Project 2: Student Intervention System

Udacity: Machine Learning Nanodegree
Hamilton Hoxie Ackerman
hoxiea@gmail.com
1/6/2016

Introduction

A major theme in contemporary education is **student success**, where a successful student is typically defined as one who manages to graduate, perhaps within a certain amount of time. In order to promote student success, i.e. increase the proportion of students successfully graduating, schools these days are quite interested in identifying students who seem to be on a failing trajectory, and intervening to help improve that trajectory.

This process of identifying students on failing trajectories can easily be posed as a machine learning problem: collect pass/fail outcomes from as many students as possible, and collect as much additional data as possible about the students; it's not hard to think of a variety of predictors that could be related to passing or failing a final exam. Then the modeling process can find structure in the data, and the resulting model can predict whether an unseen student is more likely to pass or fail, given values for the predictor variables. (Those students predicted to fail by the model will presumably be the subject of academic interventions.) When posed like this, we have a **classification** problem on our hands: our model will output a pass/fail category label, not a number.

In some classification problems (machine learning problems in general, really), model performance is king: if a more computationally expensive model would do a better job capturing the structure and generalizing to new inputs, then pay the computational costs and enjoy the improved performance. In our case, we're trying to find a model that not only performs well but also that minimizes its computational footprint and that doesn't require massive amounts of data for training.

Exploring the Data

The dataset we have to work with contains information about 395 students at the high school level. We know the pass/fail outcomes of all students: 265 passed their final exam and 130 failed their final exam, for a graduation rate of 67.09%. Because we have both features and corresponding labels, and we're interested in using this data to build a model that will produce labels for unseen observations, this is **supervised learning**.

For each of these 395 students, we have 30 features (plus the pass/fail outcome). These features span an impressive range of aspects about the students' lives: home and family situation, time spent studying and traveling to school, health situation, relationship with the school they attend, etc. At first glance, it seems that many of these variables could be related to the target.

Of the 30 features, 13 are binary, 13 are numeric, and 4 are categorical. After turning the binary variables into 1/0 numeric variables, and after splitting the categorical variable categories into dummy variables, we have 48 numeric features to work with.

Training and Evaluating Models

Model 1: Decision Tree

My first instinct when faced with this problem was to use a decision tree (DT). Not only were they the first supervised learner discussed in the supporting lectures, but they also intuitively capture the idea of what we're trying to do here: make a decision about whether students are likely to pass or fail, based on information about their lives, habits, and routines. In fact, DTs are quite intuitive by machine learning standards, so if I'm going to have to explain my algorithm to the local school district officials, a DT might be ideal.

The main weakness of a DT is its tendency to overfit the data: without any constraint on the size/complexity of the tree, the tree-creation algorithm can and will create increasingly complex splits until all data is categorized correctly, even if every observation ends up in its own leaf (barring the case in which two inputs have exactly the same feature values but different target values, in which case the algorithm should give up on those instances).

I chose to start with a DT for the reasons outlined above: they're intuitive, they're relative fast to train and predict with, and it seemed like they would make a nice baseline for comparison once I had other, more complex models trained.

Even though it doesn't seem like this will give us a good representation of DT performance for this problem, no attempts to prevent overfitting were made during this initial exploratory phase [1]. As expected for such a highly variable learning algorithm, performance in the test set was perfect (F1 = 1.0 across the board), whereas testing performance was significantly lower, which is typical of overfitting.

Training Set Size	Training		Testing	
	Time	F1 Score	Time (Prediction)	F1 Score
100	5.992e-04	1.0	1.059e-04	0.6948
200	1.043e-03	1.0	1.042e-04	0.6965
300	1.486e-03	1.0	1.049e-04	0.7409

Table 1: Performance of Decision Trees using 100, 200, and all 300 available training samples. Each value is the average of ten runs; each run used a new random sample of training points.

Model 2: Random Forest

Emerging from the Supervised Learning lecture videos, I was intrigued by the instructors' claims that ensemble learners are somehow less prone to overfitting than non-ensemble learners. One thing not discussed was how to always find a weak learner for each boosting iteration. But it occurred to me that, given the above-50% performance of the DTs constructed in Model 1 and the relatively small computational costs of building decision trees, that single decision trees could be excellent weak learners in an ensemble setting.

Alas, I'm not the first person to have thought of this. It turns out that, with some slight modifications to the general boosting procedure, this idea begets an ensemble learner knows as a "random forest." (Why? Because a forest is just a bunch of trees.) The main modifications involved include [2]:

- Only using a subset of available training data for each tree (accomplished by sampling with replacement)
- Only using a random subset of available features (typically $\sqrt{p}$, where p is the number of features) when making splits.

These modifications and the averaging of results help to prevent overfitting (by making fewer features available), but the claim is that they decrease variance without sacrificing much bias. Based on Table 1, this could be exactly what we need.

The main weakness of random forests in our context is the computational costs involved: instead of training just one tree, we're now training a forest of trees. Predictions are also more expensive because the entire forest must be consulted. Still, given how small our training datasets are, and in the interest of preventing DT overfitting, I grew some random forests with the default of 10 trees each. The results are summarized in Table 2.

Training Set Size	Training		Testing	
	Time	F1 Score	Time	F1 Score
100	1.946e-02	0.9921	8.442e-04	.7107
200	2.003e-02	.9936	8.657e-04	.7226
300	2.081e-02	0.9925	8.768e-04	.7352

Table 2: Performance of Random Forests using 100, 200, and all 300 available training samples. Each value is the average of ten runs; each run used a new random sample of training points.

As expected, Training Time was about 10x what it was for single DTs, and Testing Time was uniformly 8x. Despite doing better with 100 observations than the DT did, the n=300 Testing F1 score for RFs was lower than the corresponding DT score. So *without any tuning*, RFs perform worse than DTs in almost every regard. (I suspect that unconstrained DTs actually aren't weak enough for us to benefit from them in an ensemble setting.)

Model(s) 3: kNN, LDA, QDA

Having explored decision trees from a couple of angles, I considered a few algorithms for my third model, trying to find one that wouldn't require much tuning out of the box. I also wanted a learner that was lower-variance than DTs and RFs.

k-Nearest Neighbors (kNN) was one option. This nonparametric machine learning algorithm compares incoming unseen observations with the *k* seen observations that are "closest," for some mathematically valid definition of "close." The default *k*=5 seemed reasonable, but as Wikipedia describes, kNN is dramatically hampered by the curse of dimensionality as the number of features increases. Wikipedia mentions "p>10" as the point at which the data is considered "high-dimensional"; with 48 features in our dataset and only ~300 observations with which we train, we certainly seem to qualify.

The default distance metric, Euclidean distance, will also be a poor choice in our case, because our data is unnormalized (and there's no way to normalize the held-out data that we don't have access to, so there's no point in training models on a normalized version of the data we *do* have to work with): the numeric variables that are on a scale from 0-4 or 1-4 (Medu, traveltime, etc.) will be considered more important to get right by the algorithm, even though these variables aren't necessarily more important for classification purposes.

I ran kNN anyway, and the Testing F1 scores were comparable to what was achieved by DTs and RFs (Table 3). Training Time was the best yet, probably because there's no actual training involved with kNN (aside from perhaps loading the data into a data structure optimized for distance queries). Testing Time was an order of magnitude slower than it was for DTs, which isn't unexpected, given that DTs have already been trained by Testing time, whereas the calculations for kNN don't really begin until you're making predictions. And behind the scenes, we have to keep all observations in memory for comparisons, instead of reducing the data to a simple model that can be applied to incoming new observations, so we'll require much more memory than DTs or even a forest of 10 DTs.

Training Set Size	Training		Testing	
	Time	F1 Score	Time	F1 Score
100	7.138e-04	0.8282	1.379e-03	0.7094
200	4.981e-04	0.8306	2.044e-03	0.7122
300	6.481e-04	0.8433	2.810e-03	0.7246

Table 3: Performance of k-Nearest Neighbors using 100, 200, and all 300 available training samples. Each value is the average of ten runs; each run used a new random sample of training points.

