Modeling in Scikit-learn

Neba Nfonsang

University of Denver

2020

```
# Import necessary packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline

from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LinearRegression

from sklearn.model_selection import (train_test_split, GridSearchCV,
cross_val_score, cross_val_predict, cross_val_predict, validation_curve)

from sklearn import metrics
```

Introduction

This is a demonstration of how to construct and evaluate predictive models in Data Science and Machine Learning using scikit-learn. Scikit-learn is an open source machine learning library in Python that supports supervised and unsupervised learning. Scikit-learn contains simple and efficient tools for predictive data analysis; hence, makes the implementation of the data science workflow very easy. The package is built on NumPy, SciPy and matplotlib, and can be used with other data analysis packages in Python such as pandas.

Creating a Dataset

Let's randomly create some data that would be used to illustrate the modeling process in scikit-learn.

```
np.random.seed(2000)
X = np.random.randint(0, 100, size=1000).reshape(500, 2)
y= np.random.randint(0, 2, size=500).reshape(500, 1)
data = np.concatenate((X, y), axis=1)
data = pd.DataFrame(data, columns=["x1", "x2", "y"])
data.head()
```

```
##    x1  x2  y
## 0  54  72  1
## 1  78  77  0
## 2  54  28  1
## 3  28  50  1
## 4  55  14  0
```

Splitting the Data into Training and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(
                  data[["x1", "x2"]], data["y"].values,
                        test_size=0.30, random_state=10)

print(X_train.shape, y_train.shape)
```

```
## (350, 2) (350,)

print(X_test.shape, y_test.shape)

## (150, 2) (150,)
```

Constructing a Classification Model

We will construct a k-nearest neighbor classifier using the KNeighborsClassifier constructor from the sklearn.neighbors module. To predict a test instance, the k-nearest neighbor classifier uses k nearest instances to the test instance. The class labels of the k nearest instances are checked and the class with majority votes is returned as the predicted class for the test instance. The code for implementing k-nearest neighbors is as follows:

```
# create the knn estimator object
knn = KNeighborsClassifier(n_neighbors=5)


# use the knn estimator object to fit the model
knn = knn.fit(X_train, y_train)
```

Hyperparameters of the Model

Hyperparameters are parameters whose values are set a priori or before the model is fitted. In scikit-learn, the hyperparameters are set inside the Estimator constructor, when constructing the model. The hyperparamter values are passed as arguments into the **init** method of the Estimator class.

Values of the hyperparameters used in constructing the model can be retrieved using the `estimator.param_name` syntax. That means, the hyperparameters are attributes of the estimator object. For example; number of neighbors, k, used for constructing the k-nearest neighbor model can be retrieved as follows (by default, this value is k=5):

```
knn.n_neighbors

## 5
```

All the parameters of the constructed model can be retrieved in scikit-learn as follows:

```
knn.get_params()

## {'algorithm': 'auto', 'leaf_size': 30, 'metric': 'minkowski',
'metric_params': None, 'n_jobs': None, 'n_neighbors': 5, 'p': 2, 'weights':
'uniform'}
```

The scikit-learn documentation can always be used to understand what the hyperparameters represent and what type of values should be passed as arguments for the hyperparameters.

Predictions

Predictions in scikit-learn are implemented using the `estimator.predict()` method, which takes the input data as an argument. It is important to note that predictions can be made for both the training and test set. However, we are more interested in predicting the output of test instances. These predicted output for a training set and test set could be used to compute the overall accuracy score of the model on the training set and test set respectively. The k-nearest neighbor model can be used to make predictions on the training and test set as follows (only the first 20 output values are displayed):

```
# prediction on the training set
knn.predict(X_test)[0:20]


# prediction on the test set

## array([0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1])

knn.predict(X_test)[0:20]

## array([0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1])
```

Model Evaluation

Model evaluation involves assessing the performance of a model for various purposes. For example; the performance of a model can be assessed for:

- estimating the performance or accuracy score of the model on the test set,
- checking for overfitting,
- model selection through tuning hyperparameter tuning (selecting the model with an optimal hyperparameter value from candidate models of the same form), and
- model comparison (choosing the model with the best performance from alternative models having different forms).

There are various performance measure functions inside the sklearn.metrics module for assessing the performance of a model. The metrics module in sklearn is usually imported using the syntax:

