Gilad Gressel
December 21th, 2015

Project 2, Student Intervention System

Overview

In this project we are given a dataset of which contains many attributes of students who are either failing or passing. We are asked to create a classifier which can accurately predict which students require intervention, those who are at risk of failing, or currently failing.

This is a strict classification problem because the classifier output will be binary, either a student requires intervention or the student does not. Using 30 features and 395 examples we must build a model which can accurately perform this classification.

The Data

The data was provided by Udacity, taken from UC Irvine, originally from Paulo Cortez, University of Minho, GuimarÃ£es, Portugal, http://www3.dsi.uminho.pt/pcortez . There are a total of 395 students in the dataset and 30 features and 1 label (pass / fail). 265 students passed while 130 students failed, the current graduation rate is 67%.

A large portion of the features are non-numerical and need to be converted in order to be used in a classifier. For example the feature 'sex' has two options, either 'F' for female or 'M' for male. We can simply replace all instances of 'M' with the digit '1' and 'F' with the digit '0'. This converts a categorical feature (male versus female) into a binary one (1 versus 0), and thus makes the feature useable.

A more complicated feature type are categorical features. One example of a categorical feature is 'Mjob'. The 'Mjob' feature describes the student's mother's job, which can be one of the following options: 'teacher', 'health', 'services', 'at_home' or 'other'. Obviously we cannot convert five options into 0's and 1's. One quite natural thought would be to convert each option into a number in a range, i.e: 'teacher' → 0, 'health' → 1, 'services' → 2, etc. While this option works, it also introduces the notion that 'health' is somehow closer to 'services' than 'teacher', when this in fact is not the case.

A better option is to split the original feature 'Mjob', into 5 features, each one being a binary feature of it's own. Thus we create 'Mjob_teacher', 'Mjob_health', 'Mjob_services', etc. This approach is better because each feature is kept at the same numerical distance from all the other features. Essentially this method converts the categorical feature in a vector of binary features. After preprocessing all the features we now have 48 features and 1 label.

The code which accomplishes this task is found in the preprocessing section of the ipython notebook. Finally we separate our data into two sets, 75% for training and 25% for testing, this leaves us 296 training samples and 99 test samples.

The Models

The three models that I chose to pursue were Support Vector Machines (SVM), K-Nearest Neighbors, and Decision Trees. These three models are all classification algorithms which differ significantly from each other “under the hood”. Each algorithm has pro’s and con’s for the current task. Given that our model is restricted by computational expense, time to predict, and time to train, these three algorithms each to naturally excel in different areas. Also given that we have 48 features and 300 data points, all these models should perform adequately. In the following tables all tests were run 100 times and the mean is reported. This was done to eliminate variability that naturally occurs when randomly splitting a data set that is so small.

All tests are mean of 100 random runs			
Support Vector Classification (SVC)			
Number Samples	100	200	296
Training Set			
Train Time	0.002026	0.004884	0.009327
Prediction Time	0.001217	0.00342	0.006794
F1 Score	0.873815	0.872851	0.867183
Testing Set			
Prediction Time	0.001258	0.001819	0.0022228
F1 Score	0.796269	0.799968	0.800255

Generally speaking, SVC’s are slower to train due to the complex math, computationally expensive and moderate in time to predict. However they are generally robust and quite accurate. We see this reflected in the table, note that the prediction time is faster than the training time. The F1 score of 80 for testing is the best result we’ll see for any of the algorithms.

All tests are mean of 100 random runs			
K-Nearest Neighbors			
Number Samples	100	200	296
Training Set			
Train Time	0.000996	0.001412	0.001307
Prediction Time	0.00227	0.004764	0.008936
F1 Score	0.81595	0.822937	0.825737
Testing Set			
Prediction Time	0.002152	0.002975	0.003611
F1 Score	0.77415	0.77477	0.77982

K-Nearest Neighbors is a lazy algorithm which simply records information during training and computes only during prediction. Thus, KNN trains in constant time training, however making predictions is done in logarithmic time. We see that the train time is a constant .001 seconds and the prediction time begins to rise quite quickly as the amount of data increases. The F1 score of 77.9 is slightly lower than SVC.

All tests are mean of 100 random runs			
Decision Trees			
Number Samples	100	200	296
Training Set			
Train Time	0.001164	0.002005	0.00256
Prediction Time	0.000343	0.000376	0.000429
F1 Score	1	1	1
Testing Set			
Prediction Time	0.00032	0.000389	0.000268
F1 Score	0.707672	0.703483	0.71509

The training time of a decision tree is faster than that of SVM's but slower than KNN. It's prediction time is extremely fast because the only calculations that need to be made are a series of 'if' statements that require simple expressions to be checked. We see this reflected the table, with a prediction times in hundreds of microseconds, versus the nanosecond times of the other classifiers.

The F1 score on the training set is perfect (1), this is due to the nature of Decision Trees which are not parameterized to handle overfitting. A decision tree algorithm will grow until it can perfectly account for all the data-points in the training data. This generally will cause overfitting and is most certainly the case in my table as I ran the Decision Tree classifier on it's default settings. The best F1 score is found at 71.5, which significantly lower than the other two algorithms.

In selecting which algorithm to pursue for the final model, we can compare the three algorithms across a few different factors. The most important factor is the F1 score as this is a balanced measurement of precision and recall. There is certainly no point in a having a very fast and efficient model that always predicts that all students will pass.

	SVC	KNN	DT
F1	80	77.9	71.5

We see that SVC and KNN strongly outperform DT.
The next factor to examine is the time to train. This reflects how much computational power / time the system will need to train the classifier. Assuming that re-training will be done regularly as more data is recorded and stored, this factor is also quite important, both in the cost of

computational power and the time we wait to use the classifier. Note we will use the training times from the largest training set in order to examine the worst case.

	SVC	KNN	DT
Train Time (296 samples)	0.0093	0.0013	0.00256

In this case we see that KNN is fastest, followed by DT, and SVC the slowest. Because all the times are in milliseconds, this factor seems negligible in all cases. However these differences in time change dramatically when the data becomes large(r). For example using dummy data (created with numpy.random), of 75,000 samples,

	75k Samples, Dummy Data		
	SVC	KNN	DT
Train time	824.9	0.456	14.8

SVC takes 825 seconds (13.5 minutes) time to train. The increase in training time is far from linear, 75,000 samples is roughly 253 times larger than the original training size, yet 13.5 minutes is roughly 89,000 times larger than 9 milliseconds.

Thus we see that SVC will take significantly longer amounts of time to train as the data grows. This factor of time to train becomes significant if the school district plans on re-training regularly and amassing large amounts of data. In contrast KNN takes .46 seconds (354 times longer), and DT takes 14.8 seconds (5800 times longer) on the same 75,000 sample training set. These speeds are much faster in comparison.

Next we examine time to predict on the entire training data, in order to take the upper bound of prediction times.

	SVC	KNN	DT
Prediction Time 296 samples	0.006794	0.008936	0.000429

Here we see that DT's are the fastest, followed by SVC and KNN. Prediction time is most likely the second most important factor as predictions will most likely be made in real-time by administrators and teachers. While training can be done offline and uploaded when done, predictions should be accessible in real-time. This means that we want it to be fast. Lets look at the dummy data on this as well.

	75k Samples, Dummy Data		
	SVC	KNN	DT
Prediction Time	483.4	1257.4	0.041

Here we see that SVC's prediction time increases by 71,000 times, KNN by 140,000 times, and DT by 95 times.

It's worth noting that all the algorithms did better with more data, none did significantly better or worse with *less* data, i.e: they all increased about 1 point per 100 extra training samples.

So overall we see the highest performance with SVC, it has the highest F1-score (80) and reasonable times to train / predict. KNN performs well in F1, but very poorly in prediction which I believe will be far more important than training. DT are very fast, yet not very accurate, with a low F1 score of 71.5. SVC is our winner here.

The Best Model - Support Vector Machines - How they work.

Support Vector Machines (SVM) construct a decision boundary which maximizes the distance between both classes. What this means is that the algorithm looks through the data and is able to find a line which separates the data into two classes (passing vs failing). This line is called the decision boundary.

What is important and unique about SVM's is that when it finds a decision boundary, it makes the boundary exactly in the middle of the two classes. Imagine two buildings being separated by a road. The buildings on the left of the road are called 'L' buildings and the ones on the right are called 'R' buildings. It's possible to draw a line down the road in many places, and no matter where we draw the line it will separate the buildings, this line is our decision boundary. But you can intuitively understand that the best place to draw the line is down the *middle* of the road, somehow you know this separates the buildings the most efficiently. SVM's always draw decision boundaries which maximize the space between the classes.

We like this because it means that *newly* constructed buildings will likely be categorized correctly as 'L' buildings or 'R' buildings. Additionally using advanced mathematics we are able draw curves, circles and all sorts of interesting lines with SVM's. This makes them very robust and applicable to many different kinds of data.

The process of finding the decision boundary is done while training the classifier, the algorithm finds a hyperplane which separates the data. When it's time to make new predictions, the SVM simply has to look and see what side of the decision boundary the new data lies in.

Final F1 Score, Tuned Parameters.

After fine tuning the model with an exhaustive grid search, across many different parameters we find the best model to the following. I tested three different Kernels, each with four settings for C (an outlier / overfitting control) and for the RBF kernel, two settings for gamma.

The Kernel : RBF, C : 10, Gamma : 0.001, F1: 80.5

We see that the F1 score has only increased slightly, this is probably because the default parameters on the original model are actually quite close to the chosen ones.