Applying Supervised Learning Methods to Addiction Medicine Data

Brian Ambielli
Graduate Student, Georgia Tech
https://github.gatech.edu/bambielli3/ML-1

ABSTRACT
Supervised Learning methods, when applied to pre-labelled data sets, provide means for labelling new instances of data in the domain via classification (for discrete outputs) or regression (for continuous outputs). I applied 5 supervised learning methods to data sets provided by Chicago's Above and Beyond (AnB) Family Resource Center, with the goal of classifying patients in to the categories "Will complete programming" and "will not complete programming". All patient data has been anonymized to prevent exposure of personally identifiable information. Through the exploration of the AnB data, I gained insight into the performance of the 5 chosen supervised learning methods in relation to one another: SVM and boosting with decision tree learners were tied for accuracy, while the SVM model took around 1/10 the amount of time to train when compared to boosting, making it superior to boosting in that regard. The decision tree model was close to SVM in accuracy, but also offered results that were easier to interpret, making it the best candidate for the AnB team to discover what factors influence graduation of their programming.

KEYWORDS
Learning and Training; Health;

INTRODUCTION

By day, when I'm not studying for OMSCS classes, I'm a software engineer for a late-stage start-up in Chicago called Uptake. It was through my affiliation with Uptake that a few co-workers and I were approached by a Northwestern University professor with an interesting data classification problem he was hoping we could help solve. This professor is affiliated with the Above and Beyond (AnB) Family Resource Center here in Chicago, which provides high-touch addiction treatment and family counselling for socio-economically disadvantaged members of our Chicago community. AnB has worked with close to 2000 patients, but due to the demanding nature of the program and the often risky behaviors exhibited by the patient population it works with, it sees less than 20% of patients graduate from the program. Our professor was interested in increasing that graduation percentage, by identifying attributes of successful patients as well as factors that contribute to dropping out of the program early.

The data I will be analyzing for this project is patient intake data, which provides a snapshot of a patient's background before they are admitted to the program. The AnB admissions team uses this intake data to determine if a patient has the potential to succeed and graduate from the program. This is, at its core, a classification problem: given a set patient instances and intake attributes for each, select a training set of patient data and develop a supervised learning model that maps new patient instances to the discrete labels "will graduate program" or "will not graduate program". The randomly selected testing set will confirm whether the trained classifier is able to classify new instances to the correct label, based on how they answered their intake questions. A model like this has the potential to impact how AnB weights certain answers to intake questions during admission decisions. It also has the potential to affect some of those "on the fence" candidate applications to push them over the edge towards acceptance.

There are 1953 instances in the intake data, and around 20 attributes. Many attributes are categorical in nature, some categories with upwards of 500 distinct values. After counting there is upwards of 7919 possible values across all attributes. Many of the attributes are also sparse: for example, not all patients have a listed diagnosis, or have their employment status provided. I'm planning on running my supervised learning algorithms on two views of the intake data: one where I leave the intake data as-is, and a second view where I significantly reduce the dimensionality of the data down through a cleaning and preprocessing procedure described in the methods below.

In summary, the data chosen for this paper is interesting because of the real-world implications of the results. From a Machine Learning fundamentals perspective, I hope that the difference in dimensionality between the two views of the intake data will provide interesting contrasts when results from models are analyzed side by side. The data is also interesting in that it relatively raw in nature: for example, many category values were collected through "open answer" text fields, so category attributes do not always have consistent values… this will likely require quite a bit of data cleanup to get results that will be valuable. While cleaning data is a tedious task, I feel like this data set better represents a lot of real world data that I will examine beyond OMSCS, so it is good practice in that sense. There is certainly a balance when cleaning between "too much" and "just enough", and I hope my dimensionality reducing procedure will highlight this.

METHODS

I chose to analyze the AnB intake data using 5 different supervised learning algorithms: Decision Trees, Neural Networks, k-nearest-neighbors (KNN), decision tree boosting, and Support Vector Machines (SVM). Supervised learning describes a group of machine learning algorithms that accept pre-classified data as input, and build models that reflect the underlying function that represents the data. Supervised learning models allow us to classify new, unlabeled instances of data to the correct classifications. This type of learning is a good fit for the AnB data, since data is pre-classified based on whether a patient graduated or failed to graduate the program.

I chose to use the scikit-learn[1] python library for these analyses, which contains implementations for all the supervised learning algorithms I will be exploring in this assignment.

DERIVING ATTRIBUTES TO REDUCE DIMENSIONALITY

As noted in the introduction, I'm planning on exploring two views of the same dataset: one that only has class values derived but otherwise full dimensionality from the original data set (Minimal-Derived.csv), and the other with reduced dimensionality after the cleaning / preprocessing procedure (Derived-Final.csv). Data with high dimensionality can prove challenging for supervised learning classification methods (like KNN), where values that should be classified together are drawn apart by the high dimension of the data. This well-known problem is often referred to as the "curse of dimensionality". When category encoding methods like "One Hot Encoding" are added as a preprocessing step, high dimension data can quickly become very sparse, and sparse data becomes even more challenging to perform high quality classifications on without large amounts of training data. For reference, I counted the number of dimensions in the initial "uncleaned" data set: there were 7919 different values across all 20 attributes. This uncleaned data is in the `/data/Minimal-Derived.csv` file in the repository. Whenever I reference `Minimal-Derived` throughout this paper, I am referring to the uncleaned data set.

The reduced dimensionality of the second data view was achieved by transforming attributes of the initial data set with a cleaning procedure and a pre-processing procedure. In the case of the cleaning procedure, there were many categorical attributes in the data that contained free text entry fields when instances were collected. This led to cases where values in a category meant the same thing, but were represented differently in text. For example, in the `race` column, the race "white" was represented both as "white", and "white/Caucasian". I went through all category values that had this problem and normalized them to single values: for example, in the case above, I normalized both "white" and "white/Caucasian" to the same value "White".

The pre-processing procedure for the second data view involved the derivation of a few new attributes from pre-existing attributes that had high dimensionality. I changed the continuous ranges reflected in the "length of stay" and "age" attributes to discrete bins representing ranges of age or ranges of "length of stay". For example, someone listed as 20 years old was placed in a `20-29 age` bin. From the "referrer" column, I noticed that many referrers to AnB were parole officers, judges, or correctional facilities in Chicago. I derived a boolean attribute 'isReferrerCorrectional', to represent whether the patient was brought to AnB by someone in the correctional system. Similarly, I derived the "isEmployed" column from the "Occupation" attribute, where isEmployed reflects if the patient was employed at the time of intake. I chose to derive 7 different binary attributes from the "Diagnosis" and "Diagnosis Code" categories, one for each of the 7 most common diagnoses that were present for patients in the sample: Alcohol, Marijuana, Cocaine, Heroine, Opioid, Meth, and Tobacco. The original categories were very noisy, and it was possible for an instance to have more than 1 diagnosis. For this reason, I thought splitting out the category in to 7 binary (True/False) columns made the most sense, to try to draw some sort of value out of the noise. Finally, the class category "Completed Program" was derived from the "Discharge Type" column, where any value that indicated "completed-ness" was counted as "true" and all others false (including blanks). This work to reduce dimensionality in my data resulted in 118 values across 17 attributes, and one category for the classes to be used to classify instances (called "completed program"). The cleaned version of the data after going through the above procedures is stored in `/data/Derived-Final.csv` in my repository. Whenever I reference `Derived-Final` throughout this paper, I am referring to the cleaned data with derived values.

Besides deriving the class attribute from the original column "Discharge Type", I didn't perform any additional cleaning for the first data view. I'm hoping that the cleaning / preprocessing that the steps above accomplished will provide more interesting contrasts between data views when I compare them in the context of different supervised learning models.

ADDITIONAL PREPROCESSING

Since most the attributes in my data set were categorical with string values, I needed a way to convert these values from their string representation to a numerical representation that the sklearn supervised learning models could work with. Fortunately, sklearn offers a preprocessor called `LabelEncoder`[10] that does just that: it converts the string values of a category attribute in to numerical values.

Just using the LabelEncoder alone, though, has an unintended consequence: learning algorithms treat the ordinality of category values with significance when making decisions about the quality of the attribute. For example, an instance with a value of 10 must be somehow "better" than a value of 5 since 10 is greater than 5. This unintended numerical ordering of categories imparts an inaccurate representation of values in the category to the algorithms, since one value isn't necessarily any "better" than the other: the numbers really just indicate the order in which the LabelEncoder encountered them while encoding the column.

To get around this, I used another preprocessing step called `OneHotEncoding`[11] (also offered by sklearn) that combats this unintended ordering by converting a single attribute `A` with `n` values in to `n` Boolean attributes each representing the presence or non-presence of a value in that category. For example: my single category `gender` which has 2 values 'male' and 'female' would be One Hot Encoded in to 2 boolean attributes 'Male' and 'Female'. If an instance was male, they would have a 1 in the `isMale` attribute, and a 0 in the `isFemale` attribute. OneHotEncoding mitigates the ordinality created by the LabelEncoder, but it does cause an increase of dimensionality and sparseness in the dataset since each value in the original attribute gets its own Boolean attribute assigned.

Additionally, sklearn requires instances in a sample to not have any blank values in their attributes: values for an attribute can be sparse (i.e. a lot of 0 or false values) but not empty. I tried two different approaches to deal with blank values in my data set. The first approach involved replacing all blank values throughout every attribute in the data set with the string value "UNKNOWN". This wound up having an unintended consequence: after encoding my data using label encoding and one hot encoding as described above, the value "UNKNOWN" was being used as a new boolean attribute for use during classification. For very sparse categories, this caused "UNKNOWN" to dominate the category, which introduced more noise in my data than I had intended.

I adopted a different solution to fill in missing attribute values: a "best guess" algorithm that was inspired by our textbook section 3.7.4[2]. The algorithm examines values of the attribute for other instances in the dataset, and chooses the most likely value for the class of the current instance. For example: if an instance's race value was missing, and the instance did complete their AnB programming, the algorithm would aggregate race values for all other instances that completed the program and would set the current instance's race value to the highest value in the aggregate. An alternative implementation of this might have just taken the aggregate of the attribute values over all instances, and just set the value to the overall most frequent value. The reason I chose the first implementation over this one, was that my data was heavily weighted towards negative classifications (only about 20% of instances were positively classified). Since the data distribution is dominated by negative class instances, any sort of simple aggregates for missing attribute values would also be dominated by patterns found in negative class instances. By aggregating solely over instances that have the same class as the instance in question, I will be able to highlight any latent attribute value patterns for a class, which should help with identification of future instances.

The logic for the "best guess" algorithm, for reading data from .csv files, for parsing out instance classes from features, and for LabelEncoding/OneHotEncoding category features, can be found in the file `/Data.py`. I made sure that the Minimal-Derived and Derived-Final datasets were also both shuffled from their original ordering, to try to minimize any latent ordering bias in the data. To state it again: Minimal-Derived and Derived-Final are both pre-shuffled, so no additional shuffling was done during experimentation.

MODEL SELECTION

Before I present the results of the supervised learning models against each data set, let me first discuss how models were evaluated. There were often many parameters to choose from for each model, some of which are better on certain structures of data (high dimension vs low dimension for example) and others that allow the modeler to impart domain knowledge on the classifier. The level of domain knowledge I have in addiction medicine is low, so a reasonable starting place I chose was to create lists of model parameters from the sklearn documentation for each model, and try each combination together to see how performance changed. When a model had a parameter for number of iterations (neural networks, knn, boosting) I also varied the number of iterations used to train each model.

The ultimate performance of each parameter combination was decided by performing k-folds cross validation on the model candidate. Cross validation is a way to determine how well a model will generalize to data that was outside the scope of the training set, but still within the distribution of the underlying population. K-folds cross validation works by selecting one "fold" of data that is of size 1/k of the original data set, and holding it aside as a set of test values to be used to validate the accuracy of the trained model. The model is trained on the other k-1 "folds" of the data, and is tested on the fold selected before training. After accuracy is calculated, the fold is rotated so a different 1/k size sample is held aside as testing (no repeats), and the original testing fold is placed into the training pool for the next round of training. This rotation of folds occurs k times, so eventually every piece of data is held out of the training set exactly one time. Cross validation ensures that a particular ordering of the data doesn't just get 'lucky' and come out superior to any other ordering of the data during training. It also allows you to easily generate testing sets from your original data set, if producing more instances for testing is not an option (which is the case in my scenario).

Conveniently, sklearn offers a method called `sklearn.model_selection.cross_val_score`[12] which takes a classifier, data, and number of folds to be run during cross validation as inputs, and outputs an array of accuracy scores that correspond to how the model performed during each CV run. The mean of these scores is a reasonable metric for how well the model performed during training and testing, and how well the model can generalize to new data. Cross validation can slow down training time, since each model needs to be trained k times, but it ensures that there are no random biases latent in the ordering of the training data and ultimately provides a more accurate measurement of the ability of a classifier to generalize to new instances of data. Each parameter combination that was tested for each model was run through k=5 folds cross validation, and the parameter combination with the best cross validation average was chosen as the "best fit" model for the data set. The parameter combination + accuracy for each supervised learning model can be found as text files in the `{model_name}/final_graphs` directory in my repository.

I also collected the wall-clock duration for how long it took to compute k=5 cross validation of each model candidate, and included that as a data point in each text file. Collecting wall-clock time provided insight in to any "time-to-train" tradeoffs for certain parameter combinations. Since my data is relatively small (less than 2k instances) the wall-clock time did not exceed more than a few minutes for the most complicated of models, but it was still apparent which models appeared to scale poorly as size of data increased.

To determine whether more data would potentially influence the accuracy of a model, I also computed learning curves for each "best fit" model produced by model selection. Learning curves compare training accuracy to reported cross validation accuracy for a varying number of data instances, increasing the amount of data used to construct the models with each iteration. I used the `sklearn.model_selection.learning_curve`[13] method to compute curves. These learning curves were meant to show how variance, and model accuracy change with an increasing number of data instances. If the training and cross validation curves seem to plateau and narrow together as instances increase, then more instances will not do anything to increase the accuracy of the model. If training and cross validation curves are far apart, and have not yet plateaued, then more data points might be useful to increase the accuracy of the model. Each sample on the learning curve was computed with k=5 folds cross validation, which allowed me to obtain variance metrics as instance size increased. Ideally, we want a model that has high cross validation accuracy, and low variance, which would indicate that the model generalizes well to the underlying data distribution irrespective of the training set chosen. These learning curves can be found in the `/{model_name}/final_graphs` directory in my repository.

RESULTS

I will preface this section with one note about the data. 1634 instances (83%) in my data set were classified as "False", did not complete AnB programming. This implies that if you took 100 new instances, and randomly classified 83 of them as "False", that your simple model would perform reasonably well. Keeping this 83% number in mind is important when looking at the results from various models below: even if the training and validation errors indicate that a model performs well per the learning curves and data collected, the baseline for a naïve model that performs a random classification proportional to the distribution of the training data would perform with ~83% accuracy. Also of note: the data and graphs generated for the "Minimal-Derived" dataset will always be on the left in the figures below, and "Derived-Final" results will be on the right.

MODEL 1: DECISION TREE

The first model used to analyze the AnB intake data was a decision tree. A decision tree maps inputs to a discrete target function by classifying instances based on the values of their attributes. Attributes are used to split the population at their category boundaries. Attributes are chosen based on the amount of `information gain` an attribute provides towards the underlying classification problem. Information gain is a measure of the amount of information “encoded” in an attribute of your data set. We want our decision trees to be as compact and information rich as possible, to classify new data instances as quickly as possible. It is therefore better to organize a tree with the most information rich attributes near the top of the tree, so the most informative decisions are made first.

I sourced my decision tree implementation from `sklearn.tree.DecisionTreeClassifier`[3]. By default, it uses the `gini gain` of an attribute to determine the quality of splitting on the attribute. Gini gain is a measure of the likelihood of mis-classifying a new instance of data, if that instance were assigned a random class value[4]. Based on my research, it appears that gini gain and information gain using entropy are interchangeable when constructing decision trees, and neither results in a higher quality tree. I chose to stick with the default of the DecisionTreeClassifier class, which was to split on gini gain. Sklearn also uses the `CART`[5] algorithm instead of ID3 for building decision trees, and it is unfortunately not possible to choose a different algorithm. For pruning I decided to go with a pre-pruning method that limited the max_depth that the tree could reach. This was intended to combat overfitting of my data, by preventing the tree from exploring the full depth of attributes in the tree.

See the file /decision_tree/decision_tree.py for the code used to generate my results. Note that while it appears that re-running the code that trains decision trees will result in a different best depth between runs, the shape of the graphs below remains consistent. I varied the max depth of trees generated from 3 nodes to 100 nodes, and calculated learning curves as well as accuracy for each of the classifiers. “Minimal-Derived” represents the dataset that was not cleaned, and only has a derived attribute for the “graduated” class. “Derived-Final” represents the cleaned dataset with reduced dimensionality.

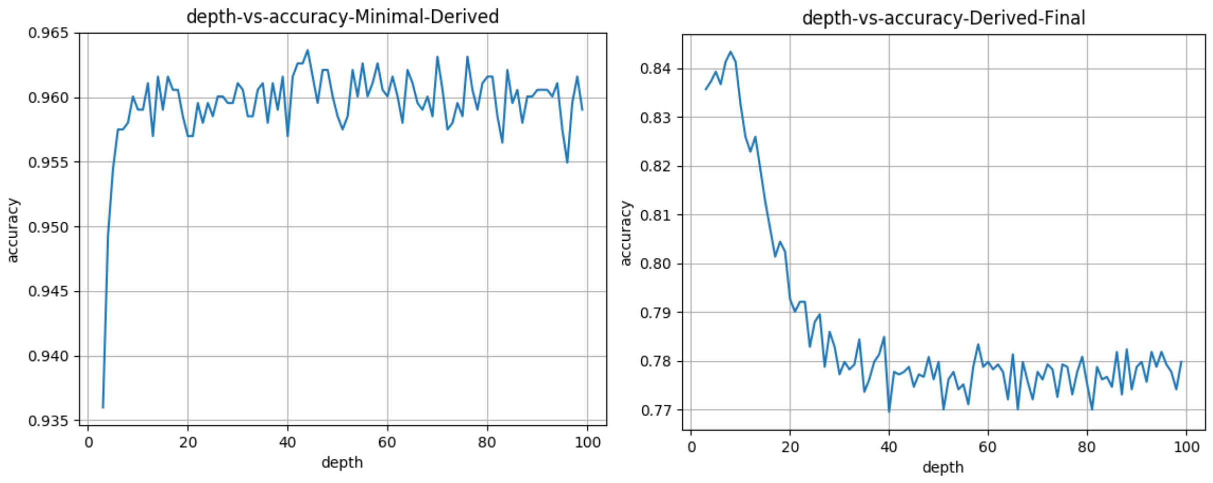


Figure 1: Decision Tree Depth vs testing accuracy of predictions after k=5 cross-validation

In the run that generated the graphs above, the Minimal-Derived dataset’s most accurate depth was 44, and the derived set’s most accurate depth was 8. Figure 1 shows how 5-fold cross-validation accuracy varied with the max_depth of a tree. The Derived-Final data appears to fall victim to overfitting at greater depths: as the max_depth of a tree increases, the cross-validation accuracy of the model falls off, and the accuracy is worse than then 83% baseline for a random classifier based on the distribution of the data overall. Compare this to the accuracy of the Minimal-Derived data, which appears to increase with depth and plateau around 96% accuracy. This tells me that during the cleaning procedure performed on the Derived-Final dataset got rid of some key indicators that are important for classifying those that will complete their programming. I will have to dig back in to find out what those indicators might be. It is also interesting that the accuracy increases so quickly at the beginning of the curve for Minimal Derived, starts to plateau around depth 10, which highlights the preference bias of the decision tree algorithms for putting the most information rich decisions at the top of the tree.

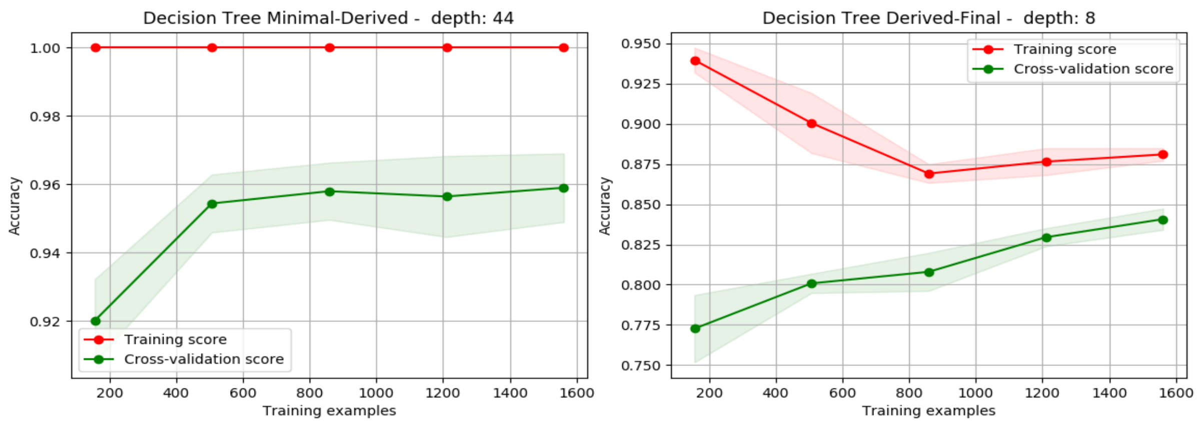


Figure 2: Decision Tree Learning curves for Most Accurate Model Depth

In figure 2, while it appears at surface level that the Minimal-Dervied dataset is falling victim to overfitting (training accuracy never drops below 1.00) the cross-validation accuracy plateaus at a consistent 96%. This was somewhat unexpected: if the model truly over fit the data, I would have expected the cross-validation error to be much lower. This indicates to me that the tree has truly picked up on something general to both the True and False classes, that is apparently missing in the Derived-Final dataset which narrows around 86% accuracy. The clear plateau of the Minimal Derived graph indicates that more training data would not necessarily provide any more information / accuracy gains than what we have already. In fact, it appears that the plateau starts around 800 samples, so after 800 there really isn't anything else to gain out of the data. In the case of the Derived Final learning curves, it appears that more data *could* increase the accuracy, since validation accuracy seems to be increasing with more and more data points. It still appears to have a plateau, though, around 86%, which again isn't much better than the 83% baseline for a random distribution-aware classifier. The bias in these cases appears to be fairly low as well, as the variance around the cross validation curves is within +/- 0.02 for the Minimal Derived set, and even lower for the Derived set. This means that the trained models did not vary much between cross validation iterations, which indicates that the models generated are good at generalizing to the underlying population. Low Bias and Low Variance are always the goal when training, and it appears that in the case of the Minimal Derived data set, we have a model that generalizes well to the underlying AnB population while still being highly accurate. A quick note on wall clock time: both datasets trained very quickly, under a quarter of a second to run k=5 cross validation at the specified depth. The Derived-Final set trained at under 1/10 of a second for all models, with time almost doubling for the Minimal-Derived case. This highlights another benefit of decision trees: accurate models are possible with very minimal training time.

See the files "/decision_tree/final_graphs/tree-viz-Derived-Final.pdf" and "/decision_tree/final_graphs/tree-viz-Minimal-Derived.pdf" to see visualizations of each tree. There is some noise in the data that needs to be filtered out from the Minimal-Derived dataset: for example, the time and date of intake and your specific birthdate are being split on in certain cases. It's also interesting to see what node is at the top of each tree: in the case of Minimal-Dervied, the top node which wound up being the most information rich was whether you had Herman Russel as your case manager. I wound up deleting the Case Manager column from the Derived-Final dataset as part of the cleaning procedure, since I thought it was too sparse and wouldn't provide value, which appears to have been a mistake.

MODEL 2: NEURAL NETWORKS

Neural networks can learn complex, non-linear models from feature-rich data. Its preference bias is for both "correct" and "concise" networks that compute the underlying relationship between inputs and outputs. They can be used for both classification and regression problems, and can be used to represent any conceivable relationship between inputs and outputs. Because of their complexity, neural networks can take a (relatively) long time to train when compared to other supervised learning approaches.

The neural network implementation I used was the `sklearn.neural_network.MLPClassifier`[6], which uses a multi-layer perceptron model plus backpropagation to construct a classifier. There are many parameters that sklearn exposes to tune the MLPClassifier, including hidden_layer_sizes, the activation function, the solver to use, and the max number of iterations to run before stopping the algorithm if it hasn't yet converged. In the file `/neural_network/neural_network.py` you can see that I ran an experiment to play with all of these parameters to determine which one performed the best for the AnB data, by first running 3 fold cross validation (for speed) on combinations of solvers and activation functions to find the combination that performed the best under conditions with a single hidden layer of 100 nodes (the default for MLPClassifier) and 1000 iterations to ensure each parameter combination converged. After determining the best combination of activation function and solver for my data, I trained 9 additional models using the best activation / solver pair with differently shaped hidden layers, and different numbers of iterations. I collected data on accuracy and wall-clock time for both the activation/solver discovery, and the layers/iterations discovery for each data set. I also created learning curves to examine the accuracy and variance of the chosen model over k=5 cross validation as number of data points increased. I'm going to spend time focusing on the Minimal-Derived data set for this analysis, since the Derived dataset did not produce results that were significantly better than a random classifier that was distribution aware (83%).

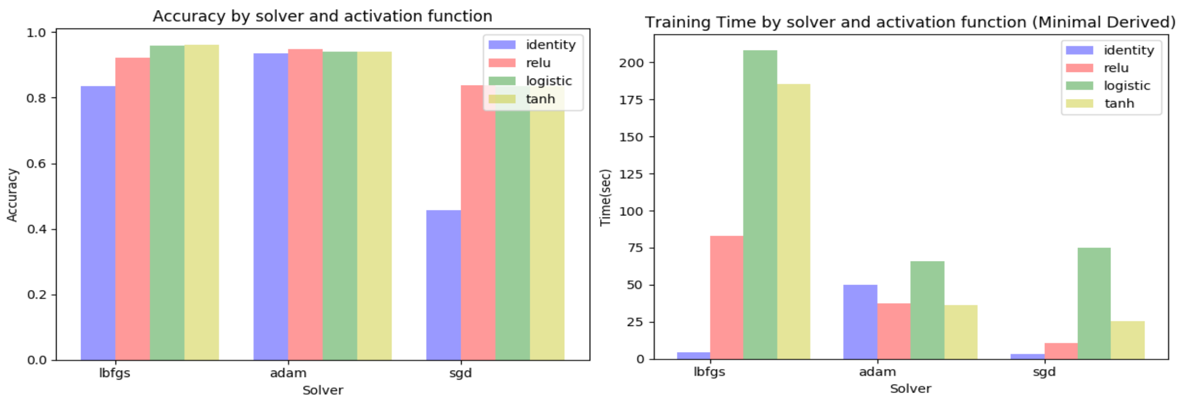


Figure 3: Minimal-Derived Neural Network Accuracy & Training Time by Solver & Activation Function

It is clear from the figures above that the right choice of solver and activation function is critical when choosing your data. Per sklearn's documentation, the `lbfgs` algorithm tends to perform better than `sgd` or `adam` when data size is small (on the order of a few thousands). The performance becomes a constraint, though, as the data set rises in complexity, which is apparent from the graph on the right of Figure 3. lbfgs performs slightly better than adam did in my experiments (about 0.01% more accurate in testing) but adam was able to complete training in close to ¾ less time under all scenarios except identity (which returned poor accuracy). If my data set was much larger, I would likely choose adam for the practical decrease in wall-clock time over lbfgs, but since training time was not the goal here (accuracy was) I went with lgfbs as my solver and tanh as my activation function for the next phase.

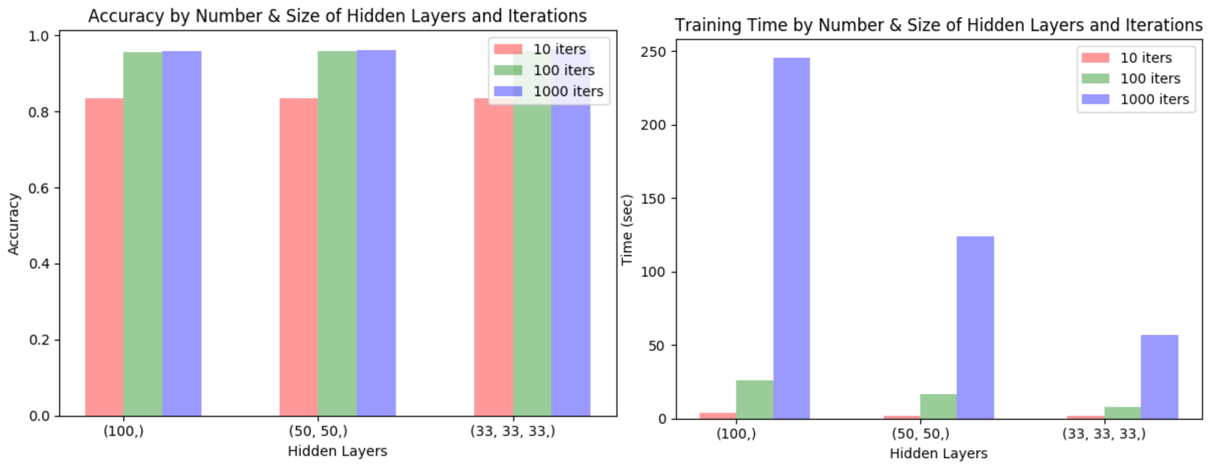


Figure 4: Minimal-Derived Neural Network Accuracy & Training Time by #of Hidden layers & Iterations

The data above makes it clear that with more iterations, your neural network will generally become more accurate. This make sense, based on the way backpropagation works: the more you can tune the weights of your network based on the desired output, the more accurately you will be able to model the underlying function (at the risk of overfitting, of course). Of the hidden layer / iteration combinations, the layer structure (33, 33, 33) which represents 3 layers of 33 nodes each, in combination with 1000 iterations wound up performing the best from an accuracy perspective. From a wall-clock time perspective, it makes sense that with more iterations time will increase proportionally. What I was not expecting was that as the number of nodes in my hidden layers became distributed throughout multiple layers, that the training time would decrease so drastically. My results show that distributing nodes throughout multiple hidden layers, instead of lumping them all together in one hidden layer, improves both accuracy of the underlying classifier as well as training time (almost ¾ decrease in wall clock time between (100) and (33, 33, 33)). The accuracy increase makes sense, since more hidden layers allows the network to better fit complex dimensions. The end parameter combination for the most accurate neural network for the Minimal-Derived data set was "solver: lbfgs, activation_function: tanh, hidden_layers: (33, 33, 33), iterations: 1000". Again, the Derived-Final dataset did not perform noticeably better than the 83% baseline, but the combination chosen for it was "solver: sgd, activation_function: tanh, hidden_layers: (100), iterations: 1000".

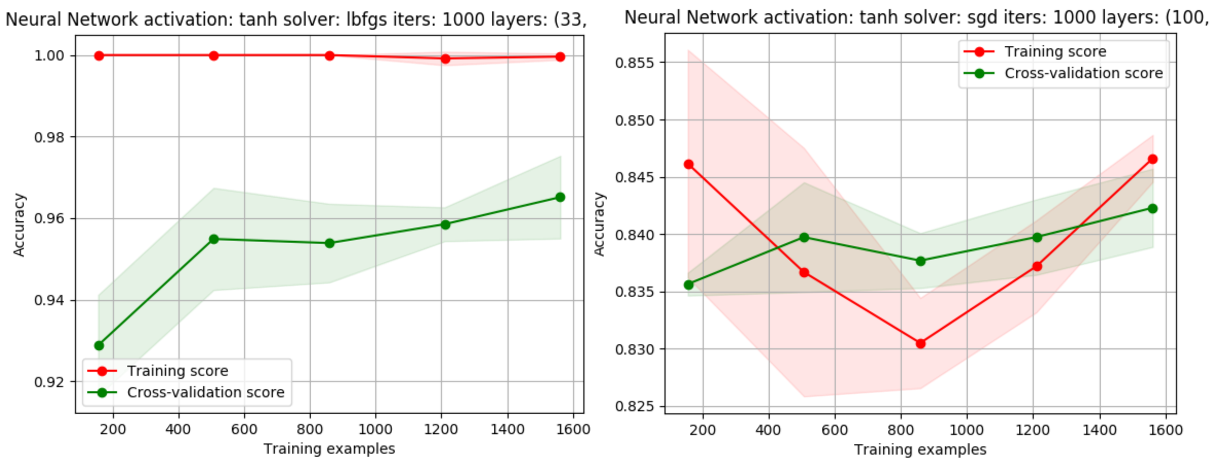


Figure 5: Neural Network Learning Curves for Most Accurate Model Parameters (Minimal-Derived on left)

Find graphs and data collected for this model in the /neural_network/final_graphs folder. Re-running the algorithms might result in slightly different graphs or results, but the general shapes should be consistent between runs.

The Minimal-Derived learning curve looks similar to the learning curve from the decision tree model, with training set staying at close to 1 on the accuracy scale, and cross validation rising and plateauing somewhat around 96% accuracy. The high accuracy during cross validation indicates that the neural network selected via the experimentation is a good fit to model the underlying population. The variance is also low at +/- 0.01, indicating that models trained with different chunks of the data set do not differ that dramatically in accuracy, which is a good sign for the capacity of the model to generalize to the population. It is unclear if the addition of more data to this model would increase accuracy, as the learning curve still appears to be rising as data increases. The Derived-Final dataset looks terrible in comparison: variance in both training score and validation score are enormous, and the training score actually dips below the validation score. This indicates that the model made some assumptions when constructing its underlying function that wound up being incorrect about the data … this was only rectified when more data points were added, as you can tell it jumps back up above the cross validation error around 1500 data points. The Derived-Final data were not interesting in the context of a neural network, perhaps because neural networks thrive on complex data and the Derived-Final dataset was greatly cleaned and had its dimensionality reduced. This led to a model that was basically no more accurate than a random classifier that is population aware. The Minimal-Derived dataset on the other hand wound up being slightly more accurate than the decision tree model, since neural networks are more apt at modeling subtle complexity in a high dimensional space than decision trees are. As a tradeoff, they also take much longer to train than decision trees: where decision trees could get close to the same amount of accuracy in under a quarter of a second of training time, neural networks took close to 3 minutes to train on the minimal derived set, with only modest accuracy gains over decision trees.

MODEL 3: K-NEAREST-NEIGHBORS (KNN)

KNN is an instance-based learning algorithm that has fast training times and longer query times. It is a "lazy learner", since training time basically involves throwing all instances in a database which doesn't take much time at all compared to constructing a full neural network that models new instances after the training set. The number selected for K indicates how many neighbors the algorithm should query to determine the class of the new instance. KNN can fall victim to the "curse of dimensionality", which is the phenomenon that as the number of dimensions grow in your data, the number of instances required to generalize appropriately increases exponentially. The function that you choose to measure distance between two features is critical when designing a KNN model, to ensure that your domain is modeled appropriately.

The KNN implementation I used was `sklearn.neighbors.KNeighborsClassifier`[7]. I decided to explore different built in strategies for weighting the influence of neighbors in the neighborhood of a query, as well as the algorithm used to compute the nearest neighbors and the number of neighbors to look for in a neighborhood before deciding on the class of a new instance. I left the rest of the parameters of the KNeighborsClassifier at their defaults. I collected wall-clock time for training each KNN model through 5-fold cross validation, collected reported accuracy data for each model parameter combination, and created learning curves for the most accurate classifier that showed how more data might affect the accuracy of the chosen classifier further. Unfortunately, due to the sparseness of the data that I was working with, even when I chose different algorithms other than brute force to calculate the nearest neighbors, brute force was used by default. This caused every single accuracy value for Minimal-Derived data set to be exactly the same irrespective of the weighting or distance metric used (see the data collected from training in knn/final_graphs). The only thing that had an influence on the accuracy of the algorithm was the number of neighbors used in the calculation, and even that didn't change the cross-validation accuracy much past the 83% baseline that we would get from a random classifier that is distribution aware. See files in `/knn/final_graphs/` for copies of the graphs and data collected.

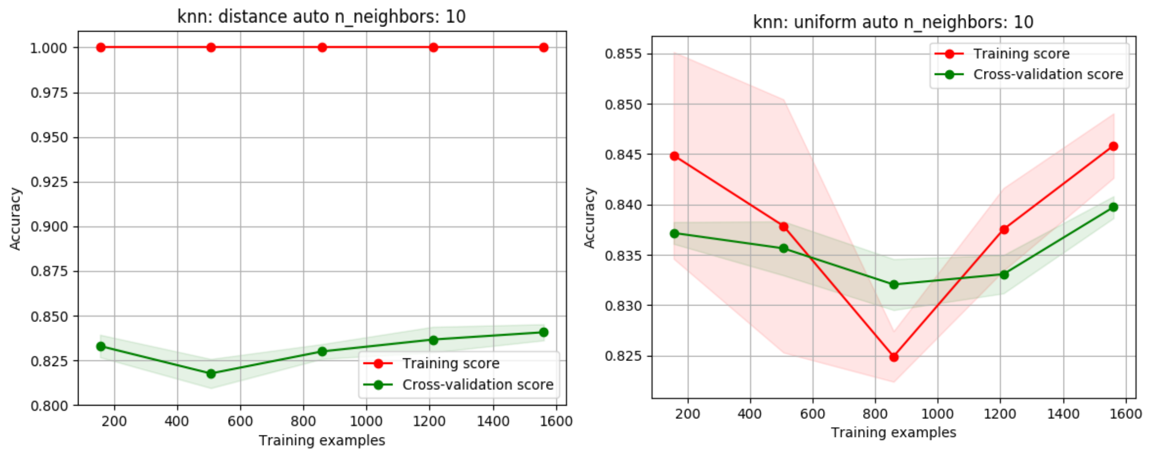


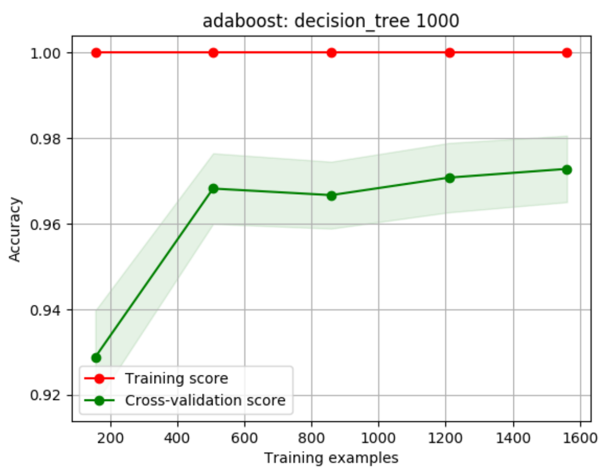
Figure 6: KNN Learning Curves for Most Accurate model parameters (Minimal-Derived on left)

Knowing that the underlying distribution of False classes in my dataset is 83%, when I see the essentially flat line at around 83% in the Minimal-Derived learning curve, it is clear to me that every instance with a "True" class in the test set is being misclassified. The training set has resulted in a completely overfit model (training set with flat accuracy of 1) that does not generalize to the underlying population. The curse of dimensionality is at play here: with close to 8000 different boolean attributes in the minimal-derived data set, and sparseness of data through those attributes, I just don't have enough instances for KNN to generalize well. Similar results are seen in the reduced dimensionality of the Derived-Final data set: cross validation accuracy does not get much above 83%. Training accuracy also averages around 84%, which means that the model hasn't over fit the data like in the Minimal-Derived, but also means that there just doesn't exist enough data to fill the dimension space properly to get any accurate results. For the scale of data that AnB has at its disposal, I don't think KNN can possibly be a good supervised learning option for classifying new instances.

MODEL 4: BOOSTING (ADABOOST)

Boosting is an example of an ensemble learner, which relies on assembling an ensemble of "weak learners" which each do slightly better than a random guess would at classifying a new example. These weak learners are then used together to submit a final vote for the classification of a new instance. If the members of the ensemble are all weak learners (better than a random guess) then boosting is guaranteed to improve as more weak learners are added to the ensemble over time. It combats overfitting in this sense, where more iterations do not cause it to model noise, but instead improves on its ability to classify the underlying model (as long as the training set is fully representative of the distribution of the population). If a weak learner classifies a portion of the training set incorrectly, those instances have their weight "boosted" in importance for the next week learner that is trained for the data set. In this way, boosting ensures that difficult to classify instances will always, eventually, be classified correctly. The boosting implementation I chose was `sklearn.ensemble.AdaBoostClassifier`[8].

I varied the classifier with different types of weak learners to see if performance would change. I used DecisionTrees, Perceptrons, and SVMs. For the DecisionTree classifier, I chose to limit each tree for Minimal-Derived and Derived-Final at a max depth of 8. In some preliminary experimentation, I thought it would be smart to choose the best depth as determined by the decision tree experiment above for each data set (depth of 44 for Minimal Derived and 8 for Derived-Final), but I found that if I chose depth 44 for the Minimal-Derived case that the same identical tree would be fit for each member of the ensemble. The boosted learner wouldn't perform any better than a single tree on its own. It is OK to prune more aggressively in a boosted case, since there isn't just one tree making all classification decisions, but a full ensemble of trees working together. I also varied the number of estimators from 10 to 1000 to see how number of estimators in an ensemble might affect accuracy. See data collected in `/adaboost/final_graphs`.



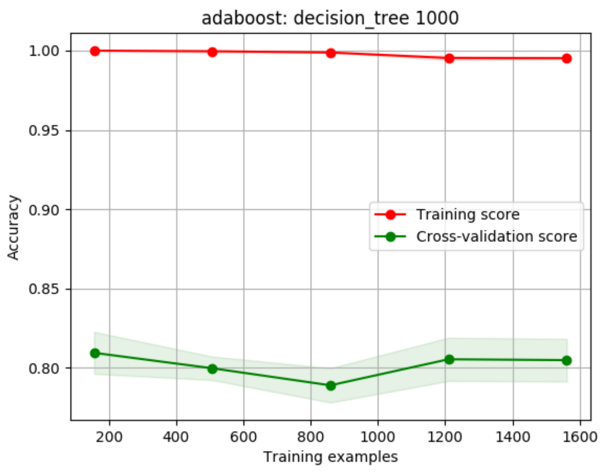


Figure 7: Adaboost decision tree learning curves (Minimal-Derived on left)

The results for my adaboosted decision trees was somewhat surprising. In the case of the Minimal-Derived data set, the boosting algorithm resulted in a model that consistently had a higher accuracy than a single decision tree did on its own (97% classification accuracy, versus 96%). Accuracy also appeared to increase with the number of weak learners I added to the ensemble (until 10k where it dipped slightly). This trend of increasing accuracy as more weak learners are added makes sense, as the ability for the ensemble to make more accuracte predictions should increase as more members are added. This is consistent with my expectations for how boosting should behave. The learning curve for the Minimal Derived data case looks similar to the single decision tree model from part 1, but the accuracy is raised due to the addition of more trees in the ensemble. It is difficult to tell if the chosen classifier has plateaued with the maximum amount of data that I had available to train with, but it does appear that the top bound of the variance is still increasing as more data is added for training (so it is improving).

The Derived-Final boosted data definitely looks different from its single decision tree counterpart. Interestingly, the accuracy of the boosted classifier seemed to decrease as more learners were added to the ensemble. This indicates to me that the learners in the ensemble were picking up on some noise in the data that actually made them perform worse together than a single tree performed alone. This is also apparent from the learning curve, which shows that more data doesn't actually seem to impact the accuracy of the decision tree ensemble, and that accuracy maxes out around 81%, which is worse performance than the 83% baseline of a distribution-aware random classifier. Boosting is picking up on noise in my dataset in this case, which likely means that one of the derived attributes I made during the cleaning procedure for the Derived-Final dataset is harming the predication capability of a boosted classifier. Also interesting is that the training set prediction accuracy in the learning curve remains close to 1 for the boosted version of Derived-Final's classifier, whereas in the single decision tree model it bends down and plateaus around 83% to meet the cross validation curve as data increases. This also indicates to me that the boosted version of decision trees has overfit the data, and that the ensemble is actually doing more harm than good.

In both Minimal Derived and Derived Final cases, as more learners were added to the ensemble, wall clock time to make predictions increased proportionally to the amount of time it took to make predictions with 1 learner. For example, In the case of Minimal-Derived, it took 1.8 seconds to make a prediction using a model with 10 weak learners, and it took over 1000 seconds to make a prediction with 10000. The same pattern was observed for the Derived Final dataset. This makes sense, since more learners are involved in the "vote" for predictions of a new instance.

MODEL 5: SUPPORT VECTOR MACHINES (SVM)

The final supervised learning method I used to analyze the AnB data was Support Vector Machines (SVM) which are used to make classifiers as general as possible by finding a function that separates classes with a maximal margin between class boundaries. SVMs perform well when data is sparse, and rely on the injection of a kernel function to determine the degree of similarity between points in the space. The kernel function is where we can inject domain knowledge into the algorithm.

The SVM implementation I chose was `sklearn.svm.SVC`[9]. For this experiment, I chose to vary the kernel function with 4 different kernel implementations provided by sklearn: linear, polynomial, rbf, and sigmoid. I also chose to vary the shape of the decision function between 'ovo' and 'ovr', but after collecting results the choice between ovo and ovr didn't have any impact in the ultimate results of the classifier (collected values were identical. Data collected for the svm model experiments can be found in `/svm/final_graphs`.

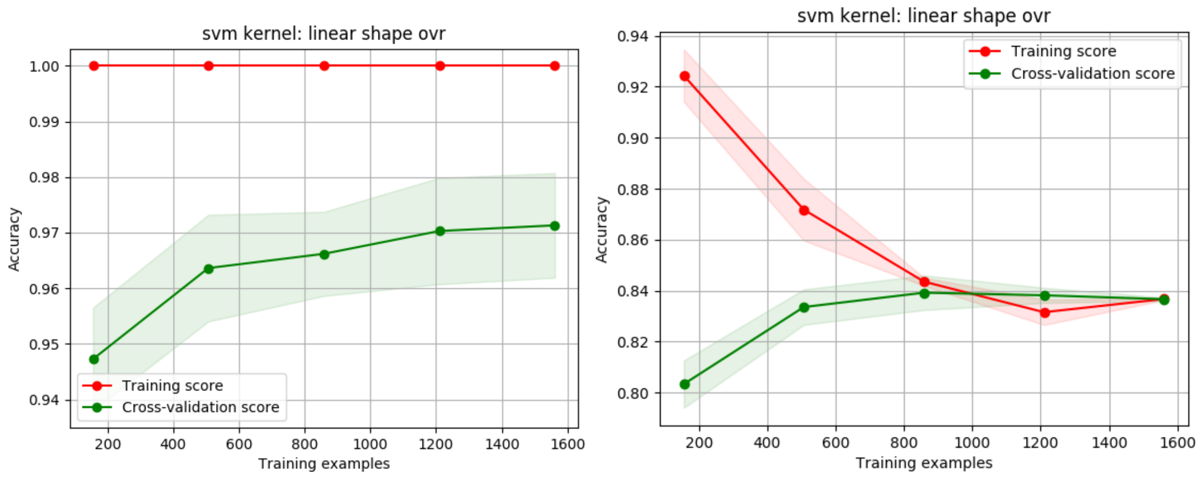


Figure 8: SVM learning curves (Minimal-Derived on Left)

The data collected from each kernel function points to the linear kernel being the best candidate for SVM for the Minimal Derived data set. The learning curve shows that as data increases, it is able to achieve upwards of 97% accuracy, which is the most accurate model out of any model I tested. Since the linear kernel performed the best, it means there is some linearly separable plane in the dimension space that split the classifiers with 97% accuracy. This is a big improvement from the 83% baseline. The derived final data returned the same accuracy score for every kernel, which was 83.66%. This is exactly the 83% baseline that a random classifier which is distribution aware would be expected to achieve. The difference in accuracy between the Minimal-Derived and Derived-Final learning curves indicate that there was some critical piece of information that was suppressed during the cleaning procedure that was indicative of a "True" class. Derived-Final's learning curve definitely plateaus, indicating that additional data will not improve the accuracy of the classifier. The Minimal-Derived CV curve appears to still be climbing, implying that more data could result in a more accurate classifier.

Wall clock time was relatively low for both datasets, with the linear kernel SVM taking the longest in both instances. The wall clock time was only 8 seconds for the Minimal-Derived dataset, which is a huge improvement when compared to similar accuracy models returned from boosted decision trees or neural networks.

CONCLUSION:

I was surprised that there was such a difference in prediction quality between the Minimal-Derived and Derived-Final datasets. I thought the cleaning procedures I ran to produce the Derived-Final dataset would remove noise from the data, and highlight important information about the underlying population. I appear to have been too aggressive in my cleaning methods, as the classifiers trained from the Derived-Final data never did much better than a random distribution-aware classifier would do (not much better than 83% in any cases). The quality of the performance of the classifiers trained from the Minimal-Derived dataset also surprised me. I spent a lot of time before training any classifiers, pouring over the data and trying to make it as clean and succinct as possible, when in hindsight this was only hurting the ability of the classification algorithms to draw insight out of the data. I think this also highlights the power of these algorithms to find underlying patterns in data, and unless you have domain knowledge about the data set, it's best to leave the data as raw as possible.

It appears that of the supervised learning models run above, for the Minimal Derived dataset, the model that resulted in the most accurate predictions of whether an instance would graduate the AnB program was a close tie between SVM and Boosted decision trees using adaboost. The accuracy predictions from both adaboost and svm obtained in my experiments were so similar, that choosing one over the other would come down to wall clock training time: how fast can you get high quality predictions. In this case, the SVM model would clearly win over the boosted decision tree. Adaboost took almost 2 minutes to train 1000 weak learners for the ensemble to achieve 97% accuracy, while it only took 8 seconds to train a linear kernel SVM to achieve the same accuracy.

The least accurate supervised learning model for my data sets by far was KNN. As an instance based learner, KNN falls victim to the "curse of dimensionality", which says that as the dimension space of your data increases, the amount of data necessary to generalize to the underlying population grows exponentially. The number of attributes in the Minimal-Derived dataset was somewhere close to 8000, and the number of attributes for Derived-Final was still 118. There are just far too many attributes and far too few instances for KNN to ever achieve accurate predictions for this data set.

Another question I considered was, of these models which would be the most useful for the AnB organization? Sure, it's helpful to have a 97% accurate classifier that can tell you whether a new patient instance is likely to complete their programming, but it would be even better if the classifier wasn't such a black box, and you could actually peek inside at the logic to understand what it is about an instance that makes them more or less likely to succeed. Since this information would be the most helpful to AnB, I think the decision tree classifier would provide the most value to them: it is still 96% accurate in its predictions of whether a new instance will complete their programming, but it is also easy to tell which attribute values played the greatest role in classifying those instances (i.e. which nodes were placed highest up in the decision tree and provided the greatest gini_gain). If AnB knows which attributes are the most important in the intake data towards successful completion of programming, and any of those intake attributes are under their control (like the case manager or therapist assigned to a patient, or even which patient diagnoses tend to lead to the most successful graduation rates) AnB can try to intervene to influence the outcomes.

Exploring this data was a great learning experience. From pre-processing of data (really "overprocessing" the data), to seeing how different learning algorithms compare under different conditions, this hands-on lab has given me a taste of what it would be like working with labeled data in the real world in a supervised learning context. To improve further on my models, I think I could apply some of the cleaning steps that I applied to the Derived-Final data to the Minimal Derived data as well: such as cleaning up category values that essentially mean the same thing, but are listed as different strings. Also, creating buckets for continuous values (like age) and deleting date values that are too specific to provide any useful information. I think a second, less aggressive cleaning of the data will provide even more insight, and can hopefully raise the accuracy of my trained classifiers even further. After that, I hope to present my findings to my colleagues at Uptake and to our professor at Northwestern, and hopefully provide the classification models to AnB to help them in their intake process going forward.

CITATIONS

[1]Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.
[2]Michalski, Ryszard S., et al. *Machine Learning: an Artificial Intelligence Approach*. McGraw Hill Education, 2013.
[3] http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
[4] http://www.bambielli.com/til/2017-10-29-gini-impurity/
[5] https://en.wikipedia.org/wiki/Predictive_analytics#Classification_and_regression_trees_.28CART.29
[6]http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier
[7]http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier
[8]http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier
[9] http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
[10] http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html
[11] http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html
[12] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html
[13] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.learning_curve.html