```
from sklearn import metrics
```

Performance Measures for a Classification Model

Classification models can be assessed using numerical measure or graphs. The functions in sklearn.metrics module for computing numerical measures of performance include:

- `accuracy_score()`
- `zero_one_loss()`
- `confusion_matrix()`
- `precision_score()`
- `recall_score()`
- `f1_score()`
- `precision_recall_fscore_support`
- `classification_report()`

Functions inside sklearn.metrics module for assessing model performance graphically include:

- `plot_confusion_matrix()`
- `plot_precision_recall_curve()`
- `plot_roc_curve()`

Performance Measures for Regression

Numerical measures for assessing regression models with continuous output include mean absolute error, mean square error and r-squared. The following functions in sklearn.metrics are used to assess the performance of a regression model:

- `mean_absolute_error()`
- `mean_squared_error()`
- `r2_score`

Other Useful Functions Inside the sklearn.metrics Module

- `euclidean_distances()`: used to compute the Euclidean distance between two instances.
- `pairwise_distances()`: used to compute the distance between an instance and other instances in a matrix.
- `pairwise_distances_argmin()`: used to compute the minimum distance between an instance and a vector of instances.
- `cohen_kappa_score()`: use to compute inter-rater reliability or the level of agreement between two raters or annotators of a classification on a classification problem (an annotator refer to a person labeling a set of training examples).

- etc…

Overfitting

Overfitting happens when the model performance on the training set is far better than the performance of the model on the test set. Overfitting helps us to understand whether the model generalizes well to outside examples not used in model construction. The accuracy scores on the training and test sets can be used to check for overfitting. If the overall accuracy of the model on the test set is far less than the accuracy of the model on the training set, then, it is likely that the model overfits the training set.

Accuracy Scores

Accuracy scores measure the overall accuracy of a model. In classification, an accuracy score represents the proportion of correct predictions in the data. The accuracy score of a classification model is computed using the syntax, `metrics.accuracy_score(y, y_pred)`, where y is the actual output value and y_pred is the predicted output value. The accuracy scores can be computed on the training and test sets as follows:

```
# prediction on the test set
y_test_pred = knn.predict(X_test)

# prediction on the training set
y_train_pred = knn.predict(X_train)

# accuracy on training set
accuracy_train = metrics.accuracy_score(y_train, y_train_pred)
accuracy_train = np.around(accuracy_train, 4)

# accuracy on test set
accuracy_test = metrics.accuracy_score(y_test, y_test_pred)
accuracy_test = np.around(accuracy_test, 4)

print("Accuracy on training set: ", accuracy_train)

## Accuracy on training set:  0.6714

print("Accuracy on test set: ", accuracy_test)

## Accuracy on test set:  0.48
```

Do you think the model overfits the training set?

Alternatively, the accuracy score can be computed using the `.score()` method of the estimator. The general syntax for the `.score()` method is `estimator.score(X, y)`. The following code shows how the `estimator.score()` is implemented.

```
# accuracy on training set
accu_train = knn.score(X_train, y_train)
accu_train = np.around(accu_train, 4)

# accuracy on test set
accu_test = knn.score(X_test, y_test)
accu_test = np.around(accu_test, 4)

print("Accuracy on training set: ", accu_train)

## Accuracy on training set:  0.6714

print("Accuracy on test set: ", accu_test)

## Accuracy on test set:  0.48
```

Error Rate

The overall error rate of a classification model is the number of incorrect predictions on a given dataset. The overall error rate can be computed on the training set or test set using the formula:

Overall error rate = 1 - overall accuracy

In scikit-learn, the `metrics.zero_one_loss(y, y_pred)` function is used to compute the error rate, where y is the actual output and y_pred is the predicted output. For example, error rate for a k-nearest neighbor model can be computed as follows:

```
# construct and fit the model
knn = KNeighborsClassifier(n_neighbors=5)
knn = knn.fit(X_train, y_train)

# make predictions on training and test sets
y_train_pred = knn.predict(X_train)
y_test_pred = knn.predict(X_test)
```

```
# compute error rates on training and test sets
error_train = metrics.zero_one_loss(y_train, y_train_pred)
error_train = np.around(error_train,4)

error_test = metrics.zero_one_loss(y_test, y_test_pred)
error_test = np.around(error_test, 4)

print("Training Error: ", error_train)

## Training Error:  0.3286

print("Test Error: ", error_test)

