Project 2- Student intervention System

Udacity - Machine Learning Nanodegree

Alexis Tremblay - 4 décembre 2015

Project Description

As education has grown to rely more and more on technology, more and more data is available for examination and prediction. Logs of student activities, grades, interactions with teachers and fellow students, and more are now captured through learning management systems like Canvas and Edmodo and available in real time. This is especially true for online classrooms, which are becoming more and more popular even at the middle and high school levels.

Within all levels of education, there exists a push to help increase the likelihood of student success without watering down the education or engaging in behaviours that raise the likelihood of passing metrics without improving the actual underlying learning. Graduation rates are often the criteria of choice for this, and educators and administrators are after new ways to predict success and failure early enough to stage effective interventions, as well as to identify the effectiveness of different interventions.

Toward that end, your goal as a software engineer hired by the local school district is to model the factors that predict how likely a student is to pass their high school final exam. The school district has a goal to reach a 95% graduation rate by the end of the decade by identifying students who need intervention before they drop out of school. You being a clever engineer decide to implement a student intervention system using concepts you learned from supervised machine learning. Instead of buying expensive servers or implementing new data models from the ground up, you reach out to a 3rd party company who can provide you the necessary software libraries and servers to run your software.

However, with limited resources and budgets, the board of supervisors wants you to find the most effective model with the least amount of computation costs (you pay the company by the memory and CPU time you use on their servers). In order to build the intervention software, you first will need to analyze the dataset on students' performance. Your goal is to choose and develop a model that will predict the likelihood that a given student will pass, thus helping diagnose whether or not an intervention is necessary. Your model must be developed based on a subset of the data that we provide to you, and it will be tested against a subset of the data that is kept hidden from the learning algorithm, in order to test the model's effectiveness on data outside the training set.

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?

This is a classification problem as we are trying to put a label on the students. Are they at risk of failing or not. If so then precautions and measures will be taken by the school to help those in need.

Exploring the data

```
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%
```

Training and Evaluating Models

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

SVC

Complexity[1]

Complexity varies between $O(n_{features}*n_{samples}^2)$ and $O(n_{features}*n_{samples}^3)$.

In our case, the number of sample is 395 and the number of features (after transformation) is 48. Because it is an eager learner the training time will grow as the dataset grows. It will grow exponentially fast.

If the dataset was sparse we could reduce the complexity, replacing $n_{features}$ by the average number of non zero values for all data points.

[1] http://scikit-learn.org/stable/modules/svm.html#complexity

Prediction time is constant as we are just letting the data point go through the learned formula, *O(1)*.

The space required is $O(n_{samples})$ for the training and constant for predicting as we only keep a few parameters to make the prediction, so *O(1)*.

The advantages of support vector machines are[2]:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, the method is likely to give poor performances.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.

Why this model:

- We usually have to worry about the curse of dimensionality where we need more and more data with high dimensions. Given the train size we have to try, as low as 100, it brings the number of datapoint closer to the number of features which would contradict the recommendations to avoid the curse of dimensionality. But because SVM deals well in those circumstances I chose to try it.

[2] http://scikit-learn.org/stable/modules/svm.html#

	Training set size		
	100	200	300
Training time (secs)	0,0040	0,0030	0,0100
Prediction time (secs)	0,001	0,003	0,009
F1 score for training set	0,8777	0,8679	0,8761
F1 score for test set	0,7746	0,7815	0,7838

kNN

Complexity

The complexity of a prediction in kNN, as we saw in class, is $k + log(n)$ if the input feature is one dimensional and ordered. Searching where a number fits is $log(n)$ and we have to do $k-1$ more comparison. So $k + log(n)$ because k can be a function of n.

But we don't have a one dimensional input feature, so ordering is sort of meaningless. Even though there are schemes to reduce the search of inserting one point in a multidimensional space, I will consider the worst case scenario. We need to have a measure of distance or similarity between two points which will be done necessarily by using all the dimensions/features of both points, wether it's euclidian distance or manhattan or any other, so $O(n_{features})$. Then we need to compare every points to the one we are inserting. Fortunately we can keep the calculated distance as we go along and keep the k top closest. $O(n_{samples})$ comparison are required. For a total of $O(n_{features} * n_{samples})$

The complexity of training, as we also saw in class is constant, $O(1)$. The space required for training and predicting is the size of the dataset, $O(n_{samples})$.

The advantages of kNN are[3]:

- Really fast training time. In fact, no time at all since it's a lazy learner. It just sits there waiting to be queried for predictions. No matter what the size of the training set is, it's a constant learning rate.

[3] http://scikit-learn.org/stable/modules/svm.html#

- Easy to add data points for training. If we want to add new samples we don't have to retrain anything.
- It works well on noisy data that is not easily separable.
- Not affected by k (depending on the choice of algorithm, but we are talking about brute force here) because we are already looking at all the datapoint.
- Simple and easy to understand.

The disadvantages of kNN include:

- Prediction is dependent on the sample size. The bigger it is, the more comparison we have to do.
- It is not clear what is the best value for k
- It is not clear what is the best distance metric
- Being distance based, all the features contribute equally which could be a bad thing if some are useless

Why this model:

- Despite its simplicity, kNN is well respected
- I wasn't sure how well the classes were separable and kNN works well in those cases
- A lot of features are binary so they mostly sit on the unit sphere and most of the features are will contribute more or less equally with some exception such as age and absence.

	Training set size		
	100	200	300
Training time (secs)	0,019	0,001	0,001
Prediction time (secs)	0,007	0,005	0,01
F1 score for training set	0,8060	0,8800	0,8809
F1 score for test set	0,7246	0,7692	0,7801

AdaBoost

Complexity

Using boosting with decision stump (decision tree with depth 1). A decision tree of depth 1 will have to analyze $n_{features}$ to find the one with the highest information gain (IG). It needs to go through n_{samples} to calculate the IG and do that $n_{estimators}$ times. Total complexity is O($n_{features}n_{samples}n_{estimators}$)

Prediction time depends on the underlying weak learner. In this case we have a decision tree. For every weak classifier we have to look at the selected feature for every node down the tree. So that would be one node per depth level and in this case just one node, the root. We have N weak learners going down depth one, so it's a constant time for prediction, O(1).

The advantages of AdaBoost are[4]:

- As we add more and more weak learners, the train and test error go down. Test error would typically start going up after a given complexity (here complexity is the number of weak learners) but not in the case of boosting.
- Training weak learners is much faster than training strong learners.
- If the weak learners do slightly better than random, they become a powerhouse when combined.

The disadvantages of AdaBoost include:

- - Becomes a bit of a black box. Hard to interpret the results
- *if any non-zero fraction of the training data is mis-labeled, the boosting algorithm tries extremely hard to correctly classify these training examples, and fails to produce a model with accuracy better than 1/2.*[5]

[4] http://scikit-learn.org/stable/modules/svm.html#

[5] http://www.cs.columbia.edu/~rocco/Public/mlj9.pdf

Why this model:

- The training speed is relatively fast compared to strong learners such as neural networks. Since this is a requirement from the board it makes sense to explore the fast options first.
- Boosting is also quite simple to explain to a board.
- From previous experience boosting often gave among the best results
- Using Decision Tree as weak learner because most of the features are binary and calculating the information is a bit faster

	Training set size		
	100	200	300
Training time (secs)	0,112	0,107	0,095
Prediction time (secs)	0,008	0,099	0,006
F1 score for training set	0,9323	0,8858	0,8813
F1 score for test set	0,7647	0,7939	0,7852

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?

I would chose SVC with a training set size of 100 for the following reasons:

- Among the three train set sizes that we tried, F1 score stays approximately the same and is on par with Adaboost on the test set. Both these algorithm beat kNN on that score.

	F1 score for test set		
Training set size	SVC	kNN	AdaBoost
100	0,7746	0,7246	0,7647
200	0,7815	0,7692	0,7939
300	0,7838	0,7801	0,7852

- Training time is not as fast as kNN, which basically has no training time, but an order of magnitude faster that Adaboost.

	Training time (secs)		
Training set size	SVC	kNN	AdaBoost
100	0,0040	0,019	0,112
200	0,0030	0,001	0,107
300	0,0100	0,001	0,095

- Prediction time is fast as AdaBoost because it is not distance based like kNN.

	Prediction time (secs)		
Training set size	SVC	kNN	AdaBoost
100	0,001	0,007	0,008
200	0,003	0,005	0,099
300	0,009	0,01	0,006

So at 100 train set size, SVC has the best performance and is among the fastest to train. A small prediction time is also important as we will be running all the students through it. It is similar to a database where we say "write once read many times". Once train it also takes almost no memory has it only needs to use a few parameters.

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).

Our objective is to separate the students that are at risk of failing on one side and those who are safe on the other side. This is no simple task as many parameters are intervening in this classification. We are trying to find a line that will best make this separation between those at risk and those who are not with a margin as large as possible. Meaning that we want to find a line that sits as comfortably as possible between the two groups without being closer to one or the other.

Intuitively we thing about a straight line, but such a line might not allow us to make the best separation as the two classes might be mixed up a little bit. So we bring the problem to a higher dimension (from m dimension to n dimension where n > m). In this new space, a straight line (hyperplane) might exists where it is much simpler to make a clean separation. So classifying a new student we will bring him to this higher dimension and see on what side of the line it falls.

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.

```
GridSearchCV(cv=None, error_score='raise',
       estimator=SVC(C=1.0, cache_size=200, class_weight=None,
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),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'kernel': ('linear', 'rbf', 'sigmoid'), 'C':
[0.1, 1, 10, 100], 'gamma': [0.001, 0.1, 10, 100]},
       pre_dispatch='2*n_jobs', refit=True,
       scoring=make_scorer(f1_score, pos_label=yes), verbose=0)
```

What is the model's final F1 score?

```
F1 score: 0.783783783784
Best params: {'kernel': 'rbf', 'C': 10, 'gamma': 0.001}
```