Case Study I: Ensemble Methods:
Evaluating the Predictive Capabilities of Predictive Testing

By Alexander Stachniak

Table of Contents

Introduction

Students who graduate from an accredited nursing program and who wish to enter the field as a Registered Nurse (RN) are required to take and pass the National Council Licensure Examination (NCLEX), a nationwide licensure exam. The NCLEX is a high-stakes test, both for graduating students and for education administrators. While students need to pass the NCLEX in order to practice as a nurse, administrators rely on NCLEX pass rates for the success of their programs. Not only can historically high NCLEX pass rates be a significant marketing tool for prospective students, but low pass rates can eventually lead to a loss of accreditation and program closure. Meanwhile, educators struggle to keep up with an ever-changing exam, student anxiety, and how to track long-term curricular and evidence-based practice changes in their field.

The complexities listed above result in a significant amount of resources being invested in NCLEX preparedness and success. The majority of nursing programs in the country make use of a predictive testing product developed by an external vendor to periodically evaluate students' probability of passing the NCLEX after graduation. These predictive testing products purport to be reliable indicators of NCLEX success, but they are validated at a national level and may not suit the needs of a particular nursing program. Given the cost of predictive testing products and the importance of maintaining high NCLEX pass rates, Nursing programs have a clear business need to evaluate the usefulness of such assessments as part of their curriculum.

In this Case Study, the author evaluates several predictive algorithms in an attempt to predict the result of the NCLEX (i.e., "Pass" or "Fail") for students of an accredited nursing program ("The Program"). As explanatory variables, the author makes use of student results from predictive testing examinations created by a widely recognized vendor ("The Vendor") from the past ten years. The author evaluates a variety of standard classification techniques, including Decision Tree, Naïve Bayes, K-Nearest Neighbors, and Logistic Regression, as well as stacking, bagging, and boosting ensemble methods. Algorithms are compared first and foremost by their ability to minimize error. Although the main goal is correct classification, minimization of error will provide a more nuanced approach to comparison. Furthermore, probabilistic predictions would be widely used in practice by educators seeking to understand where remediation is warranted. Attention will also be paid to scrutibility, complexity, and scaleability.

Data Description

The dataset contains NCLEX result data and predictive testing scores for 959 students over a ten-year period. As can be seen in Table 1, the source data was spread across a large number of static files, due to the limitations of the systems from which the data was gathered. In order to ensure that student data remains private, the data merging stage was conducted separately in a secure environment. A recursive function captured data from each file before merging together. A limited amount of preprocessing was conducted at this stage, but only insofar as it was required to properly merge the data and to protect privacy.

Table 1: Data Fields and Sources

Name	Relevant Fields	Number and Type	Source
NCLEX Testing Data	Candidate ID, Test Date, Result (Pass/Fail)	1 Excel file	State Board of Nursing
Predictive Testing Data	User ID, Assessment, Test Date, Section, Data by Section: [Raw Score, National Percentile, Program Percentile, Proficiency Level (ordinal value), National Mean, Program Mean]	1190 CSV files	Proprietary database of The Vendor

All personal identifiers (e.g., name, email, etc.) were dropped during the data merging stage, with the one exception being Student ID, which is needed as a primary key across records. To protect anonymity, Student ID was masked and randomized before being turned into an ordered integer index. Finally, data was pickled for easy import to Python.

All steps of the data cleaning and data analysis process were conducted in Python 3.6 and using standard the libraries of Pandas, Numpy, SciPy, Sci-Kit Learn, Pickle, Datetime, and Matplotlib and Seaborn for plotting.

Data Cleaning

The target variable was converted from textual representation (i.e., capitalization variations of "Pass" and "Fail") to a binary integer, 0 or 1. The original dataset only includes first-time test-takers, so any duplicates were removed as errors.

Preparing the explanatory variables was a much more challenging task. The core problem is in the variability of the data. Over the ten-year period, The Vendor updated their various exams several times (in content, structure, and scoring), and The Program altered their curriculum (in both length and order of courses). Students in The Program have the ability to take additional versions of exams as practice or remediation. Furthermore, the NCLEX exam itself underwent significant modifications in 2013. All of these variables make it extremely challenging to norm student data over time.

The solution to this problem was two-fold. First, the author sought a scoring mechanism which would be comparable over time. Luckily, The Vendor reports national means for every test they provide, means which are updated as the test is revised. It is trivial, then, to calculate the distance from the national mean for a student's raw score. This new metric would be impervious to differences in various exams, would be comparable over time, and would be resistant to future changes. It would also guard against changes to the NCLEX, as the pass rates for the NCLEX are themselves dependent on national means.

The NCLEX uses Computerized Adaptive Testing (CAT), which displays questions to candidates in a way that attempts to understand each candidate's ability while using as few questions as

possible. A standard exam requires all candidates to answer the same questions, while CAT attempts to always display questions that a candidate should find challenging (based on his or her ability). Questions, therefore, must be tested ahead of time across a wide sample of respondents, and a national mean for each question is determined. In addition, the passing standard is based, at least partially, on past NCLEX results, sample test-takers, and education readiness of high-school graduates interested in nursing, all of which are nationally averaged.

The second solution to the data cleaning problem was to find a way to convert scores for different types of tests, taken at different points of a student's career, into a standardized set of explanatory variables. Student A, for example, may have taken tests 1, 2, and 3, whereas Student B took 2, 5, and 6. Student C may have taken different versions of tests 1, 3, and 5. Although using the distance to the national mean provides a comparison between the exams, it does not resolve two issues: 1) how does one compare scores for content areas that have completely changed or disappeared, and 2) how does one deal with missing values for students who did not taken an exam.

The author's solution in this case was to group the various "sections" of exam into 160 content areas for which a student could earn a score, then average those scores together across time. So, for instance, a student might earn a score on "health assessment" for a Medical-Surgical exam as well as for a later Pediatrics exam. By averaging these results, one can compare students by their proficiency in a content areas. Because a distance metric is being used, students start with a baseline value of 0.0 for each content area, and their scores are slowly updated over time. In this way, students who have taken different versions of different exams at potentially different times in their careers can still be compared without having to throw out observations which do not contain a score for every content area.

Grouping data into content areas required that all exam sections be mapped. This process was conducted in coordination with a domain expert in the area, and was approached with an extremely light hand, mainly to correct for small variations and inconsistencies (e.g., "Non-Pharmacological Comfort Interventions" and "Nonpharmacological Comfort Interventions"). In some cases, closely related categories were combined (e.g., "Musculoskeletal" and "Musculoskeletal Disorders"). Finally, in some cases very basic skills were combined into broader categories (e.g. "Percentages" and "Fractions & Decimals" were combined into "Basic Math"). This concept mapping has the benefit of reducing the total number of explanatory variables (important given the small number of total observations), but also makes the data resistant to future changes. Should The Program ever change vendors, so long as the new vendor provides raw scores and national means at the concept level, even if the concepts themselves and the scoring function change, it would still be possible to compare past data with future data. Suggested future work in this area would be to apply clustering techniques to group the explanatory variables.

Table 2: Example of the Mapping Procedure

Inconsistent Naming	Non-Pharmacological Comfort Interventions Nonpharmacological Comfort Interventions	→	Non-Pharmacological Comfort Interventions
Closely Related Topics	Musculoskeletal Musculoskeletal Disorders	→	Musculoskeletal
Basic Skills	Percentages Fractions & Decimals	→	Basic Math

Standard data cleaning methods were also applied to this dataset. A significant number of duplicate values were dropped (a byproduct of the data retrieval process). Distance to the National Mean was calculated using Euclidean distance, then scaled to -1.0 to 1.0.

Several variables were dropped in order to focus on distance to the national mean. Assessment Name, Assessment ID, Booklet ID, and Date Taken were all dropped specifically as a result of the distance score being capable of accounting for these variations. National Mean and Program Mean were dropped (after the distance score was calculated), as neither is a student-specific variable. National Percentile would be too similar to keep alongside Distance to National Mean. The latter was preferred for its resistance to future change (percentile is not something The Program can calculate on its own). Program Percentile was also dropped in favor of a nationally validated score, which is better aligned to NCLEX scoring. Proficiency Level, which is determined by The Vendor using a proprietary algorithm, is intended to provide insight into how proficient a student is in a particular content area. Because so many values are missing, however, this variable was dropped rather than making an attempt to backwards-engineer the algorithm. Finally, Score was dropped following the calculation of Distance to National Mean, and Section was dropped in favor of content area.

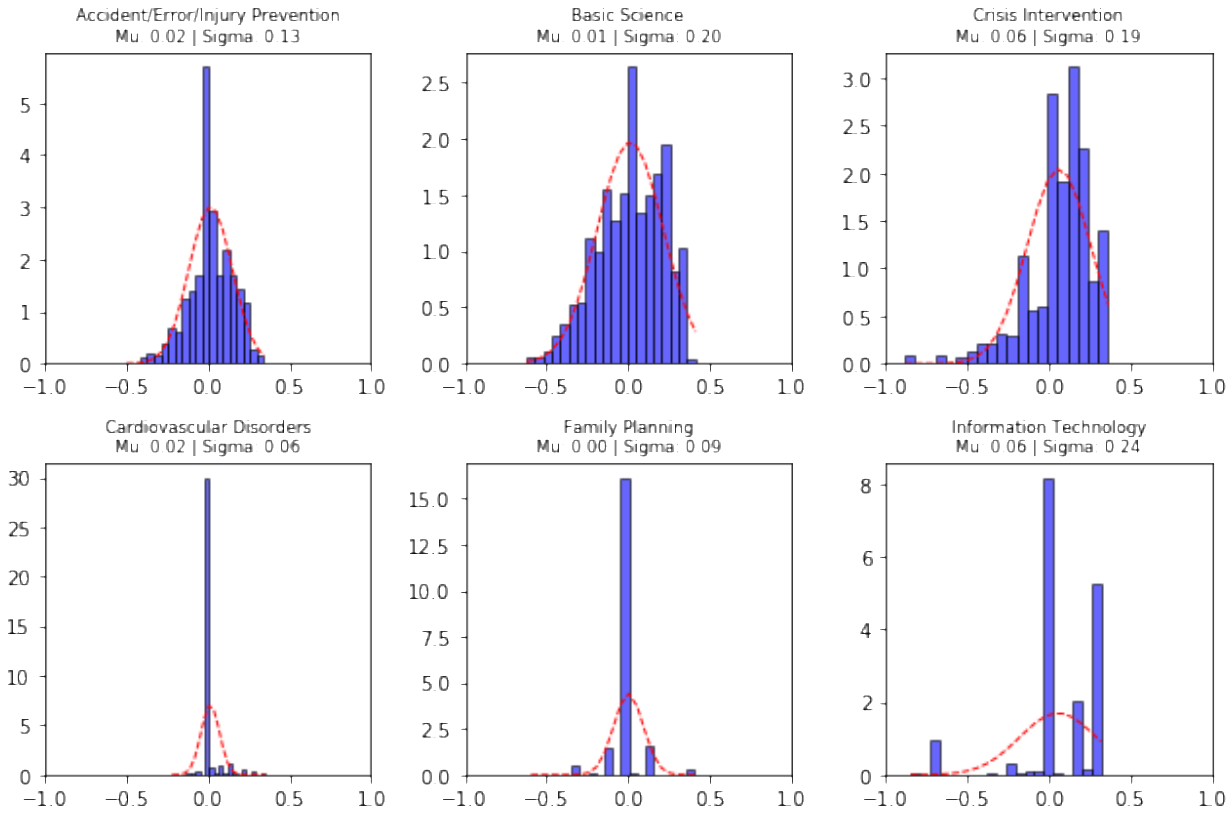
Data Analysis

The data consists of 959 observations, 160 continuous explanatory variables, and 1 binary class label as the target variable.

As a result of the data cleaning process, all 160 explanatory variables have a range of -1.0 to 1.0. Because the base score value is 0.0, and because the deviations from the base score are based on educational tests, data is expected to be normally distributed. Those which cannot be described as normally distributed represent cases where relatively few students have a score in that content area.

Due to the large number of variables, only the distributions of a few examples are displayed in Figure 1. The three concepts represented in the top row are all centered close to 0 and have a noticeable bell curve. The three concepts represented in the bottom row, however, are overrepresented at point 0.0, leading to a distorted shape.

Figure 1: Example Distributions of Explanatory Variables

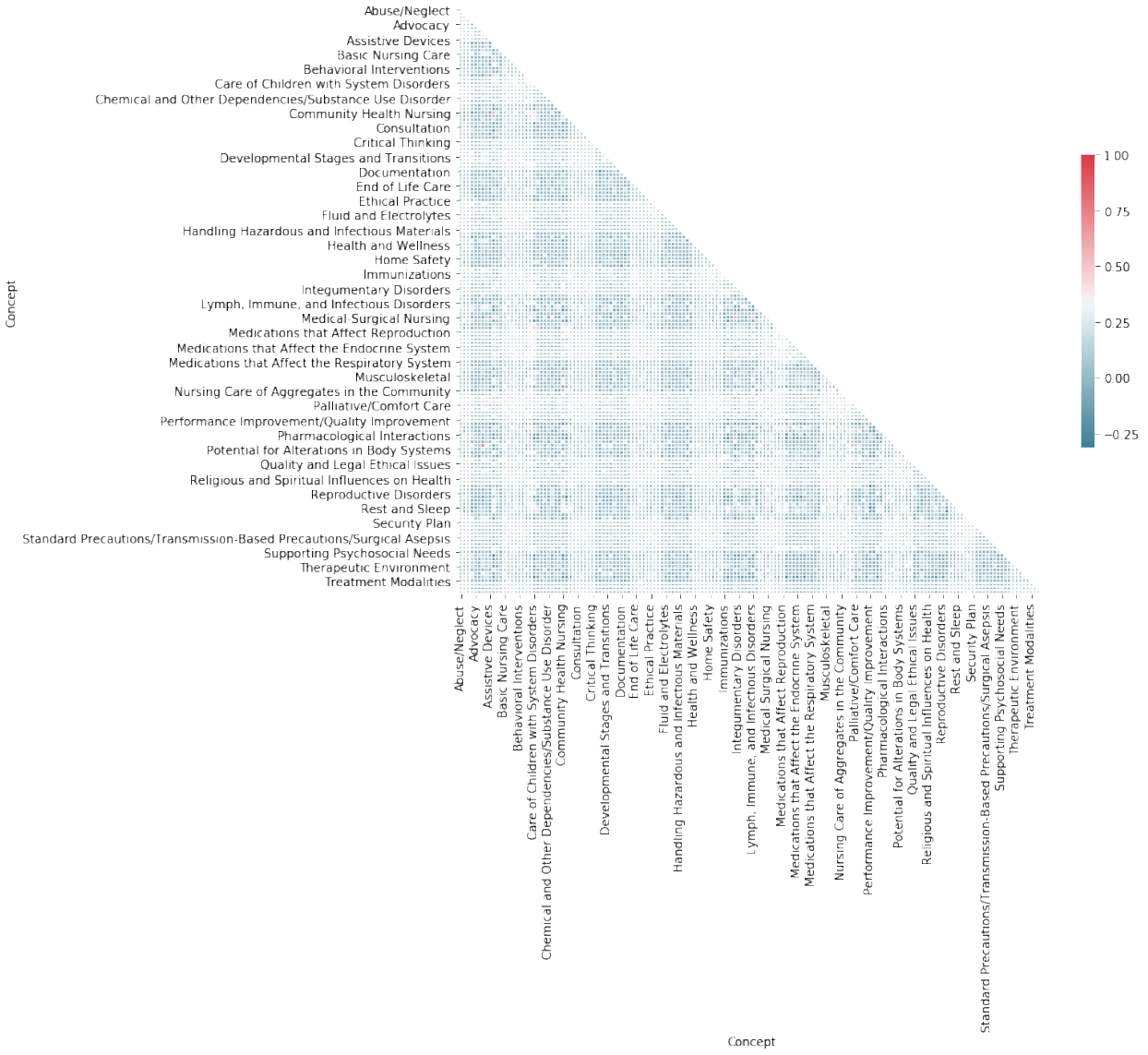


Although future work might utilize pruning to remove variables with very few reported scores, their inclusion represents an important test case for the reliability of the model. If used in practice, the model will consistently be fed student data which is incomplete (e.g., a student who is midway through her program), and so the model should be capable of producing accurate results regardless of the number of scores reported. The explanatory variables are intentionally not scaled to a normal distribution, as variables with little variance should not be over-emphasized.

One fear with a concept-based mapping approach was that scores might be highly correlated to one another. It would be reasonable to assume that students who were successful in one concept might also be successful in related concepts. However, in practice, correlations between the various concepts remained extremely low. As can be seen in Figure 2, the majority of all correlations are between -0.25 and 0.25. There exists significant banding in the figure, which is attributable to the alphabetical presentation of the explanatory variables. In other words, similar concepts tend to have similar names, leading to slightly higher correlations in

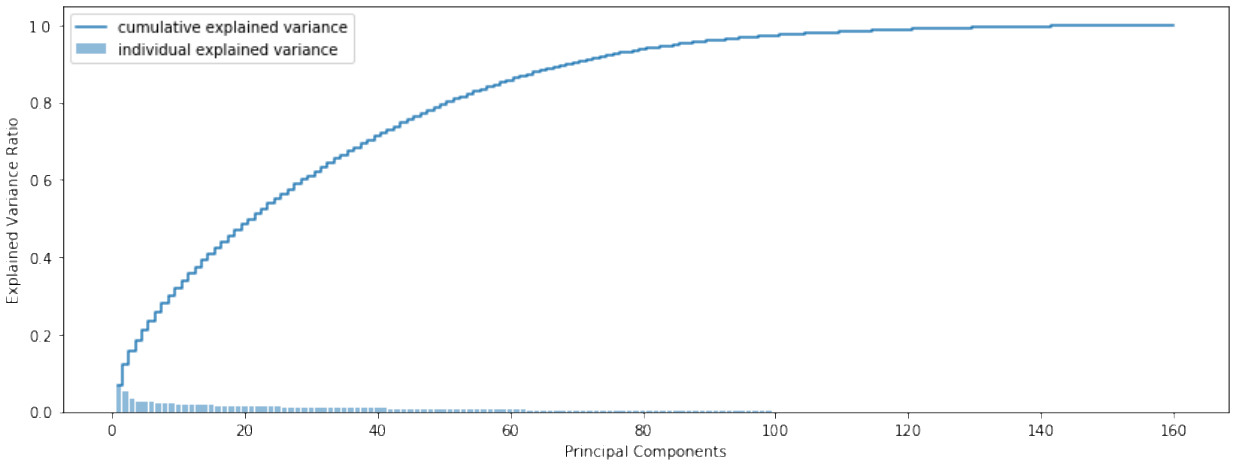
sequential alphabetical concepts. However, as stated, even these correlations are well below the threshold of what is typically considered to be high correlation.

Figure 2: Correlations between Explanatory Variables



To further demonstrate that correlation (and covariance) are not a major concern Principal Component Analysis (PCA) was used to rotate the data along the axes with the most variance. The cumulative explained variance of each of our newly created axes, or principal components, was then graphed (see Figure 3). As can be seen, even the first principal component only covers 6.85% of all variance, and it would require close to 60 principal components to reach 80% of cumulative variance. Based on these results, it is concluded that correlation and covariance are not a significant concern in the data. It is also concluded that PCA is not a valid approach for the data.

Figure 3: Cumulative Explained Variance of Explanatory Variables



Finally, feature importance is reviewed using a Random Forest classifier. As part of the algorithm, the Random Forest classifier ranks how much each explanatory variable increases the purity of the nodes. One can then extract that information to better understand the importance of each variable. The output of the top ten most important features is displayed in Table 3. As was already verified using PCA, there is no obvious subset of variables that can be extracted, and at this stage, all variables are retained. In a production environment, educators would be interested in having access to the scores of all variables, not just a select few.

Table 3: Feature Importance Using Random Forest Classifier

```
 1 (0.0561) Overall
 2 (0.0286) Management of Care
 3 (0.0247) Psychosocial Integrity
 4 (0.0197) Basic Nursing Care
 5 (0.0187) Expected Actions/Outcomes
 6 (0.0183) Alterations in Body Systems
 7 (0.0183) Ante/Intra/Postpartum and Newborn Care
 8 (0.0171) Behavioral Interventions
 9 (0.0168) Diagnostics
10 (0.0167) Potential for Alterations in Body Systems
```

Experimental Results

Before training any algorithms, 20% of all observations are reserved as a hold-out test sample, using stratification to ensure a similar distribution of target classes in training and testing sets. For each algorithm, a set of parameters and possible values are defined, then hyperparameter tuning is performed using bootstrap sampling and the log loss function as the scoring metric. Taking the best parameters, each algorithm is fit 200 more times using another round of bootstrap sampling. Finally, the algorithms are compared to one another using a variety of metrics, including log loss, accuracy, precision, and recall.

Decision Tree, Naïve Bayes, K-Nearest Neighbors, and Logistic Regression were the four standard classifiers that were tested. Three stacked ensemble algorithms were trained, the first

using all four of the standard classifiers, the second dropping Naïve Bayes, and the third dropping Logistic Regression. Finally, two bagging algorithms, Random Forest and Complete Random Trees, and one boosting algorithm, AdaBoost using Decision Trees, were trained. The results of each algorithm is published below in table format. A visualization of the results is also provided, using a two-part plot where the top plot presents boxplot distributions for all algorithms (whiskers defined by 1.5 x IQR; outliers represented by red boxes), and the bottom plot presents histogram distributions for all algorithms along with a dotted line to indicate a 95% confidence interval.

Table 4: Comparison of Log Loss Scores

Decision Tree	Mean: 0.93	Std: 0.41	95%Conf: (0.14, 1.73)
Naive Bayes	Mean: 8.29	Std: 3.59	95%Conf: (1.25, 15.33)
K-Nearest Neighbors	Mean: 1.11	Std: 0.55	95%Conf: (0.03, 2.18)
Logistic Regression	Mean: 0.42	Std: 0.04	95%Conf: (0.35, 0.50)
Stacked Ens (4 Models)	Mean: 0.39	Std: 0.03	95%Conf: (0.33, 0.46)
Stacked Ens (No GNB)	Mean: 0.31	Std: 0.02	95%Conf: (0.27, 0.36)
Stacked Ens (No Log)	Mean: 0.44	Std: 0.06	95%Conf: (0.32, 0.56)
Random Forest	Mean: 0.28	Std: 0.04	95%Conf: (0.21, 0.36)
Complete Random Trees	Mean: 0.28	Std: 0.03	95%Conf: (0.22, 0.33)
AdaBoost (Tree)	Mean: 0.47	Std: 0.11	95%Conf: (0.25, 0.69)

Figure 4: Distribution of Log Loss Scores for All Algorithms

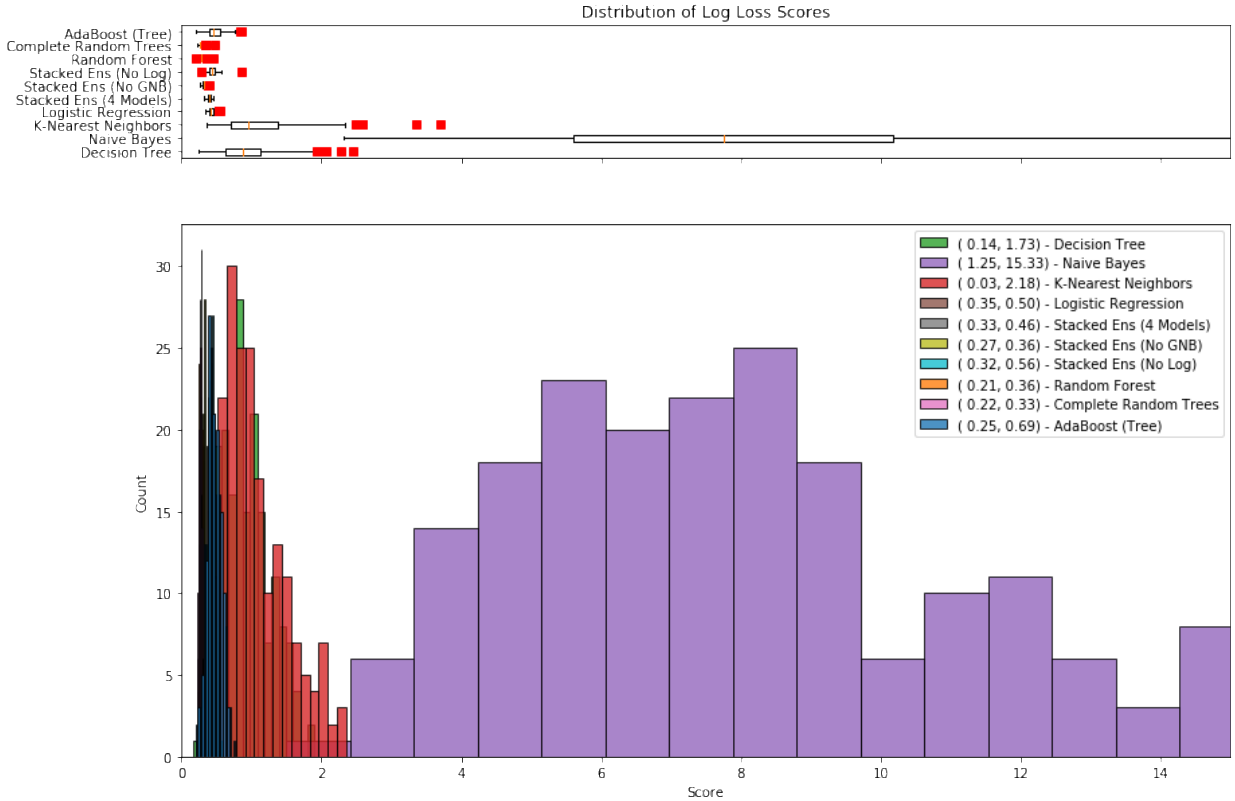


Table 5: Comparison of Accuracy Scores

Decision Tree	Mean: 0.88	Std: 0.02	95%Conf: (0.84, 0.92)
Naive Bayes	Mean: 0.55	Std: 0.09	95%Conf: (0.37, 0.73)
K-Nearest Neighbors	Mean: 0.87	Std: 0.03	95%Conf: (0.81, 0.94)
Logistic Regression	Mean: 0.82	Std: 0.02	95%Conf: (0.78, 0.86)
Stacked Ens (4 Models)	Mean: 0.88	Std: 0.02	95%Conf: (0.84, 0.91)
Stacked Ens (No GNB)	Mean: 0.89	Std: 0.01	95%Conf: (0.87, 0.92)
Stacked Ens (No Log)	Mean: 0.87	Std: 0.03	95%Conf: (0.82, 0.93)
Random Forest	Mean: 0.90	Std: 0.01	95%Conf: (0.88, 0.93)
Complete Random Trees	Mean: 0.90	Std: 0.01	95%Conf: (0.87, 0.93)
AdaBoost (Tree)	Mean: 0.90	Std: 0.01	95%Conf: (0.87, 0.93)

Figure 5: Distribution of Accuracy Scores for All Algorithms

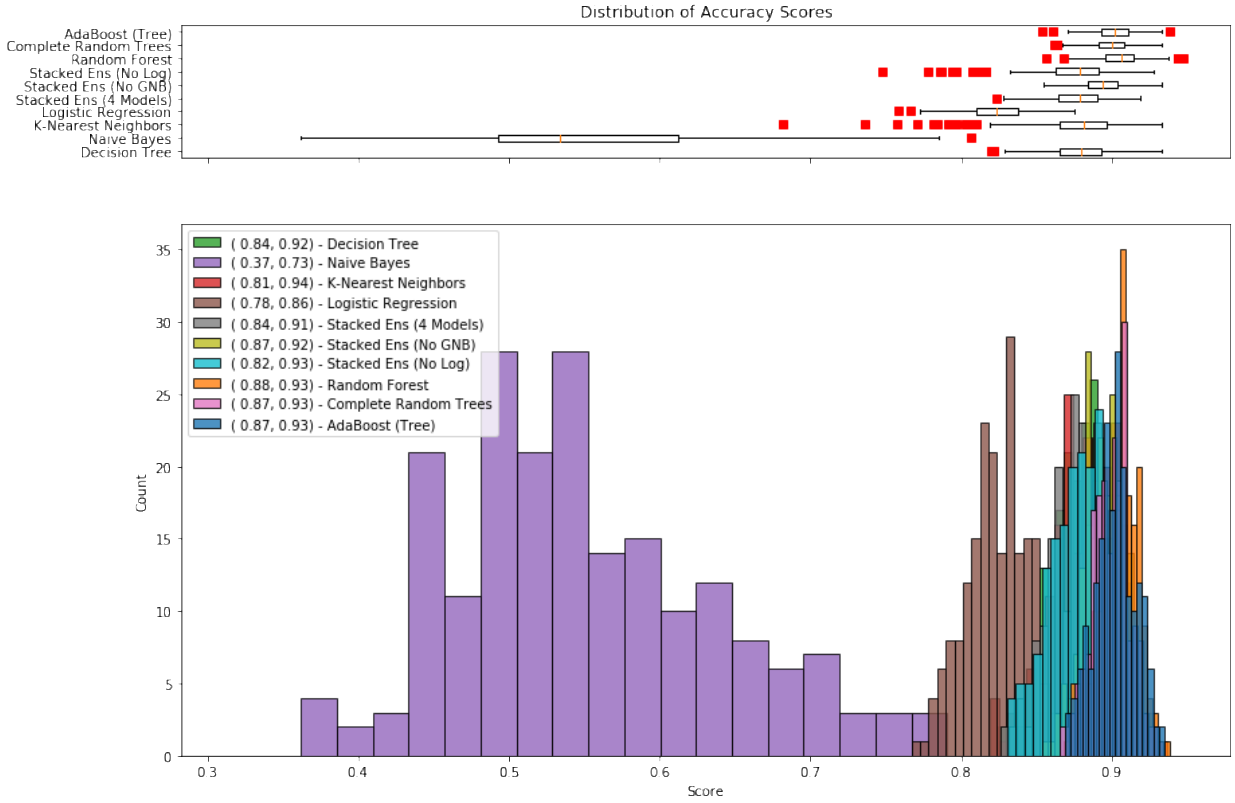


Table 6: Comparison of Recall Scores for "Fail" Class

Decision Tree	Mean: 0.23	Std: 0.08	95%Conf:	(0.08, 0.38)
Naive Bayes	Mean: 0.66	Std: 0.07	95%Conf:	(0.52, 0.79)
K-Nearest Neighbors	Mean: 0.02	Std: 0.04	95%Conf:	(-0.05, 0.09)
Logistic Regression	Mean: 0.54	Std: 0.08	95%Conf:	(0.38, 0.69)
Stacked Ens (4 Models)	Mean: 0.41	Std: 0.09	95%Conf:	(0.23, 0.58)
Stacked Ens (No GNB)	Mean: 0.21	Std: 0.07	95%Conf:	(0.06, 0.35)
Stacked Ens (No Log)	Mean: 0.28	Std: 0.09	95%Conf:	(0.10, 0.46)
Random Forest	Mean: 0.14	Std: 0.05	95%Conf:	(0.03, 0.24)
Complete Random Trees	Mean: 0.07	Std: 0.04	95%Conf:	(-0.01, 0.15)
AdaBoost (Tree)	Mean: 0.12	Std: 0.06	95%Conf:	(0.01, 0.23)

Figure 6: Distribution of Recall Scores for "Fail" Class Scores for All Algorithms

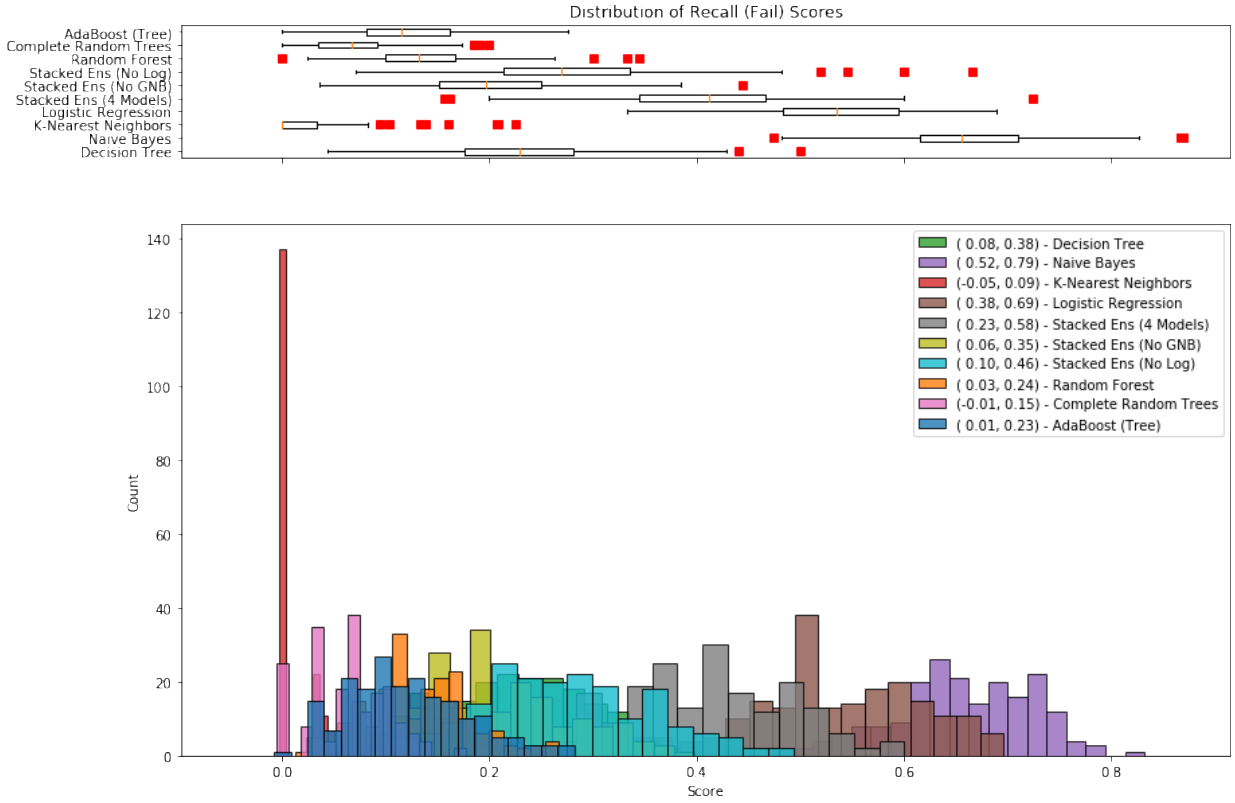
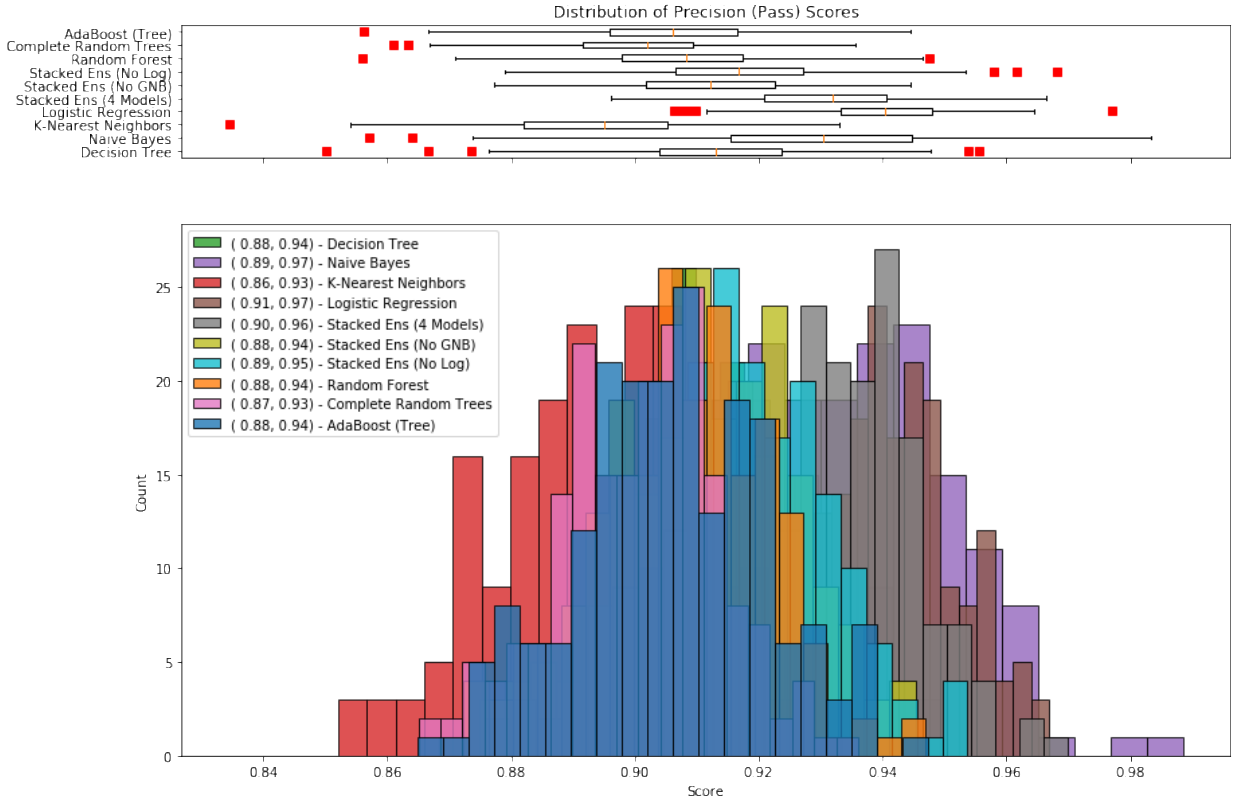


Table 7: Comparison of Precision Scores for "Pass" Class

Decision Tree	Mean: 0.91	Std: 0.02	95%Conf: (0.88, 0.94)
Naive Bayes	Mean: 0.93	Std: 0.02	95%Conf: (0.89, 0.97)
K-Nearest Neighbors	Mean: 0.89	Std: 0.02	95%Conf: (0.86, 0.93)
Logistic Regression	Mean: 0.94	Std: 0.01	95%Conf: (0.91, 0.97)
Stacked Ens (4 Models)	Mean: 0.93	Std: 0.01	95%Conf: (0.90, 0.96)
Stacked Ens (No GNB)	Mean: 0.91	Std: 0.01	95%Conf: (0.88, 0.94)
Stacked Ens (No Log)	Mean: 0.92	Std: 0.02	95%Conf: (0.89, 0.95)
Random Forest	Mean: 0.91	Std: 0.02	95%Conf: (0.88, 0.94)
Complete Random Trees	Mean: 0.90	Std: 0.01	95%Conf: (0.87, 0.93)
AdaBoost (Tree)	Mean: 0.91	Std: 0.02	95%Conf: (0.88, 0.94)

Figure 7: Distribution of Precision Scores for "Pass" Class Scores for All Algorithms



Experimental Analysis

The experimental results provide a significant amount of data to unpack. Beginning with the log loss scores, the most striking result is the extremely poor performance of the Naïve Bayes classifier with scores being an order of magnitude higher than all other classifiers. Had previous results not shown conclusively that there exists little relationship between the explanatory variables, one might wonder if the naïve assumption in Naïve Bayes, that of variable independence, was causing this behavior. That does not appear to be the case. Instead, the reason for this poor performance is likely to do with how Naïve Bayes makes predictions (by multiplying probabilities together), combined with the tendency of the log loss function to overly punish wrong predictions.

The log loss metric was chosen specifically in order to better measure small improvements in classification rate for the purpose of evaluation. If one of these models were to be used in the real world, even if two algorithms made the same class prediction, educators would assign a great deal of importance to being 95% sure that a student needs remediation versus being 51% sure that a student needs remediation. Small improvements to the model matter.

The potential problem with log loss, though, is that the metric lacks symmetry. As correct predictions approach 1.0, the log loss function will approach 0, but as incorrect predictions approach 0.0, the log loss function will approach negative infinity. What this means is that as the probability assigned to an incorrect prediction gets worse and worse, log loss will penalize that prediction exponentially with no limit to how heavy this penalty can be. Naïve Bayes, which repeatedly multiplies very small numbers together, can easily fall into the trap posed by log loss, making a prediction that is very close to 0.0 and is thus heavily penalized.

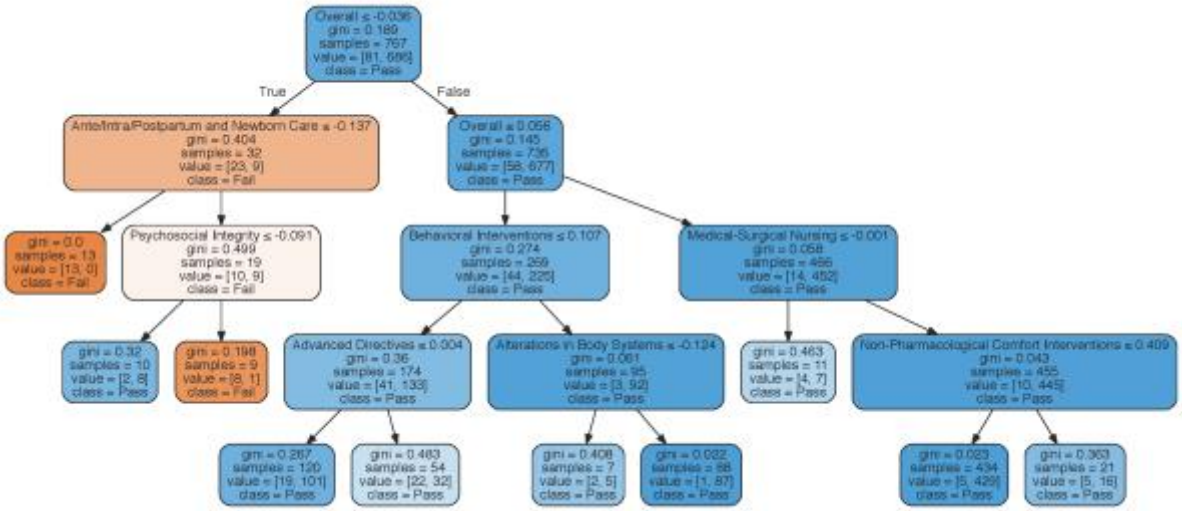
Naïve Bayes also scores poorly in terms of accuracy, with a mean of 55% and a confidence interval that dips well below random chance. However, these poor scores tell only half the story. Naïve Bayes outscores every other model in recall of the minority class ("Fail") and outscores all but one model in precision of the majority class ("Pass"). These particular scoring metrics are extremely important in this application. Educators will ultimately be most worried about predicting the minority class ("Fail"), as those are the students in need of remediation, so high recall of this class is a necessity. However, it is also important to balance this with high precision in the majority class ("Pass"). If a student is predicted to pass, the algorithm needs be extremely confident or the educator runs the risk of denying that student needed remediation resources. Ultimately, a balance is sought between the two metrics, and Naïve Bayes provides that balance very well.

The Decision Tree classifier performs average in all scoring metrics. It fails to stand out in any particular category. Through hyperparameter tuning, the gini index was chosen as the splitting criterion, the max depth of the tree was capped at four, and the minimum number of observations in any leaf node was capped at seven. The visual output (see Figure 7) is difficult to represent due to the long names of many of the variables, but the algorithm is also much less interesting than hoped. It begins by splitting on Overall, a roll-up score for exams. Students with low values are generally likely to fail the NCLEX, and those with high values are likely to succeed. A machine learning algorithm is not needed for this type of obvious result. There are some more notable splitters, however, such as "Psychosocial Integrity," a category that accounts for 6-12% of all questions on the NCLEX[1]. Additionally, splitters like "Ante/Intra/Postpartum and Newborn Care," a catch-all term for obstetrics, "Medical-Surgical Nursing," and "Behavioral Interventions" represent core areas of nursing education, all of which scored highly in feature importance testing. Although the decision tree makes sense, in that it is choosing to split on big, important concepts, it is not clear enough to use in any sort of production environment, first because it does not add much value, and second because many

[1] "2016 NCLEX-RN Detailed Test Plan." www.ncsbn.org/2016_RN_Test_Plan_Candidate.pdf.

of the leaf nodes contain large numbers of misclassified students. The lack of utility, coupled with mediocre classification performance is enough to reject this model.

Figure 8: Visualization of Trained Decision Tree Algorithm



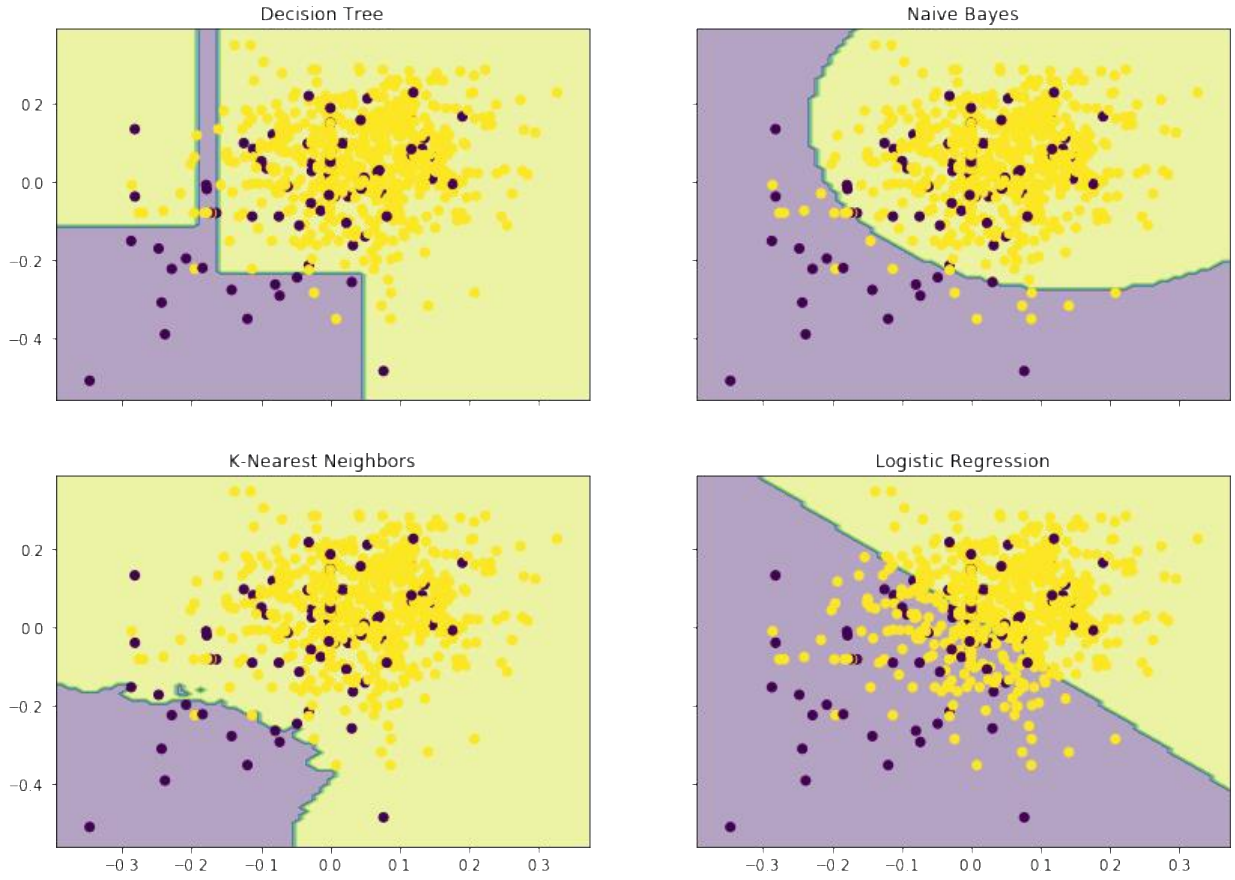
The K-Nearest Neighbors (KNN) algorithm records the second worst log loss score of the group (1.11), and with a fairly high variance as well. This makes intuitive sense as KNN is not a deterministic algorithm – the results will depend on the random seeds of the initial k points. More notable is the extremely low recall score of 0.02 for the Fail class. In other words, KNN trained to ignore the minority class, thereby earning a decent average accuracy score of 0.87 by predicting "Pass" almost every time. Obviously, such a model is not fit for a production environment. Interestingly, through hyperparameter tuning, a higher-order distance metric was chosen as the best parameter (i.e., Minkowski distance with p=3), but because the algorithm performed so poorly, it is difficult to read too much into that.

Finally, the Logistic Regression classifier appears to perform extremely well in terms of log loss (mean of 0.42), on par even with ensemble methods. The algorithm seems to stumble in terms of accuracy, however, recording the second lowest average score at 0.82. As with Naïve Bayes, however, the devil is in the details: Logistic Regression records the second highest recall score for the minority class (0.54) and the highest precision score for the majority class (0.94). The strong performance of Logistic Regression is unsurprising, especially considering that all 160 explanatory variables are continuous features. Moreover, the target variable is a binary classification, and Logistic Regression is perfectly suited to mapping an output to either the positive or negative class. Although Logistic Regression does not make use of many of the fancy statistical tricks that ensemble methods do, it just so happens to be extremely well-suited to this dataset.

It is possible to review the types of decision boundaries that each of these models forms (see Figure 9) by making predictions over a grid of points using two variables (here Psychosocial

Integrity' and Ante/Intra/Postpartum and Newborn Care are used). Starting with Logistic Regression, a clear linear split is made in the data. Although it appears that this type of algorithm would overly predict “Fail,” one needs to remember that this is only two dimensions of 160; in reality, Logistic Regression is forming a hyperplane that contains the bottom scorers in all 160 concepts. As an explanation, this is quite easy to understand.

Figure 9: Decision Boundary for Standard Classifiers



Progressing backwards, the KNN algorithm can be seen to create a globular decision boundary, which is to be expected from an algorithm that uses the distance between points. It may be possible to see some of the higher-order results in this decision boundary, especially where it appears to reach out to connect far off points. The poor recall score of this model is quite evident in how few points are contained in the negative prediction space. It may be that this model was too conservative. Similarly, the Decision Tree classifier did not predict many failures. The stepped decision boundary maps well to the algorithm itself, which creates hard cut-offs at each variable.

Finally, the distinct shape of Naïve Bayes may be attributable to the assumption of the algorithm that the data will be normally distributed, thus creating the expectation of curvature. Interestingly, this is the only decision boundary which could be described by a second order polynomial.

Based on the distinct results of the four standard classifiers, a perfect case for a stacked ensemble classifier would seem to be presented. The first stacked ensemble classifier (SE1), combines the optimal Decision Tree, KNN, Naïve Bayes, and Logistic Regression classifiers. Due to the interesting performance aspects of Naïve Bayes and Logistic Regression, a second (SE2) and third (SE3) stacked ensemble classifier were created, each of which drops one of those models. In this way, it is possible to see what kind of improvement Naïve Bayes and Logistic Regression added to the ensemble.

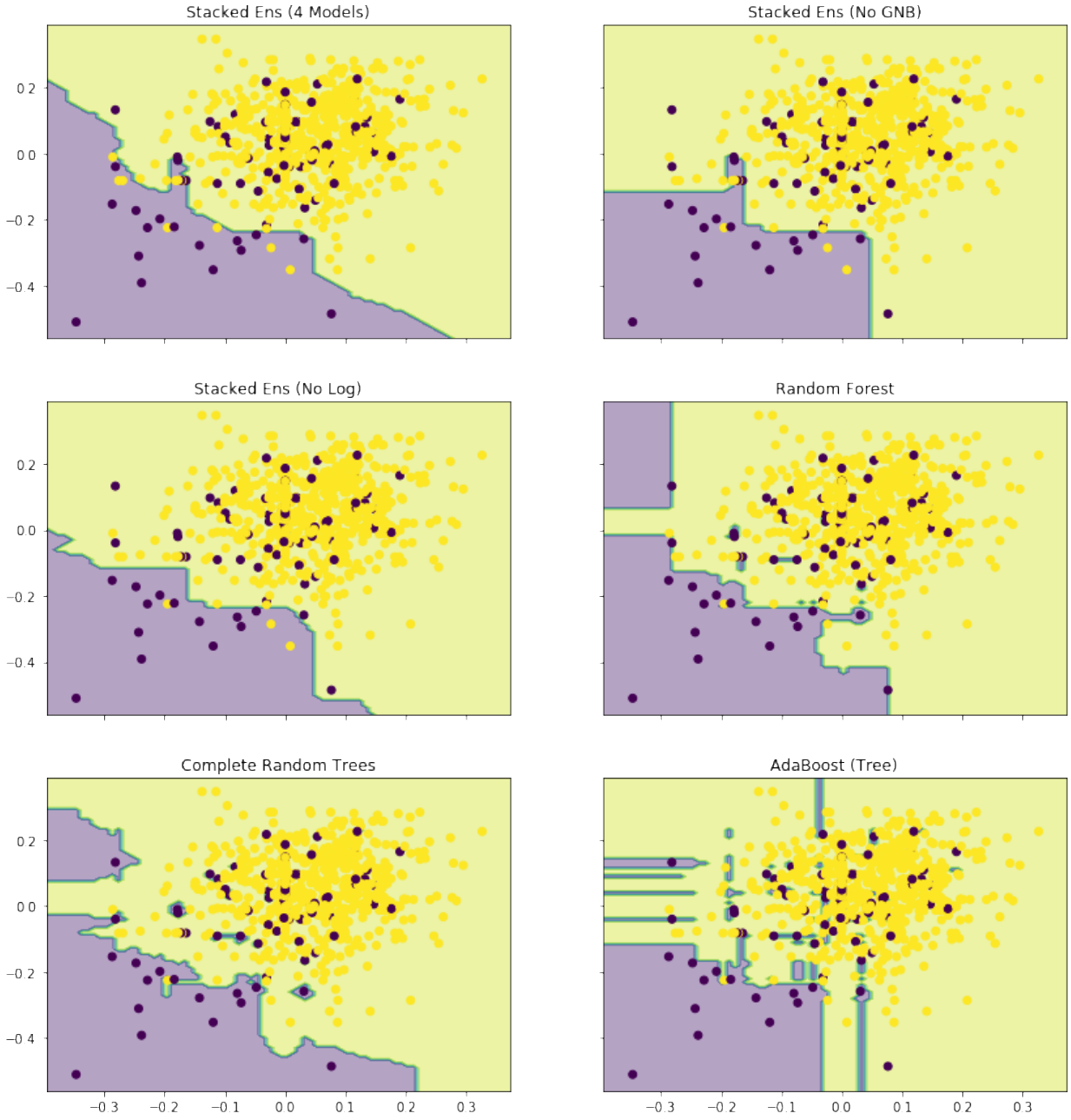
In terms of log loss, SE1 performs better than Logistic Regression alone, with an average score of 0.39. As might be expected, when Naïve Bayes is removed in SE2, the average log loss score drops to 0.31, and when Logistic Regression is removed in SE3, the average log loss score jumps up to 0.44. Accuracy scores follow a similar pattern. Obviously, the various strengths and weaknesses of the standard classifiers have a significant effect on the stacked ensemble.

Although SE1 outperforms its parts in terms of log loss and accuracy, it does not outperform either Naïve Bayes or Logistic Regression in terms recall of the minority class (mean 0.41) and precision of the majority class (mean 0.93). Given that SE2 and SE3 both perform worse than SE1 in these metrics, however, it would still seem that there is a benefit to combining multiple models.

In reviewing the decision boundaries (see Figure 10) for stacked ensemble methods, an obvious connection can be drawn between the ensemble and each of its component parts. SE1, which contains all four standard classifiers, has an obvious linear element (a la Logistic Regression), coupled with at least one small spike (a la Decision Tree), some curvature (a la Naïve Bayes), and some unpredictable variance (a la KNN). To further illustrate the point, the ensemble which removes Logistic Regression loses the linear element, and the ensemble which removes Naïve Bayes loses almost all curvature.

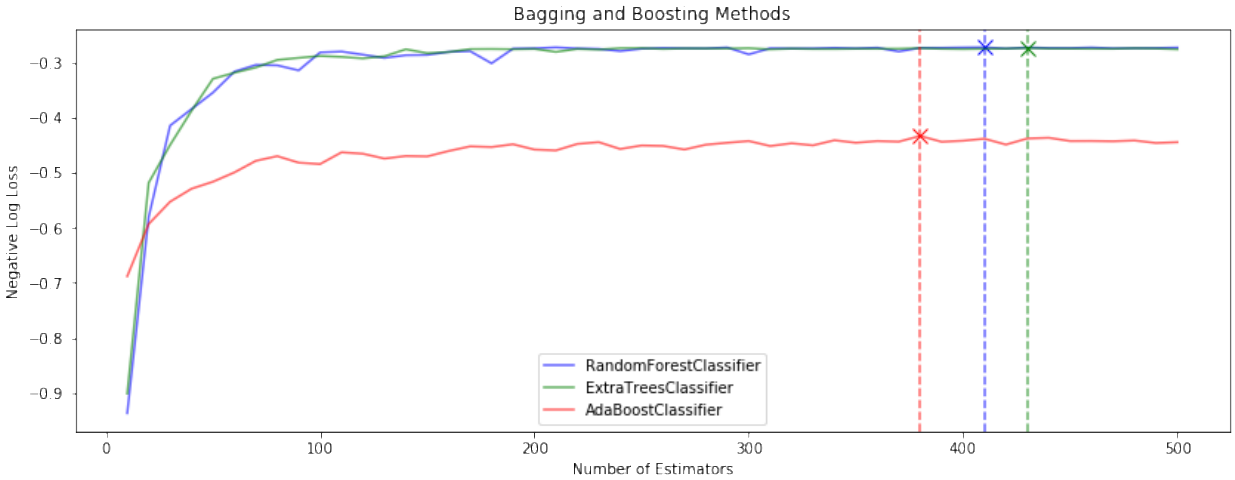
It is important to state that the stacked ensemble algorithms used equal weights for all models. Further work in this field could be to provide a more nuanced weighting mechanism which would better utilize the various strengths and weaknesses of the component parts.

Figure 10: Decision Boundary for Ensemble Classifiers



The bagging and boosting ensemble methods were trained using a small variation in the process described above. Instead of relying on hyperparameter tuning to select the "best" number of estimators to use in each algorithm, the process was paused and the tuning results were displayed graphically (see Figure 11). As can be clearly seen, there is almost no decrease in log loss score past 300 estimators (and that is being conservative) – a plateau has been reached. The standard hyperparameter tuning process would choose unnecessarily high values for the number of estimators, which could degrade the speed of the model and even introduce overfitting. For this reason, the number of estimators was set at 300 for all three algorithms.

Figure 11: Hyperparameter Tuning for Bagging and Boosting Methods



In terms of performance, Random Forest and Complete Random Trees perform remarkably well in terms of log loss, both with an average score of 0.28. AdaBoost fails to outperform the stacked ensemble methods with a mean of 0.47. All three models record the highest demonstrated accuracy scores with a mean of 0.90 and precision scores above 0.90. However, the story changes when reviewing recall of the minority class, in which Random Forest scores an average of 0.14, AdaBoost a 0.12, and Complete Random Trees a dismal 0.07. Only KNN performed worse in this metric. It would seem that, like KNN, these ensemble methods have learned to ignore the minority class.

In reviewing the shape of the decision boundaries for bagging and boosting methods, the high amount of variance that each injects into the process is on display, with all three algorithms building decision spaces that seem to reach out and grab observations. Unfortunately, this same behavior is likely an artifact of overfitting to the majority class. Though it may seem counterintuitive, in practice the smooth lines of Naïve Bayes and Logistic Regression would be preferable to avoid overfitting.

It should be noted that during the training process, an AdaBoost classifier using Naïve Bayes as the base algorithm had to be aborted. Although the initial algorithm could be trained, during the bootstrapping process, the sample weights in the algorithm for random bootstrap samples consistently converged at infinity, causing the gradient boosting to fail and the process to error out. Given the random nature of the bootstrapping process, the reason for this failure could not be immediately pinpointed. Future work in this area would be warranted.

For our final selection, the Naïve Bayes, Logistic Regression, and SE1 algorithms will be reviewed. Each of these algorithms was shown to perform much better than all others in terms of recall of the minority class and precision of the majority class. At this stage, the algorithms will be compared using a weighted F1 score. Although this metric is often used in a way that misrepresents classifier performance, it is used as the final selection criteria specifically because it will average the various precision and recall scores. It has already been shown that accuracy

can mask poor algorithm performance in an unbalanced dataset. A weighted F1 score can also mask poor performance, but not when we have already selected algorithms based on recall of the minority class. Furthermore, to ensure that the weighted F1 score is not being misused in this case, all precision and recall scores will also be reviewed (see Table 8).

Table 8: Comparison of Various Classification Scores for Top Three Algorithms

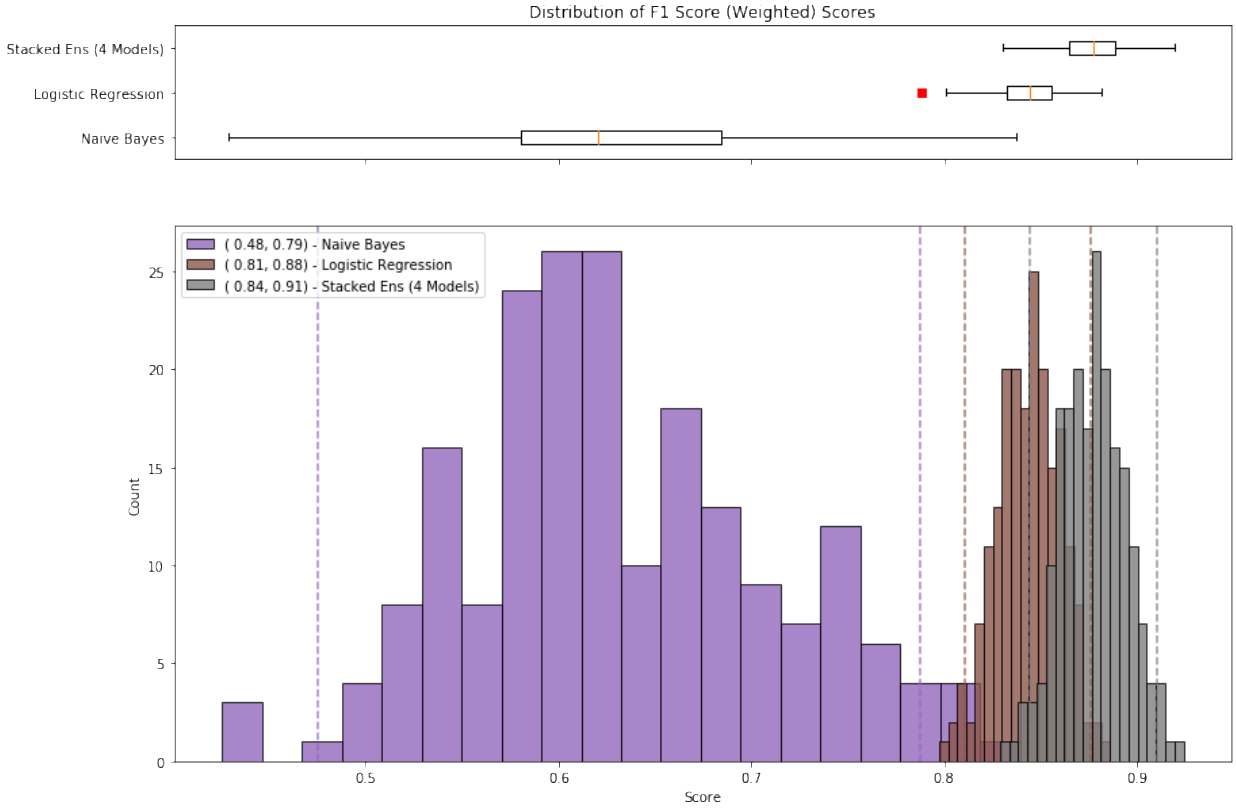
```
Comparison of F1 Score (Weighted) Scores
  Naive Bayes             Mean: 0.63  Std: 0.08  95%Conf: (0.48, 0.79)
  Logistic Regression     Mean: 0.84  Std: 0.02  95%Conf: (0.81, 0.88)
  Stacked Ens (4 Models)  Mean: 0.88  Std: 0.02  95%Conf: (0.84, 0.91)
Comparison of Precision (Fail) Scores
  Naive Bayes             Mean: 0.15  Std: 0.04  95%Conf: (0.07, 0.23)
  Logistic Regression     Mean: 0.31  Std: 0.05  95%Conf: (0.21, 0.41)
  Stacked Ens (4 Models)  Mean: 0.42  Std: 0.09  95%Conf: (0.24, 0.60)
Comparison of Recall (Fail) Scores
  Naive Bayes             Mean: 0.66  Std: 0.07  95%Conf: (0.52, 0.79)
  Logistic Regression     Mean: 0.54  Std: 0.08  95%Conf: (0.38, 0.69)
  Stacked Ens (4 Models)  Mean: 0.41  Std: 0.09  95%Conf: (0.23, 0.58)
Comparison of Precision (Pass) Scores
  Naive Bayes             Mean: 0.93  Std: 0.02  95%Conf: (0.89, 0.97)
  Logistic Regression     Mean: 0.94  Std: 0.01  95%Conf: (0.91, 0.97)
  Stacked Ens (4 Models)  Mean: 0.93  Std: 0.01  95%Conf: (0.90, 0.96)
Comparison of Recall (Pass) Scores
  Naive Bayes             Mean: 0.54  Std: 0.10  95%Conf: (0.34, 0.74)
  Logistic Regression     Mean: 0.86  Std: 0.03  95%Conf: (0.81, 0.91)
  Stacked Ens (4 Models)  Mean: 0.93  Std: 0.02  95%Conf: (0.89, 0.98)
```

Ultimately, it can be shown that while Naïve Bayes and Logistic Regression are the best classifiers in terms of recall of the minority class, in all other scores, the stacked ensemble performs best. While predicting failures may be the most important class prediction, it is inadvisable to base our final selection criteria on such a measure, as it is easily manipulated (i.e. a recall score of 1.00 is achievable by predicting that every student will fail). Instead, a balanced approach to prediction is sought, where the majority of all cases are correctly predicted, but where the minority class is given sufficient attention.

This idea of balance may also be applied to the decision boundaries. The stacked ensemble method was shown to combine the aspects of its component parts, having both linear and curved features, allowing for a more nuanced approach to classification.

At this time, reviewing the distribution of the three top algorithms may help to show their various strengths and weaknesses. In Figure 12, the distribution of weighted F1 scores is shown, along with vertical lines to represent a 95% confidence interval.

Figure 12: **Distribution of Weighted F1 Scores for Top Three Algorithms**



Although there appears to be a distinction between the stacked ensemble algorithm and the others, it is necessary that a statistical test for significance be performed. Using two one-sided T-tests, where the Bonferroni method is used to reduce the chance of making a Type I error, the improvement of the stacked ensemble over Naïve Bayes and Logistic Regression can be shown to be statistically significant at alpha=0.01 (see Table 9). Given this result, we conclude that the stacked ensemble classifier, using Naïve Bayes, Decision Trees, K-Nearest Neighbors, and Logistic Regression classifiers as base models, is the best performing classifier for the dataset in terms of accuracy.

Table 9: T-test Results Comparing Stacked Ensemble to Logistic Regression and Naïve Bayes

```
Ho: Stacked Ens (4 Models) <= Logistic Regression
Ha: Stacked Ens (4 Models) > Logistic Regression
Reject Ho when p/2 < 0.005 and t > 0
       T-statistic: 20.19, P-value: 6.594095102399295e-63
       Reject the null hypothesis.

Ho: Stacked Ens (4 Models) <= Naive Bayes
Ha: Stacked Ens (4 Models) > Naive Bayes
Reject Ho when p/2 < 0.005 and t > 0
       T-statistic: 42.51, P-value: 4.770702386511679e-150
       Reject the null hypothesis.
```

Of the three final algorithms, no algorithm is inherently more scrutible than the others. Naïve Bayes requires a deep understanding of probability theory, dependence, and Gaussian distributions. Logistic Regression requires both an understanding of linear regression, linear algebra, logarithms, and probability theory. A stacked ensemble classifier requires all this and more (ensemble methods, gradient boosting, and whatever is required of the other component algorithms), but the buy-in for the other algorithms is already so high that it is unlikely to make much difference.

In terms of time to build, time to query, and updatability, the stacked ensemble algorithm is only as strong as its weakest link. Whichever component part performs worst in these three metrics, the ensemble will contain that weakness. So, although Naïve Bayes and KNN are very easy to update, Logistic Regression and Decision Trees require the algorithm be trained from scratch to update. Thus, the stacked ensemble must be trained from scratch. Although all but KNN are fast to query, KNN is slow to query, and thus the stacked ensemble is slow to query. Finally, although all four component algorithms are fast to build, the stacked ensemble must first build all four, then would typically apply gradient descent to find the optimal weight for each model, thus significantly slowing the time to build. For the sake of completeness, average fit times and query times for all algorithms are provided in Table 10. Note that the times for stacked ensembles are not provided, as the current algorithms did not work through gradient descent. It is worth pointing out the slow fit times of bagging algorithms, and the extremely slow fit time for AdaBoost (which uses gradient descent). If the stacked ensemble is revised to find the optimal weights for each classifier, it is likely that the fit time would surpass that of AdaBoost.

Table 10: Comparison of Fit Time and Query Time for All Algorithms

Algorithm	Fit Time	Query Time
Decision Tree	0.0204	0.0017
Naive Bayes	0.0061	0.0031
K-Nearest Neighbors	0.0063	1.0106
Logistic Regression	0.0194	0.0023
Stacked Ens (4 Models)	N/A	N/A
Stacked Ens (No GNB)	N/A	N/A
Stacked Ens (No Log)	N/A	N/A
Random Forest	1.9547	0.1552
Complete Random Trees	1.2461	0.2085
AdaBoost (Tree)	7.6927	0.0812

Although the stacked ensemble method is expected to perform relatively poorly in terms of fit time, query time, and updatability, in this particular application it is not a big factor. The School (the client of the algorithm) is unlikely to need to update the algorithm more than once per academic term, or a maximum of four times per year, once all students have taken a new predictive test. A simple pipeline could be built to prepare and re-train the algorithm, and run at the end of each academic term. So, although the stacked ensemble would train slower than other models, there is no expectation of real-time performance or for quick updatability. Moreover, the algorithm would only be queried once every academic term, presumably right after it was trained, so there is also low expectations in that area.

Finally, the stacked ensemble is retrained on the training set and tested against the holdout validation set. As can be seen in Table 11, the stacked ensemble classifier maintains a low log loss score and very high accuracy. The classifier maintains a good balance of precision and recall for both the minority and majority class. The majority of failures are detected, but not at the expense of the majority class.

Table 11: Scores for Stacked Ensemble Classifier against Holdout Validation Set

```
Log Loss: 0.34
Accuracy: 0.89
             precision    recall  f1-score   support

          0       0.48      0.65      0.55        20
          1       0.96      0.92      0.94       172

avg / total       0.91      0.89      0.90       192

Confusion Matrix:
[[ 13   7]
 [ 14 158]]
```

Conclusion

All things considered, the stacked ensemble classifier with component parts of Decision Tree, Naïve Bayes, K-Nearest Neighbors, and Logistic Regression is the best algorithm for the application. The stacked ensemble classifier's attention to recall of the minority class and precision of the majority class, coupled with strong overall accuracy and a high weighted F1 score, outweighs the algorithm's slow time to build, query, and update.

Future work should focus on clustering variables as a means of feature reduction and on training weights for the stacked ensemble using gradient descent. After these improvements, it is expected that the stacked ensemble classifier can be trained to perform even better in future iterations.

Appendix: Code

```
# coding: utf-8

# # Evaluating the Predictive Capabilities of "Predictive
Testing"
# ## Predicting NCLEX Success for Nursing Students
# Author: Alexander Stachniak

# # Modules
# All Python modules used are fairly common and will be
installed as a default for users running Anaconda, with the
possible exception of graphviz.

# In[60]:


# Standard modules
import pandas as pd
import numpy as np
import pickle
import datetime

# Plotting modules
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
import matplotlib.mlab as mlab
# Display plots within Jupyter Notebook
get_ipython().magic('matplotlib inline')
# Set default plot size
plt.rcParams['figure.figsize'] = (14, 5)
# Seaborn used for its excellent correlation heatmap plot
import seaborn as sns
# Used for plotting trees
from sklearn import tree
import graphviz

# SciKit Learn Pre-Processing Methods
from sklearn.preprocessing import Imputer

# SciKit Learn Sampling and Selection Methods
from sklearn.utils import resample
from sklearn.model_selection import train_test_split,
GridSearchCV

# SciKit Learn Models
```

```
from sklearn.decomposition import PCA
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import VotingClassifier,
AdaBoostClassifier, RandomForestClassifier, ExtraTreesClassifier
from sklearn.linear_model import LogisticRegression

# SciKit Learn Metrics
from sklearn.metrics import confusion_matrix,
classification_report, accuracy_score, f1_score, log_loss,
precision_recall_fscore_support

# SciPy Modules
from scipy.stats import ttest_ind, f_oneway, norm,
percentileofscore

# Caution: use only when certain of results
import warnings
warnings.filterwarnings('ignore')


# ## Loading the Data

# In[748]:


with open('NCLEX_data.pickle', 'rb') as file:
    nclex = pickle.load(file)
with open('PA_data.pickle', 'rb') as file:
    pa = pickle.load(file)


# ## Preparing Target Variable
# Our target variable is the result of the NCLEX exam, either
"Pass" or "Fail." As our first data preparation step, we will
ensure that our data is consistent.

# In[749]:


# Set ID as the index
nclex = nclex.set_index('Student ID')
# Sort by the index (ID)
nclex = nclex.sort_index()
```

```
# In[750]:


# View data
nclex.head()


# In[751]:


# As we can see, we need to do account for some variance in how
the target attribute is stored
nclex['Result'].unique().tolist()


# In[752]:


# Convert Result to integer values
result_map = {'FAIL': 0,
              'Fail': 0,
              'fail': 0,
              'PASS': 1,
              'Pass': 1,
              'pass': 1}
nclex['Result'] = nclex['Result'].map(result_map)


# In[753]:


# View the duplicates to ensure we're not going to lose any data
nclex[nclex.index.duplicated(keep=False)]


# In[754]:


# We can use the tilde (inverse) and duplicated method to drop
duplicates
nclex = nclex[~nclex.index.duplicated(keep='first')]


# In[755]:


# View target data
```

```
nclex.head()


# In[756]:


# Review class imbalance
nclex['Result'].value_counts()


# ## Inner Join of Data
# We want to be sure that we are not retaining student data in
one dataframe that does not exist in the others. So, we will
perform an inner join of student IDs, and retain data only for
those IDs moving forward.

# In[757]:


# Multi-part operation:
# 1) Merge statement to produce an inner join on two dataframes
# 2) Keep only the "Student ID" field
# 3) Drop true duplicates
# 4) Reset index from 0
student_ids = nclex.merge(pa, left_index=True, right_on='Student
ID')['Student ID'].drop_duplicates().reset_index(drop=True)


# In[758]:


# Keep only the student IDs that are represented across all
dataframes
nclex = nclex.ix[student_ids]
pa = pa[pa['Student ID'].isin(student_ids)]


# In[759]:


# Total number of observations
len(nclex)


# ## Preparing the Predictive Assessment Data
# The predictive testing product contained multiple categories
of assessments (e.g. Medical/Surgical, Pediatrics, etc.) as well
```

```
as multiple versions within each category containing different
questions. Assessment and Assessment ID identify the assessment,
and Booklet ID is a unique identifier for the test-taker. Date
Taken identifies the date and time that the exam was begun.
#
# For each assessment, several numerical scores are provided:
# * Score (raw score)
# * National Mean
# * National Percentile
# * Program Mean
# * Program Percentile
# * Proficiency Level (meant to indicate level of mastery; is
based on a proprietary algorithm).
#
# Each assessment is further broken down by section, where each
section indicates a theoretical grouping of questions intended
to provide an indication of the level of mastery of specific
sub-topics. Numerical scores are provided both for the overall
assessment and for each individual section.

# In[760]:


pa.head()


# ### Cleaning the Data
# There are a few things we need to do to clean up the data so
that it is easier to work with.

# In[761]:


# Rename section headings that are 'overall' scores using a
lambda function
def overall(Assessment, Section):
    if Section == Assessment:
        return 'Overall'
    else:
        return Section
pa['Section'] = pa.apply(lambda x: overall(x['Assessment'],
x['Section']), axis=1)


# ### Removing Duplicates
# Before we can run any kind of analysis, we need to remove
duplicates from our data. Some duplicates are true duplicates
```

```
(all fields duplicated), so those are dropped without any loss
of data. There also appears to be some malformed data, where
everything is duplicated but all of the numeric scores are lost
(value = 'Nan'). Again, these are dropped without any loss of
data.

# In[762]:


# Drop true duplicates
pa = pa.drop_duplicates()


# In[763]:


# Drop NaN in National Percentile and/or National Mean (only
accompanies malformed duplicates)
pa = pa.dropna(axis=0,how='any',subset=['National Percentile'])
pa = pa.dropna(axis=0,how='any',subset=['National Mean'])


# ### Missing Values
# We will need a plan for dealing with any missing values before
we can run any analysis. First, let's take a look at where our
missing values reside.

# In[764]:


# Which columns contain NaNs?
pa.isnull().sum()


# #### Impute Missing Values for Program Mean
# Missing values in Program Mean are clustered in two specific
assessments. Because we have almost 7,000 NaNs, and we don't
want to have to drop this data when modeling, we will impute the
missing values. Program Mean should be very simple to impute,
however, since it should just be a mean value of all the scores
for a particular section of a particular assessment. In order to
get a little more granular than just replacing NaNs with the
mean of the entire column (which wouldn't account for
differences in sections or assessments), we will iterate over
each section to impute mean values.

# In[765]:
```

```
# First, let's look at the data.
# We make a copy of the df to avoid making unwanted changes
Prog_Nans = pa.copy(deep=True)
Prog_Nans = Prog_Nans[Prog_Nans['Program Mean'].isnull()]
Prog_Nans.head()


# In[766]:


# These NaNs are clustered in the Community Health assessments
Prog_Nans['Assessment'].unique()


# In[767]:


# Closer look at missing values
for assessment in Prog_Nans['Assessment'].unique():
    NaNs = len(pa[(pa['Assessment'] == assessment) &
(pa['Program Mean'].isnull())])
    Total = len(pa[(pa['Assessment'] == assessment)])
    print('Assessment: {}'.format(assessment))
    print('{0:5d} NaNs / {1:5d} Total = {2:6.2f}%
Missing'.format(NaNs, Total, (NaNs/Total)*100))


# In[768]:


# Initialize the mean_imputer (axis=0 means it will impute mean
of column)
mean_imputer = Imputer(missing_values='NaN', strategy='mean',
axis=0)


# In[769]:


# Iterate through each section of each assessment (for localized
means)
for assessment in Prog_Nans['Assessment'].unique():
    for section in Prog_Nans[Prog_Nans['Assessment'] ==
assessment]['Section'].unique():
        # Copy df and remove unnecessary features
```

```
        impute_df = Prog_Nans.copy(deep=True)
        impute_df = impute_df[(impute_df['Assessment'] ==
assessment) & (impute_df['Section'] == section)]
        impute_df = impute_df[['Program Mean', 'Score']]
        # Fit and transform imputer
        impute_df['Program Mean'] =
mean_imputer.fit_transform(impute_df)
        # Add imputed values to df using fillna
        pa['Program Mean'].fillna(value=impute_df['Program
Mean'], inplace=True)


# #### Impute Missing Values for Program Percentile
# Missing values for Program Percentile should be easy to
compute now that we have all of the Program Means. We'll use
scipy's percentileofscore function to generate our percentiles.
Again, we want to calculate percentiles for each section of each
assessment. We want to apply a function over each row of a
Pandas dataframe, but we need to be careful to pass both the
full range of scores and a single score to each call of the
percentile function.

# In[770]:


# Copy df where Program Percentile is NaN
Percentile_Nans = pa.copy(deep=True)
Percentile_Nans = Percentile_Nans[Percentile_Nans['Program
Percentile'].isnull()]


# In[771]:


# Note that in this function we need to take the score for each
observation and determine the percentile against
# the overall range of scores. So the first argument in
percentileofscore is not taken row by row.
def percentile(row):
    'A simple function to calculate percentiles'
    return percentileofscore(impute_df['Score'], row['Score'])


# In[772]:
```

```
# Iterate through each section of each assessment (localized
percentiles)
for assessment in Percentile_Nans['Assessment'].unique():
    for section in Percentile_Nans[Percentile_Nans['Assessment']
== assessment]['Section'].unique():
        # Copy df and remove unnecessary features
        impute_df = Percentile_Nans.copy(deep=True)
        impute_df = impute_df[(impute_df['Assessment'] ==
assessment) & (impute_df['Section'] == section)]
        impute_df = impute_df[['Program Percentile', 'Score']]
        # Apply percentile function
        impute_df['Program Percentile'] =
impute_df.apply(percentile, axis=1)
        # Add imputed values
        pa['Program Percentile'].fillna(value=impute_df['Program
Percentile'], inplace=True)


# In[773]:


# Which columns still contain NaNs?
pa.isnull().sum()


# In[774]:


# Percent of data missing for Proficiency Level
NaNs = pa['Proficiency Level'].isnull().sum()
Total = pa.shape[0]
print('Proficiency Level: {0:5d} NaNs / {1:6d} Total = {2:5.2f}%
Missing'.format(NaNs, Total, (NaNs/Total)*100))


# #### Dropping Proficiency Level
# Proficiency level is not based simply on a cutoff value of the
raw score for a test, but is a proprietary algorithm.
Proficiency levels change over time and are different for
different assessments. Although we could attempt to develop a
model to predict proficiency level based on the other variables,
such a model would ensure a high amount of covariance between
proficiency level and whatever other variables were used in the
model. This covariance would eventually cause problems down the
road. Moreover, we would much rather make use of the raw score
and national mean to create a "Dist to National Mean" (more on
this later). Avoiding any reliance on a proprietary algorithm
```

```
will make our model more robust and resistant to future changes.
For example, should the client elect to use a different
predictive testing vendor, so long as that vendor reports raw
scores and national means, it is likely that we could make it
work with past data. For this reason, as well as the high amount
of missing data, we elect to drop Proficiency Level rather than
impute missing values.

# ## Correct Section Names
# Each exam is broken down into "sections" which represent a
theoretical category of knowledge/learning. There are 285 unique
section names, however, there are quite a few inconsistencies.
Over time, the testing vendor corrected errors and made slight
changes to many sections, such as "Therapeutic Communications"
to "Therapeutic Communication". Because the students' scores on
these sections will represent our explanatory variables, we must
first normalize all of the obvious inconsistencies. After
normalizing, we are left with 212 unique section headings.
#
# We may also wish to map these sections to higher order
concepts. For example, we may be better served by combining
"Spelling" and "Grammar" into "Basic English" to represent a
combination of a student's skill in one particular area.

# In[775]:


# Read in section names and conceptual connections
sections_df = pd.read_csv('section_names.csv')


# In[776]:


sections_df.head()


# In[777]:


def add_to_dict(key, value, user_dict):
    user_dict[key] = value


# In[778]:
```

```
def create_concept_map(key1, key2, value, user_dict):
    # Create dictionary levels as needed
    if key1 not in user_dict.keys():
        user_dict[key1] = {}
        if key2 not in user_dict[key1].keys():
            user_dict[key1][key2] = {}
    # Add to dictionary
    user_dict[key1][key2] = value


# In[779]:


# Initialize empty dictionaries for storage
original_to_corrected = {}
original_to_concept = {}
original_to_root = {}
concept_map = {}
# Iterate through to build dictionaries and new field in df
for dct,col in zip([original_to_corrected, original_to_concept,
original_to_root], sections_df.columns[1:]):
    # Build dictionary
    sections_df.apply(lambda x: add_to_dict(x['Original'],
x[col], dct), axis=1);
    # Add field to df
    pa[col] = pa['Section'].replace(dct)
# Build a concept map
sections_df.apply(lambda x: create_concept_map(x['Root'],
x['Concept'], x['Corrected'], concept_map), axis=1);


# In[780]:


pa.head()


# In[781]:


# Review Number of Fields at Each Level
print('Root Concepts: {0}\nGeneral Concepts: {1}\nCorrected
Sections: {2}'.format(len(pa['Root'].unique()),
len(pa['Concept'].unique()), len(pa['Corrected'].unique())))


# ### Create New Features
```

```
# Intuitively we might expect Distance to the National Mean to
match very well with our overall prediction of NCLEX success. To
understand why, we need to understand more about the NCLEX.
#
# The NCLEX uses Computerized Adaptive Testing (CAT), which
displays questions to candidates in a way that attempts to
understand each candidate's ability while using as few questions
as possible. A standard exam requires all candidates to answer
the same questions, while CAT attempts to always display
questions that a candidate should find challenging (based on his
or her ability).
#
# But how can the NCLEX know how challenging each question is?
The answer is that questions must be tested ahead of time across
a wide sample of respondents. This is the first instance where
something similar to a "National Mean" plays a role in
determining pass rate.
#
# The second instance is in the actual decision rules of whether
a candidate passes or fails. The passing standard is based, at
least partially, on past NCLEX results, sample test-takers, and
education readiness of high-school graduates interested in
nursing. All of these are related to a National Mean.

# In[782]:


# Calculate distance to National Mean
pa['Dist to National Mean'] = pa.apply(lambda x: x['Score'] -
x['National Mean'],axis=1)


# # Reshape Data and Combine Scores
# We would like to use the National Mean as our baseline score
in all cases (note that the National Mean is dependent on the
Assessment, and may change over time; we are interested in
whatever the current mean is).
#
# We have a significant amount of inconsistencies that can arise
in our data, especially over time:
# * Each student will start out having only taken a small
portion of the total number of Assessments, and will not
complete all Assessments until they graduate (at which point, it
is too late for remediation).
# * The educational program can change (e.g., offer Assessments
in a different order, or skip some entirely).
```

```
# * An individual student could take courses part-time, or out
of order, causing the Assessments to be taken out of order.
# * An individual student could take multiple versions of an
assessment (as practice or as remediation).
# We are looking for a method of aggregating results that is
resistant to these differences. In order for our model to be
useful, we need to be able to make a prediction for any student,
regardless of their place in the program and regardless of the
order they have taken assessments.
#
# The solution is to start every student at a baseline of 0.0
for each concept. Whenever a student takes an assessment in
which he or she is graded on a particular concept, we will
calculate the distance to the national mean for that particular
assessment and concept, then average their distance scores
together as their new baseline.

# In[783]:


# Make a deep copy of the data
df = pa.copy(deep=True)


# Groupby is an example of a Panda's function that has been
optimized using row calculations. When we take the mean of "Dist
to National Mean" while grouping by Student ID and Concept, we
accomplish the same thing that we would if we iterated over each
Student ID, then iterated over each Concept, averaged the
matching values in the dataframe, then added to a new dataframe.
The difference is that Groupby does all of these calculations as
row calculations and doesn't have to perform costly look-ups,
increasing the speed exponentially.

# In[784]:


# Groupby, Average, and Unstack
df = df['Dist to National Mean'].groupby([df['Student ID'],
df['Concept']]).mean().unstack()


# In[785]:


# Divide by 100 to standardize the data. Fill NaNs with 0.0
df = df.divide(100, fill_value=0.0)
```

```
# In[786]:


df.head()


# ## Reviewing for Correlations and Covariance
# The various concepts can be shown to have very low correlation
to one another. In the plot below, we notice some significant
banding, which we can attribute to the variables being listed
alphabetically. Similar concepts tend to have similar names, but
even still, correlations are very low.
#
# This is also demonstrated in the Principal Component Analysis
(PCA) example, in which we would need over 50 principal
components to account for 80% of the variance. These results
suggest that we do not need to be concerned about covariance,
but also that dimensionality reduction will not be a viable
option.

# In[787]:


# Calculate correlation matrix
corr = df.corr()
# Create a mask for the upper triangle
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
# Set up plot
f, ax = plt.subplots(figsize=(11,9))
# Generate a colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)
# Plot
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=1, square=True,
linewidths=.5, cbar_kws={"shrink": .5}, ax=ax)


# In[788]:


# Take a look at the actual correlation values
corr


# ### PCA Example
```

```
# The following example prints the explained variance ratio for
each principal component, as well as plotting the cumulative
explained variance ratio. As we can see, PCA is not a good fit
for this data.

# In[789]:


# Principal Component Analysis
pca = PCA(n_components=None)
pa_pca = pca.fit_transform(df)
print('Explained Variance')
for pc in range(len(pca.explained_variance_ratio_)):
    print('PC{0}: {1:5.2f}%'.format(pc+1,
pca.explained_variance_ratio_[pc]*100))
# Cumulative variance
cum_var_exp = np.cumsum(pca.explained_variance_ratio_)
# Plot it
plt.bar(range(1,len(pca.explained_variance_ratio_)+1),
pca.explained_variance_ratio_, alpha=0.5, align='center',
label='individual explained variance')
plt.step(range(1,len(pca.explained_variance_ratio_)+1),
cum_var_exp, where='mid', label='cumulative explained variance')
plt.ylabel('Explained Variance Ratio')
plt.xlabel('Principal Components')
plt.legend(loc='best')
plt.show()


# # Pickle Data for Future Use
# Though not a necessary step, pickling the data at this point
will allow us to avoid re-running the cleaning process above
each time we revisit our data for modeling. This step allows us
to "freeze" our data at this point in time.

# In[790]:


# Pickle the data for future use
with open('vars.pickle', 'wb') as f:
    pickle.dump(df, f, pickle.HIGHEST_PROTOCOL)
with open('target.pickle', 'wb') as f:
    pickle.dump(nclex, f, pickle.HIGHEST_PROTOCOL)
with open('concept_map.pickle', 'wb') as f:
    pickle.dump(concept_map, f, pickle.HIGHEST_PROTOCOL)
```

```
# # Load Data

# In[2]:


with open('vars.pickle', 'rb') as file:
    df = pickle.load(file)
with open('target.pickle', 'rb') as file:
    target = pickle.load(file)
with open('concept_map.pickle', 'rb') as file:
    concept_map = pickle.load(file)


# # Data Distribution

# In[3]:


def plot_histograms (df, **kwargs):
    '''Function to plot multiple distributions of variables from
a Pandas dataframe.

    @ Parameters:
    ---------------
    df: a Pandas dataframe or series

    @ kwargs (optional):
    ---------------
    num_cols: the number of columns to plot in subplot (integer)
    ax_width: the width of each column (integer)
    bin_size: the number of bins to use in the histograms
(integer)
    color: a color value for all plots
    xlim: a minimum and maximum range to apply to all histograms
(list with two values)
    title_size: a valid matplotlib text size (defaul: 'x-small')
    '''
    # Pull kwargs, if offered
    num_cols = kwargs.pop('num_cols', 5)
    ax_width = kwargs.pop('ax_width', 3)
    bin_size = kwargs.pop('bin_size', 20)
    color = kwargs.pop('color', 'blue')
    xlim = kwargs.pop('xlim', [-1.0, 1.0])
    title_size = kwargs.pop('title_size', 'x-small')

    # Determine basic sizes
    columns = df.columns
```

```
    num_items = len(columns)
    num_rows = int(np.ceil(num_items / num_cols))

    # Set plot size
    plot_width = num_cols * ax_width
    plot_height = num_rows * ax_width
    plt.rcParams['figure.figsize'] = (plot_width, plot_height)

    # Build array of subplots
    f, axarr = plt.subplots(num_rows, num_cols)

    item_locator = 0
    # Iterate through rows and columns
    for r in range(num_rows):
        for c in range(num_cols):
            if item_locator < num_items:
                # Gather data
                temp_data = df[columns[item_locator]]
                # Gather mean and standard deviation
                mu = np.mean(temp_data)
                sigma = np.std(temp_data)
                # Prepare Title
                title = '{0}\nMu: {1:.2f} | Sigma:
{2:.2f}'.format(columns[item_locator], mu, sigma)
                # Plot histogram
                N, bins, patches = axarr[r, c].hist(temp_data,
bins=bin_size, normed=1, color=color, edgecolor='black',
alpha=0.6)
                # Add 'best fit' line
                y = mlab.normpdf(bins, mu, sigma)
                l = axarr[r, c].plot(bins, y, 'r--',
linewidth=1)
                # Plot options
                axarr[r, c].set_title(title, size=title_size)
                axarr[r, c].set_xlim(xlim)
                # Increment item_locator
            item_locator += 1
    plt.tight_layout()
    plt.show()


# In[4]:


# Plot a few sample distributions to get a feel for the data
```

```
sample_cases = ['Accident/Error/Injury Prevention', 'Basic
Science', 'Crisis Intervention', 'Cardiovascular Disorders',
'Family Planning', 'Information Technology']
plot_histograms(df[sample_cases], num_cols=3,
title_size='small')


# # Examining Feature Importance Using Random Forest

# In[29]:


# Convert columns to numpy array for indexing purposes
feat_list = np.array(x_train.columns)
# Fit forest
forest = RandomForestClassifier(n_estimators=10000, n_jobs=-1)
forest.fit(x_train, y_train)
# Gather feature importances
importances = forest.feature_importances_


# In[30]:


# Gather ranked indices of feature importances
indices = np.argsort(importances)[::-1]
for f in range(len(feat_list)):
    print('{0:3d} ({1:.4f}) {2}'.format(f + 1,
importances[indices[f]], feat_list[indices[f]]))


# # Helper Functions and Classes

# To determine the best parameters to use in our models, we will
follow the same pattern each time, identifying a list of
hyperparameters we want to test and then calling GridSearchCV to
tune those hyperparameters. The function below is a simple
wrapper for this procedure so that we can avoid repeating
ourselves as much as possible.

# In[4]:


def tune_parameters (x, y, clf, params, **kwargs):
    '''A simple wrapper for GridSearchCV to speed up repetitive
process.
```

```
    @ Parameters:
    ---------------
    x: array of explanatory variables
    y: array of target variable
    clf: an initialized Sci-Kit Learn classifier
    params: dictionary of parameters over which to search

    @ **kwargs (optional):
    ----------------------
    folds: number of cross-validation folds to use (see
GridSearchCV docs for other options)
    scoring_metric: a scorer to evaluate hyperparameters (if
None, will use model default)

    @ Returns:
    ----------------------
    best_params_: a dictionary of the best parameters
    cv_results_: a dictionary of all results
    '''

    # Gather data from kwargs, if supplied; otherwise, use
defaults
    folds = kwargs.pop('folds', 10)
    scoring_metric = kwargs.pop('scoring_metric', None)

    # Initialize grid search instance
    gs = GridSearchCV(clf, param_grid=params, cv=folds,
scoring=scoring_metric, n_jobs=-1, refit=False)
    # Fit grid search model
    gs.fit(x, y)

    return gs.best_params_, gs.cv_results_


# The class below is a bootstrap method designed as an iterator
that will work with GridSearchCV. GridSearchCV is designed to be
able to take custom cross validation methods, but it only
provides k-fold methods out of the box. Building our own
iterator allows us to use bootstrapping during hyperparameter
tuning. This is not strictly necessary, and will slow down the
grid search somewhat, but would expect it to produce more
accurate results, especially as we increase m, the number of
times to sample with replacement.

# In[5]:
```

```
class Bootstrap_Indices:
    '''A cross-validation iterator that will work with
GridSearchCV.
    The iterator uses Sci-Kit Learn's resample method to sample
with replacement and
    uses Numpy's in1d method to determine which indices will be
in the test set. Also
    relies on Numpy's arange method.

    @ Parameters:
    ---------------
    n: number of observations
    m: number of times to sample with replacement

    @ Returns:
    ---------------
    an iterable of idx_train, idx_test indices
    '''
    def __init__(self, n, m):
        self.n = n
        self.m = m
        self.i = 0

    def __len__(self):
        return self.m

    def __iter__(self):
        idx = np.arange(self.n)
        while self.i < self.m:
            self.i += 1
            idx_train = resample(idx)
            mask = np.in1d(idx, idx_train, invert=True)
            idx_test = idx[mask]
            yield idx_train, idx_test


# After we tune our hyperparameters, we need to fit our model
using bootstrapping and a scoring function of choice to then
compare our various models and test for significance

# In[6]:


def bootstrap_model (x, y, model, bootstrap_iterator):
    '''
    Function fits a given model using bootstrapping method and
returns an array of scores
```

```
    using the given scoring function. User must also pass the
number of epochs, i.e., the
    number of bootstrapped samples to take and fit the model on.

    The Sci-Kit Learn function, resample, provides a consistent
way to take a random sample
    of our data. The defaults are to use replacement, return
same shape as input, and use
    the random state instance of numpy.random

    @Parameters:
    --------------------
    x: explanatory variables array
    y: target array
    model: an instantiated Sci-Kit Learn model (i.e., parameters
already set)
    bootstrap_iterator: an iterator that implements bootstrap
indexing
    scoring_function: a valid Sci-Kit Learn scorer. User could
pass their own scoring function,
        but it should be built using .make_scorer and should
include the following parameters:
            y_true (array of true target results)
            y_pred (array of predicted results)
            OR
            y_score (array of probabilities of true class)
        If scorer does not include these parameters in the first
two positions, error will be raised.
    pred_method: an indication of the input needed by
scoring_function (either 'labels' or 'probabilities')

    @Returns:
    --------------------
    score_array: an array of sequential scoring values (length =
epochs)
    '''
    # Initialize scoring arrays
    accuracy_array = np.zeros(len(bootstrap_iterator))
    f1_array = np.zeros(len(bootstrap_iterator))
    log_loss_array = np.zeros(len(bootstrap_iterator))
    precision_array_1 = np.zeros(len(bootstrap_iterator))
    recall_array_1 = np.zeros(len(bootstrap_iterator))
    fscore_array_1 = np.zeros(len(bootstrap_iterator))
    support_array_1 = np.zeros(len(bootstrap_iterator))
    precision_array_0 = np.zeros(len(bootstrap_iterator))
    recall_array_0 = np.zeros(len(bootstrap_iterator))
    fscore_array_0 = np.zeros(len(bootstrap_iterator))
```

```
    support_array_0 = np.zeros(len(bootstrap_iterator))
    # Iterate through bootstrap iterator
    counter = 0
    for idx in iter(bootstrap_iterator):
        # Resample
        x_train = x.iloc[idx[0]]
        y_train = y.iloc[idx[0]]
        x_test = x.iloc[idx[1]]
        y_test = y.iloc[idx[1]]
        # Fit model
        model.fit(x_train, y_train)
        # Make predictions
        pred_array = model.predict(x_test)
        prob_array = model.predict_proba(x_test)
        # Score model
        accuracy_array[counter] = accuracy_score(y_test,
pred_array)
        f1_array[counter] = f1_score(y_test, pred_array,
average='weighted')
        log_loss_array[counter] = log_loss(y_test, prob_array)
        precision_array_1[counter], recall_array_1[counter],
fscore_array_1[counter], support_array_1[counter] =
precision_recall_fscore_support(y_test, pred_array, pos_label=1,
average='binary')
        precision_array_0[counter], recall_array_0[counter],
fscore_array_0[counter], support_array_0[counter] =
precision_recall_fscore_support(y_test, pred_array, pos_label=0,
average='binary')
        counter += 1

    return accuracy_array, f1_array, log_loss_array,
precision_array_1, recall_array_1, fscore_array_1,
support_array_1, precision_array_0, recall_array_0,
fscore_array_0, support_array_0


# In[7]:


def comparison (*args, **kwargs):
    '''This function prints a comparison of scores.

    @ Parameters:
    -------------
    *args: Any number of arrays for which you want to compare
confidence intervals
    **kwargs:
```

```
        C: Confidence Interval (default = 0.95)
        labels: ordered list of labels to apply to ordered args
        scorer: name of the scoring function being compared.'''
    C = kwargs.pop('C', 0.95)
    labels = kwargs.pop('labels', ['Model {}'.format(x) for x in
range(len(args))])
    scorer = kwargs.pop('scorer', None)
    max_l = len(max(labels, key=len))

    print('Comparison of {} Scores'.format(scorer))
    for arg, label in zip(args, labels):
        # Calculate confidence interval
        conf = norm.interval(C, loc=arg.mean(), scale=arg.std())
        print('  {0:{1}}  Mean: {2:.2f}  Std: {3:.2f}  95%Conf:
({4:.2f}, {5:.2f})'.format(label, max_l, arg.mean(), arg.std(),
conf[0], conf[1]))


# In[33]:


def plot_distributions(*args, **kwargs):
    '''This function plots the distribution and confidence
interval of all arrays passed to it.

    @ Parameters:
    --------------
    *args: Any number of arrays for which you want to visualize
the distribution
    **kwargs:
        C: Confidence Interval (default = 0.95)
        num_bins: number of bins to use in histogram
(default=20)
        labels: ordered list of labels to apply to ordered args
        colors: ordered list of colors to apply to ordered args
        x_lim: list representing the left and right x limits
        plot_width: an integer value representing width in
inches
        plot_height: an integer value representing height in
inches
        limit_outliers: if True, histograms will ignore any
outliers (1.5 * IQR). Default=False.
        alpha: alpha value for plot opacity.
        scorer: name of the scoring function being compared.
        conf_lines: if 'True' will plot vertical lines to show
confidence intervals
    '''
```

```
    # Gather data from kwargs, if supplied; otherwise, use
defaults
    C = kwargs.pop('C', 0.95)
    num_bins = kwargs.pop('num_bins', 20)
    labels = kwargs.pop('labels', ['Model {}'.format(x) for x in
range(len(args))])
    colors = kwargs.pop('colors',
cm.rainbow(np.linspace(0,1,len(args))))
    x_lim = kwargs.pop('x_lim', [None, None])
    plot_width = kwargs.pop('plot_width', 14)
    plot_height = kwargs.pop('plot_height', 5)
    limit_outliers = kwargs.pop('limit_outliers', False)
    alpha = kwargs.pop('alpha', 0.6)
    scorer = kwargs.pop('scorer', None)
    conf_lines = kwargs.pop('conf_lines', False)

    # Set plot dimensions
    plt.rcParams['figure.figsize'] = (plot_width, plot_height)

    # Create subplots
    fig, (ax1, ax2) = plt.subplots(2, sharex=True,
gridspec_kw={"height_ratios": (.20, .80)})

    # Plot boxplots
    ax1.boxplot(args, vert=False, sym='rs', labels=labels)
    # Remove y-axis and set overall title (looks better than a
sup title)
    #ax1.axes.get_yaxis().set_visible(False)
    if scorer:
        ax1.set_title('Distribution of {}
Scores'.format(scorer))
    else:
        ax1.set_title('Distribution of Scores')

    # Plot histograms
    for arg, color, label in zip(args, colors, labels):
        # Determine whisker_range for histograms
        if limit_outliers == True:
            p_25 = np.percentile(arg, 25)
            p_75 = np.percentile(arg, 75)
            iqr = p_75 - p_25
            whisker_range = (p_25 - (iqr * 1.5), p_75 + (iqr *
1.5))
        else:
            whisker_range = None
        # Calculate confidence interval
```

```
        conf = norm.interval(C, loc=arg.mean(), scale=arg.std())
        # Prepare label
        l = '({0: .2f}, {1:.2f}) - {2}'.format(conf[0], conf[1],
label)
        # Plot histogram
        N, bins, patches = ax2.hist(arg, bins=num_bins,
range=whisker_range, color=color, edgecolor='black',
alpha=alpha, label=l)
        # Plot confidence intervals
        if conf_lines == True:
            for val in conf:
                ax2.axvline(x=val, color=color, linestyle='--',
alpha=alpha)

    # Set plot attributes
    plt.legend(loc='1')
    plt.xlabel('Score')
    plt.ylabel('Count')
    plt.xlim(x_lim)
    plt.show()


# In[9]:


def plot_decision_boundaries (X, Y, x_lab, y_lab, *args,
**kwargs):
    '''Plots the decision boundaries of any number of
classifiers.
    The following code is originally based on the visualization
technique
    presented by Sci-Kit Learn, at
    http://scikit-
learn.org/stable/auto_examples/ensemble/plot_voting_decision_reg
ions.html.
    Significant modifications have been made.

    @ Parameters:
    ---------------
    X: explanatory variables
    Y: target feature
    x_lab: the explanatory variable to plot on the x-axis
    y_lab: the explanatory variable to plot on the y-axis
    *args: list of initialized classifiers

    @ **kwargs (optional):
    ---------------
```

```
    offset: +/- amount to extend mesh grid
    labels: list of classifier names
    num_cols: number of columns in plot
    ax_width: width of each column
    ax_height: height of each column
    '''
    # Pull kwargs, if offered
    offset = kwargs.pop('offset', 0.05)
    labels = kwargs.pop('labels', ['Model {}'.format(x) for x in
range(len(args))])
    num_cols = kwargs.pop('num_cols', 2)
    ax_width = kwargs.pop('ax_width', 7)
    ax_height = kwargs.pop('ax_height', 5)

    # Determine basic sizes
    num_items = len(args)
    num_rows = int(np.ceil(num_items / num_cols))

    # Set plot size
    plot_width = num_cols * ax_width
    plot_height = num_rows * ax_height
    plt.rcParams['figure.figsize'] = (plot_width, plot_height)

    # Ensure Target Classification is an array
    cls = Y.as_matrix()
    # Gather data
    cols = [x_lab, y_lab]
    x = X[x_lab]
    y = X[y_lab]

    # Set up mesh grid for contour predictions
    x_min, x_max = x.min() - offset, x.max() + offset
    y_min, y_max = y.min() - offset, y.max() + offset
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))

    # Prepare subplots in grid
    f, axarr = plt.subplots(num_rows, num_cols, sharex='col',
sharey='row')

    item_locator = 0
    # Iterate through rows and columns
    for r in range(num_rows):
        for c in range(num_cols):
            # Fit algorithm
            args[item_locator].fit(X[cols], Y)
            # Make predictions across mesh grid space
```

```
            z = args[item_locator].predict(np.c_[xx.ravel(),
yy.ravel()])
            z = z.reshape(xx.shape)
            # Plot contour and scatter
            axarr[r, c].contourf(xx, yy, z, alpha=0.4)
            axarr[r, c].scatter(x, y, c=cls)
            axarr[r, c].set_title(labels[item_locator])
            item_locator += 1

    plt.show()


# In[102]:


def compare_scores (*args, alternate='less', **kwargs):
    '''Function performs a series of t-tests, comparing the
first element in *args to all
    other elements. To avoid increasing the likelihood of making
a Type I error, we use the
    Bonferroni method, which divides the alpha value by the
number of hypotheses tested.

    The T-test function provided in SciPy always produces a two-
side test, but the user
    can choose a one-sided test by supplying the necessary
parameter.

    We would reject the null hypothesis when:
        * H0: a <= b, Ha: a > b : reject H0 when p/2 < alpha and
t > 0
        * H0: a >= b, Ha: a < b : reject H0 when p/2 < alpha and
t < 0
        * H0: a = b, Ha: a != b : reject H0 when p < alpha

    @ Parameters:
    ---------------
    *args: list of score arrays, where first in list is compared
to all others
    alternate: value of "less", "more", or "unequal" - will
determine test that is run.
        User should assume "less" means that they are testing if
the scores of the first
        argument are less than the scores of all other
arguments.

    @ **kwargs (optional):
```

```
    ---------------
    alpha: alpha value for testing (function will automatically
update if required for
        one-sided test and to take into account Bonferroni
method)
    labels: a list of names of the score arrays
    '''
    # Gather keyword arguments, if any
    alpha = kwargs.pop('alpha', 0.05)
    labels = kwargs.pop('labels', ['Model {}'.format(x) for x in
range(len(args))])

    # Determine appropriate signs for hypotheses
    if alternate == 'less':
        hyp_sign_1 = '>='
        hyp_sign_2 = '<'
        q = 2
    elif alternate == 'more':
        hyp_sign_1 = '<='
        hyp_sign_2 = '>'
        q = 2
    elif alternate == 'unequal':
        hyp_sign_1 = '='
        hyp_sign_2 = '!='
        q = 1

    # Determine the Bonferroni correction based on number of
hypotheses to test
    m = len(args) - 1
    bonferroni = alpha / m

    for arg, label in zip(args[1:], labels[1:]):
        # Calculate t and p
        t_statistic, p_value = ttest_ind(args[0], arg)
        hyp_state = 'Ho: {0} {1} {2}\nHa: {0} {3} {2}\nReject Ho
when p/2 < {4} and t {3} 0'.format(labels[0], hyp_sign_1, label,
hyp_sign_2, bonferroni)
        hyp_test = '\tT-statistic: {0:.2f}, P-value:
{1}'.format(t_statistic, p_value)
        # Determine whether to reject null hypothesis
        if p_value / q < bonferroni:
            if alternate == 'less' and t_statistic < 0:
                res = 'Reject'
            elif alternate == 'more' and t_statistic > 0:
                res = 'Reject'
            elif alternate == 'unequal':
                res = 'Reject'
```

```
            else:
                res = 'Fail to reject'
        else:
            res = 'Fail to reject'
        hyp_result = '\t{0} the null hypothesis.'.format(res)
        print('\n'.join([hyp_state, hyp_test, hyp_result]) +
'\n')


# # Reserve 20% of Data for a Holdout Validation Set
# Before we begin hyperparameter tuning, we reserve 20% of our
data as a holdout validation set. Because we have a class
imbalance, we stratify the split.

# In[12]:


x_train, x_test, y_train, y_test = train_test_split(df,
target['Result'], test_size=0.2, random_state=25,
stratify=target['Result'])


# ### Naming Scheme

# In[128]:


# Extended names for each of our models
model_names = {'adt': 'AdaBoost (Tree)',
               'rfc': 'Random Forest',
               'tree': 'Decision Tree',
               'knn': 'K-Nearest Neighbors',
               'gnb': 'Naive Bayes',
               'log': 'Logistic Regression',
               'etc': 'Complete Random Trees',
               'se1': 'Stacked Ens (4 Models)',
               'se2': 'Stacked Ens (No GNB)',
               'se3': 'Stacked Ens (No Log)'}


# In[68]:


# Standard color scheme
model_colors = {'adt': 'C0',
               'rfc': 'C1',
               'tree': 'C2',
```

```
                'knn': 'C3',
                'gnb': 'C4',
                'log': 'C5',
                'etc': 'C6',
                'se1': 'C7',
                'se2': 'C8',
                'se3': 'C9'}


# # Standard Classifiers
# ### Define Parameters and Initialize Models

# In[14]:


# Decision Tree Classifier
tree_params = {'criterion': ['gini', 'entropy'],
               'max_depth': [4, 5, 6, 7, 8],
               'min_samples_leaf': [3, 5, 7]}
tree_init = DecisionTreeClassifier()


# In[15]:


# Gaussian Naive Bayes Classifier
gnb_params = {}
gnb_init = GaussianNB()


# In[16]:


# K-Nearest Neighbors Classifier
knn_params = {'n_neighbors': [3, 5, 7, 9, 11],
              'p': [1,2,3],
              'n_jobs': [-1]}
knn_init = KNeighborsClassifier()


# In[17]:


# Logistic Regression Classifier
log_params = {'C':[1, 10, 100, 1000],
              'tol':[1e-1,1e-2,1e-3,1e-4],
              'penalty':['l1', 'l2'],
```

```
              'class_weight': ['balanced'],
              'n_jobs': [-1]}
log_init = LogisticRegression()


# # Tune Hyperparameters
# For the majority of our classifiers, we will perform
hyperparameter tuning using GridSearchCV, then using the best
set of parameters we will build our model using bootstrapping in
preparation for comparing each model.

# In[22]:


# Combine various parameters for each model into lists
model_names = ['tree', 'gnb', 'knn', 'log']
model_params = [tree_params, gnb_params, knn_params, log_params]
model_inits = [tree_init, gnb_init, knn_init, log_init]


# In[23]:


# Set the scoring metric to use
# Note that we use "negative log loss" because we need a metric
that we can maximize
scoring_param = 'neg_log_loss'


# In[24]:


# Set variables for bootstrapping iterators
bootstrap_n = x_train.shape[0]
bootstrap_m_1 = 20
bootstrap_m_2 = 200


# In[25]:


# Initialize dictionary for storage
models = {}
# Iterate through models
for clf, params, init in zip(model_names, model_params,
model_inits):
    # Save data and parameters
```

```
    models[clf] = {}
    models[clf][clf + '_params'] = params
    models[clf][clf + '_init'] = init
    # Initialize first bootstrap iterator
    bootstrap_1 = Bootstrap_Indices(bootstrap_n, bootstrap_m_1)
    # Tune hyperparameters
    models[clf][clf + '_best_params'], models[clf][clf +
'_cv_results'] = tune_parameters(x_train, y_train, init, params,
folds=bootstrap_1, scoring_metric=scoring_param)
    # Update each of the model parameters with best values
    models[clf][clf + '_model'] = models[clf][clf + '_init']
    for key, value in models[clf][clf + '_best_params'].items():
        models[clf][clf + '_model'].__setattr__(key, value)
    # Initialize second bootstrap iterator
    bootstrap_2 = Bootstrap_Indices(bootstrap_n, bootstrap_m_2)
    # Gather an array of scores for significance testing using
bootstrapping
    models[clf]['accuracy_scores'], models[clf]['f1_scores'],
models[clf]['log_loss_scores'], models[clf]['precision_1'],
models[clf]['recall_1'], models[clf]['fscore_1'],
models[clf]['support_1'], models[clf]['precision_0'],
models[clf]['recall_0'], models[clf]['fscore_0'],
models[clf]['support_0'] = bootstrap_model(x_train, y_train,
models[clf][clf + '_model'], bootstrap_2)



# # Stacked Ensemble Classifiers
# ### Define Parameters and Initialize Models

# In[26]:


# Using all four basic classifiers
se1_params = [('gnb', models['gnb']['gnb_model']),
              ('tree', models['tree']['tree_model']),
              ('knn', models['knn']['knn_model']),
              ('log', models['log']['log_model'])]

# Ignoring Naive Bayes
se2_params = [('tree', models['tree']['tree_model']),
              ('knn', models['knn']['knn_model']),
              ('log', models['log']['log_model'])]

# Ignoring Logistic Regression
se3_params = [('gnb', models['gnb']['gnb_model']),
              ('tree', models['tree']['tree_model']),
```

```
                ('knn', models['knn']['knn_model'])]


# In[27]:


# Initialize various parameters for each model
model_names = ['se1', 'se2', 'se3']
model_params = [se1_params, se2_params, se3_params]
# Iterate through models
for clf, params, init in zip(model_names, model_params,
model_inits):
    # Save data and parameters
    models[clf] = {}
    models[clf][clf + '_params'] = params
    # Initialize model (we use 'soft' voting to use
probabilities rather than predictions)
    models[clf][clf + '_model'] = VotingClassifier(params,
voting='soft', n_jobs=-1)
    # Initialize bootstrap iterator
    bootstrap_2 = Bootstrap_Indices(bootstrap_n, bootstrap_m_2)
    # Fit model using bootstrapping
    models[clf]['accuracy_scores'], models[clf]['f1_scores'],
models[clf]['log_loss_scores'], models[clf]['precision_1'],
models[clf]['recall_1'], models[clf]['fscore_1'],
models[clf]['support_1'], models[clf]['precision_0'],
models[clf]['recall_0'], models[clf]['fscore_0'],
models[clf]['support_0'] = bootstrap_model(x_train, y_train,
models[clf][clf + '_model'], bootstrap_2)


# # Bagging and Boosting Classifiers

# ### Tuning Parameters for Bagging and Boosting Models
# With the bagging and boosting techniques we will use, the
number of estimators will tend to have diminishing returns after
a certain point. Although hyperparameter tuning will choose the
"best" number of estimators, this is likely to be far more than
is necessary in practice, which will increase our total training
time as well as have the potential to introduce overfitting. For
this reason, we will pause after hyperparameter tuning to review
and manually choose the number of estimators for our final
model.

# In[55]:
```

```
# Random Forest Classifier
rfc_params = {'n_estimators': np.arange(10, 501, 10)}
rfc_init = RandomForestClassifier(n_jobs=-1,
max_features='sqrt')


# In[56]:


# Complete Random Trees Classifier
etc_params = {'n_estimators': np.arange(10, 501, 10)}
etc_init = ExtraTreesClassifier(n_jobs=-1, max_features='sqrt')


# In[57]:


# AdaBoost Classifier with Decision Tree
base_tree = DecisionTreeClassifier(max_depth=4,
min_samples_leaf=7)
adt_params = {'n_estimators': np.arange(10, 501, 10)}
adt_init = AdaBoostClassifier(base_estimator=base_tree)


# In[58]:


# Initialize various parameters for each model
model_names = ['rfc', 'etc', 'adt']
model_params = [rfc_params, etc_params, adt_params]
model_inits = [rfc_init, etc_init, adt_init]


# In[33]:


# Iterate through models
for clf, params, init in zip(model_names, model_params,
model_inits):
    # Save data and parameters
    models[clf] = {}
    models[clf][clf + '_params'] = params
    models[clf][clf + '_init'] = init
    # Initialize bootstrap iterator
    bootstrap = Bootstrap_Indices(bootstrap_n, bootstrap_m_1)
    # Tune hyperparameters
```

```
    models[clf][clf + '_best_params'], models[clf][clf +
'_cv_results'] = tune_parameters(x_train, y_train, init, params,
folds=bootstrap, scoring_metric=scoring_param)


# In[61]:


# Review results of hyperparameter tuning
fig, ax = plt.subplots()
# Iterate over various models
for clf, color in zip(model_names, ['blue', 'green', 'red']):
    # Plot log loss
    ax.plot(np.arange(10, 501, 10), models[clf][clf +
'_cv_results']['mean_test_score'], label=models[clf][clf +
'_init'].__class__.__name__, color=color, alpha=0.6)
    # Plot a vertical line and a marker to indicate "best
parameter"
    ax.axvline(models[clf][clf +
'_best_params']['n_estimators'], linestyle='--', color=color,
alpha=0.6)
    ax.plot(models[clf][clf + '_best_params']['n_estimators'],
models[clf][clf + '_cv_results']['mean_test_score'].max(),
marker='x', color=color, markersize=10)
# Set plot options
plt.title('Bagging and Boosting Methods')
plt.ylabel('Negative Log Loss')
plt.xlabel('Number of Estimators')
plt.legend()
plt.show()


# For all of our algorithms, we see almost no gains past 300
estimators (and that is being conservative). Although
GridSearchCV would choose much higher numbers for all but one of
the algorithms, it is obvious that we are reaching a plateau of
performance past which we will not see any noticeable gains but
are likely to degrade the speed of our model or even overfit.
For this reason, we will cap all algorithms at 300 estimators.

# In[35]:


# Choose our parameters
rfc_param = {'n_estimators': 300}
etc_param = {'n_estimators': 300}
adt_param = {'n_estimators': 300}
```

```
chosen_params = [rfc_param, etc_param, adt_param]


# In[36]:


model_names = ['rfc', 'etc', 'adt']
chose_params = [rfc_param, etc_param, adt_param]
model_inits = [rfc_init, etc_init, adt_init]
# Iterate through models
for clf, param, init in zip(model_names, chosen_params,
model_inits):
    # Update each of the model parameters with chosen values
    models[clf][clf + '_model'] = models[clf][clf + '_init']
    for key, value in param.items():
        models[clf][clf + '_model'].__setattr__(key, value)
    # Initialize second bootstrap iterator
    bootstrap_2 = Bootstrap_Indices(bootstrap_n, bootstrap_m_2)
    # Gather an array of scores for significance testing using
bootstrapping
    models[clf]['accuracy_scores'], models[clf]['f1_scores'],
models[clf]['log_loss_scores'], models[clf]['precision_1'],
models[clf]['recall_1'], models[clf]['fscore_1'],
models[clf]['support_1'], models[clf]['precision_0'],
models[clf]['recall_0'], models[clf]['fscore_0'],
models[clf]['support_0'] = bootstrap_model(x_train, y_train,
models[clf][clf + '_model'], bootstrap_2)


# ### Save Data for Future Use
# Given the amount of time it takes to perform hyperparameter
tuning and bootstrapping, we will save all of our results to
avoid repeating the calculations

# In[37]:


# Pickle the data for future use
with open('models.pickle', 'wb') as f:
    pickle.dump(models, f, pickle.HIGHEST_PROTOCOL)


# ### Open Data

# In[15]:
```

```
with open('models.pickle', 'rb') as file:
     models = pickle.load(file)


# # Algorithm Results

# In[36]:


v_1 = ['tree', 'gnb', 'knn', 'log', 'se1', 'se2', 'se3', 'rfc',
'etc', 'adt']
scores = [models[clf]['log_loss_scores'] for clf in v_1]
names = [model_names[clf] for clf in v_1]
colors = [model_colors[clf] for clf in v_1]
plot_distributions(*scores, labels=names, scorer='Log Loss',
colors=colors, num_bins=20, x_lim=[0.0, 15.0], alpha=0.8,
plot_height=10, limit_outliers=True)


# In[18]:


comparison(*scores, labels=names, scorer='Log Loss')


# In[37]:


v_1 = ['tree', 'gnb', 'knn', 'log', 'se1', 'se2', 'se3', 'rfc',
'etc', 'adt']
scores = [models[clf]['accuracy_scores'] for clf in v_1]
names = [model_names[clf] for clf in v_1]
colors = [model_colors[clf] for clf in v_1]
plot_distributions(*scores, labels=names, scorer='Accuracy',
colors=colors, num_bins=20, alpha=0.8, plot_height=10,
limit_outliers=True)


# In[20]:


comparison(*scores, labels=names, scorer='Accuracy')


# In[38]:
```

```
v_1 = ['tree', 'gnb', 'knn', 'log', 'se1', 'se2', 'se3', 'rfc',
'etc', 'adt']
scores = [models[clf]['recall_0'] for clf in v_1]
names = [model_names[clf] for clf in v_1]
colors = [model_colors[clf] for clf in v_1]
plot_distributions(*scores, labels=names, scorer='Recall
(Fail)', colors=colors, num_bins=20, alpha=0.8, plot_height=10,
limit_outliers=True)


# In[39]:


comparison(*scores, labels=names, scorer='Recall (Fail)')


# In[40]:


v_1 = ['tree', 'gnb', 'knn', 'log', 'se1', 'se2', 'se3', 'rfc',
'etc', 'adt']
scores = [models[clf]['precision_1'] for clf in v_1]
names = [model_names[clf] for clf in v_1]
colors = [model_colors[clf] for clf in v_1]
plot_distributions(*scores, labels=names, scorer='Precision
(Pass)', colors=colors, num_bins=20, alpha=0.8, plot_height=10,
limit_outliers=True)


# In[41]:


comparison(*scores, labels=names, scorer='Precision (Pass)')


# # Plot Decision Boundaries

# In[26]:


# Determine variables to plot against
x_lab = 'Psychosocial Integrity'
y_lab = 'Ante/Intra/Postpartum and Newborn Care'


# In[43]:
```

```
# Decision boundaries for standard classifiers
v_1 = ['tree', 'gnb', 'knn', 'log']
clfs = [models[clf][clf + '_model'] for clf in v_1]
names = [model_names[clf] for clf in v_1]
plot_decision_boundaries(x_train, y_train, x_lab, y_lab, *clfs,
labels=names)


# In[44]:


# Decision boundaries for ensemble methods
v_1 = ['se1', 'se2', 'se3', 'rfc', 'etc', 'adt']
clfs = [models[clf][clf + '_model'] for clf in v_1]
names = [model_names[clf] for clf in v_1]
plot_decision_boundaries(x_train, y_train, x_lab, y_lab, *clfs,
labels=names)


# # Export Decision Tree Graph

# In[45]:


# Review the parameters chosen in hyperparameter tuning
models['tree']['tree_best_params']


# In[46]:


# Train the algorithm
models['tree']['tree_model'].fit(x_train, y_train)


# In[47]:


# Export tree graph (displays poorly within Jupyter Notebook)
dot_data = tree.export_graphviz(models['tree']['tree_model'],
out_file=None,
                          feature_names=x_train.columns,
                          class_names=['Fail', 'Pass'],
                          filled=True, rounded=True,
                          special_characters=True)
graph = graphviz.Source(dot_data)
```

```
graph.render('DecisionTree')


# # Final Comparisons

# In[111]:


v_1 = ['gnb', 'log', 'se1']
scores = [models[clf]['f1_scores'] for clf in v_1]
names = [model_names[clf] for clf in v_1]
colors = [model_colors[clf] for clf in v_1]
plot_distributions(*scores, labels=names, scorer='F1 Score
(Weighted)', colors=colors, num_bins=20, conf_lines=True,
alpha=0.8, plot_height=10, limit_outliers=True)


# In[72]:


comparison(*scores, labels=names, scorer='F1 Score (Weighted)')


# In[104]:


scores = [models[clf]['precision_0'] for clf in v_1]
comparison(*scores, labels=names, scorer='Precision (Fail)')


# In[105]:


scores = [models[clf]['recall_0'] for clf in v_1]
comparison(*scores, labels=names, scorer='Recall (Fail)')


# In[106]:


scores = [models[clf]['precision_1'] for clf in v_1]
comparison(*scores, labels=names, scorer='Precision (Pass)')


# In[107]:
```

```
scores = [models[clf]['recall_1'] for clf in v_1]
comparison(*scores, labels=names, scorer='Recall (Pass)')


# # Perform T-Tests

# In[114]:


compare_scores(*scores[::-1], labels=names[::-1],
alternate='more', alpha=0.01)


# # Evaluate Time to Fit and Time to Query

# In[139]:


v_1 = ['tree', 'gnb', 'knn', 'log', 'se1', 'se2', 'se3', 'rfc',
'etc', 'adt']
max_l = len(max(model_names.values(), key=len))
print('Comparison of Fit Time and Query Time:')
print('{0:<{1}} {2:>10}\t{3:>10}'.format('Algorithm', max_l,
'Fit Time', 'Query Time'))
for clf in v_1:
    if clf + '_cv_results' in models[clf].keys():
        idx = models[clf][clf +
'_cv_results']['rank_test_score'].argmin()
        fit_time = models[clf][clf +
'_cv_results']['mean_fit_time'][idx]
        score_time = models[clf][clf +
'_cv_results']['mean_score_time'][idx]
        print('{0:<{1}}
{2:>10.4f}\t{3:>10.4f}'.format(model_names[clf], max_l,
fit_time, score_time))
    else:
        fit_time = 'N/A'
        score_time = 'N/A'
        print('{0:<{1}}
{2:>10}\t{3:>10}'.format(model_names[clf], max_l, fit_time,
score_time))


# # Train Algorithm and Report Scores against Holdout Set

# In[147]:
```

```
clf = models['se1']['se1_model']
clf.fit(x_train, y_train)
y_pred = clf.predict(x_test)
y_pred_prob = clf.predict_proba(x_test)
print('Log Loss: {0:.2f}'.format(log_loss(y_test, y_pred_prob)))
print('Accuracy: {0:.2f}'.format(accuracy_score(y_test,
y_pred)))
print(classification_report(y_test, y_pred))
print('Confusion Matrix:\n{}'.format(confusion_matrix(y_test,
y_pred)))


# # Train Production Model on All Data

# In[145]:


models['se1']['production_model'] = models['se1']['se1_model']
models['se1']['production_model'].fit(df, target['Result']);
```