Cogs 118A Final Project
Reproducing The Comparison Of Supervised Learning Algorithms

Palaksh Vardhan Rungta PRUNGTA@UCSD.EDU
Junior at UCSD,
9450 Gilman Drive,
San Diego, CA

Abstract

The paper by Caruana, R., & Niculescu-Mizil, A. does in-depth research on the differences between the performance of multiple supervised learning algorithms on the binary classification task. Their goal was to explore which algorithms and its hyperparameters set had an optimal performance across various types of datasets. This paper is an attempt to reproduce and extend their work as it tests out three supervised learning algorithms - ANN, Random Forest, and Logistic Regression - over four different datasets to find out how the algorithms behave on different types of binary classification datasets.
Keywords: Logistic Regression, Artificial Neural Networks, Random Forest, Binary classification.

1. Introduction

There is a huge variety of supervised machine learning algorithms today. We could say that there is an algorithm for almost every type of problem out there. However, these algorithms are no one-size-fits-all solution as each of these have their own pros and cons, and vary in use from problem to problem. When choosing an algorithm for a specific problem, one may be concerned about its runtime or the model's performance over several metrics, all of which require tuning the model's hyperparameters appropriately. Thus, this paper, as an extension and reproduction of the paper by Caruana, R., & Niculescu-Mizil, A., aims to look into three separate supervised learning algorithms - ANN, Random Forest, and Logistic Regression - over four different types of binary classification problems to find out which algorithms produce the best results after choosing the appropriate hyperparameters. We measure the models' performance over three metrics - Accuracy, F1 score, and ROC AUC which are all in the range [0, 1].

2. Methodology

2.1 Datasets and Description

All of the data used was taken from the UCI machine learning repository. We used the following datasets for the binary classification task:

Occupancy: This dataset is used to determine whether a room is occupied or not based on various attributes.

Gamma: This dataset is used to classify radiations as either gamma or hadron from the cosmic rays in the atmosphere.

Eye: This dataset is used to determine whether the eye is closed or not based on EEG measurements used as attributes.

Adult: This dataset is used to predict whether the salary of an individual is greater than $50,000 or not based on various attributes. As this dataset had categorical values, we used one hot encoding to convert them to numerical values which increased the number of attributes. This dataset was used in the Caruana, R., & Niculescu-Mizil, A. paper.

Refer to Table 1 below to know more about the datasets and its features. Also refer to Appendix D for more details on data cleaning.

Dataset	Attributes	Train Size	Test Size	% Positive Class
Occupancy	5	5000	15560	23.1%
Gamma	10	5000	14020	64.8%
Eye	14	5000	9980	44.8%
Adult	14/108	5000	27561	75.9%

Table 1: About Datasets

2.2 Algorithms and Hyperparameters

To find the optimal performance of an algorithm over a dataset, we need to find the best hyperparameters required for the task. In our paper, we used the following hyperparameters to tune the models as mentioned in the Caruana, R., & Niculescu-Mizil, A. paper through scikit-learn's implementation of the following classifiers:

ANN: To run an artificial neural network, we used two different stochastic gradient descent solvers named adam and sgd in sklearn along with varying the momentum parameter of the sgd solver and the number of hidden layers parameters for both solvers.

Parameters implemented via sklearn.neural_network.MLPClassifier:

- Solvers: adam and sgd.
- Number of hidden layers of the perceptron: [1, 2, 4, 8, 32, 128]
- Momentum (only for sgd): [0, 0.2, 0.5, 0.9]

Logistic Regression: We used the probabilistic logistic regression algorithm by varying its regularization parameter and also a non regularized model.

Parameters implemented via sklearn.linear_model.LogisticRegression:

- Regularization parameter C (with l2): [10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}, 10^{4}]
- Penalty: 'l2' and 'none' (no regularization).

Random Forest: Random forest bootstraps the training data to make multiple decision trees. We used random forest with 1024 trees and varied the number of features to split at each node which influences its depth.

Parameters implemented via sklearn.ensemble.RandomForestClassifier:

- Number of trees: 1024
- Features to split: [1, 2, 4, 6, 8, 12, 16, 20]

2.3 Metrics

We are using three metrics to measure our model's performance and decide which hyperparameters to use - Accuracy, F1 Score, and ROC AUC. As they all range from [0, 1] with 1 being the best performance and 0 being the lowest, it makes it easier for us to average the performance of the model over the three metrics.

Accuracy is a metric based on the number of true positives and negatives in the tested dataset.
F1 score is a metric based on the precision and recall of the tested dataset.
Receiver Operator Characteristics - Area Under the Curve (ROC AUC) is a metric based on the number of true positives and false positives in the tested dataset.

These three metrics measure performance differently which can provide valuable insights as a high value in one of the metrics does not necessarily imply a high value in the other metric.

3. Experiment

3.1 Procedure

We closely follow the procedure outlined by the Caruana, R., & Niculescu-Mizil, A. paper. To pick the optimal algorithm for our different types of datasets, we took 5 trials for each algorithm and dataset combination. In each trial, we randomly split the dataset into 5000 instances of training data without replacement, and the remaining as testing data. To deal with class imbalance in training data, we stratified the splitting process which meant there was roughly an equal percentage of positive and negative classes in both the training and testing data. We then performed a 5 fold cross validation on the training data and varied the hyperparameters settings (as mentioned in Section 2.2) using a Grid Search. The mean across the 5 folds was taken for each of the three metrics with which we were able to determine the optimal hyperparameters. After this, we tuned our model with the 3 sets of hyperparameters corresponding to the 3 different metrics and fit it on the whole training data. Here, we

measured the performance of the 3 models on the testing data and the training data using the 3 metrics. We then stored the mean across the 3 models, to obtain the testing and training set performances over 3 metrics. We stored this data for each trial and performed the whole process for each dataset and algorithm combination (12 times). Refer to Appendix D for the code used to run the above process.

3.2 Results

We summarized our testing set performance results for each algorithm and metric in Table 2 below. Table 2 contains the mean metric performance of each algorithm across the 4 datasets (and its 5 trials). Notice that we have bolded the best performing algorithm per metric in Table 2. We have also marked cells which are not statistically different (p-value > 0.05) to the best algorithm in a given metric with an asterisk (*). We used a two sample t-test to find the statistical significance. Refer to Appendix C (i) for exact p-values between algorithms.

Algorithms	Accuracy	F1 Score	ROC AUC	Mean
Logistic Regression	0.8161762	0.8183189	0.7808568	0.805118
ANN	0.8850869*	0.8872495*	0.8532641*	0.875200*
Random Forest	**0.9036053**	**0.9187275**	**0.8762643**	**0.899532**

Table 2: Average metric performance of each algorithm across 4 datasets

Similar to Table 2, we have summarized the testing set performance for each algorithm and dataset below by taking the mean across the 3 metrics in Table 3. We have bolded the algorithm which performed the best in each dataset and marked cells that are not statistically significant (p-value > 0.05) from the best algorithm with an asterisk, measured with a two samples t-test. Refer to Appendix C (ii) for exact p-values between algorithms.

Algorithms	Adult	Gamma	Eye	Occupancy	Mean
Logistic Regression	0.8383774*	0.7928565	0.6035762	0.9856592*	0.805118
ANN	0.8343662*	0.8686939*	0.8495585	0.9481821*	0.875200*
Random Forest	**0.8442684**	**0.8726706**	**0.8938351**	**0.9873552**	**0.899532**

Table 3: Average dataset performance of each algorithm across 3 metrics

For the raw testing data performance for each algorithm, dataset, and metric combination, please refer to Appendix B and its sub sections.

4. Discussion

From Tables 2 and 3, we can observe that the Random Forest algorithm has the best performance across all datasets and metrics. ANN is a close second when averaged over datasets, as indicated by the asterisk in Table 2, and a close second for the datasets Adult, Gamma, and Occupancy when averaged over metrics, as indicated by the asterisk in Table 3. We can observe that Logistic Regression's performance comes close to these two algorithms' in the Adult and the Occupancy datasets, but performs the worst in the other two. This is expected as Logistic Regression assumes linearity which helps with its low training runtime, but can be a poor fit for datasets that are not linearly separable. We can observe its poor performance over non-linearly separable datasets even in its training set performance scores for the Gamma and Eye datasets which can be found in Appendix A (iii). Thus, Logistic Regression should be avoided at all costs if the dataset is highly complex and not linearly separable.

ANN, on the other hand, has a close performance to Random Forest almost everywhere. The perceptron learning algorithm follows a stochastic gradient descent approach, which works in the four datasets we have taken, but may fail if you choose the wrong hyperparameters on problems with a more complex loss function. Appendix A (iv) has ANN's training set performance, which is slightly higher, but quite similar to its testing set performance, indicating it has not overfit the training data.

Finally, we have our best performing algorithm - Random Forest - which has the highest performance in both Tables 2 and 3. While this is the case for the above 4 datasets, we cannot necessarily generalize that Random Forest is a better algorithm than the other two. As you may see in Appendix A (i), (ii), and (v), Random Forest has a perfect training score of 1.0 under accuracy, F1 score, and ROC AUC throughout all datasets. We can see that the difference between training and testing score performance is the most drastic in Random Forest compared to any other algorithm. This is a strong indication of overfitting the model on training data which can cause poor performance over in the testing data for complex problems. This can also be seen in Table 3 where Random Forest does the worst on the Adult dataset compared to others, indicating that the model was the most overfit on the training data of Adult. Thus, even though Random Forest has performed the best out of the three algorithms, it is possible that using the same model (same hyperparameters) can cause poor performance on a different type of dataset due to the extreme amounts of overfitting of the training data. Hence, if the complexity of the problems increase, it is a better idea to tune the Random Forest model such that it is not prone to overfitting, or use a different model such as ANN which was not prone to overfitting.

Overall, the above results generally agree with the reference paper from Caruana, R., & Niculescu-Mizil, A. As we tested 3 different datasets from the reference paper, we have no way of comparing those results, however, comparing the testing set performance of the algorithms on the Adult dataset, we can observe that our three algorithms have similar performances. An extension we could add to this paper would be to find a dataset/problem

that causes a poorer performance on the Random Forest algorithm, analyze what factors cause it, and observe how other algorithms such as ANN and Logistic Regression perform on it.

Acknowledgement

I would like to dedicate this section to acknowledge all the help and encouragement the Cogs 118A instructors have provided every single day. They have helped me get through critical core concepts and conceptual questions without which this project could not have been completed. I would also like to thank the instructors for using Piazza as a collaborative platform to exchange questions and answers which benefits all students.

References

Accurate occupancy detection of an office room from light, temperature, humidity and CO2 measurements using statistical learning models. Luis M. Candanedo, VÃ©ronique Feldheim. Energy and Buildings. Volume 112, 15 January 2016, Pages 28-39.

Caruana, Rich., & Niculescu-Mizil, Alexandru (2006). "An Empirical Comparison of Supervised Learning Algorithms". Ithaca, NY: Cornell University, Department of Computer Science. Pages 1-8, www.cs.cornell.edu/~caruana/ctp/ct.papers/caruana.icml06.pdf.

Dua, D. "UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]." Irvine, CA: University of California, School of Information and Computer Science. n.p.: n.p., 12 . 18 Mar. 2021.

GeeksforGeeks. "Adding New Column to Existing DataFrame in Pandas." *GeeksforGeeks*, 12 Dec. 2018, www.geeksforgeeks.org/adding-new-column-to-existing-dataframe-in-pandas.

GeeksforGeeks. "Advantages and Disadvantages of Logistic Regression." *GeeksforGeeks*, 2 Sept. 2020, www.geeksforgeeks.org/advantages-and-disadvantages-of-logistic-regression.

GeeksforGeeks. "Select All Columns, except One given Column in a Pandas DataFrame." *GeeksforGeeks*, 20 Aug. 2020, www.geeksforgeeks.org/select-all-columns-except-one-given-column-in-a-pandas-dataframe.

Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.

