Comparing Supervised Learning Classification Algorithms

Shania Ie sie@ucsd.edu
University of California, San Diego, La Jolla, CA 92092 USA

Abstract

This paper focuses on comparing and evaluating supervised learning algorithms across metrics and binary classification datasets. It closely replicates the methods as described in Caruana and Niculescu-Mizil's 2006 paper (CNM06). The selected algorithms are Logistic Regression, Random Forests, and Decision Trees; with accuracy, the precision, area under receiving operating characteristic curve, and F1 score as the performance measurements. All of the algorithms will undergo 5-fold cross-validation to select the best hyperparameters and create a model based on each metric. The result is used to assess the performance of different learning methods across varying problems.

Keywords: Binary Classification, Supervised Learning Algorithm, Logistic Regression, Random Forest, Decision Trees, 5-fold Cross Validation, Accuracy, Precision, ROC-AUC, F1 score

1. Introduction

With the emergence of new learning algorithms such as SVM, bagging, and random forest, Caruana and Niculescu-Mizil referenced STATLOG (King et al., 1995) as a fundamental part of their study; an empirical evaluation between current and older learning algorithms. To evaluate the performances, CNM06 selected eight different metrics grouping them into the threshold to see if the performance surpasses the baseline for correct predictions, ordering/rank, and probability metrics to see the percentage of positive classes. Threshold metrics include accuracy, F-score, and lift, while ordering/rank includes ROC curve, average precision, and precision/recall break-even point. Lastly, squared error and cross-entropy are grouped under probability metrics.

Besides, CNM06 performed calibrations on the classifiers such as Platt Scaling and Isotonic regression which were not included in this experiment. CNM06 performed the selected

algorithms and trained them on 11 different datasets using the optimal hyperparameters. They evaluated the performance according to the metric score.

The results of their analysis concluded that Boosted Decision Trees, Random Forest, and Bagged Decision Trees are the three algorithms that perform consistently well throughout the different datasets. In contrast, Naïve Bayes, Logistic Regression, and Decision Trees are among the three worst-performing with an uncalibrated Naïve Bayes ranking the worst. Even though the three best algorithms perform very well on average, it excels at different categories and can perform poorly even when other algorithms perform better.

This study is a replica of CNM06 on a smaller scale, using only three algorithms (Logistic Regression, Random Forest, and Decision TREE) on five different datasets; three of which were listed in CNM06 (ADULT, LETTER P1, and LETTER P2) and two original data (BANK, and EMPLOYEE). Instead of evaluating across eight performance metrics, only four will be used to evaluate including accuracy, precision, ROC-AUC, and F1-score. Calibration and optimum selection will not be included as part of the analysis. The goal is to evaluate the performance of different algorithms with limited quantification and see if it supports CNM06.

2. Methodology

2.1 Learning Algorithms

Three different learning algorithms were chosen to explore the comparison between hyperparameters as mentioned in CNM06. Also, supplementary parameters were provided and slightly tweaked to analyze the extent of each learning algorithm's performance.

Logistic Regression (LOGREG): Unregularized and regularized models were trained. Regularized models include both Lasso (L1) and Ridge (L2) regressions, with parameters ranging from 10-8 to 10-4 by factors of 10. Lasso regression solver was set to SAGA, whilst Ridge regression solvers were set to both SAG and SAGA. No penalty was set to an unregularized model; SAG and SAGA were used as the chosen solvers. Max iteration was set to 5000.

Random Forest (RF): The number of trees in the forest was set to 1024 estimators. Size of maximum feature set at each split range from 1,2,4,6,8,12,16, or 20. Both GINI and ENTROPY were used as a criterion to measure the quality of the split.

Decision Trees (DT): Setting and comparing different splitting criteria to GINI for impurity or ENTROPY for information gain. Both criteria are used to measure the quality of the split. By default, the splitter is set to "best" which would choose the optimal split at each node. The

minimum samples leaf ranges from 1,2,4,6,8,10,12,14,16,18, or 20, which controls the tree depth.

All numerical data are scaled using Standard Scaler, thus attributes are transformed to 0 mean and 1 standard deviation. Any categorical data are encoded into integer values using One Hot Encoding. In total, four different models are trained in each trial on each problem.

2.2 Performance Metrics

The models were evaluated based on these four-performance metrics, which are accuracy (ACC), precision (PREC), area under receiving operating characteristic curve (ROC AUC), and F1 score (F1). Accuracy calculates the ratio between the number of correct predictions over the total number of predictions. Although, it is not ideal to use accuracy as a metric to define the performance of the algorithm on an imbalanced dataset. It has the tendency to produce misleading results unless the anomaly (imbalance class distribution) is eliminated/fixed.

With this in mind, adding PREC, ROC-AUC, and F1 would give a better idea of the performance. Precision calculates the ratio between the number of correct positive predictions compared to the number of actual positive predictions. It is a great metric for binary classification tasks and gives an idea of how well the model is predicting positive labels. F1 score calculates the weighted average of precision and recall values (Yildrim, 2020). It is a more fitting metric for dealing with an imbalanced dataset as it considers both false positives and negatives. ROC-AUC shows the performance of each model at varying threshold values and represents the probability degree of separability. Higher ROC-AUC values signify that model can distinguish true positive rates against false-positive rates.

2.3 Comparing Across Performance Metrics

All performance metrics are normalized to the range [0,1]; to enable averaging across metrics and datasets. As mentioned in CNM06, the ROC-AUC metric tends to be independent of the data, meaning that conditions are not influenced by unusual events. On the other hand, the accuracy metric is very dependent and sensitive towards imbalanced class label distribution, thus the score does not reflect its true performance. As such, scores follow the Bayes optimal rate with 0 as the baseline and 1 as the optimal value/best performance. However, it is important

to consider that the Bayes optimal rate does not give a good estimate on real problems, therefore the best performance score would only signify as a rough approximation.

2.4 Data Sets

Five binary classification datasets were selected, this includes ADULT, BANK, and LETTER which can be found in the UCI Repository (Blake & Merz, 1998). ADULT is a census income dataset, setting those whose earning exceeds $50k/year as the positive label and the remaining as a zero label. BANK is a marketing-related dataset gathered from the Portuguese banking institutions. The predicted label is to determine whether the client enlists in the subscription with 'yes' as the positive label and 'no' as the zero labels. LETTER was separated into two different problems. LETTER P1 had "O" as the positive label, whilst LETTER P2 had "A-M" as the positive label. Both problems treated the remaining letters as a negative label. Noticeably, LETTER P1 had a highly imbalanced class label with only 4% of its total instances being positive (Inferred from Table 1). The EMPLOYEE dataset is a human resource problem that sets the label as the status of the worker. The positive label is assigned when the worker left, while the rest is set to a zero label. ADULT, BANK, and EMPLOYEE contain categorical attributes, which were transformed into Boolean values using One Hot Encoding.

Table 1. Description of Problems

PROBLEM	#ATTR	TRAIN SIZE	TEST SIZE	%POZ
ADULT	14/104	5000	44722	25%
BANK	21/52	5000	25488	13%
EMPLOYEE	10/18	5000	9999	24%
LETTER P1.	16	5000	15000	4%
LETTER P2.	16	5000	15000	50%

3. Performances by Metric

For each trial, each dataset will undergo a splitting process, where 5000 samples are randomly selected for training and the remaining are used for the testing set. 5-fold cross-validation is then implemented on the training set to select the best hyperparameters, depending on the scoring metrics and algorithms. The new parameters are then used to create a model, which will be fitted on the training samples and used to predict the testing set. The metric score of the

prediction is gathered and compared between algorithms and problems. There are five trials in total, therefore the process above will be repeated accordingly.

As seen Table 2, shows the normalized test performances of each algorithm on each of the selected metrics, averaged over all five datasets. Each table entry is the average of the optimal model's score on the selected test sets. Therefore, it averages out across five trials and five problems. The last column, MEAN, is the average of the performance scores over the four metrics.

The best performing algorithm is marked in boldfaced. Independent t-tests are conducted between the best-performing ones in the metrics versus the rest of the algorithms. Algorithms that are statistically insignificant from the best algorithm ($p > 0.05$) are marked with Asterisks (*). While those that are statistically significant ($p < 0.05$) are neither boldfaced nor Asterisks. Corresponding p-values from the t-test results can be found in Table 11 of Appendix.

Table 2. Test Performances by Metric (averaged over five problems)

MODEL	ACC	PREC	ROC-AUC	F1	MEAN
LOGREG	0.7106	0.4146	0.7733	0.5431	0.6104
RF	**0.9275**	**0.8595**	0.8374*	**0.7707**	**0.8488**
DT	0.8969*	0.7273	**0.8654**	0.7428*	0.8081

The results of the test performances by metric and algorithm combinations are depicted in Table 2. (Refer to Table 6 – 10. in the Appendix to see the compilation of raw test scores per trial and dataset). Overall, Random Forest shows the best performance on imbalanced datasets when taking the average across all four metrics. Decision Tree has a compatible overall performance, but Logistic Regression did significantly worse.

Accuracy: Random Forest had the best performance in terms of accuracy. The difference between Random Forest and Decision Tree is statistically insignificant, with a $p = 0.0661$ (Table 11. Appendix). While Logistic Regression showed a significant statistical difference between the two groups with $p = 2.58435e\text{-}05$ (Table 11). Comparing with the train performances by metric (Table 4 of the appendix), accuracy performances have a similar rank with Random Forest performing best at 0.9994. Similarly, Decision Tree had a high performance at 0.9962 accuracies and Logistic Regression with the lowest accuracy at 0.7139.

Precision: Random Forest ranked first with the best precision. However, t-test comparisons between the two algorithms showed statistical significance as the comparison with Logistic Regression had $p = 3.6037e\text{-}12$ and Decision Tree had $p = 0.0025$ (Table 11). Logistic

Regression did significantly worse as the imbalanced dataset oftentimes resulted in a null precision. Similarly, the precision performances throughout the training set ranked the same (Refer to Table 4).

ROC-AUC: The decision Tree had the best performance for ROC-AUC. The t-test between the best performing model and Random Forest indicates that it is statistically insignificant as the average and distribution of the group are similar, with $p = 0.2622$ (Table 11). In comparison with the training performance (Table 4), Random Forest scored higher than Decision Tree which could signify overfitting. Logistic Regression did not show any significant improvements from the training score.

F1: Random Forest had the highest performance and showed a statistical difference between Logistic Regression. Decision Tree, on the other hand, had a statistically similar distribution with $p = 0.5429$ (Table 11). Logistic Regression did significantly worse and when comparing to the training score in Table 4, only showed a slight improvement.

4. Performances by Problem

Table 3 show the normalized test performances for each algorithm across different datasets, averaged over all four metrics. Each table entry includes the average score of the model on a specific dataset. Hence, it takes the average across five trials and four metrics. The MEAN column calculates the average performance of the model across all the datasets. As mentioned above, best performing models are in boldface and statistically significant values are marked with Asterisks (*).

Table 3. Test Performances by Metric (averaged over four metrics)

MODEL	ADULT	BANK	EMPLOYEE	LTR. P1	LTR. P2	MEAN
LOGREG	0.6973*	0.6564*	0.5061	0.4730	0.7192	0.6104
RF	**0.7548**	**0.6912**	**0.9329**	**0.9168**	**0.9482**	**0.8488**
DT	0.7146*	0.6632*	0.9046	0.8746*	0.8835	0.8081

The results of the test performances by dataset and algorithm combinations are shown in Table 3 (Refer to Table 6 – 10. in the Appendix to see all the raw test scores categorized by dataset and metrics). As shown in the MEAN column, Random Forest had the best overall performance, showing that it is suitable for both imbalanced and balanced datasets. On the other hand, Logistic Regression had the worst performance as it is very sensitive towards imbalanced class distribution.

Adult: Random Forest had the best performance for the ADULT dataset. Logistic Regression had p = 0.1146, and Decision Trees had p = 0.1197 (Table 12). Both show that they are statistically insignificant compared to Random Forest. Comparing the testing and the training performances (Table 5), Random Forest was able to classify the dataset perfectly. However, this is a sign of overfitting as the performance significantly drops in the testing set. This phenomenon applies to both Logistic Regression and Decision Trees.

Bank: Random Forest performed better than other algorithms, with an average of 0.6912 (Table 3). However, the performance is comparable throughout as it does not show any significant difference between the Logistic Regression and Decision Tree average. Like in the ADULT dataset, Logistic Regression and Decision Trees are statistically similar, with p = 0.5732 and the latter being p = 0.5974 (Table 12). In the training performance (Table 5), Random Forest has a perfect score with an average of 1 across all metrics. This is an indication of overfitting as mentioned above.

Employee: Random Forest had the best performance compared to other algorithms, with an average of 0.9329 (Table 3). However, unlike ADULT and BANK, the performance of both Logistic Regression and Decision Tree are statistically significant (See Table 12. for corresponding P-values). Logistic Regression is shown to perform worse, with an average of 0.5061. Decision Tree, on the other hand, did slightly worse than Random Forest, with an average of 0.9046 (Table 3). Comparing the performance with the training score, Random Forest did not achieve a perfect score like the other datasets, however still performs well. While other algorithms had a better outcome than in the testing performance.

Letter P1: Random Forest had the best performance overall, with an average of 0.9168 (Table 3). Logistic Regression did significantly worse than other datasets, with an average of 0.4730. This is due to the highly imbalanced class distribution and overall low performance across different metrics. The Decision Tree is shown to be statistically insignificant as the distribution of the two groups is similar, with not much variability. When comparing the training performances, Random Forest had a perfect score of 1. While the performance of Logistic Regression is more or less the same, showing a really low score. This might indicate that Random Forest does significantly better when handling a highly imbalanced dataset.

Letter P2: Random Forest had the best performance in this dataset, with an average of 0.9482. Both Logistic Regression and Decision Trees are statistically significant, with p = 7.8730e-38 and the latter p = 1.1730e-19. When comparing the performance with the training scores,

Random Forest shows that it classifies the labels perfectly. On the other hand, Decision Tree had a significantly higher score than the testing set. While Logistic Regression had a comparable performance for both sets.

5. Discussion

A common trend can be spotted from the performances of algorithms by problems and metrics. Logistic Regression did worse when dealing with imbalanced data throughout the whole trial, while the performance score of Random Forest and Decision Trees are neck to neck. Although the performance of Logistic Regression is not optimal, it does not reflect the actual performance when dealing with highly imbalanced class distribution; as it can work extremely well when conditions fit. However, the redistribution of imbalanced classes was not dealt with in this experiment. This raises a precision warning stating that there are not enough true positives, resulting in a precision value of 0. This situation could be avoided by implement resampling with oversampling minority classes, under-sampling majority classes, or generating synthetic data. This can be used as an evaluation of hyperparameters using different class weights.