But given that nothing so far had significantly outperformed the simplest model I tried (DTs), I wanted to try something simple, ideally one that didn't have any hyperparameters at all. In that vein, I turned to **Linear Discriminant Analysis (LDA)**.

As the `sklearn` documentation describes, "[LDA and similar method Quadratic Discriminant Analysis (QDA)] are attractive because they have closed-form solutions that can be easily computed, are inherently multiclass, have proven to work well in practice and have no hyperparameters to tune" [3]. Given the computational constraints of our student success problem, LDA could make a great choice. (In hindsight, LDA would make a much better "baseline model" than DTs.)

LDA and QDA are some of the simplest "decision boundary" algorithms out there. In both cases, the data distribution for each class (i.e. Pr(X|y)) is modeled as a multivariate Gaussian. This is the multi-dimensional analog of the Normal distribution, and has two parameters: mean and covariate matrix. The sample mean for each class is used in both LDA and QDA. The difference between the two methods arises in the estimation of the covariance matrix: in LDA, the same covariance matrix is estimated from all observations and used for all classes; in QDA, each class has its own covariance matrix, estimated from the elements belonging to that class, yielding quadratic decision boundaries instead of linear decision boundaries.

The major drawback of LDA is that, in some cases, it's not variable enough to capture the structure in the data. (If you require a more complex decision boundary, it's time for QDA or a Support Vector Machine.) In addition, the assumption that observations are multivariate Gaussian is often violated in practice, and yet the methods often still work reasonably well in practice.

"Works well in practice" was certainly true in our case (Table 4):

- Training Time is comparable to DTs, the fastest algorithm so far that actually trains
- Prediction Time is the fastest we've seen yet
- Testing F1 scores are comparable to what was achieved by all previous models; the 0.7353 seen when n=300 is the best performance yet, and the 0.6999 for n = 100 is slightly better than what DTs did with n=100

Training Set Size	Training		Testing	
	Time	F1 Score	Time	F1 Score
100	1.505e-03	0.889	1.058e-04	0.6999
200	1.849e-03	0.8726	9.792e-05	0.7329
300	2.173e-03	0.8514	9.706e-05	0.7353

Table 4: Performance of Linear Discriminant Analysis (LDA) using 100, 200, and all 300 available training samples. Each value is the average of ten runs; each run used a new random sample of training points.

Final Model: Tuned Decision Tree

Of the four models analyzed above, I was most impressed by Linear Discriminant Analysis: fast Training and Prediction times, minimal memory footprint, very explainable algorithm, and totally respectable accuracy (including that nice n=200 performance). But in terms of tunability, LDA leaves a lot to be desired. There are a few different solvers available, but exploring them didn't yield anything particularly interesting; at the end of the day, we're essentially coming up with the optimal linear decision boundary between two classes, and there are only so many ways to achieve that.

So instead, I fell back to the Decision Tree: also fast and interpretable, and in contrast with LDA, there are a variety of hyperparameters that can be tuned to improve DT performance, mostly to avoid the overfitting exhibited in Table 1. A constrained tree with improved Testing F1 Score (and likely an even better prediction time, due to the tree being smaller) could be a great model in the context of this problem.

I performed a 5-fold cross-validated gridsearch across the 4-dimensional parameter space consisting of:

- the decision criterion for making a split (gini, entropy)
- the maximum allowed depth of the tree (1..20)
- the minimum number of samples required at a node to warrant a split (1..10)
- the minimum number of samples required at a leaf node (1..10)

I repeated this gridsearch on two sets of data:

1. The 300 observations used by all other models for training, with the 95 validation observations set aside, mostly so that I could compare the Testing F1 score to the scores achieved previously.
2. All 395 observations, giving the decision tree as much information as possible but without the ability to validate performance on unseen observations.

In both cases, the winning hyperparameter combination was laughably simple: the gini split criterion and a maximum tree depth of 1. (With a maximum depth of 1, the minimum samples for splits and leafs are essentially irrelevant.) For n=300, the Training F1 was 0.82423, the Testing F1 score was 0.7714, and the Prediction Time was 2.64e-04. For the full n=395 search, the 5-fold cross-validation F1 score was **0.8110**.

A tree with a maximum depth of 1 has literally one variable and a single split on that variable; it's the simplest nontrivial decision tree possible. Remarkably, that was apparently the optimal strategy for predicting pass/fail in unseen observations.

What was the one variable needed? It was the same for both n=300 and n=395: *failures*, which captures the number of past class failures. The README file for the project tells us that *failures* captures "the number of past class failures (numeric: n if 1 <= n < 3, else 4)." This suggests that possible values are (1, 2, 4), though in the actual data, the values present are (0, 1, 2, 3), which makes more sense as a count of past failures. And in both cases, the optimal split was exactly the same: predict *Pass* if failures = 0, otherwise predict *Fail* (Figure 1).

And so, ladies and gentlemen of the board of supervisors, that's my final recommendation: focus your academic intervention efforts on those students who have a history of failing the final exam at least once. This model can be implemented in a spreadsheet, database, or in your head with an exceptionally small computational footprint, and according to my investigations, will generalize well to new students you're trying to classify.

Bibliography

[1]: Ackerman, Hamilton. "Why Not Gridsearch When Training and Evaluating Models?"Udacity | Free Online Courses & Nanodegree Programs - Udacity. Web. 7 Jan. 2016. <https://discussions.udacity.com/t/why-not-gridsearch-when-training-and-evaluating-models/42985>

[2]: "1.11. Ensemble Methods." 1.11. Ensemble Methods — Scikit-learn 0.17 Documentation. Web. 2 Jan. 2016. <http://scikit-learn.org/stable/modules/ensemble.html#forest>

[3]: "1.2. Linear and Quadratic Discriminant Analysis¶." 1.2. Linear and Quadratic Discriminant Analysis — Scikit-learn 0.17 Documentation. Web. 4 Jan. 2016. <http://scikit-learn.org/stable/modules/lda_qda.html>

Supplementary Material

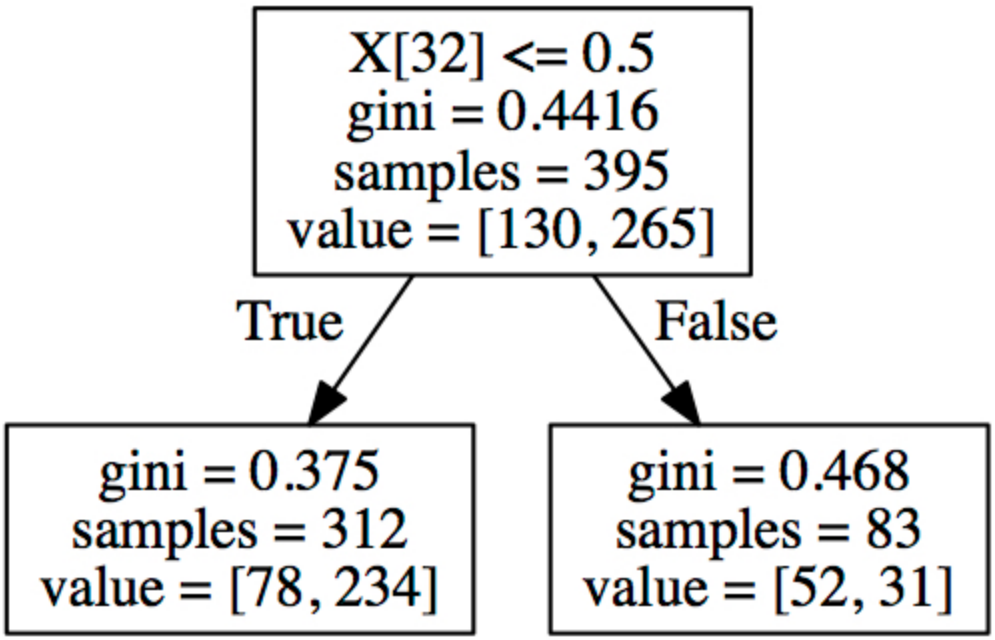

Figure 1: My final model for this problem: a tuned decision tree. The model predicts "pass" if the student has 0 past failures, and otherwise predicts "fail." This model achieves a 5-fold cross-validated F1 score of 0.8110 on all available training data.