Yadav, Dinesh. "Categorical Encoding Using Label-Encoding and One-Hot-Encoder." *Medium*, Towards Data Science, 9 Dec. 2019, towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd.

"Instructions for Formatting JMLR Articles." *Journal of Machine Learning Research*, www.jmlr.org/format/format.html. Accessed 18 Mar. 2021.

"Pandas.DataFrame.Append — Pandas 1.2.3 Documentation." *Pandas*, pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.append.html. Accessed 17 Mar. 2021.

"Pandas.DataFrame.Iloc — Pandas 1.2.3 Documentation." *Pandas*, pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html. Accessed 17 Mar. 2021.

"Pandas.DataFrame.Mean — Pandas 1.2.3 Documentation." *Pandas*, pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.mean.html. Accessed 17 Mar. 2021.

"Pandas.DataFrame.Rename — Pandas 1.2.3 Documentation." *Pandas*, pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html. Accessed 16 Mar. 2021.

"Pandas.Get_dummies — Pandas 1.2.3 Documentation." *Pandas*, pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html. Accessed 17 Mar. 2021.

"Pandas.Series.Idxmax — Pandas 1.2.3 Documentation." *Pandas*, pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.idxmax.html. Accessed 17 Mar. 2021.

"Scipy.Stats.Ttest_ind — SciPy v1.6.1 Reference Guide." *Scipy*, docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html. Accessed 17 Mar. 2021.

"Sklearn.Ensemble.RandomForestClassifier — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. Accessed 17 Mar. 2021.

"Sklearn.Linear_model.LogisticRegression — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression. Accessed 16 Mar. 2021.

"Sklearn.Metrics.Accuracy_score — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html. Accessed 17 Mar. 2021.

"Sklearn.Metrics.F1_score — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html. Accessed 17 Mar. 2021.

"Sklearn.Metrics.Roc_auc_score — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html. Accessed 17 Mar. 2021.

"Sklearn.Model_selection.GridSearchCV — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. Accessed 16 Mar. 2021.

"Sklearn.Model_selection.Train_test_split — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. Accessed 16 Mar. 2021.

"Sklearn.Neural_network.MLPClassifier — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html. Accessed 18 Mar. 2021.

"Sklearn.Preprocessing.StandardScaler — Scikit-Learn 0.24.1 Documentation." *Sklearn*, scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html. Accessed 17 Mar. 2021.

Appendix

Appendix A (i): Average training scores over datasets

Algorithms	Accuracy	F1 Score	ROC AUC	Mean
Logistic Regression	0.82014667	0.82158417	0.7853737	0.8090349
ANN	0.91171333	0.91102137	0.88408149	0.9022720
Random Forest	1.0	1.0	1.0	1.0

Appendix A (i): Average training performance of each algorithm across datasets like Table 2

Appendix A (ii): Average training scores over metrics

Algorithms	Adult	Gamma	Eye	Occupancy	Mean
Logistic Regression	0.8444225	0.7965953	0.6096383	0.9854833	0.8090349
ANN	0.8616511	0.897867	0.9020485	0.9475218	0.9022720
Random Forest	1.0	1.0	1.0	1.0	1.0

Appendix A (ii): Average training performance of each algorithm across metrics like Table 3

Appendix A (iii): Logistic Regression - Raw training scores

Algorithm	Dataset	Trial #	Accuracy	F1 Score	ROC AUC
Logistic Regression	Occupancy	1	0.990267	0.979296	0.992460
		2	0.990267	0.979308	0.992662
		3	0.987733	0.974048	0.990813
		4	0.988267	0.975028	0.989443
		5	0.987867	0.974294	0.990496
	Adult	1	0.855533	0.907524	0.771391
		2	0.850400	0.903955	0.767633
		3	0.853600	0.906202	0.769740
		4	0.853400	0.905877	0.771782
		5	0.859000	0.909348	0.780953
	Gamma	1	0.795533	0.850921	0.751458
		2	0.792400	0.848908	0.747220
		3	0.793800	0.849920	0.748776
		4	0.791400	0.848188	0.746101
		5	0.791133	0.848317	0.744854
	Eye	1	0.645000	0.554132	0.630745
		2	0.642200	0.546054	0.627087
		3	0.643933	0.558705	0.630771
		4	0.642533	0.547052	0.627528
		5	0.648667	0.564607	0.635561

Appendix A (iii): Raw training score performance for Logistic Regression

Appendix A (iv): ANN - Raw training scores

Algorithm	Dataset	Trial #	Accuracy	F1 Score	ROC AUC
ANN	Occupancy	1	0.990000	0.978749	0.992388
		2	0.987400	0.973286	0.989587
		3	0.988867	0.976342	0.990843
		4	0.987733	0.973974	0.989803
		5	0.915600	0.650945	0.827310
	Adult	1	0.875933	0.919882	0.808930
		2	0.855467	0.906605	0.781650
		3	0.862333	0.910947	0.792222
		4	0.875400	0.919513	0.808767
		5	0.877200	0.920815	0.809102
	Gamma	1	0.897267	0.923036	0.875083
		2	0.899733	0.925573	0.873600
		3	0.889867	0.917718	0.865688
		4	0.895333	0.921651	0.872508
		5	0.902733	0.927088	0.881121
	Eye	1	0.905333	0.893389	0.903351
		2	0.912533	0.900623	0.909840
		3	0.902600	0.889798	0.900112
		4	0.899600	0.887722	0.897943
		5	0.913333	0.902770	0.911780

Appendix A (iv): Raw training score performance for ANN

Appendix A (v): Random Forest - Raw training scores

Algorithm	Dataset	Trial #	Accuracy	F1 Score	ROC AUC
Random Forest	Occupancy	1	1.0	1.0	1.0
		2	1.0	1.0	1.0
		3	1.0	1.0	1.0
		4	1.0	1.0	1.0
		5	1.0	1.0	1.0
	Adult	1	1.0	1.0	1.0
		2	1.0	1.0	1.0
		3	1.0	1.0	1.0
		4	1.0	1.0	1.0
		5	1.0	1.0	1.0
	Gamma	1	1.0	1.0	1.0
		2	1.0	1.0	1.0
		3	1.0	1.0	1.0
		4	1.0	1.0	1.0
		5	1.0	1.0	1.0
	Eye	1	1.0	1.0	1.0
		2	1.0	1.0	1.0
		3	1.0	1.0	1.0
		4	1.0	1.0	1.0
		5	1.0	1.0	1.0

Appendix A (v): Raw training score performance for Random Forest

Appendix B

Appendix B (i): Logistic Regression - Raw testing scores

Algorithm	Dataset	Trial #	Accuracy	F1 Score	ROC AUC
Logistic Regression	Occupancy	1	0.988689	0.976009	0.991186
		2	0.988625	0.975871	0.991079
		3	0.989482	0.977648	0.991636
		4	0.988603	0.975829	0.991098
		5	0.989524	0.977749	0.991859
	Adult	1	0.848034	0.902607	0.762453
		2	0.847466	0.902146	0.762764
		3	0.849135	0.903437	0.762526
		4	0.847478	0.901594	0.769065
		5	0.848590	0.902752	0.765614
	Gamma	1	0.787756	0.845063	0.743462
		2	0.788160	0.846559	0.740417
		3	0.787660	0.844459	0.744890
		4	0.788469	0.847043	0.739990
		5	0.792867	0.849715	0.746337
	Eye	1	0.640381	0.554106	0.627143
		2	0.643487	0.549278	0.628675
		3	0.636807	0.549805	0.623556
		4	0.635438	0.537480	0.620254
		5	0.636874	0.547226	0.623133

Appendix B (i): Raw test score performance for Logistic Regression

Appendix B (ii): ANN - Raw testing scores

Algorithm	Dataset	Trial #	Accuracy	F1 Score	ROC AUC
ANN	Occupancy	1	0.989310	0.977261	0.991038
		2	0.989267	0.977159	0.990816
		3	0.988689	0.976035	0.991575
		4	0.989139	0.976950	0.991608
		5	0.915510	0.650810	0.827564
	Adult	1	0.842991	0.898827	0.761532
		2	0.841539	0.897946	0.758930
		3	0.840511	0.896399	0.766843
		4	0.843595	0.899297	0.761484
		5	0.844309	0.899849	0.761440
	Gamma	1	0.867000	0.901320	0.837497
		2	0.867118	0.901594	0.836846
		3	0.865383	0.900209	0.835383
		4	0.870233	0.902945	0.844585
		5	0.863742	0.897777	0.838775
	Eye	1	0.852705	0.833784	0.849976
		2	0.855177	0.837082	0.852778
		3	0.859185	0.839727	0.855722
		4	0.852772	0.834691	0.850375
		5	0.863560	0.845327	0.860514

Appendix B (ii): Raw test score performance for ANN

Appendix B (iii): Random Forest - Raw testing scores

Algorithm	Dataset	Trial #	Accuracy	F1 Score	ROC AUC
Random Forest	Occupancy	1	0.991003	0.980704	0.990517
		2	0.991302	0.981333	0.990680
		3	0.991624	0.981877	0.988294
		4	0.990510	0.979659	0.990035
		5	0.991517	0.981719	0.989554
	Adult	1	0.851094	0.903683	0.776796
		2	0.849860	0.902442	0.780082
		3	0.854505	0.905966	0.780569
		4	0.852993	0.905655	0.770760
		5	0.851844	0.904570	0.773209
	Gamma	1	0.869662	0.903544	0.839334
		2	0.872658	0.905300	0.844754
		3	0.868569	0.902460	0.839372
		4	0.870827	0.904279	0.841191
		5	0.873704	0.905410	0.848995
	Eye	1	0.894723	0.878918	0.890697
		2	0.910922	0.898612	0.908011
		3	0.895858	0.879761	0.891499
		4	0.897929	0.881761	0.893294
		5	0.901002	0.886896	0.897644

Appendix B (iii): Raw test score performance for Random Forest

Appendix C

Appendix C (i): p-value calculations for Table 2

Algorithm		p-values		
		Accuracy	F1 Score	ROC AUC
Logistic Regression	ANN	0.03444	0.09947	0.04686
	Random Forest	0.00819	0.01262	0.01032
ANN	Logistic Regression	0.03444	0.09947	0.04686
	Random Forest	0.29643	0.09588	0.36791
Random Forest	Logistic Regression	0.00819	0.01262	0.01032
	ANN	0.29643	0.09588	0.36791

Appendix C (i): p-values for each algorithms across datasets per metric for Table 2

Appendix C (ii): p-value calculations for Table 3

Algorithm		p-values			
		Adult	Gamma	Eye	Occupancy
Logistic Regression	ANN	0.85212	$3.77*10^{-6}$	$2.23*10^{-19}$	0.13134
	Random Forest	0.77807	$1.59*10^{-6}$	$2.64*10^{-21}$	0.42998
ANN	Logistic Regression	0.85212	$3.77*10^{-6}$	$2.23*10^{-19}$	0.13134
	Random Forest	0.63369	0.68169	$2.78*10^{-13}$	0.11498
Random Forest	Logistic Regression	0.77807	$1.59*10^{-6}$	$2.64*10^{-21}$	0.42998
	ANN	0.63369	0.68169	$2.78*10^{-13}$	0.11498

Appendix C (ii): p-values for each algorithms across metrics per dataset for Table 3

Appendix D: Code

Jupyter Notebook Code:

Sources

Data:

- https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+
- https://archive.ics.uci.edu/ml/datasets/MAGIC+Gamma+Telescope
- https://archive.ics.uci.edu/ml/datasets/EEG+Eye+State
- https://archive.ics.uci.edu/ml/datasets/Adult

Data Cleaning

We are going to use 4 datasets which we are going to refer to as Adult, Gamma, Eye, and Occupancy. All these datasets are taken from the UCI repository, and have been linked in the Sources section.

