Performance Improvement via Ensemble Methods on Imbalanced Data: Polish Bankruptcy Case

Faruk Buldur, *MSc. in Data Science, S025445,* M. Furkan Oruc, *MSc. in Civil Engineering, S025464*

***Abstract*—This paper presents different methods to improve performance of machine learning algorithms when we are dealing with the imbalanced data. Since most of the real life applications require working with imbalanced data sets, focusing on the methods for performance improvement of machine learning algorithms on imbalanced data is worth it. The highest performer has been Tree Based EasyEnsemble, also followed by Random Forest based SMOTE and Logistic Regression based RusBoost. In this project, we have created data sets with different balance ratios by using famous fashion-mnist data. We have tried our candidates on fashion-mnist to see the improvement. After being sure that methods we used on fashion-mnist brings us a significant improvement, we directed towards a real life problem bankruptcy prediction in Polish firms. Evaluation outcomes on both data sets are benchmarked with simple and ensemble learning algorithms, as well as literature[1].**

***Index Terms*—machine learning, imbalance, Tree Based EasyEnsemble, Polish Bankruptcy, SMOTE,**

I. Introduction

CLASS imbalance problem is a prevalent challenge among numerous data sets through which researchers deal with. Imbalance in a data set can be defined as one or more classes being dominant on other class or classes up to a certain degree. In some cases, imbalance can be irrelevant due to the extremity of features' values, though, in other cases it's a problem researchers have to deal with occasionally, regardless of domain. Class imbalance maybe an issue brought by nature[2], which is widely influenced by many factors coming together in an unbiased fashion. On the other hand, occasions including a human element through the process are also widely prone to class imbalance in various domains. Some of the real world representations of class imbalance can be financial fraud [2], oil spills which can be detected from radar visualizations, manufacturing faults in cases of defections and diagnosis of diseases in the medical domain [2]. Minority classes may carry critical value to be detected in many cases, which is another representation of significance in the proposed problem.

Machine learning algorithms in their simplest forms do not serve the needs of data sets where one of the classes largely outnumber one another. Due to this fact, numerous approaches are being developed, largely classified under three main categories: algorithm based, data based and hybrid ones. [3] Through data based techniques, manipulation of the data set via arranging the number of representation of classes is a largely adopted methodology. This is conducted via either minimizing the largely represented class to a rather moderate number of representations or synthesizing new minority class members through artificial steps, which involve probabilistic optimization due to importance of genuinity's preservation. In the algorithm based perspective, many algorithms such as decision tree, support vector machines, back propagation neural networks and nearest neighbor are reported to be adequate in determination of outcomes.[4] Algorithm based approaches include manipulation and improvement of algorithms, arrangement of parameters in an optimized fashion and modification. Hybrid methods are the ones which provide an opportunity to merge different perspectives and approach imbalance via combining different algorithms and data level approaches. It's also mentioned as ensemble methodology by researchers. [4]

Research in the scope of imbalance data is separated into several perspectives. Some researchers focus on optimization of algorithms and solutions to tackle the problem [4], while some others focus on perfection of measurement efforts in the evaluation process of an application. The nature of imbalance is an important research topic as well. [5]

This paper primarily focuses on different solutions which can be utilized to overcome challenges provided by the imbalance nature of the data in order to improve the performance of classification models on pre-processed and manipulated Fashion-Mnist data set and Polish Bankruptcy data set [4] via ensemble methods and combination of different approaches. An examination of different algorithms such as logistic regression, decision tree and random forest, K Nearest Neighbor under the umbrella of various sampling, bagging and boosting applications such as Adaboost,[6] Rusboost[7] and EasyEnsemble.[3] An approach in evaluation of imbalance data focused solutions is also provided. Different algorithms are compared under the same and different conditions as well.

Oversampling and under sampling can be defined as data based manipulation techniques respectively focus on minority class and majority class. Synthetic Minority Oversampling Technique (SMOTE) and Random Under Sampling[7] Techniques are utilized. Both methodologies are utilized with the same and different base algorithms to enable comparison under different evaluation metrics. In order to compare with oversampling and under sampling, Boosting technique is also utilized. AdaBoost[6], one of the most common boosting algorithms is practiced. AdaBoost's operation is based on creation of ensemble methods in an interactive way. AdaBoost boosts performance of machine learning algorithms since it assigns weights to minority classes which are more prone to be misclassified.

This study presents an all around comparison of oversampling and under sampling techniques together with bagging and boosting, as well as algorithms in a comprehensive way.

Balance	# of T-Shirt in Train Set	# of Shirt in Train Set	# of T-Shirt in Test Set	# of Shirt in Test Set
50-50	6000	6000	1000	1000
70-30	6000	2571	1000	429
90-10	6000	667	1000	111
95-5	6000	316	1000	53

Fig. 1. Four Different Data Sets Generated from Fashion-MNIST with Different Balance Ratios

II. Data Sets

We have used two different data sets in this study. One of them is Fashion-MNIST data set and the other is Polish Bankruptcy data set. We were planning to use the MNIST data set for the first part of the project. However, after some research we have changed our focus towards Fashion-MNIST since MNIST data is relatively simpler for the benchmark models even if imbalance introduced in the set. We have chosen Polish Bankrupcty Data as our second data set to see if our proposed methods to improve imbalance bring an improvement to another domain and more practical, imbalance in nature.

A. Fashion-MNIST

Even though it's simplicity , MNIST[8] is a widely used data set in machine learning applications. We were planning to use MNIST as a starting point of our study. However, after we researched on the problem and the suitability of data set for our purposes, we have encountered with another data set provided commonly called as Fashion-MNIST[9].

Fashion-MNIST has its similarities and differences from MNIST data set. MNIST data set includes images of handwritten digits, whereas Fashion-MNIST includes images of fashion products in gray-scale. As it is stated in the referred paper and github page for the data set [1] it is more challenging for same algorithms to achieve similar performances on Fashion-MNIST than MNIST data set. We are delibaretely avoid from MNIST, since we do not want implementation of benchmark algorithms to achieve higher scores and not leave us a room to improve.

Fashion-MNIST has 10 different classes of fashion products. However, for our purpose it is redundant to create a multi class classification problem. Hence, we researched and chose the most confused two classes which are class 0 (T-Shirt/Top) and class 6 (Shirt). After choosing two classes from Fashion-MNIST, we have implemented an algorithm to pick random instances from these two classes to create pre-designated balance ratio between classes. At the end of this procedure, we have created 4 different data sets out of 2 classes of Fashion-MNIST with balance ratios 50%-50%, 70%-30%, 90%-10%, 95%-5%. Number of instances and their imbalance ratios can be seen in Fig. 1

B. Polish Bankruptcy Data Set

The data set is about bankruptcy of Polish companies in manufacturing sector. There are more than ten thousand instances in the data set with 64 different financial features spanning a time period of more than 5 year. Data is imbalanced due to very nature of bankruptcy, most of the companies recorded remains operative, whereas a few goes bankrupt. This imbalanced nature of the problem leaves us a point to be improved. Our main aim in the study to apply the methods that brings us improvement in the Fashion-Mnist data to Polish bankrupcty data in order to see the generalizability of our methods to real-life scenarios.

	Data Set 1	Data Set 2	Data Set 3	Data Set 4	Data Set 5
Attr37	0.389925	0.444117	0.450919	0.453636	0.453636
Attr21	0.230824	0.311019	0.076835	NaN	NaN
Attr27	0.044258	0.069399	0.068076	0.065462	0.065462
Attr60	0.019212	0.053377	0.056365	0.062704	0.062704
Attr45	0.019069	0.053180	0.056270	0.062602	0.062602
Attr24	0.017646	0.022117	NaN	NaN	NaN
Attr64	NaN	NaN	0.021708	0.023591	0.023591
Attr54	NaN	NaN	NaN	0.023591	0.023591

Fig. 2. Ratio of missing values in the data set for features with highest number of missing values

Data in this data set collected from Emerging Markets Information Service [2] as per stated in the paper[1] providing the data set and can be reached from Center for Machine Learning and Intelligent Systems [3]. Emerging Markets Information Service provides same data for a number emerging countries, this leaves us a room to expand our research by comparing our model between countries as a further study.

There are missing values in the data set as it can be seen in Fig. 2 , our expectation was to impute missing values with the state-of-art techniques. We have tried K-Nearest Neighbour imputation which imputes missing data based on its closest neighbours. Furthermore, we have implemented expectation maximization for imputation of missing data which maximizes the expectation of missing data by using means and covariances of attributes [10]. Lastly, we have used another technique called multiple imputation by chained equations. This techniques is basically iterative training procedure on the data to predict missing values[11]. However, none of those techniques gave us a better picture than filling missing values only with the attribute means. We have dropped Attribute 37, since missing ratio is quite high to bring information to our study. Detailed explanation of the features can be seen in Fig. 3. We have tried to improve the performance by accurately imputing missing values, but since this is not the main focus of our research we have applied the best performing method and decided to not spend more time on it.

III. Classification Metrics

To comprehend and interpret the outcome of the implementations, different evaluation metrics are utilized. Table I provides a confusion matrix scheme, which provide the basis

[1] https://github.com/zalandoresearch/fashion-mnist

[2] https://www.emis.com/

[3] http://archive.ics.uci.edu/ml/datasets/Polish+companies+bankruptcy+data

Features of Polish Bankrupcty Data Set	
X1 net profit / total assets	X33 operating expenses / short-term liabilities
X2 total liabilities / total assets	X34 operating expenses / total liabilities
X3 working capital / total assets	X35 profit on sales / total assets
X4 current assets / short-term liabilities	X36 total sales / total assets
X5 [(cash + short-term securities + receivables - short-term liabilities) / (operating expenses - depreciation)] * 365	X37 (current assets - inventories) / long-term liabilities
X6 retained earnings / total assets	X38 constant capital / total assets
X7 EBIT / total assets	X39 profit on sales / sales
X8 book value of equity / total liabilities	X40 (current assets - inventory - receivables) / short-term liabilities
X9 sales / total assets	X41 total liabilities / ((profit on operating activities + depreciation) * (12/365))
X10 equity / total assets	X42 profit on operating activities / sales
X11 (gross profit + extraordinary items + financial expenses) / total assets	X43 rotation receivables + inventory turnover in days
X12 gross profit / short-term liabilities	X44 (receivables * 365) / sales
X13 (gross profit + depreciation) / sales	X45 net profit / inventory
X14 (gross profit + interest) / total assets	X46 (current assets - inventory) / short-term liabilities
X15 (total liabilities * 365) / (gross profit + depreciation)	X47 (inventory * 365) / cost of products sold
X16 (gross profit + depreciation) / total liabilities	X48 EBITDA (profit on operating activities - depreciation) / total assets
X17 total assets / total liabilities	X49 EBITDA (profit on operating activities - depreciation) / sales
X18 gross profit / total assets	X50 current assets / total liabilities
X19 gross profit / sales	X51 short-term liabilities / total assets
X20 (inventory * 365) / sales	X52 (short-term liabilities * 365) / cost of products sold)
X21 sales (n) / sales (n-1)	X53 equity / fixed assets
X22 profit on operating activities / total assets	X54 constant capital / fixed assets
X23 net profit / sales	X55 working capital
X24 gross profit (in 3 years) / total assets	X56 (sales - cost of products sold) / sales
X25 (equity - share capital) / total assets	X57 (current assets - inventory - short-term liabilities) / (sales - gross profit - depreciation)
X26 (net profit + depreciation) / total liabilities	X58 total costs /total sales
X27 profit on operating activities / financial expenses	X59 long-term liabilities / equity
X28 working capital / fixed assets	X60 sales / inventory
X29 logarithm of total assets	X61 sales / receivables
X30 (total liabilities - cash) / sales	X62 (short-term liabilities *365) / sales
X31 (gross profit + interest) / sales	X63 sales / short-term liabilities
X32 (current liabilities * 365) / cost of products sold	X64 sales / fixed assets

Fig. 3. Polish Bankruptcy Data Set Features and Their Descriptions[1]

Confusion Matrix		TRUTH	
		TRUE	FALSE
PREDICTION	TRUE	True Positive	False Positive
	FALSE	False Negative	True Negative

TABLE I
CONFUSION MATRIX

for all classification metrics. Confusion matrix provides on overview of the outcome of a machine learning classification in the most comprehensive way. On the other hand, metrics provided by a confusion matrix provides significance in the sense it provides hidden truths behind the data, which is a crucial subject in the nature of imbalance. However, it is not an efficient way of presenting the outcomes for multiple methods and multiple data sets. There are metrics to summarize data inside confusion matrix and present in more compact way. However, summarization comes with its cost of loosing information. Each metric presented below has its pros and cons in different applications.

A. Accuracy

Accuracy can be defined as the most commonly used metric. On the other hand, it's important to mention that accuracy can be prone to misrepresentation in imbalance situations. Although it's common, accuracy may not be a means of identification of success in an imbalanced classification application. Accuracy provides biases results since it's quite dependant on the majority class's number of members. In the case where an algorithm makes moderate predictions, accuracy metric misrepresents the outcome significantly.[12]

$$Accuracy = \left(\frac{TruePositive + TrueNegative}{TotalPopulation} \right) \quad (1)$$

B. Precision

Precision indicates the correctly assigning capability of an algorithm via scaling all positive assigning behaviour. In other words, precision defines how picky an algorithm is, in the process of assigning the instances to the true class. In the scope of this project, true class is identified as the minority class. Assignment of true class as minority class helps identification of purity in the minority assignments.[12]

$$Precision = \left(\frac{TruePositive}{TruePositive + FalsePositive} \right) \quad (2)$$

C. Recall

Recall indicates the correctly assigning capability of an algorithm via scaling all minority instances in the denominator as follows. Recall is an important indicator of success in the scope of this paper since it's crucial to detect minority class among the majority instances and recall is the true indicator in this sense.

$$Recall = \left(\frac{TruePositive}{TruePositive + FalseNegative} \right) \quad (3)$$

D. Roc-Auc Score

Roc-Auc is based on Reciever Operator Characteristics curve which is set up based on a probabilistic curve indicating as a reference in imbalance cases. score indicates how good each class is separated within each other as a result of the classification. It's a significant method since it's one of the least biased evaluation metrics to imbalance. [13] Area under the curve is an evaluation of how well an algorithm makes its prediction.

In the light of the detailed explanation of classification metrics presented above, we will report AUC score and Recall score along with this study. Since our main purpose is to improve the performance of algorithms in classifying minority class correctly in the case of imbalance, recall score is the correct metric to track. Besides, we want to keep track of our algorithm's overall performance while improving performance on minority class. Accuracy is not a good candidate to track since its highly prone to imbalance. Thus, we will report AUC score to track the overall performance of our models.

IV. BENCHMARK METHODS

Our main in this study is to firstly show that baseline, generally accepted machine learning algorithms are prone to imbalance and secondly present some methods to improve this baseline level. Therefore, we have chosen a number of algorithms to create a benchmark to our study. We have used perceptron classifier, random forest classifier and multi layer perceptron with one hidden layer and twenty-five hidden units.

After creating the benchmark in order to maximize the prediction recall, as well as AUC Score, a combination of learning algorithms have been applied to both data sets. Through these combinations, both data level and algorithm level manipulations and improvements have been ensembled.

A. Machine Learning Algorithms

Tree based learning algorithms are widely accepted as one of the most reliable approaches to imbalanced data sets. Besides Random Forest and Decision Trees implemented with Boosting and Bagging [14], Logistic Regression, K Nearest Neighbor, Single Layer Perceptron and Multi Layer Perceptron have been also combined with other learning approaches. Utilization of algorithms via combining with hybrid methods with a wide set of evaluation metrics helped identification of the most efficient predictors in a comparison.

V. Improvement Methods

Through improvement process, machine learning algorithms are combined with various data level and algorithm level methodologies to achieve further significant results and reach interpretations which may provide value[15]. Below some of those approaches enabling ensemble are provided.

A. Bagging

Bootstrap Aggregating or Bagging can be identified as the methodology through which multiple learners are activated in different parts of the data set and by means of these activators, an aggregation of prediction is conducted. [14] Normally, even though bootstrapping creates different samples among the data via replacement, aggregation can be interpreted as a process through which number of votes are considered as the decision parameter through these different learners. K Nearest Neighbors is mentioned to be one of the most stable methods combined with bagging.

B. Boosting

AdaBoost applies an ensembling method in an iterative fashion.[6] Right after assignment of equal weights to each instance of the training set, weights are updated to achieve the goal of successfully classifying the instance. The weighting process is conducted based on if either misclassification occurs or not. In the end, the entire training model applies a voting system to decide which test instance should be labelled with which class. In the real world, other than skewed being, many challenges are brought with imbalanced cases, such as small data set or overlapping issues. Due to combined challenges, traditional boosting approaches such as AdaBoost is generally combined with oversampling or undersampling methodologies[16] to achieve significant results in imbalance cases. GradientBoost can be also expressed as another strong boosting method though it's not utilized in the scope of this paper.[8]

C. Oversampling and Undersampling Methods

Oversampling and Undersampling are a part of data level approaches. Data sampling techniques balance training samples in imbalanced data sets towards a more equally represented fashion. This process is either conducted via adding samples to minority class or subtracting samples from the majority class. Through this study, SMOTE (Synthetic Minority Oversampling Technique)[17] and RUSBoost[7] are utilized in means of data sampling, combined with different machine learning algorithms. SMOTE can be defined as the process trough which synthetic examples of the minority class is generated, instead of basic replication or any other means. This oversampling process has been inspired by handwritten images data generation process. RUSBoost can be defined as a random under sampling technique based on SMOTEBoost algorithm[17]. RUSBoost achieves a higher prediction accuracy via reducing the computational time through decreasing the size of the data set. The only major difference of it can be mentioned as its lack of necessity in weight initialization through the under sampling process.

D. Ensemble

Ensemble approaches utilize traditional algorithm based methods such as boosting and bagging [18] with data based methodologies which are mainly either oversampling or under sampling of data set via different methodologies. Ensemble approach is favored by researchers in the most recent studies as the most reliable methodology in approaching imbalanced data sets.[14]

E. Hybrid Ensemble

EasyEnsemble[3] is also an Undersampling methodology through which performs sample creations from majority class and for each sample, a different classifier is built by it. What makes EasyEnsemble important in the scope of this research is its approach in undersampling, a sensitivity towards data being useful in many aspects and undersampling's impact on this value. Later on, EasyEnsemble utilizes AdaBoost by means of a voting process to make up the final decision.[15] In other words, by means of utilization of voting system, EasyEnsemble focuses on majority class to perform assignments. The process of utilizing AdaBoost makes EasyEnsemble an ensemble of ensembles. One important aspect of EasyEnsemble can be also assessed as not overlooking the important majority class samples, which create significant impact in seperation of classes by various researchers. EasyEnsemble is also defined as a hybrid ensemble[15]and it combines both boosting and bagging with a preprocess.

Through the applications, oversampling and undersampling techniques are combined with tree, logistic regression, perceptron and KNN based approaches. One disadvantage of data level approaches can be expressed as dependency to the subject dataset.

VI. Results

Above mentioned algorithms are trained and tested in two datasets. Manipulated and binary prepared Fashion-MNIST

dataset is utilized as a gateway dataset to test hypotheses before testing algorithms with Polish Bankruptcy dataset. The experiment has been conducted via Python.

While simple machine learning algorithms provide non satisfactory evaluation metrics as ROC-AUC score, precision and recall, ensemble methods provide significant increase in all of the metrics. Among all, Tree Based EasyEnsemble provides top rates of Roc-Auc Score, Precision and Accuracy among almost all approaches. EasyEnsemble is compared with SMOTE, Adaboost, RusBoost, Bagging with different base algorithms and more than 15 different ensemble and machine learning applications. EasyEnsemble presented the best outcome among all in evaluation with Roc-Auc score, precision and recall. Random Forest based SMOTE also proved to be a solid algorithm in the sense of imbalanced datasets, which is proved by both results in two datasets.

The highest Roc-Auc score by tree based EasyEnsemble results in 90-10 balance rate in the Fashion-MNIST data set.It's also important to note that this outcome is even higher than 70-30 balance, which is another interpretation of EasyEnsemble being significantly suitable to imbalanced cases. Tree based EasyEnsemble provided the top recall score in Fashion-MNIST results as well. It can be also stated that being recall score an important parameter in identification of success in imbalanced data sets, EasyEnsemble can be identified as the unbiased predictor among presented algorithms. Logistic Regression based EasyEnsemble also provided significant results, though very close to tree based, an overall comparison proved that tree based EasyEnsemble presents the most accurate outcome.

A. Results on Fashion-MNIST

We have presented our results on Fashion-MNIST data by reporting firstly AUC scores in Fig. 4 and secondly Recall scores in Fig. 5 as the reasons discussed in the classification metrics section. Fashion-MNIST has provided solid comparisons with different learning based ensembles as well as different data level approaches to the imbalanced data set.

Fashion-MNIST has been a test step in order to efficiently test hypotheses and to reduce complexity. By means of Fashion-MNIST's practicality, various ensembles have been tested with Fashion-MNIST. Even though Fashion-MNIST has been a data set which is artificially designed to be prone to imbalance, evaluation outcomes of Fashion-MNIST has provided consistent outcomes with naturally imbalanced Polish Bankruptcy data set. Artficial imbalancing process did not result with a bias, which might be also a point to dig deeper in the future.

B. Results on Polish Bankruptcy

We have presented our results on Polish Data Set by reporting firstly AUC scores in Fig. 6 and secondly Recall scores in Fig. 7 as the reasons discussed in the classification metrics section.

While we are testing our baseline methods on Polish Data Set, we have implemented Stratified 10-Fold cross validation. Since our data is imbalanced stratification needed to assess the quality of our model and since number of instances in each data set is not enough to measure the quality in seperate train and test splits we used 10-Fold cross validation[19]. We have reported mean and standard deviation of the metrics acquired from different folds. Best performing method is the random forest as a baseline method. Multi layer perceptron and single layer perceptron was not be able to produce meaningful results prior to improvements offered. Using SMOTE to improve the quality resulted in significant improvement in the mean auc score for each data set. Using tree based easy ensemble and tree based RusBoost resulted in much better auc scores.

We can see improvement in the recall score in Fig. 7. Three different approaches to handle imbalance in the data provides us a significant improvement.

