student_intervention

April 18, 2016

1 Project 2: Supervised Learning

1.0.1 Building a Student Intervention System

1.1 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: Classification. This is because the output of our function should tell us weather or not to intervene with a given student. This means constructing a boolean function.

1.2 2. Exploring the Data

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

To execute a code cell, click inside it and press ***Shift+Enter****.*

```
In [2]: # Import libraries
        import numpy as np
        import pandas as pd
```

```
In [3]: # 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

Use the code block below to compute these values. Instructions/steps are marked using ***TODOs****.*

```
In [4]: # TODO: Compute desired values - replace each '?' with an appropriate expression/function call
        n_students = student_data.index.size
        n_features = student_data.columns.size - 1
        n_passed = student_data[ student_data['passed'] == 'yes' ].index.size
        n_failed = n_students - n_passed
        grad_rate = 100*float(n_passed) / float(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)
        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 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.

Let's first separate our data into feature and target columns, and see if any features are non-numeric.
Note: For this dataset, the last column ("passed") is the target or label we are trying to predict.

```
In [5]: # 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 [6]: # 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)
        y_all = y_all.replace(['yes','no'],[1,0])
        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 [7]: # 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
        from sklearn.cross_validation import train_test_split
        X_train, X_test, y_train, y_test = train_test_split( X_all , y_all , test_size = num_test, rando

        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 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 is the theoretical O(n) time & space complexity in terms of input size?
- 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.

1.5 Answer

In the following n_f denotes the number of featurs, and n denotes the number of samples. In the cases below, the numerically observed training times do not seem to corroborate the theoretical Big O predictions. In fact, training time actually appears to drop as the amount of data increases. I can not completely explain this except to say that the big O complexity computations only apply in the limit of large n. Perhaps there is simply not enough data to observe this asymptotic behvior.

1.5.1 Decision Trees :

- **Complexity:** I'm not sure how to compute the complexities for a generic decision tree. According to the sci-kit learn documentation (http://scikit-learn.org/stable/modules/tree.html#complexity), if you assume that the data can be split perfectly at each level of the tree then you can calculate the following. Training time is $O(n_f n_s^2 \log(n_s))$ in time. Query time is $O(\log(n_f))$. The storage cost of the tree is $O(n)$. It's notable that the training time complexity differs from standard algorithms like C4.5 which have a complexity of $O(nn_f^2)$.
- **Applications:** Decision trees are useful when individual features can efficiently split the data. For example, in the case of determining if it will rain, knowing weather or not there are clouds in the sky is a feature worth splitting your algorithm on. Moreover, Decision trees are flexible with regards to the feature space. The feature space can be numerical or categorical or a product of both.
- **Strengths:** Decisision trees are human interpretable (assuming the features are), and they are space efficient. Morevoer, they have very low bias. In the language of function spaces, the decision trees are generated by rectangular indicator functions on the feature space. The function space generated by these box-functions is just the space of measurable function on the feature space (with range in the target set). In otherwords, Decision trees have lots of expressive power.
- **Weaknesses:** Practical training algorithms (i.e. polynomial time complexity), such as C4.5, will always produce "suboptimal" trees (where optimality is determined by the the height of the tree) because the building of an optimal decision tree is an NP problem (see doi:10.1016/0020-0190(76)90095-8). One side-effect of this sub-optimality is that the classifier itself is less stable with respect to perturbations in the data, as a number of seemingly different trees will exhibit the same performance. Morevoer, Decision trees are subject to overfitting. This is especially the case when the leaves contain only one member of the training set, which is the natural place where training ends unless you actively prevent this outcome through pruning or other variance reduction measures.
- **Why should we condiser it?:** We should consider decision trees because our data is a mixture of numerical and categorical. Moreover, much of our data is binary, and thus well suited to the decision tree paradigm (if in fact a good decision tree can be found).

Performance:

train set size	training time	prediction times (training/test)	F1 scores (training/test)
100	0.001682	0.000491 / 0.000803	0.828571 / 0.794326
200	0.001660	0.000582 / 0.000631	0.826667 / 0.779412
300	0.003153	0.000743 / 0.000495	0.803695 / 0.755556

1.5.2 Naive Bayes :

- **Complexity:** Training for Naive Bayes entails computing and storing the probabilities $P(y)$ and $P(x_i \mid y)$. Estimating $P(y)$ via its proportion of the training set requires counting, and is thus an $O(n)$ time, $O(n)$ space operation. Similarly, estimating $P(x_i \mid y)$ is a $O(nn_y)$ in time and $O(n)$ in space where n_y is the number of labels (perhaps this can be improved, but this is what I get using a naive for-loop). Querying is (worst case scenario) $O(n)$ in time, and $O(1)$ in additional space.
- **Applications:** Naive Bayes assumes that for a given label, each feature is independent of the others (i.e. we assume $x_i \mid y$ is independent of $x_j \mid y$). Verifying independence is typically not possible, but you can certainly accumulate evidence for this assumption. Even if the assumption is false, the algorithm will still work.
- **Strengths:** Fast training time. If you are only going to make a single query this might be a very efficient algorithm. Additionally, if the independence assumption is true, Naive Bayes can have very low bias.
- **Weaknesses:** If your features are dependent, than you forfit the oppurtunity to exploit dimension reductions induced by such depedencies. Also, there is potential for overfitting, as Naive Bayes is a lazy learner and thus prone to simply memorizing the training set rather then generalizing it. If a particular (discrete) feature value is not present in the training set Naive Bayes will estimate it's probability to be 0. As a result, the output $\hat{y} = \text{argsup} P(x \mid y)P(y)$ is ill-defined when this feature is encountered because the $P(x \mid y)$ will be estimated as 0.
- **Why did you choose this?** I have not tested for independence, however if the samples are independent, than this algorithm might beat the other algorithms I've chosen to test. Many educators implicitly hold this indpendence assumption (at least in some high level sense) when they speak of each student being judged independently, so perhaps there is some merit to it.

Performance:

training set size	training time	prediction times (train/test)	F1 scores (training/test)
100	0.001682	0.000491 / 0.000803	0.828571 / 0.794326
200	0.001660	0.000582 / 0.000631	0.826667 / 0.779412
300	0.003153	0.000743 / 0.000495	0.803695 / 0.755556

1.5.3 K-nearest neighbors :

- **Complexity:** Training is $O(1)$ in time and $O(n)$ in space. Querying is $O(log(n) + k)$ in time if the feature vectors are sorted and $O(n + k)$ otherwise. Querying is $O(1)$ additional space, in time.
- **Applictions:** Such an algorithm is biased towards smoother data, and useful when the numer of queries is small because we need to store and search among all samples in the training set for each query (expensive). Moreover, the algorithm is useful when the data is mixed between categorical and numerical.
- **Strengths:** Very fast training time. Highly flexible, as one can choose the distance function as well as k.

- **Weakness:** Perhaps it is too flexible. It takes a lot of work to optimize the parameters. When k is small, then the training set might be too influential, and we might overfit (i.e. high variance). When k is large we might suffer from high bias.
- **Reason for choosing this algorithm:** Unlike Naive Bayes, no such independence assumption is made. Moreover, the metric can be chosen to specialize the algorithm to data sets with Boolean features and numerical features, such as what we have here. The flexibility of this algorithms gives it more knobs to tweak and it might have superior performance to the other algorithms after doing a grid search over the parameter space.

Performance:

training set size	training time	prediction times (training/test)	F1 scores (training/test)
100	0.000797	0.012753 / 0.012097	0.765957 / 0.729927
200	0.001103	0.017608 / 0.008643	0.840764 / 0.727273
300	0.001711	0.025993 / 0.007770	0.824295 / 0.757143

In [29]:
```
# Train a model
import time

def train_classifier(clf, X_train, y_train):
    #print "Training {}...".format(clf.__class__.__name__)
    start = time.time()
    clf.fit(X_train, y_train)
    end = time.time()
    #print "Training time (secs): {:.7f}".format(end - start)
    return end-start
```

In [30]:
```
# Predict on training set and compute F1 score
from sklearn.metrics import f1_score

def predict_labels(clf, features, target):
    #print "Predicting labels using {}...".format(clf.__class__.__name__)
    start = time.time()
    y_pred = clf.predict(features)
    end = time.time()
    #print "Prediction time (secs): {:.7f}".format(end - start)
    return f1_score(target.values, y_pred, pos_label=1), end-start
```

In [33]:
```
# Train and predict using different training set sizes
def train_predict(clf, X_train, y_train, X_test, y_test):
    #print "------------------------------------------"
    #print "Training set size: {}".format(len(X_train))
    training_time = train_classifier(clf, X_train, y_train)
    F1_train, pred_time_train = predict_labels(clf, X_train, y_train)
    F1_test, pred_time_test = predict_labels(clf, X_test, y_test)
    print "{:d} | {:5f} | {:5f} / {:5f} | {:5f} / {:5f} ".format(len(X_train) ,\
                                                                training_time,\
                                                                pred_time_train,\
                                                                pred_time_test,\
                                                                F1_train, F1_test )

# TODO: Run the helper function above for desired subsets of training data
# Note: Keep the test set constant
```

In [34]:
```
# TODO: Train and predict using two other models
# TODO: Choose a model, import it and instantiate an object
num_train_arr = [100,200,300]

from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import BernoulliNB
from sklearn.neighbors import KNeighborsClassifier

dt_clf = DecisionTreeClassifier(random_state = 32)
nb_clf = BernoulliNB()
knn_clf = KNeighborsClassifier(metric='jaccard')
clf_arr = [dt_clf,nb_clf,knn_clf]

for clf in clf_arr:
    print "Training {}...".format(clf.__class__.__name__)
    for num_train in num_train_arr:
        f1_dt = train_predict( clf , X_train[0:num_train], y_train[0:num_train], X_test, y_tes
```

```
Training DecisionTreeClassifier...
100 | 0.001620 | 0.000258 / 0.000252 | 1.000000 / 0.603448
200 | 0.005706 | 0.000516 / 0.000322 | 1.000000 / 0.716418
300 | 0.003874 | 0.000516 / 0.000315 | 1.000000 / 0.600000
Training BernoulliNB...
100 | 0.001682 | 0.000491 / 0.000803 | 0.828571 / 0.794326
200 | 0.001660 | 0.000582 / 0.000631 | 0.826667 / 0.779412
300 | 0.003153 | 0.000743 / 0.000495 | 0.803695 / 0.755556
Training KNeighborsClassifier...
100 | 0.000797 | 0.012753 / 0.012097 | 0.765957 / 0.729927
200 | 0.001103 | 0.017608 / 0.008643 | 0.840764 / 0.727273
300 | 0.001711 | 0.025993 / 0.007770 | 0.824295 / 0.757143
```

1.6 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?

Choice of model: I've come to the conclusion that the naive algorithm is best suited to the data. The F_1-scores for Decision trees seem to indicate both algorithms are suffering over-fit when we use the out-of-the-box implementations. This is because the F_1-scores seem to vary by more than 0.15 when comparing the score of the test set vs the training set. Additionally, while the k-nearest neighbors algorithm does not appear to suffer over-fitting, it still has a lower F_1 score than Naive bayes on smaller test set size. A high F_1-score indicates: 1. When a student is likely to fail, our algorithm will suggest intervention (i.e. high recall) 2. When the algorithm suggests intervention, it is usually the case that he or she is in danger of failing (i.e. high precision)

These are both things which we'd like to maximize in our intervention algorithm.

Given the low number of records being considered (hundreds to thousands) we should not be too concerned with training and prediction times complexities. Thus a performance score, like F_1, alone might be a reasonable criterion.

How the model works: In this problem we'd like to know weather or not to intervene with a given student's coursework. Naive-Bayes will output "yes" or "no" by estimating the most likely outcome, given the students profile. For example, if we only tracked the gender, school district, and success/failure of past students we could use the estimate

$$\text{probability of a female in disctrict } d \text{ failing} = \frac{\text{number of females in } d \text{ that failed}}{\text{number of females in } d}.$$

This might be an okay estimate. However, we need to at least ensure that the denominator of the above fraction is not 0. For example, there may be no females in certain districts. However, we may get around this issue if knowing that failed student is female tells you nothing about her district, and vice versa. This indpendence assumption allows us to use the estimate:

$$\text{probability of a female in disctrict } d \text{ failing} = \frac{\text{number of females that failed}}{\text{number of failed students}} \times \frac{\text{number of students in } d \text{ that failed}}{\text{number of failed students}} \times \frac{\text{number}}{\text{numb}}$$

In this case we can obtain a non-trivial esimate for the probability of failure, even if our sample of district d contains only males. In anycase, the classification for a female in district would be "yes intervene" if her probability of failure exceeded her probability of success.

** Fine tuning:** The assumption on $P(x_i \mid y)$ is based on a smoothed version of a sum of dirac-delta distrbutions centered at the training features which have label y. The parameter $\alpha \in [0, 1]$ is a smoothing parameter. We test 10 values of α. Morever, it is not clear if one should fit the prior value $p(y)$ to the data, or assume the data is not a good sample and assume complete ignorance by taking the more modest estimate of a uniform distirbution. This is set by the parameter "fit_prior" which is a Boolean parameter. Finally, the parameter "binarize" provides a thresh-hold which helps convert non-binary features into binary ones. We test 21 values for this parameter. Given this parameter space, we find the optimal value for α is 1.0, the optimal value for "fit prior" is "True", and the optimal value for "binarize" is 1.0. These values yield an F_1 score on the test set of 0.786, which is a higher than any of the values appearing in the earlier table.

```
In [14]: # TODO: Fine-tune your model and report the best F1 score, use GridSearch
         from sklearn.grid_search import GridSearchCV
         alpha_vals = np.linspace(0.0,1.0,10)
         fit_prior_vals = [True,False]
         binarize_vals = np.linspace(0.0, 2.0, 21)
         parameters = { 'alpha':alpha_vals, 'fit_prior':fit_prior_vals, 'binarize':binarize_vals}
         model = BernoulliNB()
         clf = GridSearchCV( model , parameters , scoring = "f1" )
         clf.fit( X_train , y_train )
         print clf.best_estimator_
         print 'f1 score = {:f}'.format(f1_score( clf.predict(X_test) , y_test))
```

```
BernoulliNB(alpha=1.0, binarize=1.0, class_prior=None, fit_prior=True)
f1 score = 0.785714
```

In [ ]: