

Jupyter

pip install jupyter	installs jupyter
jupyter notebook	starts jupyter notebook
Creating a notebook	go to new on the upper right and click on python
Run	shift + enter
File menu	can create a new Notebook or open a preexisting one. This is also where you would go to rename a Notebook. I think the most interesting menu item is the Save and Checkpoint option. This allows you to create checkpoints that you can roll back to if you need to.
Edit menu	Here you can cut, copy, and paste cells. This is also where you would go if you wanted to delete, split, or merge a cell. You can reorder cells here too.
View menu	useful for toggling the visibility of the header and toolbar. You can also toggle Line Numbers within cells on or off. This is also where you would go if you want to mess about with the cell's toolbar.

Jupyter (cont)

Insert menu	just for inserting cells above or below the currently selected cell.
Cell menu	allows you to run one cell, a group of cells, or all the cells. You can also go here to change a cell's type, although the toolbar is more intuitive for that. The other handy feature in this menu is the ability to clear a cell's output.
Kernel cell	is for working with the kernel that is running in the background. Here you can restart the kernel, reconnect to it, shut it down, or even change which kernel your Notebook is using.
Widgets menu	is for saving and clearing widget state. Widgets are basically JavaScript widgets that you can add to your cells to make dynamic content using Python (or another Kernel).

Jupyter (cont)

Help menu	which is where you go to learn about the Notebook's keyboard shortcuts, a user interface tour, and lots of reference material.
Running tab	will tell you which Notebooks and Terminals you are currently running.
cell types: Code	cell where you write code
cell types: Raw	is only intended for special use cases when using the nbconvert command line tool. Basically it allows you to control the formatting in a very specific way when converting from a Notebook to another format.
cell types: Heading	The Heading cell type is no longer supported and will display a dialog that says as much. Instead, you are supposed to use Markdown for your Headings.



By Remidy08

cheatography.com/remidy08/

Not published yet.

Last updated 6th September, 2022.

Page 1 of 9.

Sponsored by [CrosswordCheats.com](https://crosswordcheats.com)

Learn to solve cryptic crosswords!

<http://crosswordcheats.com>

Jupyter (cont)

cell types: Jupyter Notebook supports
 Markdown Markdown, which is a markup language that is a superset of HTML. Next up some of the possible utilities of this type of cell will be shown. Once a markdown cell is written, its text cannot be changed.

`_italic_ or italic`

`# Header 1`

`## Header 2`

`### Header 3`

You can create a list (bullet points) by using dashes, plus signs, or asterisks. There needs to be a space between the marker and the letters. To make sub lists, press tab first

For inline code highlighting, just surround the code with backticks. If you want to insert a block of code, you can use triple backticks and also specify the programming language:

``python ...` in multiple lines`

Jupyter (cont)

Exporting notebooks When you are working with Jupyter Notebooks, you will find that you need to share your results with non-technical people. When that happens, you can use the nbconvert tool which comes with Jupyter Notebook to convert or export your Notebook into one of the following formats: HTML, LaTeX, PDF, RevealJS, Markdown, ReStructured Text, Executable script

How to Use nbconvert Open up a terminal and navigate to the folder that contains the Notebook you wish to convert. The basic conversion command looks like this: `jupyter nbconvert <input notebook> --to <output format>`. Example: `upyter nbconvert py_examples.ipynb --to pdf`

Jupyter (cont)

You can also export your currently running Notebook by going to the File menu and choosing the Download as option. This option allows you to download in all the formats that nbconvert supports. However I recommend doing so as you can use nbconvert to export multiple Notebooks at once, which is something that the menu does not support.

Extensions A Notebook extension (nbextension) is a JavaScript module that you load in most of the views in the Notebook's frontend.

Where Do I Get Extensions? You can use Google or search for Jupyter Notebook extensions.

How Do I Install Them? `jupyter nbextension install EXTENSION_NAME`

enable an extension after installing it `jupyter nbextension enable EXTENSION_NAME`

installing python packages `! pip install package_name --user`

If you see a greyed out menu item, try changing the cell's type and see if the item becomes available to use.