## Test Error:  0.52
```

Cross Validation

Cross validation is another approached used for evaluating the performance of a model. This approach is advantageous when the training set has a small sample size. Cross validation maximizes the number of training examples to produce better accuracy score estimates. There are different types of cross validation.

K-fold cross validation is a common type of cross validation where the training set is partitioned into k-folds or groups. The model is trained iteratively, and the number of iterations equals the number of folds, k. During each iteration, a single fold is selected as the validation (or test) set and the rest of the folds are used for training. After the model is trained, the accuracy of the model is measured on the validation (or test) set and recorded. The process is repeated until all the folds are used as validation (or test) sets.

The final cross validation accuracy score of the model on the validation (or test) set is computed by averaging the cross-validation accuracy scores for all the iterations. This average or final accuracy on the validation (or test) set is a better estimate of the model's generalization performance. In scikit-learn, the `cross_val_score()` function inside the sklearn.model_selection module is used to implement cross validation, to compute the average performance accuracy score of the model on the validation (or test) set. Some

important arguments that passed into the `cross_val_score()`function include the `estimator`,`X`, `y`, `scoring`, and `cv`. By default, `scoring=None`, which implies the estimator's `.score()` method would be used. Alternatively, `scoring="Accuracy"` will invoke the estimator's .score() method. The default value of the `cv` hyperparameter is 5.

The The cross-validation score of a k-nearest neighbor model can be obtained as shown below:

```
# construct and fit the model
knn = KNeighborsClassifier()
knn = knn.fit(X_train, y_train)

# get the cross accuracy score on the validation set for each iteration
scores = cross_val_score(estimator=knn, X=X_train, y=y_train,
scoring="accuracy", cv=8)

scores

## array([0.40909091, 0.63636364, 0.38636364, 0.52272727, 0.47727273,
##        0.43181818, 0.58139535, 0.62790698])
```

The scores obtained are the predicted accuracy scores on the validation (or test) fold for each iteration. A typical generalization accuracy of the model is computed by averaging the accuracy scores for all the iteration. In code, this can be implemented using the `np.mean()` function or call the mean method on the numpy array object, `array_object.mean()`:

```
cross_val_acc = np.mean(scores)
cross_val_acc = np.around(cross_val_acc, 4)
print("Cross validation accuracy score: ", cross_val_acc)
# np.around(scores.mean(), 4) could also be used to compute the mean

## Cross validation accuracy score:  0.5091
```

Hyperparameter Tuning and Model Selection

Different approaches can be used to tune the hyperparameters of the model. Hyperparameter tuning requires a training set for training the model and a validation set for evaluating the performance of the model. Let's examine four different approaches of tuning hyperparameters.

Use a for loop to construct models with various hyperparameters A for loop can be used to iteratively construct and fit models with various possible parameter values. We will need to first split the training set into a smaller training set and a validation set. The performance of each model is then evaluated and the hyperparameter corresponding to the model with the best performance accuracy score on test set is selected as the optimal hyperparameter. The best performing model is also selected, and its performance is evaluated on an outside sample not used for training or validation. This approach is efficient for tuning a single hyperparameter. If there are multiple hyperparameters to be tuned, other approaches such as grid search cross validation in scikit-learn can be used. Generally, to implement hyperparameter tuning, we need to first split the training data into a smaller training set and a validation set.

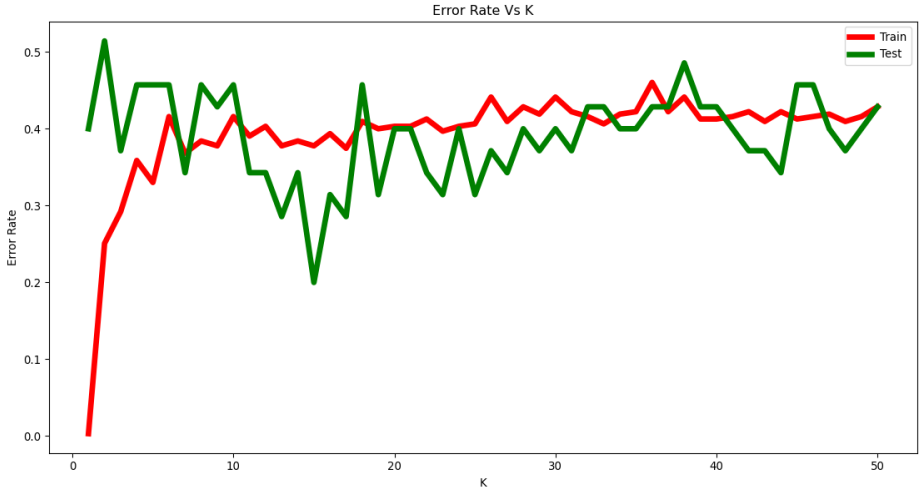
```
# split the training set into smaller training set and validation sets
X_train2, X_val, y_train2, y_val = train_test_split(
                  X_train, y_train, test_size=0.10, random_state=10)
```

The code below shows how to use a `for loop` to tune the hyperparameters of a model. We start by initializing a grid of parameters.

```
# construct and fit the model
k=range(1, 51)
train_accuracy = []
val_accuracy = []
for hyperparam in k:
    knn = KNeighborsClassifier(n_neighbors=hyperparam)
    knn = knn.fit(X_train2, y_train2)
```

```
    # accuracy scores on training and validation sets
    train_acc = knn.score(X_train2, y_train2)
    val_acc = knn.score(X_val, y_val)
    train_accuracy.append(train_acc)
    val_accuracy.append(val_acc)

# plot the error rates vs the hyperparameter values
plt.figure(figsize=(14, 7))
plt.plot(k, 1-np.array(train_accuracy), lw=5, c="red");
plt.plot(k, 1-np.array(val_accuracy), lw=5, c="green");
plt.legend(["Train", "Test"]);
plt.title("Error Rate Vs K");
plt.xlabel("K");
plt.ylabel("Error Rate");
plt.show()
```



We can find the optimal k-value that gives the smallest error rate on the test set using the `np.argmin()` function:

```
val_error = 1-np.array(val_accuracy)
k_best = k[np.argmin(val_error)]
print("Optimal value of k: ", k_best)

## Optimal value of k:  15
```

We can now construct the "best" model with the optimal hyperparameter value, k_best

```
knn_best = KNeighborsClassifier(n_neighbors=k_best)
knn_best = knn_best.fit(X_train2, y_train2)


print("Accuracy on training set: ", np.around(knn_best.score(X_train2,
y_train2), 4))

## Accuracy on training set:  0.6222

print("Accuracy on test set: ", np.around(knn_best.score(X_test, y_test), 4))

## Accuracy on test set:  0.48
```

Note that the accuracy scores obtained can be influence by the value of the random state for the random number generator used in splitting the data. That is, if the random state value is changed, a different sample with different sample characteristics will be obtained. This can result to unstable results or accuracy scores especially if the training sample size is small.

Using a `for loop` and cross validation for hyperparameter tuning

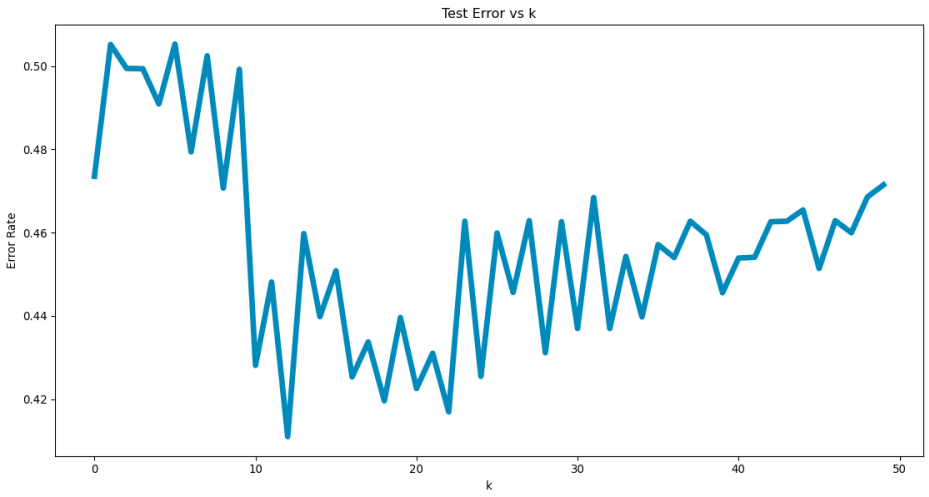
Cross validation can also be used to tune the hyperparameters of a model. The cross-validation accuracy score is estimated for models with different parameter values. The model with the best cross validation accuracy score is selected as the best model and the hyperparameter corresponding to the best model is selected as the optimal hyperparameter. A `for loop` and cross validation can be implemented as follows:

```
test_errors= []
for i in range(1, 51):
    knn = KNeighborsClassifier(n_neighbors=i)
```

```
    knn = knn.fit(X_train, y_train)
    scores = cross_val_score(estimator=knn, X=X_train,
                             y=y_train, scoring="accuracy", cv=8)
    test_errors.append(1-np.mean(scores))
```

The overall error of the model on the test set is plotted as shown below:

```
# plotting the error rate vs k
plt.figure(figsize=(14, 7));
plt.title("Test Error vs k");
plt.xlabel("k");
plt.ylabel("Error Rate");
plt.plot(test_errors, color="#008ABC", lw=5);
plt.show()
```



We can retrieve the optimal hyperparameter using `np.argmin()`.

```
# compute the optimal k
k = range(1, 51)
k_best1 =k[np.argmin(test_errors)]
print("Optimal value of k: ", k_best1)

## Optimal value of k:  13
```

The optimal hyperparameter value is then used to construct the best model (model with the best performance accuracy score on the test set).

```
# construct the best model and use it for prediction
knn_best = KNeighborsClassifier(n_neighbors=k_best1)
knn_best = knn.fit(X_train, y_train)
print("Accuracy score of the best model on the training set: ",
np.round(knn.score(X_train, y_train), 4))

## Accuracy score of the best model on the training set:  0.6

print("Accuracy score of the best model on the test set: ",
np.round(knn.score(X_test, y_test), 4))

## Accuracy score of the best model on the test set:  0.4533
```

Using grid search cross validation for hyperparameter tuning

Grid search cross validation can be implemented using the `GridSearchCV()` constructor inside the sklearn.model_selection module in scikit-learn. The `GridSearchCV()` constructor requires an estimator and a grid of hyperparameter values to be passed as arguments. The grid of hyperparameters are passed as a dictionary and more than one hyperparameter could be specified in the grid. The grid search cross validation is implemented as shown below:

```
knn = KNeighborsClassifier()
param_grid = {"n_neighbors":range(1,51)}


grid = GridSearchCV(estimator=knn, param_grid=param_grid, scoring=None, cv=8)
grid = grid.fit(X_train, y_train)


print("Optimal value of k: ", grid.best_params_)

## Optimal value of k:  {'n_neighbors': 13}

print("Accuracy on training set: ", grid.score(X_train, y_train))
```

```
## Accuracy on training set:  0.6485714285714286

print("Accuracy on test set: ", grid.score(X_test, y_test))

## Accuracy on test set:  0.49333333333333335
```

Note that the `.score()` and `.predict()` methods can be used with the grid search cross validation object:

```
# print the predicted output data given the test input data
grid.predict(X_test)[0:20]

## array([0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1])

# print the predicted output values given the training input data
grid.predict(X_test)[0:20]

## array([0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1])
```

Using the `validation_curve()` in scikit-learn for hyperparameter tuning

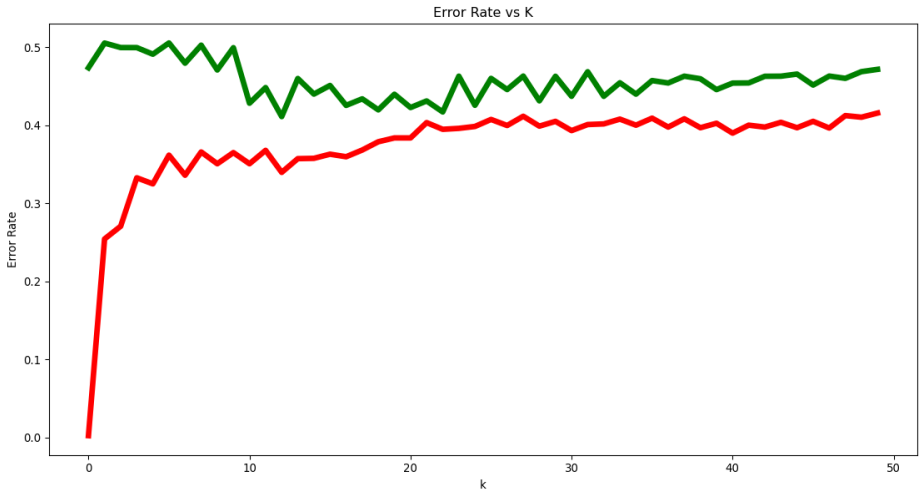
For each hyperparameter value in a grid of hyperparameters, the `validation_curve()` function inside the sklearn.model_selection module returns the accuracy scores for each cross-validation iteration. For each hyperparameter value, the accuracy scores for each cross-validation iteration need to be averaged to obtain the final cross validation accuracy scores on the training and validation sets. The training errors are computed from the accuracy scores and plotted with their corresponding k values.

```
knn = KNeighborsClassifier()


train_acc, test_acc = validation_curve(knn, X_train, y_train,
                                       param_name="n_neighbors",
                                       param_range=range(1, 51), cv=8)


train_acc_average = np.mean(train_acc, axis=1)
test_acc_average = np.mean(test_acc, axis=1)
```

```
plt.figure(figsize=(14, 7));
plt.title("Error Rate vs K");
plt.xlabel("k");
plt.ylabel("Error Rate");
plt.plot(1-train_acc_average, color="red", lw=5);
plt.plot(1-test_acc_average, color="green", lw=5);
plt.show()
```



Now, let's take a look at some of the tools inside the `sklearn.metrics` module that are used for assessing classification models.

Confusion Matrix

A confusion matrix is basically a cross tabulation of the actual output and the predicted output. The `confusion_matrix()` function inside the `sklearn.metrics` module is used to generate a confusion matrix:

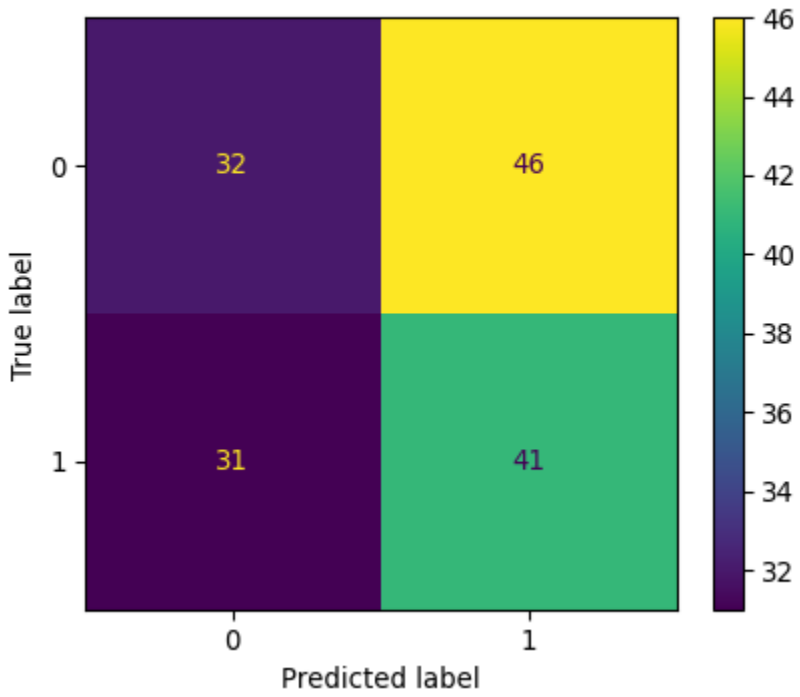
```
knn = KNeighborsClassifier(n_neighbors=15)
knn = knn.fit(X_train, y_train)
y_test_pred = knn.predict(X_test)

metrics.confusion_matrix(y_test, y_test_pred, labels=[0, 1])
```

```
## array([[32, 46],
##        [31, 41]], dtype=int64)
```

The confusion matrix can be visualized using a confusion matrix plot:

```
metrics.plot_confusion_matrix(knn, X_test, y_test);
plt.show()
```



When is it appropriate to use overall accuracy, precision, recall, and f1 score?

Note that overall accuracy is more useful for a balanced classification problem (with approximately the same number of positives and negatives in the data). For an imbalance classification problem, it is important to evaluate the model using precision, recall and f1 score.

Precision

Precision is the number of true positives (correctly classified positive cases) divided the number of cases predicted to be positive. $ Precision = $

The `precison_score()` function inside the `sklearn.metrics` module is used to compute the precision score:

```
precision = metrics.precision_score(y_test, y_test_pred, labels=None)
print("Precision: ", np.around(precision,4))

## Precision:  0.4713
```

Using the results on the confusion matrix plot, we can compute precision as:

```
precision = np.around(41/(41 + 46), 4)
print("Precision: ", precision)

## Precision:  0.4713
```

Recall

Recall is the number true positives divided by the number of positive cases in the data: $ Recall = $

The `recall_score()` function inside the `sklearn.metrics` module could be used to compute the recall score of a model:

```
recall = metrics.recall_score(y_test, y_test_pred, labels=None)
print("Recall: ", np.around(recall,4))

## Recall:  0.5694
```

Using the results on the confusion matrix plot, we can explicitly compute recall as:

```
recall = np.around(41/(41 + 31), 4)
print("Recall: ", recall)

## Recall:  0.5694
```

F1 score

It is possible to have good precision and poor recall or poor precision and good recall. F1 score can be used to capture recall and precision in a single score. F1 score is a

combination of recall and precision. F1 score reaches its best value at 1 and worst score at 0. The formula for the F1 score is:

$ F_1Score = $ The F1 score of a model can be computed using the `f1_score()` function inside the `sklearn.metrics` module:

```
f1_score = metrics.f1_score(y_test, y_test_pred)
f1_score = np.around(f1_score, 4)
print("F1 score: ", f1_score)

## F1 score:  0.5157
```

Using a formula, f1 score can be directly calculated as

```
f1 = 2*precision*recall/(precision + recall)
print("F1 score:", np.around(f1, 4))

## F1 score: 0.5157
```

classification report

Classification report contains information about precision, recall and f1 score for class 0 and class 1. A classification report is generated in sklearn as follows:

```
print(metrics.classification_report(y_test, y_test_pred))

##               precision    recall  f1-score   support
##
##            0       0.51      0.41      0.45        78
##            1       0.47      0.57      0.52        72
##
##     accuracy                           0.49       150
##    macro avg       0.49      0.49      0.48       150
## weighted avg       0.49      0.49      0.48       150
```

Constructing and Evaluating a Regression Model

This section discusses how regression models are generally constructed and evaluated in scikit-learn. The scikit learn framework of constructing and fitting models is the same for both regression and classification models. Basically, model objects called estimators are constructed using an Estimator class, and the `fit()` and method of the estimator are called on the estimator to fit the model. . ### Creating Data for Regression

```
np.random.seed(1234)
data2 = np.random.randint(low=0, high=100, size=7000).reshape(1000, 7)
data2 = np.around(data2/100, 2)
data2 = pd.DataFrame(data2,
                     columns=["x1", "x2", "x3", "x4", "x5", "x6", "x7"])
data2 = data2.assign(y= lambda x: .2*data2["x7"]**2)
data2.tail()
```

```
##        x1    x2    x3    x4    x5    x6    x7        y
## 995  0.10  0.87  0.20  0.39  0.29  0.64  0.55  0.06050
## 996  0.06  0.68  0.52  0.44  0.13  0.42  0.54  0.05832
## 997  0.24  0.04  0.18  0.96  0.03  0.18  0.14  0.00392
## 998  0.29  0.64  0.75  0.52  0.88  0.53  0.54  0.05832
## 999  0.35  0.60  0.81  0.60  0.67  0.83  0.41  0.03362
```

```
data2.iloc[:, 0:-1].values
```

```
## array([[0.47, 0.83, 0.38, ..., 0.76, 0.24, 0.15],
##        [0.49, 0.23, 0.26, ..., 0.43, 0.3 , 0.26],
##        [0.58, 0.92, 0.69, ..., 0.73, 0.47, 0.5 ],
##        ...,
##        [0.24, 0.04, 0.18, ..., 0.03, 0.18, 0.14],
##        [0.29, 0.64, 0.75, ..., 0.88, 0.53, 0.54],
##        [0.35, 0.6 , 0.81, ..., 0.67, 0.83, 0.41]])
```

Splitting the Data into Training and Test Sets

```
X_train2, X_test2, y_train2, y_test2 = train_test_split(
                  data2.iloc[:, 0:-1], data2["y"].values,
                        test_size=0.30, random_state=10)


print(X_train2.shape, y_train2.shape)

## (700, 7) (700,)

print(X_test2.shape, y_test2.shape)

## (300, 7) (300,)
```

Construct and Fit the Regression Model

```
reg = LinearRegression()
reg = reg.fit(X_train2, y_train2)


# Extract the Parameters of the Regression
print("Intercept: ", np.round(reg.intercept_, 4))

## Intercept:  -0.0342

print("Coefficients: ", np.round(reg.coef_, 4))

## Coefficients:  [ 0.0044  0.0003 -0.0008 -0.0015  0.0004  0.0026  0.1956]
```

Use Regression Model to Predict Output

```
# predict output of training set
y_train2_pred = reg.predict(X_train2)
y_train2_pred[0:20]

## array([ 0.0851329 ,  0.0150013 ,  0.051813  ,  0.08373183, -0.02517656,
##         0.16229868, -0.01900224,  0.05818082,  0.13544068,  0.09834299,
##         0.09561954,  0.13610667,  0.15370497,  0.10830301,  0.09055921,
##         0.14200097, -0.01598772,  0.02281856,  0.12073892,  0.09614222])
```

```
# predict output of test set
y_test2_pred = reg.predict(X_test2)
y_test2_pred[0:20]

## array([ 0.00048953,  0.00786294,  0.12560854,  0.0442773 ,  0.11699493,
##         0.13221465,  0.01588463,  0.00503851,  0.07726548,  0.05128265,
##         0.05826805,  0.04159423,  0.13921864,  0.06508612, -0.02579713,
##         0.15163879,  0.1103732 ,  0.04693609,  0.02935483,  0.08123976])
```

Compute the Accuracy of the Model

```
r_squared = metrics.r2_score(y_test2, y_test2_pred)
print("R-squared: ", np.around(r_squared, 4))

## R-squared:  0.9344
```

R-squared can also be obtained using the `.score()` method of the estimator:

```
# Accuracy of the regression model on the test set
acc_test2 = reg.score(X_test2, y_test2)
acc_test2 = np.around(acc_test2, 4)
print("Accuracy on test set: ", acc_test2)

## Accuracy on test set:  0.9344

# Accuracy of the regression model on the training set
acc_train2 = reg.score(X_train2, y_train2)
acc_train2 = np.around(acc_train2, 4)
print("Accuracy on trainnig set: ", acc_train2)

## Accuracy on trainnig set:  0.9321
```

Fit the Regression Model Using Cross Valiation

```
reg = LinearRegression()
reg = reg.fit(X_train2, y_train2)


# get the cross accuracy score on the validation set for each iteration
scores = cross_val_score(estimator=reg, X=X_train2, y=y_train2, scoring="r2",
```

```
cv=10)


scores

## array([0.91291371, 0.92380188, 0.93752187, 0.92814238, 0.92497196,
##        0.94504254, 0.94223414, 0.92604978, 0.92470762, 0.93237634])

# cross validation score
r_squared_cv = scores.mean()
print("Cross validation score: ", r_squared_cv)

## Cross validation score:  0.9297762218121969
```

Predictions with Cross Validation

A regression model constructed with cross validation can be used to predict the output values of input data as follows:

```
y_test_pred_cv = cross_val_predict(reg, X_test2, y_test2)
y_test_pred_cv = np.around(y_test_pred_cv, 4)
y_test_pred_cv[0:20]

## array([ 1.000e-04,  4.800e-03,  1.223e-01,  4.620e-02,  1.154e-01,
##         1.319e-01,  1.540e-02,  3.500e-03,  7.850e-02,  5.050e-02,
##         5.790e-02,  4.030e-02,  1.396e-01,  6.400e-02, -2.690e-02,
##         1.521e-01,  1.096e-01,  4.700e-02,  2.920e-02,  7.980e-02])
```

Using Pipeline to Chain Multiple Steps in the Data Analysis Workflow

Let's transform the data using a min-max scaler, then construct and fit a regression model using Pipeline. The `Pipeline()` constructor takes a list of (name, object) tuples where the last object is the estimator. The objects passed in the `Pipeline()` may include a scaler or an estimator. The fit and transform methods can be applied on a pipeline object.

```
chain = [("scaler", MinMaxScaler()), ("lin_reg", LinearRegression())]
pipe = Pipeline(chain)
pipe = pipe.fit(X_train2, y_train2)
```

```
test_acc2 = pipe.score(X_train2, y_train2)
print("Accuracy on test set: ", np.around(test_acc2, 4))

## Accuracy on test set:  0.9321

# cross validation with pipeline
pipe_cv = Pipeline(chain)
scores = cross_val_score(pipe_cv, X_train2, y_train2)
print("Cross validation score: ", np.around(scores.mean(), 4))

## Cross validation score:  0.9301
```

Conclusion

We have explored scikit-learn's framework for constructing and evaluating predictive models (classification and regression models). We also examined how to tune hyperparameters in relation to model selection. Finally, we demonstrated how to chain multiple steps in the data analysis workflow using `Pipeline()`. The framework explored so far can be applied to other types of classification and regression problems beyond the k-nearest neighbors and linear regression examples analyzed.