student_intervention

June 18, 2016

1 Machine Learning Engineer Nanodegree

1.1 Supervised Learning

1.2 Project 2: Building a Student Intervention System

Welcome to the second project of the Machine Learning Engineer Nanodegree! In this notebook, some template code has already been provided for you, and it will be your job to implement the additional functionality necessary to successfully complete this project. Sections that begin with **'Implementation'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section and the specifics of the implementation are marked in the code block with a `'TODO'` statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

1.2.1 Question 1 - Classification vs. Regression

Your goal for this project is to identify students who might need early intervention before they fail to graduate. Which type of supervised learning problem is this, classification or regression? Why?

Answer: A regression problem would consists of a typically numerical solution range, and a classification problem would consist of choices referred to as classes. A machine learning problem where students are to be identified as 'needs intervention' or 'doesn't need intervention' is a classification problem.

1.3 Exploring the Data

Run the code cell below to load necessary Python libraries and load the student data. Note that the last column from this dataset, `'passed'`, will be our target label (whether the student graduated or didn't graduate). All other columns are features about each student.

```
In [7]: # Import libraries
        import numpy as np
        import pandas as pd
        from time import time
        from sklearn.metrics import f1_score

        # Read student data
        student_data = pd.read_csv("student-data.csv")
        print "Student data read successfully!"
```

```
Student data read successfully!
```

1.3.1 Implementation: Data Exploration

Let's begin by investigating the dataset to determine how many students we have information on, and learn about the graduation rate among these students. In the code cell below, you will need to compute the following: - The total number of students, `n_students`. - The total number of features for each student, `n_features`. - The number of those students who passed, `n_passed`. - The number of those students who failed, `n_failed`. - The graduation rate of the class, `grad_rate`, in percent (%).

```
In [8]: # TODO: Calculate number of students
        n_students = len(student_data[:])

        # TODO: Calculate number of features
        n_features = len(student_data.columns[:-1])

        # TODO: Calculate passing students
        n_passed = len(student_data[student_data['passed']=='yes'])

        # TODO: Calculate failing students
        n_failed = len(student_data[student_data['passed']=='no'])

        # TODO: Calculate graduation rate
        grad_rate = 100*n_passed/(n_passed+n_failed)

        # Print the results
        print "Total number of students: {}".format(n_students)
        print "Number of features: {}".format(n_features)
        print "Number of students who passed: {}".format(n_passed)
        print "Number of students who failed: {}".format(n_failed)
        print "Graduation rate of the class: {:.2f}%".format(grad_rate)
```

```
Total number of students: 395
Number of features: 30
Number of students who passed: 265
Number of students who failed: 130
Graduation rate of the class: 67.00%
```

1.4 Preparing the Data

In this section, we will prepare the data for modeling, training and testing.

1.4.1 Identify feature and target columns

It is often the case that the data you obtain contains non-numeric features. This can be a problem, as most machine learning algorithms expect numeric data to perform computations with.

Run the code cell below to separate the student data into feature and target columns to see if any features are non-numeric.

```
In [9]: # Extract feature columns
        feature_cols = list(student_data.columns[:-1])

        # Extract target column 'passed'
        target_col = student_data.columns[-1]

        # Show the list of columns
        print "Feature columns:\n{}".format(feature_cols)
        print "\nTarget column: {}".format(target_col)
```

```
# Separate the data into feature data and target data (X_all and y_all, respectively)
X_all = student_data[feature_cols]
y_all = student_data[target_col]

# Show the feature information by printing the first five rows
print "\nFeature values:"
print X_all.head()
```

```
Feature columns:
['school', 'sex', 'age', 'address', 'famsize', 'Pstatus', 'Medu', 'Fedu', 'Mjob', 'Fjob', 'reason', 'gu

Target column: passed

Feature values:
  school sex  age address famsize Pstatus  Medu  Fedu     Mjob      Fjob  \
0     GP   F   18       U     GT3       A     4     4  at_home   teacher
1     GP   F   17       U     GT3       T     1     1  at_home     other
2     GP   F   15       U     LE3       T     1     1  at_home     other
3     GP   F   15       U     GT3       T     4     2   health  services
4     GP   F   16       U     GT3       T     3     3    other     other

    ...    higher internet  romantic  famrel  freetime goout Dalc Walc health  \
0   ...       yes       no        no       4         3     4    1    1      3
1   ...       yes      yes        no       5         3     3    1    1      3
2   ...       yes      yes        no       4         3     2    2    3      3
3   ...       yes      yes       yes       3         2     2    1    1      5
4   ...       yes       no        no       4         3     2    1    2      5

  absences
0        6
1        4
2       10
3        2
4        4

[5 rows x 30 columns]
```

1.4.2 Preprocess Feature Columns

As you can see, there are several non-numeric columns that need to be converted! Many of them are simply `yes`/`no`, e.g. `internet`. These can be reasonably converted into `1`/`0` (binary) values.

Other columns, like `Mjob` and `Fjob`, have more than two values, and are known as categorical variables. The recommended way to handle such a column is to create as many columns as possible values (e.g. `Fjob_teacher`, `Fjob_other`, `Fjob_services`, etc.), and assign a `1` to one of them and `0` to all others.

These generated columns are sometimes called dummy variables, and we will use the `pandas.get_dummies()` function to perform this transformation. Run the code cell below to perform the preprocessing routine discussed in this section.

```
In [13]: def preprocess_features(X):
             ''' Preprocesses the student data and converts non-numeric binary variables into
                 binary (0/1) variables. Converts categorical variables into dummy variables. '''

             # Initialize new output DataFrame
             output = pd.DataFrame(index = X.index)
```

```
            #print output
        # Investigate each feature column for the data
        for col, col_data in X.iteritems():

            # If data type is non-numeric, replace all yes/no values with 1/0
            if col_data.dtype == object:
                col_data = col_data.replace(['yes', 'no'], [1, 0])

            # If data type is categorical, convert to dummy variables
            if col_data.dtype == object:
                # Example: 'school' => 'school_GP' and 'school_MS'
                col_data = pd.get_dummies(col_data, prefix = col)

            # Collect the revised columns
            output = output.join(col_data)

        return output

    X_all = preprocess_features(X_all)
    print "Processed feature columns ({} total features):\n{} and the length of all new columns are
```

```
Processed feature columns (48 total features):
['school_GP', 'school_MS', 'sex_F', 'sex_M', 'age', 'address_R', 'address_U', 'famsize_GT3', 'famsize_LE3'
```

1.4.3 Implementation: Training and Testing Data Split

So far, we have converted all categorical features into numeric values. For the next step, we split the data (both features and corresponding labels) into training and test sets. In the following code cell below, you will need to implement the following: - Randomly shuffle and split the data (`X_all`, `y_all`) into training and testing subsets. - Use 300 training points (approximately 75%) and 95 testing points (approximately 25%). - Set a `random_state` for the function(s) you use, if provided. - Store the results in `X_train`, `X_test`, `y_train`, and `y_test`.

```
In [15]: from numpy import array
         from sklearn import cross_validation
         import numpy as np
         from __future__ import division
         # TODO: Import any additional functionality you may need here

         # TODO: Set the number of training points
         num_train = 300

         # Set the number of testing points
         num_test = X_all.shape[0] - num_train
         testPercentage= float(format(num_test/(num_train+num_test), '.2f'))

         # TODO: Shuffle and split the dataset into the number of training and testing points above
         X_train = []
         y_train=[]
         X_test=[]
         y_test=[]
         rs = cross_validation.ShuffleSplit(n_students, n_iter=1,test_size=testPercentage, random_state
         for train, test in rs:
             for val in train:
                 X_train.append(X_all.loc[val])
```

```
            y_train.append(y_all.loc[val])

        for val in test:
            X_test.append(X_all.loc[val])
            y_test.append(y_all.loc[val])
    # Return the training and testing data subsets
    X_train = array(X_train)
    y_train = array(y_train)
    X_test = array(X_test)
    y_test = array(y_test)

    # Show the results of the split
    print "Training set has {} samples.".format(X_train.shape[0])
    print "Testing set has {} samples.".format(X_test.shape[0])
```

```
Training set has 300 samples.
Testing set has 95 samples.
```

1.5 Training and Evaluating Models

In this section, you will choose 3 supervised learning models that are appropriate for this problem and available in `scikit-learn`. You will first discuss the reasoning behind choosing these three models by considering what you know about the data and each model's strengths and weaknesses. You will then fit the model to varying sizes of training data (100 data points, 200 data points, and 300 data points) and measure the F1 score. You will need to produce three tables (one for each model) that shows the training set size, training time, prediction time, F1 score on the training set, and F1 score on the testing set.

1.5.1 Question 2 - Model Application

List three supervised learning models that are appropriate for this problem. What are the general applications of each model? What are their strengths and weaknesses? Given what you know about the data, why did you choose these models to be applied?

Answer: The three supervised learning models that will be used in this project are Gausian Naive Bayes, support vector classifier, and decision tree classifier. The naive bayes classifiers work well in most real world situations without the need for alot of data. They work faster than other sophisticated methods, due to theoretical reasons which allieve the curse of dimensionality. Nevertheless, they are known to be bad estimators, so outputs from 'predict_proba' should be taken cautiously. I choose gausian naive bayes classifier to allieviate the limited data. More specifically we have 41 features to analyze, coordinated with limited training examples. Next the decision tree advantages are: easy to interpret visually, cost of using tree is logarithmic, can handle both categorical and numerical training data, can handle multi output models. The disadvantages to decision trees include easy overfitting, typically an NP-complete problem, and the trees could be biased if training examples have dominant classes. I choose decision trees for this model since it flows well after a quick analysis with the naive bayes classifier. Finally the SVC model will be applied with the data. The advantages of support vector machines are: great in high dimensional spaces, effetive when dimensions are greater than number of samples, versatile when kernel functions are applied. The disadvantages of support vector machines are: an expensive five-fold cross-validation must be used to provide probability estimates, and if the number of features are much greater than the number of samples - the performance is poor. I choose support vector classification since we have a high ratio of features to training samples while not having more features than training samples.

1.5.2 Setup

Run the code cell below to initialize three helper functions which you can use for training and testing the three supervised learning models you've chosen above. The functions are as follows: - `train_classifier` - takes as input a classifier and training data and fits the classifier to the data. - `predict_labels` - takes as input a fit classifier, features, and a target labeling and makes predictions using the F1 score. - `train_predict` - takes as input a classifier, and the training and testing data, and performs `train_clasifier` and `predict_labels`. - This function will report the F1 score for both the training and testing data separately.

```
In [29]: def train_classifier(clf, X_train, y_train):
             ''' Fits a classifier to the training data. '''

             # Start the clock, train the classifier, then stop the clock
             start = time()
             clf.fit(X_train, y_train)
             end = time()

             # Print the results
             print "Trained model in {:.4f} seconds".format(end - start)


         def predict_labels(clf, features, target):
             ''' Makes predictions using a fit classifier based on F1 score. '''

             # Start the clock, make predictions, then stop the clock
             start = time()
             y_pred = clf.predict(features)
             end = time()

             # Print and return results
             print "Made predictions in {:.4f} seconds.".format(end - start)
             return f1_score(target, y_pred, pos_label='yes')


         def train_predict(clf, X_train, y_train, X_test, y_test):
             ''' Train and predict using a classifer based on F1 score. '''

             # Indicate the classifier and the training set size
             print "Training a {} using a training set size of {}. . .".format(clf.__class__.__name__,

             # Train the classifier
             train_classifier(clf, X_train, y_train)

             # Print the results of prediction for both training and testing
             print "F1 score for training set: {:.4f}.".format(predict_labels(clf, X_train, y_train))
             print "F1 score for test set: {:.4f}.".format(predict_labels(clf, X_test, y_test))
```

1.5.3 Implementation: Model Performance Metrics

With the predefined functions above, you will now import the three supervised learning models of your choice and run the `train_predict` function for each one. Remember that you will need to train and predict on each classifier for three different training set sizes: 100, 200, and 300. Hence, you should expect to have 9 different outputs below — 3 for each model using the varying training set sizes. In the following code cell, you will need to implement the following: - Import the three supervised learning models you've discussed in the previous section. - Initialize the three models and store them in `clf_A`, `clf_B`, and `clf_C`. - Use a `random_state` for each model you use, if provided. - **Note:** Use the default settings for each model — you will tune one specific model in a later section. - Create the different training set sizes to be used to train each model. - Do not reshuffle and resplit the data! The new training points should be drawn from `X_train` and `y_train`. - Fit each model with each training set size and make predictions on the test set (9 in total). **Note:** Three tables are provided after the following code cell which can be used to store your results.

```
In [36]: from sklearn.svm import SVC
         from sklearn import tree
         from sklearn.naive_bayes import GaussianNB

         # TODO: Initialize the three models
         clf_A = GaussianNB()
         clf_B = tree.DecisionTreeClassifier()
         clf_C = SVC()

         def dataSplitter(trainPoints, X_all, y_all):
             # TODO: Set the number of training points
             num_train = trainPoints

             # Set the number of testing points
             num_test = X_all.shape[0] - num_train
             testPercentage= float(format(num_test/(num_train+num_test), '.2f'))

             # TODO: Shuffle and split the dataset into the number of training and testing points above
             X_train = []
             y_train=[]
             X_test=[]
             y_test=[]
             rs = cross_validation.ShuffleSplit(n_students, n_iter=1,test_size=testPercentage, random_s
             for train, test in rs:
                 for val in train:
                     X_train.append(X_all.loc[val])
                     y_train.append(y_all.loc[val])

                 for val in test:
                     X_test.append(X_all.loc[val])
                     y_test.append(y_all.loc[val])
             # Return the training and testing data subsets
             X_train = array(X_train)
             y_train = array(y_train)
             X_test = array(X_test)
             y_test = array(y_test)
             #print len(X_train), len(X_test), len(y_train), len(y_test)
             return X_train, X_test, y_train, y_test

         # TODO: Set up the training set sizes
         X_train_100,X_test_100,y_train_100,y_test_100 = dataSplitter(100,X_all,y_all)
         #col_data = col_data.replace(['yes', 'no'], [1, 0])

         X_train_200,X_test_200,y_train_200,y_test_200 = dataSplitter(200,X_all,y_all)


         X_train_300,X_test_300,y_train_300,y_test_300 = dataSplitter(300,X_all,y_all)

         train_classifier(clf_A , X_train_100, y_train_100)
         predict_labels(clf_A , X_test_100, y_test_100)
         train_predict(clf_A , X_train_100, y_train_100, X_test_100, y_test_100)
         print '\n'

         train_classifier(clf_A , X_train_200, y_train_200)
```

```
predict_labels(clf_A , X_test_200, y_test_200)
train_predict(clf_A , X_train_200, y_train_200, X_test_200, y_test_200)
print '\n'

train_classifier(clf_A , X_train_300, y_train_300)
predict_labels(clf_A , X_test_300, y_test_300)
train_predict(clf_A , X_train_300, y_train_300, X_test_300, y_test_300)

print '\n\n'
