student_intervention

June 28, 2017

1 Machine Learning Engineer Nanodegree

1.1 Supervised Learning

1.2 Project: 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: This is a regression problem as we want to classify the students into 2 classes: those who need early intervention and those who don't.

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 [2]: # 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 [16]: # TODO: Calculate number of students
         n_students = student_data.shape[0]

         # TODO: Calculate number of features
         n_features = student_data.shape[1]

         # TODO: Calculate passing students
         n_passed = (student_data.loc[student_data['passed']=='yes']).shape[0]

         # TODO: Calculate failing students
         n_failed = (student_data.loc[student_data['passed']=='no']).shape[0]

         # TODO: Calculate graduation rate
         grad_rate = n_passed/float(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: 31
Number of students who passed: 265
Number of students who failed: 130
Graduation rate of the class: 0.67%
```

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 [17]: # 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', 'reas

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 [18]: 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)

             # 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{}".format(len(X_all.columns),
```

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

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 [19]: # TODO: Import any additional functionality you may need here
         from sklearn.cross_validation import train_test_split

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

         # 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
         X_train, X_test, y_train, y_test= train_test_split(X_all, y_all, train_size=num_train

         # 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.
```

```
c:\python27\lib\site-packages\sklearn\cross_validation.py:44: DeprecationWarning: This module
  "This module will be removed in 0.20.", DeprecationWarning)
```

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.

The following supervised learning models are currently available in `scikit-learn` that you may choose from: - Gaussian Naive Bayes (GaussianNB) - Decision Trees - Ensemble Methods (Bagging, AdaBoost, Random Forest, Gradient Boosting) - K-Nearest Neighbors (KNeighbors) - Stochastic Gradient Descent (SGDC) - Support Vector Machines (SVM) - Logistic Regression

1.5.1 Question 2 - Model Application

List three supervised learning models that are appropriate for this problem. For each model chosen - Describe one real-world application in industry where the model can be applied. *(You may need to do a small bit of research for this — give references!)* - What are the strengths of the model; when does it perform well? - What are the weaknesses of the model; when does it perform poorly? - What makes this model a good candidate for the problem, given what you know about the data?

Answer: 1. kNN

Strength: * fast training * effective for larger training data * can work with noizy data

Weaknesses: * need to tune number of neighbors

Use in real life: Recommender systems, Gene expression

Why we use it: is is generally fast and according to the task time is the crucial factor for us

2. Decision Tree

Strength: * simple to read and interpret * requires little data preparation * operates in white box

Weaknesses: * overfitting is often a problem * can be unstable * can be biased when some class dominates

Uses: decision analysis

Why I use it: it is easy to apply and I expect it to perform well as there is quite clear structure in the data

3. SVM

Strengths: * unlikely to overfit

Weaknesses: * need to select kernel * choosing parameters is painful

Uses: text categorization, biology

Why I use it: with small training set, I expect other models to have some level of overfitting. SVMs don't have a tendency to overfit that much.

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 [20]: 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.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 {}. . .".format(clf.__class__._

    # 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_
    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 [24]: # TODO: Import the three supervised learning models from sklearn
         from sklearn import tree
         from sklearn import svm
         from sklearn import neighbors
```

```
# TODO: Initialize the three models
clf_A = tree.DecisionTreeClassifier()
clf_B = svm.SVC()
clf_C = neighbors.KNeighborsClassifier()

# 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[:300]
y_train_300 = y_train[:300]

# TODO: Execute the 'train_predict' function for each classifier and each training se
train_predict(clf_A, X_train_100, y_train_100, X_test, y_test)
train_predict(clf_A, X_train_200, y_train_200, X_test, y_test)
train_predict(clf_A, X_train_300, y_train_300, X_test, y_test)
train_predict(clf_B, X_train_100, y_train_100, X_test, y_test)
train_predict(clf_B, X_train_200, y_train_200, X_test, y_test)
train_predict(clf_B, X_train_300, y_train_300, X_test, y_test)
train_predict(clf_C, X_train_100, y_train_100, X_test, y_test)
train_predict(clf_C, X_train_200, y_train_200, X_test, y_test)
train_predict(clf_C, X_train_300, y_train_300, X_test, y_test)
```

```
Training a DecisionTreeClassifier using a training set size of 100. . .
Trained model in 0.0110 seconds
Made predictions in 0.0010 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0000 seconds.
F1 score for test set: 0.6325.
Training a DecisionTreeClassifier using a training set size of 200. . .
Trained model in 0.0030 seconds
Made predictions in 0.0010 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0010 seconds.
F1 score for test set: 0.7424.
Training a DecisionTreeClassifier using a training set size of 300. . .
Trained model in 0.0040 seconds
Made predictions in 0.0010 seconds.
F1 score for training set: 1.0000.
Made predictions in 0.0010 seconds.
F1 score for test set: 0.6333.
Training a SVC using a training set size of 100. . .
Trained model in 0.0050 seconds
Made predictions in 0.0020 seconds.
F1 score for training set: 0.8777.
```

```
Made predictions in 0.0010 seconds.
F1 score for test set: 0.7746.
Training a SVC using a training set size of 200. . .
Trained model in 0.0050 seconds
Made predictions in 0.0040 seconds.
F1 score for training set: 0.8679.
Made predictions in 0.0020 seconds.
F1 score for test set: 0.7815.
Training a SVC using a training set size of 300. . .
Trained model in 0.0100 seconds
Made predictions in 0.0070 seconds.
F1 score for training set: 0.8761.
Made predictions in 0.0000 seconds.
F1 score for test set: 0.7838.
Training a KNeighborsClassifier using a training set size of 100. . .
Trained model in 0.0000 seconds
Made predictions in 0.0000 seconds.
F1 score for training set: 0.8060.
Made predictions in 0.0030 seconds.
F1 score for test set: 0.7246.
Training a KNeighborsClassifier using a training set size of 200. . .
Trained model in 0.0020 seconds
Made predictions in 0.0040 seconds.
F1 score for training set: 0.8800.
Made predictions in 0.0030 seconds.
F1 score for test set: 0.7692.
Training a KNeighborsClassifier using a training set size of 300. . .
Trained model in 0.0010 seconds
Made predictions in 0.0070 seconds.
F1 score for training set: 0.8809.
Made predictions in 0.0040 seconds.
F1 score for test set: 0.7801.
```

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 - Decision Tree**

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.0110	0.0000	1	0.63
200	0.0030	0.0010	1	0.7424
300	0.0040	0.0010	1	0.6333

** Classifer 2 - SVM **

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0.0050	0.0010	0.8777	0.7746
200	0.0050	0.0020	0.8679	0.7815
300	0.0100	0.000	0.8761	0.7838

** Classifer 3 - kNN**

Training Set Size	Training Time	Prediction Time (test)	F1 Score (train)	F1 Score (test)
100	0	0.0030	0.8060	0.7246
200	0.0020	0.0030	0.8800	0.7692
300	0.0010	0.0040	0.8809	0.7801

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 - Choosing 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: As it can be seen from obtained results, Decision Tree model overfited a lot resulting in the poorest result, so we are not even considering it. The choice is then between Support Vector Machines and k Nearest Neighbors. Although on the train data SVM made less accurate predictions than kNN, on the test data SVM performed better. At the same time, KNN took less time to make predictions. That is why I advice to use KNN - for larger data set (and assuming we will deal with larger sets in future) the accuracy is almost the same ad in the SVM. Also we can see that accuracy tends to increase as the dataset increases, so we can expect it to increase even more as we add more students. On the same time, SVM's accuracy has smaller dependency on the dataset size, so it is unlikely that it will increase. KNN algorithm will save the CPU time and therefore money, as well as resulting in adequate accuracy.

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. Be sure that you are describing the major qualities of the model, such as how the model is trained and how the model makes a prediction. Avoid using advanced mathematical or technical jargon, such as describing equations or discussing the algorithm implementation.

Answer: In simple words, the algorithm is looking at several closest students from the one who we want to classify as "needing intervention" or "not needing intervention" and makes prediction equal to the majority of the classes of the neighbors. So, if e.g. out of 3 students that are the most similar to the one we are trying to classify, 2 graduated successfully, then our student is also likely to graduate.

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 [34]: # 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 = {'n_neighbors' : [1,2,3,4,5],
                       'p': [1,2]}

         # TODO: Initialize the classifier
         clf = neighbors.KNeighborsClassifier()

         # 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 meth
         grid_obj = GridSearchCV(clf, parameters, scoring=f1_scorer, cv=5, verbose=0)

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

         # Get the estimator
         clf = grid_obj.best_estimator_

         # 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_t
         print "Tuned model has a testing F1 score of {:.4f}.".format(predict_labels(clf, X_te
```

```
Made predictions in 0.0160 seconds.
Tuned model has a training F1 score of 0.8578.
Made predictions in 0.0000 seconds.
Tuned model has a testing F1 score of 0.7887.
```

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: I tried tuning the number of neighbors and the distance metric. The distance metric did not actually affect the result, while modifying the neighbors amount improved the accuracy slightly. The train score decreased, but the fact that the difference between train score and test score is now smaller means that the model can generalize better.

> **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.