Regressors Documentation

Release 0.0.1

Nikhil Haas

December 08, 2015

Contents

Contents:

CHAPTER 1

Regressors

Easy utilities for fitting various regressors, extracting stats, and making relevant plots

- Free software: ISC license
- Documentation: https://regressors.readthedocs.org.

1.1 Features

- TODO

1.2 Credits

Tools used in rendering this package:

- Cookiecutter
- cookiecutter-pypackage

CHAPTER 2

Installation

You should have *Numpy* and *SciPy* installed prior to installation.

If you have virtualenvwrapper installed:

```
$ mkvirtualenv regressors
$ pip install numpy scipy
$ pip install regressors
```

Or, at the command line:

```
$ easy_install numpy scipy
$ easy_install regressors
```

CHAPTER 3

Usage

Below are some demonstrations of using Regressors in a project. We'll import a the Boston data set first to demonstrate the functions' usage:

```
import numpy as np
from sklearn import datasets
boston = datasets.load_boston()
which_betas = np.ones(13, dtype=bool)
which_betas[3] = False  # Eliminate dummy variable
X = boston.data[:, which_betas]
y = boston.target
```

3.1 Obtaining Summary Statistics

There are several functions provided that compute various statistics about some of the regression models in scikit-learn. These functions are:

1. *regressors.stats.sse(clf, X, y)*
2. *regressors.stats.adj_r2_score(clf, X, y)*
3. *regressors.stats.coef_se(clf, X, y)*
4. *regressors.stats.coef_tval(clf, X, y)*
5. *regressors.stats.coef_pval(clf, X, y)*
6. *regressors.stats.f_stat(clf, X, y)*
7. *regressors.stats.residuals(clf, X, y)*
8. *regressors.stats.summary(clf, X, y, Xlabels)*

The last function, *summary()*, outputs the metrics seen above in a nice format.

An example with is developed below for a better understanding of these functions. Here, we use an ordinary least squares regression model, but another, such as Lasso, could be used.

3.1.1 SSE

To calculate the SSE:

```
from sklearn import linear_model
from regressors import stats
ols = linear_model.LinearRegression()
ols.fit(X, y)

stats.sse(ols, X, y)
```

Output: 11299.555410604258

3.1.2 Adjusted R-Squared

To calculate the adjusted R-squared:

```
from sklearn import linear_model
from regressors import stats
ols = linear_model.LinearRegression()
ols.fit(X, y)

stats.adj_r2_score(ols, X, y)
```

Output: 0.72903560136853518

3.1.3 Standard Error of Beta Coefficients

To calculate the standard error of beta coefficients:

```
from sklearn import linear_model
from regressors import stats
ols = linear_model.LinearRegression()
ols.fit(X, y)

stats.coef_se(ols, X, y)
```

Output:

```
array([  4.91564654e+00,   3.15831325e-02,   1.07052582e-02,
         5.58441441e-02,   3.59192651e+00,   2.72990186e-01,
         9.62874778e-03,   1.80529926e-01,   6.15688821e-02,
         1.05459120e-03,   8.89940838e-02,   1.12619897e-03,
         4.21280888e-02])
```

3.1.4 T-values of Beta Coefficients

To calculate the t-values beta coefficients:

```
from sklearn import linear_model
from regressors import stats
ols = linear_model.LinearRegression()
ols.fit(X, y)

stats.coef_tval(ols, X, y)
```

Output:

```
array([  7.51173468,  -3.55339694,   4.39272142,   0.72781367,
        -4.84335873,  14.08541122,   0.29566133,  -8.22887   ,
         5.32566707, -13.03948192, -11.14380943,   8.72558338,
        -12.69733326])
```

3.1.5 P-values of Beta Coefficients

To calculate the p-values of beta coefficients:

```
from sklearn import linear_model
from regressors import stats
ols = linear_model.LinearRegression()
ols.fit(X, y)

stats.coef_pval(ols, X, y)
```