Evaluation Metrics and Scoring

Importing	<pre>from sklearn.metrics import confusion_matrix confusion = confusion_matri- x(y_test, LogisticRegression- (C=0.1).fit(X_train, y_train).pre- dict(X_test))</pre>
Accuracy	$(TP+TN)/(TP+TN+FP+FN)$
Precision (positive predictive value)	$TP/(TP+FP)$
Recall	$TP/(TP+FN)$
f-score	$2 * (precision-recall) / (precision+recall)$
Importing f-score	<pre>from sklearn.metrics import f1_score</pre>
f1_score	<pre>f1_score(y_test, pred_most_fr- equent)))</pre>
Importing classific- ation report	<pre>from sklearn.metrics import classification_report classification_report(y_test, model, target_names=["not nine", "nine"])</pre>
Prediction threshold	<pre>y_pred_lower_threshold = svc.decision_function(X_test) > -.8</pre>
Classific- ation report	<pre>classification_report(y_test, y_pred_lower_threshold)</pre>
Importing precis- on_recall- _curve	<pre>from sklearn.metrics import precision_recall_curve</pre>

Evaluation Metrics and Scoring (cont)

using the curve	<pre>precision, recall, thresholds = precision_recall_curve(y_test, svc.decision_function(X_test))</pre>
find threshold closest to zero	<pre>close_zero = np.argmin(np.ab- s(thresholds))</pre>
	<pre>plt.plot(precision[close_zero], recall[close_zero], 'o', marker- size=10, label="threshold zero", fillstyle="none", c='k', mew=2)</pre>
for random forest	<pre>precision_rf, recall_rf, thresh- olds_rf = precision_recall_c- urve(y_test, rf.predict_prob- a(X_test)[: , 1]) plt.plot(precision_rf[close_de- fault_rf], recall_rf[close_defau- lt_rf], '^', c='k', markersize=10, label="threshold 0.5 rf", fillst- yle="none", mew=2)</pre>
	<pre>plt.xlabel("Precision") plt.ylabel- ("Recall") plt.legend(loc="bes- t")</pre>
averag- e_precisi- on_score (area under the curve)	<pre>from sklearn.metrics import average_precision_score</pre>

Evaluation Metrics and Scoring (cont)

	<pre>ap_rf = average_precision_score(- y_test, rf.predict_proba(X_test)[: , 1]) ap_svc = average_precision_sco- re(y_test, svc.decision_function- (X_test))</pre>
ROC curve	<pre>from sklearn.metrics import roc_curve fpr, tpr, thresholds = roc_curve- (y_test, svc.decision_function(X_- test)) plt.plot(fpr, tpr, label="ROC Curve") close_zero = np.argmin(np.abs(t- hresholds)) plt.plot(fpr[close_zero], tpr[close- _zero], 'o', markersize=10, label="threshold zero", fillstyle- ="none", c='k', mew=2)</pre>
ROC curve's AUC	<pre>from sklearn.metrics import roc_auc_score rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[: , 1]) svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))</pre>



Evaluation Metrics and Scoring (cont)

Micro average computes the total number of false positives, false negatives, and true positives over all classes, and then computes precision, recall, and fscore using these counts.

```
f1_score(y_test, pred,
         average="micro"))
```

Macro average omputes the unweighted per-class f-scores. This gives equal weight to all classes, no matter what their size is.

```
f1_score(y_test, pred,
         average="macro"))
```

To change how to evaluate function in CV and grid search add the following argument to functions, such as, `ross_val_score`

```
scoring="accuracy"
```

If you do set a threshold, you need to be careful not to do so using the test set. As with any other parameter, setting a decision threshold on the test set is likely to yield overly optimistic results. Use a validation set or cross-validation instead.

Iris data set

importing data set from `sklearn.datasets`
import `load_iris`

```
iris_dataset = load_iris()
```

data set keys (`iris_dataset.keys()`)

Iris data set (cont)

Split the data into training and testing

```
X_train, X_test, y_train, y_test =
train_test_split( iris_dataset['d-
ata'], iris_dataset['target'], train_
size=0.8, test_size=0.2, random-
_state=0, shuffle=True(default,
shuffles the data),stratify=Non-
e(default))
```

scatter matrix `pd.plotting.scatter_matrix(iris_d-
ataframe, c=y_train, figsize=(15,
15), marker='o', hist_kwds={'bins':
20}, s=60, alpha=.8 (transpar-
ency), cmap=mglearn.cm3)`

Supervised Learning

classification n, the goal is to predict a class label, which is a choice from a predefined list of possibilities

regression the goal is to predict a continuous number, or a float-ing-point number in progra-mming terms (or real number in mathematical terms)

graphic that shows nearest neighbor `mglearn.plots.plot_knn_clas-
sification(n_neighbors=1)`

Preprocessing and Scaling

Importing from `sklear-
n.preproc-
essing`
import `MinMax-
Scaler`

Shifts the data such that all features are exactly between 0 and 1

```
scaler =
MinMaxSca-
ler(copy=-
True, featur-
e_range=(0,
1))
```

```
scaler.fit(X-
_train)
```

