, *alpha=None*, *ic_type='ebic'*, *ic_coef=1.0*, *cv=1*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=0*, *important_search=128*, *screening_size=- 1*, *primary_model_fit_max_iter=10*, *primary_model_fit_epsilon=1e-08*, *approximate_Newton=False*)

Adaptive Best-Subset Selection(ABESS) algorithm for Poisson regression.

Parameters

- **path_type** (*{"seq", "gs"}, optional, default="seq"*) -- The method to be used to select the optimal support size.
  - For path_type = "seq", we solve the best subset selection problem for each size in support_size.
  - For path_type = "gs", we solve the best subset selection problem with support size ranged in (s_min, s_max), where the specific support size to be considered is determined by golden section.

- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes. Only used when path_type = "seq".
- **s_min** (*int, optional, default=0*) -- The lower bound of golden-section-search for sparsity searching.
- **s_max** (*int, optional, default=min(n, int(n/(log(log(n))log(p)))).*) -- The higher bound of golden-section-search for sparsity searching.
- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **alpha** (*float, optional, default=0*) -- Constant that multiples the L2 term in loss function, controlling regularization strength. It should be non-negative.
  - If alpha = 0, it indicates ordinary least square.
- **ic_type** (*{'aic', 'bic', 'gic', 'ebic'}, optional, default='ebic'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.
  - If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p))))$.
- **primary_model_fit_max_iter** (*int, optional, default=10*) -- The maximal number of iteration for primary_model_fit.

- **primary_model_fit_epsilon** (*float, optional, default=1e-08*) -- The epsilon (threshold) of iteration for primary_model_fit.
- **splicing_type** (*{0, 1}, optional, default=0*) -- The type of splicing: "0" for decreasing by half, "1" for decresing by one.
- **important_search** (*int, optional, default=128*) -- The size of inactive set during updating active set when splicing. It should be a non-positive integer and if important_search=0, it would be set as the size of whole inactive set.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.linear import PoissonRegression
>>> from abess.datasets import make_glm_data
>>> import numpy as np
>>> np.random.seed(12345)
>>> data = make_glm_data(n = 100, p = 50, k = 10, family = 'poisson')
>>> model = PoissonRegression(support_size = 10)
>>> model.fit(data.x, data.y)
PoissonRegression(always_select=[], support_size=10)
>>> model.predict(data.x)[1:4]
array([1.06757251e+00, 8.92711312e-01, 5.64414159e-01, 1.35820866e+00])
```

```
>>> ### Sparsity unknown
>>>
>>> # path_type="seq"
>>> model = PoissonRegression(path_type = "seq")
>>> model.fit(data.x, data.y)
PoissonRegression(always_select=[])
>>> model.predict(data.x)[1:4]
array([1.03373139e+00, 4.32229653e-01, 4.48811009e-01, 2.27170366e+00])
>>>
>>> # path_type="gs"
>>> model = PoissonRegression(path_type="gs")
>>> model.fit(data.x, data.y)
PoissonRegression(always_select=[], path_type='gs')
>>> model.predict(data.x)[1:4]
array([1.03373139e+00, 4.32229653e-01, 4.48811009e-01, 2.27170366e+00])
```

coef_

Estimated coefficients for the best subset selection problem.

Type array-like, shape(p_features,) or (p_features, M_responses)

intercept_

The intercept in the model.

Type float or array-like, shape(M_responses,)

train_loss_

The loss on training data.

Type float

eval_loss_

- If cv=1, it stores the score under chosen information criterion.
- If cv>1, it stores the test loss under cross-validation.

Type float

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

predict(*X*)

Predict on given data.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **y** -- Prediction of the mean on X.

Return type array-like, shape(n_samples,)

score(*X*, *y*, *sample_weight=None*)

Give new data, and it returns the D^2 score.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix.
- **y** (*array-like, shape(n_samples, p_features)*) -- Real response for given X.
- **sample_weight** (*array-like, shape(n_samples,), default=None*) -- Sample weights.

Returns **score** -- D^2 score.

Return type float

fit(*X=None*, *y=None*, *is_normal=True*, *sample_weight=None*, *cv_fold_id=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like of shape(n_samples, p_features)*) -- Training data matrix. It should be a numpy array.
- **y** (*array-like of shape(n_samples,) or (n_samples, M_responses)*) -- Training response values. It should be a numpy array.
  - For regression problem, the element of y should be float.
  - For classification problem, the element of y should be either 0 or 1. In multinomial regression, the p features are actually dummy variables.
  - For survival data, y should be a $n \times 2$ array, where the columns indicates "censoring" and "time", respectively.
- **is_normal** (*bool, optional, default=True*) -- whether normalize the variables array before fitting the algorithm.
- **sample_weight** (*array-like, shape (n_samples,), optional*) -- Individual weights for each sample. Only used for is_weight=True. Default=np.ones(n).

- **cv_fold_id** (*array-like, shape (n_samples,), optional, default=None*) -- An array indicates different folds in CV. Samples in the same fold should be given the same number.
- **sparse_matrix** (*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) -- If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** -- Parameter names mapped to their values.

Return type dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) -- Estimator parameters.

Returns **self** -- Estimator instance.

Return type estimator instance

CoxPHSurvivalAnalysis

Warning: In the old version of abess (before 0.4.0), this model is named *abess.linear.abessCox*. Please note that it will be deprecated in version 0.6.0.

class abess.linear.**CoxPHSurvivalAnalysis**(*path_type='seq', support_size=None, s_min=None, s_max=None, group=None, alpha=None, ic_type='ebic', ic_coef=1.0, cv=1, thread=1, A_init=None, always_select=None, max_iter=20, exchange_num=5, is_warm_start=True, splicing_type=0, important_search=128, screening_size=- 1, primary_model_fit_max_iter=10, primary_model_fit_epsilon=1e-08, approximate_Newton=False*)

Adaptive Best-Subset Selection (ABESS) algorithm for Cox proportional hazards model.

Parameters

- **path_type** (*{"seq", "gs"}, optional, default="seq"*) -- The method to be used to select the optimal support size.
  - For path_type = "seq", we solve the best subset selection problem for each size in support_size.
  - For path_type = "gs", we solve the best subset selection problem with support size ranged in (s_min, s_max), where the specific support size to be considered is determined by golden section.

- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes. Only used when path_type = "seq".
- **s_min** (*int, optional, default=0*) -- The lower bound of golden-section-search for sparsity searching.
- **s_max** (*int, optional, default=min(n, int(n/(log(log(n))log(p)))).*) -- The higher bound of golden-section-search for sparsity searching.
- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **alpha** (*float, optional, default=0*) -- Constant that multiples the L2 term in loss function, controlling regularization strength. It should be non-negative.
  - If alpha = 0, it indicates ordinary least square.
- **ic_type** (*{'aic', 'bic', 'gic', 'ebic'}, optional, default='ebic'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.
  - If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p))))$.
- **primary_model_fit_max_iter** (*int, optional, default=10*) -- The maximal number of iteration for primary_model_fit.

- **primary_model_fit_epsilon** (*float, optional, default=1e-08*) -- The epsilon (threshold) of iteration for primary_model_fit.
- **splicing_type** (*{0, 1}, optional, default=0*) -- The type of splicing: "0" for decreasing by half, "1" for decresing by one.
- **important_search** (*int, optional, default=128*) -- The size of inactive set during updating active set when splicing. It should be a non-positive integer and if important_search=0, it would be set as the size of whole inactive set.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.linear import CoxPHSurvivalAnalysis
>>> from abess.datasets import make_glm_data
>>> import numpy as np
>>> np.random.seed(12345)
>>> data = make_glm_data(n = 100, p = 50, k = 10, family = 'cox')
censoring rate:0.65
>>> model = CoxPHSurvivalAnalysis(support_size = 10)
>>> model.fit(data.x, data.y)
CoxPHSurvivalAnalysis(always_select=[], support_size=10)
>>> model.predict(data.x)[1:4]
array([1.08176927e+00, 6.37029117e-04, 3.64112556e-06, 4.09523406e+05])
```

```
>>> ### Sparsity unknown
>>>
>>> # path_type="seq"
>>> model = CoxPHSurvivalAnalysis(path_type = "seq")
>>> model.fit(data.x, data.y)
CoxPHSurvivalAnalysis(always_select=[], support_size=10)
>>> model.predict(data.x)[1:4]
array([1.08176927e+00, 6.37029117e-04, 3.64112556e-06, 4.09523406e+05])
>>>
>>> # path_type="gs"
>>> model = CoxPHSurvivalAnalysis(path_type="gs")
>>> model.fit(data.x, data.y)
CoxPHSurvivalAnalysis(always_select=[], path_type='gs')
>>> model.predict(data.x)[1:4]
array([1.07629689e+00, 6.47263126e-04, 4.30660826e-06, 3.66389638e+05])
```

coef_

Estimated coefficients for the best subset selection problem.

Type array-like, shape(p_features,) or (p_features, M_responses)

intercept_

The intercept in the model.

Type float or array-like, shape(M_responses,)

train_loss_

The loss on training data.

Type float

eval_loss_

- If cv=1, it stores the score under chosen information criterion.
- If cv>1, it stores the test loss under cross-validation.

Type float

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

predict(*X*)

Returns the time-independent part of hazard function, i.e. $\exp(X\beta)$ on given data.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **y** -- Return $\exp(X\beta)$.

Return type array-like, shape(n_samples,)

score(*X*, *y*, *sample_weight=None*)

Give data, and it returns C-index.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix.
- **y** (*array-like, shape(n_samples, p_features)*) -- Real response for given X.
- **sample_weight** (*array-like, shape(n_samples,), default=None*) -- Sample weights.

Returns **score** -- C-index.

Return type float

predict_survival_function(*X*)

Predict survival function. The survival function for an individual with feature vector x is defined as

$$S(t \mid x) = S_0(t)^{\exp(x^\top \beta)},$$

where $S_0(t)$ is the baseline survival function, estimated by Breslow's estimator.

Parameters **X** (*array-like, shape = (n_samples, n_features)*) -- Data matrix.

Returns **survival** -- Predicted survival functions.

Return type ndarray of StepFunction, shape = (n_samples,)

fit(*X=None*, *y=None*, *is_normal=True*, *sample_weight=None*, *cv_fold_id=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like of shape(n_samples, p_features)*) -- Training data matrix. It should be a numpy array.
- **y** (*array-like of shape(n_samples,) or (n_samples, M_responses)*) -- Training response values. It should be a numpy array.

- For regression problem, the element of y should be float.
- For classification problem, the element of y should be either 0 or 1. In multinomial regression, the p features are actually dummy variables.
- For survival data, y should be a $n \times 2$ array, where the columns indicates "censoring" and "time", respectively.

