Tree Based Ensembles for Predicting Survival from Thoracic Surgery

Samuel Jackson, Aberystwyth University

I. Introduction

Thoracic surgery is a major invasive surgery involving operating on the lungs of a patient. The authors of ref. [1] collected several pieces of possibly relevant data on a number of patients who went on to have thoracic surgery. The data also includes a record of whether a given patient survived for longer than one year after the surgery. This paper looks at using a reduced subset of the features and patients from the dataset in [1] to classify patients based on whether or not they will survive for one year after the surgery. This paper compares four different classifiers: Random Forests [2], Extremely Randomised Trees [3], AdaBoost [4], and Gradient Boosting [5].

The format of the rest of this paper is structured as follows: section II outlines the preprocessing steps performed on the dataset and describes the classifiers used. Section III presents the performance of the classifiers on the dataset and explores different feature sets. The effects of resampling the dataset are also shown. Section IV discusses the results and presents possible justification for the performance based on the properties of the classifier and dataset. Finally, a summary and of possible future directions are discussed in section V.

TABLE I
Description of Columns in the Thoracic Surgery Dataset

Column	Type	Description
DGN	Nominal	Diagnosis: Specific combination of ICD-10 codes for primary and secondary as well multiple tumours if any (DGN3, DGN2, DGN4, DGN6, DGN5, DGN8, DGN1)
PRE4	Numeric	Forced vital capacity (FVC)
PRE5	Numeric	Volume that has been exhaled at the end of the first second of forced expiration (FEV1)
PRE6	Ordinal	Performance status - Zubrod scale (PRZ2, PRZ1, PRZ0)
PRE7	Nominal	Pain before surgery (T,F)
PRE8	Nominal	Haemoptysis before surgery (T,F)
PRE9	Nominal	Dyspnoea before surgery (T,F)
PRE10	Nominal	Cough before surgery (T,F)
PRE11	Nominal	Weakness before surgery (T,F)
PRE14	Ordinal	T in clinical TNM - size of the original tumour, from OC11 (smallest) to OC14 (largest) (OC11, OC14, OC12, OC13)
PRE17	Nominal	Type 2 DM - diabetes mellitus (T,F)
PRE25	Nominal	Peripheral arterial diseases (PAD) (T,F)
PRE30	Nominal	Smoking (T,F)
PRE32	Nominal	Asthma (T,F)
AGE	Numeric	Age at surgery
Risk1Y	Nominal	1 year survival period - (T)rue value if died (T,F) (Class Label)

II. Methods

A. Dataset and Preprocessing

The thoracic surgery dataset used consists of 16 predictors and 300 instances. Table I gives a description of each predictor derived from the original UCI dataset repository [6]. The dataset includes a mixture of both categorical (nominal and ordinal) and continuous data. The final (17^{th}) column of the dataset is the binary class label with value 0 if the patient survived and 1 if they died within one year of surgery.

Several initial observations can be made about dataset prior to any preprocessing steps. One key thing to note about the dataset as a whole is that there is a slight imbalance between the two classes. Only 28% of the dataset is of the positive class (28% of patients died). While this imbalance is not extreme, it can have repercussions for the performance of the classifiers. The accuracy paradox [7] states that a classifier with high accuracy can be built from highly imbalanced data by always predicting the negative class.

The predictor PRE32 is zero for all of the patients in the training dataset. This predictor therefore has zero variation and will not help to discriminate between instance. PRE32 is therefore discarded during preprocessing.

PRE5 appears to have some extreme values. PRE5 corresponds to the FEV1 measure. This would suggest that some patients have an unusually high forced expiration volume. Also, all of the outliers are of the same class. This could cause the classifiers to fit to noise rather than to properly generalise. These instances were therefore removed from the dataset. No reduction in performance was witnessed during cross validation for all classifiers after their removal.

The feature DGN is a nominal categorical predicator. This feature was transformed into series of new features via one hot encoding. Each new predictor is a binary feature which is one if the patient falls into the category and zero otherwise. The original DGN feature is drop after the 7 new binary features are created.

Finally, after all preprocessing is complete, a Random Forest is trained on the dataset (with default parameters) and the resulting variable importance measure is computed. Any features with a variable importance of zero are dropped. The variable importance of the preprocessed features (before any are dropped) is shown in figure 1.

B. Classifiers

Four classifiers were chosen for use on the dataset. The four classifiers used are Random Forests, Gradient Boosting, AdaBoost, and Extra Trees. The implementations of all four classifiers are taken directly from the scikit-learn library

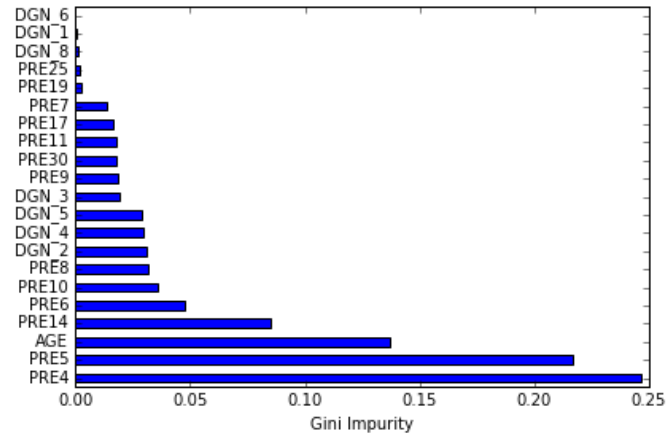

Fig. 1. Feature importance for all of the features after preprocessing. Variables with the prefix DNG_ are the one-hot encoded versions of the original DNG variable. The numerical predictors PRE4, PRE5, and AGE are shown to be the most important to survival prediction.

[8]. All of these methods are ensemble methods. Ensemble methods compose together multiple weak learners to produce a single strong learner. In this paper, all of the algorithms used are also tree based (although this is not necessarily the case for Gradient Boosting and AdaBoost). This means that for each classifier the base learner is a decision tree.

Random forests [2] are perhaps the simplest method of the four. Random forests are simply a collection of n decision trees which are individually trained on the data. The "random" in the name comes from the fact that each tree is trained on both a random sample of the dataset (tree bagging) and also on a random subset of the features (feature bagging). The decision tree weak learners typically overfit to the data. However because the results of all trees are averaged over the variance in the final model's prediction is significantly reduced. The random element for both training instances and features is used to prevent many highly correlated trees from occurring and therefore reduce overfitting.

Extremely Randomised trees [3] (also called Extra-Trees) takes the randomisation aspect of random forests one step further. Random features are still used but training data is not bootstrapped. This aims to reduce model bias. Additionally a random split for each feature in the subset of features is chosen instead of the just computing the optimal feature and split combination. This has the benefit of potentially lowering the variance in the model.

The AdaBoost algorithm [4] is a generalised method for combining the performance of many weak learners. AdaBoost stands for "adaptive boosting" and boosting is a core component in the training stage. AdaBoost works by first fitting a weak learner to the training dataset and classifying each instance. The error in the classification can be used to re-weight each example in the dataset. Misclassified samples are therefore more likely to be classified correctly in future iterations. Likewise, instances that are correctly classified can be weighted much lower, as they are easier to correctly identify. AdaBoost can be seen as an additive method in that each tree is built on the error of the previous one. Adaboost does not necessarily have to be used with decision trees, but in this paper only decision tree based AdaBoost is considered.

Gradient Boosting Machines [5] also use a boosting based approach to learning. Like AdaBoost they are an additive method that iteratively fits a collection of weak learners. Where the two differ is in the way that instances are "weighted". The weighting function in AdaBoost can be seen as a special type of loss function. The gradient of a differentiable loss function can be use to steer the search towards the optimum decision function. In gradient boosting machines the new function added to the mix is the one which is the closest to parallel with the negative gradient of the observed data.

These algorithms were chosen to showcase a broad range of different ensemble algorithms. Ensemble methods often outperform a single strong learner due to the diversity present in the model. All of the models are also decision tree based which work well with a mixture of continuous and discrete values like those present in the dataset.

Two common techniques used in training ensemble algorithms are bagging and boosting. This paper compares two algorithms based on boosting (AdaBoost and Gradient Boosting) and one based on bagging (Random Forests). Extra-Trees also utilise bagging for feature subsets. These seem like good candidates based on the dataset for two reasons: 1) bagging can be used address the imbalance in the dataset by equally resampling each of the datasets and 2) there doesn't seem to be a clear separation between classes in the dataset which potentially makes boosting a good technique as it should help to push the algorithm towards classifying the difficult missed examples.

C. Hyperparameters & Tuning

Before all experiments were carried out on the dataset, the hyper-parameters of each classifier were tuned to hopefully achieve optimum performance. The values and number hyper-parameters are dependant both on the implementation of the classifier and the dataset itself. If there is more than one hyper-parameter for a classifier (as is the case with all classifiers used here) then ideally combinations of all hyper-parameters should be explored. Sadly, this means that the space of potential hyper-parameters choices explodes as the number of hyper-parameters increases.

Due to the relatively small size of the dataset, the space of potential parameters for each classifier is explored using a grid search. In a grid search, a selection of hyper-parameter values are explicitly enumerated. Each potential value for a hyper-parameter is tested in combination with every other hyper-parameter value. The speed of the grid search is bearable due to the classifier being relatively quick to train on this small dataset. The performance of a set of parameters was evaluated using stratified k-fold cross validation with ROC AUC as the scoring metric.

For Random Forests a single grid search was performed over the tree parameters *max_depth*, *max_features*, *min_samples_split* and *min_samples_leaf*. *max_depth* and *max_features* were trained over the range 2 - 20 in steps

of 3. *min_samples_split* and *min_samples_leaf* were trained over all values in the range 1 - 5. The number of trees used was fixed to 50 during this search. This is because a small number of trees will be quick to train (and hence the search will complete faster). Generally speaking the performance of the forest should improve as the number of trees increases, so this can be trained afterwards.

The results of tuning the tree parameters showed that *min_samples_split* and *min_samples_leaf* should be set to 1. This seems logical due to the small number of positive samples. *max_depth*, *max_features* were optimised as 16 and 5 respectively. These seem reasonable given the low number of predictive features and the fact that trees in random forests should typically overfit (hence the large maximum depth). After this trial another grid search was performed to find the optimum number of trees over the range 50 - 500 in steps of 50. This suggested that 100 trees should be used.

The training procedure for Extremely Random Trees was identical to Random Forests with the results being similar. After tuning 200 trees were used and *max_features* was set to 16 and *max_depth* set to 19.

Adaboost was tuned by fixing the maximum depth of the decision tree and performing a grid search over the number of trees (50 - 1000 in steps of 50) and the learning rate (with values 0.1, 0.5, 0.01, and 0.005) together. This was repeated for multiple values of the maximum depth (tested with values 2, 4, and 6). After tuning 400 trees were used with a *learning_rate* of 0.5 and *max_depth* of 4.

Gradient Boosting has many parameters that need to be explored, many of which can interact with one another and tuning in the wrong order can lead to poor results. The number of parameters can also be awkward to train due to the speed of training. The tuning procedure for gradient boosting was therefore as follows:

- Fix all of the parameters to be reasonable initial guesses and fix the learning rate to be quite high (0.1).
- Find the optimum number of estimators for the given leaning rate (searched over the range 20 - 150 in steps of 10).
- Tune the tree based parameters *max_depth* and *min_samples_split* (searched over the ranges 5-16 in steps of 2 and 1-20 in steps of 3 respectively).
- Tune *max_features* (5-20 in steps of 2)
- Tune the subsample ratio (values: 0.6, 0.7, 0.75, 0.8, 0.85, and 0.9).
- Finally using all previously tuned parameters increase the number of estimators while simultaneously decreasing the learning rate.

The final values for the tuned parameters used are shown in table II.

TABLE II
TUNED PARAMETERS FOR THE GRADIENT BOOSTING CLASSIFIER

Parameter	Value
learning_rate	0.01
max_depth	9
max_features	11
min_samples_leaf	1
min_samples_split	7
n_estimators	1000
subsample	0.8

TABLE III
MEAN F1, F2 AND F0.5 SCORES FOR ALL FOUR CLASSIFIERS OVER 10 ROUNDS OF 5-FOLD CROSS VALIDATION.

	RandomForest	ExtraTrees	GradientBoost	AdaBoost
F1	0.597687	0.588262	0.624193	0.624870
F2	0.519876	0.539351	0.570319	0.574151
F0.5	0.713685	0.656862	0.698756	0.691422

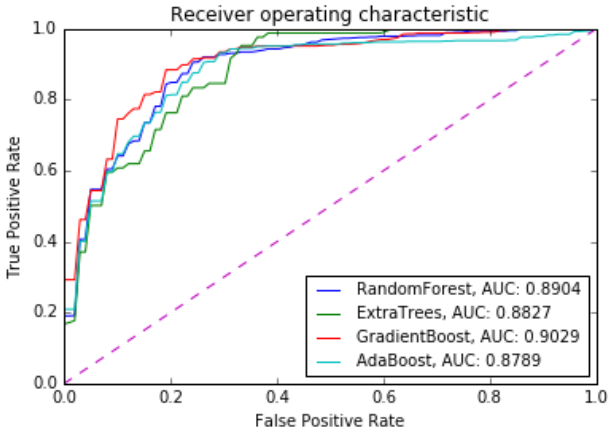

Fig. 2. Mean ROC curves and the mean AUC for all four classifiers over 10 rounds of 5-fold cross validation. All four classifiers perform similarly, with Extra Trees producing the best AUC. All four curves are slightly skewed towards the right of the plot, suggestive of poor recall. The F2 score (table III and figure 3 confirms this.

III. RESULTS

A. Performance Evaluation

Each classifier in section II-B was trained using stratified 5-fold cross validation. Stratification was performed to ensure that there was a representative sample of positive classes in each fold. For all classifiers cross validation was repeated ten times, each with a new set of folds to ensure consistent results.

Figure 2 shows the mean ROC curve and mean AUC for each of the classifiers after cross validation. The performance of each classifier appears to be very similar. Notably the ROC curve for each type of classifier is shifted to the right of the graph, suggesting that they all exhibit a low recall rate.

Table III and figure 3 confirm this indication. Table III shows the F measure with a β parameter of 1, 2, and 0.5. Figure 3 shows a bar chart of the F2 scores in table III. The performance of all classifiers measured with the F2 score (which weights recall more highly than precision) is much lower in comparison to the F0.5 and F1 scores. This further confirms that all classifiers have a problem with recall.

B. Feature Engineering

In addition to the preprocessing steps outlined in II-A several combinations of new features were generated from the existing predictors. Firstly, as a large portion of the features

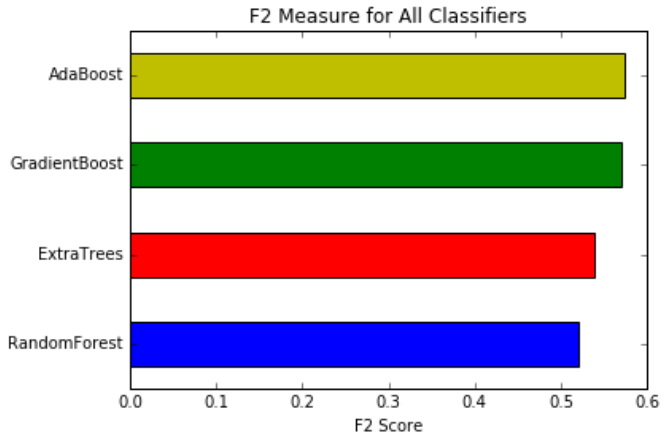

Fig. 3. Bar chart showing the F2 score for all classifiers taken from table III.

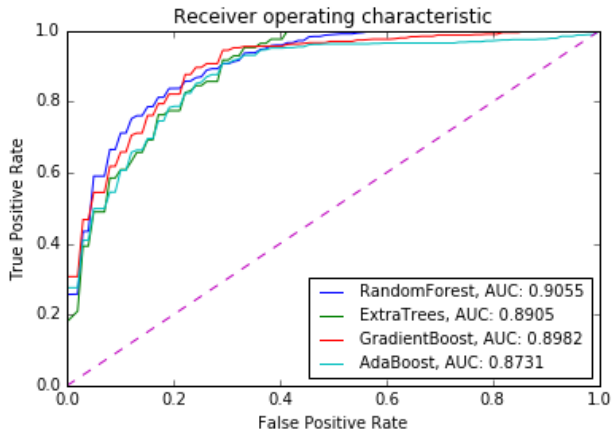

Fig. 5. ROC curves for each of the classifiers with the spirometry features.

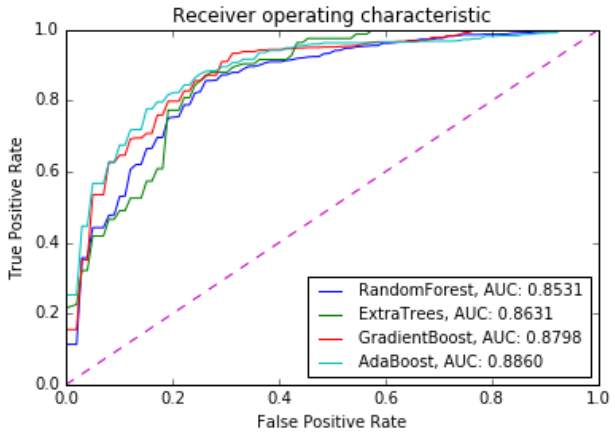

Fig. 4. ROC curves for each of the classifiers with the additional binary features. Performance is notably worse compared to the initial run.

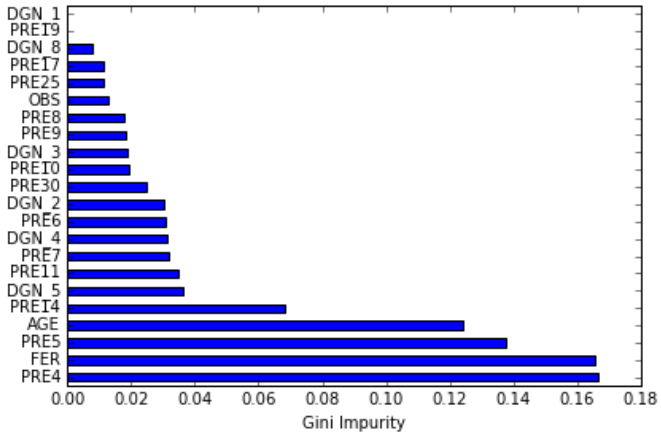

Fig. 6. Variable importance for each of the features with the spirometry features included.

are binary, a set of new features were created based on logical binary operators. The creation of the binary features is as follows: all pairs of binary features are enumerated. From each pair three new features are created by combining the pair using logical OR, AND and XOR.

Figure 4 shows the ROC curves for the same dataset but with the additional binary features. appended. Table IV shows the corresponding F-scores for each classifier. From these results it is easy to see that all of the classifiers appear to perform worse with the new features. This may be due to their already limited contribution and that fact that the additional dimensionality is hindering progress.

The second modification to the original dataset is to create a couple of new features called FER and OBS. FER is the FEV1/FVC ratio which is spirometry measurement defined as $(FEV1/FVC) \cdot 100$ [9]. It is interpreted as the percentage of FVC expelled in the first second of a forced expiration. A ratio value below <70% can be suggestive of an obstructive disease. Using this information another feature (OBS) is generated from the ratio. OBS is a binary feature with value 1 when a patient has a ratio <70%.

From figure 5 it can be seen that their is a slight improvement over the original ROC AUC scores using these additional spirometry features. This is backed up by plotting the feature importance obtained from training a random forest on the dataset (see 1). The two new features, particularly the FER feature are providing useful training information. This seems sensible as the new features are just combinations of existing well performing features.

TABLE IV
F SCORES FOR THE DATASET INCLUDING BINARY FEATURES

	RandomForest	ExtraTrees	GradientBoost	AdaBoost
F1	0.532500	0.571757	0.611276	0.652767
F2	0.459203	0.517610	0.543585	0.598086
F0.5	0.644672	0.646104	0.707112	0.728810

TABLE V
F SCORES FOR THE DATASET INCLUDING SPIROMETRY FEATURES

	RandomForest	ExtraTrees	GradientBoost	AdaBoost
F1	0.566668	0.607952	0.601360	0.619702
F2	0.484577	0.562422	0.540406	0.560452
F0.5	0.696171	0.670030	0.687827	0.699803

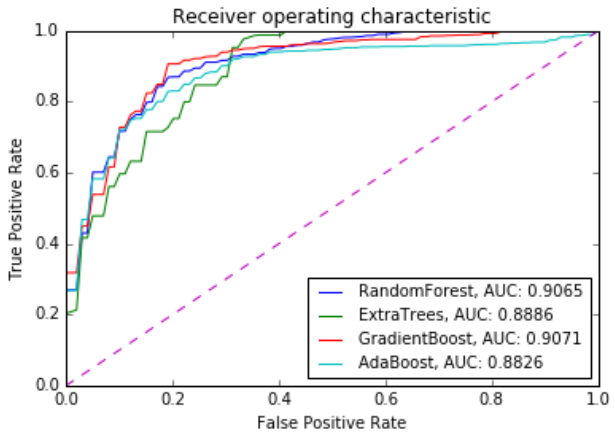

Fig. 7. ROC curves for each of the classifiers with the polynomial combination features.

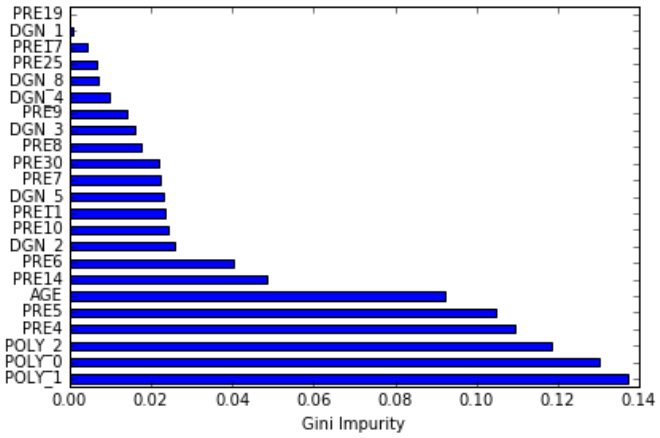

Fig. 8. Variable importance for each of the features with the polynomial combination features included.

Motivated by the results of the previous test, a selection of new features were created from all order 2 polynomial combinations of the two best predictors: PRE4 and PRE5. This means that the new features are of the form a^2, ab, b^2 where a and b are PRE4 and PRE5 respectively.

This polynomial combination led to the best in the feature engineering results across all classifiers under cross validation. Figure 7 shows the ROC curves with the additional features included. The F-scores for each classifier are show in table VI. The contribution of the new features can be seen in the variable importance plot (figure 8).

TABLE VI
F SCORES FOR THE DATASET INCLUDING POLYNOMIAL COMBINATION FEATURES

	RandomForest	ExtraTrees	GradientBoost	AdaBoost
F1	0.596854	0.586917	0.617784	0.658825
F2	0.508636	0.539122	0.563169	0.608835
F0.5	0.733492	0.653059	0.692061	0.723445

TABLE VII
MEAN F1, F2 AND F0.5 SCORES FOR ALL CLASSIFIERS AFTER MONTE CARLO CROSS VALIDATION WITH SMOTE RESAMPLING WITH A RATIO OF 0.8

	RandomForest	ExtraTrees	GradientBoost	AdaBoost
F1	0.608007	0.603050	0.628132	0.609344
F2	0.636271	0.643457	0.652385	0.640834
F0.5	0.585051	0.570976	0.608498	0.583632

C. Dataset Balancing

As mentioned in section II-A the thoracic surgery dataset is class imbalanced with only 28% of the dataset being of the positive class. One technique to combat class imbalance is to resample the dataset to put more emphasis on the known positive examples. A popular technique for resampling data is SMOTE [10]. SMOTE rebalances a dataset by creating new synthetic training to balance out the majority class. SMOTE is typically combined with under-sampling of the majority class to produce a final dataset that is re-weighted in favour of the minority class.

The results for the classifiers in part III-A shows that they have lower recall than precision. Rebalancing the dataset should show a decrease in precision and an increase in recall rate. This can be desirable in a dataset such as this where recall may be more important than precision. It is probably more desirable overestimate the number people who are likely to die from surgery than to achieve better precision.

SMOTE datasets cannot be validated using conventional k-fold cross validation. This is because the testing fold would contain synthetically generated training examples which are obviously not representative of the ground truth. Instead, in order to achieve a representative sample of performance, "Monte Carlo" cross-validation [11] is used. Before a any resampling is applied, the data set is randomly split into a training and testing set. The split is stratified according to the class labels. All reported experiments use and 80/20 split. Resampling is then applied to the training dataset only, with the testing set remaining untouched. This process is then repeated for the desired number of iterations and the resulting performance measures are averaged. In all experiments the number of iterations performed was 50.

Figure 9 shows ROC curve and mean AUC scores for each of the classifiers using SMOTE with a resampling ratio of 0.8. Table VII shows the F1, F2, and F0.5 scores for each of the classifiers. Comparing this table to the results of III shows a clear difference in the F2 score. Recall weighted performance is now better both than F1 and F0.5. This improvement comes at the cost of a decrease in both the AUC and F0.5 measures. Increasing the oversampling ratio or under-sampling the majority class accentuates this effect.

IV. DISCUSSION

The experiments in section III have shown a variety of different approaches to predicting surgery survival with ensemble methods. Some of the best performance was achieved using the just the basic preprocessing steps outlined in section II-A.

Looking at the initial performance evaluation (figure 2) it can be seen that all classifiers performed reasonable similarly

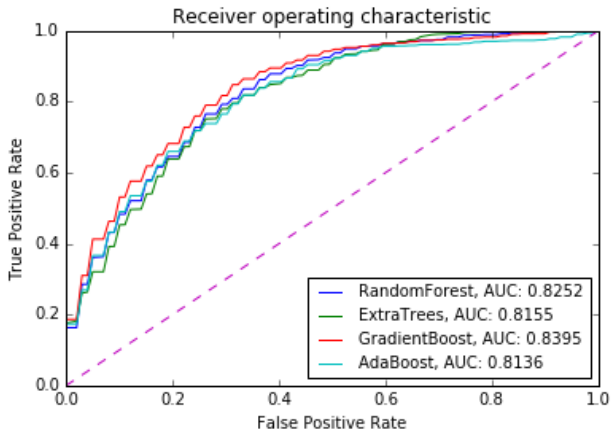

Fig. 9. ROC curves for all four classifiers with SMOTE oversampling with a ratio of 0.8. Each curve represents the average over 50 iterations of Monte Carlo validation. The ROC curves for all classifiers are less skewed compared to figure 2.

with Gradient Boosting narrowly coming out ahead. The weakest performer was AdaBoost. Looking at the F-scores for each of the classifiers (table III) is more informative. All classifiers can be seen to perform comparable. Each performed weaker under the F2 measure which weights recall more highly than precision. It is the higher recall rate which is primarily driving the improvement of Gradient Boosting over the other classifiers in this trial.

Motivated by this baseline evaluation, this paper explored alternative feature representations through "feature engineering". The first attempt was to create combinations of binary features from the existing dataset. This actually lead to worse results compared to the original preprocessing steps. None of the new binary features significantly contributed new information for the algorithm to work with. This probably meant that the increase in dimensionality out weighed any small gains delivered by the new representation. Each tree in the ensembles will only look at a limited number of features, so adding lots of redundant features is only likely to decrease performance while increasing training time. Note that an attempt was made to reduce the number of features by discarding the n weakest features using both variable importance and PCA, but neither method improved the results of this test.

The second feature engineering experiment was to a couple of features generated from spirometry theory. Both the FER and OBS feature suggested are directly derived from the existing predictors in the dataset. The bar chart in figure 6 shows that these features do appear to contribute some additional information for the classifiers to work with. This is reflected in the AUC scores and the corresponding F-scores. The result for AUC is nearly identical between Gradient Boosting, Extra-Trees, and Random forests. The most successful (by a tiny margin) showed that Random forests performed the best in terms in AUC but this was only due to high precision. The F2 and F1 score are both much reduced in in the Random Forest trial. With these new features the best candidate appears to be Extra-Trees which has an improvement across all three F-scores.

The third experiment involving feature engineering was to create a new batch of features by creating polynomial combinations of the features PRE4 and PRE5 which were shown to be the strongest predictors in figure 1. This lead to the best AUC scores out of all the trails with three out of the four classifiers pushing into the 0.9 range. The variable importance plot (8) shows that many of the polynomial features are the most successful contributors. The F-scores reflect this result and show higher scores across the board. By far the biggest increase was in terms of precision. In particular, AdaBoost faired much better with polynomial features.

Finally an experiment was carried looking at improving the performance by resampling the dataset using SMOTE. While random forests and extra-trees already carry out bagging which already balances the dataset during training, this method could of potentially helped the performance of the boosting classifiers. Figure 9 shows a marked decrease in performance across all classifiers. This is probably due to the synthetic examples not realistically reflecting the distribution of positive examples in the dataset. What is more interesting is the fact that the F2 scores for each classifier are improved by applying SMOTE but the precision is dramatically hindered. This is could be due to the synthetic examples "expanding" the region around positive examples which the algorithm considers to be positive. This is probably not representative of the true decision boundary, but has the effect of increasing recall as more examples are likely to land with the expanded positive region. While this test resulted in much worse performance it could still be of interest. In predicting thoracic surgery survival it is more desirable to have high recall than high precision.

V. Conclusions

In conclusion this paper examined the effect of four different ensemble methods on a variety of engineered features. The effect of resampling the dataset was also explored. The final classifier used on the unused testing data for submission was trained using both the additional polynomial and spirometry features. This lead to best performance with the classifier. This classifier had a final AUC after cross validation of .

While this is one of the best AUC scores achieved across all experiments there is clearly some room from improvement. One area of improvement worth exploring would be to look at more automated methods of feature selection. One such method could be recursive feature elimination in conjunction with a model that estimates feature relevance (such as random forests). Alternatively a non-linear dimensionality reduction technique could be used to find projections of the feature space onto a lower dimensional embedding. This could be particular beneficial in the case of the binary features where the feature space is relatively much larger.

Another avenue for exploration would be to look at a different branch of algorithms. For example a penalised SVM could be experimented with. The original authors of [1] propose a boosted SVM which performs reasonable well on the expanded dataset. Any alternative classifier will probably benefit from some from of bagging or resampling more than the ensemble methods.

The experiments in this paper show that any further predictive progress appears to be hindered by the low recall rate. This is a common trade-off in machine learning. Implementing this system in the real world would most likely require favouring a lower F0.5 score for a higher F2 for safety reasons. Any progress beyond the AUC achieved in this paper is likely to require a combination of further creative feature engineering and a good mix of bagging/resampling.

REFERENCES

[1] M. Zięba, J. M. Tomczak, M. Lubicz, and J. Świątek, "Boosted svm for extracting rules from imbalanced data in application to prediction of the post-operative life expectancy in the lung cancer patients," *Applied soft computing*, vol. 14, pp. 99–108, 2014.

[2] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[3] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[4] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[5] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in neurorobotics*, vol. 7, 2013.

[6] "Thoracic Surgery Data Set," http://archive.ics.uci.edu/ml/datasets/Thoracic+Surgery+Data, accessed: 2016-04-29.

[7] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *The Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] "Spirometry, Patient Website," http://patient.info/doctor/spirometry-pro, accessed: 2016-05-04.

[10] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, pp. 321–357, 2002.

[11] W. Dubitzky, M. Granzow, and D. P. Berrar, *Fundamentals of data mining in genomics and proteomics*. Springer Science & Business Media, 2007.

APPENDIX A
THIRD PARTY LIBRARIES

The code use to produce the results in the paper rely upon a number of different third party libraries. The libraries used and the relevant version of each is show in the table below. lease note that the *UnbalancedData* library is not currently directly available through *pip* by default and so must be installed directly from the GitHub repository using the following command:

```
pip install git+https://github.com/fmfn/UnbalancedDataset
```

TABLE VIII
THIRD PARTY LIBRARIES AND THE ACCOMPANYING VERSIONS USED IN ALL THE FOLLOWING CODE SAMPLES.

Name	Version
pandas	0.18.0
sklearn	0.17.1
UnbalancedDataset	0.1
matplotlib	1.5.1
numpy	1.11.0
scipy	0.17.0

APPENDIX B
PYTHON SCRIPT FOR FINAL CLASSIFIER

This appendix contains the python script for creating the final classifier used to make the predictions on the test dataset for this assignment. This will save a CSV file in the data folder called *submission.csv*.

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier

df = pd.DataFrame.from_csv("../data/train_risk.csv", index_col=False)
test = pd.DataFrame.from_csv("../data/test_risk.csv", index_col=False)
X, y = df[df.columns[:-1]], df[df.columns[-1]]


def encode_onehot(x_data, column_name, digitize=False):
    """ Encode a catagorical column from a data frame into a
    data frame of one hot features
    """
    data = x_data[[column_name]]

    if digitize:
        data = np.digitize(data, np.arange(data.min(), data.max(), 10))

    enc = preprocessing.OneHotEncoder()
    features = enc.fit_transform(data).toarray()
    names = ['%s_%d' % (column_name, i) for i in enc.active_features_]
    features = pd.DataFrame(features, columns=names, index=x_data.index)
    return features


def create_spiro_features(x_data):
    """ Creare spriometry based features """
    # create new feature FER
    # this is the raito of FEV1 and FVC
    FER = (x_data.PRE5 / x_data.PRE4) * 100
    FER.index = x_data.index

    # create a new feature OBS
    # this is whether the instance has a FER below 70%
    # which implies an obstructive disease.
    OBS = pd.Series(np.zeros(x_data.AGE.shape))
    OBS.index = x_data.index
    OBS.loc[FER < 70] = 1.0

    spiro = pd.concat([FER, OBS], axis=1)
    spiro.columns = ['FER', 'OBS']
    return spiro


```

```
def create_poly_features(x_data, names):
    """ Create new features base on Polynomials of the original best two predictors """
    poly = preprocessing.PolynomialFeatures(2, include_bias=False, interaction_only=True)
    poly_features = pd.DataFrame(poly.fit_transform(x_data[names]), index=x_data.index)
    poly_features.columns = ["POLY_%d" % i for i in poly_features.columns]
    return poly_features


def preprocess(x_data, y_data=None):
    # drop zero var PRE32
    Xp = x_data.drop("PRE32", axis=1)

    # remove outliers
    if y_data is not None:
        mask = Xp.PRE5 < 30
        Xp = Xp.loc[mask]
        Yp = y_data.copy()
        Yp = Yp.loc[mask]
    else:
        Yp = None

    # encode catagorical data as one hot vectors
    one_hot_names = ["DGN"]
    encoded = map(lambda name: encode_onehot(Xp, name), one_hot_names)
    # combine into a single data frame
    new_features = pd.concat(encoded, axis=1)

    # drop the catagorical variables that have been encoded
    Xp.drop(["DGN"], inplace=True, axis=1)
    # add new features
    Xp = pd.concat([Xp, new_features], axis=1)

    return Xp, Yp

Xp, Yp = preprocess(X, y)

scaler = preprocessing.StandardScaler()

rf_params = {
    'bootstrap': True,
    'class_weight': 'balanced',
    'criterion': 'gini',
    'max_depth': 16,
    'max_features': 1,
    'max_leaf_nodes': None,
    'min_samples_leaf': 1,
    'min_samples_split': 1,
    'min_weight_fraction_leaf': 0.0,
    'n_estimators': 300
}

rf = RandomForestClassifier(**rf_params)
rf_pipe = Pipeline([('scaler', scaler), ('RandomForestClassifier', rf)])

model = {'name': 'GradientBoost', 'model': rf_pipe}

# Create training features
spiro_features = create_spiro_features(Xp)
poly_features = create_poly_features(Xp, ['PRE4', 'PRE5'])
Xp_all = pd.concat([Xp, poly_features, spiro_features], axis=1)
Xp_all.drop(['DGN_1', 'DGN_8'], axis=1, inplace=True)

# Create testing features
Xtest, _ = preprocess(test, y_data=None)
Xtest = Xtest.drop('test_id', axis=1)

test_spiro_features = create_spiro_features(Xtest)
test_poly_features = create_poly_features(Xtest, ['PRE4', 'PRE5'])
Xtest = pd.concat([Xtest, test_spiro_features, test_poly_features], axis=1)

# Build model
final_model = model['model']
final_model.fit(Xp_all, Yp)
predicted_prob = pd.Series(final_model.predict_proba(Xtest)[:, 1])
predicted_label = pd.Series(final_model.predict(Xtest))

final_submission = pd.concat([test.test_id, predicted_label, predicted_prob], axis=1)
```

```
final_submission.columns = ['test_id', 'predicted_label', 'predicted_output']
final_submission.to_csv('../data/submission.csv', index=False)
```

../src/classify.py

APPENDIX C
IPYTHON NOTEBOOK AND ADDITIONAL PYTHON MODULES FOR ANALYSIS, TRAINING AND TUNING

This listing shows the contents of the analysis IPython notebook as a python script. For a better formatted version of this code install IPython and open the Analysis.ipynb file provided with the assignment submission. This IPython notebook contains all of the code for analysing the data, tuning the algorithms, and performing both stratified k-fold and Mote Carlo cross validation. Two additional modules (*pipeline* and *roc_analysis*) are also provided as listing at the end of this section.

```

# coding: utf-8

# In[1536]:

get_ipython().magic(u'matplotlib inline')
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# ### Loading the Datasets

# In[2556]:

df = pd.DataFrame.from_csv("../data/train_risk.csv", index_col=False)
test = pd.DataFrame.from_csv("../data/test_risk.csv", index_col=False)
X, y = df[df.columns[:-1]], df[df.columns[-1]]


# ## Analysing the Data
#
# Looking at the difference between the number of positive and negative samples in the dataset shows that there are more negative examples than positive examples. Only 28% of all samples are of the positive class.

# In[428]:

def class_balance_summary(y):
    """ Summarise the imbalance in the dataset"""
    total_size = y.size
    negative_class = y[y == 0].size
    positive_class = y[y > 0].size
    ratio = positive_class / float(positive_class + negative_class)

    print "Total number of samples: %d" % total_size
    print "Number of positive samples: %d" % positive_class
    print "Number of negative samples: %d" % negative_class
    print "Ratio of positive to total number of samples: %.2f" % ratio


class_balance_summary(y)


# Some initial observations about the data before it is preprocessed:
#  - PRE32 is all zeros. This can be removed
#  - PRE14 looks catagorical. Should be split into multiple binary variables
#  - DGN looks catagorical. As above.
#  - PRE5 looks to have some outliers. See box plot below. Potentially remove or split into two extra variable?

# In[2391]:

X.head()


# Box plot below shows the outliers in PRE5. It is worth noting that all of these outliers are of the negative class. This variable is the volume that can be exhaled in one second given full inhilation. It is likely that these values are therefore errors in reporting as it is unlikely that humans can exhale such a large volume so quickly.

# In[3501]:
```

```
# X.PRE5.plot(kind='box')
X.PRE5.plot(kind='box')
print y[X.PRE5 > 30 ]


# ## Preprocessing
#
# Create a new matrix of preprocessed features. This will encode catagorical data as one hot vectors, remove outliers, and normalise the data.

# In[4081]:

from sklearn import preprocessing

def encode_onehot(x_data, column_name, digitize=False):
    """ Encode a catagorical column from a data frame into a data frame of one hot features"""
    data = x_data[[column_name]]

    if digitize:
        data = np.digitize(data, np.arange(data.min(), data.max(), 10))

    enc = preprocessing.OneHotEncoder()
    features = enc.fit_transform(data).toarray()
    names = ['%s_%d' % (column_name, i) for i in enc.active_features_]
    features = pd.DataFrame(features, columns=names, index=x_data.index)
    return features


def preprocess(x_data, y_data=None):
    # drop zero var PRE32
    Xp = x_data.drop("PRE32", axis=1)

    # remove outliers
    if y_data is not None:
        mask = Xp.PRE5 < 30
        Xp = Xp.loc[mask]
        Yp = y_data.copy()
        Yp = Yp.loc[mask]
    else:
        Yp = None

    # encode catagorical data as one hot vectors
    one_hot_names = ["DGN"]
    encoded = map(lambda name: encode_onehot(Xp, name), one_hot_names)
    #combine into a single data frame
    new_features = pd.concat(encoded, axis=1)

    # drop the catagorical variables that have been encoded
    Xp.drop(["DGN"], inplace=True, axis=1)
    # add new features
    Xp = pd.concat([Xp, new_features], axis=1)

    return Xp, Yp

Xp, Yp = preprocess(X, y)
Xp.head()


# Measure the effectiveness of each feature using the variable importance measure from a Random Forest

# In[4043]:

def measure_importance(x_data, y_data):
    rf_selector = RandomForestClassifier(criterion='gini', class_weight='balanced')
    rf_selector.fit(scaler.fit_transform(x_data), y_data)
    feature_importance = pd.Series(rf_selector.feature_importances_, index=x_data.columns).sort_values(ascending=False)
    feature_importance.plot(kind='bar')
    return feature_importance

feature_importance = measure_importance(Xp, Yp)
Xp.drop(feature_importance[feature_importance == 0].index, inplace=True, axis=1)


# In[3967]:
```

```
feature_importance.plot(kind='barh')
plt.xlabel('Gini Impurity')
plt.tight_layout()
plt.savefig("img/feature_importance.png")


# The numerical features appear to be the most important ones. Plot a scatter plot matrix to see how the
    how the correlate with each other

# In[3757]:

pd.tools.plotting.scatter_matrix(Xp[['PRE4', 'PRE5', 'AGE']], c=Yp)


# ## Tuning Model Parameters
#
# Given the current status of the data tune the model parameters to it before we evalute the overall
    performance. Note that all of the tuning presented here is orientated towards obtaining the highest AUC
     score. Other metrics might be more desirable given the problem domain, but AUC is the measurement used
     for assignment points.

# In[3532]:

from sklearn import cross_validation
skf = cross_validation.StratifiedKFold(Yp, n_folds=5)


# ### Random Forest Tuning
# Run a grid search over a range of parameters for a Random Forest. The dataset is small enough that we can
    do them all at once. ```n_estimators``` is neglected because this should always improve as it is
    increased so we should attempt to make it as large as possible subject to lack of improvement

# In[3835]:

param_grid = {"max_depth": range(2, 20, 3),
              "max_features": range(2, 20, 3),
              "min_samples_split": range(1, 5),
              "min_samples_leaf": range(1, 5),
             }

rf = RandomForestClassifier(class_weight='balanced', n_estimators=50, random_state=50)
rf_clf = grid_search.GridSearchCV(rf, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
rf_clf.fit(Xp, Yp)


# In[3836]:

print rf_clf.best_params_


# Now take a look at the number of estimators and see where performance begins to level off.

# In[3838]:

param_grid = {"n_estimators": range(50, 500, 50)}
const_params = {'max_features': 1, 'min_samples_split': 1, 'max_depth': 16, 'min_samples_leaf': 1}

rf = RandomForestClassifier(class_weight='balanced', n_estimators=50, random_state=50, **const_params)
rf_clf2 = grid_search.GridSearchCV(rf, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
rf_clf2.fit(Xp, Yp)


# The best parameters for ```n_estimators``` levels off after around 300 estimators

# In[3840]:

plt.plot([d[0]['n_estimators'] for d in rf_clf2.grid_scores_], [d[1] for d in rf_clf2.grid_scores_])
print rf_clf2.best_params_
print rf_clf2.best_score_


# In[3725]:

rf_clf2.best_estimator_.get_params()


# ### Gradient Boosting Tuning
```

```
#
# Gradient boosting is difficult to tune effectively. [This guide](http://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/) suggests starting by fixing the learning rate and number of estimators to a relatively low number in order to tune the other hyperparameters. After they are optimised the learning rate is gradually lowered and the number of estimators increased until we find convergance on the optimum parameters

# In[3587]:

param_grid = [
   {'n_estimators': range(20,150,10)}
]

const_params = {'learning_rate': 0.1, 'min_samples_split': 1, 'min_samples_leaf': 3, 'max_depth': 8, 'max_features': 'sqrt', 'subsample': 0.8}
gbc = GradientBoostingClassifier(random_state=50, **const_params)

gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
gbc_clf.fit(Xp, Yp)


# ```n_estimators``` plateaus at around 100, so we'll use this instead of the optimum as less trees == quicker training and we'll need to decrease the learning rate and increase the number of trees later in the tuning anyway.

# In[3588]:

plt.plot([d[0]['n_estimators'] for d in gbc_clf.grid_scores_], [d[1] for d in gbc_clf.grid_scores_])
print gbc_clf.best_params_


# Now tune the ```max_depth``` and the ```min_samples_split``` parameters.

# In[3594]:

const_params = {'n_estimators':100,
                'learning_rate': 0.1,
                'min_samples_leaf': 3,
                'max_features': 'sqrt',
                'subsample': 0.8
               }

param_grid = [
    {'max_depth':range(5,16,2), 'min_samples_split':range(1, 20, 3)}
]


gbc = GradientBoostingClassifier(random_state=50, **const_params)
gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
gbc_clf.fit(Xp, Yp)


# In[3595]:

print gbc_clf.best_params_
gbc_clf.grid_scores_


# Now train ```max_features```:

# In[3600]:

const_params = {'n_estimators':100,
                'learning_rate': 0.1,
                'min_samples_leaf': 3,
                'max_features': 'sqrt',
                'max_depth': 9,
                'min_samples_split': 7,
                'subsample': 0.8
               }

param_grid = [
    {'max_features':range(5,20,2)}
]


gbc = GradientBoostingClassifier(random_state=50, **const_params)
```

```
gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
gbc_clf.fit(Xp, Yp)


# In[3601]:

plt.plot([d[0]['max_features'] for d in gbc_clf.grid_scores_], [d[1] for d in gbc_clf.grid_scores_])
print gbc_clf.best_params_


# Now train to tune the ```subsample``` rate.

# In[3603]:

const_params = {'n_estimators':100,
                'learning_rate': 0.1,
                'min_samples_leaf': 3,
                'max_features': 'sqrt',
                'max_depth': 9,
                'min_samples_split': 7,
                'max_features': 11,
                'subsample': 0.8
               }

param_grid = [
    {'subsample': [0.6, 0.7, 0.75, 0.8, 0.85, 0.9]}
]


gbc = GradientBoostingClassifier(random_state=50, **const_params)
gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
gbc_clf.fit(Xp, Yp)


# In[3604]:

plt.plot([d[0]['subsample'] for d in gbc_clf.grid_scores_], [d[1] for d in gbc_clf.grid_scores_])
print gbc_clf.best_params_


# Now cross validate with all the parameters set:

# In[3614]:

const_params = {
                'min_samples_leaf': 1,
                'min_samples_split': 7,
                'max_depth': 9,
                'max_features': 11,
                'subsample': 0.8
               }

param_grid = [
    {'n_estimators': [100], 'learning_rate': [0.1]},
    {'n_estimators': [200], 'learning_rate': [0.05]},
    {'n_estimators': [1000], 'learning_rate': [0.01]},
    {'n_estimators': [1500], 'learning_rate': [0.005]},
]

gbc = GradientBoostingClassifier(random_state=50, **const_params)

gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
gbc_clf.fit(Xp, Yp)


# In[3718]:

print gbc_clf.best_params_
gbc_clf.grid_scores_


# In[3854]:

p = pd.DataFrame(gbc_clf.best_estimator_.get_params(), index=['Value']).T
p.index.name = "Parameter"
print p.to_latex()

```

```
# ### AdaBoost Tuning
# Perhaps the easiest train due to a fairly limited number of parameters. Adjusting the ```max_depth``` suggests that 4 appears to be roughly the best option for the depth of the decision trees.

# In[3764]:

param_grid = {"n_estimators": range(50, 1000, 50), 'learning_rate': [0.1, 0.5, 0.01, 0.005]}

dt = DecisionTreeClassifier(class_weight='balanced', max_depth=4)
adb = AdaBoostClassifier(dt)
adb_clf = grid_search.GridSearchCV(adb, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
adb_clf.fit(Xp, Yp)


# In[3781]:

print adb_clf.best_params_
print adb_clf.best_score_
adb_clf.grid_scores_


# ### Extremely Random Trees Tuning
#
# This is very similar to Random Forests. In fact we will start with the same parameter set for the grid search.

# In[3800]:

param_grid = {"max_depth": range(2, 20, 3),
              "max_features": range(2, 20, 3),
              "min_samples_split": range(1, 5),
              "min_samples_leaf": range(1, 5),
             }
etc = ExtraTreesClassifier(class_weight='balanced', bootstrap=True, n_estimators=50, random_state=50)
etc_clf = grid_search.GridSearchCV(etc, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
etc_clf.fit(Xp, Yp)


# In[3801]:

print etc_clf.best_params_
print etc_clf.best_score_


# Now check increasing the number of estimators and find the drop off point

# In[3805]:

param_grid = {"n_estimators": range(50, 500, 50)}
const_params = {'max_features': 16, 'min_samples_split': 1, 'max_depth': 19, 'min_samples_leaf': 1}

etc = ExtraTreesClassifier(class_weight='balanced', bootstrap=True, random_state=50, **const_params)
etc_clf2 = grid_search.GridSearchCV(etc, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
etc_clf2.fit(Xp, Yp)


# In[3806]:

plt.plot([d[0]['n_estimators'] for d in etc_clf2.grid_scores_], [d[1] for d in etc_clf2.grid_scores_])
print etc_clf2.best_params_
print etc_clf2.best_score_


# In[3807]:

etc_clf2.best_estimator_.get_params()


# ## Model Performance
# Test the performance of each of the models on the preprocessed dataset before trying any more complicated feature engineering/resampling. This should give us some rough baseline AUC measures to work with. Firstly, set up the models. This creates a set of pipelines for each of the models we want to use.

# In[3971]:

from sklearn.pipeline import Pipeline
```

```
from sklearn.svm import SVC
from sklearn.ensemble import ExtraTreesClassifier, AdaBoostClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
reload(pipeline)
import pipeline
reload(roc_analysis)
from roc_analysis import ROCAnalysisScorer

scaler = preprocessing.StandardScaler()

# set up classifier objects
knn = KNeighborsClassifier(n_neighbors=5, weights='distance')
dct = DecisionTreeClassifier(class_weight='balanced', max_depth=4)
abt = AdaBoostClassifier(dct, n_estimators=400, learning_rate=0.5)

gbc_params = {
    'min_samples_leaf': 1,
    'min_samples_split': 7,
    'max_depth': 9,
    'max_features': 11,
    'subsample': 0.8,
    'n_estimators': 1000,
    'learning_rate': 0.01
}
gbc = GradientBoostingClassifier(**gbc_params)

exf_params = {
    'bootstrap': True,
    'class_weight': 'balanced',
    'criterion': 'gini',
    'max_depth': 19,
    'max_features': 16,
    'max_leaf_nodes': None,
    'min_samples_leaf': 1,
    'min_samples_split': 1,
    'min_weight_fraction_leaf': 0.0,
    'n_estimators': 200,
    'n_jobs': 1,
    'oob_score': False,
    'random_state': 50,
    'verbose': 0,
    'warm_start': False
}


exf = ExtraTreesClassifier(**exf_params)

rf_params = {
    'bootstrap': True,
    'class_weight': 'balanced',
    'criterion': 'gini',
    'max_depth': 16,
    'max_features': 1,
    'max_leaf_nodes': None,
    'min_samples_leaf': 1,
    'min_samples_split': 1,
    'min_weight_fraction_leaf': 0.0,
    'n_estimators': 300
}
rf_balanced = RandomForestClassifier(**rf_params)

# create pipelines for each model
abt_pipe = Pipeline([('scaler', scaler), ('AdaBoost', abt)])
exf_pipe = Pipeline([('scaler', scaler), ('ExtraTrees', exf)])
gbc_pipe = Pipeline([('scaler', scaler), ('GradientBoostingClassifer', gbc)])
rfs_pipe = Pipeline([('scaler', scaler), ('RandomForest', rf_balanced)])

# create list of model data
models = [
    {'name': 'AdaBoost', 'model': abt_pipe},
    {'name': 'ExtraTrees', 'model': exf_pipe},
    {'name': 'RandomForest', 'model': rfs_pipe},
    {'name': 'GradientBoost', 'model': gbc_pipe},
]

# set the same training set for all models.
```

```
# this is just the preprocessed dataset.
for model in models:
    model['train_data'] = (Xp, Yp)


# Define some useful helper functions for summarising the results of k-fold/monte carlo cross validation

# In[3829]:

def f_score_summary(scorers):
    """ Create a summary of the average f-scores for all folds/trials"""
    series = []
    columns = []
    for key, scorer in scorers.iteritems():
        f_scores = [np.mean(scorer.f1scores_), np.mean(scorer.f2scores_), np.mean(scorer.fhalf_scores_)]
        s = pd.Series(f_scores, index=['F1', 'F2', 'F0.5'])
        series.append(s)
        columns.append(key)

    frame = pd.concat(series, axis = 1)
    frame.columns = columns
    return frame

def summarise_scorers(scorers):
    """ Create a summary of the scorers AUCs for all folds/trials"""
    names = [name for name in scorers.keys()]
    aucs = [scorer.aucs_ for scorer in scorers.values()]
    aucs = pd.DataFrame(np.array(aucs).T, columns=names)
    return aucs.describe()


# Perform n iterations of k fold cross validation. Here I am using 10 iterations and 5 folds at each
    iteration.

# In[3972]:

scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)


# Plot an ROC curve and the mean AUCs.

# In[3973]:

get_ipython().magic(u'matplotlib inline')
for key, scorer in scorers.iteritems():
    scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)

plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '--')
# plt.savefig("img/roc_cv.png")


# Plot bar chart of the F2 scores

# In[3862]:

f_scores = f_score_summary(scorers)
ax = f_scores.loc['F2'].plot(kind='barh', title='F2 Measure for All Classifiers', color=['b', 'r', 'g', 'y'
    ])
ax.set_xlabel('F2 Score')
plt.tight_layout()
plt.savefig('img/f2_score.png')


# Summarise the F scores

# In[3833]:

f_scores = f_score_summary(scorers)
print f_scores.to_latex()
f_scores


# ## Feature Engineering
#
# Test creating some new features based on combinations of existing ones in the dataset. Cross validate
    each set of new features to see if it improves performance.

```

```
# ### Binary Features

# In[3917]:

import itertools

def binary_combinations(x_data, names):
    name_pairs = itertools.combinations(names, 2)
    features = []
    for a_name, b_name in name_pairs:
        a, b = x_data[a_name], x_data[b_name]
        features.append(np.logical_xor(a, b).astype(int))
        features.append(np.logical_and(a, b).astype(int))
        features.append(np.logical_or(a, b).astype(int))

    return pd.DataFrame(np.array(features).T, index=x_data.index)

binary_features = binary_combinations(Xp, ['PRE7', 'PRE8', 'PRE9', 'PRE10', 'PRE11', 'PRE17', 'PRE30'])
Xp_binary = pd.concat([Xp, binary_features], axis=1)
feature_importance = measure_importance(Xp_binary, Yp)
Xp_binary.drop(feature_importance[feature_importance == 0].index, inplace=True, axis=1)


# In[3879]:

for model in models:
    model['train_data'] = (Xp_binary, Yp)

scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)


# In[3881]:

get_ipython().magic(u'matplotlib inline')
for key, scorer in scorers.iteritems():
    scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)

plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '--')
plt.savefig("img/roc_binary_features.png")


# In[3882]:

f_scores = f_score_summary(scorers)
print f_scores.to_latex()
f_scores


# ### Spirometry Based Features

# In[4031]:

def create_spiro_features(x_data):
    # create new feature FER
    # this is the raito of FEV1 and FVC
    FER = (x_data.PRE5 / x_data.PRE4) * 100
    FER.index = x_data.index

    # create a new feature OBS
    # this is whether the instance has a FER below 70%
    # which implies an obstructive disease.
    OBS = pd.Series(np.zeros(x_data.AGE.shape))
    OBS.index = x_data.index
    OBS.loc[FER < 70] = 1.0

    spiro = pd.concat([FER, OBS], axis=1)
    spiro.columns = ['FER', 'OBS']
    return spiro


# In[4032]:

spiro_features = create_spiro_features(Xp)
Xp_spiro = pd.concat([Xp, spiro_features], axis=1)
feature_importance = measure_importance(Xp_spiro, Yp)
Xp_spiro.drop(feature_importance[feature_importance == 0].index, inplace=True, axis=1)

```

```
# In[3954]:

for model in models:
    model['train_data'] = (Xp_spiro, Yp)

scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)


# In[3955]:

get_ipython().magic(u'matplotlib inline')
for key, scorer in scorers.iteritems():
    scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)

plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '--')
plt.savefig("img/roc_spiro_features.png")


# In[3957]:

feature_importance.plot(kind='barh')
plt.xlabel('Gini Impurity')
plt.tight_layout()
plt.savefig("img/importance_spiro_features.png")


# In[3958]:

f_scores = f_score_summary(scorers)
print f_scores.to_latex()
f_scores


# ### Polynomial Combinations

# In[3959]:

def create_poly_features(x_data, names):
    # create new features base on Polynomials of the original best two predictors
    poly = sklearn.preprocessing.PolynomialFeatures(2, include_bias=False, interaction_only=True)
    poly_features = pd.DataFrame(poly.fit_transform(x_data[names]), index=x_data.index)
    poly_features.columns = ["POLY_%d" % i for i in poly_features.columns]
    return poly_features

poly_features = create_poly_features(Xp, ['PRE4', 'PRE5'])
Xp_poly = pd.concat([Xp, poly_features], axis=1)
feature_importance = measure_importance(Xp_poly, Yp)
Xp_poly.drop(feature_importance[feature_importance == 0].index, inplace=True, axis=1)


# In[3960]:

for model in models:
    model['train_data'] = (Xp_poly, Yp)

scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)


# In[3961]:

get_ipython().magic(u'matplotlib inline')
for key, scorer in scorers.iteritems():
    scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)

plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '--')
plt.savefig("img/roc_poly_features.png")


# In[3962]:

feature_importance.plot(kind='barh')
plt.xlabel('Gini Impurity')
plt.tight_layout()
plt.savefig("img/importance_poly_features.png")


```

```
# In[3964]:

f_scores = f_score_summary(scorers)
print f_scores.to_latex()
f_scores


# ## Resampling the Dataset
#
# Testing whether using resampling improves performance

# ### Testing with regular Over/Under sampling

# In[3906]:

splitter = pipeline.OverUnderSplitter(test_size=0.2, under_sample=0.4, over_sample=0.8)
overunder_scorers = pipeline.monte_carlo_validation(Xp, Yp, models, splitter, n=50)


# In[3822]:

for key, scorer in overunder_scorers.iteritems():
    scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)


# In[3823]:

f_score_summary(overunder_scorers)


# In[3824]:

summarise_scorers(smote_scorers)


# ### Testing with SMOTE + Undersampling

# In[3907]:

smote_params = {'kind': 'regular', 'k':3, 'ratio': 0.8, 'verbose': 1}
splitter = pipeline.SMOTESplitter(test_size=0.2, under_sample=1.0, smote_params=smote_params)
smote_scorers = pipeline.monte_carlo_validation(Xp, Yp, models, splitter, n=50)


# In[3910]:

for key, scorer in smote_scorers.iteritems():
    scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)

plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '--')
plt.savefig("img/roc_smote.png")


# In[3911]:

smote_f_scores = f_score_summary(smote_scorers)
print smote_f_scores.to_latex()
smote_f_scores


# ## Best Classifier

# In[4060]:

spiro_features = create_spiro_features(Xp)
poly_features = create_poly_features(Xp, ['PRE4', 'PRE5'])
Xp_all = pd.concat([Xp, poly_features, spiro_features], axis=1)
Xp_all.drop(['DGN_1', 'DGN_8'], axis=1, inplace=True)
for model in models:
    model['train_data'] = (Xp_all, Yp)

scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)


# In[4061]:

for key, scorer in scorers.iteritems():
```

```
    scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)

plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '--')


# In[3981]:

f_scores = f_score_summary(scorers)
f_scores


# ## Predicton on Test Set
#
# Finally, based on the best combination of techniques used in the preceeding sections, and using the
    classifier with the best AUC performance, make probalistic predictions based on the unlabelled test
    data.

# In[4083]:

Xtest, _ = preprocess(test, y_data=None)
Xtest = Xtest.drop('test_id', axis=1)

test_spiro_features = create_spiro_features(Xtest)
test_poly_features = create_poly_features(Xtest, ['PRE4', 'PRE5'])
Xtest = pd.concat([Xtest, test_spiro_features, test_poly_features], axis=1)

print Xtest.columns.size, Xp_all.columns.size


# In[4084]:

gbc_final = models[3]['model']
gbc_final.fit(Xp_all, Yp)
predicted_prob = pd.Series(gbc_final.predict_proba(Xtest)[:, 0])


# In[4085]:

predicted_label = predicted_prob.copy()
predicted_label[predicted_label>=0.5] = 1
predicted_label[predicted_label<0.5] = 0

final_submission = pd.concat([test.test_id, predicted_label, predicted_prob], axis=1)
final_submission.columns = ['test_id', 'predicted_label', 'predicted_output']
final_submission
```

../src/analysis.py

A. ROC Analysis Module

```
from sklearn import metrics
import numpy as np
from scipy import interp
import matplotlib.pyplot as plt
from scipy.stats import threshold

class ROCAnalysisScorer:
    """ Custom scorer to capute both the AUC score and the ROC curves.

    A normal scorer for sklearn cannot capture multiple scores at once.
    This object allows use to calculate multiple quantiies in one pass
    of a cross validation object.

    This will also capture the F scores
    """
    def __init__(self):
        self.rates_ = []
        self.aucs_ = []
        self.fhalf_scores_ = []
        self.f2scores_ = []
        self.f1scores_ = []

    def __call__(self, ground_truth, predictions, **kwargs):
        """ Custom __call__ function to make the object look like a
        function thanks to python's duck typing.
```

```
        """
        return self.auc_score(ground_truth, predictions, **kwargs)

    def auc_score(self, ground_truth, predictions, **kwargs):
        """ Calculate the AUC score for this particular trial.

        This will also calculate the F scores and ROC curves

        Args:
            ground_truth: vector of class labels
            predictions: vector of predicted class labels

        Returns:
            AUC score for this trial
        """

        # calculate f scores
        thresholded = threshold(predictions[:, 1], threshmin=0.5)
        thresholded = threshold(thresholded, threshmax=0.5, newval=1.0).astype(int)
        fhalf_score = metrics.fbeta_score(ground_truth.astype(int), thresholded, beta=0.5)
        f2_score = metrics.fbeta_score(ground_truth.astype(int), thresholded, beta=2)
        f1_score = metrics.fbeta_score(ground_truth.astype(int), thresholded, beta=1)

        # calculate ROC curve and AUC
        fpr, tpr, _ = metrics.roc_curve(ground_truth, predictions[:, 1])
        area = metrics.auc(fpr, tpr)

        self.fhalf_scores_.append(fhalf_score)
        self.f2scores_.append(f2_score)
        self.f1scores_.append(f1_score)
        self.rates_.append((fpr, tpr))
        self.aucs_.append(area)
        return area

    def mean_roc_metrics(self):
        """ Compute the mean AUC and mean ROC curve """
        mean_tpr = 0.0
        mean_fpr = np.linspace(0, 1, 100)

        for fpr, tpr in self.rates_:
            mean_tpr += interp(mean_fpr, fpr, tpr)

        mean_tpr = mean_tpr / len(self.rates_)
        area =  metrics.auc(mean_fpr, mean_tpr)
        return mean_fpr, mean_tpr, area

    def plot_roc_curve(self, title=None, labels=None, show_all=True, chance_line=False, mean_line=False,
    mean_label="Mean"):
        """ Plot ROC curves for all trials attached to this object

        Args:
            title: the title for the plot
            labels: the labels to use for each line. Either list, string or None
            show_all: show all plots or only the mean line
            chance_line: show the chance line
            mean_line: show the mean line
            mean_label: label for the mean line
        """
        if show_all:
            for i, ((fpr, tpr), area) in enumerate(zip(self.rates_, self.aucs_)):
                if labels is None:
                    name = "ROC %d" % (i+1)
                elif isinstance(labels, list):
                    name = labels[i]
                elif isinstance(labels, str):
                    name = labels
                plt.plot(fpr, tpr, label='%s AUC = %0.4f' % (name, area))

        if mean_line and len(self.rates_) > 1:
            mean_fpr, mean_tpr, area = self.mean_roc_metrics()
            plt.plot(mean_fpr, mean_tpr, label="%s, AUC: %.4f" % (mean_label, area))

        if chance_line:
            line = np.arange(0, 1.1, 0.1)
            plt.plot(line, line, "--", label="Chance")

        if title is None:
```

```
            title = 'Receiver operating characteristic'

        plt.title(title)
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.legend(loc="lower right", prop={'size': 10})


def plot_confusion_matrix(cm, title='Confusion matrix', cmap=plt.cm.Blues):
    """ Plot a confusion matrix

    Args:
        cm: square matrix representing the confusion matrix
        title: title to show on the plot
        cmap: the colour map to use
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)

    width, height = cm.shape
    for x in xrange(width):
        for y in xrange(height):
            plt.annotate(str(cm[x][y]), xy=(y, x),
                    horizontalalignment='center',
                    verticalalignment='center', size=20)

    plt.title(title)
    plt.colorbar()
    plt.show()
```

../src/roc_analysis.py

B. Pipeline Module

```
from roc_analysis import ROCAnalysisScorer
from sklearn import cross_validation
from sklearn.metrics import make_scorer
from unbalanced_dataset import SMOTE, UnderSampler, OverSampler


def cv_pipeline(model, x_data, y_data, cv=None):
    """ Cross Validate a model pipeline

    Args:
        model: A sklearn Pipeline object to cross validate
        x_data: Feature matrix
        y_data: Class labels
        cv: A predefined sklearn cross validation object
    Returns:
        A ROCAnalysisScorer object with the true/false positive rates and AUCs
        for all folds
    """
    roc_data = ROCAnalysisScorer()
    roc_data_scorer = make_scorer(roc_data, greater_is_better=True, needs_proba=True, average='weighted')
    cross_validation.cross_val_score(model, x_data, y_data, cv=cv, scoring=roc_data_scorer)
    return roc_data


def test_pipeline(model, x_data, y_data, x_test, y_test):
    """ Test a model on a data

    Args:
        model: A sklearn Pipeline object to test
        x_data: Feature matrix
        y_data: Class labels
    Returns:
        A ROCAnalysisScorer object with the true/fals positive rates and AUCS
        for the test
    """
    model.fit(x_data, y_data)
    y_hat = model.predict_proba(x_test)

    test_result = ROCAnalysisScorer()
    test_result.auc_score(y_test, y_hat)
    return test_result

```

```
43
44 def score_pipeline(data, cv=None):
45     """ Score a pipline using either cross validation (if a cross validation
46     object is provided) or using a training/test split
47
48     Args:
49         data: A dictionary object with the model and training or testing data
50         cv: Optional cross validation object to use
51     Returns:
52         A tuple of cross validation results and testing results
53     """
54
55     model = data['model']
56
57     cv_results = None
58     test_results = None
59
60     x_data, y_data = data['train_data']
61
62     if cv is not None:
63         cv_results = cv_pipeline(model, x_data, y_data, cv=cv)
64
65     if 'test_data' in data:
66         x_test, y_test = data['test_data']
67         test_results = test_pipeline(model, x_data, y_data, x_test, y_test)
68
69     return (cv_results, test_results)
70
71
72 def repeated_cross_fold_validation(models, n=10, k=5):
73     """ Run cross validation on a set of models n times
74
75     All models are tested using the same cross validation splits
76     at each iteration.
77
78     Args:
79         models: List of dictionaries containing the model
80         and training or testing data.
81         n: number of iterations to repeat cross validation (default 10)
82         k: number of folds to use at each iteration (default 5)
83     Returns:
84         A list of scorer objects of type ROCAnalysisScorer, one for each model
85         passed.
86     """
87
88     scorers = {}
89
90     for i in range(n):
91         # create a new cross validation set for each iteration & test.
92         skf = cross_validation.StratifiedKFold(models[0]['train_data'][1], n_folds=k)
93
94         for model in models:
95             model_name = model['name']
96             if model_name not in scorers:
97                 scorers[model_name] = ROCAnalysisScorer()
98
99             results = score_pipeline(model, cv=skf)
100
101            # for each model collect the results into a single scorer.
102            # note: no average is made at this stage. The results of each
103            # of the k folds is collected into a single k * n list for
104            # the model.
105            scorers[model_name].f1scores_ += results[0].f1scores_
106            scorers[model_name].f2scores_ += results[0].f2scores_
107            scorers[model_name].fhalf_scores_ += results[0].fhalf_scores_
108            scorers[model_name].rates_ += results[0].rates_
109            scorers[model_name].aucs_ += results[0].aucs_
110
111    return scorers
112
113
114 def monte_carlo_validation(x_data, y_data, models, splitter, n=10):
115     """ Run Monte Carlo cross validation on a set of models n times.
116
117     This will randomly split the training and test data n times
118     and evaluate the performance of each model on each split.
119
```

```
    Args:
        x_data: Feature matrix
        y_data: Class labels
        models: List of dictionaries containing the model and
        training or testing data.
        splitter: A test splitter object that creates random training
        test splits.
        n: number of iterations to perform (default 10)
    Returns:
        A list of scorer objects of type ROCAnalysisScorer, one for each
        model passed.
    """
    scorers = {}

    for i in range(n):
        x_train, y_train, x_valid, y_valid = splitter.split(x_data, y_data)

        for model in models:
            model_name = model['name']
            if model_name not in scorers:
                scorers[model_name] = ROCAnalysisScorer()

            model['train_data'] = (x_train, y_train)
            model['test_data'] = (x_valid, y_valid)

            results = score_pipeline(model)

            # for each model collect the results into a single scorer.
            # note: no average is made at this stage. The results of each
            # of the k folds is collected into a single k * n list for
            # the model.
            scorers[model_name].f1scores_ += results[1].f1scores_
            scorers[model_name].f2scores_ += results[1].f2scores_
            scorers[model_name].fhalf_scores_ += results[1].fhalf_scores_
            scorers[model_name].rates_ += results[1].rates_
            scorers[model_name].aucs_ += results[1].aucs_

    return scorers


class TestSplitter(object):
    """ TestSplitter base class.

    This splits a feature matrix and class labels vector
    into a training and testing split. This can be used to
    create more complicated splitters for resampling.
    """
    def __init__(self, test_size=0.2):
        self._test_size = test_size

    def split(self, x_data, y_data):
        Xtest, Xvalid = cross_validation.train_test_split(x_data, test_size=self._test_size, stratify=
    y_data)
        Ytest, Yvalid = y_data.loc[Xtest.index], y_data.loc[Xvalid.index]
        return Xtest, Ytest, Xvalid, Yvalid


class SMOTESplitter(TestSplitter):
    """ Test splitter for SMOTE datasets.

    This splitter will apply smote the the training portion of the dataset
    but will leave the testing part of the split untouched.
    """
    def __init__(self, under_sample=1.0, smote_params={}, **kwargs):
        super(SMOTESplitter, self).__init__(**kwargs)
        self._under_sample = under_sample
        self._smote_params = smote_params

    def split(self, x_data, y_data):
        Xt, Yt, Xv, Yv = super(SMOTESplitter, self).split(x_data, y_data)
        Xt_smote, Yt_smote = SMOTE(**self._smote_params).fit_transform(Xt.as_matrix(), Yt.as_matrix())
        Xt_smote, Yt_smote = UnderSampler(ratio=self._under_sample).fit_transform(Xt_smote, Yt_smote)
        return Xt_smote, Yt_smote, Xv, Yv


class OverUnderSplitter(TestSplitter):
    """ Test splitter for under and/or over sampling datasets.
```

```
196
197     This splitter will apply under and/or over sampling the the training
198     portion of the dataset but will leave the testing part of the split
199     untouched.
200     """
201     def __init__(self, under_sample=1.0, over_sample=1.0, **kwargs):
202         super(OverUnderSplitter, self).__init__(**kwargs)
203         self._under_sample = under_sample
204         self._over_sample = over_sample
205
206     def split(self, x_data, y_data):
207         Xt, Yt, Xv, Yv = super(OverUnderSplitter, self).split(x_data, y_data)
208         Xt_smote, Yt_smote = OverSampler(ratio=self._over_sample).fit_transform(Xt.as_matrix(), Yt.
    as_matrix())
209         Xt_smote, Yt_smote = UnderSampler(ratio=self._under_sample).fit_transform(Xt_smote, Yt_smote)
210         return Xt_smote, Yt_smote, Xv, Yv
```

../src/pipeline.py

Appendix D
Final Predictions

This listing shows the final submission CSV file generated from the code in B.

```
1  test_id,predicted_label,predicted_output
2  1,1,0.930796334882
3  2,1,0.568602386556
4  3,0,0.199756244441
5  4,0,0.223257609134
6  5,0,0.108468352895
7  6,1,0.709492781501
8  7,1,0.66811145872
9  8,1,0.526671436189
10 9,1,0.671576496177
11 10,0,0.025455110533
12 11,1,0.877360128857
13 12,1,0.660378904833
14 13,0,0.0772411738539
15 14,0,0.330609992952
16 15,0,0.481235379542
17 16,1,0.52081781105
18 17,0,0.281606722376
19 18,1,0.750899234225
20 19,1,0.746028344924
21 20,0,0.166716367839
22 21,0,0.417874548666
23 22,0,0.132900496642
24 23,0,0.219897431747
25 24,0,0.385683520678
26 25,1,0.948763261003
27 26,1,0.886118661525
28 27,0,0.153187765575
29 28,1,0.852164540869
30 29,1,0.862054431064
31 30,0,0.486335619324
32 31,0,0.256972605341
33 32,1,0.813361534401
34 33,1,0.719935958519
35 34,0,0.164955118934
36 35,1,0.645222046581
37 36,1,0.931596388285
38 37,1,0.690185156805
39 38,1,0.856189695914
40 39,1,0.874164532702
41 40,0,0.412985681004
42 41,1,0.768172217822
43 42,1,0.661658525652
44 43,0,0.0842976335621
45 44,0,0.455756501102
46 45,1,0.628059089869
47 46,1,0.882605272545
48 47,1,0.854273384453
49 48,1,0.727771904074
50 49,1,0.635179527198
51 50,0,0.123188376206
52 51,1,0.782379275211
```

```
52,0,0.0228209667991
53,1,0.789979159422
54,1,0.592845245625
55,0,0.468246812012
56,0,0.124413267301
57,1,0.762847930744
58,1,0.66547093919
59,0,0.0957612397443
60,0,0.361182401436
61,1,0.876937602066
62,1,0.816143971828
63,0,0.0190173595981
64,0,0.0792392458244
65,1,0.707652072617
66,0,0.496436132178
67,1,0.951665094386
68,0,0.288903303383
69,1,0.70995081876
70,0,0.224498004322
71,1,0.9183269844
72,1,0.937519404279
73,1,0.786448314956
74,1,0.88587461236
75,0,0.393104262221
76,1,0.845700917349
77,0,0.248636740584
78,0,0.346568635705
79,1,0.756725311104
80,1,0.724599547024
81,1,0.759772839791
82,1,0.763201534397
83,1,0.93771418578
84,0,0.271315754528
85,0,0.0879091265642
86,0,0.475492266561
87,1,0.773400467965
88,1,0.741701905905
89,0,0.344979199097
90,1,0.817128811821
91,0,0.421556736447
92,1,0.920134078405
93,1,0.945645950574
94,1,0.691924546397
95,1,0.82519102713
96,0,0.0390692900319
97,1,0.702271942814
98,1,0.919053885818
99,1,0.81743386827
100,0,0.199586488799
```

../data/submission.csv