To apply the transformation that we just learned—that is, to actually scale the training data—we use the transform method of the scaler

```
scaler.trans-
form(X-
_train)
```

To apply the SVM to the scaled data, we also need to transform the test set.

```
X_test-
_scaled =
scaler.trans-
form(X_test)
```

learning an SVM on the scaled training data

```
svm =
SVC(C=100)
```

```
svm.fit(X_tr-
ain_scaled,
y_train)
```

Importing

```
from sklear-
n.preproc-
essing
import
StandardS-
caler
```

preprocessing using zero mean and unit variance scaling

```
scaler =
StandardS-
caler()
```

Ridge regression

Ridge regression is a model tuning method that is used to analyse any data that suffers from multicollinearity. This method performs L2 regularization. When the issue of multicollinearity occurs, least-squares are unbiased, and variances are large, this results in predicted values being far away from the actual values.



By **Remidy08**
cheatography.com/remidy08/

Not published yet.
Last updated 6th September, 2022.
Page 4 of 9.

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
<http://crosswordcheats.com>

Ridge regression (cont)

Importing	from sklearn.linear_model import Ridge
Train	ridge = Ridge().fit(X_train, y_train)
R ²	ridge.score(X_train, y_train)

plt.hlines(y-indexes where to plot the lines=0, xmin=0, xmax=len(lr.coef_))	Plot horizontal lines at each y from xmin to xmax.
---	---

The Ridge model makes a trade-off between the simplicity of the model (near-zero coefficients) and its performance on the training set. How much importance the model places on simplicity versus training set performance can be specified by the user, using the alpha parameter. Increasing alpha forces coefficients to move more toward zero, which decreases training set performance but might help generalization.

Linear models for classification

Importing logistic regression	from sklearn.linear_model import LogisticRegression
Train	LogisticRegression(C=100).fit(X_train, y_train)
Score	logreg.score(X_train, y_train)
Predict	y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)

Importing SVM	from sklearn.svm import LinearSVC
---------------	--------------------------------------

Using low values of C will cause the algorithms to try to adjust to the "majority" of data points, while using a higher value of C stresses the importance that each individual data point be classified correctly.

Grid Search

validation set	X_trainval, X_test, y_trainval, y_test = train_test_split(iris.data, iris.target, random_ _state=0)
----------------	---

	X_train, X_valid, y_train, y_valid = train_test_split(X_trainval, y_trainval, random_state=1)
--	---

Grid Search with Cross-Validation	from sklearn.model_selection import GridSearchCV
-----------------------------------	---

Training	grid_search = GridSearchCV(SVC(), param_grid, cv=5)
----------	---

Find best parameters	grid_search.best_params_
----------------------	--------------------------

return best score	grid_search.best_score_
-------------------	-------------------------

best_estimator_	access the model with the best parameters trained on the whole training set
-----------------	---

results of a grid search can be found in	grid_search.cv_results_
--	-------------------------

CV grid search	GridSearchCV(SVC(), param_grid, cv=5)
	param_grid = [{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}, {'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}, {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]

Grid Search (cont)

nested cross-validation	scores = cross_val_score(GridS- earchCV(SVC(), param_grid, cv=5), iris.data, iris.target, cv=5)
-------------------------	---

Grid search is a tuning technique that attempts to compute the optimum values of hyperparameters. It is an exhaustive search that is performed on a the specific parameter values of a model.

Decision trees

Importing data	from sklearn.tree import DecisionTreeClassifier
----------------	--

Tree	tree = DecisionTreeClassifier(random_state=0)
------	---

Train	tree.fit(X_train, y_train)
-------	----------------------------

Score	tree.score(X_train, y_train)
-------	------------------------------

Pre-pruning	Argument in DecisionTree- Classifier: max_depth=4
-------------	--

Other arguments	max_leaf_nodes, or min_sam- ples_leaf
-----------------	--

Import tree diagram	from sklearn.tree import export_graphviz
---------------------	---

Build tree diagram	export_graphviz(tree, out_fi- le="tree.dot", class_names=- ["malignant", "benign"], featur- e_names=cancer.feature_n- ames, impurity=False, filled- =True)
--------------------	---

Feature importance	tree.feature_importances_
--------------------	---------------------------

Predict	tree.predict(X_all)
---------	---------------------

Decision trees (cont)