- **is_normal** (*bool, optional, default=True*) -- whether normalize the variables array before fitting the algorithm.
- **sample_weight** (*array-like, shape (n_samples,), optional*) -- Individual weights for each sample. Only used for is_weight=True. Default=np.ones(n).
- **cv_fold_id** (*array-like, shape (n_samples,), optional, default=None*) -- An array indicates different folds in CV. Samples in the same fold should be given the same number.
- **sparse_matrix** (*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) -- If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** -- Parameter names mapped to their values.

Return type dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) -- Estimator parameters.

Returns **self** -- Estimator instance.

Return type estimator instance

MultiTaskRegression

Warning: In the old version of abess (before 0.4.0), this model is named *abess.linear.abessMultigaussian*. Please note that it will be deprecated in version 0.6.0.

class abess.linear.**MultiTaskRegression**(*path_type='seq'*, *support_size=None*, *s_min=None*, *s_max=None*, *group=None*, *alpha=None*, *ic_type='ebic'*, *ic_coef=1.0*, *cv=1*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=0*, *important_search=128*, *screening_size=- 1*, *covariance_update=False*)

Adaptive Best-Subset Selection(ABESS) algorithm for multitasklearning.

Parameters

- **path_type** (*{"seq", "gs"}, optional, default="seq"*) -- The method to be used to select the optimal support size.
  - For path_type = "seq", we solve the best subset selection problem for each size in support_size.
  - For path_type = "gs", we solve the best subset selection problem with support size ranged in (s_min, s_max), where the specific support size to be considered is determined by golden section.
- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes. Only used when path_type = "seq".
- **s_min** (*int, optional, default=0*) -- The lower bound of golden-section-search for sparsity searching.
- **s_max** (*int, optional, default=min(n, int(n/(log(log(n))log(p))))*.) -- The higher bound of golden-section-search for sparsity searching.
- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **alpha** (*float, optional, default=0*) -- Constant that multiples the L2 term in loss function, controlling regularization strength. It should be non-negative.
  - If alpha = 0, it indicates ordinary least square.
- **ic_type** (*{'aic', 'bic', 'gic', 'ebic'}, optional, default='ebic'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.

- If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p)))))$.

- **primary_model_fit_max_iter** (*int, optional, default=10*) -- The maximal number of iteration for primary_model_fit.
- **primary_model_fit_epsilon** (*float, optional, default=1e-08*) -- The epsilon (threshold) of iteration for primary_model_fit.
- **splicing_type** (*{0, 1}, optional, default=0*) -- The type of splicing: "0" for decreasing by half, "1" for decresing by one.
- **important_search** (*int, optional, default=128*) -- The size of inactive set during updating active set when splicing. It should be a non-positive integer and if important_search=0, it would be set as the size of whole inactive set.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.linear import MultiTaskRegression
>>> from abess.datasets import make_multivariate_glm_data
>>> import numpy as np
>>> np.random.seed(12345)
>>> data = make_multivariate_glm_data(
>>>     n = 100, p = 50, k = 10, M = 3, family = 'multigaussian')
>>> model = MultiTaskRegression(support_size = 10)
>>> model.fit(data.x, data.y)
MultiTaskRegression(always_select=[], support_size=10)
>>> model.predict(data.x)[1:5, ]
array([[1., 0., 0.],
   [0., 0., 1.],
   [1., 0., 0.],
   [1., 0., 0.],
   [0., 0., 1.]])
```

```
>>> ### Sparsity unknown
>>>
>>> # path_type="seq"
>>> model = MultiTaskRegression(path_type = "seq")
>>> model.fit(data.x, data.y)
MultiTaskRegression(always_select=[])
>>> model.predict(data.x)[1:5, ]
array([[1., 0., 0.],
   [0., 0., 1.],
   [1., 0., 0.],
   [1., 0., 0.],
   [0., 0., 1.]])
>>>
>>> # path_type="gs"
>>> model = MultiTaskRegression(path_type="gs")
```

(continues on next page)

(continued from previous page)

```
>>> model.fit(data.x, data.y)
MultiTaskRegression(always_select=[], path_type='gs')
>>> model.predict(data.x)[1:5, ]
array([[1., 0., 0.],
   [0., 0., 1.],
   [1., 0., 0.],
   [1., 0., 0.],
   [0., 0., 1.]])
```

coef_

Estimated coefficients for the best subset selection problem.

Type array-like, shape(p_features,) or (p_features, M_responses)

intercept_

The intercept in the model.

Type float or array-like, shape(M_responses,)

train_loss_

The loss on training data.

Type float

eval_loss_

- If cv=1, it stores the score under chosen information criterion.
- If cv>1, it stores the test loss under cross-validation.

Type float

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

predict(*X*)

Prediction of the mean of each response on given data.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **y** -- Prediction of the mean of each response on given X. Each column indicates one response.

Return type array-like, shape(n_samples, M_responses)

score(*X*, *y*, *sample_weight=None*)

Give data, and it returns the coefficient of determination.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix.
- **y** (*array-like, shape(n_samples, M_responses)*) -- Real responses for given X.
- **sample_weight** (*array-like, shape(n_samples,), default=None*) -- Sample weights.

Returns **score** -- R^2 score.

Return type float

fit(*X=None*, *y=None*, *is_normal=True*, *sample_weight=None*, *cv_fold_id=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like of shape(n_samples, p_features)*) -- Training data matrix. It should be a numpy array.
- **y** (*array-like of shape(n_samples,) or (n_samples, M_responses)*) -- Training response values. It should be a numpy array.
  - For regression problem, the element of y should be float.
  - For classification problem, the element of y should be either 0 or 1. In multinomial regression, the p features are actually dummy variables.
  - For survival data, y should be a $n \times 2$ array, where the columns indicates "censoring" and "time", respectively.
- **is_normal** (*bool, optional, default=True*) -- whether normalize the variables array before fitting the algorithm.
- **sample_weight** (*array-like, shape (n_samples,), optional*) -- Individual weights for each sample. Only used for is_weight=True. Default=np.ones(n).
- **cv_fold_id** (*array-like, shape (n_samples,), optional, default=None*) -- An array indicates different folds in CV. Samples in the same fold should be given the same number.
- **sparse_matrix**(*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) -- If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** -- Parameter names mapped to their values.

Return type dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) -- Estimator parameters.

Returns **self** -- Estimator instance.

Return type estimator instance

MultinomialRegression

Warning: In the old version of abess (before 0.4.0), this model is named *abess.linear.abessMultinomial*. Please note that it will be deprecated in version 0.6.0.

class abess.linear.**MultinomialRegression**(*path_type='seq'*, *support_size=None*, *s_min=None*, *s_max=None*, *group=None*, *alpha=None*, *ic_type='ebic'*, *ic_coef=1.0*, *cv=1*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=0*, *important_search=128*, *screening_size=- 1*, *primary_model_fit_max_iter=10*, *primary_model_fit_epsilon=1e-08*)

Adaptive Best-Subset Selection(ABESS) algorithm for multiclassification problem.

Parameters

- **path_type** (*{"seq", "gs"}, optional, default="seq"*) -- The method to be used to select the optimal support size.
  - For path_type = "seq", we solve the best subset selection problem for each size in support_size.
  - For path_type = "gs", we solve the best subset selection problem with support size ranged in (s_min, s_max), where the specific support size to be considered is determined by golden section.
- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes. Only used when path_type = "seq".
- **s_min** (*int, optional, default=0*) -- The lower bound of golden-section-search for sparsity searching.
- **s_max** (*int, optional, default=min(n, int(n/(log(log(n))log(p))))*.) -- The higher bound of golden-section-search for sparsity searching.
- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **alpha** (*float, optional, default=0*) -- Constant that multiples the L2 term in loss function, controlling regularization strength. It should be non-negative.
  - If alpha = 0, it indicates ordinary least square.
- **ic_type** (*{'aic', 'bic', 'gic', 'ebic'}, optional, default='ebic'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.

- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.
  - If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p))))$.
- **primary_model_fit_max_iter** (*int, optional, default=10*) -- The maximal number of iteration for primary_model_fit.
- **primary_model_fit_epsilon** (*float, optional, default=1e-08*) -- The epsilon (threshold) of iteration for primary_model_fit.
- **splicing_type** (*{0, 1}, optional, default=0*) -- The type of splicing: "0" for decreasing by half, "1" for decresing by one.
- **important_search** (*int, optional, default=128*) -- The size of inactive set during updating active set when splicing. It should be a non-positive integer and if important_search=0, it would be set as the size of whole inactive set.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.linear import MultinomialRegression
>>> from abess.datasets import make_multivariate_glm_data
>>> import numpy as np
>>> np.random.seed(12345)
>>> data = make_multivariate_glm_data(
>>>     n = 100, p = 50, k = 10, M = 3, family = 'multinomial')
>>> model = MultinomialRegression(support_size = 10)
>>> model.fit(data.x, data.y)
MultinomialRegression(always_select=[], support_size=10)
>>> model.predict(data.x)[0:10, ]
array([1, 0, 0, 0, 1, 1, 1, 2, 1, 2])
```

```
>>> ### Sparsity unknown
>>>
>>> # path_type="seq"
>>> model = MultinomialRegression(path_type = "seq")
>>> model.fit(data.x, data.y)
MultinomialRegression(always_select=[])
>>> model.predict(data.x)[0:10, ]
array([1, 2, 0, 0, 1, 1, 1, 2, 1, 2])
```

```
>>>
>>> # path_type="gs"
>>> model = MultinomialRegression(path_type="gs")
>>> model.fit(data.x, data.y)
MultinomialRegression(always_select=[], path_type='gs')
>>> model.predict(data.x)[0:10, ]
array([1, 2, 0, 0, 1, 1, 1, 2, 1, 2])
```

coef_

Estimated coefficients for the best subset selection problem.

Type array-like, shape(p_features,) or (p_features, M_responses)

intercept_

The intercept in the model.

Type float or array-like, shape(M_responses,)

train_loss_

The loss on training data.

Type float

eval_loss_

- If cv=1, it stores the score under chosen information criterion.
- If cv>1, it stores the test loss under cross-validation.

Type float

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

predict_proba(*X*)

Give the probabilities of new data being assigned to different classes.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **proba** -- Returns the probability of given samples for each class. Each column indicates one class.

Return type array-like, shape(n_samples, M_responses)

`predict`(*X*)

Return the most possible class for given data.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **y** -- Predicted class label for each sample in X.

Return type array-like, shape(n_samples,)

`score`(*X*, *y*, *sample_weight=None*)

Give new data, and it returns the prediction accuracy.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Test data.
- **y** (*array-like, shape(n_samples, M_responses)*) -- Test response (dummy variables of real class).
- **sample_weight** (*array-like, shape(n_samples,), default=None*) -- Sample weights.

Returns **score** -- the mean prediction accuracy.

Return type float

`fit`(*X=None*, *y=None*, *is_normal=True*, *sample_weight=None*, *cv_fold_id=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like of shape(n_samples, p_features)*) -- Training data matrix. It should be a numpy array.
- **y** (*array-like of shape(n_samples,) or (n_samples, M_responses)*) -- Training response values. It should be a numpy array.
  - For regression problem, the element of y should be float.
  - For classification problem, the element of y should be either 0 or 1. In multinomial regression, the p features are actually dummy variables.
  - For survival data, y should be a $n \times 2$ array, where the columns indicates "censoring" and "time", respectively.
- **is_normal** (*bool, optional, default=True*) -- whether normalize the variables array before fitting the algorithm.
- **sample_weight** (*array-like, shape (n_samples,), optional*) -- Individual weights for each sample. Only used for is_weight=True. Default=np.ones(n).
- **cv_fold_id** (*array-like, shape (n_samples,), optional, default=None*) -- An array indicates different folds in CV. Samples in the same fold should be given the same number.
- **sparse_matrix**(*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) -- If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** -- Parameter names mapped to their values.

Return type dict

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) -- Estimator parameters.

Returns **self** -- Estimator instance.

Return type estimator instance

GammaRegression

Warning: In the old version of abess (before 0.4.0), this model is named *abess.linear.abessGamma*. Please note that it will be deprecated in version 0.6.0.

`class` abess.linear.`GammaRegression`(*path_type='seq'*, *support_size=None*, *s_min=None*, *s_max=None*, *group=None*, *alpha=None*, *ic_type='ebic'*, *ic_coef=1.0*, *cv=1*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=0*, *important_search=128*, *screening_size=- 1*, *primary_model_fit_max_iter=10*, *primary_model_fit_epsilon=1e-08*, *approximate_Newton=False*)

Adaptive Best-Subset Selection(ABESS) algorithm for Gamma regression.

Parameters

- **path_type** (*{"seq", "gs"}, optional, default="seq"*) -- The method to be used to select the optimal support size.
  - For path_type = "seq", we solve the best subset selection problem for each size in support_size.
  - For path_type = "gs", we solve the best subset selection problem with support size ranged in (s_min, s_max), where the specific support size to be considered is determined by golden section.
- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes. Only used when path_type = "seq".
- **s_min** (*int, optional, default=0*) -- The lower bound of golden-section-search for sparsity searching.
- **s_max** (*int, optional, default=min(n, int(n/(log(log(n))log(p))))*.) -- The higher bound of golden-section-search for sparsity searching.
- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **alpha** (*float, optional, default=0*) -- Constant that multiples the L2 term in loss function, controlling regularization strength. It should be non-negative.
  - If alpha = 0, it indicates ordinary least square.

- **ic_type** (*{'aic', 'bic', 'gic', 'ebic'}, optional, default='ebic'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.
  - If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p))))$.
- **primary_model_fit_max_iter** (*int, optional, default=10*) -- The maximal number of iteration for primary_model_fit.
- **primary_model_fit_epsilon** (*float, optional, default=1e-08*) -- The epsilon (threshold) of iteration for primary_model_fit.
- **splicing_type** (*{0, 1}, optional, default=0*) -- The type of splicing: "0" for decreasing by half, "1" for decresing by one.
- **important_search** (*int, optional, default=128*) -- The size of inactive set during updating active set when splicing. It should be a non-positive integer and if important_search=0, it would be set as the size of whole inactive set.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.linear import GammaRegression
>>> from abess.datasets import make_glm_data
>>> import numpy as np
>>> np.random.seed(12345)
>>> data = make_glm_data(n = 100, p = 50, k = 10, family = 'gamma')
>>> model = GammaRegression(support_size = 10)
>>> model.fit(data.x, data.y)
GammaRegression(always_select=[], support_size=10)
>>> model.predict(data.x)[1:4]
array([1.34510045e+22, 2.34908508e+30, 1.91570199e+21, 1.29563315e+25])
```

```
>>> ### Sparsity unknown
>>>
>>> # path_type="seq"
>>> model = GammaRegression(path_type = "seq")
>>> model.fit(data.x, data.y)
GammaRegression(always_select=[])
>>> model.predict(data.x)[1:4]
array([7.03065424e+19, 7.03065424e+19, 7.03065424e+19, 7.03065424e+19])
>>>
>>> # path_type="gs"
>>> model = GammaRegression(path_type="gs")
>>> model.fit(data.x, data.y)
GammaRegression(always_select=[], path_type='gs')
>>> model.predict(data.x)[1:4]
array([7.03065424e+19, 7.03065424e+19, 7.03065424e+19, 7.03065424e+19])
```

coef_

Estimated coefficients for the best subset selection problem.

Type array-like, shape(p_features,) or (p_features, M_responses)

intercept_

The intercept in the model.

Type float or array-like, shape(M_responses,)

train_loss_

The loss on training data.

Type float

eval_loss_

- If cv=1, it stores the score under chosen information criterion.
- If cv>1, it stores the test loss under cross-validation.

Type float

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

predict(*X*)

Predict on given data.

Parameters **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix to be predicted.

Returns **y** -- Prediction of the mean on given X.

Return type array-like, shape(n_samples,)

score(*X*, *y*, *sample_weight=None*)

Give new data, and it returns the prediction error.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Sample matrix.
- **y** (*array-like, shape(n_samples, p_features)*) -- Real response for given X.
- **sample_weight** (*array-like, shape(n_samples,), default=None*) -- Sample weights.

Returns **score** -- Prediction error.

Return type float

fit(*X=None*, *y=None*, *is_normal=True*, *sample_weight=None*, *cv_fold_id=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like of shape(n_samples, p_features)*) -- Training data matrix. It should be a numpy array.
- **y** (*array-like of shape(n_samples,) or (n_samples, M_responses)*) -- Training response values. It should be a numpy array.
  - For regression problem, the element of y should be float.
  - For classification problem, the element of y should be either 0 or 1. In multinomial regression, the p features are actually dummy variables.
  - For survival data, y should be a $n \times 2$ array, where the columns indicates "censoring" and "time", respectively.
- **is_normal** (*bool, optional, default=True*) -- whether normalize the variables array before fitting the algorithm.
- **sample_weight** (*array-like, shape (n_samples,), optional*) -- Individual weights for each sample. Only used for is_weight=True. Default=np.ones(n).
- **cv_fold_id** (*array-like, shape (n_samples,), optional, default=None*) -- An array indicates different folds in CV. Samples in the same fold should be given the same number.
- **sparse_matrix**(*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) -- If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** -- Parameter names mapped to their values.

Return type dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) -- Estimator parameters.

Returns **self** -- Estimator instance.

Return type estimator instance

OrdinalRegression

class abess.linear.**OrdinalRegression**(*path_type='seq'*, *support_size=None*, *s_min=None*, *s_max=None*, *group=None*, *alpha=None*, *ic_type='ebic'*, *ic_coef=1.0*, *cv=1*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=0*, *important_search=128*, *screening_size=- 1*, *primary_model_fit_max_iter=10*, *primary_model_fit_epsilon=1e-08*, *approximate_Newton=False*)

Adaptive Best-Subset Selection(ABESS) algorithm for ordinal regression problem.

Parameters

- **path_type** (*{"seq", "gs"}, optional, default="seq"*) -- The method to be used to select the optimal support size.
  - For path_type = "seq", we solve the best subset selection problem for each size in support_size.
  - For path_type = "gs", we solve the best subset selection problem with support size ranged in (s_min, s_max), where the specific support size to be considered is determined by golden section.
- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes. Only used when path_type = "seq".
- **s_min** (*int, optional, default=0*) -- The lower bound of golden-section-search for sparsity searching.
- **s_max** (*int, optional, default=min(n, int(n/(log(log(n))log(p))))*.) -- The higher bound of golden-section-search for sparsity searching.
- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **alpha** (*float, optional, default=0*) -- Constant that multiples the L2 term in loss function, controlling regularization strength. It should be non-negative.

  - If alpha = 0, it indicates ordinary least square.

- **ic_type** (*{'aic', 'bic', 'gic', 'ebic'}, optional, default='ebic'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.
  - If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p)))))$.
- **primary_model_fit_max_iter** (*int, optional, default=10*) -- The maximal number of iteration for primary_model_fit.
- **primary_model_fit_epsilon** (*float, optional, default=1e-08*) -- The epsilon (threshold) of iteration for primary_model_fit.
- **splicing_type** (*{0, 1}, optional, default=0*) -- The type of splicing: "0" for decreasing by half, "1" for decresing by one.
- **important_search** (*int, optional, default=128*) -- The size of inactive set during updating active set when splicing. It should be a non-positive integer and if important_search=0, it would be set as the size of whole inactive set.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.linear import OrdinalRegression
>>> from abess.datasets import make_glm_data
>>> import numpy as np
>>> np.random.seed(12345)
>>> data = make_glm_data(n = 1000, p = 50, k = 10, family = 'ordinal')
>>> print((np.nonzero(data.coef_)[0]))
[ 0  4 10 14 26 29 34 38 47 48]
>>> model = OrdinalRegression(support_size = 10)
>>> model.fit(data.x, data.y)
classes: [0. 1. 2.]
OrdinalRegression(support_size=10)
>>> print((np.nonzero(model.coef_)[0]))
[ 0  4 10 14 26 29 38 40 47 48]
```

```
>>> ### Sparsity unknown
>>>
>>> # path_type="seq"
>>> model = OrdinalRegression(path_type = "seq")
>>> model.fit(data.x, data.y)
classes: [0. 1. 2.]
OrdinalRegression()
>>> print((np.nonzero(model.coef_)[0]))
[ 0  4  8 10 14 26 29 38 40 47 48]
>>>
>>> # path_type="gs"
>>> model = OrdinalRegression(path_type="gs")
>>> model.fit(data.x, data.y)
classes: [0. 1. 2.]
OrdinalRegression(path_type='gs')
>>> print((np.nonzero(model.coef_)[0]))
[ 0  4 10 14 26 29 38 47 48]
```

coef_

Estimated coefficients for the best subset selection problem.

Type array-like, shape(p_features,) or (p_features, M_responses)

intercept_

The intercept in the model.

Type float or array-like, shape(M_responses,)

train_loss_

The loss on training data.

Type float

eval_loss_

- If cv=1, it stores the score under chosen information criterion.
- If cv>1, it stores the test loss under cross-validation.

Type float

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

fit(*X=None*, *y=None*, *is_normal=True*, *sample_weight=None*, *cv_fold_id=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like of shape(n_samples, p_features)*) -- Training data matrix. It should be a numpy array.
- **y** (*array-like of shape(n_samples,) or (n_samples, M_responses)*) -- Training response values. It should be a numpy array.
  - For regression problem, the element of y should be float.
  - For classification problem, the element of y should be either 0 or 1. In multinomial regression, the p features are actually dummy variables.
  - For survival data, y should be a $n \times 2$ array, where the columns indicates "censoring" and "time", respectively.
- **is_normal** (*bool, optional, default=True*) -- whether normalize the variables array before fitting the algorithm.
- **sample_weight** (*array-like, shape (n_samples,), optional*) -- Individual weights for each sample. Only used for is_weight=True. Default=np.ones(n).
- **cv_fold_id** (*array-like, shape (n_samples,), optional, default=None*) -- An array indicates different folds in CV. Samples in the same fold should be given the same number.
- **sparse_matrix**(*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) -- If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** -- Parameter names mapped to their values.

Return type dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) -- Estimator parameters.

Returns **self** -- Estimator instance.

Return type estimator instance

7.3.2 Principal Component Analysis (PCA)

This page contains links to `abess.decomposition` module and its functions. It is designed for the best-subset selection for PCA problem.

> **Warning:** In the old version of abess (before 0.4.0), this model is named `abess.pca`. Please note that it will be deprecated in version 0.6.0.

SparsePCA

> **Warning:** In the old version of abess (before 0.4.0), this model is named `abess.pca.abessPCA`. Please note that it will be deprecated in version 0.6.0.

class abess.decomposition.**SparsePCA**(*support_size=None*, *group=None*, *ic_type='loss'*, *ic_coef=1.0*, *cv=1*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=1*, *screening_size=- 1*)

Adaptive Best-Subset Selection(ABESS) algorithm for principal component analysis.

Parameters

- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes.
- **group** (*int, optional, default=np.ones(p)*) -- The group index for each variable.
- **ic_type** (*{'aic', 'bic', 'gic', 'ebic', 'loss'}, optional, default='loss'*) -- The type of criterion for choosing the support size if *cv=1*.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **cv** (*int, optional, default=1*) -- The folds number when use the cross-validation method.
  - If cv=1, cross-validation would not be used.
  - If cv>1, support size will be chosen by CV's test loss, instead of IC.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.

- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **screening_size** (*int, optional, default=-1*) -- The number of variables remaining after screening. It should be a non-negative number smaller than p, but larger than any value in support_size.
  - If screening_size=-1, screening will not be used.
  - If screening_size=0, screening_size will be set as $min(p, int(n/($ $log($ $log(n))$ $log(p))))$.
- **splicing_type** (*{0, 1}, optional, default=1*) -- The type of splicing. "0" for decreasing by half, "1" for decresing by one.

coef_

The first k principal axes in feature space, which are sorted by decreasing explained variance.

Type array-like, shape(p_features,) or (p_features, k)

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.decomposition import SparsePCA
>>> import numpy as np
>>> np.random.seed(12345)
>>> model = SparsePCA(support_size = 10)
>>>
>>> ### X known
>>> X = np.random.randn(100, 50)
>>> model.fit(X)
SparsePCA(always_select=[], support_size=10)
>>> # print(model.coef_)
>>> print(model.coef_[1:6,])
[[6.36598737e-314]
 [1.06099790e-313]
 [1.48539705e-313]
 [1.90979621e-313]
 [2.33419537e-313]]
>>>
>>> ### X unknown, but Sigma known
>>> model.fit(Sigma = np.cov(X.T))
SparsePCA(always_select=[], support_size=10)
>>> # print(model.coef_)
```

(continues on next page)

(continued from previous page)

```
>>> print(model.coef_[1:6,])
[[6.36598737e-314]
 [1.06099790e-313]
 [1.48539705e-313]
 [1.90979621e-313]
 [2.33419537e-313]]
```

transform(*X*)

For PCA model, apply dimensionality reduction to given data.

Parameters **X** (*array-like, shape (n_samples, p_features)*) -- Sample matrix to be transformed.

ratio(*X*)

Give new data, and it returns the explained ratio.

Parameters **X** (*array-like, shape (n_samples, n_features)*) -- Sample matrix.

fit(*X=None*, *y=None*, *is_normal=False*, *Sigma=None*, *number=1*, *n=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Training data.
- **y** (*ignore*) -- Ignore.
- **is_normal** (*bool, optional, default=False*) -- whether normalize the variables array before fitting the algorithm.
- **weight** (*array-like, shape(n_samples,), optional, default=np.ones(n)*) -- Individual weights for each sample. Only used for is_weight=True.
- **Sigma** (*array-like, shape(p_features, p_features), optional*) -- default=np.cov(X.T). Sample covariance matrix. For PCA, it can be given as input, instead of X. But if X is given, Sigma will be set to np.cov(X.T).
- **number** (*int, optional, default=1*) -- Indicates the number of PCs returned.
- **n** (*int, optional, default=X.shape[0] or 1*) -- Sample size.
  - if X is given, it would be X.shape[0] by default;
  - if X is not given (Sigma is given), it would be 1 by default.
- **sparse_matrix** (*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

fit_transform(*X=None*, *y=None*, *is_normal=False*, *Sigma=None*, *number=1*, *n=None*, *sparse_matrix=False*)

Fit and transform the sample matrix. Returns transformed data in expected dimension.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Training data.
- **y** (*ignore*) -- Ignore.
- **is_normal** (*bool, optional, default=False*) -- whether normalize the variables array before fitting the algorithm.

- **weight** (*array-like, shape(n_samples,), optional, default=np.ones(n)*) -- Individual weights for each sample. Only used for is_weight=True.
- **Sigma** (*array-like, shape(p_features, p_features), optional*) -- default=np.cov(X.T). Sample covariance matrix. For PCA, it can be given as input, instead of X. But if X is given, Sigma will be set to np.cov(X.T).
- **number** (*int, optional, default=1*) -- Indicates the number of PCs returned.
- **n** (*int, optional, default=X.shape[0] or 1*) -- Sample size.
  - if X is given, it would be X.shape[0] by default;
  - if X is not given (Sigma is given), it would be 1 by default.

RobustPCA

Warning: In the old version of abess (before 0.4.0), this class is named `abess.pca.abessRPCA`. Please note that it will be deprecated in version 0.6.0.

class abess.decomposition.**RobustPCA**(*support_size=None*, *ic_type='gic'*, *ic_coef=1.0*, *thread=1*, *A_init=None*, *always_select=None*, *max_iter=20*, *exchange_num=5*, *is_warm_start=True*, *splicing_type=1*)

Adaptive Best-Subset Selection(ABESS) algorithm for robust principal component analysis.

Parameters

- **support_size** (*array-like, optional*) -- default=range(min(n, int(n/(log(log(n))log(p))))). An integer vector representing the alternative support sizes.
- **ic_type** (*{'aic', 'bic', 'gic', 'ebic', 'loss'}, optional, default='gic'*) -- The type of criterion for choosing the support size.
- **ic_coef** (*float, optional, default=1.0*) -- Constant that controls the regularization strength on chosen information criterion.
- **thread** (*int, optional, default=1*) -- Max number of multithreads.
  - If thread = 0, the maximum number of threads supported by the device will be used.
- **A_init** (*array-like, optional, default=None*) -- Initial active set before the first splicing.
- **always_select** (*array-like, optional, default=None*) -- An array contains the indexes of variables we want to consider in the model.
- **max_iter** (*int, optional, default=20*) -- Maximum number of iterations taken for the splicing algorithm to converge. Due to the limitation of loss reduction, the splicing algorithm must be able to converge. The number of iterations is only to simplify the implementation.
- **is_warm_start** (*bool, optional, default=True*) -- When tuning the optimal parameter combination, whether to use the last solution as a warm start to accelerate the iterative convergence of the splicing algorithm.
- **splicing_type** (*{0, 1}, optional, default=1*) -- The type of splicing. "0" for decreasing by half, "1" for decresing by one.

coef_

The transformed sample matrix after robust PCA.

Type array-like, shape(n_samples, p_features)

References

- Junxian Zhu, Canhong Wen, Jin Zhu, Heping Zhang, and Xueqin Wang. A polynomial algorithm for best-subset selection problem. Proceedings of the National Academy of Sciences, 117(52):33117-33123, 2020.

Examples

```
>>> ### Sparsity known
>>>
>>> from abess.decomposition import RobustPCA
>>> import numpy as np
>>> np.random.seed(12345)
>>> model = RobustPCA(support_size = 10)
>>>
>>> ### X known
>>> X = np.random.randn(100, 50)
>>> model.fit(X, r = 10)
RobustPCA(always_select=[], support_size=10)
>>> print(model.coef_)
[[0.         0.         0.         ... 0.         3.71203604 0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]
```

fit(*X*, *y=None*, *r=None*, *sparse_matrix=False*)

The fit function is used to transfer the information of data and return the fit result.

Parameters

- **X** (*array-like, shape(n_samples, p_features)*) -- Training data.
- **y** (*ignore*) -- Ignore.
- **r** (*int*) -- Rank of the (recovered) information matrix L. It should be smaller than rank of X (at least smaller than X.shape[1]).
- **sparse_matrix** (*bool, optional, default=False*) -- Set as True to treat X as sparse matrix during fitting. It would be automatically set as True when X has the sparse matrix type defined in scipy.sparse.

7.3.3 Data Generation

This page contains links to `abess.datasets` module and its functions. It is a convenient modules for generating data.

make_glm_data

class abess.datasets.**make_glm_data**(*n*, *p*, *k*, *family*, *rho=0*, *corr_type='const'*, *sigma=1*, *coef_=None*, *censoring=True*, *c=1*, *scal=10*, *snr=None*, *class_num=3*)

Generate a dataset with single response.

Parameters

- **n** (*int*) -- The number of observations.
- **p** (*int*) -- The number of predictors of interest.
- **k** (*int*) -- The number of nonzero coefficients in the underlying regression model.
- **family** (*{gaussian, binomial, poisson, gamma, cox}*) -- The distribution of the simulated response. "gaussian" for univariate quantitative response, "binomial" for binary classification response, "poisson" for counting response, "gamma" for positive continuous response, "cox" for left-censored response.
- **rho** (*float, optional, default=0*) -- A parameter used to characterize the pairwise correlation in predictors.
- **corr_type** (*string, optional, default="const"*) -- The structure of correlation matrix. "const" for constant pairwise correlation, "exp" for pairwise correlation with exponential decay.
- **sigma** (*float, optional, default=1*) -- The variance of the gaussian noise. It would be unused if snr is not None.
- **coef** (*array_like, optional, default=None*) -- The coefficient values in the underlying regression model.
- **censoring** (*bool, optional, default=True*) -- For Cox data, it indicates whether censoring is existed.
- **c** (*int, optional, default=1*) -- For Cox data and censoring=True, it indicates the maximum censoring time. So that all observations have chances to be censored at (0, c).
- **scal** (*float, optional, default=10*) -- The scale of survival time in Cox data.
- **snr** (*float, optional, default=None*) -- A numerical value controlling the signal-to-noise ratio (SNR) in gaussian data.
- **class_num** (*int, optional, default=3*) -- The number of possible classes in oridinal dataset, i.e. $y \in \{0, 1, 2, ...,$

x

Design matrix of predictors.

Type array-like, shape(n, p)

y

Response variable.

Type array-like, shape(n,)

`coef_`

The coefficients used in the underlying regression model. It has k nonzero values.

Type array-like, shape(p,)

Notes

The output, whose type is named `data`, contains three elements: `x`, `y` and `coef_`, which correspond the variables, responses and coefficients, respectively.

Each row of `x` or `y` indicates a sample and is independent to the other.

We denote x, y, β for one sample in the math formulas below.

- Linear Regression
  - Usage: `family='gaussian'[, sigma=...]`
  - Model: $y \sim N(\mu, \sigma^2),\ \mu = x^T\beta$.
    - the coefficient $\beta \sim U[m, 100m]$, where $m = 5\sqrt{2\log p/n}$;
    - the variance $\sigma = 1$.
- Logistic Regression
  - Usage: `family='binomial'`
  - Model: $y \sim \text{Binom}(\pi),\ \text{logit}(\pi) = x^T\beta$.
    - the coefficient $\beta \sim U[2m, 10m]$, where $m = 5\sqrt{2\log p/n}$.
- Poisson Regression
  - Usage: `family='poisson'`
  - Model: $y \sim \text{Poisson}(\lambda),\ \lambda = \exp(x^T\beta)$.
    - the coefficient $\beta \sim U[2m, 10m]$, where $m = 5\sqrt{2\log p/n}$.
- Gamma Regression
  - Usage: `family='gamma'`
  - Model: $y \sim \text{Gamma}(k, \theta),\ k\theta = \exp(x^T\beta + \epsilon), k \sim U[0.1, 100.1]$ in shape-scale definition.
    - the coefficient $\beta \sim U[m, 100m]$, where $m = 5\sqrt{2\log p/n}$.
- Cox PH Survival Analysis
  - Usage: `family='cox'[, scal=..., censoring=..., c=...]`
  - Model: $y = \min(t, C)$, where $t = \left[-\dfrac{\log U}{\exp(X\beta)}\right]^s,\ U \sim N(0,1),\ s = \dfrac{1}{\text{scal}}$ and censoring time $C \sim U(0, c)$.
    - the coefficient $\beta \sim U[2m, 10m]$, where $m = 5\sqrt{2\log p/n}$;
    - the scale of survival time scal $= 10$;
    - censoring is enabled, and max censoring time $c = 1$.
- Ordinal Regression
  - Usage: `family='ordinal'[, class_num=...]`

- Model: $y \in \{0, 1, \ldots, n_{class}\}$, $\mathbb{P}(y \leq i) = \dfrac{1}{1 + \exp(-x^T\beta - \varepsilon_i)}$, where $i \in \{0, 1, \ldots, n_{class}\}$ and $\forall i < j, \varepsilon_i < \varepsilon_j$.
  - the coefficient $\beta \sim U[-M, M]$, where $M = 125\sqrt{2 \log p/n}$;
  - the intercept: $\forall i, \varepsilon_i \sim U[-M, M]$;
  - the number of classes $n_{class} = 3$.

make_multivariate_glm_data

class abess.datasets.**make_multivariate_glm_data**(*n=100*, *p=100*, *k=10*, *family='multigaussian'*, *rho=0.5*, *corr_type='const'*, *coef_=None*, *M=1*, *sparse_ratio=None*)

Generate a dataset with multi-responses.

Parameters

- **n** (*int, optional, default=100*) -- The number of observations.
- **p** (*int, optional, default=100*) -- The number of predictors of interest.
- **family** (*{multigaussian, multinomial, poisson}, optional*) -- default="multigaussian". The distribution of the simulated multi-response. "multigaussian" for multivariate quantitative responses, "multinomial" for multiple classification responses, "poisson" for counting responses.
- **k** (*int, optional, default=10*) -- The number of nonzero coefficients in the underlying regression model.
- **M** (*int, optional, default=1*) -- The number of responses.
- **rho** (*float, optional, default=0.5*) -- A parameter used to characterize the pairwise correlation in predictors.
- **corr_type** (*string, optional, default="const"*) -- The structure of correlation matrix. "const" for constant pairwise correlation, "exp" for pairwise correlation with exponential decay.
- **coef** (*array_like, optional, default=None*) -- The coefficient values in the underlying regression model.
- **sparse_ratio** (*float, optional, default=None*) -- The sparse ratio of predictor matrix (x).

x

Design matrix of predictors.

Type array-like, shape(n, p)

y

Response variable.

Type array-like, shape(n, M)

coef_

The coefficients used in the underlying regression model. It is rowwise sparse, with k nonzero rows.

Type array-like, shape(p, M)

Notes

The output, whose type is named `data`, contains three elements: `x`, `y` and `coef_`, which correspond the variables, responses and coefficients, respectively.

Note that the `y` and `coef_` here are both matrix:

1. each row of `x` and `y` indicates a sample;
2. each column of `coef_` corresponds to the effect on one response. It is rowwise sparsity. Under this setting, a "useful" variable is relevant to all responses.

We x, y, β for one sample in the math formulas below.

- Multitask Regression
  - Usage: `family='multigaussian'`
  - Model: $y \sim MVN(\mu, \Sigma),\ \mu^T = x^T\beta$.
    - the variance $\Sigma = \text{diag}(1, 1, \cdots, 1)$;
    - the coefficient β contains 30% "strong" values, 40% "moderate" values and the rest are "weak". They come from $N(0, 10)$, $N(0, 5)$ and $N(0, 2)$, respectively.
- Multinomial Regression
  - Usage: `family='multinomial'`
  - Model: y is a "0-1" array with only one "1". Its index is chosed under probabilities $\pi = \exp(x^T\beta)$.
    - the coefficient β contains 30% "strong" values, 40% "moderate" values and the rest are "weak". They come from $N(0, 10)$, $N(0, 5)$ and $N(0, 2)$, respectively.

7.4 Contributing

Contributions are welcome! No matter your current skills, it's possible to make valuable contribution to the `abess`.

7.4.1 Bug Report or New Feature Request

Bugs Report

If you've found a bug about `abess`, please open an issue at github issues or send an email to Jin Zhu at zhuj37@mail2.sysu.edu.cn. When reporting a bug, please include:

- codes to reproduce the bug.
- your operating system and Python or R version.
- any details about your local setup that might be helpful in troubleshooting.

We strongly encourage to spend some time trying to make it as minimal as possible: the more time you spend doing this, the easier it will be for the abess-team to fix it.

Suggest New Features

If you're working on best subset selection for some problem that can not be handled by the `abess` library, it is encouraged to share your new features suggestion to us. You can open an issue at github issues to post your suggestion. When suggesting a new feature, please:

- explain in detail how it would work.
- keep the scope as narrow as possible, to make it easier to understand and implementation.
- provide few important literatures if possible.

7.4.2 Contribute documentation

General development procedure

If you're a more experienced with the `abess` and are looking forward to improve your open source development skills, the next step up is to contribute a pull request to a `abess` documentation.

In most of case, the workflow is given below. But if you are not familar with git and github, we suggest you install the github desktop that provide a user-friendly interaction interface for simplifying documentation contribution. You can fetch the documentation about Git here.

1. Fork the master repository by clicking on the "Fork" button on the top right of the page, which would create a copy to your own GitHub account;
2. Clone your fork of abess to the local by Git;

```
$ git clone https://github.com/YourAccount/abess.git
$ cd abess
```

3. Create a new branch, e.g. named `docsdev`, to hold your development changes:

```
$ git branch docsdev
$ git checkout docsdev
```

   Work on the `docsdev` branch for documentation development.

4. Commit your improvements and contribution to documentation (forming an ideally legible commit history):

```
$ git add changed_files
$ git commit -m "some commits"
$ git push
```

5. Merge your branch with the master branch that have the up-to-date codes in abess;
6. Submit a pull request explaining your contribution for documentation.

The online documentation for our project are generated by generation by Sphinx and sphinix-gallery for python and pkgdown for R. Our website is published via readthedocs.

Python document

The Python document includes two parts. The first part depicts the APIs in `abess`, and the second part aims to show simple use-cases and on-board new users. After describing the two parts, we detailly explain the procedure for *python document development*, which unfolds the Step 4 in *general development procedure*.

Python API

For the development of Python documentation, there is a little difference between a new method and a new function. A new method need a brief introduction and some examples, such as [link]; and a new function under should at least contain an introduction and the parameters it requires, such as [link]. Also note that the style of Python document is similar to numpydoc.

The development of Python API's documentation mainly relies on Sphinx, sphinx-gallery (support markdown for Sphinx), sphinx-rtd-theme (support "Read the Docs" theme for Sphinx) and so on.

Please make sure all packages in `docs/requirements.txt` have been installed.

Tutorials

A tutorial is a long-form guide to some essential functions in the `abess` package. We recommend to use a tutorial to:

- describes the problem that one function can solve;
- show readers how to solve with the function;
- give more details about the potential advanced usage.

A typical online vignette example is present [here].

The development of the tutorial relies on sphinix-gallery.

Document development

Before developing document, we presume that you have already complete the steps 1-3 described in *general development procedure*, and you have installed necessary packages, including: `sphinix-gallery`, `Sphinx`, `nbsphinx`, `myst-parser`, `sphinx-rtd-theme`.

There are five basic steps to write documentation for the Python document:

1. If you contributing a new document, please create a new `.rst` file in the `docs/Python-package` directory for describing APIs or `docs/Tutorial` for new Tutorials, and write the documentation in the file. If you would like to modify documents, please modify the corresponding `.rst` files in the `docs` directory.
2. Go to the `docs` directory (e.g., via `cd docs`), and convert `.rst` files to `.html` files by running the following command in the terminal:

   ```
   $ make html
   ```

   and preview new documentation by opening/refreshing the `index.html` files in `docs/_build/html` directory.
3. Repeat step 1 and step 2 until you are satisfied with the documentation.
4. If you use some packages in Pypi, please add these package into `docs/requirements.txt` (for example, the `geomstats` package) so that the servers provided by Readthedocs pre-install these packages.
5. Submit a pull request from the `docsdev` branch in your repository `YourAccount/abess` to the `master` branch in the repository `abess-team/abess`.

More advanced topics for writing documentation are available at: Sphinx.

R document

The R document includes two parts. The first part depicts the APIs in the *abess* R package, and the second part aims to show simple use-cases and on-board new users.

R function

For the development of R documentation, the most important thing to know is that the abess R package relies on roxygen2 package. This means that documentation is found in the R code close to the source of each function. Before writing the documentation, it would be better to ensure the usage of the Rd tags.

There are four basic steps to write documentation for the R function in abess:

1. Add comments to `R` files in `R-package/R` directory.
2. Run `devtools::document()` in R to convert roxygen comments to `.Rd` files.
3. Preview documentation with `?`.
4. Repeat steps 1-3 until you are satisfied with the documentation.

More advanced topics for writing object documentation are available at: https://r-pkgs.org/man.html.

Online vignette

The aim of a online R vignette is the same as a tutorial for Python package. A typical online vignette example is presented in this [link]. We strongly recommend to use R markdown (`.Rmd` files) to organize a online vignette.

There are also four steps to write online vignettes:

1. Add/modify to `.Rmd` files in `R-package/vignettes` directory.
2. Run `pkgdown::build_articles()` in R to convert `.Rmd` files to webpages. (Make sure the `pkgdown` R package has been installed.)
3. Preview the webpages.
4. Repeat steps 1-3 until you are satisfied with the vignettes.

You can learn many detail about `pkgdown` package and R markdown in pkgdown's website and Hadley's website, respectively.

7.4.3 Contribute Python/R code

If you are a experienced programmer, you might want to help new features development or bug fixing for the abess library. The preferred workflow for contributing code to abess is to fork the master repository on GitHub, clone, and develop on a branch:

1. Before contributing, you should always open an issue and make sure someone from the abess team agrees that your work is really contributive, and is happy with your proposal. We don't want you to spend a bunch of time on something that we are working on or we don't think is a good idea;
2. Fork the master repository by clicking on the "Fork" button on the top right of the page, which would create a copy to your own GitHub account. If you have forked abess repository, enter it and click "Fetch upsteam", followed by "Fetch and merge" to ensure the code is the latest one;

3. Clone your fork of abess to the local by Git;

```
$ git clone https://github.com/your_account_name/abess.git
$ cd abess
```

4. Create a new branch, e.g. named `my_branch`, to hold your development changes:

```
$ git branch my_branch
$ git checkout my_branch
```

   It is preferred to work on your own branch instead of the master one;

5. While developing code, make sure to read the abess style guide (PEP8 for Python, tidyverse for R) which will make sure that your new code and documentation matches the existing style. This makes the review process much smoother. For more details about code developing, read the Code Developing description for abess library;

6. After finishing the development and making sure it works well, you can push them onto your repository:

```
$ git add changed_files
$ git commit -m "some commits"
$ git push
```

7. Look back to GItHub, merge your branch with the master branch that have the up-to-date codes in abess; and click the "Contribute" button on your fork to open pull request. Now, we can receive your contribution.

7.4.4 Develop New Features

In this tutorial, we will show you how to develop a new algorithm for specific best-subset problem with `abess`'s procedure.

Preliminaries

We have endeavor to make developing new features easily. Before developing the code, please make sure you have:

- understood the ABESS algorithm;
- installed `abess` via the code in github by following Installation instruction;
- read the Appendix: Architecture of abess of `abess` library;
- some experience on writing R or Python code.

Core C++

The main files related to the core are in `src`, which are written in C++. Among them, some important files:

- `api.cpp/api.h` contain the API's, which are the entrance for both R & Python.
- `AlgorithmXXX.h` records the implement of each concrete algorithm;

If you want to add a new algorithm, all of them should be noticed.

Besides, we have implemented ABESS algorithms for generalized linear model on `src/AlgorithmGLM.h` [code temp] and principal component analysis (PCA) on `src/AlgorithmPCA.h` [code temp]. You can check them to help your own developing.

Write an API

API's are all defined in the `src/api.cpp`[code temp] and the related header file `src/api.h`[code temp]. We have written some API functions (e.g. `abessGLM_API()`), so you can either add a new function for the new algorithm or simply add into existing one.

First of all, the algorithm name and its number should be determined.

The format of a new algorithm's name is "**abess+your_algorithm**", which means that using abess to solve the problem, and its number should be an integer unused by others.

In the following part, we suppose to create an algorithm named `abess_new_algorithm` with number 123.

Next, four important data type should be determined:

- T1 : type of Y
- T2 : type of coefficients
- T3 : type of intercept
- T4 : type of X

The algorithm variable are based on them: [code link]

```
vector<Algorithm<{T1}, {T2}, {T3}, {T4}>*> algorithm_list(algorithm_list_size);
```

Take `LinearRegression` (the linear regression on abess) as an example,

- Y: dense vector
- Coefficients: dense vector
- Intercept: numeric
- X: dense/sparse matrix

so that we define:

```
vector<Algorithm<Eigen::VectorXd, Eigen::VectorXd, double, Eigen::MatrixXd> *> algorithm_
↪list_uni_dense(algorithm_list_size);
vector<Algorithm<Eigen::VectorXd, Eigen::VectorXd, double, Eigen::SparseMatrix<double>>␣
↪*> algorithm_list_uni_sparse(algorithm_list_size);
```

After that, request memory to initial the algorithm: [code link]

```
for (int i = 0; i < algorithm_list_size; i++)
{
    if (model_type == 123)    // number of algorithm
    {
        algorithm_list[i] = new abessLm<{T4}>(...);
    }
}
```

Finally, call `abessWorkflow()`, which would compute the result: [code link]

```
// "List" is a preset structure to store results
List out_result = abessWorkflow<{T1}, {T2}, {T3}, {T4}>(..., algorithm_list);
```

Implement your Algorithm

The implemented algorithms are stored in `src/AlgorithmXXX.h`. We have implemented some algorithms (e.g. `AlgorithmGLM.h`), so you can either create a new file containing new algorithm or simply add into existing one.

The new algorithm should inherit a base class, called *Algorithm*, which defined in `Algorithm.h`. And then rewrite some virtual function interfaces to fit specify problem. The implementation is modularized such that you can easily extend the package.

A concrete algorithm is like: [code link]

```
#include "Algorithm.h"

template <class T4>
class abess_new_algorithm : public Algorithm<{T1}, {T2}, {T3}, T4>  // T1, T2, T3 are␣
↪the same as above, which are fixed.
{
public:
    // constructor and destructor
    abess_new_algorithm(...) : Algorithm<...>::Algorithm(...){};
    ~abess_new_algorithm(){};

    void primary_model_fit(...){
        // solve the subproblem under given active set
        // record the sparse answer in variable "beta"
    };

    double loss_function(...){
        // define and compute loss under given active set
        // return the current loss
    };

    void sacrifice(...){
        // define and compute sacrifice for all variables (both forward and backward)
        // record sacrifice in variable "bd"
    };

    double effective_number_of_parameter(...){
        // return effective number of parameter
    };
}
```

Note that `sacrifice` function here would compute "forward/backward sacrifices" and record them in `bd`.

- For active variable, the lower (backward) sacrifice is, the more likely it will be dropped;
- For inactive variable, the higher (forward) sacrifice is, the more likely it will come into use.

If you create a new file to store the algorithm, remember to include it inside `src/api.cpp`. [code temp]

Now your new method has been connected to the whole frame. In the next section, we focus on how to build R or Python package based on the core code.

R & Python Package

R Package

To make sure your code available for R, run

```
R CMD INSTALL R-package
```

Then, this package would be installed into R session if the R package dependence (`Rcpp` and `Matrix`) have been installed.

After that, the object in R can be passed to Cpp via the unified API `abessCpp`. We strongly suggest the R function is named as `abessXXX` and use `roxygen2` to write R documentation and `devtools` to configure your package.

Python Package

First of all, you should ensure the C++ code available for Python, `cd` into directory `abess/python` and run `$ python setup.py install`. (Same steps in Installation)

It may take a few minutes to install:

- if the installation throw some errors, it means that the C++ code may be wrong;
- if the installation runs without errors, it will finish with message like "*Finished processing dependencies for abess*".

Then create a new python file in `python/abess` or open an existed file, such as `python/abess/linear.py`, to add a python API for your new method.

A simple new method can be added like: [code temp].

```
# all algorithms should inherit the base class `bess_base`
from .bess_base import bess_base

class new_algorithm(bess_base):
    """
    Here is some introduction.
    """
    def __init__(self, ...):
        super(abess_new_algorithm, self).__init__(
            algorithm_type="abess",
            model_type="new_algorithm",
            # other init
        )
    def fit(self, ...):
        # override `bess_base.fit()`, if necessary

    def custom_function(self, ...):
        # some custom functions, e.g. predict
```

The base class implements a `fit` function, which plays a role on checking input and calling C++ API to compute results. You may want to override it for custom features. [code temp].

Then, the final step is to link this Python class with the model type number (it has been defined in Section **Core C++**). In the `fit` function, you would find somewhere like:

```
if self.model_type == "new_algorithm":
    model_type_int = 123    # same number in C++
```

Finally, don't forget to import the new algorithm in `python/abess/__init__.py`.

Now run `$ python setup.py install` again and this time the installation would be finished quickly. Congratulation! Your work can now be used by:

```
from abess import new_algorithm
```

bess_base

As we show above, any new methods are based on `bess_base`, which can be found in `bess_base.py`: [code link]

```
from sklearn.base import BaseEstimator
class bess_base(BaseEstimator):
     def __init__(...):
        # some init
     def fit(...):
        # check input, warp with cpp
```

Actually, it is based on `sklearn.base.BaseEstimator` [code link]. Two methods, `get_params` and `set_params` are offered in this base class.

In our package, we write an method called `fit` to realize the abess process. Of cause, you can also override it like `SparsePCA`.

Verify you result

After programming the code, it is necessary to verify the contributed function can return a reasonable result. Here, we share our experience for it. Notice that the core our algorithm are forward and backward sacrifices, as long as they are properly programming, the contributed function would work well.

- Check `primary_model_fit` and `loss_function`

Secondly, we recommend you consider `primary_model_fit` for the computation of backward sacrifices. To check whether it works well, you can leverage the parameter `always.include` in R. Actually, when the number of elements pass to `always.include` is equal to `support.size` (`always_include` and `support_size` in Python), our algorithm is no need to do variable selection since all element must be selected, and thus, our implementation framework would just simply solving a convex problem by conducting `primary_model_fit` and the solution should match to (or close to) the function implemented in R/Python. Take the PCA task as an example, we should expect that, the results returned by `abess`:

```
data(USArrests)
abess_fit <- abesspca(USArrests, always.include = c(1:3), support.size = 3)
as.vector(spca_fit[["coef"]])[1:3]
```

should match with that returned by the `princomp` function:

```
princomp_fit <- loadings(princomp(USArrests[, 1:3]))[, 1]
princomp_fit
```

Actually, in our implementation, the results returned in two code blocks is match in magnitude. If the results are match, you can congratulate for your correct coding. We also recommend you write a automatic test case for this following the content below.

At the same time, you can see whether the `loss_function` is right by comparing `spca_fit[["loss"]]` and the variance of the first principal component.

- Check `sacrifice`

Thirdly, we recommend you consider `sacrifice`. Checking the function `sacrifice` needs more efforts. Monte Carlo studies should be conduct to check whether `sacrifice` is properly programmed such that the effective/relevant variables can be detected when sample size is large. We strongly recommend to check the result by setting: - sample size at least 1000 - dimension is less than 50 - the true support size is less than 5 - variables are independence - the support size from 0 to the ground true - the l_2 regularization is zero.

In most of the cases, this setting is very helpful for checking code. Generally, the output of `abess` would match to the correct under this setting. Take linear regression in R as our example, the code for checking is demonstrated below:

```
n <- 1000
p <- 50
support_size <- 3
dataset <- generate.data(n, p, support_size, seed = 1)
abess_fit <- abess(dataset[["x"]], dataset[["y"]], support.size = 0:support_size)
## estimated support:
extract(abess_fit, support.size = support_size)[["support.vars"]]
## true support:
which(dataset[["beta"]] != 0)
```

In this example, the estimated support set is the same as the true.

Write test cases

It is always a good habit to do some test for the changed package. Contributions with test cases included are easier to accept.

We use testthat for unit tests in R and pytest in Python. You may need to install first.

You can find some examples here and please feel free to add your test code into it (or create a new test file) under the test folder:

- R test folder: `abess/R-package/tests/testthat`.
- Python test folder: `python/pytest`.

A good test code should contain:

- possible input modes as well as some wrong input;
- check whether the output is expected;
- possible extreme cases;

All test under pytest folder should be checked after coding.

7.4.5 Develop New Features: GLM

Specifically, it could be pretty easy that you don't even need to understand the whole ABESS algorithm before developing a best-subset problem on Generalized Linear Models (GLM). In this tutorial, we will show you how to implement it.

For more general models, please check Develop New Features instead.

Preliminaries

We have endeavor to make developing new features easily. Before developing the code, please make sure you have:

- installed `abess` via the code in github by following Installation instruction;
- some experience on writing R or Python code.

Generalized linear model

In mathematics, we often denote the variables as $\mathbf{X}$ and the outcome as $\mathbf{y}$, which is assumed to be generated from a distribution in an exponential family, e.g.

- Normal Distribution;
- Binomial Distribution;
- Possion Distribution;
- ...

And the generalized linear model would be like:

$$\mathbb{E}(\mathbf{y}|\mathbf{X}) = \mu = g^{-1}(\mathbf{X}\beta),$$

where $\mathbb{E}(\mathbf{y}|\mathbf{X})$ is the expected value of $\mathbf{y}$ conditional on $\mathbf{X}$; g is the link function; β is the model parameters. Let's take the logistic regression as an example, where:

$$\mathbb{E}(y) = \mathbb{P}(y = 1) = p, \quad g^{-1}(z) = \frac{1}{1 + \exp(-z)}.$$

Core C++ for GLM

The first step is to write an API, which is the same as Develop New Features: Write an API.

Then, to implemented algorithms on GLM, you need to focus on `src/AlgorithmGLM.h`, where we have implemented a base model called `_abessGLM` and the new algorithm should inherit it.

```
template <class T4>
class abess_new_GLM_algorithm : public _abessGLM<{T1}, {T2}, {T3}, T4>  // T1, T2, T3␣
↪are the same as above, which are fixed.
{
public:
    // constructor and destructor
    abess_new_GLM_algorithm(...) : _abessGLM<...>::_abessGLM(...){};
    ~abess_new_GLM_algorithm(){};

    Eigen::MatrixXd gradian_core(...) {
```

(continues on next page)

(continued from previous page)

```
        // the gradian matrix can be expressed as G = X^T * A,
        // returns the gradian core A
    };
    Eigen::VectorXd hessian_core(...) {
        // the hessian matrix can be expressed as H = X^T * D * X,
        // returns the (diagnal values of) diagnal matrix D.
    };
    {T1} inv_link_function(...) {
        // returns inverse link function g^{-1}(X, beta),
        // i.e. the predicted y
    };
    {T1} log_probability(...) {
        // returns log P(y | X, beta)
    };
    bool null_model(...) {
        // returns a null model,
        // i.e. given only y, fit an intercept
    };
}
```

Compared with the general implement, it do not require touching the core of abess, but only use the knowledge of model itself.

Let's still discuss the logistic model and consider we hope to maximize log-likelihood: [code link]

$$
\begin{aligned}
l &= \frac{1}{2}\log\prod_i \mathbb{P}(y_i|X_i,\beta) \\
&= \frac{1}{2}\sum_i \left[y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \\
&= -\frac{1}{2}\sum_i \left[y_i\log(1-e^{-X_i^T\beta}) + (1-y_i)\log(1-e^{X_i^T\beta})\right],
\end{aligned}
$$

From this formula, we can get the `inv_link_function` and `log_probability`. And we continue on its derivatives on β:

$$
\begin{aligned}
\frac{\partial l}{\partial \beta} &= \sum_i X_i y_i (1-y_i), \\
\frac{\partial^2 l}{\partial \beta^2} &= \sum_i X_i X_i^T y_i (1-y_i)
\end{aligned}
$$

From this formula, we can get the `gradian_core` and `hessian_core`. Finally, the `null_model` should be:

$$
\mathbb{E}(y) = g^{-1}(C), \quad i.e. \quad C = g(\overline{y}),
$$

where $\overline{y}$ is the mean of y.

Now your new method has been connected to the whole frame. You can continue on the following steps like Develop New Features: R & Python Package.

7.4.6 After Code Developing

CodeFactor

We check the C++, R and Python format by CodeFactor. More specifically, the formatters and rules are:

- C++: CppLint with CPPLINT.cfg
- Python: Pylint with .pylintrc
- R: Lintr with .lintr

Each pull request will be checked, and some recommendations will be given if not passed. But don't be worry about those complex rules, most of them can be formatted automatically by some tools.

> Note that there may be few problems that the auto-fix tools can NOT deal with. In that situation, please update your pull request following the suggestions by CodeFactor.

Auto-format

C++

Clang-Format is a powerful tool to format C/C++ code. You can install it quickly:

- Linux: `$ sudo apt install clang-format`;
- MacOS: `$ brew install clang-format`;
- Windows: download it from LLVM;

with VS Code

If you use Visual Studio Code for coding, an extension called C/C++ supports auto-fix by Clang-Format.

However, in order to adapt to our rules, you need to add some statements in `setting.json` (the configuration file for Visual Studio Code):

```
"C_Cpp.clang_format_fallbackStyle": "{BasedOnStyle: Google, UseTab: Never, IndentWidth:␣
↪4, TabWidth: 4, ColumnLimit: 120, Standard: Cpp11}",
"files.insertFinalNewline": true,
"files.trimFinalNewlines": true,
// "editor.formatOnSave": true  // enable auto-fix after saving a file
```

After that, you can right-click on an C++ file and then click "Format Document". That will be done.

with command line

Besides, Clang-Format supports using directly in command line or based on a configuration file. You can check them here. The configuration is similar as above:

```
# `.clang-format` in the same directory of your C++ files
BasedOnStyle: Google
UseTab: Never
IndentWidth: 4
TabWidth: 4
```

(continues on next page)

(continued from previous page)

```
ColumnLimit: 120
Standard: Cpp11
```

And then run `$ clang-format -style=file some_code.cpp > some_code_formatted.cpp` in command line. The formatted code is stored in `some_code_formatted.cpp` now.

Python

Autopep8 can be used in formatting Python code. You can easily install it by `$ pip install autopep8`.

with VS Code

Visual Studio Code can deal with Python auto-fix too, with Python extension.

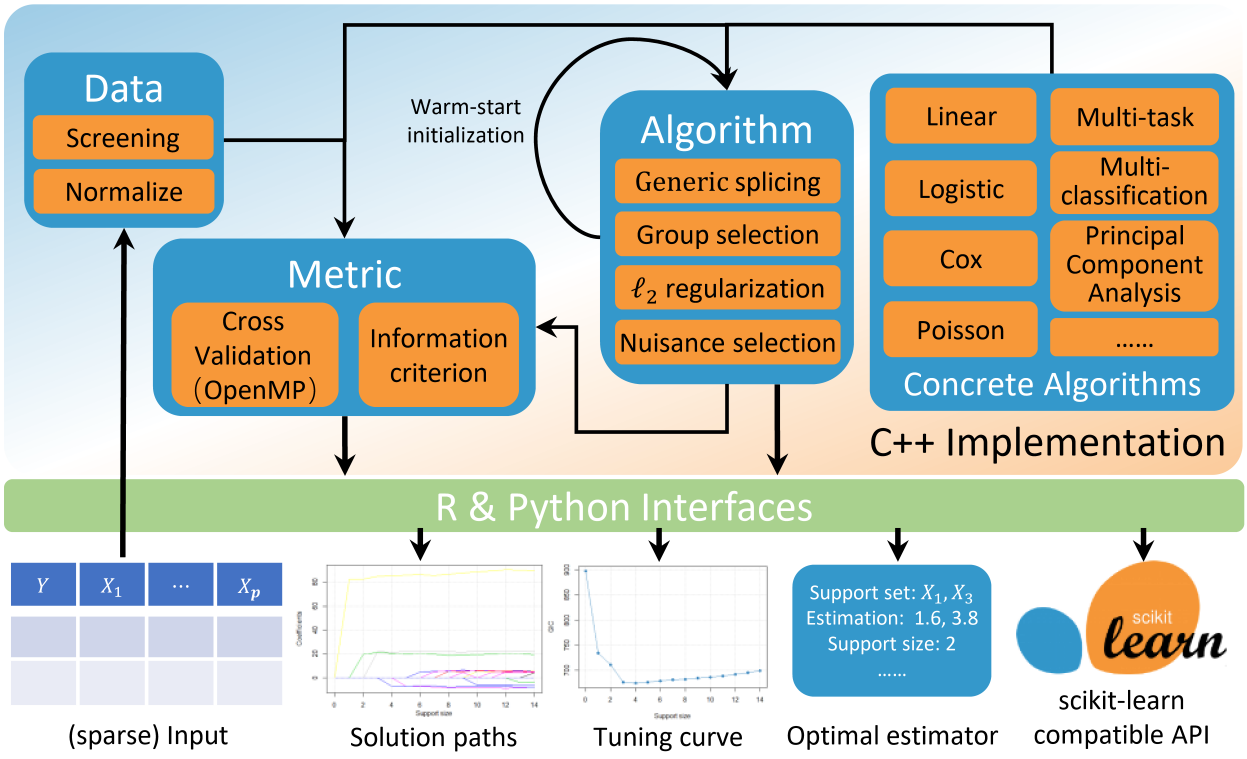
There is no more steps to do. Right-click on an Python file and then click "Format Document". That will be done.

with command line

As we memtioned above, the default setting of Autopep8 is enough for us. Hence run `$ autopep8 some_code.py > some_code_formatted.py` and the formatted code is stored in `some_code_formatted.py` now.

7.4.7 Appendix: Architecture of abess

In this page, we briefly introduce our core code of `abess`, which is summarized in the Figure below.



The core code of `abess` is built with C++ and the figure above shows the software architecture of `abess` and each building block will be described as follows.

- The **Data** class accept the (sparse) tabular data from R and Python interfaces, and returns a object containing the predictors are (optionally) screened or normalized.
- The **Algorithm** class, as the core class in `abess`, implements the generic splicing procedure for best subset selection with the support for L_2-regularization for parameters, group-structure predictors, and nuisance selection. The concrete algorithms are programmed in the subclass of **Algorithm** by rewriting the virtual function interfaces of class **Algorithm**. Seven implemented best subset selection tasks for supervised learning and unsupervised learning are presented in the above Figure. Beyond that, the modularized design facilitates users extend the library to various machine learning tasks by writing subclass of **Algorithm** class.
- The **Metric** The serves as a evaluator. It evaluates the estimation returned by **Algorithm** by cross validation or information criterion like Akaike information criterion and high dimensional Bayesian information criterion.
- Finally, **R or Python interfaces** collects the results from **Metric** and **Algorithm**. In R package, S3 methods are programmed such that generic functions (like `print`, `coef` and `plot`) can be directly used to attain the best subset selection results, and visualize solution paths and tuning parameter curve. In Python package, each model in abess is a sub-class of `scikit-learn`'s BaseEstimator class such that users can not only use a familiar API to train a model but also seamlessly combine model in `abess` preprocessing, feature transformation, and model selection module within `scikit-learn`.

7.5 FAQ

Some frequently asked questions would be shown here. If the error you met is not contained here, please open an issue on out project https://github.com/abess-team/abess/issues.

7.5.1 Python package

Installation failed

Compliers

First of all, please check the version of Python and GCC. To make sure that `abess` package runs correctly,

- Python 3.5 or later is required
- GCC 4.7 or later is required (support c++11)

What's more, the newer version is recommended. So if you meet some errors, please try to update the complier first.

Besides, in Windows, you may receive an error said "*error: Microsoft Visual C++ *version* is required*". To fix it, you need to check if MSVC is installed and enabled correctly. That is, to download Microsoft C++ Build Tools and install (or re-install) the "Desktop development with C++" module inside.

Permission

If you receive an error said "*Can't create or remove files in install directory*" during the installation, this may be caused by permission denied. The step below would help with it.

- For Linux/MacOS: run `$ python setup.py install --user` or `$ sudo python setup.py install` instead.
- For Windows: run `$ python setup.py install --user` or run the command as an administrator.

Import failed

Folder name

Make sure your working folder path is not named "abess". If not, Python would not import the `abess` packages and give some errors.

Example is not reproducible

Datasets

Some examples are based on data generated by `abess.datasets`. Since it depends on `numpy.random` and it would causes differences on a different OS or CPU, you might get a different dataset and finally a different result.

7.5.2 R package

- Update Rcpp package if you encounter the following errors:

```
function 'Rcpp_precious_remove' not provided by package 'Rcpp'
```

7.6 Changelog

7.6.1 Version 0.4.6

- R package
- Python package
  - Support *score* function for all GLM estimators.
  - Rearrange some arguments to improve legibility. Please check here for the latest API.
  - Better docstring, e.g. move important arguments to the front.
  - Combine *metrics.py* and *functions.py*.
- C++
  - Support the base model for GLM. The Sparse GLM model can be implemented much easilier.
  - Re-write logistic, poisson and gamma regression on the basis of GLM base model.

7.6.2 Versions 0.4.2 -- 0.4.5

- R package
  - Change the structure of R package such that the parameter check can be reused by different methods. As a by-production, code coverage for R package is impressively improved.
  - Support ordinal regression
  - Update README.md to synchronize with the layout change of abess official website.
- Python package
  - Fix bugs in sparse principal component analysis
  - Support ordinal regression
  - Support predicting survival function in `CoxPHSurvivalAnalysis()`
  - Modify python package to adapt to the criteria of conda-forge and `abess` is going to appear on conda-forge.
  - Spectra library is no long appear in `python/include` directory
  - Improve pytest by suppress unnecessary come from `scikit-learn` and the warning about API-name change. Moreover, some test will be skipped if some dependencies are missing.
  - Add estimator check from `scikit-learn` into pytest
  - Refine the configuration in `setup.py` to facilitate the source code installation
  - Support `get_params` and `set_params` methods for each model
- C++
  - Support ordinal regression
  - Fix bugs for multiple-regressors' API
  - Add more comments to improve readability, mainly in `Algorithm.h`, `utilities.h`, and `workflow.h`
- Project development
  - Test the package automatic submission. (It explains why the version number is quickly shifted.)
  - Python maintainer changes from Kangkang Jiang to Junhao Huang!

7.6.3 Version 0.4.1

- R package
  - Support user-specified initial active set.
- Python package
  - The API name shifts from `abessXXX` to `xxxRegression` and from `abessXXX` to `SparsePCA`
  - Improve the PEP8 criteria and `scikit-learn` criterion
  - The interface between python and cpp changes from swig to pybind11.
  - On Windows, the recommended C++ compiler for `abess` package installation shifts from Mingw to Microsoft Visual Studio because it is suggested that MinGW works with all Python versions up to 3.4.
  - Using cibuildwheel and github action to build and test *wheel* files automatically
  - Fix bugs in sparse principal component analysis

- Project development
  - Documentation
    * Add instruction for Gamma regression.
    * Update the usage of `support_size` in PCA.
    * Use Sphinx-Gallery for website layout, and update the layout of the `Tutorial` section.

7.6.4 Version 0.4.0

It is the fourth stable release for `abess`. More features and concrete algorithms are supported now and the main Cpp code has been refactored to improve scalability.

- Cpp
  - New features:
    * Support user-specified cross validation division.
    * Support user-specified initial active set.
    * Support flexible support size for sequentially best subset selection for principal component analysis (PCA).
  - New best subset selection tasks:
    * Generalized linear model when the link function is gamma distribution.
    * Robust principal component analysis (RPCA).
  - Performance improvement:
    * Bug fixed
- Python
  - New best subset selection features and tasks implemented in Cpp are wrapped in Python functions.
  - More comprehensive test files.
  - A new release in Pypi.
- R package
  - New best subset selection features and tasks implemented in Cpp are wrapped in R functions.
  - A new release in CRAN.
- Project development
  - Source code
    * Refactoring the Cpp source code to improve its readability and scalability. Please check Code Developing section for more details.
    * Combine all parameters (e.g. support_size and lambda) in one list to improve expandability.
    * Move the core code `src` directory to the root of repository.
  - Documentation
    * Add instruction for robust principal component analysis in Tutorial.
    * Add instruction for user-specified cross validation division in Advanced Features.
    * Update development guideline according to cpp source code change in Code Developing.

        * Adding more details and giving more links related to core functions.
    - Code coverage
        * Add more test suites to improve coverage and stability
    - Code format
        * Code format is checked by CodeFactor. For more details, please check Code Format.

7.6.5 Version 0.3.0

It is the third stable release for `abess`. This version improve the runtime performance, the clarity of project's documentation, and add helpful continuous integration.

- Cpp
    - New features:
        * Support important searching to significantly improve computational efficiency when dimensionality is large.
    - Performance improvement:
        * Update the version of dependencies: from Spectra 0.9.0 to 1.0.0
        * Bug fixed
- R package
    - Support important searching for generalized linear model in `abess`
    - A new release in CRAN.
- Python package
    - Remove useless parameter to improve clarity.
    - Support important searching for generalized linear model `abessLm`, `abessLogistic`, `abessPoisson`, `abessCox`, `abessMlm`, `abessMultinomial`
    - A new release in Pypi.
- Project development
    - Code coverage
        * Check line covering rate for both Python and R. And the coverage rates are summarized and report.
        * Add more test suites to improve coverage and stability
    - Documentation
        * Add docs2search for the R package's website
        * Add a logo for the project
        * Improve documentation by adding two tutorial sections: `detail of algorithm` and `power of abess`.
    - Improve code coverage
    - Continuous integration
        * Check the installation in Windows, Mac, and Linux
        * Automatically generate the `.whl` files and publish the Python package into Pypi when tagging the project in github.

7.6.6 Version 0.2.0

It is the second stable release for `abess`. This version includes multiple several generic features, and optimize memory usage when input data is a sparse matrix. We also significantly enhancements to the project' documentation.

- Cpp
  - New generic best subset features:
    - The selection of group-structured best subset selection;
    - Ridge-regularized penalty for parameter as a generic component.
  - New best subset selection tasks:
    - principal component analysis
  - Performance improvement:
    - Support sparse matrix as input
    - Support golden section search for optimal support size. It is much faster than sequentially searching strategy.
    - The logic behind cross validation is optimized to gain speed improvement
    - Covariance update
    - Bug fixed
- R package
  - New best subset selection features and tasks implemented in Cpp are wrapped in R functions.
  - `abesspca` supports best subset selection for the first loading vector in principal component analysis. A iterative algorithm supports multiple loading vectors.
  - Generic S3 function for `abesspca`.
  - Both `abess` and `abesspca` supports sparse matrix input (inherit from class "sparseMatrix" as in package Matrix).
  - Upload to CRAN.
- Python package
  - New best subset selection features and tasks implemented in Cpp are wrapped in Python functions.
  - *abessPCA* supports best subset selection for the first loading vector in principal component analysis. A iterative algorithm supports multiple loading vectors.
  - Support integration with `scikit-learn`. It is compatible with model evaluation and selection module with `scikit-learn`.
  - Initial Upload to Pypi.
- Project development
  - Documentation
    - A more clear project website layout.
    - Add an instruction for
    - Add tutorials to show simple use-cases and non-trival examples of typical use-cases of the software.
    - Link to R-package website.
    - Add an instruction to help package development.

  - Code coverage for line covering rate for Python.
  - Continuous integration:
    * Change toolbox from Travis CI to Github-Action.
    * Auto deploy code coverage result to codecov.

7.6.7 Version 0.1.0

We're happy to announce the first major stable version of `abess`. This version includes multiple new algorithms and features. Here are some highlights of the big updates.

- Cpp
  - New generic best subset features:
    * generic splicing technique
    * nuisance selection
  - New best subset selection tasks:
    * linear regression
    * logistic regression
    * poisson regression
    * cox proportional hazard regression
    * multi-gaussian regression
    * multi-nominal regression.
  - Cross validation and information criterion to select the optimal support size
  - Performance improvement:
    * Support OPENMP for the parallelism when performing cross validation
    * Warm start initialization
  - Create a List object to: 1. facilitate transfer the data object from Cpp to Python; 2. use the maximum compatible code for python and R
- R package
  - All best subset selection features and tasks implemented in Cpp are wrapped in a R function `abess`.
  - Unified API for cross validation and information criterion to select the optimal support size.
  - Support generic S3 functions like `coef` and `plot` in R.
  - A short vignettes for demonstrating the usage of package.
  - Support formula interface.
  - Support convenient function for generating synthetic dataset.
  - Initial upload to CRAN.
- Python
  - All best subset selection features implemented in Cpp are wrapped in a Python according to tasks. For instance, *abessLm* supports best subset selection for the linear model.

  - Write the Python class on the basis of `scikit-learn` package. The usage of the python package is the same as the common module in `scikit-learn`.
  - Support convenient function for generating synthetic dataset in Python.

- Project developing
  - Build R package website via the `pkgdown` package.
  - Build a documentation website on based the Python package via the `sphnix` package.
  - The website is continuous integrated via Travis CI. The content will automatically change whether a Travis CI is triggered.
  - Complete testing for R functions in package.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX