Machine Learning Engineer Nanodegree

Supervised Learning

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.

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: I would say that this is a typical classification problem. It's because the labels are discrete and non-continous. It only has two outcomes passing or failing.

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 [54]:
```
# Import libraries
import numpy as np
import pandas as pd
from time import time
from sklearn.metrics import f1_score

# Read student data

rs = 40
student_data = pd.read_csv("student-data.csv")
print ("Student data read successfully!")
```

```
Student data read successfully!
```

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 [55]:
```
n_students = student_data.shape[0]
n_features = student_data.shape[1] - 1
n_passed = np.sum(student_data['passed'] == 'yes')
n_failed = np.sum(student_data['passed'] == 'no')
grad_rate = float(n_passed) / n_students
# 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: 0.67%
```

Preparing the Data

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

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 [56]:
```
# 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', 'guardian', 'traveltime', 'stud
ytime', 'failures', 'schoolsup', 'famsup', 'paid', 'activities',
'nursery', 'higher', 'internet', 'romantic', 'famrel', 'freetim
e', 'goout', 'Dalc', 'Walc', 'health', 'absences']

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  s
ervices
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]
```

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()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html?highlight=get_dummies#pandas.get_dummies) function to perform this transformation. Run the code cell below to perform the preprocessing routine discussed in this section.

In [57]:
```
def preprocess_features(X):
    ''' Preprocesses the student data and converts non-numeric bi
nary variables into
        binary (0/1) variables. Converts categorical variables in
to dummy variables. '''

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

    # 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{}".forma
t(len(X_all.columns), list(X_all.columns)))
```

```
Processed feature columns (48 total features):
['school_GP', 'school_MS', 'sex_F', 'sex_M', 'age', 'address_R',
'address_U', 'famsize_GT3', 'famsize_LE3', 'Pstatus_A', 'Pstatus_
T', 'Medu', 'Fedu', 'Mjob_at_home', 'Mjob_health', 'Mjob_other',
'Mjob_services', 'Mjob_teacher', 'Fjob_at_home', 'Fjob_health',
'Fjob_other', 'Fjob_services', 'Fjob_teacher', 'reason_course',
'reason_home', 'reason_other', 'reason_reputation', 'guardian_fat
her', 'guardian_mother', 'guardian_other', 'traveltime', 'studyti
me', 'failures', 'schoolsup', 'famsup', 'paid', 'activities', 'nu
rsery', 'higher', 'internet', 'romantic', 'famrel', 'freetime',
'goout', 'Dalc', 'Walc', 'health', 'absences']
```

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 [58]:
```
# TODO: Import any additional functionality you may need here

from sklearn.utils import shuffle
from sklearn.cross_validation import train_test_split

# TODO: Set the number of training points


num_train = 300  # about 75% of the data


# Set the number of testing points
num_test = X_all.shape[0] - num_train

# TODO: Shuffle and split the dataset into the number of training
and testing points above
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all,
train_size=num_train,
                                                    random_state=
rs)


# 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.
```

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 F_1 score. You will need to produce three tables (one for each model) that shows the training set size, training time, prediction time, F_1 score on the training set, and F_1 score on the testing set.

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 models I choose are :- Decision tree, support vector machines (svm) and adaboost.

Decision tree--

General applications, strengths and weakness: Decision Trees are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

Advantages: Simple to understand and to interpret. Trees can be visualised. Requires little data preparation. The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree. Able to handle both numerical and categorical data. Other techniques are usually specialised in analysing datasets that have only one type of variable. Able to handle multi-output problems. Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret. Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model. Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

Disadvantages: Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem. Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble. The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement. There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems. Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

Why choose this model: Decision Trees is a good classification tool. It is capable of performing multi-class classification on a dataset. Because we have relative large features in our dataset, Decision Trees would be a potential cadidate.

SUPPORT VECTOR MACHINE (SVM)

General applications, strengths and weakness: SVMs are a set of learning methods used for classification, regression and outlier detection.

Advantages: Effective in high dimensional spaces. Still effective in cases where number of dimensions is greater than the number of samples. Uses a subset of training points in the decision function (support vectors), so it's also memory efficient. Versatile: different Kernel functions can be specified for the decision functions.

Disadvantages: If the number of features is much greater than the number of samples, the method is likely to give poor performances. SVMs do not directly provide probability estimates.

Why choose this model: This is two-category classification problem. Comparing to the training size, the feature size is reasonable. We can take advantage of the kernel function in SVMs to separate two kinds of different students.

ADABOOST

General applications, strengths and weakness: Boosting is an algorithm that uses the weighted average of many weak classifiers (in the case of AdaBoost, its linear hyperplanes) to predict the output class.

Advantages: One advantage of this algorithm is that its weighting is updated during each iteration to incentivize the correctly classify points in the next iteration that it got wrong in the previous iteration. In theory this helps the algorithm find a good fit to the data. Also, since it is an ensemble method (averaging outputs of other models) it has the property that is should generalize well.

Disadvantages: The real drawback for this approach is the time it takes to train. Essentially, a linear hyperplane decision boundary gets fit to the data during each iteration, and then those hyperplanes are weighted averaged together. That’s a lot of computation that the school might not want due to their concerns over cost and processing time.

Why choose this model: an ensemble of Decision Trees will always outperform a single decision tree. Given that the data set size is quite small (300 data points), the Decision Tree might tend to overfit so a boosted ensemble of Decision Trees might be worth trying.

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 F_1 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 F_1 score for both the training and testing data separately.

In [59]:
```
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 scor
e. '''

    # 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 - sta
rt))
    return f1_score(target.values, 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 {}. . .".f
ormat(clf.__class__.__name__, len(X_train)))

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

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

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 [60]:
```
# TODO: Import the three supervised learning models from sklearn
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier

# TODO: Initialize the three models
clf_A = DecisionTreeClassifier(random_state=rs)
clf_B = SVC(random_state=rs)
clf_C = GradientBoostingClassifier(random_state=rs)

# TODO: Set up the training set sizes
X_train_100 = X_train[:100]
y_train_100 = y_train[:100]

X_train_200 = X_train[:200]
y_train_200 = y_train[:200]

X_train_300 = X_train
y_train_300 = y_train

# TODO: Execute the 'train_predict' function for each classifier
and each training set size
for clf in [clf_A, clf_B, clf_C]:
    for i in range(100, 400, 100):
        train_predict(clf, X_train[:i], y_train[:i], X_test, y_te
st)
    print ('\n')
```

```
Training a DecisionTreeClassifier using a training set size of 10
0. . .
Trained model in 0.0022 seconds
Made predictions in 0.0005 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0003 seconds.
F1 score for test set: 0.6281.
Training a DecisionTreeClassifier using a training set size of 20
0. . .
Trained model in 0.0020 seconds
Made predictions in 0.0005 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0007 seconds.
F1 score for test set: 0.7538.
Training a DecisionTreeClassifier using a training set size of 30
0. . .
Trained model in 0.0034 seconds
Made predictions in 0.0016 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0005 seconds.
F1 score for test set: 0.7692.


Training a SVC using a training set size of 100. . .
Trained model in 0.0016 seconds
Made predictions in 0.0010 seconds.
F1 score for training set: 0.8750.
Made predictions in 0.0013 seconds.
F1 score for test set: 0.8199.
Training a SVC using a training set size of 200. . .
Trained model in 0.0058 seconds
Made predictions in 0.0031 seconds.
F1 score for training set: 0.8664.
Made predictions in 0.0015 seconds.
F1 score for test set: 0.7895.
Training a SVC using a training set size of 300. . .
Trained model in 0.0095 seconds
Made predictions in 0.0063 seconds.
F1 score for training set: 0.8786.
Made predictions in 0.0022 seconds.
F1 score for test set: 0.8366.


Training a GradientBoostingClassifier using a training set size o
f 100. . .
Trained model in 0.0602 seconds
Made predictions in 0.0006 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0005 seconds.
F1 score for test set: 0.7917.
Training a GradientBoostingClassifier using a training set size o
f 200. . .
Trained model in 0.0899 seconds
Made predictions in 0.0021 seconds.
F1 score for training set: 0.9888.
Made predictions in 0.0009 seconds.
F1 score for test set: 0.8227.
```

```
Training a GradientBoostingClassifier using a training set size o
f 300. . .
Trained model in 0.1253 seconds
Made predictions in 0.0011 seconds.
F1 score for training set: 0.9706.
Made predictions in 0.0005 seconds.
F1 score for test set: 0.8000.
```

Tabular Results

Edit the cell below to see how a table can be designed in Markdown (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet#tables). You can record your results from above in the tables provided.

Classifer 1 - Decision Tree

Training Set Size	Prediction Time (train)	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.0005	0.0003	1.0000	0.6281
200	0.0005	0.0007	1.0000	0.7538
300	0.0016	0.0005	1.0000	0.7692

Classifer 2 - SVM

Training Set Size	Prediction Time (train)	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.0010	0.0013	0.8750	0.8199
200	0.0031	0.0015	0.8664	0.7895
300	0.0063	0.0022	0.8786	0.8366

Classifer 3 - Gradient Boosting Classifer

Training Set Size	Prediction Time (train)	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.0006	0.0005	1.0000	0.7917
200	0.0021	0.0009	0.9888	0.8227
300	0.0011	0.0005	0.9706	0.8000

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 F_1 score.

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:

Based on the experiments performed earlier, I'd say the best model is the Gradient Boosted Trees Classifier (GBT). In terms of training time, the GBT is at most 70 times slower than the SVM and the DT models, with figures at 76ms for GBT, 1.1ms for both SVMs and DTs for the 300 data set training time.

I still stick with GBT, because even though much slower that SVMs and DTs, the actual training times are very small and negligible, with at most 76 ms for GBT for the 300 data set.

Also, for prediction times, GBT seem to have much smaller times than SVMs and DTs. The figures stand at around 1ms and 4ms for the training and test prediction times for GBT respectively, while for SVMs it's slightly more time - 4.8ms for training and 1.4ms for testing. And finally for DT it's still abit slower for training (2ms) but faster for testing (2ms). These are all figures for the size 300 dataset.

I'd argue for GBT (in tie with DT) in this case because the I believe testing might be more important than training, as testing might be done on an on-going basis while training might only be done once in a while, say at the beginning/end of an academic year.

So for computational times, GBTs win in tie with DT, and might be significantly faster than SVMs for prediction.

Now, for accuracy, GBTs are much better than DT (0.8000 for GBTs and 0.7692 for DTs for size 300 dataset), but slightly worse than SVMs (0.8366 for SVMs for size 300 dataset).

So GBTs are preferred to DTs in terms of accuracy. This is also because DTs tend to overfit, while GBTs ease the overfit problem of DTs by taking an ensemble of many different weak DTs.

So, a fair comparison should be in favor of GBT, because they have the fastest prediction times with moderately good accuracies.

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:

Gradient Boosting works by fitting a classifier a number of times (or viewing the student's data a number of times, each time using a weak classifier, usually a weak Decision tree) to classify the data and identify which student should pass and which should fail.

At first, GBT will initialise the model by fitting a weak classifier (a classifier which fails to accurately predict which student fails or passes) to the dataset, thereby generating prediction errors (quantified mistake of the weak classifier)

Then, at each iteration, the following happens in GBT: The errors from the initial model are fitted again using another DT A simple multiplier is obtain by optimisation methods; This multiplier is multiplied to the error-fitted model, and combined with the original model to produce a stronger model.

This process is repeated for a certain number of times to produce even stronger models. Finally, the strong model is outputted from the loop above, and this strong model is then used to test new points.

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` (http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html) and `sklearn.metrics.make_scorer` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html).
- 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 F_1 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 [53]:
```
# TODO: Import 'gridSearchCV' and 'make_scorer'
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import make_scorer

# TODO: Create the parameters list you wish to tune
parameters = {'max_depth': [1,2,3],
              'learning_rate': [0.05, 0.1, 0.15],
              'n_estimators': [50, 75, 100],
              'subsample': [1.0, 0.9]
             }

# TODO: Initialize the classifier
clf = GradientBoostingClassifier(random_state=rs)

# 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 = GridSearchCV(clf, parameters, scoring=f1_scorer)

# TODO: Fit the grid search object to the training data and find the optimal parameters
grid_obj.fit(X_train, y_train)

# Get the estimator
clf = grid_obj.best_estimator_

print (clf.get_params(), '\n')

# Report the final F1 score for training and testing after parameter tuning
print ("Tuned model has a training F1 score of {:.4f}.".format(predict_labels(clf, X_train, y_train)))
print ("Tuned model has a testing F1 score of {:.4f}.".format(predict_labels(clf, X_test, y_test)))
```

```
{'n_estimators': 100, 'presort': 'auto', 'min_samples_leaf': 1,
'min_weight_fraction_leaf': 0.0, 'verbose': 0, 'min_samples_spli
t': 2, 'learning_rate': 0.05, 'max_features': None, 'max_leaf_nod
es': None, 'init': None, 'loss': 'deviance', 'random_state': 40,
'max_depth': 1, 'subsample': 1.0, 'warm_start': False}

Made predictions in 0.0009 seconds.
Tuned model has a training F1 score of 0.8276.
Made predictions in 0.0004 seconds.
Tuned model has a testing F1 score of 0.8477.
```

Question 5 - Final F_1 Score

What is the final model's F_1 score for training and testing? How does that score compare to the untuned model?

Answer:

The final model's F1 score for training was 0.8276. This was slightly higher than that of the default model, 0.8227. The final model's F1 score for testing was 0.8477. This was substantially higher than that of the default model, 0.8000. As a side note, when tuning the algorithm, I noticed that the model might be struggling with overfit. As seen from the parameters chosen, the model prefers very shallow trees, and when I tried tuning the max_features parameter, it chose log2 features!

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.