

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 three different classifiers: random forests [2], extremely randomised trees [3], and gradient boosting [4].

The format of the result 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. 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 discussion of possible future directions are discussed in section V.

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 [5]. The dataset includes a mixture of both categorical (nominal and ordinal) and continuous data. The final (17th) 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 [6] states that a classifier with high accuracy can be built from highly imbalanced training 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

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)true value if died (T,F) (Class Label)

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 predictor. 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 dropped 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 [7]. All of these methods are known as ensemble methods. Ensemble methods compose together multiple weak learners

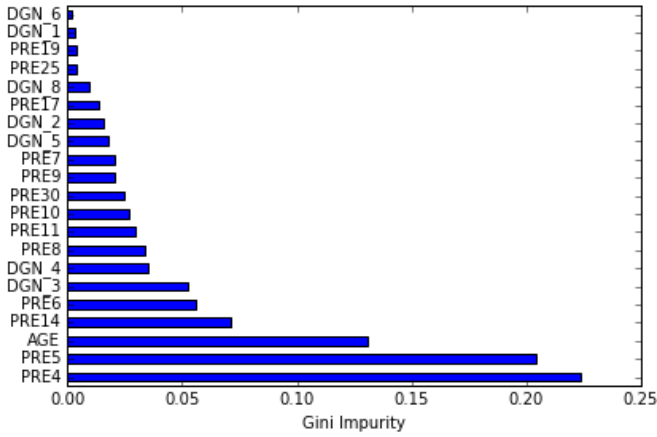


Fig. 1. Feature importance for all of the features after preprocessing.

to produce a single strong classifier. In this paper, all of the algorithms used are also tree based (although this is not necessarily the case with 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 all trees are averaged over the resulting classification the overall variance in the final model is significantly reduced. The random element for both training instances and features is used to prevent many highly correlated trees from occurring which should reduce overfitting.

Extremely Randomized trees [3] (also called Extra Trees) takes the randomisation aspect of random forests one step further. Both bagging and random features are used, but 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.

The AdaBoost algorithm [8] is a generalised method from 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. This means that misclassified samples are then more likely to be classified correctly in future iterations. Likewise, instances that are correctly classifier can be weighted much lower, as they are easier to correctly identify. Thus 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 we only consider decision tree based AdaBoost.

Gradient Boosting Machines [4] 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 our 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 two based on bagging (Random Forests and Extra Trees). 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 classify the difficult examples it’s it misses.

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 explodes with 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, all 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 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.

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

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 learning 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 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	0.601006	0.606542	0.623161	0.607031
F2	0.522676	0.546392	0.567522	0.549125
F0.5	0.715848	0.688120	0.700784	0.683489

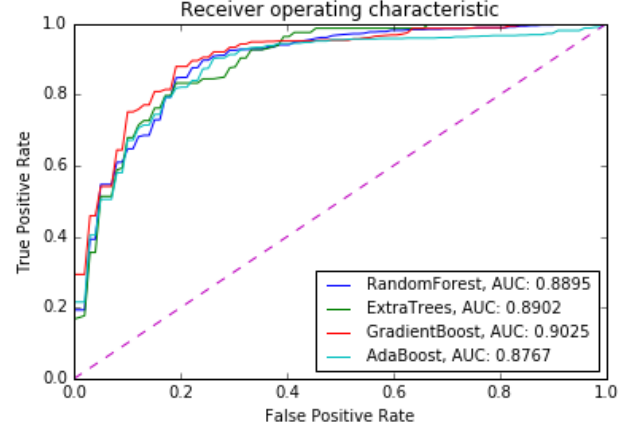


Fig. 2. Mean ROC curves and the mean AUC for all four classifiers over 10 rounds of 5-fold cross validation. All four classifiers perform similarly, with Extra Trees producing the best AUC. All four curves are slightly skewed towards the right of the plot, suggestive of poor recall. The F2 score (table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

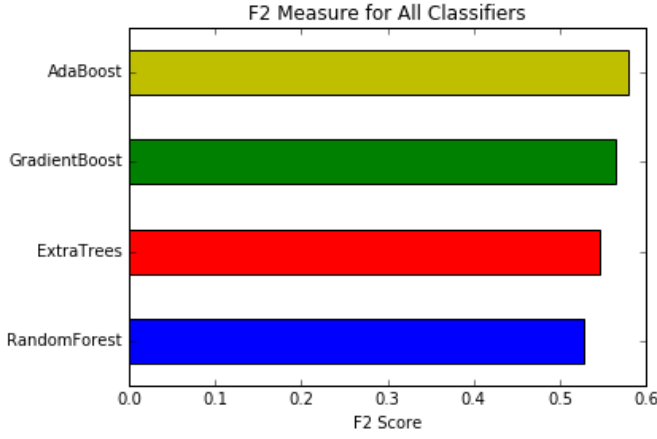


Fig. 3. Bar chart showing the F2 score for all classifiers taken from table III.

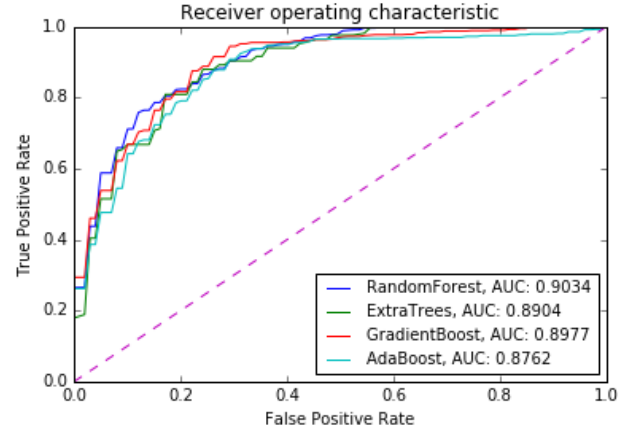


Fig. 5. ROC curves for each of the classifiers with the spirometry features.

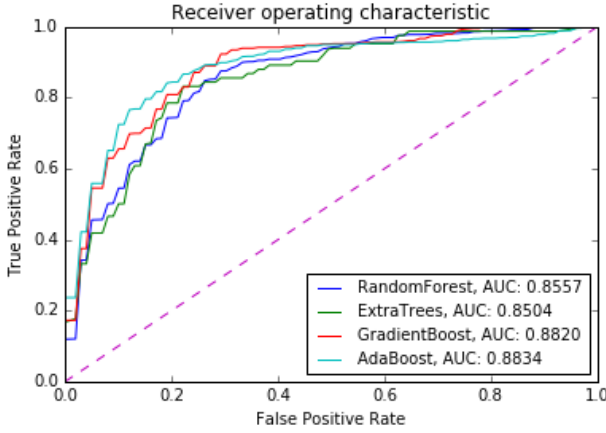


Fig. 4. ROC curves for each of the classifiers with the additional binary features. Performance is notably worse compared to the initial run.