In [1]:
```
# importing appropriate packages
import pandas as pd
import numpy as np
import seaborn as sns
import random
from scipy.io import arff
import sklearn.model_selection
```

Adult Dataset

Let us first clean the Adult dataset:

In [2]:
```
column_header_adult = ['age', 'workclass', 'fnlwgt', 'education',
                       'education_num', 'marital_status', 'occupation',
                       'relationship', 'race', 'sex', 'capital_gain',
                       'capital_loss', 'hours/wk', 'native_country',
                       'salary']

adult = pd.read_csv('adult.data', names = column_header_adult)
print(adult.shape)
```

```
(32561, 15)
```

In [3]:
```
# Looking at the data
adult
```

Out[3]:

	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_los
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	
...	...	...	...	...	...	...	...	...	...	...	...	...
32556	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	
32557	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	
32558	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	
32559	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	
32560	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	

32561 rows × 15 columns

We are going to encode the ' <=50K' as positive class (+1), and the ' >50K' as -1 for the negative class

In [4]:
```
# Converting salary data into positive class and negative class for binary classification
salary2 = []
for i in adult['salary']:
    if i == ' <=50K':
        salary2.append(1)
    elif i == ' >50K':
        salary2.append(-1)

adult['salary'] = salary2
adult.head()
```

Out[4]:

	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_loss	h
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	

As we have a lot of categorical values, we are going to one-hot encode the data points using the get_dummies method in pandas

In [5]:
```
# Obtaining a one-hot encoding of all categorical values
adult_list_of_categorical_columns = ['workclass', 'education', 'marital_status',
                                     'occupation', 'relationship', 'race', 'sex',
                                     'native_country']
adult = pd.get_dummies(adult, prefix = adult_list_of_categorical_columns)
adult.head()
```

Out[5]:

	age	fnlwgt	education_num	capital_gain	capital_loss	hours/wk	salary	workclass_?	workclass_Federal-gov	workclass_Local-gov	...	native_country_Portugal	nativ Pu
0	39	77516	13	2174	0	40	1	0	0	0	...	0	
1	50	83311	13	0	0	13	1	0	0	0	...	0	
2	38	215646	9	0	0	40	1	0	0	0	...	0	
3	53	234721	7	0	0	40	1	0	0	0	...	0	
4	28	338409	13	0	0	40	1	0	0	0	...	0	

5 rows × 109 columns

In [6]:
```
# Saving cleaned dataframe to csv
adult.to_csv('adult_cleaned.csv', index = False)
```

Gamma Dataset

Cleaning the Gamma dataset:

In [7]:
```
gamma_columns = ['fLength', 'fWidth', 'fSize', 'fConc', 'fConc1',
                 'fAsym', 'fM3Long', 'fM3Trans', 'fAlpha', 'fDist', 'class']
gamma = pd.read_csv('magic04.data', names = gamma_columns)
print(gamma.shape)
```

```
(19020, 11)
```

In [8]:
```
# Lookg at the dataset
gamma
```

Out[8]:

	fLength	fWidth	fSize	fConc	fConc1	fAsym	fM3Long	fM3Trans	fAlpha	fDist	class
0	28.7967	16.0021	2.6449	0.3918	0.1982	27.7004	22.0110	-8.2027	40.0920	81.8828	g
1	31.6036	11.7235	2.5185	0.5303	0.3773	26.2722	23.8238	-9.9574	6.3609	205.2610	g
2	162.0520	136.0310	4.0612	0.0374	0.0187	116.7410	-64.8580	-45.2160	76.9600	256.7880	g
3	23.8172	9.5728	2.3385	0.6147	0.3922	27.2107	-6.4633	-7.1513	10.4490	116.7370	g
4	75.1362	30.9205	3.1611	0.3168	0.1832	-5.5277	28.5525	21.8393	4.6480	356.4620	g
...	...	...	...	...	...	...	...	...	...	...	...
19015	21.3846	10.9170	2.6161	0.5857	0.3934	15.2618	11.5245	2.8766	2.4229	106.8258	h
19016	28.9452	6.7020	2.2672	0.5351	0.2784	37.0816	13.1853	-2.9632	86.7975	247.4560	h
19017	75.4455	47.5305	3.4483	0.1417	0.0549	-9.3561	41.0562	-9.4662	30.2987	256.5166	h
19018	120.5135	76.9018	3.9939	0.0944	0.0683	5.8043	-93.5224	-63.8389	84.6874	408.3166	h
19019	187.1814	53.0014	3.2093	0.2876	0.1539	-167.3125	-168.4558	31.4755	52.7310	272.3174	h

19020 rows × 11 columns

We are going to encode the 'g' as positive class (+1), and the 'h' as -1 for the negative class

In [9]:
```
# Converting cateogrical class data into positive class and negative class for binary classification
class2 = []
for i in gamma['class']:
    if i == 'g':
        class2.append(1)
    elif i == 'h':
        class2.append(-1)

gamma['class'] = class2
gamma.head()
```

Out[9]:

	fLength	fWidth	fSize	fConc	fConc1	fAsym	fM3Long	fM3Trans	fAlpha	fDist	class
0	28.7967	16.0021	2.6449	0.3918	0.1982	27.7004	22.0110	-8.2027	40.0920	81.8828	1
1	31.6036	11.7235	2.5185	0.5303	0.3773	26.2722	23.8238	-9.9574	6.3609	205.2610	1
2	162.0520	136.0310	4.0612	0.0374	0.0187	116.7410	-64.8580	-45.2160	76.9600	256.7880	1
3	23.8172	9.5728	2.3385	0.6147	0.3922	27.2107	-6.4633	-7.1513	10.4490	116.7370	1
4	75.1362	30.9205	3.1611	0.3168	0.1832	-5.5277	28.5525	21.8393	4.6480	356.4620	1

In [10]:
```
# Saving cleaned dataframe to csv
gamma.to_csv('gamma_cleaned.csv', index = False)
```

Eye Dataset

Cleaning the Eye dataset:

In [11]:
```
eye = arff.loadarff('EEG_Eye_State.arff')
eye_df = pd.DataFrame(eye[0])
```

We are going to encode the '1' as positive class, and the '0' as -1 for the negative class.

Note: Here, the 1 and 0 are in byte format, which we encode as integers

In [12]:
```
# Converting cateogrical class data into positive class and negative class for binary classification
eye_detection2 = []
for i in eye_df['eyeDetection']:
    if i == b'1':
        eye_detection2.append(1)
    elif i == b'0':
        eye_detection2.append(-1)
eye_df['eyeDetection'] = eye_detection2
```

In [13]:
```
eye_df.head()
```

Out[13]:

	AF3	F7	F3	FC5	T7	P7	O1	O2	P8	T8	FC6	F4	F8	AF4	eyeDe
0	4329.23	4009.23	4289.23	4148.21	4350.26	4586.15	4096.92	4641.03	4222.05	4238.46	4211.28	4280.51	4635.90	4393.85	
1	4324.62	4004.62	4293.85	4148.72	4342.05	4586.67	4097.44	4638.97	4210.77	4226.67	4207.69	4279.49	4632.82	4384.10	
2	4327.69	4006.67	4295.38	4156.41	4336.92	4583.59	4096.92	4630.26	4207.69	4222.05	4206.67	4282.05	4628.72	4389.23	
3	4328.72	4011.79	4296.41	4155.90	4343.59	4582.56	4097.44	4630.77	4217.44	4235.38	4210.77	4287.69	4632.31	4396.41	
4	4326.15	4011.79	4292.31	4151.28	4347.69	4586.67	4095.90	4627.69	4210.77	4244.10	4212.82	4288.21	4632.82	4398.46	

In [14]:
```
# Saving cleaned eye_df dataframe to csv
eye_df.to_csv('eye_cleaned.csv', index = False)
```

Occupancy Dataset

Finally, we use the Occupancy dataset. The data given to us already has the overall data split into three groups: training data, testing data 1, testing data 2. As we want our training size to be different from their training size, and we want to randomly select this each time we perform a trial, we are going to group all the 3 individual datasets into one big dataset we will call 'occupany_df'

In [15]:
```
occupancy_training = pd.read_csv('occupancy_data/datatraining.txt')
occupancy_test1 = pd.read_csv('occupancy_data/datatest.txt')
occupancy_test2 = pd.read_csv('occupancy_data/datatest2.txt')

print(occupancy_training.shape, occupancy_test1.shape, occupancy_test2.shape)
```

```
(8143, 7) (2665, 7) (9752, 7)
```

In [16]:
```
# Combining the datasets
occupancy_test = occupancy_test1.append(occupancy_test2)
occupancy_df = occupancy_training.append(occupancy_test).reset_index()
```

In [17]:
```
occupancy_df
```

Out[17]:

	index	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
0	1	2015-02-04 17:51:00	23.180	27.2720	426.00	721.25	0.004793	1
1	2	2015-02-04 17:51:59	23.150	27.2675	429.50	714.00	0.004783	1
2	3	2015-02-04 17:53:00	23.150	27.2450	426.00	713.50	0.004779	1
3	4	2015-02-04 17:54:00	23.150	27.2000	426.00	708.25	0.004772	1
4	5	2015-02-04 17:55:00	23.100	27.2000	426.00	704.50	0.004757	1
...	...	...	...	...	...	...	...	...
20555	9748	2015-02-18 09:15:00	20.815	27.7175	429.75	1505.25	0.004213	1
20556	9749	2015-02-18 09:16:00	20.865	27.7450	423.50	1514.50	0.004230	1
20557	9750	2015-02-18 09:16:59	20.890	27.7450	423.50	1521.50	0.004237	1
20558	9751	2015-02-18 09:17:59	20.890	28.0225	418.75	1632.00	0.004279	1
20559	9752	2015-02-18 09:19:00	21.000	28.1000	409.00	1864.00	0.004321	1

20560 rows × 8 columns

For consistency, we are going to encode the '1' as positive class, and the '0' as -1 for the negative class

In [18]:
```
# Converting cateogrical class data into positive class and negative class for binary classification
occupancy2 = []
for i in occupancy_df['Occupancy']:
    if i == 1:
        occupancy2.append(1)
    elif i == 0:
        occupancy2.append(-1)

occupancy_df['Occupancy'] = occupancy2
occupancy_df.head()
```

Out[18]:

	index	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
0	1	2015-02-04 17:51:00	23.18	27.2720	426.0	721.25	0.004793	1
1	2	2015-02-04 17:51:59	23.15	27.2675	429.5	714.00	0.004783	1
2	3	2015-02-04 17:53:00	23.15	27.2450	426.0	713.50	0.004779	1
3	4	2015-02-04 17:54:00	23.15	27.2000	426.0	708.25	0.004772	1
4	5	2015-02-04 17:55:00	23.10	27.2000	426.0	704.50	0.004757	1

We don't require the date column as it is not a feature of our dataset useful for predicting occupancy

In [19]:
```
occupancy_df = occupancy_df.drop(['date', 'index'], axis = 1)
occupancy_df
```

Out[19]:

	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
0	23.180	27.2720	426.00	721.25	0.004793	1
1	23.150	27.2675	429.50	714.00	0.004783	1
2	23.150	27.2450	426.00	713.50	0.004779	1
3	23.150	27.2000	426.00	708.25	0.004772	1
4	23.100	27.2000	426.00	704.50	0.004757	1
...	...	...	...	...	...	...
20555	20.815	27.7175	429.75	1505.25	0.004213	1
20556	20.865	27.7450	423.50	1514.50	0.004230	1
20557	20.890	27.7450	423.50	1521.50	0.004237	1
20558	20.890	28.0225	418.75	1632.00	0.004279	1
20559	21.000	28.1000	409.00	1864.00	0.004321	1

20560 rows × 6 columns

In [20]:
```
# Saving cleaned occupancy_df dataframe to csv
occupancy_df.to_csv('occupancy_cleaned.csv', index = False)
```

Logistic Regression

In [1]:
```
import pandas as pd
import numpy as np
from scipy.io import arff
import sklearn.model_selection
import sklearn.preprocessing
import sklearn.linear_model
import sklearn.metrics
import scipy.stats
```

Importing the 'Adult' dataset

In [2]:
```
adult = pd.read_csv('adult_cleaned.csv')
adult.head()
```

Out[2]:

	age	fnlwgt	education_num	capital_gain	capital_loss	hours/wk	salary	workclass_ ?	workclass_ Federal-gov	workclass_ Local-gov	...	native_country_ Portugal	nativ Pu
0	39	77516	13	2174	0	40	1	0	0	0	...	0	
1	50	83311	13	0	0	13	1	0	0	0	...	0	
2	38	215646	9	0	0	40	1	0	0	0	...	0	
3	53	234721	7	0	0	40	1	0	0	0	...	0	
4	28	338409	13	0	0	40	1	0	0	0	...	0	

5 rows × 109 columns

In [3]:
```
# Converting pandas series to numpy arrays
adult_Y = adult['salary'].to_numpy()

adult_X = adult.iloc[:, adult.columns != 'salary'].to_numpy()
#adult_X
```

In [4]:
```
# Scaling the data using a Standard scaler
std_scaler = sklearn.preprocessing.StandardScaler()
```

In [5]:
```
adult_X_std = std_scaler.fit_transform(adult_X)
```

Using the 'Gamma' dataset now

In [6]:
```
gamma = pd.read_csv('gamma_cleaned.csv')
gamma.head()
```

Out[6]:

	fLength	fWidth	fSize	fConc	fConc1	fAsym	fM3Long	fM3Trans	fAlpha	fDist	class
0	28.7967	16.0021	2.6449	0.3918	0.1982	27.7004	22.0110	-8.2027	40.0920	81.8828	1
1	31.6036	11.7235	2.5185	0.5303	0.3773	26.2722	23.8238	-9.9574	6.3609	205.2610	1
2	162.0520	136.0310	4.0612	0.0374	0.0187	116.7410	-64.8580	-45.2160	76.9600	256.7880	1
3	23.8172	9.5728	2.3385	0.6147	0.3922	27.2107	-6.4633	-7.1513	10.4490	116.7370	1
4	75.1362	30.9205	3.1611	0.3168	0.1832	-5.5277	28.5525	21.8393	4.6480	356.4620	1

In [7]:
```
# Converting pandas series to numpy arrays
gamma_Y = gamma['class'].to_numpy()

gamma_X = gamma.iloc[:, gamma.columns != 'class'].to_numpy()
```

In [8]:
```
# Scaling the gamma data using a Standard scaler
gamma_X_std = std_scaler.fit_transform(gamma_X)
```

Importing the cleaned 'Eye' data:

In [9]:
```
eye_df = pd.read_csv('eye_cleaned.csv')
eye_df.head()
```

Out[9]:

	AF3	F7	F3	FC5	T7	P7	O1	O2	P8	T8	FC6	F4	F8	AF4	eyeDe
0	4329.23	4009.23	4289.23	4148.21	4350.26	4586.15	4096.92	4641.03	4222.05	4238.46	4211.28	4280.51	4635.90	4393.85	
1	4324.62	4004.62	4293.85	4148.72	4342.05	4586.67	4097.44	4638.97	4210.77	4226.67	4207.69	4279.49	4632.82	4384.10	
2	4327.69	4006.67	4295.38	4156.41	4336.92	4583.59	4096.92	4630.26	4207.69	4222.05	4206.67	4282.05	4628.72	4389.23	
3	4328.72	4011.79	4296.41	4155.90	4343.59	4582.56	4097.44	4630.77	4217.44	4235.38	4210.77	4287.69	4632.31	4396.41	
4	4326.15	4011.79	4292.31	4151.28	4347.69	4586.67	4095.90	4627.69	4210.77	4244.10	4212.82	4288.21	4632.82	4398.46	

In [10]:
```
# Converting pandas series to numpy arrays
eye_Y = eye_df['eyeDetection'].to_numpy()
eye_X = eye_df.iloc[:, eye_df.columns != 'eyeDetection'].to_numpy()
```

In [11]:
```
# Scaling the eye data arrays using a Standard scaler
eye_X_std = std_scaler.fit_transform(eye_X)
```

Importing the cleaned 'Occupancy' dataset

In [12]:
```
occupancy_df = pd.read_csv('occupancy_cleaned.csv')
occupancy_df.head()
```

Out[12]:

	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
0	23.18	27.2720	426.0	721.25	0.004793	1
1	23.15	27.2675	429.5	714.00	0.004783	1
2	23.15	27.2450	426.0	713.50	0.004779	1
3	23.15	27.2000	426.0	708.25	0.004772	1
4	23.10	27.2000	426.0	704.50	0.004757	1

In [13]:
```
# Converting pandas series to numpy arrays
occupancy_Y = occupancy_df['Occupancy'].to_numpy()
occupancy_X = occupancy_df.iloc[:, occupancy_df.columns != 'Occupancy'].to_numpy()
```

In [14]:
```
# Scaling the occupancy dataset using a Standard scaler
occupancy_X_std = std_scaler.fit_transform(occupancy_X)
```

Running 5 trials of logistic regression on each dataset:

In [15]:
```
# List of all hyper parameters - for logistic regression with regularization
log_ridge_params = [1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e+0, 1e+1, 1e+2, 1e+3, 1e+4]
```

In [16]:
```
# This function can be called on a given dataset and it will run
# Logistic Regression classification on it
def logistic_regression_solver(X_data, Y_data):

    # These lists will store the training and testing
    # performance metrics across all 5 trials
    trial_scores_test = []
    trial_scores_train = []

    for i in range(5):
        # Splitting data into training (5000 samples) and testing (remaining data points)
        # Note: we are shuffling the data for randomness

        X_train, data_X_test, Y_train, data_Y_test = sklearn.model_selection.train_test_split(
            X_data, Y_data, train_size = 5000, shuffle = True, stratify = Y_data)

        # Creating a LogisticRegression Classifier model with some parameters (that will change below)
        log_mod_reg = sklearn.linear_model.LogisticRegression(solver = 'lbfgs',
                                                              penalty = 'l2', max_iter = 1000)

        # GridSearch object to cycle through hyperparameters
        data_grid_search = sklearn.model_selection.GridSearchCV(estimator = log_mod_reg,
                                                                 cv = 5, #for 5 fold cross validation
                                                                 param_grid = [
                                                                     {'C': log_ridge_params,
                                                                      'penalty':['l2'],
                                                                      'solver':['lbfgs']},

                                                                     {'penalty':['none'],
                                                                      'solver':['lbfgs']}
                                                                 ],
                                                                 scoring = ['accuracy', 'f1', 'roc_auc'],
                                                                 refit = 'accuracy',
                                                                     n_jobs = -1)
```

```
# Fitting the training data to do 5 fold cross validation
data_grid_search.fit(X_train, Y_train)
data_grid_results = pd.DataFrame(data_grid_search.cv_results_)

# Obtaining the dataframe indicies corresponding to the best accuracy, f1, and roc_auc
data_ind_best_accuracy = data_grid_results['mean_test_accuracy'].idxmax()
data_ind_best_roc_auc = data_grid_results['mean_test_roc_auc'].idxmax()
data_ind_best_f1 = data_grid_results['mean_test_f1'].idxmax()

# Obtaining the corresponding hyperparameters from the index value above
data_log_accuracy_best_param_C = data_grid_results['param_C'][data_ind_best_accuracy]
data_log_accuracy_best_param_penalty = data_grid_results['param_penalty'][data_ind_best_accuracy]

data_log_f1_best_param_C = data_grid_results['param_C'][data_ind_best_f1]
data_log_f1_best_param_penalty = data_grid_results['param_penalty'][data_ind_best_f1]

data_log_roc_auc_best_param_C = data_grid_results['param_C'][data_ind_best_roc_auc]
data_log_roc_auc_best_param_penalty = data_grid_results['param_penalty'][data_ind_best_roc_auc]

# Creating a list of best hyperparams per metric
list_best_param_accuracy = [data_log_accuracy_best_param_C,
                            data_log_accuracy_best_param_penalty]

list_best_param_f1 = [data_log_f1_best_param_C,
                      data_log_f1_best_param_penalty]

list_best_param_roc_auc = [data_log_roc_auc_best_param_C,
                           data_log_roc_auc_best_param_penalty]
```

```
# This is used to create a dataframe to cycle through the parameters while retraining
overall_best_params = [list_best_param_accuracy, list_best_param_f1, list_best_param_roc_auc]

# The row indicies are ['accuracy', 'f1', 'roc_auc']
best_hyperparam = pd.DataFrame(overall_best_params, columns = ['C', 'penalty'])

# These training and testing lists store the performance of each
# model (3 models as there are 3 metrics)
model_scores_test = []
model_scores_train = []

for i in range(len(best_hyperparam)):

    # Stores the training and testing scores of a model in the loop
    test_scores = []
    train_scores = []

    if (best_hyperparam['penalty'][i] == 'none'):
        log_mod_reg_final = sklearn.linear_model.LogisticRegression(penalty = 'none',
                                                                    solver = 'lbfgs',
                                                                    max_iter = 1000)

    else:
        log_mod_reg_final = sklearn.linear_model.LogisticRegression(penalty = 'l2',
                                                                    solver = 'lbfgs',
                                                                    C = best_hyperparam['C'][i],
                                                                    max_iter = 1000)

    # Fitting the corresponding model once more with the whole training data
    log_mod_reg_final.fit(X_train, Y_train)

    # Test data performance
    data_Y_pred = log_mod_reg_final.predict(data_X_test)

    test_scores.append(sklearn.metrics.accuracy_score(data_Y_test, data_Y_pred))
    test_scores.append(sklearn.metrics.f1_score(data_Y_test, data_Y_pred))
    test_scores.append(sklearn.metrics.roc_auc_score(data_Y_test, data_Y_pred))

    model_scores_test.append(test_scores)

    # Training data performance
    data_Y_pred_train = log_mod_reg_final.predict(X_train)

    train_scores.append(sklearn.metrics.accuracy_score(Y_train, data_Y_pred_train))
    train_scores.append(sklearn.metrics.f1_score(Y_train, data_Y_pred_train))
    train_scores.append(sklearn.metrics.roc_auc_score(Y_train, data_Y_pred_train))

    model_scores_train.append(train_scores)

# Storing the performance of each testing data model
model_df_test = pd.DataFrame(model_scores_test, columns = ['accuracy', 'f1', 'roc_auc'])
model_mean_test = model_df_test.mean().to_numpy()

# Storing the performance of each training data model
```

```
        model_df_train = pd.DataFrame(model_scores_train, columns = ['accuracy', 'f1', 'roc_auc'])
        model_mean_train = model_df_train.mean().to_numpy()

        trial_scores_test.append(model_mean_test)
        trial_scores_train.append(model_mean_train)

    # These datasets contain all the 3 metrics performances of each trial in a dataframe format
    log_data_trial_test = pd.DataFrame(trial_scores_test, columns = ['accuracy', 'f1', 'roc_auc'])
    log_data_trial_train = pd.DataFrame(trial_scores_train, columns = ['accuracy', 'f1', 'roc_auc'])

    return log_data_trial_test, log_data_trial_train
```

Now we can call the above Logistic Regression function on each dataset to obtain its testing and training data performances

In [17]:
```
raw_test_adult, raw_train_adult = logistic_regression_solver(adult_X_std, adult_Y)
```

In [18]:
```
raw_train_adult
```

Out[18]:

	accuracy	f1	roc_auc
0	0.855533	0.907524	0.771391
1	0.850400	0.903955	0.767633
2	0.853600	0.906202	0.769740
3	0.853400	0.905877	0.771782
4	0.859000	0.909348	0.780953

In [19]:
```
raw_test_gamma, raw_train_gamma = logistic_regression_solver(gamma_X_std, gamma_Y)
raw_test_gamma
```

Out[19]:

	accuracy	f1	roc_auc
0	0.787756	0.845063	0.743462
1	0.788160	0.846559	0.740417
2	0.787660	0.844459	0.744890
3	0.788469	0.847043	0.739990
4	0.792867	0.849715	0.746337