Balance	50-50	70-30	90-10	95-5
MLP	0.859	0.824143	0.780293	0.782953
MLP SMOTE	0.859	0.832763	0.828887	0.745255
SLP	0.824	0.704179	0.679680	0.509434
SLP SMOTE	0.824	0.832763	0.828887	0.745255
RF	0.836	0.812340	0.759261	0.725415
RF SMOTE	0.836	0.832763	0.828887	0.745255
LR Based Rusboost	0.842	0.834767	0.838396	0.815358
Tree Based Easy Ensemble	0.827	0.832598	0.841905	0.816226
LR Based Easy Ensemble	0.833	0.834263	0.837910	0.797792
Adaboost Mean	0.831	0.813978	0.790802	0.776519
Tree Based RusBoost Mean	0.827	0.832763	0.828887	0.745255

Fig. 4. ROC-AUC scores of the offered improvement models on Fashion-MNIST data set with different level of imbalance

Balance	50-50	70-30	90-10	95-5
MLP	0.823	0.713287	0.585586	0.584906
MLP SMOTE	0.823	0.806527	0.774775	0.641509
SLP	0.741	0.974359	0.360360	0.018868
SLP SMOTE	0.741	0.806527	0.774775	0.641509
RF	0.782	0.652681	0.522523	0.452830
RF SMOTE	0.782	0.806527	0.774775	0.641509
LR Based Rusboost	0.831	0.799534	0.792793	0.754717
Tree Based Easy Ensemble	0.815	0.804196	0.810811	0.792453
LR Based Easy Ensemble	0.817	0.806527	0.819820	0.773585
Adaboost Mean	0.825	0.710956	0.603604	0.566038
Tree Based RusBoost Mean	0.815	0.806527	0.774775	0.641509

Fig. 5. Recall scores of the offered improvement models on on Fashion-MNIST data set with different level of imbalance

VII. Conclusion

In summary, this study shows that Tree Based EasyEnsembling is the best performing ensemble among different combinations of learning methods, which include data level, algorithm level and hybrid methods, as well as a hybrid ensemble in the scope of this study. Due to the fact that EasyEnsembling

Data Sets	Data Set 1	Data Set 2	Data Set 3	Data Set 4
MLP Mean	0.500000	0.499949	0.499900	0.504360
MLP STD	0.000000	0.000154	0.000200	0.006594
MLP SMOTE Mean	0.680908	0.651633	0.700095	0.721761
SLP Mean	0.499926	0.505069	0.499950	0.517512
SLP STD	0.000222	0.015207	0.000150	0.049716
SLP SMOTE Mean	0.518787	0.504489	0.497938	0.564466
RF Mean	0.652524	0.597625	0.572054	0.593091
RF STD	0.033776	0.027187	0.037462	0.026708
RF SMOTE Mean	0.732397	0.688115	0.673225	0.702986
LR Based Rusboost Mean	0.577110	0.500000	0.508620	0.679341
Tree Based Easy Ensemble Mean	0.839197	0.781516	0.785921	0.811797
LR Based Easy Ensemble Mean	0.535616	0.530405	0.622505	0.678886
Adaboost Mean	0.695884	0.574153	0.587430	0.639693
Tree Based RusBoost Mean	0.794653	0.721882	0.747628	0.738981

Fig. 6. ROC-AUC scores of the offered improvement models on Polish Bankruptcy Data Set

Data Sets	Data Set 1	Data Set 2	Data Set 3	Data Set 4
MLP Mean	0.000000	0.000000	0.000000	0.009691
MLP STD	0.000000	0.000000	0.000000	0.013070
MLP SMOTE Mean	0.686111	0.675000	0.737224	0.679374
SLP Mean	0.000000	0.017500	0.000000	0.090196
SLP STD	0.000000	0.052500	0.000000	0.264117
SLP SMOTE Mean	0.896296	0.045000	0.993878	0.651131
RF Mean	0.306085	0.197500	0.147306	0.188122
RF STD	0.067282	0.054141	0.075274	0.053234
RF SMOTE Mean	0.476190	0.392500	0.369633	0.426885
LR Based Rusboost Mean	0.859259	0.000000	0.040816	0.553243
Tree Based Easy Ensemble Mean	0.845503	0.795000	0.781878	0.800264
LR Based Easy Ensemble Mean	0.180952	0.117500	0.499102	0.512783
Adaboost Mean	0.398280	0.152500	0.181755	0.287255
Tree Based RusBoost Mean	0.719709	0.635000	0.676612	0.645475

Fig. 7. Recall scores of the offered improvement models on Polish Bankruptcy Data Set

being a member of hybrid ensembles family also represents an important aspect regarding ensemble approaches. The study also shows that logistic regression is an important machine learning algorithm in imbalanced data sets focused researches. The study also tests a rather unorthodox methodology in testing imbalanced data focused machine learning ensembles via manipulated Fashion-MNIST data set. The Fashion-MNIST data set provides useful and compatible results with naturally imbalanced Polish Bankruptcy data set, which may help future researchers in the test process of their improving imbalanced data based solutions focused researches.

REFERENCES

[1] M. Zikeba, S. K. Tomczak, and J. M. Tomczak, "Ensemble boosted trees with synthetic features generation in application to bankruptcy prediction," *Expert Systems with Applications*, 2016.

[2] Yanminsun, A. Wong, and M. S. Kamel, "Classification of imbalanced data: a review," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 23, 11 2011.

[3] T. Liu, in *2009 International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing*.

[4] A. Ali, S. M. Shamsuddin, and A. Ralescu, "Classification with class imbalance problem: A review," vol. 7, pp. 176–204, 01 2015.

[5] N. Chawla, N. Japkowicz, and A. Kołcz, "Editorial: Special issue on learning from imbalanced data sets," *SIGKDD Explorations*, vol. 6, pp. 1–6, 06 2004.

[6] T. Chengsheng, L. Huacheng, and X. Bing, "Adaboost typical algorithm and its application research," *MATEC Web of Conferences*, vol. 139, p. 00222, 01 2017.

[7] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "Rusboost: A hybrid approach to alleviating class imbalance," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 40, no. 1, pp. 185–197, 2010.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Intelligent Signal Processing*. IEEE Press, 2001, pp. 306–351.

[9] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017. [Online]. Available: http://arxiv.org/abs/1708.07747

[10] T. Aljuaid and S. Sasi, "Proper imputation techniques for missing values in data sets," 08 2016, pp. 1–5.

[11] M. J. Azur, E. A. Stuart, C. Frangakis, and P. J. Leaf, "Multiple imputation by chained equations: what is it and how does it work?" *International Journal of Methods in Psychiatric Research*, vol. 20, no. 1, pp. 40–49, 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/mpr.329

[12] D. M. W. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2020.

[13] D. McClish, "Analyzing a portion of the roc curve," *Medical decision making : an international journal of the Society for Medical Decision Making*, vol. 9, pp. 190–5, 08 1989.

[14] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, and F. Herrera, "A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 4, pp. 463–484, 2012.

[15] G. Rekha, "A novel approach for solving skewed classification problem using cluster based ensemble method," p. 1, 2020. [Online]. Available: http://aimsciences.org//article/id/6a112bae-7a14-491b-a086-b7d82d2cc59f

[16] X. Liu, J. Wu, and Z. Zhou, "Exploratory undersampling for class-imbalance learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 2, pp. 539–550, 2009.

[17] N. Chawla, K. Bowyer, L. Hall, and W. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Intell. Res. (JAIR)*, vol. 16, pp. 321–357, 06 2002.

[18] L. Breiman, *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996. [Online]. Available: https://doi.org/10.1023/a:1018054314350

[19] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 36, no. 2, pp. 111–147, 1974. [Online]. Available: http://www.jstor.org/stable/2984809

CS554-Imbalance-Improvement

January 20, 2021

1 Importing the Libraries

[47]:
```
from scipy.io import arff
import pandas as pd
from mnist import MNIST
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from package import (data_balancer, MultiLayerPerceptron, softmax, sigmoid,␣
↪Loss_Perceptron,Weight_Initialization,
                        KNeighbors, forest, svm, perceptron)
from sklearn.metrics import (confusion_matrix, recall_score, precision_score,␣
↪roc_auc_score,
                                accuracy_score, f1_score,␣
↪plot_confusion_matrix, confusion_matrix,
                                roc_curve, auc)
from sklearn.model_selection import (cross_val_score, cross_validate,␣
↪train_test_split)
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import Perceptron
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from CUSBoost import CUSBoostClassifier
from rusboost import RusBoost
from sklearn.preprocessing import LabelEncoder, Normalizer, normalize
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import math
import time
sns.set_style('dark')
```

2 Loading Data Sets

2.0.1 Bankruptcy Data of the Polish Companies

[69]:
```
data = arff.loadarff('data/1year.arff')
polish_1 = pd.DataFrame(data[0])
data = arff.loadarff('data/2year.arff')
polish_2 = pd.DataFrame(data[0])
data = arff.loadarff('data/3year.arff')
polish_3 = pd.DataFrame(data[0])
data = arff.loadarff('data/4year.arff')
polish_4 = pd.DataFrame(data[0])
data = arff.loadarff('data/5year.arff')
polish_5 = pd.DataFrame(data[0])
```

[70]:
```
polish_1['class'] = polish_1['class'].apply(lambda x: int(x))
polish_2['class'] = polish_2['class'].apply(lambda x: int(x))
polish_3['class'] = polish_3['class'].apply(lambda x: int(x))
polish_4['class'] = polish_4['class'].apply(lambda x: int(x))
```

3 Performance Before Improvement

3.0.1 Fashion MNIST Data Set with Different Balance Ratios

[4]:
```
np.random.seed(60) # reproducability
mndata = MNIST('./mnist-fashion')

# read training images and corresponding labels
tr_images, tr_labels = mndata.load_training()
# read test images and corresponding labels
tt_images, tt_labels = mndata.load_testing()

# convert lists into numpy format and apply normalization
tr_images = np.array(tr_images) / 255. # shape (60000, 784)
tr_labels = np.array(tr_labels)         # shape (60000,)
tt_images = np.array(tt_images) / 255. # shape (10000, 784)
tt_labels = np.array(tt_labels)         # shape (10000,)
```

[6]:
```
X_train_50, y_train_50, Balance, Balance_Changed_50_tr =␣
↪data_balancer(tr_images, tr_labels, 0, 6, 0.5, 0.5)
X_test_50, y_test_50, Balance, Balance_Changed_50_tt = data_balancer(tt_images,␣
↪tt_labels, 0, 6, 0.5, 0.5)

X_train_70, y_train_70, Balance, Balance_Changed_70_tr =␣
↪data_balancer(tr_images, tr_labels, 0, 6, 0.7, 0.3)
X_test_70, y_test_70, Balance, Balance_Changed_70_tt = data_balancer(tt_images,␣
↪tt_labels, 0, 6, 0.7, 0.3)
```

```
X_train_90, y_train_90, Balance, Balance_Changed_90_tr =␣
 ↪data_balancer(tr_images, tr_labels, 0, 6, 0.9, 0.1)
X_test_90, y_test_90, Balance, Balance_Changed_90_tt = data_balancer(tt_images,␣
 ↪tt_labels, 0, 6, 0.9, 0.1)

X_train_95, y_train_95, Balance, Balance_Changed_95_tr =␣
 ↪data_balancer(tr_images, tr_labels, 0, 6, 0.95, 0.05)
X_test_95, y_test_95, Balance, Balance_Changed_95_tt = data_balancer(tt_images,␣
 ↪tt_labels, 0, 6, 0.95, 0.05)

print(f'{(Balance_Changed_50_tr,Balance_Changed_50_tt)} \n\
{(Balance_Changed_70_tr,Balance_Changed_70_tt)} \n\
{(Balance_Changed_90_tr,Balance_Changed_90_tt)} \n\
{(Balance_Changed_95_tr,Balance_Changed_95_tt)}')
```

```
((0.5, 0.5), (0.5, 0.5))
((0.7, 0.3), (0.7, 0.3))
((0.9, 0.1), (0.9, 0.1))
((0.95, 0.05), (0.95, 0.05))
```

[68]:
```
num_0_train = [(y_train_50 == 0).sum(), (y_train_70 == 0).sum(), (y_train_90 ==␣
 ↪0).sum(), (y_train_95 == 0).sum()]
num_0_test = [(y_test_50 == 0).sum(), (y_test_70 == 0).sum(), (y_test_90 == 0).
 ↪sum(), (y_test_95 == 0).sum()]

num_6_train = [(y_train_50 == 6).sum(), (y_train_70 == 6).sum(), (y_train_90 ==␣
 ↪6).sum(), (y_train_95 == 6).sum()]
num_6_test = [(y_test_50 == 6).sum(), (y_test_70 == 6).sum(), (y_test_90 == 6).
 ↪sum(), (y_test_95 == 6).sum()]

instance = pd.concat([pd.Series(num_0_train, name='# of T-Shirt in Train Set'),
                      pd.Series(num_6_train, name='# of Shirt in Train Set'),
                      pd.Series(num_0_test, name='# of T-Shirt in Test Set'),
                      pd.Series(num_6_test, name='# of Shirt in Test Set'),
                      pd.Series(['50-50', '70-30', '90-10', '95-5'],␣
 ↪name='Balance')], axis=1)

instance.set_index('Balance', inplace=True)
display(instance)
```

```
         # of T-Shirt in Train Set  # of Shirt in Train Set  \
Balance
50-50                         6000                     6000
70-30                         6000                     2571
90-10                         6000                      667
95-5                          6000                      316
```

Balance	# of T-Shirt in Test Set	# of Shirt in Test Set
50-50	1000	1000
70-30	1000	429
90-10	1000	111
95-5	1000	53

3.1 On Fashion MNIST

3.1.1 Multilayer Perceptron

[146]:
```
mlp_50 = MLPClassifier(hidden_layer_sizes=25, verbose=False).fit(X_train_50,␣
 ↪y_train_50)
y_pred_50 = mlp_50.predict(X_test_50)
Precision_50, Recall_50, ROC_AUC_50 = precision_score(y_test_50, y_pred_50,␣
 ↪pos_label=6),recall_score(y_test_50, y_pred_50, pos_label=6),␣
 ↪roc_auc_score(y_test_50, y_pred_50)

mlp_70 = MLPClassifier(hidden_layer_sizes=25, verbose=False).fit(X_train_70,␣
 ↪y_train_70)
y_pred_70 = mlp_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

mlp_90 = MLPClassifier(hidden_layer_sizes=25, verbose=False).fit(X_train_90,␣
 ↪y_train_90)
y_pred_90 = mlp_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

mlp_95 = MLPClassifier(hidden_layer_sizes=25, verbose=False).fit(X_train_95,␣
 ↪y_train_95)
y_pred_95 = mlp_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)


MLP_AUC = [ROC_AUC_50,ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
MLP_Recall = [Recall_50, Recall_70, Recall_90, Recall_95]
MLP_Precision = [Precision_50, Precision_70, Precision_90, Precision_95]
```

3.1.2 K-Neighbours Classifier

[13]:
```
Accuracy_50_value, Roc_Auc_50_value, Misclassified_Count_50_value =␣
 ↪KNeighbors(X_train_50, y_train_50, X_test_50, y_test_50)
Accuracy_70_value, Roc_Auc_70_value, Misclassified_Count_70_value =␣
 ↪KNeighbors(X_train_70, y_train_70, X_test_70, y_test_70)
Accuracy_90_value, Roc_Auc_90_value, Misclassified_Count_90_value =␣
 ↪KNeighbors(X_train_90, y_train_90, X_test_90, y_test_90)
Accuracy_95_value, Roc_Auc_95_value, Misclassified_Count_95_value =␣
 ↪KNeighbors(X_train_95, y_train_95, X_test_95, y_test_95)

KNN_AUC = [Roc_Auc_50_value, Roc_Auc_70_value, Roc_Auc_90_value,␣
 ↪Roc_Auc_95_value]
KNN_Accuracy = [Accuracy_50_value, Accuracy_70_value, Accuracy_90_value,␣
 ↪Accuracy_95_value]
```

3.1.3 Random Forest Classifier

[145]:
```
RF_50 = RandomForestClassifier(criterion = 'entropy', n_estimators = 10,␣
 ↪random_state = 60).fit(X_train_50, y_train_50)
y_pred_50 = RF_50.predict(X_test_50)
Precision_50, Recall_50, ROC_AUC_50 = precision_score(y_test_50, y_pred_50,␣
 ↪pos_label=6),recall_score(y_test_50, y_pred_50, pos_label=6),␣
 ↪roc_auc_score(y_test_50, y_pred_50)


RF_70 = RandomForestClassifier(criterion = 'entropy', n_estimators = 10,␣
 ↪random_state = 60).fit(X_train_70, y_train_70)
y_pred_70 = RF_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

RF_90 = RandomForestClassifier(criterion = 'entropy', n_estimators = 10,␣
 ↪random_state = 60).fit(X_train_90, y_train_90)
y_pred_90 = RF_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

RF_95 = RandomForestClassifier(criterion = 'entropy', n_estimators = 10,␣
 ↪random_state = 60).fit(X_train_95, y_train_95)
y_pred_95 = RF_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)
```

```
RF_AUC = [ROC_AUC_50, ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
RF_Recall = [Recall_50, Recall_70, Recall_90, Recall_95]
RF_Precision = [Precision_50,Precision_70, Precision_90, Precision_95]
```

3.1.4 Perceptron Classifier

[157]:
```
SLP_50 = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
 ↪random_state = 60).fit(X_train_50, y_train_50)
y_pred_50 = SLP_50.predict(X_test_50)
Precision_50, Recall_50, ROC_AUC_50 = precision_score(y_test_50, y_pred_50,␣
 ↪pos_label=6),recall_score(y_test_50, y_pred_50, pos_label=6),␣
 ↪roc_auc_score(y_test_50, y_pred_50)

SLP_70 = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
 ↪random_state = 60).fit(X_train_70, y_train_70)
y_pred_70 = SLP_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

SLP_90 = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
 ↪random_state = 60).fit(X_train_90, y_train_90)
y_pred_90 = SLP_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

SLP_95 = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
 ↪random_state = 60).fit(X_train_95, y_train_95)
y_pred_95 = SLP_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)


SLP_AUC = [ROC_AUC_50,ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
SLP_Recall = [Recall_50, Recall_70, Recall_90, Recall_95]
SLP_Precision = [Precision_50, Precision_70, Precision_90, Precision_95]
```

3.1.5 Summary & Results

[158]:
```
AUC_DF = pd.concat([pd.Series(MLP_AUC, name='MLP'),
                    pd.Series(SLP_AUC, name='SLP'),
                    pd.Series(RF_AUC, name='RF'),
                    #pd.Series(SVM_AUC, name='SVM'),
                    #pd.Series(KNN_AUC, name='KNN'),
                    pd.Series(['50-50', '70-30', '90-10', '95-5'],␣
 ↪name='Balance')], axis=1)

AUC_DF.set_index('Balance', inplace=True)
display(AUC_DF)
```

Balance	MLP	SLP	RF
50-50	0.859000	0.824000	0.836000
70-30	0.824143	0.704179	0.812340
90-10	0.780293	0.679680	0.759261
95-5	0.782953	0.509434	0.725415

[159]:
```
Recall_DF = pd.concat([pd.Series(MLP_Recall, name='MLP'),
                    pd.Series(SLP_Recall, name='SLP'),
                    pd.Series(RF_Recall, name='RF'),
                    #pd.Series(SVM_Recall, name='SVM'),
                    #pd.Series(KNN_Recall, name='KNN'),
                    pd.Series(['50-50', '70-30', '90-10', '95-5'],␣
 ↪name='Balance')], axis=1)

Recall_DF.set_index('Balance', inplace=True)
display(Recall_DF)
```

Balance	MLP	SLP	RF
50-50	0.823000	0.741000	0.782000
70-30	0.713287	0.974359	0.652681
90-10	0.585586	0.360360	0.522523
95-5	0.584906	0.018868	0.452830

[160]:
```
Precision_DF = pd.concat([pd.Series(MLP_Precision, name='MLP'),
                    pd.Series(SLP_Precision, name='SLP'),
                    pd.Series(RF_Precision, name='RF'),
                    #pd.Series(SVM_Precision, name='SVM'),
                    #pd.Series(KNN_Precision, name='KNN'),
                    pd.Series(['50-50', '70-30', '90-10', '95-5'],␣
 ↪name='Balance')], axis=1)
```

```
Precision_DF.set_index('Balance', inplace=True)
display(Precision_DF)
```

Balance	MLP	SLP	RF
50-50	0.886853	0.888489	0.876682
70-30	0.824798	0.424797	0.909091
90-10	0.722222	0.975610	0.935484
95-5	0.620000	1.000000	0.923077

3.2 On Polish Bankruptcy Data Set

3.2.1 Handling Missing Values in the Data

[71]:
```
polish_1_mean_filled = polish_1.fillna(polish_1.mean(axis=0), axis=0)
y_1 = polish_1_mean_filled['class']
X_1 = polish_1_mean_filled.drop(['class','Attr37'], axis=1)
X_1 = normalize(X_1, axis=0)
polish_2_mean_filled = polish_2.fillna(polish_1.mean(axis=0), axis=0)
y_2 = polish_2_mean_filled['class']
X_2 = polish_2_mean_filled.drop(['class','Attr37'], axis=1)
X_2 = normalize(X_2, axis=0)
polish_3_mean_filled = polish_3.fillna(polish_1.mean(axis=0), axis=0)
y_3 = polish_3_mean_filled['class']
X_3 = polish_3_mean_filled.drop(['class','Attr37'], axis=1)
X_3 = normalize(X_3, axis=0)
polish_4_mean_filled = polish_4.fillna(polish_1.mean(axis=0), axis=0)
y_4 = polish_4_mean_filled['class']
X_4 = polish_4_mean_filled.drop(['class','Attr37'], axis=1)
X_4 = normalize(X_4, axis=0)
```

3.2.2 Multilayer Perceptron Classifier

[114]:
```
Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
```

```
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = MLPClassifier(hidden_layer_sizes=25, verbose=False)
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))



MLP_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
MLP_AUC_STD = [np.array(Roc_Auc_1).std(), np.array(Roc_Auc_2).std(), np.
 ↪array(Roc_Auc_3).std(), np.array(Roc_Auc_4).std()]

MLP_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
MLP_Precision_STD = [np.array(Precision_1).std(), np.array(Precision_2).std(),␣
 ↪np.array(Precision_3).std(), np.array(Precision_4).std()]

MLP_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
MLP_Recall_STD = [np.array(Recall_1).std(), np.array(Recall_2).std(), np.
 ↪array(Recall_3).std(), np.array(Recall_4).std()]
```

3.2.3 K-Neighbours Classifier

[97]:
```
Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = KNeighborsClassifier(n_neighbors=3, p=2, metric='minkowski')
```

```
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))


KNN_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
KNN_AUC_STD = [np.array(Roc_Auc_1).std(), np.array(Roc_Auc_2).std(), np.
 ↪array(Roc_Auc_3).std(), np.array(Roc_Auc_4).std()]

KNN_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
KNN_Precision_STD = [np.array(Precision_1).std(), np.array(Precision_2).std(),␣
 ↪np.array(Precision_3).std(), np.array(Precision_4).std()]

KNN_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
KNN_Recall_STD = [np.array(Recall_1).std(), np.array(Recall_2).std(), np.
 ↪array(Recall_3).std(), np.array(Recall_4).std()]
```

3.2.4 Random Forest Classifier

[115]:
```
Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = RandomForestClassifier(criterion = 'entropy', n_estimators =␣
 ↪10, random_state = 60)
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
```

```
        Roc_auc.append(roc_auc_score(y_test, y_pred))


RF_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
RF_AUC_STD = [np.array(Roc_Auc_1).std(), np.array(Roc_Auc_2).std(), np.
 ↪array(Roc_Auc_3).std(), np.array(Roc_Auc_4).std()]

RF_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
RF_Precision_STD = [np.array(Precision_1).std(), np.array(Precision_2).std(),␣
 ↪np.array(Precision_3).std(), np.array(Precision_4).std()]

RF_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
RF_Recall_STD = [np.array(Recall_1).std(), np.array(Recall_2).std(), np.
 ↪array(Recall_3).std(), np.array(Recall_4).std()]
```

