Gilad Gressel
February 15, 2016

Project 2, Student Intervention System

Overview

In this project we are given a dataset which contains many attributes of students who are either failing or passing. We are asked to create a classifier that will be able to predict whether or not a student is failing based on these attributes. This is a strict classification problem because the classifier output will be binary, either a student requires intervention or the student does not.

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'. 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. 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 (and thus appropriate for the task at hand), which differ significantly from each other "under the hood". Training on 48 features and 296 data points, all these models should perform adequately, however we will see variations in time to train/predict and the F1 score.

Support Vector Machines (SVM)

SVM's maximize the margin in the decision boundary which increases the likelihood that future predictions will be correct. They also incorporate a parameter which helps with overfitting (C), and due to the kernel trick they can be adapted to many different types of data-sets, allowing the user to inject expert domain knowledge into the model. The downsides of SVM's are that they are generally computationally expensive due to the quadratic programming involved.

Support Vector Classification (SVC)			
Number Samples	100	200	296
Training Set			
Train Time	.002	.004	.007
Prediction Time	.001	.002	.005
F1 Score	.9659	.92957	.9121
Testing Set			
Prediction Time	.001	.001	.003
F1 Score	.84523	.82716	**.827**

K-Nearest Neighbors (KNN)

K-Nearest Neighbors is a non-parametric lazy algorithm which only records information during training and makes all computation during prediction. KNN is strong in that it is capable of learning complex concepts with a very simple method, however it's weakness lies in slow prediction times. Also, due to instance based learning (non-parametric), we cannot interpret any meaning from the trained learner, i.e. KNN cannot tell us any generalizations about the features, it can only make predictions

K-Nearest Neighbors			
Number Samples	100	200	296
Training Set			
Train Time	.004	.001	.001
Prediction Time	.003	.006	.008
F1 Score	.769	.811	.815
Testing Set			
Prediction Time	.001	.003	.003
F1 Score	.726	.7333	**.76129**

We see that the train time is a constantly less than a millisecond (.001) and the prediction time begins to rise quite quickly as the amount of data increases.

Decision Trees (DT)

Decisions trees are: trained quickly, predict quickly (logarithmic to the number of features), do not need pre-processing of features, can handle multi class classifications, and automatically sort features by relevance (information gain). The downsides of DT's is that they are prone to overfitting (though, hyper-parameters exist to combat this) and that they create biased trees if the classes are unbalanced in the training set.

Decision Trees			
Number Samples	100	200	296
Training Set			
Train Time	.002	.001	.002
Prediction Time	.001	00	0.0
F1 Score	1.0	1.0	1.0
Testing Set			
Prediction Time	.0.	0.0	0.0
F1 Score	.6504	.6718	**.688**

We see that the training time of a decision tree is faster than that of SVM's but slower than KNN. Prediction times are the fastest of all three. The F1 score on the training set is perfect (1), this is because the DT is not hyper-parameterized to handle overfitting (pruning, max-depth etc). The best F1 test score is found at 68.8 which is significantly lower than the other two algorithms.

Selecting the Best Model

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, and this metric tells us how much we can trust the results of the classifier.

	SVC	KNN	DT
F1	82.7	76.1	68.8

SVC is the clear winner here.

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 important. 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)	0.007	0.001	0.002

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,

	SVC	KNN	DT
Train Time (75k Samples, Dummy Data)	833.61	0.449	12.059

SVC takes 833.61 seconds (13.9 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 14.5 minutes is roughly 119,087 times larger than 7 milliseconds. In contrast KNN takes .449 seconds (449 times longer), and DT takes 12.059 seconds (6029.5 times longer) on the same 75,000 sample training set. KNN is the winner here.

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)	0.003	0.008	0

Here we see that DT's are the fastest, followed by SVC and KNN. Lets look at the dummy data on this as well.

	SVC	KNN	DT
Prediction Time (75k Dummy Data)	427.217	1110.638	.036

Here we see that SVC's prediction time increases by 142,405 times, KNN by 138,829 times, and DT by… much less! (the prediction time being limited by 3 decimal places is our problem here). DT is definitely our winner here. Note that SVC is still much faster than KNN.

We have a different algorithm winning in each category, so we must rank our performance metrics by order of importance. As stated before, the F1 score is certainly the most important metric, as there is no point to having a classifier which is wrong. This rules out DT's and leaves us to decide between KNN and SVC.

The second most import metric is time to predict. 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. Thus we have a clear winner as SVC is much faster than KNN in predictions.

It's worth noting that while that two algorithms did better with more data (KNN and DT)and SVC did worse. This is quite strange, and I can only guess that reshuffling the data would change these results. Either way, SVC is our strongest performer, and I would recommend all schools use *all* their data to train their models, because the computation expense for this sized data is negligible.

So overall we see the highest performance with SVC, it has the highest F1-score (83) and reasonable times to train / predict. KNN performs moderately in F1, but very poorly in prediction time. DT are very fast, yet not very accurate, with a low F1 score of 68.8. 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) as much as possible. 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. We like this because it means that *newly* constructed buildings will likely be categorized correctly as 'L' buildings or 'R' buildings. This is what SVM's do. 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.

Now that we have chosen our model, we tune it's hyper-parameters in order to maximize it's effectiveness. This is done by brute force, utilizing sk-learn's gridsearchCV module. It will train and test many models with many different hyper-parameters, and remember which performed best. We will use the F1-score as our performance metric.

In order to find the correct parameters and an F1-score reflective of this data set, we will run gridsearch on the training data, this will tune the model and provide us with the best possible classifier from this particular training data.
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 results:

Tuned Parameters

The Kernel: *RBF*, C: *10* Gamma: *0.001*

Using these tuned parameters we will now build our classifier on the training data, and test it against the held-out test set. The resulting F1-score is: 83.01, which is .4 points higher than the original F1 score, so the tuning did improve the model.