student_intervention

April 12, 2016

1 Project 2: Supervised Learning

Building a Student Intervention System

1.1 Classification vs Regression

Your goal is to identify students who might need early intervention - which type of supervised machine learning problem is this, classification or regression? Why?

Answer: I think this type of problem is classification, because the target data is discrete, students who passed and students who failed.

1.2 Exploring the Data

Let's go ahead and read in the student dataset first.

```
In [1]: # Import libraries
        import numpy as np
        import pandas as pd
```

```
In [2]: # Read student data
        student_data = pd.read_csv("student-data.csv")
        print "Student data read successfully!"
        # Note: The last column 'passed' is the target/label, all other are feature columns
```

```
Student data read successfully!
```

Now, can you find out the following facts about the dataset? - Total number of students - Number of students who passed - Number of students who failed - Graduation rate of the class (%) - Number of features

```
In [3]: n_students = student_data.shape[0]
        n_features = student_data.shape[1]
        n_passed = student_data[student_data['passed']=='yes'].shape[0]
        n_failed = n_students - n_passed
        grad_rate = (n_passed * 100.0) / n_students
        print "Total number of students: {}".format(n_students)
        print "Number of students who passed: {}".format(n_passed)
        print "Number of students who failed: {}".format(n_failed)
        print "Number of features: {}".format(n_features-1)
        print "Graduation rate of the class: {:.2f}%".format(grad_rate)
```

```
Total number of students: 395
Number of students who passed: 265
Number of students who failed: 130
Number of features: 30
Graduation rate of the class: 67.09%
```

1.3 Preparing the Data

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

1.3.1 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.

```
In [4]: # Extract feature (X) and target (y) columns
        feature_cols = list(student_data.columns[:-1])  # all columns but last are features
        target_col = student_data.columns[-1]  # last column is the target/label
        print "Feature column(s):-\n{}".format(feature_cols)
        print "Target column: {}".format(target_col)

        X_all = student_data[feature_cols]  # feature values for all students
        y_all = student_data[target_col]  # corresponding targets/labels
        print "\nFeature values:-"
        print X_all.head()  # print the first 5 rows
```

```
Feature column(s):-
['school', 'sex', 'age', 'address', 'famsize', 'Pstatus', 'Medu', 'Fedu', 'Mjob', 'Fjob', 'reason', 'gua
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]
```

1.3.2 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.

```
In [5]: # Preprocess feature columns
        def preprocess_features(X):
            outX = pd.DataFrame(index=X.index)  # output dataframe, initially empty

            # Check each column
            for col, col_data in X.iteritems():
                # If data type is non-numeric, try to replace all yes/no values with 1/0
                if col_data.dtype == object:
                    col_data = col_data.replace(['yes', 'no'], [1, 0])
                # Note: This should change the data type for yes/no columns to int

                # If still non-numeric, convert to one or more dummy variables
                if col_data.dtype == object:
                    col_data = pd.get_dummies(col_data, prefix=col)  # e.g. 'school' => 'school_GP', 's

                outX = outX.join(col_data)  # collect column(s) in output dataframe

            return outX

        X_all = preprocess_features(X_all)
        print "Processed feature columns ({}):-\n{}".format(len(X_all.columns), list(X_all.columns))
```

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

1.3.3 Split data into training and test sets

So far, we have converted all categorical features into numeric values. In this next step, we split the data (both features and corresponding labels) into training and test sets.

```
In [6]: from sklearn.cross_validation import train_test_split
        # First, decide how many training vs test samples you want
        num_all = student_data.shape[0]  # same as len(student_data)
        num_train = 300  # about 75% of the data
        num_test = num_all - num_train


        # TODO: Then, select features (X) and corresponding labels (y) for the training and test sets
        # Note: Shuffle the data or randomly select samples to avoid any bias due to ordering in the da
        X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_test, train_siz
        print "Training set: {} samples".format(X_train.shape[0])
        print "Test set: {} samples".format(X_test.shape[0])
        # Note: If you need a validation set, extract it from within training data
```

```
Training set: 300 samples
Test set: 95 samples
```

1.4 Training and Evaluating Models

Choose 3 supervised learning models that are available in scikit-learn, and appropriate for this problem. For each model:

- What are the general applications of this model? What are its strengths and weaknesses?