[75]:
```
#scoring = ['precision_macro', 'recall_macro', 'f1_macro', 'roc_auc']
#scores = cross_validate(myforest, X, y, cv=10, scoring='roc_auc')
#print(f'{scores["test_score"].mean():.3f}, {scores["test_score"].std():.3f}')
```

[76]:
```
#from sklearn.metrics import SCORERS
#sorted(SCORERS.keys())
```

3.2.5 Perceptron Classifier

[116]:
```
Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
 ↪random_state = 60)
        model.fit(X_train, y_train)
```

```
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))


SLP_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
SLP_AUC_STD = [np.array(Roc_Auc_1).std(), np.array(Roc_Auc_2).std(), np.
 ↪array(Roc_Auc_3).std(), np.array(Roc_Auc_4).std()]

SLP_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
SLP_Precision_STD = [np.array(Precision_1).std(), np.array(Precision_2).std(),␣
 ↪np.array(Precision_3).std(), np.array(Precision_4).std()]

SLP_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
SLP_Recall_STD = [np.array(Recall_1).std(), np.array(Recall_2).std(), np.
 ↪array(Recall_3).std(), np.array(Recall_4).std()]
```

3.2.6 Summary & Results

[100]:
```
Polish_AUC_DF = pd.concat([pd.Series(MLP_AUC_Mean, name='MLP Mean'),
                    pd.Series(MLP_AUC_STD, name='MLP STD'),
                    pd.Series(SLP_AUC_Mean, name='SLP Mean'),
                    pd.Series(SLP_AUC_STD, name='SLP STD'),
                    pd.Series(RF_AUC_Mean, name='RF Mean'),
                    pd.Series(RF_AUC_STD, name='RF STD'),
                    pd.Series(KNN_AUC_Mean, name='KNN Mean'),
                    pd.Series(KNN_AUC_STD, name='KNN STD'),
                    pd.Series(['Data Set 1', 'Data Set 2', 'Data Set 3', 'Data␣
 ↪Set 4'], name='Data Sets')], axis=1)

Polish_AUC_DF.set_index('Data Sets', inplace=True)
display(Polish_AUC_DF)
```

```
            MLP Mean   MLP STD  SLP Mean   SLP STD   RF Mean    RF STD  \
Data Sets
Data Set 1  0.500000  0.000000  0.508290  0.030694  0.636188  0.033861
Data Set 2  0.500000  0.000000  0.503313  0.009939  0.594130  0.020262
Data Set 3  0.499900  0.000200  0.499950  0.000150  0.575377  0.016510
Data Set 4  0.503272  0.006291  0.537221  0.073880  0.594900  0.034896

            KNN Mean   KNN STD
Data Sets
Data Set 1  0.512018  0.012928
```

```
Data Set 2  0.511572  0.012791
Data Set 3  0.510846  0.010213
Data Set 4  0.594414  0.027377
```

[101]:
```
Polish_Recall_DF = pd.concat([pd.Series(MLP_Recall_Mean, name='MLP Mean'),
                    pd.Series(MLP_Recall_STD, name='MLP STD'),
                    pd.Series(SLP_Recall_Mean, name='SLP Mean'),
                    pd.Series(SLP_Recall_STD, name='SLP STD'),
                    pd.Series(RF_Recall_Mean, name='RF Mean'),
                    pd.Series(RF_Recall_STD, name='RF STD'),
                    pd.Series(KNN_Recall_Mean, name='KNN Mean'),
                    pd.Series(KNN_Recall_STD, name='KNN STD'),
                    pd.Series(['Data Set 1', 'Data Set 2', 'Data Set 3', 'Data␣
 ↪Set 4'], name='Data Sets')], axis=1)

Polish_Recall_DF.set_index('Data Sets', inplace=True)
display(Polish_Recall_DF)
```

```
            MLP Mean   MLP STD  SLP Mean   SLP STD   RF Mean    RF STD  \
Data Sets
Data Set 1   0.00000  0.000000  0.207407  0.360688  0.273413  0.067572
Data Set 2   0.00000  0.000000  0.010000  0.030000  0.190000  0.040620
Data Set 3   0.00000  0.000000  0.000000  0.000000  0.153551  0.032816
Data Set 4   0.00773  0.012791  0.176923  0.350380  0.192496  0.070111

            KNN Mean   KNN STD
Data Sets
Data Set 1  0.029365  0.027300
Data Set 2  0.030000  0.024495
Data Set 3  0.030286  0.020676
Data Set 4  0.200038  0.055400
```

[102]:
```
Polish_Precision_DF = pd.concat([pd.Series(MLP_Precision_Mean, name='MLP Mean'),
                    pd.Series(MLP_Precision_STD, name='MLP STD'),
                    pd.Series(SLP_Precision_Mean, name='SLP Mean'),
                    pd.Series(SLP_Precision_STD, name='SLP STD'),
                    pd.Series(RF_Precision_Mean, name='RF Mean'),
                    pd.Series(RF_Precision_STD, name='RF STD'),
                    pd.Series(KNN_Precision_Mean, name='KNN Mean'),
                    pd.Series(KNN_Precision_STD, name='KNN STD'),
                    pd.Series(['Data Set 1', 'Data Set 2', 'Data Set 3', 'Data␣
 ↪Set 4'], name='Data Sets')], axis=1)

Polish_Precision_DF.set_index('Data Sets', inplace=True)
display(Polish_Precision_DF)
```

```
            MLP Mean   MLP STD  SLP Mean   SLP STD   RF Mean    RF STD  \
Data Sets
Data Set 1  0.000000  0.000000  0.017156  0.029986  0.921944  0.100250
Data Set 2  0.000000  0.000000  0.010811  0.032432  0.825631  0.103399
Data Set 3  0.000000  0.000000  0.000000  0.000000  0.738387  0.136822
Data Set 4  0.183333  0.320156  0.117519  0.296193  0.802780  0.100421

            KNN Mean   KNN STD
Data Sets
Data Set 1  0.174167  0.169413
Data Set 2  0.167222  0.129379
Data Set 3  0.147222  0.084778
Data Set 4  0.493887  0.091544
```

4 Performance After Improvement (Oversampling Methods (SMOTE))

4.1 On Fashion-MNIST

[131]:
```
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import KFold,StratifiedKFold
from sklearn.metrics import f1_score
```

[132]:
```
print(f'70-30: Before OverSampling, counts of label "0": {sum(y_train_70 ==␣
 ↪0)}, counts of label "6": {sum(y_train_70 == 6)}')
print(f'90-10: Before OverSampling, counts of label "0": {sum(y_train_90 ==␣
 ↪0)}, counts of label "6": {sum(y_train_90 == 6)}')
print(f'95- 5: Before OverSampling, counts of label "0": {sum(y_train_95 ==␣
 ↪0)}, counts of label "6": {sum(y_train_95 == 6)}')

sm = SMOTE(random_state = 2)
X_train_res_70, y_train_res_70 = sm.fit_sample(X_train_70, y_train_70.ravel(), )
X_train_res_90, y_train_res_90 = sm.fit_sample(X_train_90, y_train_90.ravel(), )
X_train_res_95, y_train_res_95 = sm.fit_sample(X_train_95, y_train_95.ravel(), )

print(f'70-30: After OverSampling, counts of label "0": {sum(y_train_res_70 ==␣
 ↪0)}, counts of label "6": {sum(y_train_res_70 == 6)}')
print(f'90-10: After OverSampling, counts of label "0": {sum(y_train_res_90 ==␣
 ↪0)}, counts of label "6": {sum(y_train_res_90 == 6)}')
print(f'95- 5: After OverSampling, counts of label "0": {sum(y_train_res_95 ==␣
 ↪0)}, counts of label "6": {sum(y_train_res_95 == 6)}')
```

```
70-30: Before OverSampling, counts of label "0": 6000, counts of label "6": 2571
90-10: Before OverSampling, counts of label "0": 6000, counts of label "6": 667
95- 5: Before OverSampling, counts of label "0": 6000, counts of label "6": 316
```

```
70-30: After OverSampling, counts of label "0": 6000, counts of label "6": 6000
90-10: After OverSampling, counts of label "0": 6000, counts of label "6": 6000
95- 5: After OverSampling, counts of label "0": 6000, counts of label "6": 6000
```

4.1.1 MLP

[139]:
```
mlp_70 = MLPClassifier(hidden_layer_sizes=25, verbose=False).
 ↪fit(X_train_res_70, y_train_res_70)
y_pred_70 = mlp_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

mlp_90 = MLPClassifier(hidden_layer_sizes=25, verbose=False).
 ↪fit(X_train_res_90, y_train_res_90)
y_pred_90 = mlp_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

mlp_95 = MLPClassifier(hidden_layer_sizes=25, verbose=False).
 ↪fit(X_train_res_95, y_train_res_95)
y_pred_95 = mlp_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)
```

[212]:
```
MLP_SMOTE_AUC = [MLP_AUC[0],ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
MLP_SMOTE_Recall = [MLP_Recall[0],Recall_70, Recall_90, Recall_95]
MLP_SMOTE_Precision = [Precision_70, Precision_90, Precision_95]
```

4.1.2 SLP

[140]:
```
SLP_70 = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
 ↪random_state = 60).fit(X_train_res_70, y_train_res_70)
y_pred_70 = SLP_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

SLP_90 = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
 ↪random_state = 60).fit(X_train_res_90, y_train_res_90)
y_pred_90 = SLP_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)
```

```
SLP_95 = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
 ↪random_state = 60).fit(X_train_res_95, y_train_res_95)
y_pred_95 = SLP_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)
```

[213]:
```
SLP_SMOTE_AUC = [SLP_AUC[0],ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
SLP_SMOTE_Recall = [SLP_Recall[0],Recall_70, Recall_90, Recall_95]
SLP_SMOTE_Precision = [Precision_70, Precision_90, Precision_95]
```

4.1.3 Random Forest

[141]:
```
RF_70 = RandomForestClassifier(criterion = 'entropy', n_estimators = 10,␣
 ↪random_state = 60).fit(X_train_res_70, y_train_res_70)
y_pred_70 = RF_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

RF_90 = RandomForestClassifier(criterion = 'entropy', n_estimators = 10,␣
 ↪random_state = 60).fit(X_train_res_90, y_train_res_90)
y_pred_90 = RF_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

RF_95 = RandomForestClassifier(criterion = 'entropy', n_estimators = 10,␣
 ↪random_state = 60).fit(X_train_res_95, y_train_res_95)
y_pred_95 = RF_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)
```

[214]:
```
RF_SMOTE_AUC = [RF_AUC[0], ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
RF_SMOTE_Recall = [RF_Recall[0],Recall_70, Recall_90, Recall_95]
RF_SMOTE_Precision = [Precision_70, Precision_90, Precision_95]
```

4.1.4 Results

[161]:
```
AUC_DF = pd.concat([pd.Series(MLP_AUC[1:], name='MLP'),
                    pd.Series(MLP_SMOTE_AUC, name='MLP SMOTE'),
                    pd.Series(SLP_AUC[1:], name='SLP'),
                    pd.Series(SLP_SMOTE_AUC, name='SLP SMOTE'),
                    pd.Series(RF_AUC[1:], name='RF'),
                    pd.Series(RF_SMOTE_AUC, name='RF SMOTE'),
```

```
#                    pd.Series(CUSBoost_AUC, name='CUSBoost'),
                   pd.Series(['70-30', '90-10', '95-5'], name='Balance')],␣
 ↪axis=1)

AUC_DF.set_index('Balance', inplace=True)
display(AUC_DF)
```

Balance	MLP	MLP SMOTE	SLP	SLP SMOTE	RF	RF SMOTE
70-30	0.824143	0.840260	0.704179	0.775872	0.812340	0.812319
90-10	0.780293	0.792311	0.679680	0.846878	0.759261	0.780293
95-5	0.782953	0.747651	0.509434	0.797425	0.725415	0.767585

[162]:
```
Recall_DF = pd.concat([pd.Series(MLP_Recall[1:], name='MLP'),
                   pd.Series(MLP_SMOTE_Recall, name='MLP SMOTE'),
                   pd.Series(SLP_Recall[1:], name='SLP'),
                   pd.Series(SLP_SMOTE_Recall, name='SLP SMOTE'),
                   pd.Series(RF_Recall[1:], name='RF'),
                   pd.Series(RF_SMOTE_Recall, name='RF SMOTE'),
#                    pd.Series(CUSBoost_Recall, name='CUSBoost'),
                   pd.Series(['70-30', '90-10', '95-5'], name='Balance')],␣
 ↪axis=1)

Recall_DF.set_index('Balance', inplace=True)
display(Recall_DF)
```

Balance	MLP	MLP SMOTE	SLP	SLP SMOTE	RF	RF SMOTE
70-30	0.713287	0.813520	0.974359	0.589744	0.652681	0.694639
90-10	0.585586	0.621622	0.360360	0.756757	0.522523	0.585586
95-5	0.584906	0.528302	0.018868	0.735849	0.452830	0.547170

[163]:
```
Precision_DF = pd.concat([pd.Series(MLP_Precision[1:], name='MLP'),
                   pd.Series(MLP_SMOTE_Precision, name='MLP SMOTE'),
                   pd.Series(SLP_Precision[1:], name='SLP'),
                   pd.Series(SLP_SMOTE_Precision, name='SLP SMOTE'),
                   pd.Series(RF_Precision[1:], name='RF'),
                   pd.Series(RF_SMOTE_Precision, name='RF SMOTE'),
#                    pd.Series(CUSBoost_Precision, name='CUSBoost'),
                   pd.Series(['70-30', '90-10', '95-5'], name='Balance')],␣
 ↪axis=1)

Precision_DF.set_index('Balance', inplace=True)
display(Precision_DF)
```

Balance	MLP	MLP SMOTE	SLP	SLP SMOTE	RF	RF SMOTE

Balance						
70-30	0.824798	0.724066	0.424797	0.869416	0.909091	0.809783
90-10	0.722222	0.650943	0.975610	0.571429	0.935484	0.722222
95-5	0.620000	0.459016	1.000000	0.216667	0.923077	0.707317

4.2 On Polish Bankruptcy

4.2.1 MLP

[117]:
```
Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        sm = SMOTE(random_state=2)
        X_train_oversampled, y_train_oversampled = sm.fit_sample(X_train,␣
 ↪y_train)
        model = MLPClassifier(hidden_layer_sizes=25, verbose=False)
        model.fit(X_train_oversampled, y_train_oversampled )
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))
```

[118]:
```
MLP_SMOTE_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(),␣
 ↪np.array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
MLP_SMOTE_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
MLP_SMOTE_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(),␣
 ↪np.array(Recall_3).mean(), np.array(Recall_4).mean()]
```

4.2.2 Random Forest

[119]:
```
Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()  # See comment on ravel and  y_train
        sm = SMOTE(random_state=2)
        X_train_oversampled, y_train_oversampled = sm.fit_sample(X_train,␣
 ↪y_train)
        model = RandomForestClassifier(criterion = 'entropy', n_estimators = 10)
        model.fit(X_train_oversampled, y_train_oversampled )
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))
```

[120]:
```
RF_SMOTE_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
RF_SMOTE_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
RF_SMOTE_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(),␣
 ↪np.array(Recall_3).mean(), np.array(Recall_4).mean()]
```

4.2.3 SLP

[121]:
```
Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
```

```
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        sm = SMOTE(random_state=2)
        X_train_oversampled, y_train_oversampled = sm.fit_sample(X_train,␣
↪y_train)
        model = Perceptron(penalty = 'elasticnet', max_iter = 30, eta0 = 0.001,␣
↪random_state = 60)
        model.fit(X_train_oversampled, y_train_oversampled )
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))
```

[122]:
```
SLP_SMOTE_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(),␣
↪np.array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
SLP_SMOTE_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
SLP_SMOTE_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(),␣
↪np.array(Recall_3).mean(), np.array(Recall_4).mean()]
```

4.2.4 Results

[195]:
```
Polish_AUC_DF = pd.concat([pd.Series(MLP_AUC_Mean, name='MLP Mean'),
                           pd.Series(MLP_AUC_STD, name='MLP STD'),
                           pd.Series(MLP_SMOTE_AUC_Mean, name='MLP SMOTE␣
↪Mean'),
                           pd.Series(SLP_AUC_Mean, name='SLP Mean'),
                           pd.Series(SLP_AUC_STD, name='SLP STD'),
                           pd.Series(SLP_SMOTE_AUC_Mean, name='SLP SMOTE␣
↪Mean'),
                           pd.Series(RF_AUC_Mean, name='RF Mean'),
                           pd.Series(RF_AUC_STD, name='RF STD'),
                           pd.Series(RF_SMOTE_AUC_Mean, name='RF SMOTE Mean'),
                           pd.Series(['Data Set 1', 'Data Set 2', 'Data Set␣
↪3', 'Data Set 4'],
                           name='Data Sets')], axis=1)

Polish_AUC_DF.set_index('Data Sets', inplace=True)
display(Polish_AUC_DF.T)
```

Data Sets	Data Set 1	Data Set 2	Data Set 3	Data Set 4
MLP Mean	0.500000	0.499949	0.499900	0.504360
MLP STD	0.000000	0.000154	0.000200	0.006594

MLP SMOTE Mean	0.680908	0.651633	0.700095	0.721761
SLP Mean	0.499926	0.505069	0.499950	0.517512
SLP STD	0.000222	0.015207	0.000150	0.049716
SLP SMOTE Mean	0.518787	0.504489	0.497938	0.564466
RF Mean	0.652524	0.597625	0.572054	0.593091
RF STD	0.033776	0.027187	0.037462	0.026708
RF SMOTE Mean	0.732397	0.688115	0.673225	0.702986

[194]:
```
Polish_Recall_DF = pd.concat([pd.Series(MLP_Recall_Mean, name='MLP Mean'),
                   pd.Series(MLP_Recall_STD, name='MLP STD'),
                   pd.Series(MLP_SMOTE_Recall_Mean, name='MLP SMOTE Mean'),
                   pd.Series(SLP_Recall_Mean, name='SLP Mean'),
                   pd.Series(SLP_Recall_STD, name='SLP STD'),
                   pd.Series(SLP_SMOTE_Recall_Mean, name='SLP SMOTE Mean'),
                   pd.Series(RF_Recall_Mean, name='RF Mean'),
                   pd.Series(RF_Recall_STD, name='RF STD'),
                   pd.Series(RF_SMOTE_Recall_Mean, name='RF SMOTE Mean'),
                   pd.Series(['Data Set 1', 'Data Set 2', 'Data Set 3', 'Data␣
 ↪Set 4'], name='Data Sets')], axis=1)

Polish_Recall_DF.set_index('Data Sets', inplace=True)
display(Polish_Recall_DF.T)
```

Data Sets	Data Set 1	Data Set 2	Data Set 3	Data Set 4
MLP Mean	0.000000	0.000000	0.000000	0.009691
MLP STD	0.000000	0.000000	0.000000	0.013070
MLP SMOTE Mean	0.686111	0.675000	0.737224	0.679374
SLP Mean	0.000000	0.017500	0.000000	0.090196
SLP STD	0.000000	0.052500	0.000000	0.264117
SLP SMOTE Mean	0.896296	0.045000	0.993878	0.651131
RF Mean	0.306085	0.197500	0.147306	0.188122
RF STD	0.067282	0.054141	0.075274	0.053234
RF SMOTE Mean	0.476190	0.392500	0.369633	0.426885

[193]:
```
Polish_Precision_DF = pd.concat([pd.Series(MLP_Precision_Mean, name='MLP Mean'),

                   pd.Series(MLP_Precision_STD, name='MLP STD'),
                   pd.Series(MLP_SMOTE_Precision_Mean, name='MLP SMOTE Mean'),
                   pd.Series(SLP_Precision_Mean, name='SLP Mean'),

                   pd.Series(SLP_Precision_STD, name='SLP STD'),
                   pd.Series(SLP_SMOTE_Precision_Mean, name='SLP SMOTE Mean'),
                   pd.Series(RF_Precision_Mean, name='RF Mean'),

                   pd.Series(RF_Precision_STD, name='RF STD'),
                   pd.Series(RF_SMOTE_Precision_Mean, name='RF SMOTE Mean'),
```

```
                    pd.Series(['Data Set 1', 'Data Set 2', 'Data Set 3', 'Data␣
↪Set 4'], name='Data Sets')], axis=1)

Polish_Precision_DF.set_index('Data Sets', inplace=True)
display(Polish_Precision_DF.T)
```

Data Sets	Data Set 1	Data Set 2	Data Set 3	Data Set 4
MLP Mean	0.000000	0.000000	0.000000	0.266667
MLP STD	0.000000	0.000000	0.000000	0.351188
MLP SMOTE Mean	0.078615	0.069204	0.097969	0.138998
SLP Mean	0.000000	0.008861	0.000000	0.108108
SLP STD	0.000000	0.026582	0.000000	0.298278
SLP SMOTE Mean	0.046519	0.040367	0.046946	0.111684
RF Mean	0.924123	0.773562	0.681538	0.855120
RF STD	0.081685	0.100095	0.160512	0.120812
RF SMOTE Mean	0.633422	0.496432	0.441367	0.533220

5 Performance After Improvement (Undersampling & Ensembling Methods)

5.1 On Polish Bankruptcy

5.1.1 RUSBoost

[181]:
```
from imblearn.ensemble import RUSBoostClassifier
from sklearn.linear_model import LogisticRegression
base = LogisticRegression()

Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = RUSBoostClassifier(base_estimator = base, random_state=0)
        model.fit(X_train, y_train)
```

```
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))

RUS_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
RUS_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
RUS_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
```

5.1.2 Easy Ensemble

[173]:
```
from imblearn.ensemble import EasyEnsembleClassifier

Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = EasyEnsembleClassifier(random_state=0)
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))

Easy_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
Easy_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
Easy_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
```

5.1.3 Logistic Based Easy Ensemble

[175]:
```
from imblearn.ensemble import EasyEnsembleClassifier

base = LogisticRegression(max_iter = 1000)

Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = EasyEnsembleClassifier(base_estimator = base, random_state=0)
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))

Easy_LR_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
Easy_LR_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
Easy_LR_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
```

5.1.4 AdaBoost Classifier

[177]:
```
from sklearn.ensemble import AdaBoostClassifier

Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
```

```
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = AdaBoostClassifier(n_estimators = 100)
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))

Ada_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
Ada_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
Ada_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
```

5.1.5 Tree Based Rusboost

[179]:
```
from imblearn.ensemble import RUSBoostClassifier


Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4 = [], [], [], []
Recall_1, Recall_2, Recall_3, Recall_4 = [], [], [], []
Precision_1, Precision_2, Precision_3, Precision_4 = [], [], [], []
Xtt = [X_1, X_2, X_3, X_4]
ytt = [y_1, y_2, y_3, y_4]
Roc_Auc = [Roc_Auc_1, Roc_Auc_2, Roc_Auc_3, Roc_Auc_4]
Precision = [Precision_1, Precision_2, Precision_3, Precision_4]
Recall = [Recall_1, Recall_2, Recall_3, Recall_4]

for X,y,Roc_auc, precision, recall in zip(Xtt, ytt, Roc_Auc, Precision, Recall):
    kf = StratifiedKFold(n_splits=10, shuffle=True)
    for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
        X_train = X[train_index]
        y_train = y[train_index].ravel()
        X_test = X[test_index]
        y_test = y[test_index].ravel()
        model = RUSBoostClassifier(random_state=0)
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
```

```
        precision.append(precision_score(y_test, y_pred))
        recall.append(recall_score(y_test, y_pred))
        Roc_auc.append(roc_auc_score(y_test, y_pred))

RUStree_AUC_Mean = [np.array(Roc_Auc_1).mean(), np.array(Roc_Auc_2).mean(), np.
 ↪array(Roc_Auc_3).mean(), np.array(Roc_Auc_4).mean()]
RUStree_Precision_Mean = [np.array(Precision_1).mean(), np.array(Precision_2).
 ↪mean(), np.array(Precision_3).mean(), np.array(Precision_4).mean()]
RUStree_Recall_Mean = [np.array(Recall_1).mean(), np.array(Recall_2).mean(), np.
 ↪array(Recall_3).mean(), np.array(Recall_4).mean()]
```

5.1.6 Summary & Results of Metrics

[199]:
```
Polish_AUC_DF_Total = pd.concat([pd.Series(MLP_AUC_Mean, name='MLP Mean'),
                                 pd.Series(MLP_AUC_STD, name='MLP STD'),
                                 pd.Series(MLP_SMOTE_AUC_Mean, name='MLP SMOTE␣
 ↪Mean'),
                                 pd.Series(SLP_AUC_Mean, name='SLP Mean'),
                                 pd.Series(SLP_AUC_STD, name='SLP STD'),
                                 pd.Series(SLP_SMOTE_AUC_Mean, name='SLP SMOTE␣
 ↪Mean'),
                                 pd.Series(RF_AUC_Mean, name='RF Mean'),
                                 pd.Series(RF_AUC_STD, name='RF STD'),
                                 pd.Series(RF_SMOTE_AUC_Mean, name='RF SMOTE Mean'),
                                 pd.Series(RUS_AUC_Mean, name='LR Based Rusboost␣
 ↪Mean'),
                                 pd.Series(Easy_AUC_Mean, name='Tree Based Easy␣
 ↪Ensemble Mean'),
                                 pd.Series(Easy_LR_AUC_Mean, name='LR Based Easy␣
 ↪Ensemble Mean'),
                                 pd.Series(Ada_AUC_Mean, name='Adaboost Mean'),
                                 pd.Series(RUStree_AUC_Mean, name='Tree Based␣
 ↪RusBoost Mean'),
                                 pd.Series(['Data Set 1', 'Data Set 2', 'Data Set␣
 ↪3', 'Data Set 4'],
                                 name='Data Sets')], axis=1)

Polish_AUC_DF_Total.set_index('Data Sets', inplace=True)
display(Polish_AUC_DF_Total.T)
```

Data Sets	Data Set 1	Data Set 2	Data Set 3	Data Set 4
MLP Mean	0.500000	0.499949	0.499900	0.504360
MLP STD	0.000000	0.000154	0.000200	0.006594
MLP SMOTE Mean	0.680908	0.651633	0.700095	0.721761
SLP Mean	0.499926	0.505069	0.499950	0.517512
SLP STD	0.000222	0.015207	0.000150	0.049716

SLP SMOTE Mean	0.518787	0.504489	0.497938	0.564466
RF Mean	0.652524	0.597625	0.572054	0.593091
RF STD	0.033776	0.027187	0.037462	0.026708
RF SMOTE Mean	0.732397	0.688115	0.673225	0.702986
LR Based Rusboost Mean	0.577110	0.500000	0.508620	0.679341
Tree Based Easy Ensemble Mean	0.839197	0.781516	0.785921	0.811797
LR Based Easy Ensemble Mean	0.535616	0.530405	0.622505	0.678886
Adaboost Mean	0.695884	0.574153	0.587430	0.639693
Tree Based RusBoost Mean	0.794653	0.721882	0.747628	0.738981

[200]:
```
Additional_Polish_Recall_DF_Total = pd.concat([pd.Series(MLP_Recall_Mean,␣
↪name='MLP Mean'),
                                    pd.Series(MLP_Recall_STD, name='MLP STD'),
                                    pd.Series(MLP_SMOTE_Recall_Mean, name='MLP␣
↪SMOTE Mean'),
                                    pd.Series(SLP_Recall_Mean, name='SLP Mean'),
                                    pd.Series(SLP_Recall_STD, name='SLP STD'),
                                    pd.Series(SLP_SMOTE_Recall_Mean, name='SLP␣
↪SMOTE Mean'),
                                    pd.Series(RF_Recall_Mean, name='RF Mean'),
                                    pd.Series(RF_Recall_STD, name='RF STD'),
                                    pd.Series(RF_SMOTE_Recall_Mean, name='RF␣
↪SMOTE Mean'),
                                    pd.Series(RUS_Recall_Mean, name='LR Based␣
↪Rusboost Mean'),
                                    pd.Series(Easy_Recall_Mean, name='Tree␣
↪Based Easy Ensemble Mean'),
                                    pd.Series(Easy_LR_Recall_Mean, name='LR␣
↪Based Easy Ensemble Mean'),
                                    pd.Series(Ada_Recall_Mean, name='Adaboost␣
↪Mean'),
                                    pd.Series(RUStree_Recall_Mean, name='Tree␣
↪Based RusBoost Mean'),
                                    #pd.Series(KNN_Precision_Mean, name='KNN␣
↪Mean'),
                                    #pd.Series(KNN_Precision_STD, name='KNN␣
↪STD'),
                                    pd.Series(['Data Set 1', 'Data Set 2',␣
↪'Data Set 3', 'Data Set 4'],
                                    name='Data Sets')], axis=1)

Additional_Polish_Recall_DF_Total.set_index('Data Sets', inplace=True)
display(Additional_Polish_Recall_DF_Total.T)
```

Data Sets	Data Set 1	Data Set 2	Data Set 3	Data Set 4
MLP Mean	0.000000	0.000000	0.000000	0.009691

```
MLP STD                              0.000000    0.000000    0.000000    0.013070
MLP SMOTE Mean                       0.686111    0.675000    0.737224    0.679374
SLP Mean                             0.000000    0.017500    0.000000    0.090196
SLP STD                              0.000000    0.052500    0.000000    0.264117
SLP SMOTE Mean                       0.896296    0.045000    0.993878    0.651131
RF Mean                              0.306085    0.197500    0.147306    0.188122
RF STD                               0.067282    0.054141    0.075274    0.053234
RF SMOTE Mean                        0.476190    0.392500    0.369633    0.426885
LR Based Rusboost Mean               0.859259    0.000000    0.040816    0.553243
Tree Based Easy Ensemble Mean        0.845503    0.795000    0.781878    0.800264
LR Based Easy Ensemble Mean          0.180952    0.117500    0.499102    0.512783
Adaboost Mean                        0.398280    0.152500    0.181755    0.287255
Tree Based RusBoost Mean             0.719709    0.635000    0.676612    0.645475
```

[192]:
```
Additional_Polish_Precision_DF = pd.concat([pd.Series(RUS_Precision_Mean,␣
↪name='LR Based Rusboost Mean'),
                                    pd.Series(Easy_Precision_Mean, name='Tree␣
↪Based Easy Ensemble Mean'),
                                    pd.Series(Easy_LR_Precision_Mean, name='LR␣
↪Based Easy Ensemble Mean'),
                                    pd.Series(Ada_Precision_Mean,␣
↪name='Adaboost Mean'),
                                    pd.Series(RUStree_Precision_Mean,␣
↪name='Tree Based RusBoost Mean'),
                                    pd.Series(RF_Precision_Mean, name='RF␣
↪Mean'),
                                    pd.Series(RF_Precision_STD, name='RF STD'),
                                    pd.Series(RF_SMOTE_Precision_Mean, name='RF␣
↪SMOTE Mean'),
                                    pd.Series(RF_SMOTE_Precision_Mean, name='RF␣
↪SMOTE Mean'),
                                    pd.Series(['Data Set 1', 'Data Set 2',␣
↪'Data Set 3', 'Data Set 4'],
                                    name='Data Sets')], axis=1)

Additional_Polish_Precision_DF.set_index('Data Sets', inplace=True)
display(Additional_Polish_Precision_DF)
```

```
            LR Based Rusboost Mean  Tree Based Easy Ensemble Mean  \
Data Sets
Data Set 1                0.050045                       0.169129
Data Set 2                0.000000                       0.123247
Data Set 3                0.007812                       0.156111
Data Set 4                0.137260                       0.201222

            LR Based Easy Ensemble Mean  Adaboost Mean  \
```

```
Data Sets
Data Set 1                    0.061040       0.712352
Data Set 2                    0.084123       0.604087
Data Set 3                    0.088578       0.584354
Data Set 4                    0.156117       0.671085

            Tree Based RusBoost Mean   RF Mean    RF STD  RF SMOTE Mean  \
Data Sets
Data Set 1                  0.182755  0.924123  0.081685       0.633422
Data Set 2                  0.120053  0.773562  0.100095       0.496432
Data Set 3                  0.155765  0.681538  0.160512       0.441367
Data Set 4                  0.180453  0.855120  0.120812       0.533220

            RF SMOTE Mean
Data Sets
Data Set 1       0.633422
Data Set 2       0.496432
Data Set 3       0.441367
Data Set 4       0.533220
```

5.2 On Fashion-MNIST

5.2.1 RUSBoost Classifier

[201]:
```
from imblearn.ensemble import RUSBoostClassifier
from sklearn.linear_model import LogisticRegression
base = LogisticRegression()

RUS_50 = RUSBoostClassifier(base_estimator = base, random_state=0).
 ↪fit(X_train_50, y_train_50)
y_pred_50 = RUS_50.predict(X_test_50)
Precision_50, Recall_50, ROC_AUC_50 = precision_score(y_test_50, y_pred_50,␣
 ↪pos_label=6),recall_score(y_test_50, y_pred_50, pos_label=6),␣
 ↪roc_auc_score(y_test_50, y_pred_50)

RUS_70 = RUSBoostClassifier(base_estimator = base, random_state=0).
 ↪fit(X_train_70, y_train_70)
y_pred_70 = RUS_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

RUS_90 = RUSBoostClassifier(base_estimator = base, random_state=0).
 ↪fit(X_train_90, y_train_90)
y_pred_90 = RUS_90.predict(X_test_90)
```

```
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

RUS_95 = RUSBoostClassifier(base_estimator = base, random_state=0).
 ↪fit(X_train_95, y_train_95)
y_pred_95 = RUS_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)


RUS_FM_AUC = [ROC_AUC_50,ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
RUS_FM_Recall = [Recall_50, Recall_70, Recall_90, Recall_95]
RUS_FM_Precision = [Precision_50, Precision_70, Precision_90, Precision_95]
```

5.2.2 Easy Ensemble

[202]:
```
from imblearn.ensemble import EasyEnsembleClassifier

Easy_50 = EasyEnsembleClassifier(random_state=0).fit(X_train_50, y_train_50)
y_pred_50 = Easy_50.predict(X_test_50)
Precision_50, Recall_50, ROC_AUC_50 = precision_score(y_test_50, y_pred_50,␣
 ↪pos_label=6),recall_score(y_test_50, y_pred_50, pos_label=6),␣
 ↪roc_auc_score(y_test_50, y_pred_50)

Easy_70 = EasyEnsembleClassifier(random_state=0).fit(X_train_70, y_train_70)
y_pred_70 = Easy_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

Easy_90 = EasyEnsembleClassifier(random_state=0).fit(X_train_90, y_train_90)
y_pred_90 = Easy_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

Easy_95 = EasyEnsembleClassifier(random_state=0).fit(X_train_95, y_train_95)
y_pred_95 = Easy_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)


Easy_FM_AUC = [ROC_AUC_50,ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
```

```
Easy_FM_Recall = [Recall_50, Recall_70, Recall_90, Recall_95]
Easy_FM_Precision = [Precision_50, Precision_70, Precision_90, Precision_95]
```

5.2.3 Logistic Based Easy Ensemble

[206]:
```
from imblearn.ensemble import EasyEnsembleClassifier

base = LogisticRegression(max_iter = 1000)

Easy_50 = EasyEnsembleClassifier(base_estimator=base,random_state=0).
 ↪fit(X_train_50, y_train_50)
y_pred_50 = Easy_50.predict(X_test_50)
Precision_50, Recall_50, ROC_AUC_50 = precision_score(y_test_50, y_pred_50,␣
 ↪pos_label=6),recall_score(y_test_50, y_pred_50, pos_label=6),␣
 ↪roc_auc_score(y_test_50, y_pred_50)

Easy_70 = EasyEnsembleClassifier(base_estimator=base,random_state=0).
 ↪fit(X_train_70, y_train_70)
y_pred_70 = Easy_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

Easy_90 = EasyEnsembleClassifier(base_estimator=base,random_state=0).
 ↪fit(X_train_90, y_train_90)
y_pred_90 = Easy_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

Easy_95 = EasyEnsembleClassifier(base_estimator=base,random_state=0).
 ↪fit(X_train_95, y_train_95)
y_pred_95 = Easy_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)


EasyLR_FM_AUC = [ROC_AUC_50,ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
EasyLR_FM_Recall = [Recall_50, Recall_70, Recall_90, Recall_95]
EasyLR_FM_Precision = [Precision_50, Precision_70, Precision_90, Precision_95]
```

5.2.4 Adaboost Classifier

[207]:
```
Easy_50 = AdaBoostClassifier(n_estimators = 100).fit(X_train_50, y_train_50)
y_pred_50 = Easy_50.predict(X_test_50)
Precision_50, Recall_50, ROC_AUC_50 = precision_score(y_test_50, y_pred_50,␣
 ↪pos_label=6),recall_score(y_test_50, y_pred_50, pos_label=6),␣
 ↪roc_auc_score(y_test_50, y_pred_50)

Easy_70 = AdaBoostClassifier(n_estimators = 100).fit(X_train_70, y_train_70)
y_pred_70 = Easy_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

Easy_90 = AdaBoostClassifier(n_estimators = 100).fit(X_train_90, y_train_90)
y_pred_90 = Easy_90.predict(X_test_90)
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

Easy_95 = AdaBoostClassifier(n_estimators = 100).fit(X_train_95, y_train_95)
y_pred_95 = Easy_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)


Ada_FM_AUC = [ROC_AUC_50,ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
Ada_FM_Recall = [Recall_50, Recall_70, Recall_90, Recall_95]
Ada_FM_Precision = [Precision_50, Precision_70, Precision_90, Precision_95]
```

5.2.5 Tree Based Rusboost

[208]:
```
Easy_50 = RUSBoostClassifier(random_state=0).fit(X_train_50, y_train_50)
y_pred_50 = Easy_50.predict(X_test_50)
Precision_50, Recall_50, ROC_AUC_50 = precision_score(y_test_50, y_pred_50,␣
 ↪pos_label=6),recall_score(y_test_50, y_pred_50, pos_label=6),␣
 ↪roc_auc_score(y_test_50, y_pred_50)

Easy_70 = RUSBoostClassifier(random_state=0).fit(X_train_70, y_train_70)
y_pred_70 = Easy_70.predict(X_test_70)
Precision_70, Recall_70, ROC_AUC_70 = precision_score(y_test_70, y_pred_70,␣
 ↪pos_label=6),recall_score(y_test_70, y_pred_70, pos_label=6),␣
 ↪roc_auc_score(y_test_70, y_pred_70)

Easy_90 = RUSBoostClassifier(random_state=0).fit(X_train_90, y_train_90)
y_pred_90 = Easy_90.predict(X_test_90)
```

```
Precision_90, Recall_90, ROC_AUC_90 = precision_score(y_test_90, y_pred_90,␣
 ↪pos_label=6),recall_score(y_test_90, y_pred_90, pos_label=6),␣
 ↪roc_auc_score(y_test_90, y_pred_90)

Easy_95 = RUSBoostClassifier(random_state=0).fit(X_train_95, y_train_95)
y_pred_95 = Easy_95.predict(X_test_95)
Precision_95, Recall_95, ROC_AUC_95 = precision_score(y_test_95, y_pred_95,␣
 ↪pos_label=6),recall_score(y_test_95, y_pred_95, pos_label=6),␣
 ↪roc_auc_score(y_test_95, y_pred_95)


RusTree_FM_AUC = [ROC_AUC_50,ROC_AUC_70, ROC_AUC_90, ROC_AUC_95]
RusTree_FM_Recall = [Recall_50, Recall_70, Recall_90, Recall_95]
RusTree_FM_Precision = [Precision_50, Precision_70, Precision_90, Precision_95]
```

5.2.6 Summary & Results of Metrics

[217]:
```
FM_AUC_DF_Total = pd.concat([pd.Series(MLP_AUC, name='MLP'),
                             pd.Series(MLP_SMOTE_AUC, name='MLP SMOTE'),
                             pd.Series(SLP_AUC, name='SLP'),
                             pd.Series(SLP_SMOTE_AUC, name='SLP SMOTE'),
                             pd.Series(RF_AUC, name='RF'),
                             pd.Series(RF_SMOTE_AUC, name='RF SMOTE'),
                             pd.Series(RUS_FM_AUC, name='LR Based Rusboost'),
                             pd.Series(Easy_FM_AUC, name='Tree Based Easy␣
 ↪Ensemble'),
                             pd.Series(EasyLR_FM_AUC, name='LR Based Easy␣
 ↪Ensemble'),
                             pd.Series(Ada_FM_AUC, name='Adaboost Mean'),
                             pd.Series(RusTree_FM_AUC, name='Tree Based RusBoost␣
 ↪Mean'),
                             pd.Series(['50-50','70-30', '90-10', '95-5'],
                             name='Balance')], axis=1)

FM_AUC_DF_Total.set_index('Balance', inplace=True)
display(FM_AUC_DF_Total.T)
```

Balance	50-50	70-30	90-10	95-5
MLP	0.859	0.824143	0.780293	0.782953
MLP SMOTE	0.859	0.832763	0.828887	0.745255
SLP	0.824	0.704179	0.679680	0.509434
SLP SMOTE	0.824	0.832763	0.828887	0.745255
RF	0.836	0.812340	0.759261	0.725415
RF SMOTE	0.836	0.832763	0.828887	0.745255
LR Based Rusboost	0.842	0.834767	0.838396	0.815358
Tree Based Easy Ensemble	0.827	0.832598	0.841905	0.816226
LR Based Easy Ensemble	0.833	0.834263	0.837910	0.797792

```
Adaboost Mean              0.831  0.813978  0.790802  0.776519
Tree Based RusBoost Mean   0.827  0.832763  0.828887  0.745255
```

[218]:
```
FM_Recall_DF_Total = pd.concat([pd.Series(MLP_Recall, name='MLP'),
                                pd.Series(MLP_SMOTE_Recall, name='MLP SMOTE'),
                                pd.Series(SLP_Recall, name='SLP'),
                                pd.Series(SLP_SMOTE_Recall, name='SLP SMOTE'),
                                pd.Series(RF_Recall, name='RF'),
                                pd.Series(RF_SMOTE_Recall, name='RF SMOTE'),
                                pd.Series(RUS_FM_Recall, name='LR Based Rusboost'),
                                pd.Series(Easy_FM_Recall, name='Tree Based Easy␣
 ↪Ensemble'),
                                pd.Series(EasyLR_FM_Recall, name='LR Based Easy␣
 ↪Ensemble'),
                                pd.Series(Ada_FM_Recall, name='Adaboost Mean'),
                                pd.Series(RusTree_FM_Recall, name='Tree Based␣
 ↪RusBoost Mean'),
                                pd.Series(['50-50','70-30', '90-10', '95-5'],
                                name='Balance')], axis=1)

FM_Recall_DF_Total.set_index('Balance', inplace=True)
display(FM_Recall_DF_Total.T)
```

```
Balance                    50-50     70-30     90-10      95-5
MLP                        0.823  0.713287  0.585586  0.584906
MLP SMOTE                  0.823  0.806527  0.774775  0.641509
SLP                        0.741  0.974359  0.360360  0.018868
SLP SMOTE                  0.741  0.806527  0.774775  0.641509
RF                         0.782  0.652681  0.522523  0.452830
RF SMOTE                   0.782  0.806527  0.774775  0.641509
LR Based Rusboost          0.831  0.799534  0.792793  0.754717
Tree Based Easy Ensemble   0.815  0.804196  0.810811  0.792453
LR Based Easy Ensemble     0.817  0.806527  0.819820  0.773585
Adaboost Mean              0.825  0.710956  0.603604  0.566038
Tree Based RusBoost Mean   0.815  0.806527  0.774775  0.641509
```

6 APPENDIX

[ ]:
```
def data_balancer(X, y, class_1, class_2, ratio_1, ratio_2):
    import pandas as pd
    ratio_2 = 1-ratio_1

    augmented = pd.concat([pd.DataFrame(X),pd.DataFrame(y,␣
 ↪columns=['label'])],axis=1)
```

```
    augmented_filtered = augmented[(augmented['label']==class_1) |␣
↪(augmented['label']==class_2)]
    augmented_filtered.reset_index()
    num_class_1 = augmented_filtered[augmented_filtered['label'] == class_1].
↪shape[0]
    num_class_2 = augmented_filtered[augmented_filtered['label'] == class_2].
↪shape[0]
    #if num_class_2>num_class_1:
    if (ratio_2/ratio_1) > (num_class_2/num_class_1):
        expected_num_class_1 = round(num_class_2/(ratio_2/ratio_1),0)
        expected_num_class_2 = num_class_2
    else:
        expected_num_class_1 = num_class_1
        expected_num_class_2 = round(num_class_1/(ratio_1/ratio_2),0)

    drop_num_class_1 = int(num_class_1 - expected_num_class_1)
    drop_num_class_2 = int(num_class_2 - expected_num_class_2)

    augmented_filtered.
↪drop(augmented_filtered[augmented_filtered['label']==class_1].
↪sample(n=drop_num_class_1).index,
    axis=0, inplace=True)
    augmented_filtered.
↪drop(augmented_filtered[augmented_filtered['label']==class_2].
↪sample(n=drop_num_class_2).index,
    axis=0, inplace=True)

    num_class_1_after_balance = augmented_filtered[augmented_filtered['label']␣
↪== class_1].shape[0]
    num_class_2_after_balance = augmented_filtered[augmented_filtered['label']␣
↪== class_2].shape[0]

    y_filtered = augmented_filtered['label'].to_numpy()
    augmented_filtered.drop(labels='label', axis=1, inplace=True)
    X_filtered = augmented_filtered.to_numpy()
    Balance = (num_class_1/(num_class_1+num_class_2), num_class_2/
↪(num_class_1+num_class_2))
    Balance_After_Operation = (round(num_class_1_after_balance/
↪(num_class_1_after_balance+num_class_2_after_balance),2),
    round(num_class_2_after_balance/
↪(num_class_1_after_balance+num_class_2_after_balance),2))
    return X_filtered, y_filtered, Balance, Balance_After_Operation
```