In [20]:
```
raw_test_eye, raw_train_eye = logistic_regression_solver(eye_X_std, eye_Y)
raw_test_eye
```

Out[20]:

	accuracy	f1	roc_auc
0	0.640381	0.554106	0.627143
1	0.643487	0.549278	0.628675
2	0.636807	0.549805	0.623556
3	0.635438	0.537480	0.620254
4	0.636874	0.547226	0.623133

In [21]:
```
raw_test_occupancy, raw_train_occupancy = logistic_regression_solver(occupancy_X_std, occupancy_Y)
raw_test_occupancy
```

Out[21]:

	accuracy	f1	roc_auc
0	0.988689	0.976009	0.991186
1	0.988625	0.975871	0.991079
2	0.989482	0.977648	0.991636
3	0.988603	0.975829	0.991098
4	0.989524	0.977749	0.991859

In [22]:
```
# Obtaining the mean across all 5 trials per dataset for the testing data

mean_adult_log_score = raw_test_adult.mean().to_numpy()
mean_gamma_log_score = raw_test_gamma.mean().to_numpy()
mean_eye_log_score = raw_test_eye.mean().to_numpy()
mean_occupancy_log_score = raw_test_occupancy.mean().to_numpy()
```

In [23]:
```
# Obtaining the mean across all 5 trials per dataset for the training data

mean_adult_log_score_train = raw_train_adult.mean().to_numpy()
mean_gamma_log_score_train = raw_train_gamma.mean().to_numpy()
mean_eye_log_score_train = raw_train_eye.mean().to_numpy()
mean_occupancy_log_score_train = raw_train_occupancy.mean().to_numpy()
```

In [24]:
```
# Saving the average metrics for both testing and training so we don't have to rerun the code

list_of_metrics_across_datasets = [mean_adult_log_score, mean_gamma_log_score,
                                   mean_eye_log_score, mean_occupancy_log_score]

list_of_metrics_across_datasets_train = [mean_adult_log_score_train, mean_gamma_log_score_train,
                                   mean_eye_log_score_train, mean_occupancy_log_score_train]
```

In [25]:
```
pd.DataFrame(list_of_metrics_across_datasets).to_csv('Log_Reg_metrics.csv', index = False)
pd.DataFrame(list_of_metrics_across_datasets_train).to_csv('Log_Reg_metrics_train.csv', index = False)
```

In [27]:
```
# Also saving the raw test scores for later use if needed
raw_test_occupancy.to_csv('Log_occupancy_test.csv', index = False)
raw_test_adult.to_csv('Log_adult_test.csv', index = False)
raw_test_gamma.to_csv('Log_gamma_test.csv', index = False)
raw_test_eye.to_csv('Log_eye_test.csv', index = False)
```

In [28]:
```
# saving the raw train scores for later use if needed
raw_train_occupancy.to_csv('Log_occupancy_train.csv', index = False)
raw_train_adult.to_csv('Log_adult_train.csv', index = False)
raw_train_gamma.to_csv('Log_gamma_train.csv', index = False)
raw_train_eye.to_csv('Log_eye_train.csv', index = False)
```

Random Forest

In [1]:
```
import pandas as pd
import numpy as np
from scipy.io import arff
import sklearn.model_selection
import sklearn.preprocessing
import sklearn.metrics
import sklearn.ensemble
```

Importing the 'Adult' dataset

In [2]:
```
adult = pd.read_csv('adult_cleaned.csv')
adult.head()
```

Out[2]:

	age	fnlwgt	education_num	capital_gain	capital_loss	hours/wk	salary	workclass_ ?	workclass_ Federal-gov	workclass_ Local-gov	...	native_country_ Portugal	nativ Pu
0	39	77516	13	2174	0	40	1	0	0	0	...	0	
1	50	83311	13	0	0	13	1	0	0	0	...	0	
2	38	215646	9	0	0	40	1	0	0	0	...	0	
3	53	234721	7	0	0	40	1	0	0	0	...	0	
4	28	338409	13	0	0	40	1	0	0	0	...	0	

5 rows × 109 columns

In [3]:
```
# Converting pandas series to numpy arrays
adult_Y = adult['salary'].to_numpy()

adult_X = adult.iloc[:, adult.columns != 'salary'].to_numpy()
#adult_X
```

In [4]:
```
# Scaling the data using a Standard scaler
std_scaler = sklearn.preprocessing.StandardScaler()
```

In [5]:
```
adult_X_std = std_scaler.fit_transform(adult_X)
```

Using the 'Gamma' dataset now

In [6]:
```
gamma = pd.read_csv('gamma_cleaned.csv')
gamma.head()
```

Out[6]:

	fLength	fWidth	fSize	fConc	fConc1	fAsym	fM3Long	fM3Trans	fAlpha	fDist	class
0	28.7967	16.0021	2.6449	0.3918	0.1982	27.7004	22.0110	-8.2027	40.0920	81.8828	1
1	31.6036	11.7235	2.5185	0.5303	0.3773	26.2722	23.8238	-9.9574	6.3609	205.2610	1
2	162.0520	136.0310	4.0612	0.0374	0.0187	116.7410	-64.8580	-45.2160	76.9600	256.7880	1
3	23.8172	9.5728	2.3385	0.6147	0.3922	27.2107	-6.4633	-7.1513	10.4490	116.7370	1
4	75.1362	30.9205	3.1611	0.3168	0.1832	-5.5277	28.5525	21.8393	4.6480	356.4620	1

In [7]:
```
# Converting pandas series to numpy arrays
gamma_Y = gamma['class'].to_numpy()

gamma_X = gamma.iloc[:, gamma.columns != 'class'].to_numpy()
```

In [8]:
```
# Scaling the gamma data using a Standard scaler
gamma_X_std = std_scaler.fit_transform(gamma_X)
```

Using the 'Eye' dataset now

In [9]:
```
eye_df = pd.read_csv('eye_cleaned.csv')
eye_df.head()
```

Out[9]:

	AF3	F7	F3	FC5	T7	P7	O1	O2	P8	T8	FC6	F4	F8	AF4	eyeDe
0	4329.23	4009.23	4289.23	4148.21	4350.26	4586.15	4096.92	4641.03	4222.05	4238.46	4211.28	4280.51	4635.90	4393.85	
1	4324.62	4004.62	4293.85	4148.72	4342.05	4586.67	4097.44	4638.97	4210.77	4226.67	4207.69	4279.49	4632.82	4384.10	
2	4327.69	4006.67	4295.38	4156.41	4336.92	4583.59	4096.92	4630.26	4207.69	4222.05	4206.67	4282.05	4628.72	4389.23	
3	4328.72	4011.79	4296.41	4155.90	4343.59	4582.56	4097.44	4630.77	4217.44	4235.38	4210.77	4287.69	4632.31	4396.41	
4	4326.15	4011.79	4292.31	4151.28	4347.69	4586.67	4095.90	4627.69	4210.77	4244.10	4212.82	4288.21	4632.82	4398.46	

In [10]:
```
# Converting pandas series to numpy arrays
eye_Y = eye_df['eyeDetection'].to_numpy()
eye_X = eye_df.iloc[:, eye_df.columns != 'eyeDetection'].to_numpy()
```

In [11]:
```
# Scaling the eye data arrays using a Standard scaler
eye_X_std = std_scaler.fit_transform(eye_X)
```

Using the 'Occupancy' dataset now:

In [12]:
```
occupancy_df = pd.read_csv('occupancy_cleaned.csv')
occupancy_df.head()
```

Out[12]:

	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
0	23.18	27.2720	426.0	721.25	0.004793	1
1	23.15	27.2675	429.5	714.00	0.004783	1
2	23.15	27.2450	426.0	713.50	0.004779	1
3	23.15	27.2000	426.0	708.25	0.004772	1
4	23.10	27.2000	426.0	704.50	0.004757	1

In [13]:
```
# Converting pandas series to numpy arrays
occupancy_Y = occupancy_df['Occupancy'].to_numpy()
occupancy_X = occupancy_df.iloc[:, occupancy_df.columns != 'Occupancy'].to_numpy()
```

In [14]:
```
occupancy_X_std = std_scaler.fit_transform(occupancy_X)
```

Running 5 trials of Random Forest on each dataset:

In [15]:
```
# List of all hyper parameters
max_features_split = [1, 2, 4, 6, 8, 12, 16, 20]
```

In [16]:
```
# This function can be called on a given dataset and it will run
# Random Forest classification on it
def random_forest_solver(X_data, Y_data):

    # These lists will store the training and testing
    # performance metrics across all 5 trials
    trial_scores_test = []
    trial_scores_train = []

    for i in range(5):
        # Splitting data into training (5000 samples) and testing (remaining data points)
        # Note: we are shuffling the data before sampling for randomness

        X_train, data_X_test, Y_train, data_Y_test = sklearn.model_selection.train_test_split(
            X_data, Y_data, train_size = 5000, shuffle = True, stratify = Y_data)

        # Creating a Random Forest Classifier object with 1024 trees
        rf_classifier = sklearn.ensemble.RandomForestClassifier(n_estimators = 1024)

        # This list will contain the final max_features_split hyperparameters
        # based on the data inputted
        new_list_max_features = []
        max_vars_in_data = X_train.shape[1]

        # If the length of the data is smaller than a number in
        # max_features_split, it will append values smaller than it
        # into the list new_list_max_features
        for i in max_features_split:
            if i <= max_vars_in_data:
                new_list_max_features.append(i)

        # GridSearch object to cycle through hyperparameters along with cross validation
        data_grid_search = sklearn.model_selection.GridSearchCV(estimator = rf_classifier,
                                                                 cv = 5,
                                                                 param_grid = {'max_features': new_list_max_features},
                                                                 scoring = ['accuracy', 'f1', 'roc_auc'],
```

```
                                                    refit = 'accuracy',
                                                         n_jobs = -1)

    # Fitting the training data to perform a 5 fold cross validation
    data_grid_search.fit(X_train, Y_train)
    data_grid_results = pd.DataFrame(data_grid_search.cv_results_)

    # Obtaining the dataframe indicies corresponding to the best accuracy, f1, and roc_auc
    data_ind_best_accuracy = data_grid_results['mean_test_accuracy'].idxmax()
    data_ind_best_roc_auc = data_grid_results['mean_test_roc_auc'].idxmax()
    data_ind_best_f1 = data_grid_results['mean_test_f1'].idxmax()

    # Obtaining the corresponding hyperparameters from the index value above
    data_accuracy_best_param_features = data_grid_results['param_max_features'][data_ind_best_accuracy]
    data_f1_best_param_features = data_grid_results['param_max_features'][data_ind_best_f1]
    data_roc_auc_best_param_features = data_grid_results['param_max_features'][data_ind_best_roc_auc]

    # Creating a list of best hyperparams per metric
    list_best_param_accuracy = [data_accuracy_best_param_features]
    list_best_param_f1 = [data_f1_best_param_features]
    list_best_param_roc_auc = [data_roc_auc_best_param_features]

    # This is used to create a dataframe to cycle through the parameters while re-training
    overall_best_params = [list_best_param_accuracy, list_best_param_f1, list_best_param_roc_auc]

    # The row indicies are ['accuracy', 'f1', 'roc_auc']
    best_hyperparam = pd.DataFrame(overall_best_params, columns = ['max_features'])
```

```
    # These training and testing lists store the performance of each
    # model (3 models as there are 3 metrics)
    model_scores_test = []
    model_scores_train = []

    for i in range(len(best_hyperparam)):

        # Stores the training and testing scores of a model in the loop
        test_scores = []
        train_scores = []

        # Creating model with the best hyperparameter
        rf_final = sklearn.ensemble.RandomForestClassifier(n_estimators = 1024,
                                                           max_features = best_hyperparam['max_featur
```

```
        # Fitting the corresponding model once more with the whole training data
        rf_final.fit(X_train, Y_train)

        # Test data performance
        data_Y_pred = rf_final.predict(data_X_test)

        test_scores.append(sklearn.metrics.accuracy_score(data_Y_test, data_Y_pred))
        test_scores.append(sklearn.metrics.f1_score(data_Y_test, data_Y_pred))
        test_scores.append(sklearn.metrics.roc_auc_score(data_Y_test, data_Y_pred))

        model_scores_test.append(test_scores)

        # Training data performance
        data_Y_pred_train = rf_final.predict(X_train)

        train_scores.append(sklearn.metrics.accuracy_score(Y_train, data_Y_pred_train))
        train_scores.append(sklearn.metrics.f1_score(Y_train, data_Y_pred_train))
        train_scores.append(sklearn.metrics.roc_auc_score(Y_train, data_Y_pred_train))

        model_scores_train.append(train_scores)

    # Storing the performance of each testing data model
    model_df_test = pd.DataFrame(model_scores_test, columns = ['accuracy', 'f1', 'roc_auc'])
    model_mean_test = model_df_test.mean().to_numpy()

    # Storing the performance of each training data model
    model_df_train = pd.DataFrame(model_scores_train, columns = ['accuracy', 'f1', 'roc_auc'])
    model_mean_train = model_df_train.mean().to_numpy()

    trial_scores_test.append(model_mean_test)
    trial_scores_train.append(model_mean_train)

# These datasets contain all the 3 metrics performances of each trial in a dataframe format
RF_data_trial_test = pd.DataFrame(trial_scores_test, columns = ['accuracy', 'f1', 'roc_auc'])
RF_data_trial_train = pd.DataFrame(trial_scores_train, columns = ['accuracy', 'f1', 'roc_auc'])

return RF_data_trial_test, RF_data_trial_train
```

Now we can call the above Random Forest classification function on each dataset to obtain its testing and training data performances.

In [17]:
```
raw_test_occupancy, raw_train_occupancy = random_forest_solver(occupancy_X_std, occupancy_Y)
raw_test_occupancy
```

Out[17]:

	accuracy	f1	roc_auc
0	0.991003	0.980704	0.990517
1	0.991302	0.981333	0.990680
2	0.991624	0.981877	0.988294
3	0.990510	0.979659	0.990035
4	0.991517	0.981719	0.989554

In [18]:
```
raw_test_adult, raw_train_adult = random_forest_solver(adult_X_std, adult_Y)
raw_test_adult
```

Out[18]:

	accuracy	f1	roc_auc
0	0.851094	0.903683	0.776796
1	0.849860	0.902442	0.780082
2	0.854505	0.905966	0.780569
3	0.852993	0.905655	0.770760
4	0.851844	0.904570	0.773209

In [19]:
```
raw_test_gamma, raw_train_gamma = random_forest_solver(gamma_X_std, gamma_Y)
raw_test_gamma
```

Out[19]:

	accuracy	f1	roc_auc
0	0.869662	0.903544	0.839334
1	0.872658	0.905300	0.844754
2	0.868569	0.902460	0.839372
3	0.870827	0.904279	0.841191
4	0.873704	0.905410	0.848995

In [20]:
```
raw_test_eye, raw_train_eye = random_forest_solver(eye_X_std, eye_Y)
raw_test_eye
```

Out[20]:

	accuracy	f1	roc_auc
0	0.894723	0.878918	0.890697
1	0.910922	0.898612	0.908011
2	0.895858	0.879761	0.891499
3	0.897929	0.881761	0.893294
4	0.901002	0.886896	0.897644

In [21]:
```
# Obtaining the mean across all 5 trials per dataset for the testing data

mean_adult_rf_score = raw_test_adult.mean().to_numpy()
mean_gamma_rf_score = raw_test_gamma.mean().to_numpy()
mean_eye_rf_score = raw_test_eye.mean().to_numpy()
mean_occupancy_rf_score = raw_test_occupancy.mean().to_numpy()
```

In [22]:
```
# Obtaining the mean across all 5 trials per dataset for the training data

mean_adult_rf_score_train = raw_train_adult.mean().to_numpy()
mean_gamma_rf_score_train = raw_train_gamma.mean().to_numpy()
mean_eye_rf_score_train = raw_train_eye.mean().to_numpy()
mean_occupancy_rf_score_train = raw_train_occupancy.mean().to_numpy()
```

In [23]:
```
# Saving the average metrics for both testing and training so we don't have to rerun the code

list_of_metrics_across_datasets = [mean_adult_rf_score, mean_gamma_rf_score,
                                   mean_eye_rf_score, mean_occupancy_rf_score]

list_of_metrics_across_datasets_train = [mean_adult_rf_score_train, mean_gamma_rf_score_train,
                                   mean_eye_rf_score_train, mean_occupancy_rf_score_train]
```

In [24]:
```
pd.DataFrame(list_of_metrics_across_datasets).to_csv('RF_metrics.csv', index = False)
pd.DataFrame(list_of_metrics_across_datasets_train).to_csv('RF_metrics_train.csv', index = False)
```

In [25]:
```
# Also saving the raw test scores for later use if needed

raw_test_occupancy.to_csv('RF_occupancy_test.csv', index = False)
raw_test_adult.to_csv('RF_adult_test.csv', index = False)
raw_test_gamma.to_csv('RF_gamma_test.csv', index = False)
raw_test_eye.to_csv('RF_eye_test.csv', index = False)
```

In [26]:
```
# saving the raw train scores for later use if needed

raw_train_occupancy.to_csv('RF_occupancy_train.csv', index = False)
raw_train_adult.to_csv('RF_adult_train.csv', index = False)
raw_train_gamma.to_csv('RF_gamma_train.csv', index = False)
raw_train_eye.to_csv('RF_eye_train.csv', index = False)
```

ANN / MLP

In [1]:
```
import pandas as pd
import numpy as np
from scipy.io import arff
import sklearn.model_selection
import sklearn.preprocessing
import sklearn.metrics
import sklearn.neural_network
```

Importing the 'Adult' dataset

In [2]:
```
adult = pd.read_csv('adult_cleaned.csv')
adult.head()
```

Out[2]:

	age	fnlwgt	education_num	capital_gain	capital_loss	hours/wk	salary	workclass_ ?	workclass_ Federal-gov	workclass_ Local-gov	...	native_country_ Portugal	nativ Pu
0	39	77516	13	2174	0	40	1	0	0	0	...	0	
1	50	83311	13	0	0	13	1	0	0	0	...	0	
2	38	215646	9	0	0	40	1	0	0	0	...	0	
3	53	234721	7	0	0	40	1	0	0	0	...	0	
4	28	338409	13	0	0	40	1	0	0	0	...	0	

5 rows × 109 columns

In [3]:
```
# Converting pandas series to numpy arrays
adult_Y = adult['salary'].to_numpy()

adult_X = adult.iloc[:, adult.columns != 'salary'].to_numpy()
#adult_X
```

In [4]:
```
# Scaling the data using a Standard scaler
std_scaler = sklearn.preprocessing.StandardScaler()
```

In [5]:
```
adult_X_std = std_scaler.fit_transform(adult_X)
```

Using the 'Gamma' dataset now

In [6]:
```
gamma = pd.read_csv('gamma_cleaned.csv')
gamma.head()
```

Out[6]:

	fLength	fWidth	fSize	fConc	fConc1	fAsym	fM3Long	fM3Trans	fAlpha	fDist	class
0	28.7967	16.0021	2.6449	0.3918	0.1982	27.7004	22.0110	-8.2027	40.0920	81.8828	1
1	31.6036	11.7235	2.5185	0.5303	0.3773	26.2722	23.8238	-9.9574	6.3609	205.2610	1
2	162.0520	136.0310	4.0612	0.0374	0.0187	116.7410	-64.8580	-45.2160	76.9600	256.7880	1
3	23.8172	9.5728	2.3385	0.6147	0.3922	27.2107	-6.4633	-7.1513	10.4490	116.7370	1
4	75.1362	30.9205	3.1611	0.3168	0.1832	-5.5277	28.5525	21.8393	4.6480	356.4620	1

In [7]:
```
# Converting pandas series to numpy arrays
gamma_Y = gamma['class'].to_numpy()

gamma_X = gamma.iloc[:, gamma.columns != 'class'].to_numpy()
```

In [8]:
```
# Scaling the gamma data using a Standard scaler
gamma_X_std = std_scaler.fit_transform(gamma_X)
```

Importing the cleaned 'Eye' data:

In [9]:
```
eye_df = pd.read_csv('eye_cleaned.csv')
eye_df.head()
```

Out[9]:

	AF3	F7	F3	FC5	T7	P7	O1	O2	P8	T8	FC6	F4	F8	AF4	eyeDe
0	4329.23	4009.23	4289.23	4148.21	4350.26	4586.15	4096.92	4641.03	4222.05	4238.46	4211.28	4280.51	4635.90	4393.85	
1	4324.62	4004.62	4293.85	4148.72	4342.05	4586.67	4097.44	4638.97	4210.77	4226.67	4207.69	4279.49	4632.82	4384.10	
2	4327.69	4006.67	4295.38	4156.41	4336.92	4583.59	4096.92	4630.26	4207.69	4222.05	4206.67	4282.05	4628.72	4389.23	
3	4328.72	4011.79	4296.41	4155.90	4343.59	4582.56	4097.44	4630.77	4217.44	4235.38	4210.77	4287.69	4632.31	4396.41	
4	4326.15	4011.79	4292.31	4151.28	4347.69	4586.67	4095.90	4627.69	4210.77	4244.10	4212.82	4288.21	4632.82	4398.46	

In [10]:
```
# Converting pandas series to numpy arrays
eye_Y = eye_df['eyeDetection'].to_numpy()
eye_X = eye_df.iloc[:, eye_df.columns != 'eyeDetection'].to_numpy()
```

In [11]:
```
# Scaling the eye data arrays using a Standard scaler
eye_X_std = std_scaler.fit_transform(eye_X)
```

Importing the cleaned 'Occupancy' dataset

In [12]:
```
occupancy_df = pd.read_csv('occupancy_cleaned.csv')
occupancy_df.head()
```

Out[12]:

	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
0	23.18	27.2720	426.0	721.25	0.004793	1
1	23.15	27.2675	429.5	714.00	0.004783	1
2	23.15	27.2450	426.0	713.50	0.004779	1
3	23.15	27.2000	426.0	708.25	0.004772	1
4	23.10	27.2000	426.0	704.50	0.004757	1

In [13]:
```
# Converting pandas series to numpy arrays
occupancy_Y = occupancy_df['Occupancy'].to_numpy()
occupancy_X = occupancy_df.iloc[:, occupancy_df.columns != 'Occupancy'].to_numpy()
```

In [14]:
```
# Scaling the occupancy dataset using a Standard scaler
occupancy_X_std = std_scaler.fit_transform(occupancy_X)
```

Running 5 trials of MLP on each dataset:

In [15]:
```
# List of all hyper parameters for ANN
ann_hidden_units = [1, 2, 4, 8, 32, 128]
ann_momentum = [0, 0.2, 0.5, 0.9]
```