Decision tree regressor importing	<code>from sklearn.tree import DecisionTreeRegressor</code>
Train	<code>DecisionTreeRegressor().fit(X_train, y_train)</code>
log	<code>y_train = np.log(data_train.price)</code>
exponential	<code>np.exp(pred_tree)</code>
Random Forest import	<code>from sklearn.ensemble import RandomForestClassifier</code>
Random Forest	<code>forest = RandomForestClassifier(n_estimators=5, random_state=2)</code>
Train	<code>forest.fit(X_train, y_train)</code>
gradient boosted trees import	<code>from sklearn.ensemble import GradientBoostingClassifier</code>
Gradient boost	<code>gbt = GradientBoostingClassifier(random_state=0)</code>
Train	<code>gbt.fit(X_train, y_train)</code>
Score	<code>gbt.score(X_test, y_test)</code>
Arguments	<code>max_depth, learning_rate</code>

often the default parameters of the random forest already work quite well. You can set `n_jobs=-1` to use all the cores in your computer in the random forest. In general, it's a good rule of thumb to use the default values: `max_features=sqrt(n_features)` for classification and `max_features=log2(n_features)` for regression. Gradient boosted trees are frequently the winning entries in machine learning competitions, and are widely used in industry. First use random than boost

Uncertainty Estimates from Classifiers

Evaluate the decision function for the samples in X.	<code>model.decision_function(X_test)[:6]</code>
Return the probability of classifying as all classes	<code>model.predict_proba(X_test)[:6]</code>

A model is called calibrated if the reported uncertainty actually matches how correct it is—in a calibrated model, a prediction made with 70% certainty would be correct 70% of the time. To summarize, `predict_proba` and `decision_function` always have shape `(n_samples, n_classes)`—apart from `decision_function` in the special binary case. In the binary case, `decision_function` only has one column, corresponding to the “positive” class `classes_`.

Feature selection

Importing variance threshold	<code>from sklearn.feature_selection import VarianceThreshold</code>
Removing columns with high variance	<code>sel = VarianceThreshold(threshold=(.8 * (1 - .8)))</code>
	<code>sel.fit_transform(X)</code>
Select-KBest	removes all but the k highest scoring features
Select-Percentile	removes all but a user-specified highest scoring percentage of features using common univariate statistical tests for each feature: false positive rate <code>SelectFpr</code> , false discovery rate <code>SelectFdr</code> , or family wise error <code>SelectFwe</code> .

Feature selection (cont)

GenericUnivariateSelect	allows to perform univariate feature selection with a configurable strategy.
importing SelectKBest	<code>from sklearn.feature_selection import SelectKBest</code>
importing chi2	<code>from sklearn.feature_selection import chi2</code>
	<code>X_new = SelectKBest(chi2, k=2).fit_transform(X, y)</code>
Recursive feature elimination	<code>from sklearn.feature_selection import RFE</code>
	<code>rfe = RFE(estimator=svc, n_features_to_select=1, step=1)</code>
	<code>rfe.fit(X, y)</code>
Recursive feature elimination with cross-validation	<code>from sklearn.feature_selection import RFECV</code>
	<code>rfecv = RFECV(estimator=svc, step=1, cv=StratifiedKFold(2), scoring="accuracy", min_features_to_select=min_features_to_select,)</code>
import StratifiedKfold	<code>from sklearn.model_selection import StratifiedKFold</code>

Model Evaluation and Improvement

Importing cross validation
from sklearn.model_selection
import cross_val_score

Cross-validation
scores = cross_val_score(
(model without fit, data, target,
cv=5)

Summarizing cross-validation scores
scores.mean()

stratified k-fold cross-validation
In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset

Provides train/test indices to split data in train/test sets.
KFold(n_splits=5, *, shuffle=False, random_state=None)

cross_val_score(logreg,
iris.data, iris.target, cv=kfold))

Importing Leave-one-out cross-validation
from sklearn.model_selection
import LeaveOneOut

Leave-one-out cross-validation
loo = LeaveOneOut()

scores = cross_val_score(lo-
greg, iris.data, iris.target,
cv=loo)

shuffle-split cross-validation
each split samples train_size many points for the training set and test_size many (disjoint) point for the test set

Model Evaluation and Improvement (cont)

import shuffle-split
from sklearn.model_selection
import ShuffleSplit

shuffle_split = ShuffleSplit-
(test_size=.5, train_size=.5,
n_splits=10)

scores = cross_val_score(lo-
greg, iris.data, iris.target,
cv=shuffle_split)

takes an array of groups as argument that we can use
GroupKFold

Import GroupKFold
from sklearn.model_selection
import GroupKFold

scores = cross_val_score(lo-
greg, X, y, groups, cv=Gro-
upKFold(n_splits=3))

Predicting with cross-validation
sklearn.model_selection.cro-
ss_val_predict(estimator, X,
y=None, , groups=None,
cv=None, n_jobs=None,
verbose=0, fit_params=None, pre_dispa-
tch='2n_jobs', method='pred-
ict')

Multilayer perceptrons (MLPs) or neural networks

Importing
from sklearn.neural_network
import MLPClassifier

Multilayer perceptrons (MLPs) or neural networks (cont)

Train mlp = MLPClassifier(algorithm='l-
bfgs', activation='tanh', random_st-
ate=0, hidden_layer_sizes=[10,1-
0]).fit(X_train, y_train)

there can be more than one hidden layers, for this, use a list on the hidden_layer_sizes
If we want a smoother decision boundary, we could add more hidden units, add a second hidden layer, or use the tanh nonlin-
earity

Naive Bayes Classifiers

Importing
from sklearn.naive_bayes
import GaussianNB

Train and predict
y_pred = gnb.fit(X_train, y_train).predict(X_test)

Function
class sklearn.naive_bayes.G-
aussianNB(*, priors=None,
var_smoothing=1e-09)

There are three kinds of naive Bayes classifiers implemented in scikit-learn: GaussianNB, BernoulliNB, and MultinomialNB. GaussianNB can be applied to any continuous data, while BernoulliNB assumes binary data and MultinomialNB assumes count data (that is, that each feature represents an integer count of something)

Linear models for multiclass classification

Importing
from sklearn.svm import
LinearSVC

Train linear SVC
linear_svm = LinearSVC().f-
it(X, y)

Import SVC
from sklearn.svm import
SVC

Linear models for multiclass classification (cont)

train `svm = SVC(kernel='rbf' (function to use with the kernel trick), C=10 (regularization parameter) , gamma=0.1 (controls the width of the Gaussian kernel)).fit(X, y)`

plot support vectors `sv= svm.support_vectors_`

class labels of support vectors are given by the sign of the dual coefficients `sv_labels = svm.dual_coef_[0].ravel() > 0`

Rescaling method for kernel SVMs `min_on_training = X_train.min(axis=0)`

`range_on_training = (X_train - min_on_training).max(axis=0)`

`X_train_scaled = (X_train - min_on_training) / range_on_training`

`X_test_scaled = (X_test - min_on_training) / range_on_training`

common

technique to extend a binary classification algorithm to a multiclass classification algorithm is the one-vs.-rest approach. In the one-vs.-rest approach, a binary model is learned for each class that tries to separate that class from all of the other classes, resulting in as many binary models as there are classes.

Lasso

Lasso using the lasso also restricts coefficients to be close to zero, but in a slightly different way, called L1 regularization. The consequence of L1 regularization is that when using the lasso, some coefficients are exactly zero. This means some features are entirely ignored by the model.

Importing `from sklearn.linear_model import Lasso`

Train `lasso = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)`

R² `lasso.score(X_train, y_train)`

Coefficients used `np.sum(lasso.coef_ != 0)`

Figure legend `plt.legend()`

In practice, ridge regression is usually the first choice between these two models. However, if you have a large amount of features and expect only a few of them to be important, Lasso might be a better choice. Note: There is a class called ElasticNet, which combines the penalties of Lasso and Ridge.

Linear models for regression

Importing `from sklearn.linear_model import LinearRegression`

Split data set (from `sklearn.model_selection` import `train_test_split(X, y, random_state=42)`)

Linear models for regression (cont)

linear regression `lr = LinearRegression().fit(X_train, y_train)`

slope `lr.coef_`

interception `lr.intercept_`

R² `lr.score(X_train, y_train)`

scikit-learn always stores anything that is derived from the training data in attributes that end with a trailing underscore. That is to separate them from parameters that are set by the user.

k-nearest neighbors

Importing `from sklearn.neighbors import KNeighborsClassifier`

k-nearest neighbors `knn = KNeighborsClassifier(n_neighbors=1(number of neighbors))`

Building a model on the training set `knn.fit(X_train, y_train)`

The fit method returns the knn object itself (and modifies it in place), so we get a string representation of our classifier. The representation shows us which parameters were used in creating the model.

Predictions `prediction = knn.predict(data)`

Accuracy `np.mean(y_pred == y_test)`
`knn.score(X_test, y_test)`

The k-nearest neighbors classification algorithm is implemented in the `KNeighborsClassifier` class in the `neighbors` module.