- Given what you know about the data so far, why did you choose this model to apply?
- Fit this model to the training data, try to predict labels (for both training and test sets), and measure the F1 score. Repeat this process with different training set sizes (100, 200, 300), keeping test set constant.

Produce a table showing training time, prediction time, F1 score on training set and F1 score on test set, for each training set size.

Note: You need to produce 3 such tables - one for each model.

1.4.1 Decision Tree Model

- Strengths:
  - It's easy to understand and to interpret.
  - The cost of predicting data is logarithmic in the number of data points used to train the tree.
  - Able to handle both numerical and categorical data.
- Weakness:
  - It's very sensitive to slight variations in the data.
  - It's easy to be overfit
- Why did you choose this model to apply?
  - Because we only have small number of data set and many features, the simple decision tree model could be the good choice.

In [7]:
```
# Train a Decision Tree model
import time

def train_classifier(clf, X_train, y_train):
    start = time.time()
    clf.fit(X_train, y_train)
    end = time.time()
    return clf, end - start

# Choose a model, import it and instantiate an object
from sklearn import tree
clf = tree.DecisionTreeClassifier()

# Fit model to training data
clf,_ = train_classifier(clf, X_train, y_train)  # note: using entire training set here
print clf
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort=False, random_state=None, splitter='best')
```

In [8]:
```
# Predict on training set and compute F1 score
from sklearn.metrics import f1_score

def predict_labels(clf, features, target):
    start = time.time()
    y_pred = clf.predict(features)
    end = time.time()
    return f1_score(target.values, y_pred, pos_label='yes'), end - start
```

```
        train_f1_score, _ = predict_labels(clf, X_train, y_train)
        print "F1 score for training set: {}".format(train_f1_score)
```

```
F1 score for training set: 1.0
```

```
In [9]: # Predict on test data
        test_f1_score, _ = predict_labels(clf, X_train, y_train)
        print "F1 score for test set: {}".format(test_f1_score)
```

```
F1 score for test set: 1.0
```

```
In [10]: # Train and predict using different training set sizes
         def train_predict(clf, X_train, y_train, X_test, y_test):
             clf, time_training = train_classifier(clf, X_train, y_train)
             score_training, _  = predict_labels(clf, X_train, y_train)
             score_testing, time_predicting = predict_labels(clf, X_test, y_test)
             return score_training, time_training, score_testing, time_predicting

         df_DT = pd.DataFrame(columns=['Set Size', 'Training time', 'Prediction time',
                                       'F1 score for training set', 'F1 score for test set'])

         for num_train in xrange(50, 350, 50):
             X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_test, trai
             score_training, time_training, score_testing, time_testing = train_predict(tree.DecisionTr
                                                                                         X_train, y_trai
             df_DT = df_DT.append({'Set Size':num_train,'Training time':time_training,
                         'Prediction time': time_testing,
                         'F1 score for training set': score_training,
                         'F1 score for test set': score_testing}, ignore_index=True)
```

```
In [11]: # Table of Decision Tree
         from IPython.display import display, HTML
         display(df_DT)
```

```
   Set Size  Training time  Prediction time  F1 score for training set  \
0        50       0.000901         0.000274                          1
1       100       0.001199         0.000986                          1
2       150       0.001897         0.000787                          1
3       200       0.003301         0.000405                          1
4       250       0.002769         0.002411                          1
5       300       0.003908         0.000298                          1

   F1 score for test set
0               0.655172
1               0.752000
2               0.638655
3               0.666667
4               0.746032
5               0.738462
```

1.4.2 SVM Model

- Strengths:
  - Effective in high dimensional spaces.