In [16]:
```
# This function can be called on a given dataset and it will run
# MLP classification on it
def ANN_solver(X_data, Y_data):

    # These lists will store the training and testing
    # performance metrics across all 5 trials
    trial_scores_test = []
    trial_scores_train = []

    for i in range(5):
        # Splitting data into training (5000 samples) and testing (remaining data points)
        # Note: we are shuffling the data before sampling for randomness

        X_train, data_X_test, Y_train, data_Y_test = sklearn.model_selection.train_test_split(
            X_data, Y_data, train_size = 5000, shuffle = True, stratify = Y_data)

        # Creating a multiple layer perceptron Classifier model
        MLP_model = sklearn.neural_network.MLPClassifier(max_iter = 2000)

        # GridSearch object to cycle through hyperparameters along with cross validation
        # (verbose = 3 for output/debugging)
        data_grid_search = sklearn.model_selection.GridSearchCV(estimator = MLP_model,
                                                                 cv = 5,
                                                                 param_grid = [{'hidden_layer_sizes': ann_hidden_units,
                                                                                'momentum': ann_momentum,
                                                                                'solver': ['sgd']},
                                                                                {'hidden_layer_sizes': ann_hidden_units,
                                                                                'solver': ['adam']}
                                                                                ],
                                                                 scoring = ['accuracy', 'f1', 'roc_auc'],
                                                                 refit = 'accuracy', verbose = 3, n_jobs = -1)

        # Fitting the training data to perform a 5 fold cross validation
        data_grid_search.fit(X_train, Y_train)
```

```
data_grid_results = pd.DataFrame(data_grid_search.cv_results_)

# Obtaining the dataframe indicies corresponding to the best accuracy, f1, and roc_auc
# and obtaining the best parameters corresponding to those index values
data_ind_best_accuracy = data_grid_results['mean_test_accuracy'].idxmax()

data_accuracy_best_param_hidden_layer = data_grid_results['param_hidden_layer_sizes'][data_ind_best_accura
data_accuracy_best_param_momentum = data_grid_results['param_momentum'][data_ind_best_accuracy]
data_accuracy_best_param_solver = data_grid_results['param_solver'][data_ind_best_accuracy]

data_ind_best_f1 = data_grid_results['mean_test_f1'].idxmax()

data_f1_best_param_hidden_layer = data_grid_results['param_hidden_layer_sizes'][data_ind_best_f1]
data_f1_best_param_momentum = data_grid_results['param_momentum'][data_ind_best_f1]
data_f1_best_param_solver = data_grid_results['param_solver'][data_ind_best_f1]

data_ind_best_roc_auc = data_grid_results['mean_test_roc_auc'].idxmax()

data_roc_auc_best_param_hidden_layer = data_grid_results['param_hidden_layer_sizes'][data_ind_best_roc_auc
data_roc_auc_best_param_momentum = data_grid_results['param_momentum'][data_ind_best_roc_auc]
data_roc_auc_best_param_solver = data_grid_results['param_solver'][data_ind_best_roc_auc]

# Creating a list of best hyperparams per metric
list_best_param_roc_auc = [data_roc_auc_best_param_hidden_layer,
                   data_roc_auc_best_param_momentum,
                  data_roc_auc_best_param_solver]

list_best_param_f1 = [data_f1_best_param_hidden_layer,
                      data_f1_best_param_momentum,
                      data_f1_best_param_solver]

list_best_param_accuracy = [data_accuracy_best_param_hidden_layer,
                            data_accuracy_best_param_momentum,
                            data_accuracy_best_param_solver]

# This is used to create a dataframe to cycle through the parameters while re-training
overall_best_params = [list_best_param_accuracy, list_best_param_f1, list_best_param_roc_auc]

# The row indicies are ['accuracy', 'f1', 'roc_auc']
best_hyperparam = pd.DataFrame(overall_best_params, columns = ['hidden_layer', 'momentum', 'solver'])

# These training and testing lists store the performance of each
# model (3 models as there are 3 metrics)
model_scores_test = []
model_scores_train = []

for i in range(len(best_hyperparam)):

    # Stores the training and testing scores of a model in the loop
    test_scores = []
    train_scores = []

    if (best_hyperparam['solver'][i] == 'adam'):
        MLP_model_final = sklearn.neural_network.MLPClassifier(
            solver = 'adam',
            hidden_layer_sizes = best_hyperparam['hidden_layer'][i],
            max_iter = 2000
        )
    else:
        MLP_model_final = sklearn.neural_network.MLPClassifier(
            solver = 'sgd',
            max_iter = 2000,
            momentum = best_hyperparam['momentum'][i],
            hidden_layer_sizes = best_hyperparam['hidden_layer'][i]
        )

    # Fitting the corresponding model once more with the whole training data
    MLP_model_final.fit(X_train, Y_train)

    # Test data performance
    data_Y_pred = MLP_model_final.predict(data_X_test)

    test_scores.append(sklearn.metrics.accuracy_score(data_Y_test, data_Y_pred))
    test_scores.append(sklearn.metrics.f1_score(data_Y_test, data_Y_pred))
    test_scores.append(sklearn.metrics.roc_auc_score(data_Y_test, data_Y_pred))

    model_scores_test.append(test_scores)

    # Training data performance
    data_Y_pred_train = MLP_model_final.predict(X_train)

    train_scores.append(sklearn.metrics.accuracy_score(Y_train, data_Y_pred_train))
    train_scores.append(sklearn.metrics.f1_score(Y_train, data_Y_pred_train))
```

```
            train_scores.append(sklearn.metrics.roc_auc_score(Y_train, data_Y_pred_train))

            model_scores_train.append(train_scores)

        # Storing the performance of each testing data model
        model_df_test = pd.DataFrame(model_scores_test, columns = ['accuracy', 'f1', 'roc_auc'])
        model_mean_test = model_df_test.mean().to_numpy()

        # Storing the performance of each training data model
        model_df_train = pd.DataFrame(model_scores_train, columns = ['accuracy', 'f1', 'roc_auc'])
        model_mean_train = model_df_train.mean().to_numpy()

        trial_scores_test.append(model_mean_test)
        trial_scores_train.append(model_mean_train)

    # These datasets contain all the 3 metrics performances of each trial in a dataframe format
    MLP_data_trial_test = pd.DataFrame(trial_scores_test, columns = ['accuracy', 'f1', 'roc_auc'])
    MLP_data_trial_train = pd.DataFrame(trial_scores_train, columns = ['accuracy', 'f1', 'roc_auc'])

    return MLP_data_trial_test, MLP_data_trial_train
```

Now we can call the above MLP classifier function on each dataset to obtain its testing and training data performances

In [17]:
```
raw_test_occupancy, raw_train_occupancy = ANN_solver(occupancy_X_std, occupancy_Y)
raw_test_occupancy
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   25.9s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.6min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  3.0min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   21.2s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.4min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.8min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   29.2s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.4min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.7min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   21.5s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.3min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.6min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   27.4s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.4min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.8min finished
```

Out[17]:

	accuracy	f1	roc_auc
0	0.989310	0.977261	0.991038
1	0.989267	0.977159	0.990816
2	0.988689	0.976035	0.991575
3	0.989139	0.976950	0.991608
4	0.915510	0.650810	0.827564

In [18]:
```
raw_test_gamma, raw_train_gamma = ANN_solver(gamma_X_std, gamma_Y)
raw_test_gamma
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   15.4s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.3min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  3.0min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   17.3s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.2min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.9min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   18.1s
[Parallel(n_jobs=-1)]: Done 128 tasks      | elapsed:  2.6min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  3.0min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   11.9s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.3min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  3.1min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   20.7s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  2.6min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  3.4min finished
```

Out[18]:

	accuracy	f1	roc_auc
0	0.867000	0.901320	0.837497
1	0.867118	0.901594	0.836846
2	0.865383	0.900209	0.835383
3	0.870233	0.902945	0.844585
4	0.863742	0.897777	0.838775

In [19]:
```
raw_test_adult, raw_train_adult = ANN_solver(adult_X_std, adult_Y)
raw_test_adult
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   21.2s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  3.0min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  5.1min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   24.5s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  4.2min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  6.6min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   39.6s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  4.5min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  6.9min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   35.1s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  4.8min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  6.9min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   45.2s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  4.6min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  6.7min finished
```

Out[19]:

	accuracy	f1	roc_auc
0	0.842991	0.898827	0.761532
1	0.841539	0.897946	0.758930
2	0.840511	0.896399	0.766843
3	0.843595	0.899297	0.761484
4	0.844309	0.899849	0.761440

In [20]:
```
raw_test_eye, raw_train_eye = ANN_solver(eye_X_std, eye_Y)
raw_test_eye
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    6.4s
[Parallel(n_jobs=-1)]: Done 128 tasks      | elapsed:   46.9s
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.0min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    8.5s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:   52.5s
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.2min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    9.8s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed: 16.6min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed: 18.0min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   10.4s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.9min finished
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    6.6s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:   41.6s
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:  2.3min finished
```

Out[20]:

	accuracy	f1	roc_auc
0	0.852705	0.833784	0.849976
1	0.855177	0.837082	0.852778
2	0.859185	0.839727	0.855722
3	0.852772	0.834691	0.850375
4	0.863560	0.845327	0.860514

In [21]:
```
# Obtaining the mean across all 5 trials per dataset for the testing data

mean_adult_ANN_score = raw_test_adult.mean().to_numpy()
mean_gamma_ANN_score = raw_test_gamma.mean().to_numpy()
mean_eye_ANN_score = raw_test_eye.mean().to_numpy()
mean_occupancy_ANN_score = raw_test_occupancy.mean().to_numpy()
```

In [22]:
```
# Obtaining the mean across all 5 trials per dataset for the training data

mean_adult_ANN_score_train = raw_train_adult.mean().to_numpy()
mean_gamma_ANN_score_train = raw_train_gamma.mean().to_numpy()
mean_eye_ANN_score_train = raw_train_eye.mean().to_numpy()
mean_occupancy_ANN_score_train = raw_train_occupancy.mean().to_numpy()
```

In [23]:
```
# Saving the average metrics for both testing and training so we don't have to rerun the code

list_of_metrics_across_datasets = [mean_adult_ANN_score, mean_gamma_ANN_score,
                                   mean_eye_ANN_score, mean_occupancy_ANN_score]

list_of_metrics_across_datasets_train = [mean_adult_ANN_score_train, mean_gamma_ANN_score_train,
                                   mean_eye_ANN_score_train, mean_occupancy_ANN_score_train]
```

In [24]:
```
pd.DataFrame(list_of_metrics_across_datasets).to_csv('ANN_metrics.csv', index = False)
pd.DataFrame(list_of_metrics_across_datasets_train).to_csv('ANN_metrics_train.csv', index = False)
```

In [25]:
```
# Also saving the raw test scores for later use if needed

raw_test_occupancy.to_csv('ANN_occupancy_test.csv', index = False)
raw_test_adult.to_csv('ANN_adult_test.csv', index = False)
raw_test_gamma.to_csv('ANN_gamma_test.csv', index = False)
raw_test_eye.to_csv('ANN_eye_test.csv', index = False)
```

In [26]:
```
# saving the raw train scores for later use if needed

raw_train_occupancy.to_csv('ANN_occupancy_train.csv', index = False)
raw_train_adult.to_csv('ANN_adult_train.csv', index = False)
raw_train_gamma.to_csv('ANN_gamma_train.csv', index = False)
raw_train_eye.to_csv('ANN_eye_train.csv', index = False)
```

Table Calculations

In [1]:
```
import pandas as pd
import numpy as np
from scipy.io import arff
import sklearn.model_selection
import sklearn.preprocessing
import sklearn.linear_model
import sklearn.metrics
import scipy.stats
```

In [2]:
```
# Importing Logistic Regression's raw test scores of the 4 datasets

Log_raw_test_occupancy = pd.read_csv('Log_occupancy_test.csv')
Log_raw_test_adult = pd.read_csv('Log_adult_test.csv')
Log_raw_test_gamma = pd.read_csv('Log_gamma_test.csv')
Log_raw_test_eye = pd.read_csv('Log_eye_test.csv')
```

In [3]:
```
# Importing ANN's raw test scores of the 4 datasets

ANN_raw_test_occupancy = pd.read_csv('ANN_occupancy_test.csv')
ANN_raw_test_adult = pd.read_csv('ANN_adult_test.csv')
ANN_raw_test_gamma = pd.read_csv('ANN_gamma_test.csv')
ANN_raw_test_eye = pd.read_csv('ANN_eye_test.csv')
```

In [4]:
```
# Importing Random Forest's raw test scores of the 4 datasets

RF_raw_test_occupancy = pd.read_csv('RF_occupancy_test.csv')
RF_raw_test_adult = pd.read_csv('RF_adult_test.csv')
RF_raw_test_gamma = pd.read_csv('RF_gamma_test.csv')
RF_raw_test_eye = pd.read_csv('RF_eye_test.csv')
```

In [5]:
```
# Obtaining the mean of all trials of all 4 datasets for ease of calculation

Log_Reg_test_df = pd.read_csv('Log_Reg_metrics.csv')
ANN_test_df = pd.read_csv('ANN_metrics.csv')
RF_test_df = pd.read_csv('RF_metrics.csv')
```

In [6]:
```
# Viewing the data

# Indicies are in the order: adult, gamma, eye, and occupancy
# Columns are in the order: accuracy, f1, and roc_auc
Log_Reg_test_df
```

Out[6]:

	0	1	2
0	0.848140	0.902507	0.764484
1	0.788982	0.846568	0.743019
2	0.638597	0.547579	0.624552
3	0.988985	0.976621	0.991372

Table 3 Calculations

Here, we take the average from left to right (over the metrics) to get the average performance for 4 datasets

In [7]:
```
Log_Reg_test_df.mean(axis = 1).to_numpy()
```

Out[7]:
```
array([0.83837737, 0.79285651, 0.60357616, 0.98565918])
```

In [8]:
```
ANN_test_df.mean(axis = 1).to_numpy()
```

Out[8]:
```
array([0.83436618, 0.86869388, 0.84955852, 0.94818211])
```

In [9]:
```
RF_test_df.mean(axis = 1).to_numpy()
```

Out[9]:
```
array([0.84426842, 0.87267064, 0.89383507, 0.98735524])
```

Table 2 Calculations

Here, we take the mean top to bottom (across all datasets) to obtain the average accuracy, f1 score, and roc_auc

In [10]:
```
print(Log_Reg_test_df.mean().to_numpy())
print(ANN_test_df.mean().to_numpy())
print(RF_test_df.mean().to_numpy())
```

```
[0.81617617 0.81831894 0.7808568 ]
[0.88508686 0.88724954 0.85326412]
[0.90360529 0.91872745 0.87626429]
```

Table 2 t-test

Here, we perform a t-test between the best algorithm (Random Forest Classifier) and the other two algortihms.

For this, we will have to first create a dataframe with data of all trials on all metrics.

In [11]:
```
# The resulting dataframe has 20 rows - 5 trials x 4 datasets
Log_raw_test_occ_adult = Log_raw_test_occupancy.append(Log_raw_test_adult)
Log_raw_test_occ_adult_gamma = Log_raw_test_occ_adult.append(Log_raw_test_gamma)
Log_raw_test_occ_adult_gamma_eye = Log_raw_test_occ_adult_gamma.append(Log_raw_test_eye)
```

In [12]:
```
ANN_raw_test_occ_adult = ANN_raw_test_occupancy.append(ANN_raw_test_adult)
ANN_raw_test_occ_adult_gamma = ANN_raw_test_occ_adult.append(ANN_raw_test_gamma)
ANN_raw_test_occ_adult_gamma_eye = ANN_raw_test_occ_adult_gamma.append(ANN_raw_test_eye)
```

In [13]:
```
RF_raw_test_occ_adult = RF_raw_test_occupancy.append(RF_raw_test_adult)
RF_raw_test_occ_adult_gamma = RF_raw_test_occ_adult.append(RF_raw_test_gamma)
RF_raw_test_occ_adult_gamma_eye = RF_raw_test_occ_adult_gamma.append(RF_raw_test_eye)
```

In [14]:
```
# Looking at the resultant dataframe for Random Forest
RF_raw_test_occ_adult_gamma_eye
```

Out[14]:

	accuracy	f1	roc_auc
0	0.991003	0.980704	0.990517
1	0.991302	0.981333	0.990680
2	0.991624	0.981877	0.988294
3	0.990510	0.979659	0.990035
4	0.991517	0.981719	0.989554
0	0.851094	0.903683	0.776796
1	0.849860	0.902442	0.780082
2	0.854505	0.905966	0.780569
3	0.852993	0.905655	0.770760
4	0.851844	0.904570	0.773209
0	0.869662	0.903544	0.839334
1	0.872658	0.905300	0.844754
2	0.868569	0.902460	0.839372
3	0.870827	0.904279	0.841191
4	0.873704	0.905410	0.848995
0	0.894723	0.878918	0.890697
1	0.910922	0.898612	0.908011
2	0.895858	0.879761	0.891499
3	0.897929	0.881761	0.893294
4	0.901002	0.886896	0.897644

In [15]:
```
# Looknig at Random Forest vs Logistic Regression for each performance metric

RF_vs_Log_Acurracy = scipy.stats.ttest_ind(
    RF_raw_test_occ_adult_gamma_eye['accuracy'].to_numpy(),
    Log_raw_test_occ_adult_gamma_eye['accuracy'].to_numpy()
)

RF_vs_Log_F1 = scipy.stats.ttest_ind(
    RF_raw_test_occ_adult_gamma_eye['f1'].to_numpy(),
    Log_raw_test_occ_adult_gamma_eye['f1'].to_numpy()
)

RF_vs_Log_ROC_AUC = scipy.stats.ttest_ind(
    RF_raw_test_occ_adult_gamma_eye['roc_auc'].to_numpy(),
    Log_raw_test_occ_adult_gamma_eye['roc_auc'].to_numpy()
)

print(RF_vs_Log_Acurracy[1], RF_vs_Log_F1[1], RF_vs_Log_ROC_AUC[1])
```

```
0.008187465318745583 0.012618893903563856 0.010315479421569445
```

In [16]:
```
# Looknig at Random Forest vs ANN for each performance metric

RF_vs_ANN_Acurracy = scipy.stats.ttest_ind(
    RF_raw_test_occ_adult_gamma_eye['accuracy'].to_numpy(),
    ANN_raw_test_occ_adult_gamma_eye['accuracy'].to_numpy()
)

RF_vs_ANN_F1 = scipy.stats.ttest_ind(
    RF_raw_test_occ_adult_gamma_eye['f1'].to_numpy(),
    ANN_raw_test_occ_adult_gamma_eye['f1'].to_numpy()
)

RF_vs_ANN_ROC_AUC = scipy.stats.ttest_ind(
    RF_raw_test_occ_adult_gamma_eye['roc_auc'].to_numpy(),
    ANN_raw_test_occ_adult_gamma_eye['roc_auc'].to_numpy()
)

print(RF_vs_ANN_Acurracy[1], RF_vs_ANN_F1[1], RF_vs_ANN_ROC_AUC[1])
```

```
0.2964318515947864 0.09588481962661524 0.36790718171376713
```

In [17]:
```
# Looknig at ANN vs Logistic Regression for each performance metric

ANN_vs_Log_Acurracy = scipy.stats.ttest_ind(
    ANN_raw_test_occ_adult_gamma_eye['accuracy'].to_numpy(),
    Log_raw_test_occ_adult_gamma_eye['accuracy'].to_numpy()
)

ANN_vs_Log_F1 = scipy.stats.ttest_ind(
    ANN_raw_test_occ_adult_gamma_eye['f1'].to_numpy(),
    Log_raw_test_occ_adult_gamma_eye['f1'].to_numpy()
)

ANN_vs_Log_ROC_AUC = scipy.stats.ttest_ind(
    ANN_raw_test_occ_adult_gamma_eye['roc_auc'].to_numpy(),
    Log_raw_test_occ_adult_gamma_eye['roc_auc'].to_numpy()
)

print(ANN_vs_Log_Acurracy[1], ANN_vs_Log_F1[1], ANN_vs_Log_ROC_AUC[1])
```

```
0.03443581961897483 0.09946683482642012 0.04685640618360718
```

Table 3 t-test

Here, we perform a t-test between the best algorithm (Random Forest Classifier) and the other two algortihms.

For this, we will have to first create a dataframe with data of all trials and metrics per dataset. We can use the helper function below for this:

In [18]:
```
# This function takes all the metrics of 5 trials and combines it into
# one list (in the order accuracy, f1, roc_auc)

def dataset_metric_converter(raw_test_data):

    test_accuracy = list(raw_test_data['accuracy'].to_numpy())
    test_f1 = list(raw_test_data['f1'].to_numpy())
    test_roc_auc = list(raw_test_data['roc_auc'].to_numpy())

    # Appending items from accuracy, f1, and roc_auc to one list
    for i in test_f1:
        test_accuracy.append(i)

    for i in test_roc_auc:
        test_accuracy.append(i)

    test_all_metrics = test_accuracy
    return test_all_metrics
```

Let's run this function on all dataset/algorithm combinations

In [19]:
```
Log_test_adult_across_metrics = dataset_metric_converter(Log_raw_test_adult)
Log_test_gamma_across_metrics = dataset_metric_converter(Log_raw_test_gamma)
Log_test_eye_across_metrics = dataset_metric_converter(Log_raw_test_eye)
Log_test_occupancy_across_metrics = dataset_metric_converter(Log_raw_test_occupancy)
```

In [20]:
```
ANN_test_adult_across_metrics = dataset_metric_converter(ANN_raw_test_adult)
ANN_test_gamma_across_metrics = dataset_metric_converter(ANN_raw_test_gamma)
ANN_test_eye_across_metrics = dataset_metric_converter(ANN_raw_test_eye)
ANN_test_occupancy_across_metrics = dataset_metric_converter(ANN_raw_test_occupancy)
```

In [21]:
```
RF_test_adult_across_metrics = dataset_metric_converter(RF_raw_test_adult)
RF_test_gamma_across_metrics = dataset_metric_converter(RF_raw_test_gamma)
RF_test_eye_across_metrics = dataset_metric_converter(RF_raw_test_eye)
RF_test_occupancy_across_metrics = dataset_metric_converter(RF_raw_test_occupancy)
```

Now we can compare the metrics of the same dataset and two different algorithms (Random Forest and the other two one by one)

In [22]:
```
scipy.stats.ttest_ind(RF_test_adult_across_metrics, ANN_test_adult_across_metrics)[1]
```

Out[22]:
```
0.6336941506555742
```

In [23]:
```
scipy.stats.ttest_ind(RF_test_adult_across_metrics,
                      Log_test_adult_across_metrics)[1]
```

Out[23]:
```
0.7780745892567839
```

In [24]:
```
scipy.stats.ttest_ind(RF_test_gamma_across_metrics,
                      ANN_test_gamma_across_metrics)[1]
```

Out[24]:
```
0.6816936081891869
```

In [25]:
```
scipy.stats.ttest_ind(RF_test_gamma_across_metrics,
                      Log_test_gamma_across_metrics)[1]
```

Out[25]:
```
1.5979646295434768e-06
```

In [26]:
```
scipy.stats.ttest_ind(RF_test_eye_across_metrics,
                      ANN_test_eye_across_metrics)[1]
```

Out[26]:
```
2.777305131112547e-13
```

In [27]:
```
scipy.stats.ttest_ind(RF_test_eye_across_metrics,
                      Log_test_eye_across_metrics)[1]
```

Out[27]:
```
2.637741738585312e-21
```

In [28]:
```
scipy.stats.ttest_ind(RF_test_occupancy_across_metrics,
                      ANN_test_occupancy_across_metrics)[1]
```

Out[28]:
```
0.11498185392499986
```

In [29]:
```
scipy.stats.ttest_ind(RF_test_occupancy_across_metrics,
                      Log_test_occupancy_across_metrics)[1]
```

Out[29]:
```
0.4299841217341497
```

Also comparing ANN with Logistic Regression for Appendix

In [30]:
```
scipy.stats.ttest_ind(ANN_test_adult_across_metrics,
                      Log_test_adult_across_metrics)[1]
```

Out[30]: 0.8521266248396446

In [31]:
```
scipy.stats.ttest_ind(ANN_test_gamma_across_metrics,
                      Log_test_gamma_across_metrics)[1]
```

Out[31]: 3.7742060078438537e-06

In [32]:
```
scipy.stats.ttest_ind(ANN_test_eye_across_metrics,
                      Log_test_eye_across_metrics)[1]
```

Out[32]: 2.2330152543339495e-19

In [33]:
```
scipy.stats.ttest_ind(ANN_test_occupancy_across_metrics,
                      Log_test_occupancy_across_metrics)[1]
```

Out[33]: 0.13134123351858837