A Guide to Scikit Learn

Lab Objective: *The scikit-learn package is the one of the fundamental tools in Python for machine learning. In this appendix we highlight and give examples of some of the more popular scikit-learn tools for classification and regression, training and testing, and complex model construction.*

> **Note**
>
> This guide corresponds to scikit-learn version 0.19.1, released in October of 2017. For current release notes, see `http://scikit-learn.org/stable/whats_new.html`. Install scikit-learn (the `sklearn` module) with `conda install scikit-learn`.

Base Classes

The general problem in machine learning is to construct a function $f : X \to Y$, called a *model* or *estimator*, that accurately represents properties of given data. The domain X is usually $\mathbb{R}^D$, and the range Y is typically either $\mathbb{R}$ (regression) or a subset of $\mathbb{Z}$ (classification). The model is trained on N *samples* $(\mathbf{x}_i)_{i=1}^N \subset X$ that usually (but not always) have N accompanying *labels* $(y_i)_{i=1}^N \subset Y$.

Scikit-learn takes a highly object-oriented approach to machine learning models. Every major Scikit-learn class inherits from `sklearn.base.BaseEstimator` and conforms to the following conventions.

1. The constructor `__init__()` receives parameter arguments for the classifier (number of neighbors for k-nearest neighbors, number of trees for random forests, and so on). Each parameter of `__init__()` must have a default value (i.e. every argument is a keyword argument), and each argument must be saved as an instance variable of the **same name** as the parameter.

2. The `fit()` method constructs the model f. It receives an $N \times D$ matrix X and, optionally, a vector $\mathbf{y}$ with N entries. Each row $\mathbf{x}_i$ of X is one sample with corresponding label y_i. By convention, `fit()` always returns `self`.

Along with the `BaseEstimator` class, there are several other "mix in" base classes in `sklearn.base` that define specific kinds of models. The three listed below are the most common.[1]

[1] See `http://scikit-learn.org/stable/modules/classes.html#base-classes` for the complete list.

- `ClassifierMixin`: for *classifiers*, estimators that take on discrete values.
- `RegressorMixin`: for *regressors*, estimators that take on continuous values.
- `TransformerMixin`: for preprocessing raw data before estimation.

Classifiers and Regressors

The `ClassifierMixin` and `RegressorMixin` both require a `predict()` method that acts as the actual model f. That is, `predict()` receives an $N \times D$ matrix X and returns N predicted labels $(y_i)_{i=1}^N$, where y_i is the label corresponding to the ith row of X.

Both of these base class have a predefined `score()` method that uses `predict()` to test the accuracy of the model. It accepts $N \times D$ test data and a vector of N corresponding labels, then reports either the classification accuracy (for classifiers) or the R^2 value of the regression (for regressors).

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.model_selection import train_test_split

# Load the iris dataset and split it into training and testing groups.
>>> iris = load_iris()
>>> X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target)
>>> print(X_train.shape, y_train.shape)
(112, 4) (112,)
>>> print(X_test.shape, y_test.shape)
(38, 4) (38,)

# Train a Gaussian Naive Bayes classifier on the iris training data.
>>> gnb = GaussianNB().fit(X_train, y_train)    # fit() returns the object.

# Test the classifier on the iris testing data.
>>> gnb.predict(X_test[:6])
array([1, 0, 0, 1, 0, 2])       # predicted labels for the first 6 samples.
>>> gnb.score(X_test, y_test)
0.94736842105263153             # predict() chooses 94.7% of the labels right.
```

The consistent structure in the various `sklearn` classes makes it easy to use a wide variety of estimators. However, that structure also makes it possible to write custom estimators that behave like native `sklearn` objects. All you need to do is write good `fit()` and `predict()` methods.

The `fit()` method should do all of the "heavy lifting" by calculating the parameters of the model. By convention, these parameters should be stored in variables that end with an underscore, such as `self.popular_label_`. The `predict()` method uses the parameters defined in `fit()` to choose a label for test data.

```
>>> import numpy as np
>>> from collections import Counter
>>> from sklearn.base import BaseEstimator, ClassifierMixin

>>> class PopularClassifier(BaseEstimator, ClassifierMixin):
...     """Classifier that always guesses the most common training label."""
```

```
...     def __init__(self, verbose=False):
...         self.verbose = verbose      # Save parameters with the same name.
...
...     def fit(self, X, y):
...         """Find and store the most common label."""
...         self.popular_label_ = Counter(y).most_common(1)[0][0]
...         if self.verbose:
...             print("The best label is", self.popular_label_)
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always guess the most popular training label."""
...         M = 1 if X.ndim == 1 else X.shape[0]
...         return np.full(M, self.popular_label_)
...
# Train a PopularClassifier on the iris data.
>>> pc = PopularClassifier(verbose=True).fit(X_train, y_train)
The best label is 0

# Score the model on the testing data.
>>> pc.score(X_test, y_test)
0.18421052631578946             # A terrible classification rate (of course)!
```

This is a terrible classifier, but it is actually implemented as `sklearn.dummy.DummyClassifier` with `strategy="most_frequent"`. Note that `score()` was inherited from `ClassifierMixin`, so it returns a classification rate. In the next example, a simplification of `sklearn.dummy.DummyRegressor`, `score()` is inherited from `RegressorMixin`, so it returns an R^2 value.

```
>>> from sklearn.base import RegressorMixin

>>> class MeanRegressor(BaseEstimator, RegressorMixin):
...     """Regressor that always guesses the mean of training data."""
...     def __init__(self, shift=0):
...         self.shift = shift
...
...     def fit(self, X, y):
...         self.mean_ = np.mean(y) + self.shift
...         return self
...
...     def predict(self, X):
...         """Always guess the mean of the training data."""
...         M = 1 if X.ndim == 0 else X.shape[0]
...         return np.full(M, self.mean_)
...
# Train on the iris data (treating it as a regression problem).
>>> mr = MeanRegressor().fit(X_train, y_train)
>>> mr.mean_
0.9464285714285714
```

```
# Get the R^2 score of the regression on the testing data.
>>> mr.score(X_train, y_train)
-0.089188817792310138          # Also pretty terrible.
```

Classifier Name	Notable Hyperparameters
discriminant_analysis.LinearDiscriminantAnalysis	
ensemble.RandomForestClassifier	n_estimators, max_depth
linear_model.LogisticRegression	penalty, C
naive_bayes.GaussianNB	
naive_bayes.MultinomialNB	
neighbors.KNeighborsClassifier	n_neighbors
svm.SVC	C, kernel
tree.DecisionTreeClassifier	max_depth
Regressor Name	**Description**
ensemble.RandomForestRegressor	n_estimators, max_depth
linear_model.LinearRegression	
neighbors.KNeighborsRegressor	n_neighbors
svm.SVR	kernel
tree.DecisionTreeRegressor	max_depth

Table A.1: Common `sklearn` classifiers and regressors.

Problem 1. Take your Naïve Bayes classifier from your homework and rewrite it as a class that inherits from `BaseEstimator` and `ClassifierMixin`. Implement `__init__()`, `fit()`, and `predict()` in a way that matches `sklearn` conventions.

Test your model on the iris dataset.

Transformers

A scikit-learn *transformer* morphs data into better stuff.

```
>>> from sklearn.base import TransformerMixin

>>> class NormalizingTransformer(BaseEstimator, TransformerMixin):
...     def fit(self, X, y=None):
...         self.mu_ = np.mean(X, axis=0)
...         self.s_ = np.std(X, axis=0)
...         return self
...
...     def transform(self, X):
...         """Center each column of X at zero and normalize it."""
...         return (X - self.mu_) / self.s_
...
# Train / transform on some uniformly distributed data.
```

```
>>> nt = NormalizingTransformer()
>>> Z = nt.fit_transform(np.random.random((10,3)))
>>> Z.mean(axis=0)      # The columns of Z are centered at zero...
array([ -3.77475828e-16,   1.33226763e-16,  -2.49800181e-16])
>>> Z.var(axis=0)       # ...and have unit variance.
array([ 1.,  1.,  1.])

# Reuse the mean and standard deviation from before with new data.
>>> Z2 = nt.transform(np.random.random((5, 3)))
```

This particular transformer is also implemented as `sklearn.preprocessing.StandardScaler`.

Problem 2. Write a transformer class where the `fit()` and `transform()` methods takes in X as a pandas Data Frame. For each numerical column, replace any `nan` entries with the mean of the column. Drop string columns. Return the data as a NumPy array.

Validation Tools

Cross Validation

The `sklearn.model_selection` module has several useful tools for testing models quickly.

- `train_test_split()` randomly splits data into training and testing sets (demonstrated previously).
- `cross_val_score()` splits the data, trains the model, and scores the model several times, reporting the score of each trial.
- `cross_validate()` does the same thing as `cross_val_score()`, but it also reports the time it took to fit, the time it took to score, and the scores for the test set as well as the training set.

```
>>> from sklearn.model_selection import cross_val_score, cross_validate

# Make (but do not train) a classifier to test.
>>> gnb = GaussianNB()

# Test the classifier on the iris dataset 4 times.
>>> cross_val_score(gnb, iris.data, iris.target, cv=4)
array([ 0.94871795,  0.94871795,  0.91666667,  1.        ])

# Get more details on the train/test procedure.
>>> cross_validate(gnb, iris.data, iris.target, cv=4)
{'fit_time': array([ 0.0007441 ,  0.00114608,  0.00094104,  0.0006249 ]),
 'score_time': array([ 0.00034094,  0.00044894,  0.00047588,  0.00038815]),
 'test_score': array([ 0.94871795,  0.94871795,  0.91666667,  1.        ]),
 'train_score': array([ 0.96396396,  0.96396396,  0.97368421,  0.95614035])}
```

Reading: `http://scikit-learn.org/stable/tutorial/statistical_inference/model_selection.html`.

Problem 3. Use `cross_validate()` to score your class from Problem 1 on the iris dataset. Do the same for a `LogisticRegressionClassifier`.

Grid Search

Problem 4. Take the cancer data set (`datasets.load_breast_cancer()`) and do a grid search on an SVM (`sklearn.linear.svm`) with the parameter `C` as .01, .1, or 1, and the parameter `kernel` as `"linear"`, `"poly"`, `"rbf"`, and `"sigmoid"`.

What is the best choice of parameters? How well does the corresponding model do?

Pipelines

Reading: `http://scikit-learn.org/stable/tutorial/statistical_inference/putting_together.html`

Watching (optional):

- `https://www.youtube.com/watch?v=KLPtEBokqQ0`
- `https://www.youtube.com/watch?v=URdnF1ZnlaE`

Problem 5. Make a pipeline of your transformer from Problem 2, a normalizing scaler transformer (`preprocessing.StandardScaler`), a PCA transformer (`decomposition.PCA`), and an SVM classifier (`svm.SVC`). Using the titanic dataset (read in as a pandas DataFrame), do a grid search for the best model, varying your parameters however you see fit.

What is your best choice of parameters? How well does the corresponding model do?

Extra credit to the student with the very best model![a] To compete, pick your best parameters, do a cross validation with 10 folds, and take the average of the test scores.

[a] This is essentially what a Kaggle competition is.