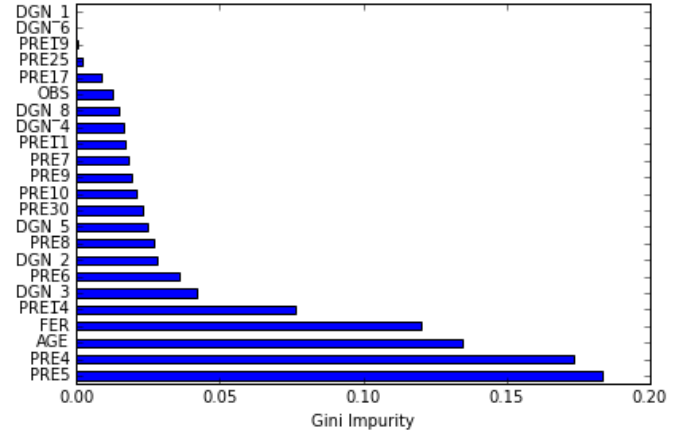


Fig. 6. Variable importance for each of the features with the spirometry features included.

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 there 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.

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.

TABLE IV
F SCORES FOR THE DATASET INCLUDING BINARY FEATURES

	RandomForest	ExtraTrees	GradientBoost	AdaBoost
F1	0.543811	0.560718	0.612539	0.646753
F2	0.465938	0.489906	0.547107	0.591129
F0.5	0.662682	0.660713	0.705097	0.721499

TABLE V
F SCORES FOR THE DATASET INCLUDING SPIROMETRY FEATURES

	RandomForest	ExtraTrees	GradientBoost	AdaBoost
F1	0.566669	0.617225	0.600845	0.614050
F2	0.483774	0.549281	0.538619	0.561441
F0.5	0.699297	0.715047	0.688793	0.683218

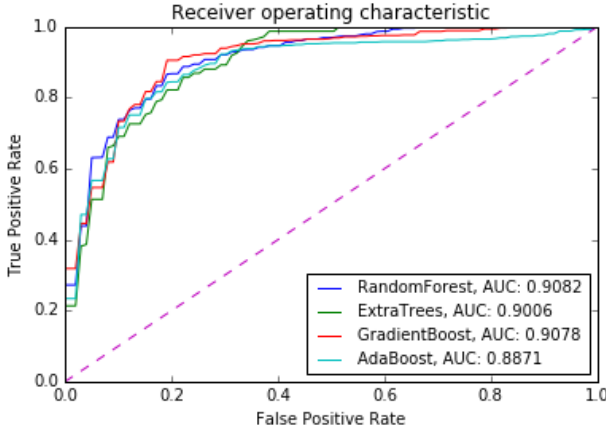


Fig. 7. ROC curves for each of the classifiers with the polynomial combination features.

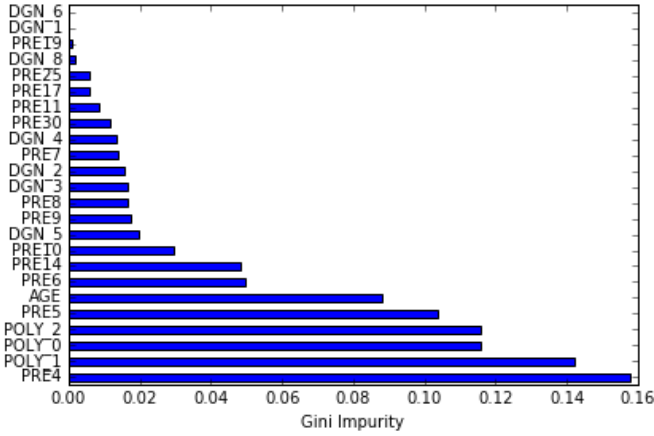


Fig. 8. Variable importance for each of the features with the polynomial combination features included.

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).

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

TABLE VI
F SCORES FOR THE DATASET INCLUDING POLYNOMIAL COMBINATION FEATURES

	RandomForest	ExtraTrees	GradientBoost	AdaBoost
F1	0.604371	0.634325	0.615442	0.662434
F2	0.517592	0.563459	0.560582	0.613066
F0.5	0.736327	0.732681	0.690843	0.726515

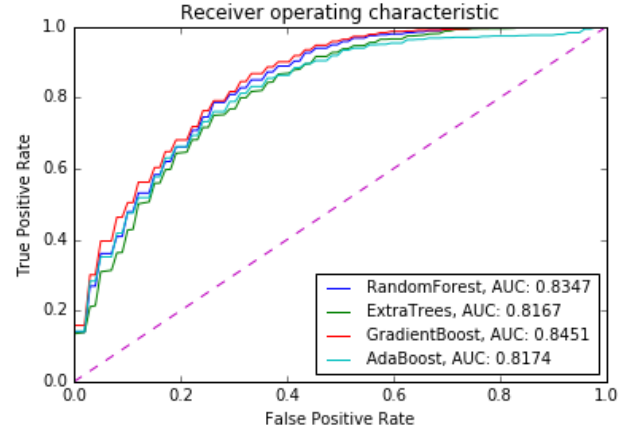


Fig. 9. ROC curves for all four classifiers with SMOTE oversampling with a ratio of 0.8. Each curve represents the average over 50 iterations of Monte Carlo validation. The ROC curves for all classifiers are less skewed compared to figure 2.

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.

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	0.614570	0.613840	0.615768	0.611035
F2	0.648492	0.650097	0.648949	0.643250
F0.5	0.587619	0.585145	0.589706	0.586983

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 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 fared 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] A. Natekin and A. Knoll, “Gradient boosting machines, a tutorial,” *Frontiers in neurorobotics*, vol. 7, 2013.
- [5] “Thoracic Surgery Data Set,” <http://archive.ics.uci.edu/ml/datasets/Thoracic+Surgery+Data>, accessed: 2016-04-29.
- [6] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.
- [7] 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.
- [8] 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.
- [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:

```
1 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*.

```
1 import pandas as pd
2 import numpy as np
3 from sklearn import preprocessing
4 from sklearn.pipeline import Pipeline
5 from sklearn.ensemble import GradientBoostingClassifier
6
7 df = pd.DataFrame.from_csv("../data/train_risk.csv", index_col=False)
8 test = pd.DataFrame.from_csv("../data/test_risk.csv", index_col=False)
9 X, y = df[df.columns[:-1]], df[df.columns[-1]]
10
11
12 def encode_onehot(x_data, column_name, digitize=False):
13     """ Encode a catagorical column from a data frame into a
14     data frame of one hot features
15     """
16     data = x_data[[column_name]]
17
18     if digitize:
19         data = np.digitize(data, np.arange(data.min(), data.max(), 10))
20
21     enc = preprocessing.OneHotEncoder()
22     features = enc.fit_transform(data).toarray()
23     names = ['%s_%d' % (column_name, i) for i in enc.active_features_]
24     features = pd.DataFrame(features, columns=names, index=x_data.index)
25     return features
26
27
28 def create_spiro_features(x_data):
29     """ Create spriometry based features """
30     # create new feature FER
31     # this is the raito of FEV1 and FVC
32     FER = (x_data.PRE5 / x_data.PRE4) * 100
33     FER.index = x_data.index
34
35     # create a new feature OBS
36     # this is whether the instance has a FER below 70%
37     # which implies an obstructive disease.
38     OBS = pd.Series(np.zeros(x_data.AGE.shape))
39     OBS.index = x_data.index
40     OBS.loc[FER < 70] = 1.0
41
42     spiro = pd.concat([FER, OBS], axis=1)
43     spiro.columns = ['FER', 'OBS']
44     return spiro
45
46
```



```

47 def create_poly_features(x_data, names):
48     """ Create new features base on Polynomials of the original best two predictors """
49     poly = preprocessing.PolynomialFeatures(2, include_bias=False, interaction_only=True)
50     poly_features = pd.DataFrame(poly.fit_transform(x_data[names]), index=x_data.index)
51     poly_features.columns = ["POLY_%d" % i for i in poly_features.columns]
52     return poly_features
53
54
55 def preprocess(x_data, y_data=None):
56     # drop zero var PRE32
57     Xp = x_data.drop("PRE32", axis=1)
58
59     # remove outliers
60     if y_data is not None:
61         mask = Xp.PRE5 < 30
62         Xp = Xp.loc[mask]
63         Yp = y_data.copy()
64         Yp = Yp.loc[mask]
65     else:
66         Yp = None
67
68     # encode catagorical data as one hot vectors
69     one_hot_names = ["DGN"]
70     encoded = map(lambda name: encode_onehot(Xp, name), one_hot_names)
71     # combine into a single data frame
72     new_features = pd.concat(encoded, axis=1)
73
74     # drop the catagorical variables that have been encoded
75     Xp.drop(["DGN"], inplace=True, axis=1)
76     # add new features
77     Xp = pd.concat([Xp, new_features], axis=1)
78
79     return Xp, Yp
80
81 Xp, Yp = preprocess(X, y)
82
83 scaler = preprocessing.StandardScaler()
84
85 gbc_params = {
86     'min_samples_leaf': 1,
87     'min_samples_split': 7,
88     'max_depth': 9,
89     'max_features': 11,
90     'subsample': 0.8,
91     'n_estimators': 1000,
92     'learning_rate': 0.01
93 }
94 gbc = GradientBoostingClassifier(**gbc_params)
95 gbc_pipe = Pipeline([('scaler', scaler), ('GradientBoostingClassifier', gbc)])
96
97 model = {'name': 'GradientBoost', 'model': gbc_pipe}
98
99 # Create training features
100 spiro_features = create_spiro_features(Xp)
101 poly_features = create_poly_features(Xp, ['PRE4', 'PRE5'])
102 Xp_all = pd.concat([Xp, poly_features, spiro_features], axis=1)
103 Xp_all.drop(['DGN_1', 'DGN_8'], axis=1, inplace=True)
104
105 # Create testing features
106 Xtest, _ = preprocess(test, y_data=None)
107 Xtest = Xtest.drop('test_id', axis=1)
108
109 test_spiro_features = create_spiro_features(Xtest)
110 test_poly_features = create_poly_features(Xtest, ['PRE4', 'PRE5'])
111 Xtest = pd.concat([Xtest, test_spiro_features, test_poly_features], axis=1)
112
113 # Build model
114 gbc_final = model['model']
115 gbc_final.fit(Xp_all, Yp)
116 predicted_prob = pd.Series(gbc_final.predict_proba(Xtest)[:, 0])
117
118 # Format output
119 predicted_label = predicted_prob.copy()
120 predicted_label[predicted_label >= 0.5] = 1
121 predicted_label[predicted_label < 0.5] = 0
122 predicted_label = predicted_label.astype(int)
123

```

```

124 final_submission = pd.concat([test.test_id, predicted_label, predicted_prob], axis=1)
125 final_submission.columns = ['test_id', 'predicted_label', 'predicted_output']
126 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 as the end of this section.

```

1  # coding: utf-8
2
3
4  # In[1536]:
5
6  get_ipython().magic(u'matplotlib inline')
7  import pandas as pd
8  import numpy as np
9  import matplotlib.pyplot as plt
10
11
12  # ### Loading the Datasets
13
14  # In[2556]:
15
16  df = pd.DataFrame.from_csv("../data/train_risk.csv", index_col=False)
17  test = pd.DataFrame.from_csv("../data/test_risk.csv", index_col=False)
18  X, y = df[df.columns[:-1]], df[df.columns[-1]]
19
20
21  # ## Analysing the Data
22  #
23  # Looking at the difference between the number of positive and negative samples in the dataset shows that
24  # there are more negative examples than positive examples. Only 28% of all samples are of the positive
25  # class.
26
27  # In[428]:
28
29  def class_balance_summary(y):
30      """ Summarise the imbalance in the dataset """
31      total_size = y.size
32      negative_class = y[y == 0].size
33      positive_class = y[y > 0].size
34      ratio = positive_class / float(positive_class + negative_class)
35
36      print "Total number of samples: %d" % total_size
37      print "Number of positive samples: %d" % positive_class
38      print "Number of negative samples: %d" % negative_class
39      print "Ratio of positive to total number of samples: %.2f" % ratio
40
41
42  class_balance_summary(y)
43
44  # Some initial observations about the data before it is preprocessed:
45  # - PRE32 is all zeros. This can be removed
46  # - PRE14 looks catagorical. Should be split into multiple binary variables
47  # - DGN looks catagorical. As above.
48  # - PRE5 looks to have some outliers. See box plot below. Potentially remove or split into two extra
49  #   variable?
50
51  # In[2391]:
52
53  X.head()
54
55  # Box plot below shows the outliers in PRE5. It is worth noting that all of these outliers are of the
56  # negative class. This variable is the volume that can be exhaled in one second given full inhalation. It
57  # is likely that these values are therefore errors in reporting as it is unlikely that humans can exhale
58  # such a large volume so quickly.

```

```

56 # In[3501]:
57
58 # X.PRE5.plot(kind='box')
59 X.PRE5.plot(kind='box')
60 print y[X.PRE5 > 30 ]
61
62
63 # ## Preprocessing
64 #
65 # Create a new matrix of preprocessed features. This will encode catagorical data as one hot vectors ,
66 # remove outliers , and normalise the data.
67
68 # In[4081]:
69
70 from sklearn import preprocessing
71
72 def encode_onehot(x_data, column_name, digitize=False):
73     """ Encode a catagorical column from a data frame into a data frame of one hot features"""
74     data = x_data[[column_name]]
75
76     if digitize:
77         data = np.digitize(data, np.arange(data.min(), data.max(), 10))
78
79     enc = preprocessing.OneHotEncoder()
80     features = enc.fit_transform(data).toarray()
81     names = ['%s_%d' % (column_name, i) for i in enc.active_features_]
82     features = pd.DataFrame(features, columns=names, index=x_data.index)
83     return features
84
85 def preprocess(x_data, y_data=None):
86     # drop zero var PRE32
87     Xp = x_data.drop("PRE32", axis=1)
88
89     # remove outliers
90     if y_data is not None:
91         mask = Xp.PRE5 < 30
92         Xp = Xp.loc[mask]
93         Yp = y_data.copy()
94         Yp = Yp.loc[mask]
95     else:
96         Yp = None
97
98     # encode catagorical data as one hot vectors
99     one_hot_names = ["DGN"]
100     encoded = map(lambda name: encode_onehot(Xp, name), one_hot_names)
101     #combine into a single data frame
102     new_features = pd.concat(encoded, axis=1)
103
104     # drop the catagorical variables that have been encoded
105     Xp.drop(["DGN"], inplace=True, axis=1)
106     # add new features
107     Xp = pd.concat([Xp, new_features], axis=1)
108
109     return Xp, Yp
110
111 Xp, Yp = preprocess(X, y)
112 Xp.head()
113
114
115 # Measure the effectiveness of each feature using the variable importance measure from a Random Forest
116
117 # In[4043]:
118
119 def measure_importance(x_data, y_data):
120     rf_selector = RandomForestClassifier(criterion='gini', class_weight='balanced')
121     rf_selector.fit(scaler.fit_transform(x_data), y_data)
122     feature_importance = pd.Series(rf_selector.feature_importances_, index=x_data.columns).sort_values(
123         ascending=False)
124     feature_importance.plot(kind='bar')
125     return feature_importance
126
127 feature_importance = measure_importance(Xp, Yp)
128 Xp.drop(feature_importance[feature_importance == 0].index, inplace=True, axis=1)
129
130 # In[3967]:

```

```

131 feature_importance.plot(kind='barh')
132 plt.xlabel('Gini Impurity')
133 plt.tight_layout()
134 plt.savefig("img/feature_importance.png")
135
136
137 # The numerical features appear to be the most important ones. Plot a scatter plot matrix to see how the
138 # how the correlate with each other
139
140 # In[3757]:
141
142 pd.tools.plotting.scatter_matrix(Xp[['PRE4', 'PRE5', 'AGE']], c=Yp)
143
144
145 # ## Tuning Model Parameters
146 #
147 # Given the current status of the data tune the model parameters to it before we evaluate the overall
148 # performance. Note that all of the tuning presented here is orientated towards obtaining the highest AUC
149 # score. Other metrics might be more desirable given the problem domain, but AUC is the measurement used
150 # for assignment points.
151
152 # In[3532]:
153
154 from sklearn import cross_validation
155 skf = cross_validation.StratifiedKFold(Yp, n_folds=5)
156
157
158 # ### Random Forest Tuning
159 # Run a grid search over a range of parameters for a Random Forest. The dataset is small enough that we can
160 # do them all at once. 'n_estimators' is neglected because this should always improve as it is
161 # increased so we should attempt to make it as large as possible subject to lack of improvement
162
163 # In[3835]:
164
165 param_grid = {"max_depth": range(2, 20, 3),
166               "max_features": range(2, 20, 3),
167               "min_samples_split": range(1, 5),
168               "min_samples_leaf": range(1, 5),
169               }
170
171 rf = RandomForestClassifier(class_weight='balanced', n_estimators=50, random_state=50)
172 rf_clf = grid_search.GridSearchCV(rf, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
173 rf_clf.fit(Xp, Yp)
174
175 # In[3836]:
176
177 print rf_clf.best_params_
178
179 # Now take a look at the number of estimators and see where performance begins to level off.
180
181 # In[3838]:
182
183 param_grid = {"n_estimators": range(50, 500, 50)}
184 const_params = {'max_features': 1, 'min_samples_split': 1, 'max_depth': 16, 'min_samples_leaf': 1}
185
186 rf = RandomForestClassifier(class_weight='balanced', n_estimators=50, random_state=50, **const_params)
187 rf_clf2 = grid_search.GridSearchCV(rf, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
188 rf_clf2.fit(Xp, Yp)
189
190 # The best parameters for 'n_estimators' levels off after around 300 estimators
191
192 # In[3840]:
193
194 plt.plot([d[0] for d in rf_clf2.grid_scores_], [d[1] for d in rf_clf2.grid_scores_])
195 print rf_clf2.best_params_
196 print rf_clf2.best_score_
197
198 # In[3725]:
199
200 rf_clf2.best_estimator_.get_params()
201

```

```

202 # ### Gradient Boosting Tuning
203 #
204 # 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 convergence on the optimum parameters
205
206 # In[3587]:
207
208 param_grid = [
209     {'n_estimators': range(20,150,10)}
210 ]
211
212 const_params = {'learning_rate': 0.1, 'min_samples_split': 1, 'min_samples_leaf': 3, 'max_depth': 8, '
    max_features': 'sqrt', 'subsample': 0.8}
213 gbc = GradientBoostingClassifier(random_state=50, **const_params)
214
215 gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
216 gbc_clf.fit(Xp, Yp)
217
218
219 # “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.
220
221 # In[3588]:
222
223 plt.plot([d[0]['n_estimators'] for d in gbc_clf.grid_scores_], [d[1] for d in gbc_clf.grid_scores_])
224 print gbc_clf.best_params_
225
226
227 # Now tune the “max_depth” and the “min_samples_split” parameters.
228
229 # In[3594]:
230
231 const_params = {'n_estimators':100,
232                 'learning_rate': 0.1,
233                 'min_samples_leaf': 3,
234                 'max_features': 'sqrt',
235                 'subsample': 0.8
236                 }
237
238 param_grid = [
239     {'max_depth': range(5,16,2), 'min_samples_split': range(1, 20, 3)}
240 ]
241
242
243 gbc = GradientBoostingClassifier(random_state=50, **const_params)
244 gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
245 gbc_clf.fit(Xp, Yp)
246
247
248 # In[3595]:
249
250 print gbc_clf.best_params_
251 gbc_clf.grid_scores_
252
253
254 # Now train “max_features”:
255
256 # In[3600]:
257
258 const_params = {'n_estimators':100,
259                 'learning_rate': 0.1,
260                 'min_samples_leaf': 3,
261                 'max_features': 'sqrt',
262                 'max_depth': 9,
263                 'min_samples_split': 7,
264                 'subsample': 0.8
265                 }
266
267 param_grid = [
268     {'max_features': range(5,20,2)}
269 ]
270
271

```

```

272 gbc = GradientBoostingClassifier(random_state=50, **const_params)
273 gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
274 gbc_clf.fit(Xp, Yp)
275
276
277 # In[3601]:
278
279 plt.plot([d[0]['max_features'] for d in gbc_clf.grid_scores_], [d[1] for d in gbc_clf.grid_scores_])
280 print gbc_clf.best_params_
281
282
283 # Now train to tune the "subsample" rate.
284
285 # In[3603]:
286
287 const_params = {'n_estimators':100,
288                 'learning_rate': 0.1,
289                 'min_samples_leaf': 3,
290                 'max_features': 'sqrt',
291                 'max_depth': 9,
292                 'min_samples_split': 7,
293                 'max_features': 11,
294                 'subsample': 0.8
295                 }
296
297 param_grid = [
298     {'subsample': [0.6, 0.7, 0.75, 0.8, 0.85, 0.9]}
299 ]
300
301
302 gbc = GradientBoostingClassifier(random_state=50, **const_params)
303 gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
304 gbc_clf.fit(Xp, Yp)
305
306
307 # In[3604]:
308
309 plt.plot([d[0]['subsample'] for d in gbc_clf.grid_scores_], [d[1] for d in gbc_clf.grid_scores_])
310 print gbc_clf.best_params_
311
312
313 # Now cross validate with all the parameters set:
314
315 # In[3614]:
316
317 const_params = {
318     'min_samples_leaf': 1,
319     'min_samples_split': 7,
320     'max_depth': 9,
321     'max_features': 11,
322     'subsample': 0.8
323 }
324
325 param_grid = [
326     {'n_estimators': [100], 'learning_rate': [0.1]},
327     {'n_estimators': [200], 'learning_rate': [0.05]},
328     {'n_estimators': [1000], 'learning_rate': [0.01]},
329     {'n_estimators': [1500], 'learning_rate': [0.005]},
330 ]
331
332 gbc = GradientBoostingClassifier(random_state=50, **const_params)
333
334 gbc_clf = grid_search.GridSearchCV(gbc, param_grid, cv=skf, scoring='roc_auc')
335 gbc_clf.fit(Xp, Yp)
336
337
338 # In[3718]:
339
340 print gbc_clf.best_params_
341 gbc_clf.grid_scores_
342
343
344 # In[3854]:
345
346 p = pd.DataFrame(gbc_clf.best_estimator_.get_params(), index=['Value']).T
347 p.index.name = "Parameter"
348 print p.to_latex()

```

```

349
350
351 # ### AdaBoost Tuning
352 # 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.
353
354 # In[3764]:
355
356 param_grid = {"n_estimators": range(50, 1000, 50), 'learning_rate': [0.1, 0.5, 0.01, 0.005]}
357
358 dt = DecisionTreeClassifier(class_weight='balanced', max_depth=4)
359 adb = AdaBoostClassifier(dt)
360 adb_clf = grid_search.GridSearchCV(adb, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
361 adb_clf.fit(Xp, Yp)
362
363
364 # In[3781]:
365
366 print adb_clf.best_params_
367 print adb_clf.best_score_
368 adb_clf.grid_scores_
369
370
371 # ### Extremely Random Trees Tuning
372 #
373 # This is very similar to Random Forests. In fact we will start with the same parameter set for the grid
    search.
374
375 # In[3800]:
376
377 param_grid = {"max_depth": range(2, 20, 3),
378               "max_features": range(2, 20, 3),
379               "min_samples_split": range(1, 5),
380               "min_samples_leaf": range(1, 5),
381               }
382 etc = ExtraTreesClassifier(class_weight='balanced', bootstrap=True, n_estimators=50, random_state=50)
383 etc_clf = grid_search.GridSearchCV(etc, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
384 etc_clf.fit(Xp, Yp)
385
386
387 # In[3801]:
388
389 print etc_clf.best_params_
390 print etc_clf.best_score_
391
392
393 # Now check increasing the number of estimators and find the drop off point
394
395 # In[3805]:
396
397 param_grid = {"n_estimators": range(50, 500, 50)}
398 const_params = {'max_features': 16, 'min_samples_split': 1, 'max_depth': 19, 'min_samples_leaf': 1}
399
400 etc = ExtraTreesClassifier(class_weight='balanced', bootstrap=True, random_state=50, **const_params)
401 etc_clf2 = grid_search.GridSearchCV(etc, param_grid, n_jobs=-1, cv=skf, scoring='roc_auc')
402 etc_clf2.fit(Xp, Yp)
403
404
405 # In[3806]:
406
407 plt.plot([d[0]['n_estimators'] for d in etc_clf2.grid_scores_], [d[1] for d in etc_clf2.grid_scores_])
408 print etc_clf2.best_params_
409 print etc_clf2.best_score_
410
411
412 # In[3807]:
413
414 etc_clf2.best_estimator_.get_params()
415
416
417 # ## Model Performance
418 # 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.
419
420 # In[3971]:
421

```



```

422 from sklearn.pipeline import Pipeline
423 from sklearn.svm import SVC
424 from sklearn.ensemble import ExtraTreesClassifier, AdaBoostClassifier
425 from sklearn.neighbors import KNeighborsClassifier
426 from sklearn.tree import DecisionTreeClassifier
427 reload(pipeline)
428 import pipeline
429 reload(roc_analysis)
430 from roc_analysis import ROCAalysisScorer
431
432 scaler = preprocessing.StandardScaler()
433
434 # set up classifier objects
435 knn = KNeighborsClassifier(n_neighbors=5, weights='distance')
436 dct = DecisionTreeClassifier(class_weight='balanced', max_depth=4)
437 abt = AdaBoostClassifier(dct, n_estimators=400, learning_rate=0.5)
438
439 gbc_params = {
440     'min_samples_leaf': 1,
441     'min_samples_split': 7,
442     'max_depth': 9,
443     'max_features': 11,
444     'subsample': 0.8,
445     'n_estimators': 1000,
446     'learning_rate': 0.01
447 }
448 gbc = GradientBoostingClassifier(**gbc_params)
449
450 exf_params = {
451     'bootstrap': True,
452     'class_weight': 'balanced',
453     'criterion': 'gini',
454     'max_depth': 19,
455     'max_features': 16,
456     'max_leaf_nodes': None,
457     'min_samples_leaf': 1,
458     'min_samples_split': 1,
459     'min_weight_fraction_leaf': 0.0,
460     'n_estimators': 200,
461     'n_jobs': 1,
462     'oob_score': False,
463     'random_state': 50,
464     'verbose': 0,
465     'warm_start': False
466 }
467
468 exf = ExtraTreesClassifier(**exf_params)
469
470 rf_params = {
471     'bootstrap': True,
472     'class_weight': 'balanced',
473     'criterion': 'gini',
474     'max_depth': 16,
475     'max_features': 1,
476     'max_leaf_nodes': None,
477     'min_samples_leaf': 1,
478     'min_samples_split': 1,
479     'min_weight_fraction_leaf': 0.0,
480     'n_estimators': 300
481 }
482 rf_balanced = RandomForestClassifier(**rf_params)
483
484 # create pipelines for each model
485 abt_pipe = Pipeline([('scaler', scaler), ('AdaBoost', abt)])
486 exf_pipe = Pipeline([('scaler', scaler), ('ExtraTrees', exf)])
487 gbc_pipe = Pipeline([('scaler', scaler), ('GradientBoostingClassifier', gbc)])
488 rfs_pipe = Pipeline([('scaler', scaler), ('RandomForest', rf_balanced)])
489
490 # create list of model data
491 models = [
492     {'name': 'AdaBoost', 'model': abt_pipe},
493     {'name': 'ExtraTrees', 'model': exf_pipe},
494     {'name': 'RandomForest', 'model': rfs_pipe},
495     {'name': 'GradientBoost', 'model': gbc_pipe},
496 ]
497
498

```

```

499 # set the same training set for all models.
500 # this is just the preprocessed dataset.
501 for model in models:
502     model['train_data'] = (Xp, Yp)
503
504
505 # Define some useful helper functions for summarising the results of k-fold/monte carlo cross validation
506
507 # In[3829]:
508
509 def f_score_summary(scorers):
510     """ Create a summary of the average f-scores for all folds/trials """
511     series = []
512     columns = []
513     for key, scorer in scorers.iteritems():
514         f_scores = [np.mean(scorer.f1scores_), np.mean(scorer.f2scores_), np.mean(scorer.fhalf_scores_)]
515         s = pd.Series(f_scores, index=['F1', 'F2', 'F0.5'])
516         series.append(s)
517         columns.append(key)
518
519     frame = pd.concat(series, axis = 1)
520     frame.columns = columns
521     return frame
522
523 def summarise_scorers(scorers):
524     """ Create a summary of the scorers AUCs for all folds/trials """
525     names = [name for name in scorers.keys()]
526     aucs = [scorer.aucs_ for scorer in scorers.values()]
527     aucs = pd.DataFrame(np.array(aucs).T, columns=names)
528     return aucs.describe()
529
530
531 # Perform n iterations of k fold cross validation. Here I am using 10 iterations and 5 folds at each
532     iteration.
533
534 # In[3972]:
535
536 scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)
537
538 # Plot an ROC curve and the mean AUCs.
539
540 # In[3973]:
541
542 get_ipython().magic(u'matplotlib inline')
543 for key, scorer in scorers.iteritems():
544     scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)
545
546 plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '—')
547 # plt.savefig("img/roc_cv.png")
548
549
550 # Plot bar chart of the F2 scores
551
552 # In[3862]:
553
554 f_scores = f_score_summary(scorers)
555 ax = f_scores.loc['F2'].plot(kind='barh', title='F2 Measure for All Classifiers', color=['b', 'r', 'g', 'y'])
556 ax.set_xlabel('F2 Score')
557 plt.tight_layout()
558 plt.savefig('img/f2_score.png')
559
560
561 # Summarise the F scores
562
563 # In[3833]:
564
565 f_scores = f_score_summary(scorers)
566 print f_scores.to_latex()
567 f_scores
568
569
570 # ## Feature Engineering
571 #
572 # Test creating some new features based on combinations of existing ones in the dataset. Cross validate
573     each set of new features to see if it improves performance.

```

```

573
574 # ### Binary Features
575
576 # In[3917]:
577
578 import itertools
579
580 def binary_combinations(x_data, names):
581     name_pairs = itertools.combinations(names, 2)
582     features = []
583     for a_name, b_name in name_pairs:
584         a, b = x_data[a_name], x_data[b_name]
585         features.append(np.logical_xor(a, b).astype(int))
586         features.append(np.logical_and(a, b).astype(int))
587         features.append(np.logical_or(a, b).astype(int))
588
589     return pd.DataFrame(np.array(features).T, index=x_data.index)
590
591 binary_features = binary_combinations(Xp, ['PRE7', 'PRE8', 'PRE9', 'PRE10', 'PRE11', 'PRE17', 'PRE30'])
592 Xp_binary = pd.concat([Xp, binary_features], axis=1)
593 feature_importance = measure_importance(Xp_binary, Yp)
594 Xp_binary.drop(feature_importance[feature_importance == 0].index, inplace=True, axis=1)
595
596 # In[3879]:
597
598 for model in models:
599     model['train_data'] = (Xp_binary, Yp)
600
601 scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)
602
603
604 # In[3881]:
605
606 get_ipython().magic(u'matplotlib inline')
607 for key, scorer in scorers.iteritems():
608     scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)
609
610 plt.plot(np.arange(0,1,1, 0.1), np.arange(0,1,1, 0.1), '—')
611 plt.savefig("img/roc_binary_features.png")
612
613
614 # In[3882]:
615
616 f_scores = f_score_summary(scorers)
617 print f_scores.to_latex()
618 f_scores
619
620
621 # ### Spirometry Based Features
622
623 # In[4031]:
624
625 def create_spiro_features(x_data):
626     # create new feature FER
627     # this is the ratio of FEV1 and FVC
628     FER = (x_data.PRE5 / x_data.PRE4) * 100
629     FER.index = x_data.index
630
631     # create a new feature OBS
632     # this is whether the instance has a FER below 70%
633     # which implies an obstructive disease.
634     OBS = pd.Series(np.zeros(x_data.AGE.shape))
635     OBS.index = x_data.index
636     OBS.loc[FER < 70] = 1.0
637
638     spiro = pd.concat([FER, OBS], axis=1)
639     spiro.columns = ['FER', 'OBS']
640     return spiro
641
642
643 # In[4032]:
644
645 spiro_features = create_spiro_features(Xp)
646 Xp_spiro = pd.concat([Xp, spiro_features], axis=1)
647 feature_importance = measure_importance(Xp_spiro, Yp)
648 Xp_spiro.drop(feature_importance[feature_importance == 0].index, inplace=True, axis=1)
649

```

```

650
651
652 # In[3954]:
653
654 for model in models:
655     model['train_data'] = (Xp_spiro, Yp)
656
657 scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)
658
659 # In[3955]:
660
661 get_ipython().magic(u'matplotlib inline')
662 for key, scorer in scorers.iteritems():
663     scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)
664
665 plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '—')
666 plt.savefig("img/roc_spiro_features.png")
667
668
669 # In[3957]:
670
671 feature_importance.plot(kind='barh')
672 plt.xlabel('Gini Impurity')
673 plt.tight_layout()
674 plt.savefig("img/importance_spiro_features.png")
675
676
677 # In[3958]:
678
679 f_scores = f_score_summary(scorers)
680 print f_scores.to_latex()
681 f_scores
682
683
684 # ### Polynomial Combinations
685
686 # In[3959]:
687
688 def create_poly_features(x_data, names):
689     # create new features base on Polynomials of the original best two predictors
690     poly = sklearn.preprocessing.PolynomialFeatures(2, include_bias=False, interaction_only=True)
691     poly_features = pd.DataFrame(poly.fit_transform(x_data[names]), index=x_data.index)
692     poly_features.columns = ["POLY%d" % i for i in poly_features.columns]
693     return poly_features
694
695
696 poly_features = create_poly_features(Xp, ['PRE4', 'PRE5'])
697 Xp_poly = pd.concat([Xp, poly_features], axis=1)
698 feature_importance = measure_importance(Xp_poly, Yp)
699 Xp_poly.drop(feature_importance[feature_importance == 0].index, inplace=True, axis=1)
700
701
702 # In[3960]:
703
704 for model in models:
705     model['train_data'] = (Xp_poly, Yp)
706
707 scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)
708
709 # In[3961]:
710
711 get_ipython().magic(u'matplotlib inline')
712 for key, scorer in scorers.iteritems():
713     scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)
714
715 plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '—')
716 plt.savefig("img/roc_poly_features.png")
717
718
719 # In[3962]:
720
721 feature_importance.plot(kind='barh')
722 plt.xlabel('Gini Impurity')
723 plt.tight_layout()
724 plt.savefig("img/importance_poly_features.png")
725
726

```

```

727 # In[3964]:
728
729 f_scores = f_score_summary(scorers)
730 print f_scores.to_latex()
731 f_scores
732
733
734 # ## Resampling the Dataset
735 #
736 # Testing whether using resampling improves performance
737
738 # ### Testing with regular Over/Under sampling
739
740 # In[3906]:
741
742 splitter = pipeline.OverUnderSplitter(test_size=0.2, under_sample=0.4, over_sample=0.8)
743 overunder_scorers = pipeline.monte_carlo_validation(Xp, Yp, models, splitter, n=50)
744
745
746 # In[3822]:
747
748 for key, scorer in overunder_scorers.iteritems():
749     scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)
750
751
752 # In[3823]:
753
754 f_score_summary(overunder_scorers)
755
756
757 # In[3824]:
758
759 summarise_scorers(smote_scorers)
760
761
762 # ### Testing with SMOTE + Undersampling
763
764 # In[3907]:
765
766 smote_params = {'kind': 'regular', 'k':3, 'ratio': 0.8, 'verbose': 1}
767 splitter = pipeline.SMOTESplitter(test_size=0.2, under_sample=1.0, smote_params=smote_params)
768 smote_scorers = pipeline.monte_carlo_validation(Xp, Yp, models, splitter, n=50)
769
770
771 # In[3910]:
772
773 for key, scorer in smote_scorers.iteritems():
774     scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)
775
776
777 plt.plot(np.arange(0,1,1, 0.1), np.arange(0,1,1, 0.1), '—')
778 plt.savefig("img/roc_smote.png")
779
780
781 # In[3911]:
782
783 smote_f_scores = f_score_summary(smote_scorers)
784 print smote_f_scores.to_latex()
785 smote_f_scores
786
787
788 # ## Best Classifier
789
790 # In[4060]:
791
792 spiro_features = create_spiro_features(Xp)
793 poly_features = create_poly_features(Xp, ['PRE4', 'PRE5'])
794 Xp_all = pd.concat([Xp, poly_features, spiro_features], axis=1)
795 Xp_all.drop(['DGN_1', 'DGN_8'], axis=1, inplace=True)
796 for model in models:
797     model['train_data'] = (Xp_all, Yp)
798
799 scorers = pipeline.repeated_cross_fold_validation(models, n=10, k=5)
800
801
802 # In[4061]:
803

```

```

804 for key, scorer in scorers.iteritems():
805     scorer.plot_roc_curve(mean_label=key, mean_line=True, show_all=False)
806
807 plt.plot(np.arange(0,1.1, 0.1), np.arange(0,1.1, 0.1), '—')
808
809
810 # In[3981]:
811
812 f_scores = f_score_summary(scorers)
813 f_scores
814
815
816 # ## Prediction on Test Set
817 #
818 # Finally, based on the best combination of techniques used in the preceeding sections, and using the
819 # classifier with the best AUC performance, make probalistic predictions based on the unlabelled test
820 # data.
821
822 # In[4083]:
823
824 Xtest, _ = preprocess(test, y_data=None)
825 Xtest = Xtest.drop('test_id', axis=1)
826
827 test_spiro_features = create_spiro_features(Xtest)
828 test_poly_features = create_poly_features(Xtest, ['PRE4', 'PRE5'])
829 Xtest = pd.concat([Xtest, test_spiro_features, test_poly_features], axis=1)
830
831 print Xtest.columns.size, Xp_all.columns.size
832
833
834 # In[4084]:
835
836 gbc_final = models[3]['model']
837 gbc_final.fit(Xp_all, Yp)
838 predicted_prob = pd.Series(gbc_final.predict_proba(Xtest)[:, 0])
839
840
841 # In[4085]:
842
843 predicted_label = predicted_prob.copy()
844 predicted_label[predicted_label >= 0.5] = 1
845 predicted_label[predicted_label < 0.5] = 0
846
847 final_submission = pd.concat([test.test_id, predicted_label, predicted_prob], axis=1)
848 final_submission.columns = ['test_id', 'predicted_label', 'predicted_output']
849 final_submission

```

./src/analysis.py

A. ROC Analysis Module

```

1 from sklearn import metrics
2 import numpy as np
3 from scipy import interp
4 import matplotlib.pyplot as plt
5 from scipy.stats import threshold
6
7 class ROCAnalysisScorer:
8     def __init__(self):
9         self.rates_ = []
10        self.aucs_ = []
11        self.fhalf_scores_ = []
12        self.f2scores_ = []
13        self.f1scores_ = []
14
15    def __call__(self, ground_truth, predictions, **kwargs):
16        return self.auc_score(ground_truth, predictions, **kwargs)
17
18    def auc_score(self, ground_truth, predictions, **kwargs):
19        thresholded = threshold(predictions[:, 1], threshmin=0.5)
20        thresholded = threshold(thresholded, threshmax=0.5, newval=1.0).astype(int)
21        fhalf_score = metrics.fbeta_score(ground_truth.astype(int), thresholded, beta=0.5)
22        f2_score = metrics.fbeta_score(ground_truth.astype(int), thresholded, beta=2)
23        f1_score = metrics.fbeta_score(ground_truth.astype(int), thresholded, beta=1)
24

```

```

25     fpr, tpr, _ = metrics.roc_curve(ground_truth, predictions[:, 1])
26     area = metrics.auc(fpr, tpr)
27
28     self.fhalf_scores_.append(fhalf_score)
29     self.f2scores_.append(f2_score)
30     self.f1scores_.append(f1_score)
31     self.rates_.append((fpr, tpr))
32     self.aucs_.append(area)
33     return area
34
35 def mean_roc_metrics(self):
36     mean_tpr = 0.0
37     mean_fpr = np.linspace(0, 1, 100)
38
39     for fpr, tpr in self.rates_:
40         mean_tpr += interp(mean_fpr, fpr, tpr)
41
42     mean_tpr = mean_tpr / len(self.rates_)
43     area = metrics.auc(mean_fpr, mean_tpr)
44     return mean_fpr, mean_tpr, area
45
46
47 def plot_roc_curve(self, title=None, labels=None, show_all=True, chance_line=False, mean_line=False,
48 mean_label="Mean"):
49     if show_all:
50         for i, ((fpr, tpr), area) in enumerate(zip(self.rates_, self.aucs_)):
51             if labels is None:
52                 name = "ROC %d" % (i+1)
53             elif isinstance(labels, list):
54                 name = labels[i]
55             elif isinstance(labels, str):
56                 name = labels
57             plt.plot(fpr, tpr, label='%s AUC = %0.4f' % (name, area))
58
59     if mean_line and len(self.rates_) > 1:
60         mean_fpr, mean_tpr, area = self.mean_roc_metrics()
61         plt.plot(mean_fpr, mean_tpr, label="%s, AUC: %0.4f" % (mean_label, area))
62
63     if chance_line:
64         line = np.arange(0, 1.1, 0.1)
65         plt.plot(line, line, "-", label="Chance")
66
67     if title is None:
68         title = 'Receiver operating characteristic'
69
70     plt.title(title)
71     plt.xlabel('False Positive Rate')
72     plt.ylabel('True Positive Rate')
73     plt.legend(loc="lower right", prop={'size': 10})
74
75 def plot_confusion_matrix(cm, title='Confusion matrix', cmap=plt.cm.Blues):
76     plt.imshow(cm, interpolation='nearest', cmap=cmap)
77
78     width, height = cm.shape
79     for x in xrange(width):
80         for y in xrange(height):
81             plt.annotate(str(cm[x][y]), xy=(y, x),
82                          horizontalalignment='center',
83                          verticalalignment='center', size=20)
84
85     plt.title(title)
86     plt.colorbar()
87     plt.show()
88

```

../src/roc_analysis.py

B. Pipeline Module

```

1 from roc_analysis import ROCAnalysisScorer
2 from sklearn import cross_validation
3 from sklearn.metrics import make_scorer
4 from unbalanced_dataset import SMOTE, UnderSampler, OverSampler
5

```



```

6 def cv_pipeline(model, x_data, y_data, cv=None):
7     """ Cross Validate a model pipeline
8
9     Args:
10         model: A sklearn Pipeline object to cross validate
11         x_data: Feature matrix
12         y_data: Class labels
13         cv: A predefined sklearn cross validation object
14     Returns:
15         A ROCAnalysisScorer object with the true/false positive rates and AUCs for all folds
16     """
17     roc_data = ROCAnalysisScorer()
18     roc_data_scorer = make_scorer(roc_data, greater_is_better=True, needs_proba=True, average='weighted')
19     cross_validation.cross_val_score(model, x_data, y_data, cv=cv, scoring=roc_data_scorer)
20     return roc_data
21
22 def test_pipeline(model, x_data, y_data, x_test, y_test):
23     model.fit(x_data, y_data)
24     y_hat = model.predict_proba(x_test)
25
26     test_result = ROCAnalysisScorer()
27     test_result.auc_score(y_test, y_hat)
28     return test_result
29
30
31 def score_pipeline(data, cv=None):
32     model = data['model']
33
34     cv_results = None
35     test_results = None
36
37     x_data, y_data = data['train_data']
38
39     if cv is not None:
40         cv_results = cv_pipeline(model, x_data, y_data, cv=cv)
41
42     if 'test_data' in data:
43         x_test, y_test = data['test_data']
44         test_results = test_pipeline(model, x_data, y_data, x_test, y_test)
45
46     return (cv_results, test_results)
47
48
49 def repeated_cross_fold_validation(models, n=10, k=5):
50     scorers = {}
51
52     for i in range(n):
53         # create a new cross validation set for each iteration & test.
54         skf = cross_validation.StratifiedKFold(models[0]['train_data'][1], n_folds=k)
55
56         for model in models:
57             model_name = model['name']
58             if model_name not in scorers:
59                 scorers[model_name] = ROCAnalysisScorer()
60
61             results = score_pipeline(model, cv=skf)
62
63             # for each model collect the results into a single scorer.
64             # note: no average is made at this stage. The results of each
65             # of the k folds is collected into a single k * n list for
66             # the model.
67             scorers[model_name].f1scores_ += results[0].f1scores_
68             scorers[model_name].f2scores_ += results[0].f2scores_
69             scorers[model_name].fhalf_scores_ += results[0].fhalf_scores_
70             scorers[model_name].rates_ += results[0].rates_
71             scorers[model_name].aucs_ += results[0].aucs_
72
73     return scorers
74
75
76 def monte_carlo_validation(x_data, y_data, models, splitter, n=10):
77     scorers = {}
78
79     for i in range(n):
80         x_train, y_train, x_valid, y_valid = splitter.split(x_data, y_data)
81
82         for model in models:

```

```

83     model_name = model['name']
84     if model_name not in scorers:
85         scorers[model_name] = ROCAnalysisScorer()
86
87     model['train_data'] = (x_train, y_train)
88     model['test_data'] = (x_valid, y_valid)
89
90     results = score_pipeline(model)
91
92     # for each model collect the results into a single scorer.
93     # note: no average is made at this stage. The results of each
94     # of the k folds is collected into a single k * n list for
95     # the model.
96     scorers[model_name].f1scores_ += results[1].f1scores_
97     scorers[model_name].f2scores_ += results[1].f2scores_
98     scorers[model_name].fhalf_scores_ += results[1].fhalf_scores_
99     scorers[model_name].rates_ += results[1].rates_
100    scorers[model_name].aucs_ += results[1].aucs_
101
102    return scorers
103
104
105    class TestSplitter(object):
106
107        def __init__(self, test_size=0.2):
108            self._test_size = test_size
109
110        def split(self, x_data, y_data):
111            Xtest, Xvalid = cross_validation.train_test_split(x_data, test_size=self._test_size, stratify=
y_data)
112            Ytest, Yvalid = y_data.loc[Xtest.index], y_data.loc[Xvalid.index]
113            return Xtest, Ytest, Xvalid, Yvalid
114
115
116    class SMOTESplitter(TestSplitter):
117
118        def __init__(self, under_sample=1.0, smote_params={}, **kwargs):
119            super(SMOTESplitter, self).__init__(**kwargs)
120            self._under_sample = under_sample
121            self._smote_params = smote_params
122
123        def split(self, x_data, y_data):
124            Xt, Yt, Xv, Yv = super(SMOTESplitter, self).split(x_data, y_data)
125            Xt_smote, Yt_smote = SMOTE(**self._smote_params).fit_transform(Xt.as_matrix(), Yt.as_matrix())
126            Xt_smote, Yt_smote = UnderSampler(ratio=self._under_sample).fit_transform(Xt_smote, Yt_smote)
127            return Xt_smote, Yt_smote, Xv, Yv
128
129    class OverUnderSplitter(TestSplitter):
130
131        def __init__(self, under_sample=1.0, over_sample=1.0, **kwargs):
132            super(OverUnderSplitter, self).__init__(**kwargs)
133            self._under_sample = under_sample
134            self._over_sample = over_sample
135
136        def split(self, x_data, y_data):
137            Xt, Yt, Xv, Yv = super(OverUnderSplitter, self).split(x_data, y_data)
138            Xt_smote, Yt_smote = OverSampler(ratio=self._over_sample).fit_transform(Xt.as_matrix(), Yt.
as_matrix())
139            Xt_smote, Yt_smote = UnderSampler(ratio=self._under_sample).fit_transform(Xt_smote, Yt_smote)
140            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
53 52,0,0.0228209667991
54 53,1,0.789979159422
55 54,1,0.592845245625
56 55,0,0.468246812012
57 56,0,0.124413267301
58 57,1,0.762847930744
59 58,1,0.66547093919
60 59,0,0.0957612397443
61 60,0,0.361182401436
62 61,1,0.876937602066
63 62,1,0.816143971828
64 63,0,0.0190173595981
65 64,0,0.0792392458244
66 65,1,0.707652072617
67 66,0,0.496436132178
68 67,1,0.951665094386
69 68,0,0.288903303383
70 69,1,0.70995081876
71 70,0,0.224498004322
72 71,1,0.9183269844
73 72,1,0.937519404279
74 73,1,0.786448314956
75 74,1,0.88587461236
76 75,0,0.393104262221
77 76,1,0.845700917349
78 77,0,0.248636740584
79 78,0,0.346568635705
80 79,1,0.756725311104
81 80,1,0.724599547024
82 81,1,0.759772839791
83 82,1,0.763201534397
84 83,1,0.93771418578
85 84,0,0.271315754528

```
86 85,0,0.0879091265642
87 86,0,0.475492266561
88 87,1,0.773400467965
89 88,1,0.741701905905
90 89,0,0.344979199097
91 90,1,0.817128811821
92 91,0,0.421556736447
93 92,1,0.920134078405
94 93,1,0.945645950574
95 94,1,0.691924546397
96 95,1,0.82519102713
97 96,0,0.0390692900319
98 97,1,0.702271942814
99 98,1,0.919053885818
100 99,1,0.81743386827
101 100,0,0.199586488799
```

../data/submission.csv