- It has a regularisation parameter, which makes the user think about avoiding over-fitting.
  - Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

- Weakness:
  - The biggest limitation of the support vector approach lies in choice of the kernel.
  - A second limitation is speed and size, both in training and testing.
  - It will not work well if there are a lof noise in data.

- Why did you choose this model to apply?
  - Because SVM could be effective in high dimensional spaces, this data set does have a lot of features.
  - Besides SVM could have better generalization than decision tree.

In [12]:
```
# Train a SVM model
from sklearn import svm
clf, _ = train_classifier(svm.SVC(class_weight='balanced'), X_train, y_train)  # note: using e
print clf
```

```
SVC(C=1.0, cache_size=200, class_weight='balanced', 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)
```

In [13]:
```
# Predict on training set and compute F1 score
train_f1_score, _ = predict_labels(clf, X_train, y_train)
print "F1 score for training set: {}".format(train_f1_score)
```

```
F1 score for training set: 0.860103626943
```

In [14]:
```
# Predict on test data
test_f1_score, _ = predict_labels(clf, X_test, y_test)
print "F1 score for test set: {}".format(test_f1_score)
```

```
F1 score for test set: 0.72
```

In [15]:
```
df_SVC = pd.DataFrame(columns=['Set Size', 'Training time', 'Prediction time',
                               'F1 score for training set', 'F1 score for test set'])
for num_train in xrange(50, 350, 50):
    X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_test, trai
    score_training, time_training, score_testing, time_testing = train_predict(svm.SVC(class_w
                                                                                X_train, y_trai
    df_SVC = df_SVC.append({'Set Size':num_train,'Training time':time_training,
                    'Prediction time': time_testing,
                    'F1 score for training set': score_training,
                    'F1 score for test set': score_testing}, ignore_index=True)
```

In [16]:
```
# Table of SVC
display(df_DT)
```

```
   Set Size  Training time  Prediction time  F1 score for training set  \
0        50       0.000901         0.000274                          1
1       100       0.001199         0.000986                          1
2       150       0.001897         0.000787                          1
3       200       0.003301         0.000405                          1
4       250       0.002769         0.002411                          1
```

```
5       300       0.003908         0.000298                         1

   F1 score for test set
0              0.655172
1              0.752000
2              0.638655
3              0.666667
4              0.746032
5              0.738462
```

1.4.3 AdaBoost Model

- Strengths:
  - It has good generalization.
  - It can achieve similar classification results with much less tweaking of parameters or settings.
  - Versatile - a wide range of base learners can be used with AdaBoost.
- Weakness:
  - It can be sensitive to noisy data and outliers.
  - Weak learner should not be too complex to avoid overfitting.
  - There needs to be enough data so that the weak learning requirement is satisfied.
- Why did you choose this model to apply?
  - Because we only have small number of data set and many features, the simple decision tree model could be the good choice for base estimator.
  - And AdaBoost could reach better generaliztion than Decision Tree does.

```
In [17]: # Train a AdaBoost model with Decision Tree as a base estimator
         from sklearn.ensemble import AdaBoostClassifier
         clf,_ = train_classifier(AdaBoostClassifier(), X_train, y_train)  # note: using entire trainin
         print clf
```

```
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
          learning_rate=1.0, n_estimators=50, random_state=None)
```

```
In [18]: # Predict on training set and compute F1 score
         train_f1_score,_ = predict_labels(clf, X_train, y_train)
         print "F1 score for training set: {}".format(train_f1_score)
```

```
F1 score for training set: 0.849056603774
```

```
In [19]: # Predict on test data
         test_f1_score,_ = predict_labels(clf, X_test, y_test)
         print "F1 score for test set: {}".format(test_f1_score)
```

```
F1 score for test set: 0.797101449275
```

```
In [20]: # Train and predict using different training set sizes
         df_Ada = pd.DataFrame(columns=['Set Size', 'Training time', 'Prediction time',
                                        'F1 score for training set', 'F1 score for test set'])
         for num_train in xrange(50, 350, 50):
             X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=num_test, trai
             score_training, time_training, score_testing, time_testing = train_predict(AdaBoostClassifi
                                                                                        X_train, y_trai
```

```
            df_Ada = df_Ada.append({'Set Size':num_train,'Training time':time_training,
                             'Prediction time': time_testing,
                             'F1 score for training set': score_training,
                             'F1 score for test set': score_testing}, ignore_index=True)
```

```
In [21]: # Table of AdaBoost
         display(df_Ada)
```

```
   Set Size  Training time  Prediction time  F1 score for training set  \
0        50       0.270257         0.009194                   1.000000
1       100       0.192780         0.009652                   0.972222
2       150       0.264084         0.010436                   0.976959
3       200       0.182143         0.009347                   0.867647
4       250       0.189830         0.009306                   0.875346
5       300       0.205099         0.009949                   0.846512

   F1 score for test set
0               0.615385
1               0.714286
2               0.744526
3               0.753846
4               0.727273
5               0.812030
```

1.5 Choosing the Best Model

- Based on the experiments you performed earlier, in 1-2 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?
- In 1-2 paragraphs explain to the board of supervisors in layman's terms how the final model chosen is supposed to work (for example if you chose a Decision Tree or Support Vector Machine, how does it make a prediction).
- Fine-tune the model. Use Gridsearch with at least one important parameter tuned and with at least 3 settings. Use the entire training set for this.
- What is the model's final F1 score?

1.5.1 Comparing Models

- Training time: AdaBoost > SVM > Decision Tree
- Predicting time: AdaBoost > SVM > Decision Tree
- F1 score for test set on size of 300: AdaBoost > SVM > Decision Tree

I would like to choose Adaboost for this case, because 1. The Fl scose of this model is the best based on the above results. 2. The training time could be acceptable, although it's highest in these three models. 3. Because our goal is to model the factors that predict how likely a student is to pass their high school final exam and take some intervention in advanced, we do have time to compute the result. So we should focus on the F1 score instead of computational cost.

1.5.2 How AdaBoost works

- AdaBoost is a type of "Ensemble Learning" where multiple learners are employed to build a stronger learning algorithm. AdaBoost works by choosing a base algorithm (e.g. decision trees) and iteratively improving it by accounting for the incorrectly classified examples in the training set.

- We assign equal weights to all the training examples and choose a base algorithm. At each step of iteration, we apply the base algorithm to the training set and increase the weights of the incorrectly classified examples. We iterate n times, each time applying base learner on the training set with updated weights. The final model is the weighted sum of the n learners.

In [22]:
```
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import f1_score
from sklearn.metrics import make_scorer
from sklearn.cross_validation import StratifiedShuffleSplit

param_grid = {"n_estimators": np.arange(3,20,2), "learning_rate": np.arange(0.5,0.8,0.02)}
scoring_function = make_scorer(f1_score, pos_label='yes', average='binary')

df = pd.DataFrame(columns=['Training Score', 'Test Score', 'Best n_estimators', 'Best learning
for train_index, test_index in StratifiedShuffleSplit(y_all, test_size=num_test, train_size=30
    X_train, X_test = X_all.iloc[train_index], X_all.iloc[test_index]
    y_train, y_test = y_all[train_index], y_all[test_index]
    clf = GridSearchCV(AdaBoostClassifier(), param_grid=param_grid, scoring=scoring_function)
    clf.fit(X_train, y_train)
    training_score,_ = predict_labels(clf.best_estimator_, X_train, y_train)
    testing_score,_ = predict_labels(clf.best_estimator_, X_test, y_test)
    df = df.append({'Training Score':training_score,'Test Score':testing_score,
                    'Best n_estimators': clf.best_params_['n_estimators'],
                    'Best learning_rate': clf.best_params_['learning_rate']}, ignore_index=Tru
    print "-------------------------------------------"
    print "Best parameters: {}".format(clf.best_params_)
    print "F1 score for training set: {}".format(training_score)
    print "F1 score for test set: {}".format(testing_score)
```

```
------------------------------------------
Best parameters: {'n_estimators': 5, 'learning_rate': 0.60000000000000009}
F1 score for training set: 0.845637583893
F1 score for test set: 0.788732394366
------------------------------------------
Best parameters: {'n_estimators': 5, 'learning_rate': 0.7200000000000002}
F1 score for training set: 0.818584070796
F1 score for test set: 0.765957446809
------------------------------------------
Best parameters: {'n_estimators': 5, 'learning_rate': 0.68000000000000016}
F1 score for training set: 0.825396825397
F1 score for test set: 0.783216783217
------------------------------------------
Best parameters: {'n_estimators': 11, 'learning_rate': 0.56000000000000005}
F1 score for training set: 0.857142857143
F1 score for test set: 0.828571428571
------------------------------------------
Best parameters: {'n_estimators': 5, 'learning_rate': 0.70000000000000018}
F1 score for training set: 0.806306306306
F1 score for test set: 0.811188811189
------------------------------------------
Best parameters: {'n_estimators': 3, 'learning_rate': 0.60000000000000009}
F1 score for training set: 0.808510638298
F1 score for test set: 0.845070422535
------------------------------------------
```

```
Best parameters: {'n_estimators': 3, 'learning_rate': 0.5}
F1 score for training set: 0.835886214442
F1 score for test set: 0.789115646259
------------------------------------------
Best parameters: {'n_estimators': 3, 'learning_rate': 0.5}
F1 score for training set: 0.833693304536
F1 score for test set: 0.826666666667
------------------------------------------
Best parameters: {'n_estimators': 3, 'learning_rate': 0.66000000000000014}
F1 score for training set: 0.810572687225
F1 score for test set: 0.830985915493
------------------------------------------
Best parameters: {'n_estimators': 7, 'learning_rate': 0.56000000000000005}
F1 score for training set: 0.843010752688
F1 score for test set: 0.805194805195
```

```
In [30]: best = df.ix[df['Test Score'].idxmax()]
         print "Max F1 score for training set: {}".format(best['Training Score'])
         print "Max F1 score for test set: {}".format(best['Test Score'])
         print "Best n_estimators: {}".format(best['Best n_estimators'])
         print "Best learning_rate: {}".format(best['Best learning_rate'])
         print "------------------------------------------"
         scores = []

         for train_index, test_index in StratifiedShuffleSplit(y_all, test_size=num_test, train_size=30
             X_train, X_test = X_all.iloc[train_index], X_all.iloc[test_index]
             y_train, y_test = y_all[train_index], y_all[test_index]
             clf = AdaBoostClassifier(n_estimators=int(best['Best n_estimators']), learning_rate=best['
             clf.fit(X_train, y_train)
             score,_ = predict_labels(clf, X_test, y_test)
             scores.append(score)

         print "Average F1 score for tuned parameters: {}".format(np.mean(scores, dtype=np.float32))
```

```
Max F1 score for training set: 0.808510638298
Max F1 score for test set: 0.845070422535
Best n_estimators: 3.0
Best learning_rate: 0.6
------------------------------------------
Average F1 score for tuned parameters: 0.81219369173
```

1.6 Feedback

Based on coufusion_matrix, the model does well with the "yes" label, but it does poorly with the "no" label. Besides I think those "no" label students should give more helps in advanced. But I have no idea how should I do. Could you give me some tips to work on?

```
In [31]: from sklearn.metrics import confusion_matrix
         print confusion_matrix(y_test, clf.predict(X_test),labels=['yes', 'no'])
```

```
[[60  4]
 [22  9]]
```