train_classifier(clf_B , X_train_100, y_train_100)
predict_labels(clf_B , X_test_100, y_test_100)
train_predict(clf_B , X_train_100, y_train_100, X_test_100, y_test_100)
print '\n'

train_classifier(clf_B , X_train_200, y_train_200)
predict_labels(clf_B , X_test_200, y_test_200)
train_predict(clf_B , X_train_200, y_train_200, X_test_200, y_test_200)
print '\n'

train_classifier(clf_B , X_train_300, y_train_300)
predict_labels(clf_B , X_test_300, y_test_300)
train_predict(clf_B , X_train_300, y_train_300, X_test_300, y_test_300)

print '\n\n'
train_classifier(clf_C , X_train_100, y_train_100)
predict_labels(clf_C , X_test_100, y_test_100)
train_predict(clf_C , X_train_100, y_train_100, X_test_100, y_test_100)
print '\n'

train_classifier(clf_C , X_train_200, y_train_200)
predict_labels(clf_C , X_test_200, y_test_200)
train_predict(clf_C , X_train_200, y_train_200, X_test_200, y_test_200)
print '\n'

train_classifier(clf_C , X_train_300, y_train_300)
predict_labels(clf_C , X_test_300, y_test_300)
train_predict(clf_C , X_train_300, y_train_300, X_test_300, y_test_300)
```

```
Trained model in 0.0016 seconds
Made predictions in 0.0012 seconds.
Training a GaussianNB using a training set size of 98. . .
Trained model in 0.0014 seconds
Made predictions in 0.0005 seconds.
F1 score for training set: 0.3059.
Made predictions in 0.0008 seconds.
F1 score for test set: 0.3664.


Trained model in 0.0015 seconds
Made predictions in 0.0007 seconds.
Training a GaussianNB using a training set size of 201. . .
Trained model in 0.0015 seconds
Made predictions in 0.0009 seconds.
F1 score for training set: 0.7956.
```

```
Made predictions in 0.0007 seconds.
F1 score for test set: 0.7331.


Trained model in 0.0017 seconds
Made predictions in 0.0005 seconds.
Training a GaussianNB using a training set size of 300. . .
Trained model in 0.0017 seconds
Made predictions in 0.0008 seconds.
F1 score for training set: 0.8088.
Made predictions in 0.0005 seconds.
F1 score for test set: 0.7500.



Trained model in 0.0014 seconds
Made predictions in 0.0002 seconds.
Training a DecisionTreeClassifier using a training set size of 98. . .
Trained model in 0.0014 seconds
Made predictions in 0.0002 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0002 seconds.
F1 score for test set: 0.6720.


Trained model in 0.0032 seconds
Made predictions in 0.0003 seconds.
Training a DecisionTreeClassifier using a training set size of 201. . .
Trained model in 0.0031 seconds
Made predictions in 0.0002 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0002 seconds.
F1 score for test set: 0.7634.


Trained model in 0.0045 seconds
Made predictions in 0.0002 seconds.
Training a DecisionTreeClassifier using a training set size of 300. . .
Trained model in 0.0045 seconds
Made predictions in 0.0003 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0002 seconds.
F1 score for test set: 0.7213.



Trained model in 0.0035 seconds
Made predictions in 0.0049 seconds.
Training a SVC using a training set size of 98. . .
Trained model in 0.0027 seconds
Made predictions in 0.0017 seconds.
F1 score for training set: 0.8727.
Made predictions in 0.0050 seconds.
F1 score for test set: 0.7801.
```

```
Trained model in 0.0092 seconds
Made predictions in 0.0066 seconds.
Training a SVC using a training set size of 201. . .
Trained model in 0.0091 seconds
Made predictions in 0.0066 seconds.
F1 score for training set: 0.8746.
Made predictions in 0.0066 seconds.
F1 score for test set: 0.7742.


Trained model in 0.0193 seconds
Made predictions in 0.0047 seconds.
Training a SVC using a training set size of 300. . .
Trained model in 0.0192 seconds
Made predictions in 0.0140 seconds.
F1 score for training set: 0.8692.
Made predictions in 0.0049 seconds.
F1 score for test set: 0.7586.
```

1.5.4 Tabular Results

Edit the cell below to see how a table can be designed in Markdown. You can record your results from above in the tables provided.

** Classifer 1 - Gaussian Naive Bayes**

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	.0016	.0011	.3059	.3664
200	.0015	.0007	.7956	.7331
300	.0017	.0005	.8088	.7500

** Classifer 2 - Decision Tree Classifier?**

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	.0014	.0003	1.000	.6613
200	.0031	.0002	1.000	.7373
300	.0046	.0002	1.000	.7241

** Classifer 3 - SVC Support Vector Classification**

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	.0035	.0049	.8727	.7801
200	.0092	.0066	.8746	.7742
300	.0193	.0047	.8692	.7586

1.6 Choosing the Best Model

In this final section, you will choose from the three supervised learning models the best model to use on the student data. You will then perform a grid search optimization for the model over the entire training set (`X_train` and `y_train`) by tuning at least one parameter to improve upon the untuned model's F1 score.

1.6.1 Question 3 - Chosing the Best Model

Based on the experiments you performed earlier, in one to two paragraphs, explain to the board of supervisors what single model you chose as the best model. Which model is generally the most appropriate based on the available data, limited resources, cost, and performance?

Answer: After testing all 3 models, one should choose the support vector classifier as the best model. The testing f1 scores were the highest for SVC compared to the other two models. When training data is limited, support vectors classifiers also performed better in comparison to the other two models. Although the most expensive model was support vector classification. If one was to choose a model based on limited data, resources, cost and peroformance, the decision tree classifier would be the best.

1.6.2 Question 4 - Model in Layman's Terms

In one to two paragraphs, explain to the board of directors in layman's terms how the final model chosen is supposed to work. For example if you've chosen to use a decision tree or a support vector machine, how does the model go about making a prediction?

Answer: The final chosen model was the support vector classifier. It selects the best line that separates the two classes. The best line not only separates the two classes, but also has the highest margins, being as far as possible from the classes but still in between. This is easier to visualize with two dimensions, but is applicable to a feature set with more.

1.6.3 Implementation: Model Tuning

Fine tune the chosen model. Use grid search (`GridSearchCV`) with at least one important parameter tuned with at least 3 different values. You will need to use the entire training set for this. In the code cell below, you will need to implement the following: - Import `sklearn.grid_search.gridSearchCV` and `sklearn.metrics.make_scorer`. - Create a dictionary of parameters you wish to tune for the chosen model. - Example: `parameters = {'parameter' : [list of values]}`. - Initialize the classifier you've chosen and store it in `clf`. - Create the F1 scoring function using `make_scorer` and store it in `f1_scorer`. - Set the `pos_label` parameter to the correct value! - Perform grid search on the classifier `clf` using `f1_scorer` as the scoring method, and store it in `grid_obj`. - Fit the grid search object to the training data (`X_train`, `y_train`), and store it in `grid_obj`.

```
In [54]: # TODO: Import 'GridSearchCV' and 'make_scorer'
         from sklearn import svm, grid_search, datasets
         from sklearn.metrics import f1_score, make_scorer

         # TODO: Create the parameters list you wish to tune
         parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}

         # TODO: Initialize the classifier
         svr = svm.SVC()

         # TODO: Make an f1 scoring function using 'make_scorer'
         f1_scorer = make_scorer(f1_score,pos_label='yes')

         # TODO: Perform grid search on the classifier using the f1_scorer as the scoring method
         grid_obj = grid_search.GridSearchCV(svr,parameters, f1_scorer)

         # TODO: Fit the grid search object to the training data and find the optimal parameters
```

```
        grid_obj = grid_obj.fit(X_all,y_all)

        # Get the estimator
        clf = grid_obj.best_estimator_
        print grid_obj.best_estimator_
        # Report the final F1 score for training and testing after parameter tuning
        print "\nTuned model has a training F1 score of {:.4f}.".format(predict_labels(clf, X_train, y
        print "Tuned model has a testing F1 score of {:.4f}.".format(predict_labels(clf, X_test, y_tes
```

```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
Made predictions in 0.0177 seconds.

Tuned model has a training F1 score of 0.8723.
Made predictions in 0.0060 seconds.
Tuned model has a testing F1 score of 0.8169.
```

1.6.4 Question 5 - Final F1 Score

What is the final model's F1 score for training and testing? How does that score compare to the untuned model?

Answer: The tuned model has a training F1 score of 0.8723 and made predictions in 0.0060 seconds. The tuned model has a testing F1 score of 0.8169. The untuned scores were lower because not all parameters have been tested. More specifically the gridsearch algorithm did an exhaustive search looking for lower error rates when training and testing. This kept the initial untuned f1 training score around the same at .87 but increased f1 testing score from around .77 to .8169.

> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to
> **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.