Udacity Machine Learning Nanodegree Project 2: Building an Intervention System

Task

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 behaviors 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.

Code

My code is available at https://github.com/Edderic/udacity-machine-learning-nanodegree/tree/master/p2-student-intervention

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?

This seems to be a classification question, where our goal is to predict whether a student is going to be successful or not. In other words, given a student, we would like to assign the classes “successful” and “not successful” to that student, so that teachers and other people could administer early interventions. This is not a regression problem because we are not trying to predict a continuous output, given a student. It could have been a regression problem if we are trying to predict the final (continuous) score of the student, but instead, we only know whether a student failed or passed.

2. Exploring the Data

In the dataset, there are 395 total students. There are 31 total features, including the target label $passed$. Out of those 395 students, 265 passed and 130 failed. Graduation rate, then, hovered around 67%.

We have several issues with this dataset. Namely, they are 1) the dataset having lots more of one target class than the other, and 2) we have so many features but so relative to the number of training instances, which might make our models more likely to suffer overfitting.

First of all, the labels are imbalanced. Instead of having similar amounts of students who passed and students who failed, we have a bit more of the latter than the former. Assuming that the test set has the same distribution of $passed$ as the training set (e.g. we have much more students who are likely to pass than students who are unlikely), and assuming that we are using the accuracy metric, we will probably score “well” just by predicting each point in the test set as passing. However, we cannot assume that the distribution of incoming students is going to be the same. It is possible that next year’s batch of students really do have a lot more students that might need intervention, for some reason. In such situations, a model that is “trigger happy” in labeling students as “likely to pass” means that our model is most likely to perform poorly. Thus we should use the $F1$ score, which takes into account the precision and recall scores – they factor false positives and false

negatives into the equation, and further give us insight as to how our machine learning model is performing. For the students that it is labelling incorrectly, is it more likely to claim that students who have passed the exam were going to fail the exam (i.e. false negative)? Or is it the other way around – does it think that students who have failed the exam were successful (i.e. false positive)? $F1$ score in this case will give us more nuanced insights than the accuracy metric.

A reason to worry about the labels being unbalanced is that some models are likely to perform poorly. Support Vector Machines (SVMs) are one example. See the following: http://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane_unbalanced.html Another consideration that is highlighted by unbalanced data is how we split the data into training and test sets. Simply splitting the data without regards to the balance of target label could yield wildly different results. For example, imagine that a simple training and test split coming from `train_test_split` function might give us a training set that consists of students who have passed the final exam, and the testing set only consisted of students that didn't. Performance would probably be poor, given that the training and testing set distributions could not be any more different. In these cases, it is advised that we use cross-validation strategies that take unbalanced data into account. http://scikit-

learn.org/stable/modules/generated/sklearn.cross_validation.StratifiedShuffleSplit.html http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.StratifiedKFold.html In this project, I used Stratified Shuffle Split, which makes sure that the training and testing sets have about the equal ratio of passed vs. failed students. It also shuffles to remove the bias that ordering of students might have in the dataset.

Second of all, overfitting might be a strong possibility. The number of training instances needed to accurately classify or predict a target label correctly increases exponentially as we increase the number of features. This is also known as the Curse of Dimensionality. See https://en.wikipedia.org/wiki/Curse_of_dimensionality We only have a relatively small number of training instances to consider. More data is generally better than less data, and we don't have that advantage here.

4. Training and Evaluating Models

Given the characteristics of the data (i.e. small number of training instances compared to the number of features which might lead to models easily overfitting, and having imbalanced target labels) and requirements of the school (i.e. we want a classifier that is correct as much as possible and takes the least amount of space and time), the three models I chose were Random Forest, Ada Boost, and Multinomial Naïve Bayes. I decided to run each

algorithm ten times instead of just once, in the process displaying the averaged training time, prediction time, and $F1$ scores. I thought that it would be a good idea because it is possible that the models during training and testing might return different timing values due to garbage collection (or some other system process) in Python. More importantly, running the algorithms several times with different training and testing splits would address the issue of having small number of training instances, and might improve performance of the models overall.

```
RandomForestClassifier(bootstrap=True, class_weight=None,
criterion='gini', max_depth=None, max_features='auto',
max_leaf_nodes=None, min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=1, oob_score=False,
random_state=None, verbose=0, warm_start=False)
```

Training Set Size

	100	*200*	*300*
Training time (secs)	0.012380	0.014373	0.016358
Prediction time (secs)	0.001203	0.001225	0.001284
F1 Score for training set	0.999259	1.000000	0.999504
F1 Score for test set	0.759810	0.784500	0.766378

To address overfitting, the first model I picked was Random Forest. Random

Forest divides the training set into smaller ones, and trains a bunch of decision trees on those smaller datasets, then makes each decision tree have a vote. The predicted class that holds the most vote becomes the final vote of the ensemble classifier. I chose that instead of just having one decision tree because a decision tree tends to overfit easily. When splitting the data, decision trees, in their quest to max out information gain (or gini) might split data based on a feature even if the relationship between a feature and the target label is really spurious. Using Random Forests, we enforce much shallower trees and hence prevent overfitting. A slight con of Random Forests is that it is a bit more complex under the hood than just a decision tree, so it might be harder to explain to someone how the ensemble is reaching a certain conclusion, especially to non-technical business people. Another con is that it seems slower than some models like Naïve Bayes on both the training and prediction parts.

```
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,

learning_rate=1.0, n_estimators=3, random_state=None)
```

Training Set Size

	100	*200*	*300*
Training time (secs)	0.001782	0.002021	0.002200
Prediction time (secs)	0.000294	0.000300	0.000297
F1 Score for training set	0.836797	0.821001	0.818734
F1 Score for test set	0.787321	0.798216	0.808246

F1 Score for test set 0.787321 0.798216 0.806240

To address imbalanced data, I thought of using AdaBoost. One fear I had was that since there were a lot more students passing than failing in the dataset, the models might be biased to classify students as “passing.” Relatively speaking, we might not have enough examples to generally tell when a student is going to fail. When training, our models might just see that most of the examples are of students that don’t need an intervention. They might not have learned from enough examples to describe a student who might need an intervention, so the models might be biased to just classify students generally as passing. I thought that AdaBoost might solve this issue when training by focusing on the examples that it got wrong. Through several iterations on the training set and a weighing process, it finds weak learners that perform better and better on the previously incorrect predictions. One con with this, similar to Random Forest, is that it is more complicated, and might seem more like a black box compared to some other models. Again, another con is that it might be slower than some other models like Naïve Bayes.

```
MultinomialNB(alpha=10, class_prior=None, fit_prior=True)
```

Training Set Size

	100	*200*	*300*
Training time (secs)	0.000610	0.000668	0.000726

Training time (secs)	0.000610	0.000668	0.000736
Prediction time (secs)	0.000106	0.000099	0.000101
F1 Score for training set	0.820898	0.800323	0.802214
F1 Score for test set	0.810732	0.805823	0.789492

The third model I used was Naïve Bayes. It is relatively quick to train and gives predictions very fast, so it addresses the need to predict things quickly so that the school could save money. The issue with Naïve Bayes is that it might be too simplistic. It's assuming that each feature is independent of other features, which might not be the case. It's possible that it might underfit our dataset. However, given the possibilty of overfitting due to us having plenty of features, I thought Naïve Bayes might be able to balance out the overfitting possibility, and in this case, it looks like it works.

5. Choosing the Best Model

The best performing model out of these three in terms of $F1$ score is AdaBoost. However, it is only marginally better than the other two. The best overall performing model, however, is Naïve Bayes. It is significantly faster than both of the other models I've tried. It is ≈ 3 times faster than AdaBoost (with 3 estimators) on training and prediction, and ≈ 23 times faster than RandomForest with 30 estimators on training and ≈ 12 times on prediction.

In layman's terms, Naïve Bayes, in our case, basically tries to calculate a

student's probability of succeeding on the final test given what we know about other students' past behavior, upbringing, and environment. It makes use of basic probability rules (i.e. Bayes' Rule) to infer a student's successfulness, given data about other students. It is also called Naïve because it makes some assumptions that are "simple", relative to what we know of the causal relationships in the world we are operating. More precisely, it assumes that qualities of a student are entirely independent (e.g. a student being really poor and a student having internet access at home are assumed to not be related at all, even though they might really be strongly correlated or even be causally related). A lot of times, this reductionist, simplistic view actually (and surprisingly) works well in practice when trying to predict an outcome.

The way I calculated the final $F1$ score of the Naïve Bayes classifier is by running it with `GridSearchCV` a hundred times and averaging the $F1$ score. I chose to do this because the `StratifyShuffleSplit` cross-validation approach to split the dataset into training and test sets has a randomization (shuffling) component before it gets fed into `GridSearchCV` (which uses cross-validation internally on the training set I supplied). The alphas I chose are as follows:

```
[0.01, 0.05 ,0.1, 1, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60]
```

Here are the results:

Alphas

count	100.000000
mean	34.100000
std	18.550777
min	5.000000
25%	15.000000
50%	27.500000
75%	50.000000
max	60.000000

The best alpha seems to hover in the range of 34.1 ± 18.6.

F1 Scores

count	100.000000
mean	0.799192
std	0.027628
min	0.701754
25%	0.787879
50%	0.805970
75%	0.812500
max	0.885246

The $F1$ score seems to hover around $79.9\% \pm 2.8\%$.