Output:

```
array([  2.66897615e-13,   4.15972994e-04,   1.36473287e-05,
         4.67064962e-01,   1.70032518e-06,   0.00000000e+00,
         7.67610259e-01,   1.55431223e-15,   1.51691918e-07,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00])
```

3.1.6 F-statistic

To calculate the F-statistic of beta coefficients:

```
from sklearn import linear_model
from regressors import stats
ols = linear_model.LinearRegression()
ols.fit(X, y)

stats.f_stat(ols, X, y)
```

Output: 114.22612261689403

3.1.7 Summary

The summary statistic table calls many of the stats outputs the statistics in an pretty format, similar to that seen in R.

The coefficients can be labeled more descriptively by passing in a list of lables. If no labels are provided, they will be generated in the format x1, x2, x3, etc.

To obtain the summary table:

```
from sklearn import linear_model
from regressors import stats
ols = linear_model.LinearRegression()
ols.fit(X, y)

xlabels = boston.feature_names[which_betas]
stats.summary(ols, X, y, xlabels)
```

Output:

```
Residuals:
     Min      1Q  Median      3Q     Max
-26.3743 -1.9207  0.6648  2.8112  13.3794


Coefficients:
             Estimate  Std. Error  t value   p value
_intercept  36.925033    4.915647   7.5117  0.000000
CRIM        -0.112227    0.031583  -3.5534  0.000416
ZN           0.047025    0.010705   4.3927  0.000014
INDUS        0.040644    0.055844   0.7278  0.467065
NOX        -17.396989    3.591927  -4.8434  0.000002
RM           3.845179    0.272990  14.0854  0.000000
AGE          0.002847    0.009629   0.2957  0.767610
DIS         -1.485557    0.180530  -8.2289  0.000000
RAD          0.327895    0.061569   5.3257  0.000000
TAX         -0.013751    0.001055 -13.0395  0.000000
PTRATIO     -0.991733    0.088994 -11.1438  0.000000
B            0.009827    0.001126   8.7256  0.000000
LSTAT       -0.534914    0.042128 -12.6973  0.000000
---
R-squared:  0.73547,    Adjusted R-squared:  0.72904
F-statistic: 114.23 on 12 features
```

3.2 Plotting

Several functions are provided to quickly and easily make plots useful for judging a model.

1. *regressors.plots.plot_residuals(clf, X, y, r_type, figsize)*
2. *regressors.plots.plot_qq(clf, X, y, figsize)*
3. *regressors.plots.plot_pca_pairs(clf_pca, x_train, y, n_components, diag, cmap, figsize)*
4. *regressors.plots.plot_scree(clf_pca, xlim, ylim, required_var, figsize)*

We will continue using the Boston data set referenced above.

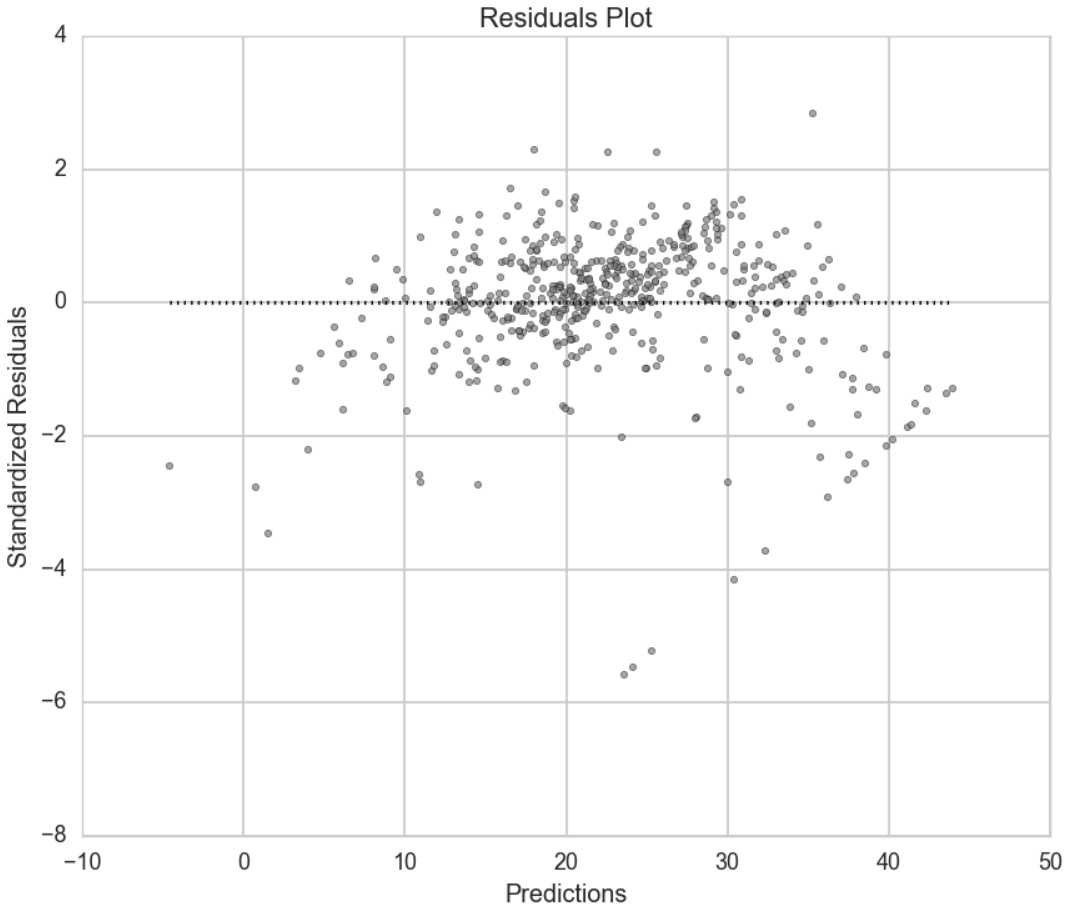
3.2.1 Residuals

Residuals can be plotted as actual residuals, standard residuals, or studentized residuals:

```
from sklearn import linear_model
from regressors import plots
ols = linear_model.LinearRegression()
ols.fit(X, y)

plots.plot_residuals(ols, X, y, r_type='standardized')
```

Plots:



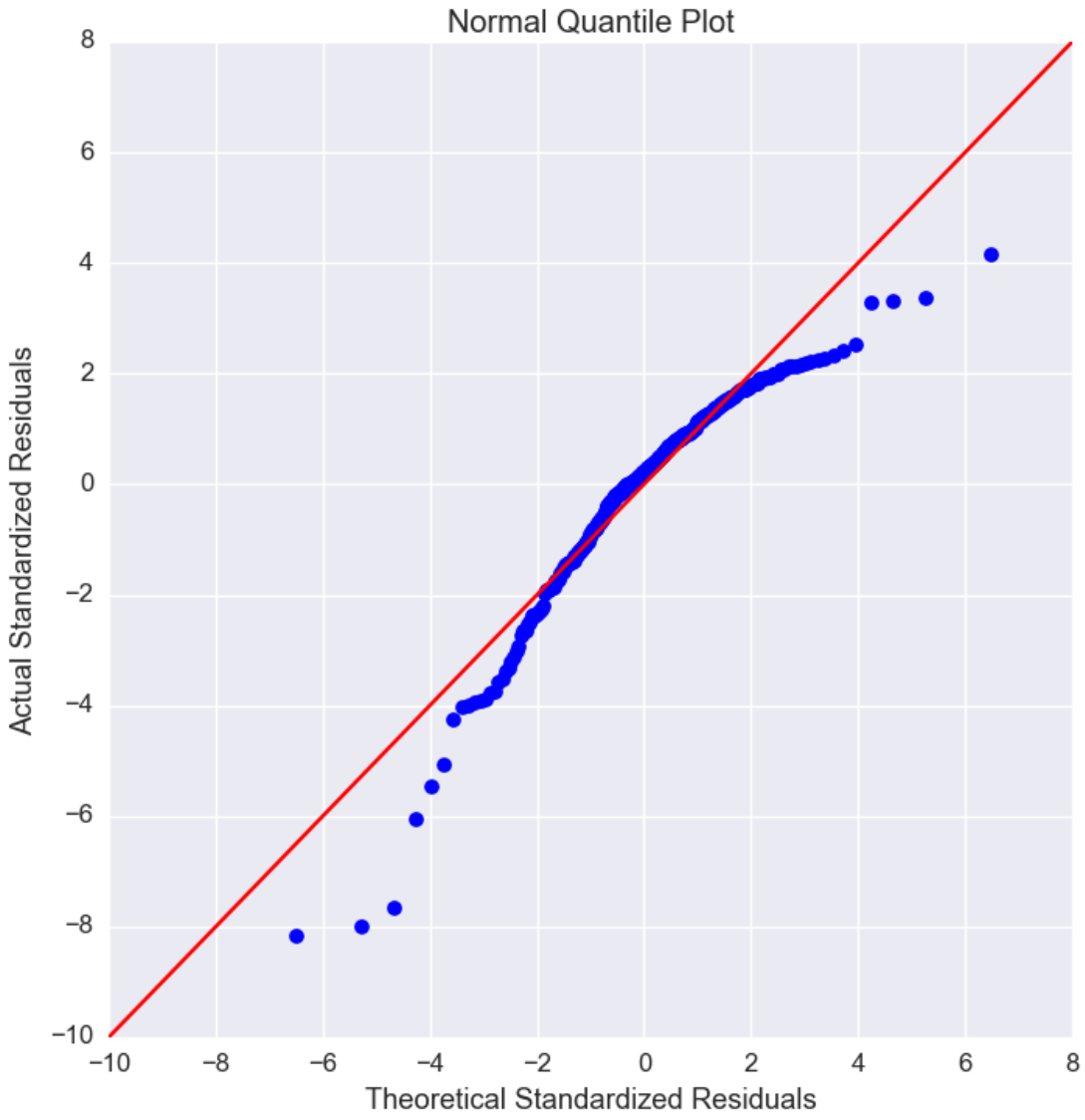
3.2.2 Q-Q Plot

Q-Q plots can quickly be obtained to aid in checking the normal assumption:

```
from sklearn import linear_model
from regressors import plots
ols = linear_model.LinearRegression()
ols.fit(X, y)

plots.plot_qq(ols, X, y, figsize=(8, 8))
```

Plots:



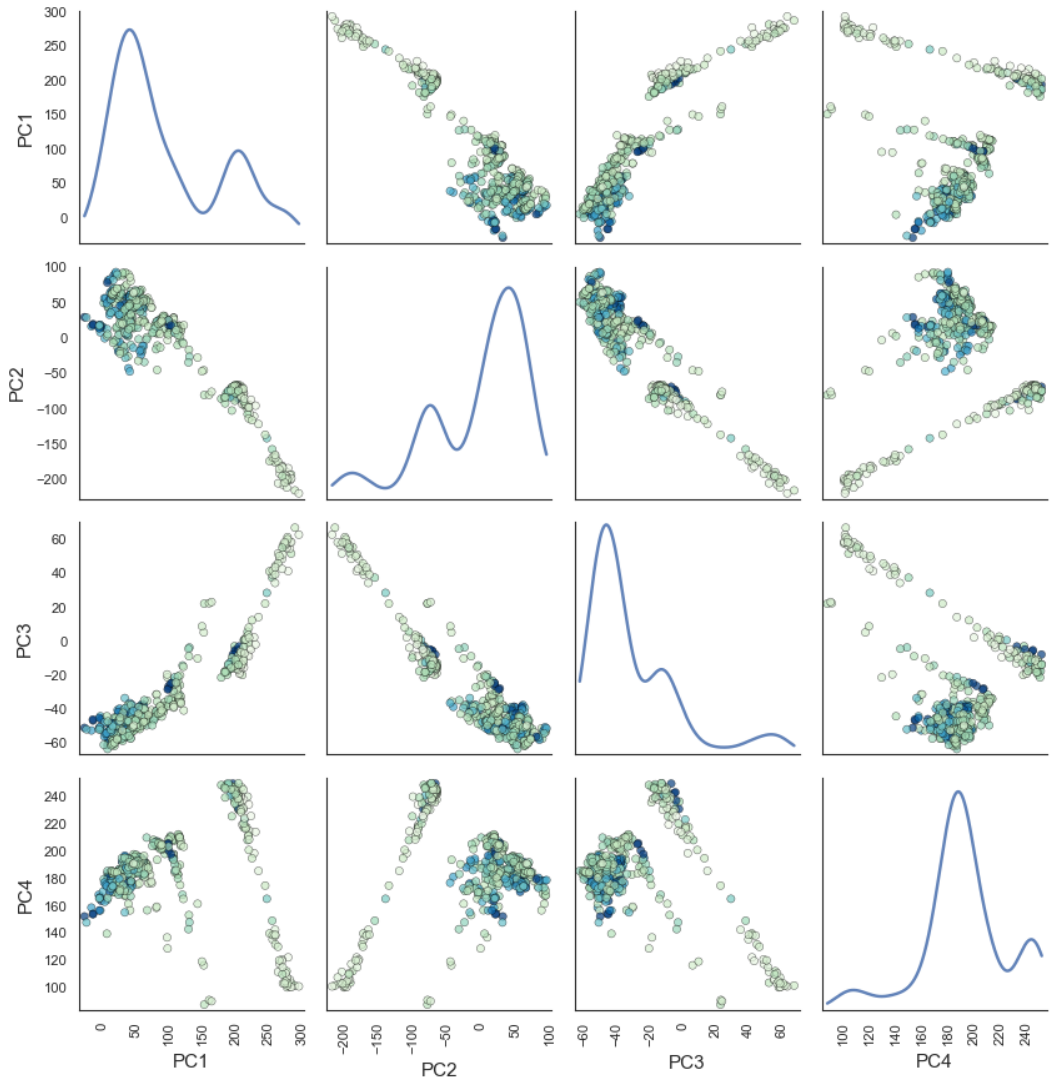
3.2.3 Principle Components Pairs

To generate a pairwise plot of principle components:

```
from sklearn import preprocessing
from sklearn import decomposition
from regressors import plots
scaler = preprocessing.StandardScaler()
x_scaled = scaler.fit_transform(X)
pcomp = decomposition.PCA()
pcomp.fit(x_scaled)

plots.plot_pca_pairs(pcomp, X, y, n_components=4, cmap="GnBu")
```

Plots:



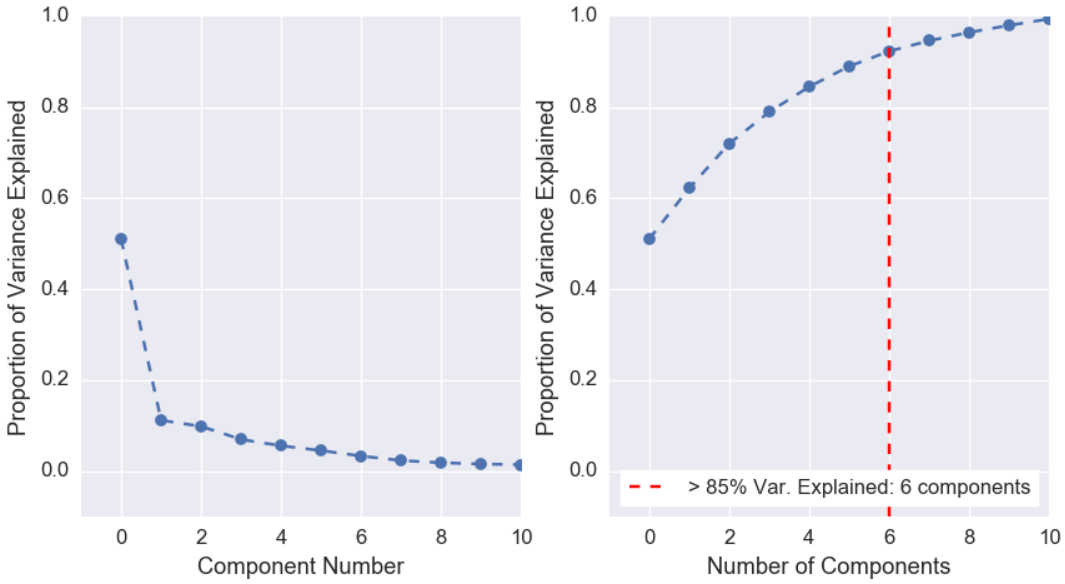
3.2.4 Scree Plot

Scree plots can be quickly generated to visualize the amount of variance represented by each principle component with a helpful marker to see where a threshold of variance is reached:

```
from sklearn import preprocessing
from sklearn import decomposition
from regressors import plots
scaler = preprocessing.StandardScaler()
x_scaled = scaler.fit_transform(X)
pcomp = decomposition.PCA()
pcomp.fit(x_scaled)
```

```
plots.plot_scree(pcomp, required_var=0.85)
```

Plots:



3.3 Principle Components Regression (PCR)

The PCR class can be used to quickly run PCR on a data set. This class provides the familiar `fit()`, `predict()`, and `score()` methods that are common to scikit-learn regression models. The type of scaler, the number of components for PCA, and the regression model are all tunable.

3.3.1 PCR Class

An example of using the PCR class:

```
from regressors import regressors
pcr = regressors.PCR(n_components=10, regression_type='ols')
pcr.fit(X, y)

# The fitted scaler, pca, and scaler models can be accessed:
scaler, pca, regression = (pcr.scaler, pcr.prcomp, pcr.regression)

# You could then make various plots, such as pca_pairs_plot(), and
# plot_residuals() with these fitted model from PCR.
```

3.3.2 Beta Coefficients

The coefficients in PCR's regression model are coefficients for the PCA space. To transform those components back to the space of the original X data:

```
from regressors import regressors
pcr = regressors.PCR(n_components=10, regression_type='ols')
```

```
pcr.fit(X, y)
pcr.beta_coef_
```

Output::

array([-0.96384079, 1.09565914, 0.27855742, -2.0139296 , 2.69901773, 0.08005632, -3.12506044, 2.85224741, -2.31531704, -2.14492552, 0.89624424, -3.81608008])

Note that the intercept is the same for the X space and the PCA space, so simply access that directly with `pcr.self.regression.intercept_`.

CHAPTER 4

Modules

4.1 regressors.plots

regressors.plots.**plot_residuals**(*clf*, *X*, *y*, *r_type=u'standardized'*, *figsize=(10, 8)*)

Plot residuals of a linear model.

Parameters

- **clf** (*sklearn.linear_model*) – A scikit-learn linear model classifier with a *predict()* method.
- **X** (*numpy.ndarray*) – Training data used to fit the classifier.
- **y** (*numpy.ndarray*) – Target training values, of shape = [n_samples].
- **r_type** (*str*) – Type of residuals to return: 'raw', 'standardized', 'studentized'. Defaults to 'standardized'.
  - 'raw' will return the raw residuals.
  - 'standardized' will return the standardized residuals, also known as internally studentized residuals, which is calculated as the residuals divided by the square root of MSE (or the STD of the residuals).
  - 'studentized' will return the externally studentized residuals, which is calculated as the raw residuals divided by sqrt(LOO-MSE * (1 - leverage_score)).
- **figsize** (*tuple*) – A tuple indicating the size of the plot to be created, with format (x-axis, y-axis). Defaults to (10, 8).

Returns The Figure instance.

Return type matplotlib.figure.Figure

4.2 regressors.stats

regressors.stats.**residuals**(*clf*, *X*, *y*, *r_type=u'standardized'*)

Calculate residuals or standardized residuals.

Parameters

- **clf** (*sklearn.linear_model*) – A scikit-learn linear model classifier with a *predict()* method.
- **X** (*numpy.ndarray*) – Training data used to fit the classifier.
- **y** (*numpy.ndarray*) – Target training values, of shape = [n_samples].

- **r_type** (*str*) – Type of residuals to return: 'raw', 'standardized', 'studentized'. Defaults to 'standardized'.
  - 'raw' will return the raw residuals.
  - 'standardized' will return the standardized residuals, also known as internally studentized residuals, which is calculated as the residuals divided by the square root of MSE (or the STD of the residuals).
  - 'studentized' will return the externally studentized residuals, which is calculated as the raw residuals divided by sqrt(LOO-MSE * (1 - leverage_score)).

Returns An array of residuals.

Return type numpy.ndarray

4.3 regressors.regressors

class regressors.regressors.**PCR**(*n_components=None*, *regression_type=u'ols'*, *alpha=1.0*, *l1_ratio=0.5*, *n_jobs=1*)

Principle components regression model.

This model solves a regression model after standard scaling the X data and performing PCA to reduce the dimensionality of X. This class simply creates a pipeline that utilizes:

1.sklearn.preprocessing.StandardScaler

2.sklearn.decomposition.PCA

3.a supported sklearn.linear_model

Attributes of the class mimic what is provided by scikit-learn's PCA and linear model classes. Additional attributes specifically relevant to PCR are also provided, such as *PCR.beta_coef_*.

Parameters

- **n_components** (*int, float, None, str*) – Number of components to keep when performing PCA. If n_components is not set all components are kept:

```
n_components == min(n_samples, n_features)
```

  If n_components == 'mle', Minka's MLE is used to guess the dimension. If `0 < n_components < 1`, selects the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components.
- **regression_type** (*str*) – The type of regression classifier to use. Must be one of 'ols', 'lasso', 'ridge', or 'elasticnet'.
- **n_jobs** (*int (optional)*) – The number of jobs to use for the computation. If `n_jobs=-1`, all CPUs are used. This will only increase speed of computation for n_targets > 1 and sufficiently large problems.
- **alpha** (*float (optional)*) – Used when regression_type is 'lasso', 'ridge', or 'elasticnet'. Represents the constant that multiplies the penalty terms. Setting `alpha=0` is equivalent to ordinary least square and it is advised in that case to instead use `regression_type='ols'`. See the scikit-learn documentation for the chosen regression model for more information in this parameter.
- **l1_ratio** (*float (optional)*) – Used when regression_type is 'elasticnet'. The ElasticNet mixing parameter, with `0 <= l1_ratio <= 1`. For `l1_ratio = 0` the penalty is an

L2 penalty. `For l1_ratio = 1` it is an L1 penalty. For `0 < l1_ratio < 1`, the penalty is a combination of L1 and L2.

scaler
sklearn.preprocessing.StandardScaler, None

The StandardScaler object used to center the X data and scale to unit variance. Must have `fit()` and `transform()` methods. Can be overridden prior to fitting to use a different scaler:

```
pcr = PCR()
# Change StandardScaler options
pcr.scaler = StandardScaler(with_mean=False, with_std=True)
pcr.fit(X, y)
```

The scaler can also be removed prior to fitting (to not scale X during fitting or predictions) with *pcr.scaler = None*.

prcomp
sklearn.decomposition.PCA

The PCA object use to perform PCA. This can also be accessed in the same way as the scaler.

regression
sklearn.linear_model

The linear model object used to perform regression. Must have `fit()` and `predict()` methods. This defaults to OLS using scikit-learn's LinearRegression classifier, but can be overridden either using the *regression_type* parameter when instantiating the class, or by replacing the regression model with a different on prior to fitting:

```
pcr = PCR(regression_type='ols')
# Examine the current regression model
print(pcr.regression)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
    normalize=False)
# Use Lasso regression with cross-validation instead of OLS
pcr.regression = linear_model.LassoCV(n_alphas=200)
print(pcr.regression)
LassoCV(alphas=None, copy_X=True, cv=None, eps=0.001,
    fit_intercept=True, max_iter=1000, n_alphas=200, n_jobs=1,
    normalize=False, positive=False, precompute='auto',
    random_state=None, selection='cyclic', tol=0.0001,
    verbose=False)
pcr.fit(X, y)
```

beta_coef_

Returns Beta coefficients, corresponding to coefficients in the original space and dimension of X. These are calculated as $B = A \cdot P$, where A is a vector of the coefficients obtained from regression on the principle components and P is the matrix of loadings from PCA.

Return type numpy.ndarray

fit (*X*, *y*)
Fit the PCR model.

Parameters

- **X** (*numpy.ndarray*) – Training data.
- **y** (*numpy.ndarray*) – Target values.

Returns An instance of self.

Return type regression.PCR

intercept_

Returns The intercept for the regression model, both in PCA-space and in the original X-space.

Return type float

predict(*X*)

Predict using the PCR model.

Parameters **X** (*numpy.ndarray*) – Samples to predict values from.

Returns Predicted values.

Return type numpy.ndarray

score(*X*, *y*)

Returns the coefficient of determination of R^2 of the predictions.

Parameters

- **X** (*numpy.ndarray*) – Training or tests samples.
- **y** (*numpy.ndarray*) – Target values.

Returns The R^2 value of the predictions.

Return type float

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/nsh87/regressors/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

5.1.4 Write Documentation

Regressors could always use more documentation, whether as part of the official Regressors docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/nsh87/regressors/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *regressors* for local development.

1. Fork the *regressors* repo on GitHub.
2. Clone your fork locally, then add the original repository as an upstream:

```
$ git clone git@github.com:your_name_here/regressors.git
$ cd regressors
$ git remote add upstream https://github.com/nsh87/regressors
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ pip install virtualenv virtualenvwrapper
$ mkvirtualenv -r requirements_dev.txt regressors
$ pip install numpy scipy
$ python setup.py develop
```

4. Create a branch for local development, branching off of *dev*:

```
$ git checkout -b name-of-your-bugfix-or-feature dev
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 regressors tests  # Check Python syntax
$ python setup.py test  # Run unittest tests
$ tox  # Run unittests and check compatibility on Python 2.6, 2.7, 3.3-5
```

   flake8 and tox will have been installed when you created the virtualenv above.

   In order to fully support tox, you will need to have Python 2.6, 2.7, 3.3, 3.4, and 3.5 available on your system. If you're using Mac OS X you can follow this guide to cleanly install multiple Python versions.

   If you are not able to get all tox environments working, that's fine, but take heed that a pull request that has not been tested against all Python versions might be rejected if it is not compatible with a specific version. You should try your best to get the `tox` command working so you can verify your code and tests against multiple Python versions. You should check Travis CI in lieu once your pull request has been submitted.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

   Write sensible commit message: read this post and this one before writing a single commit.

7. Submit a pull request through the GitHub website to merge your feature to branch *dev*. To ensure your pull request can be automatically merged, play your commits on top of the most recent *dev* branch:

```
$ git fetch upstream
$ git checkout dev
$ git merge upstream/dev
$ git checkout name-of-your-bugfix-or-feature
$ git rebase dev
```

This will pull the latest changes from the main repository and let you take care of resolving any merge conflicts that might arise in order for your pull request to be merged.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/nsh87/regressors/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_regressors
```

CHAPTER 6

Credits

6.1 Development Lead

- Nikhil Haas <nikhil@nikhilhaas.com>
- Ghizlaine Bennani
- Alex Romriell

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

CHAPTER 8

0.0.1 (2015-11-24)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

B

F

I

P

R

S