As seen from the overall performance of Random Forest and Decision Trees, they both tend to overfit. Since the default minimum sample leaf for Random Forest is 1 and the minimum sample leaf parameters passed for Decision Tree ranges from 1 to 20, the tree will keep splitting and branching until there is one feature per leaf. With that being said, if the hyperparameter sees a minimum sample leaf of 1 as the best fit, it will go in-depth and will create a complex structure that might be fitted for a very specific type of problem. This explains why the performance between the training set and testing set contrasts.

6. Conclusion

Although there were some slight modifications and addition to the study, it still closely follows the methods mentioned in the Caruana and Niculescu-Mizil (2006) paper. Throughout the experiment, Random Forest and Decision Tree showed excellent performance scores, indicating that these are robust algorithms. Random Forest happens to score exceptionally well in almost every metric, except ROC-AUC. Decision Tree had a performance score that is comparable to that of Random Forest. In summary, Random Forest and Decision Tree are great algorithms to use when facing a highly imbalanced dataset. On the other hand, Logistic Regression had the worst performance across all metrics and datasets. As stated above in the

discussion, the algorithm is sensitive towards imbalanced class distribution, thus giving a really low score in all four metrics. As a result, the performance within each dataset is low given that most are imbalanced.

When comparing the data across metrics, both precision and F1 had a low overall score. This is due to the reason that both metrics rely on the number of true positives. However, since the distribution of class labels is imbalanced, there might not be any positive values in either the training or testing set. Therefore, since precision measures the ratio of correct positive prediction and F1the weighted average of precision and recall, both will have a low value if the number of true positives is low or null. When comparing the data across problems, BANK, EMPLOYEE, and LETTER P1 have some of the lowest values. This goes hand in hand with the number of positive labels in the dataset, and since it is highly imbalanced therefore it explains the low performance. However, even with these disadvantages, Random Forest and Decision Tree were able to perform well, while Logistic Regression fails to produce a great classification. The finding from this study is comparable to those produced by Caruana and Niculescu-Mizil, therefore proving the findings of the paper.

7. Bonus

In addition to the required three (accuracy, ROC-AUC, and F1-score) metrics and four datasets, I added another set of metrics to measure and explore the performance of the algorithms. Since the result of accuracy does not reflect how well it classifies imbalanced datasets, precision would give a better idea of whether the algorithm is classifying properly. Another dataset was added to see the overall performance on another highly imbalanced dataset, strengthening the analysis on which algorithm is the most robust. Additional parameters were added such that in Logistic Regression, I analyzed the performance between Lasso, Ridge, and Unregularized models with different solvers (SAG and SAGA). The tree depth was modified, in which the minimum sample leaf was set to a list of parameters from 1 to 20 by factors of 2. This gave a better idea of the extent of the complexity of the model. Also, all the class weights for the algorithms are set to “balanced” in an attempt to solve the imbalanced distribution.

References

Cantekin, A. (2020). Https://www.kaggle.com/ahmettezcantekin/beginner-datasets. Retrieved March 18, 2021, from https://www.kaggle.com/ahmettezcantekin/beginner-datasets/activity.

Chun, N. Y. (2018, October 11). Re: One-Hot-Encode categorical variables and scale continuous ones simultaneouely [Web log comment]. Retrieved March 18, 2021, from https://stackoverflow.com/questions/43798377/one-hot-encode-categorical-variables-and-scale-continuous-ones-simultaneouely

Dua, D., & Graff, C. (2019). UCI Machine Learning Repository. Retrieved March 18, 2021, from http://archive.ics.uci.edu/ml

R. Caruana and A. Niculescu-Mizil. ”An empirical comparison of supervised learning algorithms.”In Proceedings of the 23rd international conference on Machine learning,161-168. 2006.

Sklearn.linear_model.LogisticRegression. (n.d.). Retrieved March 18, 2021, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

Sklearn.ensemble.randomforestclassifier. (n.d.). Retrieved March 18, 2021, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Sklearn.tree.decisiontreeclassifier. (n.d.). Retrieved March 18, 2021, from https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

Appendix

Table 4. Train Performances by Metric (averaged over five problems)					
MODEL	ACC	PREC	ROC-AUC	F1	MEAN
LOGREG	0.7139	0.4202	0.7811	0.5515	0.6167
RF	**0.9994**	**0.9974**	**0.9995**	**0.9987**	0.9987
DT	0.9962	0.9970	0.9088	0.8668	0.9422

Table 5. Train Performances by Metric (averaged over four metrics)						
MODEL	ADULT	BANK	EMPLOYEE	LTR. P1	LTR. P2	MEAN
LOGREG	0.7079	0.6626	0.5159	0.4739	0.7230	0.6167
RF	**1**	**1**	**0.9937**	**1**	**1**	0.9987
DT	0.8918	0.8807	0.9730	0.9910	0.9744	0.9422

Table 6. Raw Testing Performances by Trial (ADULT)						
MODEL	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5	MEAN
LOGREG/ACC	0.8074	0.8034	0.8069	0.8039	0.8023	0.8049
LOGREG/PREC	0.3951	0.4072	0.4769	0.6556	0.4993	0.4868
LOGREG/ROC	0.8163	0.8159	0.8172	0.8181	0.8162	0.8168
LOGREG/F1	0.6824	0.6790	0.6823	0.6828	0.6775	0.6808
RF/ACC	0.8492	0.8491	0.8470	0.8487	0.8484	0.8485
RF/PREC	0.7337	0.7279	0.7317	0.7385	0.7275	0.7319
RF/ROC	0.7704	0.7732	0.7663	0.7716	0.7721	0.7707
RF/F1	0.6681	0.6711	0.6612	0.6691	0.6709	0.6681
DT/ACC	0.8051	0.8058	0.8058	0.8037	0.8037	0.8048
DT/PREC	0.6033	0.5922	0.6005	0.6074	0.6022	0.6011
DT/ROC	0.7959	0.7982	0.7919	0.8026	0.7966	0.7971
DT/F1	0.6514	0.6633	0.6552	0.6531	0.6535	0.6553

Table 7. Raw Testing Performances by Trial (BANK)						
MODEL	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5	MEAN
LOGREG/ACC	0.8661	0.8773	0.8535	0.8564	0.8679	0.8642
LOGREG/PREC	0.1329	0.5818	0.4236	0.1397	0.5661	0.3688
LOGREG/ROC	0.8243	0.8262	0.8196	0.8306	0.8233	0.8248
LOGREG/F1	0.5727	0.5647	0.5665	0.5715	0.5631	0.5677
RF/ACC	0.8925	0.8952	0.8936	0.8897	0.8932	0.8928
RF/PREC	0.7280	0.7180	0.7034	0.7274	0.7073	0.7168
RF/ROC	0.6627	0.6570	0.6802	0.6677	0.6651	0.6665
RF/F1	0.4859	0.5069	0.4937	0.4750	0.4811	0.4885
DT/ACC	0.8651	0.8614	0.8632	0.8635	0.8646	0.8636
DT/PREC	0.4630	0.4521	0.4516	0.4713	0.4736	0.4623
DT/ROC	0.8285	0.8193	0.8143	0.8218	0.7882	0.8144
DT/F1	0.5094	0.5181	0.5181	0.5160	0.5008	0.5125

Table 8. Raw Testing Performances by Trial (LETTER P1)						
MODEL	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5	MEAN
LOGREG/ACC	0.8021	0.0381	0.9623	0.7875	0.7881	0.6756
LOGREG/PREC	0.1483	0.1371	0.1324	0.1371	0.1385	0.1387
LOGREG/ROC	0.8390	0.8432	0.8435	0.8345	0.8318	0.8384
LOGREG/F1	0.2538	0.2346	0.2314	0.2373	0.2396	0.2394
RF/ACC	0.9885	0.9887	0.9901	0.9898	0.9870	0.9888
RF/PREC	0.9970	0.9591	0.9833	0.9914	0.9915	0.9844
RF/ROC	0.8368	0.8790	0.8696	0.8657	0.8420	0.8586
RF/F1	0.8299	0.8431	0.8577	0.8401	0.8068	0.8355
DT/ACC	0.9851	0.9837	0.9847	0.9843	0.9839	0.9843
DT/PREC	0.8346	0.7695	0.8151	0.8029	0.8242	0.8093
DT/ROC	0.9242	0.9305	0.9361	0.9279	0.9181	0.9274
DT/F1	0.7846	0.7880	0.7654	0.7649	0.7854	0.7777

Table 9. Raw Testing Performances by Trial (EMPLOYEE)						
MODEL	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5	MEAN
LOGREG/ACC	0.7661	0.7639	0.2374	0.4172	0.2369	0.4843
LOGREG/PREC	0.3658	0.3690	0.3796	0.3912	0.3721	0.3755
LOGREG/ROC	0.6489	0.6692	0.6593	0.6863	0.6481	0.6624
LOGREG/F1	0.4958	0.4990	0.5063	0.5179	0.4922	0.5022
RF/ACC	0.9600	0.9574	0.9604	0.9617	0.9596	0.9598
RF/PREC	0.9079	0.9139	0.9186	0.9123	0.9204	0.9146
RF/ROC	0.9442	0.9372	0.9428	0.9463	0.9436	0.9428
RF/F1	0.9140	0.9097	0.9157	0.9187	0.9135	0.9143
DT/ACC	0.9396	0.9452	0.9441	0.9443	0.9331	0.9413
DT/PREC	0.8594	0.8780	0.8714	0.8669	0.8685	0.8688
DT/ROC	0.9297	0.9284	0.9341	0.9322	0.9249	0.9299
DT/F1	0.8736	0.8870	0.8818	0.8825	0.8673	0.8784

Table 10. Raw Testing Performances by Trial (LETTER P2)						
MODEL	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5	MEAN
LOGREG/ACC	0.7262	0.7204	0.7279	0.7214	0.7240	0.7240
LOGREG/PREC	0.6433	0.7176	0.7184	0.7202	0.7151	0.7029
LOGREG/ROC	0.7265	0.7211	0.7278	0.7213	0.7252	0.7244
LOGREG/F1	0.7301	0.7210	0.7314	0.7178	0.7271	0.7255
RF/ACC	0.9489	0.9507	0.9483	0.9481	0.9431	0.9478
RF/PREC	0.9459	0.9500	0.9534	0.9515	0.9470	0.9496
RF/ROC	0.9499	0.9507	0.9479	0.9477	0.9442	0.9481
RF/F1	0.9479	0.9501	0.9491	0.9458	0.9428	0.9471
DT/ACC	0.8873	0.8907	0.8967	0.8907	0.8867	0.8904
DT/PREC	0.8881	0.8912	0.8984	0.9077	0.8896	0.8950
DT/ROC	0.8528	0.8750	0.8619	0.8484	0.8538	0.8584
DT/F1	0.8843	0.8932	0.8944	0.8924	0.8865	0.8902

Table 11. P-Values for Table 2				
MODEL	ACC	PREC	ROC-AUC	F1
LOGREG	2.58435e-05	3.6037e-12	6.0841e-06	3.0792e-05
RT	1	1	0.2622	1
DT	0.0661	0.0025	1	0.5429

Table 12. P-Values for Table 3					
MODEL	ADULT	BANK	EMPLOYEE	LTR. P1	LTR. P2
LOGREG	0.1146	0.5732	3.4303e-14	1.6717e-06	7.8730e-38
RT	1	1	1	1	1
DT	0.1197	0.5974	0.0021	0.1068	1.1730e-19

In [3]:
```
import pandas as pd
import numpy as np

from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

import warnings
import sklearn.exceptions
warnings.filterwarnings("ignore", category=sklearn.exceptions.UndefinedMetricWar
```

In [4]:
```
# Loading and Cleaning Data
colnames = ['age', 'workclass', 'fnlwgt', 'education', 'education-num',
            'marital-status', 'occupation', 'relationship', 'race', 'sex',
            'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
df = pd.read_csv('adult.csv', names = colnames)
df = df.replace([' ?', ' <=50K.', ' >50K.', ' <=50K', ' >50K'], [np.NaN, 0, 1, 0
df = df.dropna()
df = df.replace([])

X = df.drop(['income'], axis=1)
y = df['income']
```

PERCENT POSITIVE

Finding percentage of positive label in the dataset

In [5]:
```
percent_pos = ((df['income']== 1).sum() / len(df['income'])) * 100
percent_pos
```

Out[5]:
```
24.78439697492371
```

In [6]:
```
# Transform X values into One Hot Encoding for categorical variables and Standar
# https://stackoverflow.com/questions/43798377/one-hot-encode-categorical-variab

# Get categorical columns
cat = list(X.select_dtypes(['object']).columns)
# Get numerical columns
cont = list(X.select_dtypes(['int64']).columns)

# Scale numerical values
cont_transform = Pipeline(steps=[('scaler', StandardScaler())])
# Encode categorical values
cat_transform = Pipeline(steps=[('categories', OneHotEncoder(sparse=False, handl
```

```
# Transform the dataset into the scaled and encoded version
preprocessor = ColumnTransformer(transformers=[('cont', cont_transform, cont),
                                               ('cat', cat_transform, cat)])
X = pd.DataFrame(preprocessor.fit_transform(X))
X
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9	...	9
0	0.034201	-1.062295	1.128753	0.142888	-0.21878	-0.078120	0.0	0.0	0.0	0.0	...	0.
1	0.866417	-1.007438	1.128753	-0.146733	-0.21878	-2.326738	0.0	0.0	0.0	0.0	...	0.
2	-0.041455	0.245284	-0.438122	-0.146733	-0.21878	-0.078120	0.0	0.0	1.0	0.0	...	0.
3	1.093385	0.425853	-1.221559	-0.146733	-0.21878	-0.078120	0.0	0.0	1.0	0.0	...	0.
4	-0.798015	1.407393	1.128753	-0.146733	-0.21878	-0.078120	0.0	0.0	1.0	0.0	...	0.
...	...	...	...	...	...	...	...	...	...	...	...	.
45217	-0.419735	0.525154	1.128753	-0.146733	-0.21878	-0.078120	0.0	0.0	1.0	0.0	...	0.
45218	0.034201	0.243135	1.128753	-0.146733	-0.21878	-0.411249	0.0	0.0	1.0	0.0	...	0.
45219	-0.041455	1.753613	1.128753	-0.146733	-0.21878	0.754701	0.0	0.0	1.0	0.0	...	0.
45220	0.412481	-1.001947	1.128753	0.579985	-0.21878	-0.078120	0.0	0.0	1.0	0.0	...	0.
45221	-0.268423	-0.071818	1.128753	-0.146733	-0.21878	1.587523	0.0	0.0	0.0	1.0	...	0.

45222 rows × 104 columns

LOGISTIC REGRESSION

In [56]:

```
def logisticRegression(X_train, X_test, y_train, y_test):
    train_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG
    test_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG:

    pipe = Pipeline(steps=[('classifier', LogisticRegression())])

    # Setting Parameters to L1 and L2 regularized + unregularized model
    parameters = [{'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                  'classifier__solver': ['saga'],
                  'classifier__penalty': ['l1'],
                  'classifier__C': np.logspace(-8,4,13)},
                 {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                  'classifier__solver': ['sag', 'saga'],
                  'classifier__penalty': ['none']},
                 {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                  'classifier__solver': ['sag', 'saga'],
                  'classifier__penalty': ['l2'],
                  'classifier__C': np.logspace(-8,4,13)}]

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(pipe, parameters, cv=StratifiedKFold(n_splits=5),
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
```

```
solution_log = pd.DataFrame(results)

# ACCURACY MODEL
solution_log['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
# creating new model with optimal hyperparameters
if 'classifier__C' in best_accuracy:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        C = best_accuracy['classifier__C'],
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
else:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
# Training on the new model
accuracy_model.fit(X_train, y_train)
y_acc_train = accuracy_model.predict(X_train)
acc_train_score = accuracy_score(y_train, y_acc_train)

# Testing and scoring the model
y_acc_test = accuracy_model.predict(X_test)
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION MODEL
solution_log['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis

# creating new model with optimal hyperparameters
if 'classifier__C' in best_precision:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         C = best_precision['classifier__C']
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
else:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC MODEL
solution_log['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'

# creating new model with optimal hyperparameters
if 'classifier__C' in best_roc_auc:
```

```
        roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                       C = best_roc_auc['classifier__C'],
                                       solver = best_roc_auc['classifier__solver
                                       max_iter = 5000,
                                       n_jobs = -1,
                                       class_weight='balanced')
    else:
        roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                       solver = best_roc_auc['classifier__solver
                                       max_iter = 5000,
                                       n_jobs = -1,
                                       class_weight='balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1 MODEL
    solution_log['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]


    # creating new model with optimal hyperparameters
    if 'classifier__C' in best_f1:
        f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                      C = best_f1['classifier__C'],
                                      solver = best_f1['classifier__solver'],
                                      max_iter = 5000,
                                      n_jobs = -1,
                                      class_weight='balanced')
    else:
        f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                      solver = best_f1['classifier__solver'],
                                      max_iter = 5000,
                                      n_jobs = -1,
                                      class_weight='balanced')

    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_log = train_metrics_log.append({'LG: Accuracy': acc_train_scor
                                                  'LG: AUC': roc_train_score, 'LG: F1':

    test_metrics_log = test_metrics_log.append({'LG: Accuracy': acc_test_score,
                                                  'LG: AUC': roc_test_score, 'LG: F1': f

    return train_metrics_log, test_metrics_log, solution_log
```

RANDOM FOREST CLASSIFIER

In [8]:
```
def randomForest(X_train, X_test, y_train, y_test):
    train_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:
    test_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:

    randomForest = RandomForestClassifier()

    # Setting parameters according to CNM06
    param_grid = {
        'n_estimators': [1024],
        'criterion': ['gini', 'entropy'],
        'max_features': [1,2,4,6,8,12,16],
        'n_jobs': [-1],
        'class_weight': ['balanced']}

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(estimator=randomForest, param_grid=param_grid, cv=Stratif
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
    solution_rf = pd.DataFrame(results)

    # ACCURACY MODEL
    solution_rf['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
    best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
    # creating new model with optimal hyperparameters
    accuracy_model = RandomForestClassifier(n_estimators = best_accuracy['n_esti
                                            criterion = best_accuracy['criterion
                                            max_features = best_accuracy['max_fe
                                            n_jobs = -1,
                                            class_weight='balanced')
    # Training on the new model
    accuracy_model.fit(X_train, y_train)
    y_acc_train = accuracy_model.predict(X_train)
    acc_train_score = accuracy_score(y_train, y_acc_train)

    # Testing and scoring the model
    y_acc_test = accuracy_model.predict(X_test)
    acc_test_score = accuracy_score(y_test, y_acc_test)

    # PRECISION MODEL
    solution_rf['Precision'] = hyperparams.cv_results_['mean_test_precision']
    best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
    # creating new model with optimal hyperparameters
    precision_model = RandomForestClassifier(n_estimators = best_precision['n_es
                                             criterion = best_precision['criteri
                                             max_features = best_precision['max_
                                             n_jobs = -1,
                                             class_weight='balanced')
    # Training on the new model
    precision_model.fit(X_train, y_train)
    y_prec_train = precision_model.predict(X_train)
    prec_train_score = precision_score(y_train, y_prec_train)

    # Testing and scoring the model
    y_prec_test = precision_model.predict(X_test)
    prec_test_score = precision_score(y_test, y_prec_test)
```

```
    # ROC AUC MODEL
    solution_rf['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
    best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
    # creating new model with optimal hyperparameters
    roc_model = RandomForestClassifier(n_estimators = best_roc_auc['n_estimators
                                       criterion = best_roc_auc['criterion'],
                                       max_features = best_roc_auc['max_features
                                       n_jobs = -1,
                                       class_weight='balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1 MODEL
    solution_rf['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
    # creating new model with optimal hyperparameters
    f1_model = RandomForestClassifier(n_estimators = best_f1['n_estimators'],
                                      criterion = best_f1['criterion'],
                                      max_features = best_f1['max_features'],
                                      n_jobs = -1,
                                      class_weight='balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_rf = train_metrics_rf.append({'RF: Accuracy': acc_train_score,
                                                'RF: AUC': roc_train_score, 'RF:

    test_metrics_rf = test_metrics_rf.append({'RF: Accuracy': acc_test_score, 'R
                                             'RF: AUC': roc_test_score, 'RF: F1

    return train_metrics_rf, test_metrics_rf, solution_rf
```

DECISION TREE CLASSIFIER

In [7]:

```
def decisionTrees(X_train, X_test, y_train, y_test):
    train_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:
    test_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:

    pipe = Pipeline(steps=[('classifier', DecisionTreeClassifier())])

    # Setting parameters according to CNM06 + passing a list of min_samples_leaf
    parameters = [{'classifier': [DecisionTreeClassifier(class_weight='balanced'
                  'classifier__criterion': ['gini', 'entropy'],
                  'classifier__splitter': ['best'],
                  'classifier__min_samples_leaf': [1,2,4,6,8,10,12,14,16,18]}]

    # Perform 5-fold cross-validation using grid search
```

```
clf = GridSearchCV(estimator=pipe, param_grid=parameters, cv=StratifiedKFold
                   scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

# Fitting training set on the cross validation
hyperparams = clf.fit(X_train, y_train)
# Storing best parameters for each metric model
results = hyperparams.cv_results_['params']
solution_dt = pd.DataFrame(results)

# ACCURACY
solution_dt['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
# creating new model with optimal hyperparameters
accuracy_model = DecisionTreeClassifier(criterion = best_accuracy['classifie
                                        splitter = best_accuracy['classifier
                                        min_samples_leaf = best_accuracy['cl
                                        class_weight = 'balanced')

# Training on the new model
accuracy_model.fit(X_train, y_train)
y_acc_train = accuracy_model.predict(X_train)
acc_train_score = accuracy_score(y_train, y_acc_train)

# Testing and scoring the model
y_acc_test = accuracy_model.predict(X_test)
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION
solution_dt['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
# creating new model with optimal hyperparameters
precision_model = DecisionTreeClassifier(criterion = best_precision['classif
                                         splitter = best_precision['classifi
                                         min_samples_leaf = best_precision['
                                         class_weight = 'balanced')
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC
solution_dt['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
# creating new model with optimal hyperparameters
roc_model = DecisionTreeClassifier(criterion = best_roc_auc['classifier__cri
                                   splitter = best_roc_auc['classifier__spli
                                   min_samples_leaf = best_roc_auc['classifi
                                   class_weight = 'balanced')
# Training on the new model
roc_model.fit(X_train, y_train)
y_roc_train = roc_model.predict(X_train)
roc_train_score = roc_auc_score(y_train, y_roc_train)

# Testing and scoring the model
y_roc_test = roc_model.predict(X_test)
roc_test_score = roc_auc_score(y_test, y_roc_test)
```

```
    # F1
    solution_dt['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
    # creating new model with optimal hyperparameters
    f1_model = DecisionTreeClassifier(criterion = best_f1['classifier__criterion
                                      splitter = best_f1['classifier__splitter']
                                      min_samples_leaf = best_f1['classifier__mi
                                      class_weight = 'balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_dt = train_metrics_dt.append({'DT: Accuracy': acc_train_score,
                                                'DT: AUC': roc_train_score, 'DT:

    test_metrics_dt = test_metrics_dt.append({'DT: Accuracy': acc_test_score, 'D
                                              'DT: AUC': roc_test_score, 'DT: F1

    return train_metrics_dt, test_metrics_dt, solution_dt
```

RUNNING DATASET ON ALL THREE ALGORITHMS

In [59]:

```
trials = 5
train_metrics = pd.DataFrame()
test_metrics = pd.DataFrame()
solution_metrics = pd.DataFrame()

# Running the trial five times
for i in range(trials):
    # Splitting data into train size = 5000
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=5000, s

    train_log, test_log, solution_log = logisticRegression(X_train, X_test, y_tr
    train_rf, test_rf, solution_rf = randomForest(X_train, X_test, y_train, y_te
    train_dt, test_dt, solution_dt = decisionTrees(X_train, X_test, y_train, y_t

    train_metrics = train_metrics.append(pd.concat([train_log, train_rf, train_d
    test_metrics = test_metrics.append(pd.concat([test_log, test_rf, test_dt], a
    solution_metrics = solution_metrics.append(pd.concat([solution_log, solution

# storing data into CSV file
train_metrics.to_csv('adult_train.csv')
test_metrics.to_csv('adult_test.csv')
solution_metrics.to_csv('adult_solution.csv')
```

```
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 32.5min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed: 19.4min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:   17.4s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 45.4min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:  7.8min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:    9.5s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 17.3min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:  8.5min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:    4.1s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 15.9min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:  4.7min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:    3.6s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 10.9min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:  4.8min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:    4.0s finished
```

In [ ]:

In [9]:
```
import pandas as pd
import numpy as np

from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

import warnings
import sklearn.exceptions
warnings.filterwarnings("ignore", category=sklearn.exceptions.UndefinedMetricWar
```

In [17]:
```
df = pd.read_csv('bank-additional-full.csv', delimiter=';')
df = df.replace(['unknown'], np.NaN)
df = df.dropna()
df['y'] = df['y'].map({'yes': 1, 'no': 0})

X = df.drop(['y'], axis=1)
y = df['y']
df
```

Out[17]:

	age	job	marital	education	default	housing	loan	contact	month	day
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	
2	37	services	married	high.school	no	yes	no	telephone	may	
3	40	admin.	married	basic.6y	no	no	no	telephone	may	
4	56	services	married	high.school	no	no	yes	telephone	may	
6	59	admin.	married	professional.course	no	no	no	telephone	may	
...	...	...	...	...	...	...	...	...	...	
41183	73	retired	married	professional.course	no	yes	no	cellular	nov	
41184	46	blue-collar	married	professional.course	no	no	no	cellular	nov	
41185	56	retired	married	university.degree	no	yes	no	cellular	nov	
41186	44	technician	married	professional.course	no	no	no	cellular	nov	
41187	74	retired	married	professional.course	no	yes	no	cellular	nov	

30488 rows × 21 columns

PERCENT POSITIVE

Finding percentage of positive label in the dataset

In [11]:
```
percent_pos = ((df['y'] == 1).sum() / len(df['y'])) * 100
percent_pos
```

Out[11]:
```
12.65743899239045
```

In [12]:
```
# Transform X values into One Hot Encoding for categorical variables and Standar
# https://stackoverflow.com/questions/43798377/one-hot-encode-categorical-variab

# Get categorical columns
cat = list(X.select_dtypes(['object']).columns)
# Get numerical columns
cont = list(X.select_dtypes(['int64']).columns)

# Scale numerical values
cont_transform = Pipeline(steps=[('scaler', StandardScaler())])
# Encode categorical values
cat_transform = Pipeline(steps=[('categories', OneHotEncoder(sparse=False, handl
# Transform the dataset into the scaled and encoded version
preprocessor = ColumnTransformer(transformers=[('cont', cont_transform, cont),
                                               ('cat', cat_transform, cat)])
X = pd.DataFrame(preprocessor.fit_transform(X))
X
```

Out[12]:

	0	1	2	3	4	5	6	7	8	9	...	42	43
0	1.642253	0.005792	-0.559335	0.211887	-0.371616	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0
1	-0.196452	-0.127944	-0.559335	0.211887	-0.371616	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
2	0.093870	-0.414520	-0.559335	0.211887	-0.371616	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0
3	1.642253	0.181559	-0.559335	0.211887	-0.371616	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
4	1.932575	-0.460373	-0.559335	0.211887	-0.371616	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
30483	3.287410	0.284727	-0.559335	0.211887	-0.371616	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
30484	0.674513	0.471957	-0.559335	0.211887	-0.371616	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0
30485	1.642253	-0.269321	-0.191702	0.211887	-0.371616	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
30486	0.480965	0.697398	-0.559335	0.211887	-0.371616	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
30487	3.384184	-0.078270	0.175930	0.211887	1.541237	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0

30488 rows × 52 columns

LOGISTIC REGRESSION

In [13]:
```
def logisticRegression(X_train, X_test, y_train, y_test):
    train_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG
    test_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG:

    pipe = Pipeline(steps=[('classifier', LogisticRegression())])

    # Setting Parameters to L1 and L2 regularized + unregularized model
```

```
parameters = [{'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
               'classifier__solver': ['saga'],
               'classifier__penalty': ['l1'],
               'classifier__C': np.logspace(-8,4,13)},
              {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
               'classifier__solver': ['sag', 'saga'],
               'classifier__penalty': ['none']},
              {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
               'classifier__solver': ['sag', 'saga'],
               'classifier__penalty': ['l2'],
               'classifier__C': np.logspace(-8,4,13)}]

# Perform 5-fold cross-validation using grid search
clf = GridSearchCV(pipe, parameters, cv=StratifiedKFold(n_splits=5),
                   scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

# Fitting training set on the cross validation
hyperparams = clf.fit(X_train, y_train)
# Storing best parameters for each metric model
results = hyperparams.cv_results_['params']
solution_log = pd.DataFrame(results)

# ACCURACY MODEL
solution_log['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
# creating new model with optimal hyperparameters
if 'classifier__C' in best_accuracy:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        C = best_accuracy['classifier__C'],
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
else:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
# Training on the new model
accuracy_model.fit(X_train, y_train)
y_acc_train = accuracy_model.predict(X_train)
acc_train_score = accuracy_score(y_train, y_acc_train)

# Testing and scoring the model
y_acc_test = accuracy_model.predict(X_test)
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION MODEL
solution_log['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis

# creating new model with optimal hyperparameters
if 'classifier__C' in best_precision:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         C = best_precision['classifier__C']
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
else:
```

```
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC MODEL
solution_log['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'

# creating new model with optimal hyperparameters
if 'classifier__C' in best_roc_auc:
    roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                   C = best_roc_auc['classifier__C'],
                                   solver = best_roc_auc['classifier__solver
                                   max_iter = 5000,
                                   n_jobs = -1,
                                   class_weight='balanced')
else:
    roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                   solver = best_roc_auc['classifier__solver
                                   max_iter = 5000,
                                   n_jobs = -1,
                                   class_weight='balanced')
# Training on the new model
roc_model.fit(X_train, y_train)
y_roc_train = roc_model.predict(X_train)
roc_train_score = roc_auc_score(y_train, y_roc_train)

# Testing and scoring the model
y_roc_test = roc_model.predict(X_test)
roc_test_score = roc_auc_score(y_test, y_roc_test)

# F1 MODEL
solution_log['F1'] = hyperparams.cv_results_['mean_test_f1']
best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]


# creating new model with optimal hyperparameters
if 'classifier__C' in best_f1:
    f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                  C = best_f1['classifier__C'],
                                  solver = best_f1['classifier__solver'],
                                  max_iter = 5000,
                                  n_jobs = -1,
                                  class_weight='balanced')
else:
    f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                  solver = best_f1['classifier__solver'],
                                  max_iter = 5000,
                                  n_jobs = -1,
                                  class_weight='balanced')
```

```
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_log = train_metrics_log.append({'LG: Accuracy': acc_train_scor
                                                  'LG: AUC': roc_train_score, 'LG: F1':

    test_metrics_log = test_metrics_log.append({'LG: Accuracy': acc_test_score,
                                                  'LG: AUC': roc_test_score, 'LG: F1': f

    return train_metrics_log, test_metrics_log, solution_log
```

RANDOM FOREST CLASSIFIER

In [14]:

```
def randomForest(X_train, X_test, y_train, y_test):
    train_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:
    test_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:

    randomForest = RandomForestClassifier()

    # Setting parameters according to CNM06
    param_grid = {
        'n_estimators': [1024],
        'criterion': ['gini', 'entropy'],
        'max_features': [1,2,4,6,8,12,16],
        'n_jobs': [-1],
        'class_weight': ['balanced']}

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(estimator=randomForest, param_grid=param_grid, cv=Stratif
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
    solution_rf = pd.DataFrame(results)

    # ACCURACY MODEL
    solution_rf['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
    best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
    # creating new model with optimal hyperparameters
    accuracy_model = RandomForestClassifier(n_estimators = best_accuracy['n_esti
                                            criterion = best_accuracy['criterion
                                            max_features = best_accuracy['max_fe
                                            n_jobs = -1,
                                            class_weight='balanced')
    # Training on the new model
    accuracy_model.fit(X_train, y_train)
    y_acc_train = accuracy_model.predict(X_train)
    acc_train_score = accuracy_score(y_train, y_acc_train)

    # Testing and scoring the model
    y_acc_test = accuracy_model.predict(X_test)
```

```
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION MODEL
solution_rf['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
# creating new model with optimal hyperparameters
precision_model = RandomForestClassifier(n_estimators = best_precision['n_es
                                         criterion = best_precision['criteri
                                         max_features = best_precision['max_
                                         n_jobs = -1,
                                         class_weight='balanced')
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC MODEL
solution_rf['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
# creating new model with optimal hyperparameters
roc_model = RandomForestClassifier(n_estimators = best_roc_auc['n_estimators
                                   criterion = best_roc_auc['criterion'],
                                   max_features = best_roc_auc['max_features
                                   n_jobs = -1,
                                   class_weight='balanced')
# Training on the new model
roc_model.fit(X_train, y_train)
y_roc_train = roc_model.predict(X_train)
roc_train_score = roc_auc_score(y_train, y_roc_train)

# Testing and scoring the model
y_roc_test = roc_model.predict(X_test)
roc_test_score = roc_auc_score(y_test, y_roc_test)

# F1 MODEL
solution_rf['F1'] = hyperparams.cv_results_['mean_test_f1']
best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
# creating new model with optimal hyperparameters
f1_model = RandomForestClassifier(n_estimators = best_f1['n_estimators'],
                                  criterion = best_f1['criterion'],
                                  max_features = best_f1['max_features'],
                                  n_jobs = -1,
                                  class_weight='balanced')
# Training on the new model
f1_model.fit(X_train, y_train)
y_f1_train = f1_model.predict(X_train)
f1_train_score = f1_score(y_train, y_f1_train)

# Testing and scoring the model
y_f1_test = f1_model.predict(X_test)
f1_test_score = f1_score(y_test, y_f1_test)

train_metrics_rf = train_metrics_rf.append({'RF: Accuracy': acc_train_score,
                                            'RF: AUC': roc_train_score, 'RF:

test_metrics_rf = test_metrics_rf.append({'RF: Accuracy': acc_test_score, 'R
                                          'RF: AUC': roc_test_score, 'RF: F1
```

```
    return train_metrics_rf, test_metrics_rf, solution_rf
```

DECISION TREE CLASSIFIER

In [15]:
```
def decisionTrees(X_train, X_test, y_train, y_test):
    train_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:
    test_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:

    pipe = Pipeline(steps=[('classifier', DecisionTreeClassifier())])

    # Setting parameters according to CNM06 + passing a list of min_samples_leaf
    parameters = [{'classifier': [DecisionTreeClassifier(class_weight='balanced'
                  'classifier__criterion': ['gini', 'entropy'],
                  'classifier__splitter': ['best'],
                  'classifier__min_samples_leaf': [1,2,4,6,8,10,12,14,16,18]}]

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(estimator=pipe, param_grid=parameters, cv=StratifiedKFold
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
    solution_dt = pd.DataFrame(results)

    # ACCURACY
    solution_dt['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
    best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
    # creating new model with optimal hyperparameters
    accuracy_model = DecisionTreeClassifier(criterion = best_accuracy['classifie
                                            splitter = best_accuracy['classifier
                                            min_samples_leaf = best_accuracy['cl
                                            class_weight = 'balanced')

    # Training on the new model
    accuracy_model.fit(X_train, y_train)
    y_acc_train = accuracy_model.predict(X_train)
    acc_train_score = accuracy_score(y_train, y_acc_train)

    # Testing and scoring the model
    y_acc_test = accuracy_model.predict(X_test)
    acc_test_score = accuracy_score(y_test, y_acc_test)

    # PRECISION
    solution_dt['Precision'] = hyperparams.cv_results_['mean_test_precision']
    best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
    # creating new model with optimal hyperparameters
    precision_model = DecisionTreeClassifier(criterion = best_precision['classif
                                             splitter = best_precision['classifi
                                             min_samples_leaf = best_precision['
                                             class_weight = 'balanced')
    # Training on the new model
    precision_model.fit(X_train, y_train)
    y_prec_train = precision_model.predict(X_train)
    prec_train_score = precision_score(y_train, y_prec_train)

    # Testing and scoring the model
```

```
    y_prec_test = precision_model.predict(X_test)
    prec_test_score = precision_score(y_test, y_prec_test)

    # ROC AUC
    solution_dt['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
    best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
    # creating new model with optimal hyperparameters
    roc_model = DecisionTreeClassifier(criterion = best_roc_auc['classifier__cri
                                       splitter = best_roc_auc['classifier__spli
                                       min_samples_leaf = best_roc_auc['classifi
                                       class_weight = 'balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1
    solution_dt['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
    # creating new model with optimal hyperparameters
    f1_model = DecisionTreeClassifier(criterion = best_f1['classifier__criterion
                                      splitter = best_f1['classifier__splitter']
                                      min_samples_leaf = best_f1['classifier__mi
                                      class_weight = 'balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_dt = train_metrics_dt.append({'DT: Accuracy': acc_train_score,
                                                'DT: AUC': roc_train_score, 'DT:

    test_metrics_dt = test_metrics_dt.append({'DT: Accuracy': acc_test_score, 'D
                                             'DT: AUC': roc_test_score, 'DT: F1

    return train_metrics_dt, test_metrics_dt, solution_dt
```

RUNNING DATASET ON ALL THREE ALGORITHMS

In [16]:

```
trials = 5
train_metrics = pd.DataFrame()
test_metrics = pd.DataFrame()
solution_metrics = pd.DataFrame()

# Running the trial five times
for i in range(trials):
    # Splitting data into train size = 5000
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=5000, s
```

```
    train_log, test_log, solution_log = logisticRegression(X_train, X_test, y_tr
    train_rf, test_rf, solution_rf = randomForest(X_train, X_test, y_train, y_te
    train_dt, test_dt, solution_dt = decisionTrees(X_train, X_test, y_train, y_t

    train_metrics = train_metrics.append(pd.concat([train_log, train_rf, train_d
    test_metrics = test_metrics.append(pd.concat([test_log, test_rf, test_dt], a
    solution_metrics = solution_metrics.append(pd.concat([solution_log, solution

# storing data into CSV file
train_metrics.to_csv('bank_train.csv')
test_metrics.to_csv('bank_test.csv')
solution_metrics.to_csv('bank_solution.csv')
```

```
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 49.3min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed: 19.7min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:   18.6s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:  5.5min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:  9.7min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:   11.8s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 12.5min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:  6.8min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:    3.1s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 10.0min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:  7.3min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:    6.4s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed: 12.7min finished
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:  6.0min finished
Fitting 5 folds for each of 22 candidates, totalling 110 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 110 out of 110 | elapsed:    3.2s finished
```

In [ ]:

In [10]:
```
import pandas as pd
import numpy as np

from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

import warnings
import sklearn.exceptions
warnings.filterwarnings("ignore", category=sklearn.exceptions.UndefinedMetricWar
```

In [11]:
```
df = pd.read_csv('employee.csv')
X = df.drop(['left'], axis=1)
y = df['left']
df
```

Out[11]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_co
0	0.38	0.53	2	157	
1	0.80	0.86	5	262	
2	0.11	0.88	7	272	
3	0.72	0.87	5	223	
4	0.37	0.52	2	159	
...	...	...	...	...	
14994	0.40	0.57	2	151	
14995	0.37	0.48	2	160	
14996	0.37	0.53	2	143	
14997	0.11	0.96	6	280	
14998	0.37	0.52	2	158	

14999 rows × 10 columns

PERCENT POSITIVE

Finding percentage of positive label in the dataset

In [12]:
```
percent_pos = ((df['left']== 1).sum() / len(df['left'])) * 100
```

```
percent_pos
```

Out[12]:
```
23.80825388359224
```

In [13]:
```
# Transform X values into One Hot Encoding for categorical variables and Standar
# https://stackoverflow.com/questions/43798377/one-hot-encode-categorical-variab

# Get categorical columns
cat = list(X.select_dtypes(['object']).columns)
# Get numerical columns
cont = list(X.select_dtypes(['int64']).columns)

# Scale numerical values
cont_transform = Pipeline(steps=[('scaler', StandardScaler())])
# Encode categorical values
cat_transform = Pipeline(steps=[('categories', OneHotEncoder(sparse=False, handl
# Transform the dataset into the scaled and encoded version
preprocessor = ColumnTransformer(transformers=[('cont', cont_transform, cont),
                                               ('cat', cat_transform, cat)])
X = pd.DataFrame(preprocessor.fit_transform(X))
X
```

Out[13]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	-1.462863	-0.882040	-0.341235	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
1	0.971113	1.220423	1.713436	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
2	2.593763	1.420657	0.343655	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
3	0.971113	0.439508	1.028546	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
4	-1.462863	-0.841993	-0.341235	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
14994	-1.462863	-1.002181	-0.341235	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14995	-1.462863	-0.821970	-0.341235	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14996	-1.462863	-1.162368	-0.341235	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14997	1.782438	1.580845	0.343655	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14998	-1.462863	-0.862016	-0.341235	-0.411165	-0.147412	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

14999 rows × 18 columns

LOGISTIC REGRESSION

In [1]:
```
def logisticRegression(X_train, X_test, y_train, y_test):
    train_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG
    test_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG:

    pipe = Pipeline(steps=[('classifier', LogisticRegression())])

    # Setting Parameters to L1 and L2 regularized + unregularized model
    parameters = [{'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                  'classifier__solver': ['saga'],
                  'classifier__penalty': ['l1'],
                  'classifier__C': np.logspace(-8,4,13)},
```

```
                {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                 'classifier__solver': ['sag', 'saga'],
                 'classifier__penalty': ['none']},
                {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                 'classifier__solver': ['sag', 'saga'],
                 'classifier__penalty': ['l2'],
                 'classifier__C': np.logspace(-8,4,13)}]

# Perform 5-fold cross-validation using grid search
clf = GridSearchCV(pipe, parameters, cv=StratifiedKFold(n_splits=5),
                   scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

# Fitting training set on the cross validation
hyperparams = clf.fit(X_train, y_train)
# Storing best parameters for each metric model
results = hyperparams.cv_results_['params']
solution_log = pd.DataFrame(results)

# ACCURACY MODEL
solution_log['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
# creating new model with optimal hyperparameters
if 'classifier__C' in best_accuracy:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        C = best_accuracy['classifier__C'],
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
else:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
# Training on the new model
accuracy_model.fit(X_train, y_train)
y_acc_train = accuracy_model.predict(X_train)
acc_train_score = accuracy_score(y_train, y_acc_train)

# Testing and scoring the model
y_acc_test = accuracy_model.predict(X_test)
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION MODEL
solution_log['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis

# creating new model with optimal hyperparameters
if 'classifier__C' in best_precision:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         C = best_precision['classifier__C']
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
else:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
```

```
                                                class_weight='balanced')
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC MODEL
solution_log['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'

# creating new model with optimal hyperparameters
if 'classifier__C' in best_roc_auc:
    roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                   C = best_roc_auc['classifier__C'],
                                   solver = best_roc_auc['classifier__solver
                                   max_iter = 5000,
                                   n_jobs = -1,
                                   class_weight='balanced')
else:
    roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                   solver = best_roc_auc['classifier__solver
                                   max_iter = 5000,
                                   n_jobs = -1,
                                   class_weight='balanced')
# Training on the new model
roc_model.fit(X_train, y_train)
y_roc_train = roc_model.predict(X_train)
roc_train_score = roc_auc_score(y_train, y_roc_train)

# Testing and scoring the model
y_roc_test = roc_model.predict(X_test)
roc_test_score = roc_auc_score(y_test, y_roc_test)

# F1 MODEL
solution_log['F1'] = hyperparams.cv_results_['mean_test_f1']
best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]


# creating new model with optimal hyperparameters
if 'classifier__C' in best_f1:
    f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                  C = best_f1['classifier__C'],
                                  solver = best_f1['classifier__solver'],
                                  max_iter = 5000,
                                  n_jobs = -1,
                                  class_weight='balanced')
else:
    f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                  solver = best_f1['classifier__solver'],
                                  max_iter = 5000,
                                  n_jobs = -1,
                                  class_weight='balanced')

# Training on the new model
f1_model.fit(X_train, y_train)
y_f1_train = f1_model.predict(X_train)
f1_train_score = f1_score(y_train, y_f1_train)
```

```
    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_log = train_metrics_log.append({'LG: Accuracy': acc_train_scor
                                                  'LG: AUC': roc_train_score, 'LG: F1':

    test_metrics_log = test_metrics_log.append({'LG: Accuracy': acc_test_score,
                                                  'LG: AUC': roc_test_score, 'LG: F1': f

    return train_metrics_log, test_metrics_log, solution_log
```

RANDOM FOREST CLASSIFIER

In [15]:

```
def randomForest(X_train, X_test, y_train, y_test):
    train_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:
    test_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:

    randomForest = RandomForestClassifier()

    # Setting parameters according to CNM06
    param_grid = {
        'n_estimators': [1024],
        'criterion': ['gini', 'entropy'],
        'max_features': [1,2,4,6,8,12,16],
        'n_jobs': [-1],
        'class_weight': ['balanced']}

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(estimator=randomForest, param_grid=param_grid, cv=Stratif
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
    solution_rf = pd.DataFrame(results)

    # ACCURACY MODEL
    solution_rf['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
    best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
    # creating new model with optimal hyperparameters
    accuracy_model = RandomForestClassifier(n_estimators = best_accuracy['n_esti
                                            criterion = best_accuracy['criterion
                                            max_features = best_accuracy['max_fe
                                            n_jobs = -1,
                                            class_weight='balanced')
    # Training on the new model
    accuracy_model.fit(X_train, y_train)
    y_acc_train = accuracy_model.predict(X_train)
    acc_train_score = accuracy_score(y_train, y_acc_train)

    # Testing and scoring the model
    y_acc_test = accuracy_model.predict(X_test)
    acc_test_score = accuracy_score(y_test, y_acc_test)

    # PRECISION MODEL
    solution_rf['Precision'] = hyperparams.cv_results_['mean_test_precision']
```

```
    best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
    # creating new model with optimal hyperparameters
    precision_model = RandomForestClassifier(n_estimators = best_precision['n_es
                                             criterion = best_precision['criteri
                                             max_features = best_precision['max_
                                             n_jobs = -1,
                                             class_weight='balanced')
    # Training on the new model
    precision_model.fit(X_train, y_train)
    y_prec_train = precision_model.predict(X_train)
    prec_train_score = precision_score(y_train, y_prec_train)

    # Testing and scoring the model
    y_prec_test = precision_model.predict(X_test)
    prec_test_score = precision_score(y_test, y_prec_test)

    # ROC AUC MODEL
    solution_rf['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
    best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
    # creating new model with optimal hyperparameters
    roc_model = RandomForestClassifier(n_estimators = best_roc_auc['n_estimators
                                       criterion = best_roc_auc['criterion'],
                                       max_features = best_roc_auc['max_features
                                       n_jobs = -1,
                                       class_weight='balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1 MODEL
    solution_rf['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
    # creating new model with optimal hyperparameters
    f1_model = RandomForestClassifier(n_estimators = best_f1['n_estimators'],
                                      criterion = best_f1['criterion'],
                                      max_features = best_f1['max_features'],
                                      n_jobs = -1,
                                      class_weight='balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_rf = train_metrics_rf.append({'RF: Accuracy': acc_train_score,
                                                'RF: AUC': roc_train_score, 'RF:

    test_metrics_rf = test_metrics_rf.append({'RF: Accuracy': acc_test_score, 'R
                                              'RF: AUC': roc_test_score, 'RF: F1

    return train_metrics_rf, test_metrics_rf, solution_rf
```

DECISION TREE CLASSIFIER

In [16]:
```
def decisionTrees(X_train, X_test, y_train, y_test):
    train_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:
    test_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:

    pipe = Pipeline(steps=[('classifier', DecisionTreeClassifier())])

    # Setting parameters according to CNM06 + passing a list of min_samples_leaf
    parameters = [{'classifier': [DecisionTreeClassifier(class_weight='balanced'
                  'classifier__criterion': ['gini', 'entropy'],
                  'classifier__splitter': ['best'],
                  'classifier__min_samples_leaf': [1,2,4,6,8,10,12,14,16,18]}]

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(estimator=pipe, param_grid=parameters, cv=StratifiedKFold
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
    solution_dt = pd.DataFrame(results)

    # ACCURACY
    solution_dt['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
    best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
    # creating new model with optimal hyperparameters
    accuracy_model = DecisionTreeClassifier(criterion = best_accuracy['classifie
                                            splitter = best_accuracy['classifier
                                            min_samples_leaf = best_accuracy['cl
                                            class_weight = 'balanced')

    # Training on the new model
    accuracy_model.fit(X_train, y_train)
    y_acc_train = accuracy_model.predict(X_train)
    acc_train_score = accuracy_score(y_train, y_acc_train)

    # Testing and scoring the model
    y_acc_test = accuracy_model.predict(X_test)
    acc_test_score = accuracy_score(y_test, y_acc_test)

    # PRECISION
    solution_dt['Precision'] = hyperparams.cv_results_['mean_test_precision']
    best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
    # creating new model with optimal hyperparameters
    precision_model = DecisionTreeClassifier(criterion = best_precision['classif
                                             splitter = best_precision['classifi
                                             min_samples_leaf = best_precision['
                                             class_weight = 'balanced')
    # Training on the new model
    precision_model.fit(X_train, y_train)
    y_prec_train = precision_model.predict(X_train)
    prec_train_score = precision_score(y_train, y_prec_train)

    # Testing and scoring the model
    y_prec_test = precision_model.predict(X_test)
    prec_test_score = precision_score(y_test, y_prec_test)

    # ROC AUC
```

```
    solution_dt['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
    best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
    # creating new model with optimal hyperparameters
    roc_model = DecisionTreeClassifier(criterion = best_roc_auc['classifier__cri
                                       splitter = best_roc_auc['classifier__spli
                                       min_samples_leaf = best_roc_auc['classifi
                                       class_weight = 'balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1
    solution_dt['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
    # creating new model with optimal hyperparameters
    f1_model = DecisionTreeClassifier(criterion = best_f1['classifier__criterion
                                      splitter = best_f1['classifier__splitter']
                                      min_samples_leaf = best_f1['classifier__mi
                                      class_weight = 'balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_dt = train_metrics_dt.append({'DT: Accuracy': acc_train_score,
                                                'DT: AUC': roc_train_score, 'DT:

    test_metrics_dt = test_metrics_dt.append({'DT: Accuracy': acc_test_score, 'D
                                              'DT: AUC': roc_test_score, 'DT: F1

    return train_metrics_dt, test_metrics_dt, solution_dt
```

In [17]:

```
trials = 5
train_metrics = pd.DataFrame()
test_metrics = pd.DataFrame()
solution_metrics = pd.DataFrame()

# Running the trial five times
for i in range(trials):
    # Splitting data into train size = 5000
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=5000, s

    train_log, test_log, solution_log = logisticRegression(X_train, X_test, y_tr
    train_rf, test_rf, solution_rf = randomForest(X_train, X_test, y_train, y_te
    train_dt, test_dt, solution_dt = decisionTrees(X_train, X_test, y_train, y_t

    train_metrics = train_metrics.append(pd.concat([train_log, train_rf, train_d
    test_metrics = test_metrics.append(pd.concat([test_log, test_rf, test_dt], a
    solution_metrics = solution_metrics.append(pd.concat([solution_log, solution

# storing data into CSV file
```

```
train_metrics.to_csv('employee_train.csv')
test_metrics.to_csv('employee_test.csv')
solution_metrics.to_csv('employee_solution.csv')
```

```
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   58.9s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 10.1min finished
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    9.6s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   56.8s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 12.2min finished
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:   11.5s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   59.2s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 13.2min finished
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    7.2s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   47.7s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 14.6min finished
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:   11.7s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:  1.4min finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed:  9.5min finished
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    7.5s finished
```

In [ ]:

In [9]:
```
import pandas as pd
import numpy as np

from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

import warnings
import sklearn.exceptions
warnings.filterwarnings("ignore", category=sklearn.exceptions.UndefinedMetricWar
```

In [10]:
```
colnames = ['letter', 'x-box','y-box', 'width', 'height', 'pixels', 'x-bar', 'y-
df = pd.read_csv('letter-recognition.csv', names= colnames)
df['letter'] = df['letter'].apply(lambda x: 1 if x == 'O' else -1)

X = df.drop(['letter'], axis=1)
y = df['letter']
df['letter'].value_counts()
```

Out[10]:
```
-1    19247
 1      753
Name: letter, dtype: int64
```

PERCENT POSITIVE

Finding percentage of positive label in the dataset

In [11]:
```
percent_pos = (df['letter']==1).sum()/len(df['letter'])*100
percent_pos
```

Out[11]:
```
3.765
```

In [12]:
```
# Standardizing for numerical variables
# Get numerical columns
cont = list(X.select_dtypes(['int64']).columns)
# Scale numerical values
cont_transform = Pipeline(steps=[('scaler', StandardScaler())])
# Transform the dataset into the scaled version
preprocessor = ColumnTransformer(transformers=[('cont', cont_transform, cont)])
X = pd.DataFrame(preprocessor.fit_transform(X))
X
```

Out[12]:

	0	1	2	3	4	5	6	7

	0	1	2	3	4	5	6	7
0	-1.057698	0.291877	-1.053277	-0.164704	-1.144013	0.544130	2.365097	-1.714360
1	0.510385	1.502358	-1.053277	0.719730	-0.687476	1.531305	-1.075326	0.137561
2	-0.012309	1.199738	0.435910	1.161947	1.138672	1.531305	-0.645273	-0.973591
3	1.555774	1.199738	0.435910	0.277513	-0.230939	-0.936631	0.644886	-0.232823
4	-1.057698	-1.826464	-1.053277	-1.933571	-1.144013	0.544130	-0.645273	0.507945
...	...	...	...	...	...	...	...	...
19995	-1.057698	-1.523844	-1.053277	-1.049137	-0.687476	0.050543	-0.215220	0.878329
19996	1.555774	0.897117	1.428701	1.161947	0.225598	-1.430218	0.214833	0.507945
19997	1.033079	0.594497	0.435910	0.719730	0.682135	-0.443044	1.504991	-0.603207
19998	-1.057698	-1.221224	-0.556881	-1.491354	-1.144013	0.544130	-0.215220	-0.973591
19999	-0.012309	0.594497	0.435910	0.277513	-0.687476	1.037718	-1.075326	-0.603207

20000 rows × 16 columns

LOGISTIC REGRESSION

In [13]:
```
def logisticRegression(X_train, X_test, y_train, y_test):
    train_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG
    test_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG:

    pipe = Pipeline(steps=[('classifier', LogisticRegression())])

    # Setting Parameters to L1 and L2 regularized + unregularized model
    parameters = [{'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                   'classifier__solver': ['saga'],
                   'classifier__penalty': ['l1'],
                   'classifier__C': np.logspace(-8,4,13)},
                  {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                   'classifier__solver': ['sag', 'saga'],
                   'classifier__penalty': ['none']},
                  {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                   'classifier__solver': ['sag', 'saga'],
                   'classifier__penalty': ['l2'],
                   'classifier__C': np.logspace(-8,4,13)}]

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(pipe, parameters, cv=StratifiedKFold(n_splits=5),
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
    solution_log = pd.DataFrame(results)

    # ACCURACY MODEL
    solution_log['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
    best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
    # creating new model with optimal hyperparameters
```

```
if 'classifier__C' in best_accuracy:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        C = best_accuracy['classifier__C'],
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
else:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
# Training on the new model
accuracy_model.fit(X_train, y_train)
y_acc_train = accuracy_model.predict(X_train)
acc_train_score = accuracy_score(y_train, y_acc_train)

# Testing and scoring the model
y_acc_test = accuracy_model.predict(X_test)
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION MODEL
solution_log['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis

# creating new model with optimal hyperparameters
if 'classifier__C' in best_precision:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         C = best_precision['classifier__C']
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
else:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC MODEL
solution_log['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'

# creating new model with optimal hyperparameters
if 'classifier__C' in best_roc_auc:
    roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                   C = best_roc_auc['classifier__C'],
                                   solver = best_roc_auc['classifier__solver
                                   max_iter = 5000,
                                   n_jobs = -1,
                                   class_weight='balanced')
```

```
    else:
        roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                       solver = best_roc_auc['classifier__solver
                                       max_iter = 5000,
                                       n_jobs = -1,
                                       class_weight='balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1 MODEL
    solution_log['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]


    # creating new model with optimal hyperparameters
    if 'classifier__C' in best_f1:
        f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                      C = best_f1['classifier__C'],
                                      solver = best_f1['classifier__solver'],
                                      max_iter = 5000,
                                      n_jobs = -1,
                                      class_weight='balanced')
    else:
        f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                      solver = best_f1['classifier__solver'],
                                      max_iter = 5000,
                                      n_jobs = -1,
                                      class_weight='balanced')

    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_log = train_metrics_log.append({'LG: Accuracy': acc_train_scor
                                        'LG: AUC': roc_train_score, 'LG: F1':

    test_metrics_log = test_metrics_log.append({'LG: Accuracy': acc_test_score,
                                        'LG: AUC': roc_test_score, 'LG: F1': f

    return train_metrics_log, test_metrics_log, solution_log
```

RANDOM FOREST CLASSIFIER

In [14]:

```
def randomForest(X_train, X_test, y_train, y_test):
    train_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:
    test_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:

    randomForest = RandomForestClassifier()
```

```
# Setting parameters according to CNM06
param_grid = {
    'n_estimators': [1024],
    'criterion': ['gini', 'entropy'],
    'max_features': [1,2,4,6,8,12,16],
    'n_jobs': [-1],
    'class_weight': ['balanced']}

# Perform 5-fold cross-validation using grid search
clf = GridSearchCV(estimator=randomForest, param_grid=param_grid, cv=Stratif
                   scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

# Fitting training set on the cross validation
hyperparams = clf.fit(X_train, y_train)
# Storing best parameters for each metric model
results = hyperparams.cv_results_['params']
solution_rf = pd.DataFrame(results)

# ACCURACY MODEL
solution_rf['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
# creating new model with optimal hyperparameters
accuracy_model = RandomForestClassifier(n_estimators = best_accuracy['n_esti
                                        criterion = best_accuracy['criterion
                                        max_features = best_accuracy['max_fe
                                        n_jobs = -1,
                                        class_weight='balanced')
# Training on the new model
accuracy_model.fit(X_train, y_train)
y_acc_train = accuracy_model.predict(X_train)
acc_train_score = accuracy_score(y_train, y_acc_train)

# Testing and scoring the model
y_acc_test = accuracy_model.predict(X_test)
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION MODEL
solution_rf['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
# creating new model with optimal hyperparameters
precision_model = RandomForestClassifier(n_estimators = best_precision['n_es
                                         criterion = best_precision['criteri
                                         max_features = best_precision['max_
                                         n_jobs = -1,
                                         class_weight='balanced')
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC MODEL
solution_rf['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
# creating new model with optimal hyperparameters
roc_model = RandomForestClassifier(n_estimators = best_roc_auc['n_estimators
                                   criterion = best_roc_auc['criterion'],
```

```
                                        max_features = best_roc_auc['max_features
                                        n_jobs = -1,
                                        class_weight='balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1 MODEL
    solution_rf['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
    # creating new model with optimal hyperparameters
    f1_model = RandomForestClassifier(n_estimators = best_f1['n_estimators'],
                                      criterion = best_f1['criterion'],
                                      max_features = best_f1['max_features'],
                                      n_jobs = -1,
                                      class_weight='balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_rf = train_metrics_rf.append({'RF: Accuracy': acc_train_score,
                                                'RF: AUC': roc_train_score, 'RF:

    test_metrics_rf = test_metrics_rf.append({'RF: Accuracy': acc_test_score, 'R
                                              'RF: AUC': roc_test_score, 'RF: F1

    return train_metrics_rf, test_metrics_rf, solution_rf
```

DECISION TREE CLASSIFIER

In [15]:

```
def decisionTrees(X_train, X_test, y_train, y_test):
    train_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:
    test_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:

    pipe = Pipeline(steps=[('classifier', DecisionTreeClassifier())])

    # Setting parameters according to CNM06 + passing a list of min_samples_leaf
    parameters = [{'classifier': [DecisionTreeClassifier(class_weight='balanced'
                  'classifier__criterion': ['gini', 'entropy'],
                  'classifier__splitter': ['best'],
                  'classifier__min_samples_leaf': [1,2,4,6,8,10,12,14,16,18]}]

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(estimator=pipe, param_grid=parameters, cv=StratifiedKFold
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
```

```
results = hyperparams.cv_results_['params']
solution_dt = pd.DataFrame(results)

# ACCURACY
solution_dt['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
# creating new model with optimal hyperparameters
accuracy_model = DecisionTreeClassifier(criterion = best_accuracy['classifie
                                        splitter = best_accuracy['classifier
                                        min_samples_leaf = best_accuracy['cl
                                        class_weight = 'balanced')

# Training on the new model
accuracy_model.fit(X_train, y_train)
y_acc_train = accuracy_model.predict(X_train)
acc_train_score = accuracy_score(y_train, y_acc_train)

# Testing and scoring the model
y_acc_test = accuracy_model.predict(X_test)
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION
solution_dt['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
# creating new model with optimal hyperparameters
precision_model = DecisionTreeClassifier(criterion = best_precision['classif
                                         splitter = best_precision['classifi
                                         min_samples_leaf = best_precision['
                                         class_weight = 'balanced')
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC
solution_dt['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
# creating new model with optimal hyperparameters
roc_model = DecisionTreeClassifier(criterion = best_roc_auc['classifier__cri
                                   splitter = best_roc_auc['classifier__spli
                                   min_samples_leaf = best_roc_auc['classifi
                                   class_weight = 'balanced')
# Training on the new model
roc_model.fit(X_train, y_train)
y_roc_train = roc_model.predict(X_train)
roc_train_score = roc_auc_score(y_train, y_roc_train)

# Testing and scoring the model
y_roc_test = roc_model.predict(X_test)
roc_test_score = roc_auc_score(y_test, y_roc_test)

# F1
solution_dt['F1'] = hyperparams.cv_results_['mean_test_f1']
best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
# creating new model with optimal hyperparameters
f1_model = DecisionTreeClassifier(criterion = best_f1['classifier__criterion
                                  splitter = best_f1['classifier__splitter']
```

```
                                        min_samples_leaf = best_f1['classifier__mi
                                        class_weight = 'balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_dt = train_metrics_dt.append({'DT: Accuracy': acc_train_score,
                                                'DT: AUC': roc_train_score, 'DT:

    test_metrics_dt = test_metrics_dt.append({'DT: Accuracy': acc_test_score, 'D
                                              'DT: AUC': roc_test_score, 'DT: F1

    return train_metrics_dt, test_metrics_dt, solution_dt
```

RUNNING DATASET ON ALL THREE ALGORITHMS

In [16]:
```
trials = 5
train_metrics = pd.DataFrame()
test_metrics = pd.DataFrame()
solution_metrics = pd.DataFrame()

# Running the trial five times
for i in range(trials):
    # Splitting data into train size = 5000
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=5000, s

    train_log, test_log, solution_log = logisticRegression(X_train, X_test, y_tr
    train_rf, test_rf, solution_rf = randomForest(X_train, X_test, y_train, y_te
    train_dt, test_dt, solution_dt = decisionTrees(X_train, X_test, y_train, y_t

    train_metrics = train_metrics.append(pd.concat([train_log, train_rf, train_d
    test_metrics = test_metrics.append(pd.concat([test_log, test_rf, test_dt], a
    solution_metrics = solution_metrics.append(pd.concat([solution_log, solution

# storing data into CSV file
train_metrics.to_csv('letterp1_train.csv')
test_metrics.to_csv('letterp1_test.csv')
solution_metrics.to_csv('letterp1_solution.csv')
```

```
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   34.5s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed:  4.1min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:    2.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:  1.3min finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed:  7.3min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:    8.2s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:  5.0min finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 10.5min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:   11.2s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   31.4s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 12.5min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:   13.3s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:  9.7min finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 13.8min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:   12.4s finished
```

In [ ]:

In [21]:
```
import pandas as pd
import re
import numpy as np

from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

import warnings
import sklearn.exceptions
warnings.filterwarnings("ignore", category=sklearn.exceptions.UndefinedMetricWar
```

In [22]:
```
colnames = ['letter', 'x-box','y-box', 'width', 'height', 'pixels', 'x-bar', 'y-
df = pd.read_csv('letter-recognition.csv', names= colnames)
p = re.compile('[A-M]')
df['letter'] = df['letter'].apply(lambda x: 1 if p.match(x) else -1)

X = df.drop(['letter'], axis=1)
y = df['letter']
df
```

Out[22]:

	letter	x-box	y-box	width	height	pixels	x-bar	y-bar	x2bar	y2bar	xybar	x2ybr	xy2br	x ege
0	-1	2	8	3	5	1	8	13	0	6	6	10	8	
1	1	5	12	3	7	2	10	5	5	4	13	3	9	
2	1	4	11	6	8	6	10	6	2	6	10	3	7	
3	-1	7	11	6	6	3	5	9	4	6	4	4	10	
4	1	2	1	3	1	1	8	6	6	6	6	5	9	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
19995	1	2	2	3	3	2	7	7	7	6	6	6	4	
19996	1	7	10	8	8	4	4	8	6	9	12	9	13	
19997	-1	6	9	6	7	5	6	11	3	7	11	9	5	
19998	-1	2	3	4	2	1	8	7	2	6	10	6	8	
19999	1	4	9	6	6	2	9	5	3	1	8	1	8	

20000 rows × 17 columns

PERCENT POSITIVE

Finding percentage of positive label in the dataset

In [23]:
```
percent_pos = (df['letter']==1).sum()/len(df['letter'])*100
percent_pos
```

Out[23]: 49.7

In [24]:
```
# Standardizing for numerical variables
# Get numerical columns
cont = list(X.select_dtypes(['int64']).columns)
# Scale numerical values
cont_transform = Pipeline(steps=[('scaler', StandardScaler())])
# Transform the dataset into the scaled version
preprocessor = ColumnTransformer(transformers=[('cont', cont_transform, cont)])
X = pd.DataFrame(preprocessor.fit_transform(X))
X
```

Out[24]:

	0	1	2	3	4	5	6	7
0	-1.057698	0.291877	-1.053277	-0.164704	-1.144013	0.544130	2.365097	-1.714360
1	0.510385	1.502358	-1.053277	0.719730	-0.687476	1.531305	-1.075326	0.137561
2	-0.012309	1.199738	0.435910	1.161947	1.138672	1.531305	-0.645273	-0.973591
3	1.555774	1.199738	0.435910	0.277513	-0.230939	-0.936631	0.644886	-0.232823
4	-1.057698	-1.826464	-1.053277	-1.933571	-1.144013	0.544130	-0.645273	0.507945
...	...	...	...	...	...	...	...	...
19995	-1.057698	-1.523844	-1.053277	-1.049137	-0.687476	0.050543	-0.215220	0.878329
19996	1.555774	0.897117	1.428701	1.161947	0.225598	-1.430218	0.214833	0.507945
19997	1.033079	0.594497	0.435910	0.719730	0.682135	-0.443044	1.504991	-0.603207
19998	-1.057698	-1.221224	-0.556881	-1.491354	-1.144013	0.544130	-0.215220	-0.973591
19999	-0.012309	0.594497	0.435910	0.277513	-0.687476	1.037718	-1.075326	-0.603207

20000 rows × 16 columns

LOGISTIC REGRESSION

In [25]:
```
def logisticRegression(X_train, X_test, y_train, y_test):
    train_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG
    test_metrics_log = pd.DataFrame(columns=['LG: Accuracy','LG: Precision','LG:

    pipe = Pipeline(steps=[('classifier', LogisticRegression())])

    # Setting Parameters to L1 and L2 regularized + unregularized model
    parameters = [{'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                  'classifier__solver': ['saga'],
                  'classifier__penalty': ['l1'],
                  'classifier__C': np.logspace(-8,4,13)},
                 {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
```

```
                'classifier__solver': ['sag', 'saga'],
                'classifier__penalty': ['none']},
               {'classifier': [LogisticRegression(max_iter=5000, n_jobs=-1, c
                'classifier__solver': ['sag', 'saga'],
                'classifier__penalty': ['l2'],
                'classifier__C': np.logspace(-8,4,13)}]

# Perform 5-fold cross-validation using grid search
clf = GridSearchCV(pipe, parameters, cv=StratifiedKFold(n_splits=5),
                   scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

# Fitting training set on the cross validation
hyperparams = clf.fit(X_train, y_train)
# Storing best parameters for each metric model
results = hyperparams.cv_results_['params']
solution_log = pd.DataFrame(results)

# ACCURACY MODEL
solution_log['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
# creating new model with optimal hyperparameters
if 'classifier__C' in best_accuracy:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        C = best_accuracy['classifier__C'],
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
else:
    accuracy_model = LogisticRegression(penalty = best_accuracy['classifier_
                                        solver = best_accuracy['classifier__
                                        max_iter = 5000,
                                        n_jobs = -1,
                                        class_weight='balanced')
# Training on the new model
accuracy_model.fit(X_train, y_train)
y_acc_train = accuracy_model.predict(X_train)
acc_train_score = accuracy_score(y_train, y_acc_train)

# Testing and scoring the model
y_acc_test = accuracy_model.predict(X_test)
acc_test_score = accuracy_score(y_test, y_acc_test)

# PRECISION MODEL
solution_log['Precision'] = hyperparams.cv_results_['mean_test_precision']
best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis

# creating new model with optimal hyperparameters
if 'classifier__C' in best_precision:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         C = best_precision['classifier__C']
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
else:
    precision_model = LogisticRegression(penalty = best_precision['classifie
                                         solver = best_precision['classifier
                                         max_iter = 5000,
                                         n_jobs = -1,
                                         class_weight='balanced')
```

```
# Training on the new model
precision_model.fit(X_train, y_train)
y_prec_train = precision_model.predict(X_train)
prec_train_score = precision_score(y_train, y_prec_train)

# Testing and scoring the model
y_prec_test = precision_model.predict(X_test)
prec_test_score = precision_score(y_test, y_prec_test)

# ROC AUC MODEL
solution_log['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'

# creating new model with optimal hyperparameters
if 'classifier__C' in best_roc_auc:
    roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                   C = best_roc_auc['classifier__C'],
                                   solver = best_roc_auc['classifier__solver
                                   max_iter = 5000,
                                   n_jobs = -1,
                                   class_weight='balanced')
else:
    roc_model = LogisticRegression(penalty = best_roc_auc['classifier__penal
                                   solver = best_roc_auc['classifier__solver
                                   max_iter = 5000,
                                   n_jobs = -1,
                                   class_weight='balanced')
# Training on the new model
roc_model.fit(X_train, y_train)
y_roc_train = roc_model.predict(X_train)
roc_train_score = roc_auc_score(y_train, y_roc_train)

# Testing and scoring the model
y_roc_test = roc_model.predict(X_test)
roc_test_score = roc_auc_score(y_test, y_roc_test)

# F1 MODEL
solution_log['F1'] = hyperparams.cv_results_['mean_test_f1']
best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]


# creating new model with optimal hyperparameters
if 'classifier__C' in best_f1:
    f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                  C = best_f1['classifier__C'],
                                  solver = best_f1['classifier__solver'],
                                  max_iter = 5000,
                                  n_jobs = -1,
                                  class_weight='balanced')
else:
    f1_model = LogisticRegression(penalty = best_f1['classifier__penalty'],
                                  solver = best_f1['classifier__solver'],
                                  max_iter = 5000,
                                  n_jobs = -1,
                                  class_weight='balanced')

# Training on the new model
f1_model.fit(X_train, y_train)
y_f1_train = f1_model.predict(X_train)
f1_train_score = f1_score(y_train, y_f1_train)
```

```
    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_log = train_metrics_log.append({'LG: Accuracy': acc_train_scor
                                                  'LG: AUC': roc_train_score, 'LG: F1':

    test_metrics_log = test_metrics_log.append({'LG: Accuracy': acc_test_score,
                                                  'LG: AUC': roc_test_score, 'LG: F1': f

    return train_metrics_log, test_metrics_log, solution_log
```

RANDOM FOREST CLASSIFIER

In [26]:

```
def randomForest(X_train, X_test, y_train, y_test):
    train_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:
    test_metrics_rf = pd.DataFrame(columns=['RF: Accuracy','RF: Precision','RF:

    randomForest = RandomForestClassifier()

    # Setting parameters according to CNM06
    param_grid = {
        'n_estimators': [1024],
        'criterion': ['gini', 'entropy'],
        'max_features': [1,2,4,6,8,12,16],
        'n_jobs': [-1],
        'class_weight': ['balanced']}

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(estimator=randomForest, param_grid=param_grid, cv=Stratif
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
    solution_rf = pd.DataFrame(results)

    # ACCURACY MODEL
    solution_rf['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
    best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
    # creating new model with optimal hyperparameters
    accuracy_model = RandomForestClassifier(n_estimators = best_accuracy['n_esti
                                            criterion = best_accuracy['criterion
                                            max_features = best_accuracy['max_fe
                                            n_jobs = -1,
                                            class_weight='balanced')
    # Training on the new model
    accuracy_model.fit(X_train, y_train)
    y_acc_train = accuracy_model.predict(X_train)
    acc_train_score = accuracy_score(y_train, y_acc_train)

    # Testing and scoring the model
    y_acc_test = accuracy_model.predict(X_test)
    acc_test_score = accuracy_score(y_test, y_acc_test)

    # PRECISION MODEL
    solution_rf['Precision'] = hyperparams.cv_results_['mean_test_precision']
    best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
```

```
    # creating new model with optimal hyperparameters
    precision_model = RandomForestClassifier(n_estimators = best_precision['n_es
                                             criterion = best_precision['criteri
                                             max_features = best_precision['max_
                                             n_jobs = -1,
                                             class_weight='balanced')
    # Training on the new model
    precision_model.fit(X_train, y_train)
    y_prec_train = precision_model.predict(X_train)
    prec_train_score = precision_score(y_train, y_prec_train)

    # Testing and scoring the model
    y_prec_test = precision_model.predict(X_test)
    prec_test_score = precision_score(y_test, y_prec_test)

    # ROC AUC MODEL
    solution_rf['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
    best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
    # creating new model with optimal hyperparameters
    roc_model = RandomForestClassifier(n_estimators = best_roc_auc['n_estimators
                                       criterion = best_roc_auc['criterion'],
                                       max_features = best_roc_auc['max_features
                                       n_jobs = -1,
                                       class_weight='balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1 MODEL
    solution_rf['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
    # creating new model with optimal hyperparameters
    f1_model = RandomForestClassifier(n_estimators = best_f1['n_estimators'],
                                      criterion = best_f1['criterion'],
                                      max_features = best_f1['max_features'],
                                      n_jobs = -1,
                                      class_weight='balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_rf = train_metrics_rf.append({'RF: Accuracy': acc_train_score,
                                                'RF: AUC': roc_train_score, 'RF:

    test_metrics_rf = test_metrics_rf.append({'RF: Accuracy': acc_test_score, 'R
                                             'RF: AUC': roc_test_score, 'RF: F1

    return train_metrics_rf, test_metrics_rf, solution_rf
```

DECISION TREE CLASSIFIER

In [27]:
```
def decisionTrees(X_train, X_test, y_train, y_test):
    train_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:
    test_metrics_dt = pd.DataFrame(columns=['DT: Accuracy','DT: Precision','DT:

    pipe = Pipeline(steps=[('classifier', DecisionTreeClassifier())])

    # Setting parameters according to CNM06 + passing a list of min_samples_leaf
    parameters = [{'classifier': [DecisionTreeClassifier(class_weight='balanced'
                  'classifier__criterion': ['gini', 'entropy'],
                  'classifier__splitter': ['best'],
                  'classifier__min_samples_leaf': [1,2,4,6,8,10,12,14,16,18]}]

    # Perform 5-fold cross-validation using grid search
    clf = GridSearchCV(estimator=pipe, param_grid=parameters, cv=StratifiedKFold
                       scoring=['accuracy', 'precision', 'roc_auc', 'f1'], refit

    # Fitting training set on the cross validation
    hyperparams = clf.fit(X_train, y_train)
    # Storing best parameters for each metric model
    results = hyperparams.cv_results_['params']
    solution_dt = pd.DataFrame(results)

    # ACCURACY
    solution_dt['Accuracy'] = hyperparams.cv_results_['mean_test_accuracy']
    best_accuracy = results[np.argmin(hyperparams.cv_results_['rank_test_accurac
    # creating new model with optimal hyperparameters
    accuracy_model = DecisionTreeClassifier(criterion = best_accuracy['classifie
                                            splitter = best_accuracy['classifier
                                            min_samples_leaf = best_accuracy['cl
                                            class_weight = 'balanced')

    # Training on the new model
    accuracy_model.fit(X_train, y_train)
    y_acc_train = accuracy_model.predict(X_train)
    acc_train_score = accuracy_score(y_train, y_acc_train)

    # Testing and scoring the model
    y_acc_test = accuracy_model.predict(X_test)
    acc_test_score = accuracy_score(y_test, y_acc_test)

    # PRECISION
    solution_dt['Precision'] = hyperparams.cv_results_['mean_test_precision']
    best_precision = results[np.argmin(hyperparams.cv_results_['rank_test_precis
    # creating new model with optimal hyperparameters
    precision_model = DecisionTreeClassifier(criterion = best_precision['classif
                                             splitter = best_precision['classifi
                                             min_samples_leaf = best_precision['
                                             class_weight = 'balanced')
    # Training on the new model
    precision_model.fit(X_train, y_train)
    y_prec_train = precision_model.predict(X_train)
    prec_train_score = precision_score(y_train, y_prec_train)

    # Testing and scoring the model
    y_prec_test = precision_model.predict(X_test)
    prec_test_score = precision_score(y_test, y_prec_test)

    # ROC AUC
```

```
    solution_dt['ROC AUC'] = hyperparams.cv_results_['mean_test_roc_auc']
    best_roc_auc = results[np.argmin(hyperparams.cv_results_['rank_test_roc_auc'
    # creating new model with optimal hyperparameters
    roc_model = DecisionTreeClassifier(criterion = best_roc_auc['classifier__cri
                                       splitter = best_roc_auc['classifier__spli
                                       min_samples_leaf = best_roc_auc['classifi
                                       class_weight = 'balanced')
    # Training on the new model
    roc_model.fit(X_train, y_train)
    y_roc_train = roc_model.predict(X_train)
    roc_train_score = roc_auc_score(y_train, y_roc_train)

    # Testing and scoring the model
    y_roc_test = roc_model.predict(X_test)
    roc_test_score = roc_auc_score(y_test, y_roc_test)

    # F1
    solution_dt['F1'] = hyperparams.cv_results_['mean_test_f1']
    best_f1 = results[np.argmin(hyperparams.cv_results_['rank_test_f1'])]
    # creating new model with optimal hyperparameters
    f1_model = DecisionTreeClassifier(criterion = best_f1['classifier__criterion
                                      splitter = best_f1['classifier__splitter']
                                      min_samples_leaf = best_f1['classifier__mi
                                      class_weight = 'balanced')
    # Training on the new model
    f1_model.fit(X_train, y_train)
    y_f1_train = f1_model.predict(X_train)
    f1_train_score = f1_score(y_train, y_f1_train)

    # Testing and scoring the model
    y_f1_test = f1_model.predict(X_test)
    f1_test_score = f1_score(y_test, y_f1_test)

    train_metrics_dt = train_metrics_dt.append({'DT: Accuracy': acc_train_score,
                                                'DT: AUC': roc_train_score, 'DT:

    test_metrics_dt = test_metrics_dt.append({'DT: Accuracy': acc_test_score, 'D
                                              'DT: AUC': roc_test_score, 'DT: F1

    return train_metrics_dt, test_metrics_dt, solution_dt
```

RUNNING DATASET ON ALL THREE ALGORITHMS

In [28]:

```
trials = 5
train_metrics = pd.DataFrame()
test_metrics = pd.DataFrame()
solution_metrics = pd.DataFrame()

# Running the trial five times
for i in range(trials):
    # Splitting data into train size = 5000
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=5000, s

    train_log, test_log, solution_log = logisticRegression(X_train, X_test, y_tr
    train_rf, test_rf, solution_rf = randomForest(X_train, X_test, y_train, y_te
    train_dt, test_dt, solution_dt = decisionTrees(X_train, X_test, y_train, y_t
```

```
    train_metrics = train_metrics.append(pd.concat([train_log, train_rf, train_d
    test_metrics = test_metrics.append(pd.concat([test_log, test_rf, test_dt], a
    solution_metrics = solution_metrics.append(pd.concat([solution_log, solution

# storing data into CSV file
train_metrics.to_csv('letterp2_train.csv')
test_metrics.to_csv('letterp2_test.csv')
solution_metrics.to_csv('letterp2_solution.csv')
```

```
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   10.8s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 13.6min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:   13.7s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   19.8s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 17.9min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:   12.1s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   15.9s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 19.2min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:   14.1s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   25.6s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 15.4min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:    4.3s finished
Fitting 5 folds for each of 41 candidates, totalling 205 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 205 out of 205 | elapsed:   11.8s finished
Fitting 5 folds for each of 14 candidates, totalling 70 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  70 out of  70 | elapsed: 10.1min finished
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  90 out of  90 | elapsed:    3.4s finished
```

In [ ]:

In [3]:
```
import pandas as pd
from statistics import mean
from scipy.stats import ttest_rel
from scipy.stats import ttest_ind
import numpy as np
```

GATHERING TRAIN METRIC RESULTS FROM ALL DATASETS

Reading the training csv of all datasets, rearranging table format, and finding the mean for each metric and algorithm combination.

In [2]:
```
adult = pd.read_csv("adult_train.csv")
adult = adult.drop(adult.iloc[:, :1], axis = 1).T
adult['Mean'] = adult.mean(axis=1)
adult.round(4)
```

Out[2]:

	0	1	2	3	4	Mean
LG: Accuracy	0.8254	0.8162	0.8032	0.8174	0.8148	0.8154
LG: Precision	0.3967	0.4304	0.4740	0.6583	0.5023	0.4923
LG: AUC	0.8329	0.8267	0.8174	0.8324	0.8269	0.8273
LG: F1	0.7079	0.6995	0.6830	0.6927	0.6990	0.6964
RF: Accuracy	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: Precision	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: AUC	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: F1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: Accuracy	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: Precision	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: AUC	0.8514	0.8465	0.8441	0.8442	0.8441	0.8461
DT: F1	0.7169	0.7320	0.7224	0.7187	0.7162	0.7213

In [3]:
```
bank = pd.read_csv("bank_train.csv")
bank = bank.drop(bank.iloc[:, :1], axis = 1).T
bank['Mean'] = bank.mean(axis=1)
bank.round(4)
```

Out[3]:

	0	1	2	3	4	Mean
LG: Accuracy	0.8748	0.8818	0.8540	0.8444	0.8766	0.8663
LG: Precision	0.1258	0.6183	0.4261	0.1351	0.5678	0.3746
LG: AUC	0.8405	0.8380	0.8380	0.8192	0.8390	0.8350
LG: F1	0.5746	0.5836	0.5799	0.5490	0.5862	0.5747
RF: Accuracy	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	0	1	2	3	4	Mean
RF: Precision	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: AUC	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: F1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: Accuracy	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: Precision	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: AUC	0.8761	0.8711	0.8729	0.8701	0.8745	0.8729
DT: F1	0.7420	0.5810	0.5968	0.6547	0.6755	0.6500

In [4]:
```
employee = pd.read_csv("employee_train.csv")
employee = employee.drop(employee.iloc[:, :1], axis = 1).T
employee['Mean'] = employee.mean(axis=1)
employee.round(4)
```

Out[4]:

	0	1	2	3	4	Mean
LG: Accuracy	0.7536	0.7580	0.2394	0.4240	0.2404	0.4831
LG: Precision	0.3947	0.3848	0.3887	0.3979	0.3870	0.3906
LG: AUC	0.6605	0.6867	0.6724	0.6801	0.6578	0.6715
LG: F1	0.5259	0.5163	0.5164	0.5203	0.5129	0.5184
RF: Accuracy	0.9974	0.9970	0.9966	0.9966	0.9964	0.9968
RF: Precision	0.9888	0.9878	0.9860	0.9865	0.9860	0.9870
RF: AUC	0.9980	0.9980	0.9976	0.9977	0.9973	0.9977
RF: F1	0.9944	0.9938	0.9929	0.9932	0.9926	0.9934
DT: Accuracy	0.9972	0.9970	0.9966	0.9966	0.9398	0.9854
DT: Precision	0.9888	0.9878	0.9860	0.9865	0.9836	0.9865
DT: AUC	0.9515	0.9566	0.9492	0.9459	0.9416	0.9489
DT: F1	0.9944	0.9938	0.9929	0.9932	0.8805	0.9710

In [5]:
```
letterp1 = pd.read_csv("letterp1_train.csv")
letterp1 = letterp1.drop(letterp1.iloc[:, :1], axis = 1).T
letterp1['Mean'] = letterp1.mean(axis=1)
letterp1.round(4)
```

Out[5]:

	0	1	2	3	4	Mean
LG: Accuracy	0.8094	0.0364	0.9626	0.7774	0.8000	0.6772
LG: Precision	0.1501	0.1225	0.1316	0.1351	0.1417	0.1362
LG: AUC	0.8581	0.8277	0.8420	0.8359	0.8561	0.8440
LG: F1	0.2584	0.2170	0.2298	0.2351	0.2504	0.2381
RF: Accuracy	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: Precision	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	0	1	2	3	4	Mean
RF: AUC	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: F1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: Accuracy	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: Precision	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: AUC	0.9781	0.9698	0.9683	0.9726	0.9725	0.9723
DT: F1	1.0000	1.0000	0.9590	1.0000	1.0000	0.9918

In [6]:
```
letterp2 = pd.read_csv("letterp2_train.csv")
letterp2 = letterp2.drop(letterp2.iloc[:, :1], axis = 1).T
letterp2['Mean'] = letterp2.mean(axis=1)
letterp2.round(4)
```

Out[6]:

	0	1	2	3	4	Mean
LG: Accuracy	0.7282	0.7126	0.7350	0.7304	0.7324	0.7277
LG: Precision	0.6572	0.7014	0.7269	0.7285	0.7208	0.7070
LG: AUC	0.7286	0.7124	0.7351	0.7304	0.7319	0.7277
LG: F1	0.7324	0.7118	0.7348	0.7334	0.7362	0.7297
RF: Accuracy	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: Precision	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: AUC	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RF: F1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
DT: Accuracy	1.0000	1.0000	1.0000	0.9772	1.0000	0.9954
DT: Precision	1.0000	1.0000	1.0000	0.9922	1.0000	0.9984
DT: AUC	0.8888	0.9248	0.9060	0.8946	0.9050	0.9039
DT: F1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

MEAN TRAINING SET PERFORMANCE

Finding the mean training performance across five trials over five datasets.

In [7]:
```
table2 = pd.DataFrame(index=['Accuracy', 'Precision', 'ROC AUC', 'F1'],
                      columns=['Logistic Regression', 'Random Forest', 'Decision
for i in range(table2.shape[0]):
    # index number loops over metrics ['Accuracy', 'Precision', 'ROC AUC', 'F1']
    # Skips every four columns to get the corresponding learning algorithm with
    table2['Logistic Regression'].iloc[i] = mean([bank['Mean'].iloc[i],adult['Me
                                                  letterp2['Mean'].iloc[i],emplo
    table2['Random Forest'].iloc[i] = mean([bank['Mean'].iloc[i+4],adult['Mean']
                                            letterp2['Mean'].iloc[i+4],emp
    table2['Decision Trees'].iloc[i] = mean([bank['Mean'].iloc[i+8],adult['Mean'
                                             letterp2['Mean'].iloc[i+8],emp

table2 = table2.T
```

```
table2['Mean'] = table2.mean(axis=1)
table2.round(4)
```

Out[7]:

	Accuracy	Precision	ROC AUC	F1	Mean
Logistic Regression	0.713936	0.420152	0.781081	0.551456	0.6167
Random Forest	0.99936	0.997401	0.999549	0.998675	0.9987
Decision Trees	0.996176	0.996992	0.908809	0.866801	0.9422

In [8]:

```
# Finding average metrics over each dataset
table3 = pd.DataFrame(columns=['Adult', 'Bank', 'Employee', 'Letter P1', 'Letter
                      index=['Logistic Regression', 'Random Forest', 'Decision T
for i in range(table3.shape[0]):
    # index number loops over learning algorithms ['Logistic Regression', 'Rando
    # getting the mean of each dataset over all four metrics
    table3['Adult'].iloc[i] = mean(adult['Mean'].iloc[4*i:4*i+4])
    table3['Bank'].iloc[i] = mean(bank['Mean'].iloc[4*i:4*i+4])
    table3['Employee'].iloc[i] = mean(employee['Mean'].iloc[4*i:4*i+4])
    table3['Letter P1'].iloc[i] = mean(letterp1['Mean'].iloc[4*i:4*i+4])
    table3['Letter P2'].iloc[i] = mean(letterp2['Mean'].iloc[4*i:4*i+4])

table3['Mean'] = table3.mean(axis=1)
table3.round(4)
```

Out[8]:

	Adult	Bank	Employee	Letter P1	Letter P2	Mean
Logistic Regression	0.707859	0.662639	0.515894	0.473863	0.723025	0.6167
Random Forest	1	1	0.993732	1	1	0.9987
Decision Trees	0.891828	0.880728	0.972969	0.991014	0.974434	0.9422

GATHERING TEST METRIC RESULTS FROM ALL DATASETS

Reading the training csv of all datasets, rearranging table format, and finding the mean for each metric and algorithm combination.

In [4]:

```
adult = pd.read_csv("adult_test.csv")
adult = adult.drop(adult.iloc[:, :1], axis = 1).T
adult['Mean'] = adult.mean(axis=1)
adult.round(4)
```

Out[4]:

	0	1	2	3	4	Mean
LG: Accuracy	0.8074	0.8040	0.8069	0.8039	0.8023	0.8049
LG: Precision	0.3951	0.4072	0.4769	0.6556	0.4993	0.4868
LG: AUC	0.8163	0.8159	0.8172	0.8181	0.8162	0.8168
LG: F1	0.6824	0.6790	0.6823	0.6828	0.6775	0.6808
RF: Accuracy	0.8492	0.8491	0.8470	0.8487	0.8484	0.8485
RF: Precision	0.7337	0.7279	0.7317	0.7385	0.7275	0.7319

	0	1	2	3	4	Mean
RF: AUC	0.7704	0.7732	0.7663	0.7716	0.7721	0.7707
RF: F1	0.6681	0.6711	0.6612	0.6691	0.6709	0.6681
DT: Accuracy	0.8051	0.8058	0.8058	0.8037	0.8037	0.8048
DT: Precision	0.6033	0.5922	0.6005	0.6074	0.6022	0.6011
DT: AUC	0.7959	0.7982	0.7919	0.8026	0.7966	0.7971
DT: F1	0.6514	0.6633	0.6552	0.6531	0.6535	0.6553

In [5]:
```
bank = pd.read_csv("bank_test.csv")
bank = bank.drop(bank.iloc[:, :1], axis = 1).T
bank['Mean'] = bank.mean(axis=1)
bank.round(4)
```

Out[5]:

	0	1	2	3	4	Mean
LG: Accuracy	0.8661	0.8773	0.8535	0.8564	0.8679	0.8642
LG: Precision	0.1329	0.5818	0.4236	0.1397	0.5661	0.3688
LG: AUC	0.8243	0.8262	0.8196	0.8306	0.8233	0.8248
LG: F1	0.5727	0.5647	0.5665	0.5715	0.5631	0.5677
RF: Accuracy	0.8925	0.8952	0.8936	0.8897	0.8932	0.8928
RF: Precision	0.7280	0.7180	0.7034	0.7274	0.7073	0.7168
RF: AUC	0.6627	0.6570	0.6802	0.6677	0.6651	0.6665
RF: F1	0.4859	0.5069	0.4937	0.4750	0.4811	0.4885
DT: Accuracy	0.8651	0.8614	0.8632	0.8635	0.8646	0.8636
DT: Precision	0.4630	0.4521	0.4516	0.4713	0.4736	0.4623
DT: AUC	0.8285	0.8193	0.8143	0.8218	0.7882	0.8144
DT: F1	0.5094	0.5181	0.5181	0.5160	0.5008	0.5125

In [6]:
```
employee = pd.read_csv("employee_test.csv")
employee = employee.drop(employee.iloc[:, :1], axis = 1).T
employee['Mean'] = employee.mean(axis=1)
employee.round(4)
```

Out[6]:

	0	1	2	3	4	Mean
LG: Accuracy	0.7661	0.7639	0.2374	0.4172	0.2369	0.4843
LG: Precision	0.3658	0.3690	0.3796	0.3912	0.3721	0.3755
LG: AUC	0.6489	0.6692	0.6593	0.6863	0.6481	0.6624
LG: F1	0.4958	0.4990	0.5063	0.5179	0.4922	0.5022
RF: Accuracy	0.9600	0.9574	0.9604	0.9617	0.9596	0.9598
RF: Precision	0.9079	0.9139	0.9186	0.9123	0.9204	0.9146
RF: AUC	0.9442	0.9372	0.9428	0.9463	0.9436	0.9428

	0	1	2	3	4	Mean
RF: F1	0.9140	0.9097	0.9157	0.9187	0.9135	0.9143
DT: Accuracy	0.9396	0.9452	0.9441	0.9443	0.9331	0.9413
DT: Precision	0.8594	0.8780	0.8714	0.8669	0.8685	0.8688
DT: AUC	0.9297	0.9284	0.9341	0.9322	0.9249	0.9299
DT: F1	0.8736	0.8870	0.8818	0.8825	0.8673	0.8784

In [7]:
```
letterp1 = pd.read_csv("letterp1_test.csv")
letterp1 = letterp1.drop(letterp1.iloc[:, :1], axis = 1).T
letterp1['Mean'] = letterp1.mean(axis=1)
letterp1.round(4)
```

Out[7]:

	0	1	2	3	4	Mean
LG: Accuracy	0.8021	0.0381	0.9623	0.7875	0.7881	0.6756
LG: Precision	0.1483	0.1371	0.1324	0.1371	0.1385	0.1387
LG: AUC	0.8390	0.8432	0.8435	0.8345	0.8318	0.8384
LG: F1	0.2538	0.2346	0.2314	0.2373	0.2396	0.2394
RF: Accuracy	0.9885	0.9887	0.9901	0.9898	0.9870	0.9888
RF: Precision	0.9970	0.9591	0.9833	0.9914	0.9915	0.9844
RF: AUC	0.8368	0.8790	0.8696	0.8657	0.8420	0.8586
RF: F1	0.8299	0.8431	0.8577	0.8401	0.8068	0.8355
DT: Accuracy	0.9851	0.9837	0.9847	0.9843	0.9839	0.9843
DT: Precision	0.8346	0.7695	0.8151	0.8029	0.8242	0.8093
DT: AUC	0.9242	0.9305	0.9361	0.9279	0.9181	0.9274
DT: F1	0.7846	0.7880	0.7654	0.7649	0.7854	0.7777

In [8]:
```
letterp2 = pd.read_csv("letterp2_test.csv")
letterp2 = letterp2.drop(letterp2.iloc[:, :1], axis = 1).T
letterp2['Mean'] = letterp2.mean(axis=1)
letterp2.round(4)
```

Out[8]:

	0	1	2	3	4	Mean
LG: Accuracy	0.7262	0.7204	0.7279	0.7214	0.7240	0.7240
LG: Precision	0.6433	0.7176	0.7184	0.7202	0.7151	0.7029
LG: AUC	0.7265	0.7211	0.7278	0.7213	0.7252	0.7244
LG: F1	0.7301	0.7210	0.7314	0.7178	0.7271	0.7255
RF: Accuracy	0.9489	0.9507	0.9483	0.9481	0.9431	0.9478
RF: Precision	0.9459	0.9500	0.9534	0.9515	0.9470	0.9496
RF: AUC	0.9499	0.9507	0.9479	0.9477	0.9442	0.9481
RF: F1	0.9479	0.9501	0.9491	0.9458	0.9428	0.9471

	0	1	2	3	4	Mean
DT: Accuracy	0.8873	0.8907	0.8967	0.8907	0.8867	0.8904
DT: Precision	0.8881	0.8912	0.8984	0.9077	0.8896	0.8950
DT: AUC	0.8528	0.8750	0.8619	0.8484	0.8538	0.8584
DT: F1	0.8843	0.8932	0.8944	0.8924	0.8865	0.8902

TABLE 2

In [9]:
```
# Finding algorithm metrics over all 5 datasets
table2 = pd.DataFrame(index=['Accuracy', 'Precision', 'ROC AUC', 'F1'],
                      columns=['Logistic Regression', 'Random Forest', 'Decision
for i in range(table2.shape[0]):
    # index number loops over metrics ['Accuracy', 'Precision', 'ROC AUC', 'F1']
    # Skips every four columns to get the corresponding learning algorithm with
    table2['Logistic Regression'].iloc[i] = mean([bank['Mean'].iloc[i],adult['Me
                                                  letterp2['Mean'].iloc[i],emplo
    table2['Random Forest'].iloc[i] = mean([bank['Mean'].iloc[i+4],adult['Mean']
                                                  letterp2['Mean'].iloc[i+4],emp
    table2['Decision Trees'].iloc[i] = mean([bank['Mean'].iloc[i+8],adult['Mean'
                                                  letterp2['Mean'].iloc[i+8],emp

table2 = table2.T
table2['Mean'] = table2.mean(axis=1)
table2.round(4)
```

Out[9]:

	Accuracy	Precision	ROC AUC	F1	Mean
Logistic Regression	0.710605	0.414565	0.773346	0.543115	0.6104
Random Forest	0.927549	0.85947	0.83735	0.77072	0.8488
Decision Trees	0.896866	0.727308	0.865417	0.742815	0.8081

In [10]:
```
table2_ttest = pd.DataFrame(index=['Accuracy', 'Precision', 'ROC AUC', 'F1'],
                            columns=['LG', 'RF', 'DT'])
for i in range(4):
    # index number loops over metrics ['Accuracy', 'Precision', 'ROC AUC', 'F1']
    # Skips every four columns to get the corresponding learning algorithm with
    # Gets the list of all the trial scores for the corresponding metrics

    # Logistic Regression
    val1 = np.concatenate([bank.iloc[i,:-1],adult.iloc[i,:-1],letterp1.iloc[i,:-
                           letterp2.iloc[i,:-1],employee.iloc[i,:-1]])
    # Random Forest
    val2 = np.concatenate([bank.iloc[i+4,:-1],adult.iloc[i+4,:-1],letterp1.iloc[
                           letterp2.iloc[i+4,:-1],employee.iloc[i+4,:-1]])
    # Decision Trees
    val3 = np.concatenate([bank.iloc[i+8,:-1],adult.iloc[i+8,:-1],letterp1.iloc[
                           letterp2.iloc[i+8,:-1],employee.iloc[i+8,:-1]])
    # Finding p-value using independent t-test
    if i == 2:
        table2_ttest['LG'].iloc[i] = ttest_ind(val3, val1)[1]
        table2_ttest['RF'].iloc[i] = ttest_ind(val3, val2)[1]
        table2_ttest['DT'].iloc[i] = float(1)
    else:
```

```
        table2_ttest['LG'].iloc[i] = ttest_ind(val2, val1)[1]
        table2_ttest['RF'].iloc[i] = float(1)
        table2_ttest['DT'].iloc[i] = ttest_ind(val2, val3)[1]

table2_ttest.T
```

Out[10]:

	Accuracy	Precision	ROC AUC	F1
LG	2.58435e-05	3.60369e-12	6.08411e-06	3.07918e-05
RF	1	1	0.262249	1
DT	0.066122	0.00247935	1	0.54288

TABLE 3

In [12]:

```
# Finding average metrics over each dataset
table3 = pd.DataFrame(columns=['Adult', 'Bank', 'Employee', 'Letter P1', 'Letter
                      index=['Logistic Regression', 'Random Forest', 'Decision T
for i in range(table3.shape[0]):
    # index number loops over learning algorithms ['Logistic Regression', 'Rando
    # getting the mean of each dataset over all four metrics
    table3['Adult'].iloc[i] = mean(adult['Mean'].iloc[4*i:4*i+4])
    table3['Bank'].iloc[i] = mean(bank['Mean'].iloc[4*i:4*i+4])
    table3['Employee'].iloc[i] = mean(employee['Mean'].iloc[4*i:4*i+4])
    table3['Letter P1'].iloc[i] = mean(letterp1['Mean'].iloc[4*i:4*i+4])
    table3['Letter P2'].iloc[i] = mean(letterp2['Mean'].iloc[4*i:4*i+4])

table3['Mean'] = table3.mean(axis=1)
table3.round(4)
```

Out[12]:

	Adult	Bank	Employee	Letter P1	Letter P2	Mean
Logistic Regression	0.697329	0.656393	0.506114	0.473017	0.719186	0.6104
Random Forest	0.754797	0.691171	0.932897	0.916847	0.94815	0.8488
Decision Trees	0.714578	0.663196	0.904595	0.874645	0.883494	0.8081

In [17]:

```
dataset = [adult, bank, employee, letterp1, letterp2]
table3_ttest = pd.DataFrame(index=['Adult', 'Bank', 'Employee', 'Letter P1', 'Le
                            columns=['LG', 'RF', 'DT'])
for i, data in enumerate(dataset):
    # index number loops over dataset ['Adult', 'Bank', 'Employee', 'Letter P1',
    # Get the list of all trail scores for the corresponding dataset
    # Skips every four columns to get the corresponding learning algorithm

    # Logistic Regression
    val1 = np.concatenate([data.iloc[:4,0],data.iloc[:4,1],data.iloc[:4,2],data.
    # Random Forest
    val2 = np.concatenate([data.iloc[4:8,0],data.iloc[4:8,1],data.iloc[4:8,2],da
    # Decision Tree
    val3 = np.concatenate([data.iloc[8:12,0],data.iloc[8:12,1],data.iloc[8:12,2]

    # Finding p-value using independent t-test
    table3_ttest['LG'].iloc[i] = ttest_ind(val2, val1)[1]
    table3_ttest['RF'].iloc[i] = float(1)
    table3_ttest['DT'].iloc[i] = ttest_ind(val2, val3)[1]
table3_ttest.T
```

Out[17]:

	Adult	Bank	Employee	Letter P1	Letter P2
LG	0.114593	0.57316	3.43025e-14	1.67169e-06	7.87304e-38
RF	1	1	1	1	1
DT	0.119651	0.597412	0.00213068	0.106842	1.17304e-19

In [ ]: