COGS 118A Final Project

Melchisedec Lee MJL002@UCSD.EDU
A15597085

Yunchun Pan YUP086@UCSD.EDU
A15195894

Duy Pham DPHAM@UCSD.EDU
A15467782

Abstract

This paper is a replication of Caruana and Niculescu-Mizil's 2006: An Empirical Comparison of Supervised Learning Algorithms, which is henceforth referred to as CNM06. We compare the performance of several common machine learning algorithms: decision tree, K-nearest neighbors, logistic regression, perceptron classifier, random forests, and support vector machine. We evaluate the performance of these algorithms in binary classification by the metrics of accuracy, precision, recall, specificity, F1 score, and area under ROC curve. Our results corroborate the conclusion by CNM06 that SVM and random forests achieve the best performance across all datasets and performance metrics. However, logistic regression and decision tree, two models reported as the worst algorithms by CNM06, achieve similarly good performance as SVM and random forests in our project, while perceptron classifier, the model that is not used in CNM06, achieves the worst performance.

Keywords: Binary Classification, Support Vector Machine, K-Nearest Neighbors, Logistic Regression, Random Forests, Perceptron Classifier, Decision Trees, Accuracy, Precision, Recall, Specificity, F1 Score, ROC AUC, Supervised Models

1. Introduction

Machine learning is one of the major disciplines of data science, as it detects patterns in big data to create predictive models for future use. Machine learning can work in complex and noisy datasets, such as complex genomic data in cancer studies (Huang et al. 2018). In addition, it is useful to know which algorithms achieve the best performance on a wide variety of datasets and conditions.

STATLOG is one of the first example of such project (Henery and Taylor 1992). It evaluates the performance of machine learning algorithms in binary classification. Binary classification is the separation of data into two groups. Following STATLOG, CNM06 uses a larger set of algorithms and datasets (Caruana and Niculescu-Mizil 2006). It uses different metrics, including accuracy and F1-scores to measure and compare performance of algorithms on 11 different datasets. In addition, CNM06 performs tests to examine the significance of the difference in performance metric scores of different algorithms. Based

on the results, CNM06 concludes that on average, SVM and random forest are the best in overall performance, while logistic regression and decision trees perform the worst.

The goal of our project is to replicate Rich Caruana's CNM06: An Empirical Comparison of Supervised Learning Algorithms. Many different machine learning algorithms can be used for binary classification, but the ones we choose are decision trees, logistic regression, perceptron classifier, support vector machine, k-nearest neighbors, and random forests. We train these methods on six different datasets, including rain in Australia to poisonous mushrooms. After training, we evaluate performance of models using six different metrics: accuracy, precision, recall, specificity, F1 score, and area under ROC curve. Then, we conduct two sampled t-tests to detect the significance of the difference in performance scores of different algorithms. Lastly, we compare the results of metrics scores to identify algorithms with best and worst performance.

2. Methodology

2.1 Learning Algorithms

Six different classifiers are included for performance comparison. These algorithms are decision trees, k-nearest neighbors, logistic regression, perceptron classifier, random forests, and support vector machine. Through 5-fold cross validation, the grid search uses the following hyper-parameters to tune each algorithm with all other parameters set to default values:

1. Decision Tree (DT) - We consider the following maximum depths for the tree: 2, 3, 4, 5, 7, 10, 13, 15, 18, and default value None. The minimum number of samples required to split an internal node and required to be at a leaf node varies between 2, 3, 5, 7, 10, 15, and 20.

2. K-Nearest Neighbors (KNN) - We explore 26 different values for the number of neighbors. These values are evenly spaced between 1 to 106 in increments of 4. The distance metrics used for the tree are Euclidean, Manhattan, and Minkowski distance.

3. Logistic Regression (LOGREG) - When we train unregularized model, algorithms used in the optimization problem are "saga" and "lbfgs". When we train the model regularized by only L2 norm, "lbfgs" is the algorithm used in the optimization problem. When we train the model regularized by L1 and L2 norm, "saga" is the algorithm used in the optimization problem. The values of ridge (regularization) parameter vary from 10^{-4} to 10^{4}. For both unregularized and regularized models, we set the maximum number of iterations to 5000 for models to converge.

4. Perceptron Classifier (PERC) - We train both unregularized and regularized models. The regularization terms used are L1, L2 norm, and elastic net. The constants of regularization term are 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, and 10^{-1}.

5. random forests (RF) - We use 1024 tree estimators and consider the following values for the minimum number of samples required for each split: 1, 2, 4, 6, 8, 12, 16, and 20.

6. Support Vector Machine (SVM) - Varying the regularization parameter between 10^{-3}, 10^{-1}, and 10, we use the following kernels for the classifier: linear, polynomial with degrees of 2 and 3, and radial with width of 10^{-3}, 10^{-2}, 10^{-1}, 1, and 2.

2.2 Performance Metrics

Six metrics, accuracy, precision, recall, specificity, F1 score, and area under ROC curve, are used to evaluate the performance of each of six algorithms in grid search used for 5-fold cross validation, training predictions, and testing predictions. These scores are in the range from 0 to 1. The equations for calculating each metric are specified below:

1. Accuracy (ACC) - Accuracy measures the proportion of correct predictions out of all predictions.
$$\frac{\sum \text{True Positive} + \sum \text{True Negatives}}{\sum \text{Total Population}}$$

2. Precision (PREC) - Precision measures the proportion of true positive samples out of all predicted positive samples.
$$\frac{\sum \text{True Positive}}{\sum \text{Predicted Condition Positive}}$$

3. Recall (REC) - Recall measures the proportion of positive samples predicted correctly out of all positive samples.
$$\frac{\sum \text{True Positive}}{\sum \text{Condition Positive}}$$

4. Specificity (SPEC) - Specificity measures the proportion of negative samples predicted correctly out of all negative samples.
$$\frac{\sum \text{True Negative}}{\sum \text{Condition Negative}}$$

5. F1 score (F1) - F1 score combines precision and recall scores into a single score.
$$\frac{2 * \text{PREC} * \text{REC}}{\text{REC} + \text{PREC}}$$

6. ROC AUC (ROC) - ROC AUC score measures the ability of the model to increase the true positive rate while minimizing the increase of the false positive rate. It is generated by plotting the true and false positive rate at different thresholds and plotting the area under this curve.

To measure the overall performance of the model, we calculate mean across metrics (MEAN) by averaging across all 6 metrics. The equation for calculating this score is

$$\frac{\text{ACC} + \text{PREC} + \text{REC} + \text{SPEC} + \text{F1} + \text{ROC}}{6}$$

2.3 Data Sets

Six data sets are retrieved from Kaggle: CHESS, SHROOMS, CARDIO, AUS, AIRBNB, and OLYMPIC. In each data set, six algorithms are trained and tested. The number of attributes, sample size of training and testing sets, and the proportion of positive labels in each data set is summarized in Table 1.

1. Cardiovascular Disease (CARDIO) - CARDIO contains 70000 observations of patients collected at the moment of medical examination. It includes a binary target variable that indicates the presence or absence of cardiovascular disease. Excluding this variable, a total of 11 features are present in this data set. Among these 11 features, age, height, weight, and gender are factual information; systolic blood pressure, diastolic blood pressure, cholesterol, and glucose are results of medical examination; smoke, alcohol intake, and physical activity are information given by the patient. In CARDIO, categorical data are converted into integers by one-hot encoding, while numerical data are scaled to 0 mean and 1 standard deviation by Z-score standardization. As age of patients are in days, some transformations are applied to convert this attribute to age in years. After transformations, all 11 features are used to predict the presence or absence of cardiovascular disease of a patient. It creates a well-balanced classification problem, with approximately 50.03% negative labels and 49.97% positive labels.

2. Rain in Australia (AUS) - AUS contains 145460 daily weather observations from many locations across Australia in about 10 years. It includes a binary target variable that indicates whether it rained the next day at the same location. Excluding this variable, a total of 22 features are included. As the original data contains missing values, we fill missing values in categorical data with the mode and those in numerical data with the mean. Some transformations are used to split the date of weather observations to year, month, and day in numerical formats. One-hot encoding is used to process categorical data into integers, while Z-score is applied to scale numerical data to 0 mean and 1 standard deviation. After transformations, all 22 features are used to predict whether it rained the next day at the same location or not. The negative and positive labels in AUS are unbalanced, with 78.0854% negative labels, 21.9146% positive labels.

3. Online Chess Games (CHESS) - CHESS contains 20058 games of chess from lichess.org. We predict whether white wins the match or not. White winning is the positive label, while black winning or a draw is the negative label. We use 8 features for prediction, and these features are the start and end timestamps of the match, the number of turns taken, the victory status (draw, timeout, or mate), the ratings of both players, the opening the players used, and the length of the opening. We apply one-hot encoding to categorical variables, which are whether the game was rated, the victory status, and the opening. In addition, we apply Z-score standardization to scale numerical variables to 0 mean and 1 standard deviation. The positive and negative-class samples in the dataset is well-balanced, with 49.86% positive labels and 50.14% negative labels.

4. Edible and Poisonous Mushrooms (SHROOMS) - SHROOMS contains 8124 hypothetical descriptions of mushrooms from the UCI Machine Learning Repository. We

predict whether the mushroom is edible or poisonous. Being edible is the negative label, while being poisonous is the positive label. We use 21 features for this prediction, including cap shape, odor, and number of rings. All these features are categorical, so we apply one-hot encoding to transform them into numbers. This is also a well-balanced dataset, with 48.20% positive labels and 51.80% negative labels.

5. AirBnB Prices (AIRBNB) - AIRBNB includes 47906 AirBnB listings of New York City in 2019. We use 7 listing features to predict whether or not an AirBnb is expensive relative to the other listings in AIRBNB. We classify the place as being expensive or not expensive by comparing its price with the median value of all AirBnB prices. If the price of an AirBnb is greater than this threshold, we label it as expensive. Otherwise, it is labeled as not expensive and have negative label. We convert the latitude, longitude, minimum nights, number of reviews, and days available to numerical values. One-hot encoding is used to process categorical data into integers, while Z-score is applied to scale numerical data to 0 mean and 1 standard deviation. Using the median allows the data to have a roughly equal number of positive and negative-class samples, classifying 49.95% places as expensive with positive label and 50.05% places as not expensive with negative labels.

6. Olympic Medalists (OLYMPIC) - OLYMPIC includes data on all 231333 Olympians from Athens 1896 to Rio 2016 and their 6 features (i.e height, weight, age, etc). However, we are only interested in the medalists of OLYMPIC. Removing non-medalists results in a dataset of 39783 samples. Then, we use medalists' physical features, country, and sport to classify them as gold medalists or not. We apply transformations to convert the age, weight, and height of the Olympians into a numerical values. One-hot encoding is used to process categorical data into integers, while Z-score is applied to scale numerical data to 0 mean and 1 standard deviation. Since there are only bronze, silver, and gold medalists, our data is roughly split into one third gold medalists and two thirds non-gold medalists, with 33.68% data being of positive labels as gold medalists, and 66.31% data being of negative labels as non-gold medalists.

Table 1. Problem Set Summary

NAME	#ATTR	TRAIN SIZE	TEST SIZE	%POS
CARDIO	11	5000	65000	49.97%
AUS	22	5000	140460	21.91%
CHESS	9/15	5000	15058	49.86%
SHROOMS	22	5000	3124	48.20%
AIRBNB	7/15	5000	43895	49.95%
OLYMPIC	6/14	5000	25181	33.69%

3. Experiments & Results

For each algorithm and dataset combination, we perform 7 trials. In each trial, we randomly select 5000 data samples as training set and include all other samples as testing set. Then, we conduct a grid search with 5-fold cross validation to find the optimal hyper-parameters per model for each of 6 performance metrics. For each metric, we train a model with the corresponding hyper-parameters on the entire 5000 samples. We then predict the labels of both training and testing set to calculate the corresponding metric on these two sets.

In each column of the following tables, the best score is bold-faced. We then perform two-sampled t-tests that compares the results of best algorithm against each remaining algorithm. If the resulting p-value is below 0.05, we consider the difference between algorithms' performance significant. Otherwise, these models perform as good as the best model. In each column, scores that are not significantly different from the best score ($p \geq 0.05$) are annotated with asterisks, while scores that indicate significantly worse performance ($p < 0.05$) are annotated with (!). The corresponding p-values are recorded into supplemental tables.

Table 2 — Test Set Performance by Metric Per Algorithm

MODEL	ACC	PREC	REC	SPEC	F1	ROC	MEAN
DT	0.78328*	0.80043*	0.69801*	0.88309 (!)	0.69009*	0.74225*	0.76619*
KNN	0.76827*	0.75903*	0.67829*	0.85024 (!)	0.67306*	0.72648*	0.74256*
LOGREG	0.78207*	0.75299*	0.63298*	0.88603*	0.65750*	0.73693*	0.74142*
PERC	0.70155 (!)	0.61399 (!)	0.61524*	0.76143 (!)	0.57914 (!)	0.67861 (!)	0.65833 (!)
RF	**0.79331**	0.76478*	0.66909*	0.85039 (!)	0.69930*	0.75285*	0.75495*
SVM	0.79090*	**0.80691**	**0.74862**	**0.96359**	**0.70647**	**0.75464**	**0.79519**

The mean test set performance across 7 trials and 6 datasets for each algorithm and performance metric combination are recorded in Table 2. Since bold-faced scores in almost all columns are produced by SVM, SVM generally gets the best score for each metric except for accuracy score. However, further observation of scores annotated with asterisks indicates that, decision tree, KNN, logistic regression, and random forests also perform similar to SVM on all metrics except for specificity score. In addition, observations of scores with (!) symbols indicate that perceptron classifier performs worse than SVM, which is suggested by its small p-values in Table 2 Supplemental. This ranking of models' performance also preserves in their mean scores averaged across all six metrics.

Table 3 — Test Set Performance by Data Set Per Algorithm

MODEL	CARDIO	AUS	CHESS	SHROOMS	AIRBNB	OLYMPIC	MEAN
DT	0.73919 (!)	0.71198*	0.67926 (!)	0.99977 (!)	0.82571 (!)	**0.64124**	0.76619*
KNN	0.66677 (!)	0.69958*	0.63454 (!)	**1.00000***	0.82976 (!)	0.62471*	0.74256*
LOGREG	0.71427 (!)	0.73958*	0.67599*	**1.00000***	0.82729*	0.49137 (!)	0.74142*
PERC	0.65575 (!)	0.48589 (!)	0.58234 (!)	0.99988 (!)	0.68440 (!)	0.54169 (!)	0.65833*
RF	0.72881 (!)	0.70487*	0.66337 (!)	**1.00000***	0.83290 (!)	0.59977*	0.75495*
SVM	**0.79510**	**0.74239**	**0.72951**	**1.00000***	**0.86815**	0.63598*	**0.79519**

We also compute the mean test set performance across 7 trials and 6 performance metrics for each algorithm and dataset combination and record results in Table 3. For combination,

the best score is bold-faced. Similar two-sampled t-tests are conducted, giving annotations to all other scores in each column. The corresponding p-values are recorded in Table 3 Supplemental.

For CARDIO, a well-balanced data set of approximately 51% negative-class and 49% positive-class samples, SVM achieves best performance, and all other algorithms perform significantly worse than SVM.

For AUS, a very unbalanced dataset of about 78% negative labels and 22% positive labels, SVM also achieves best performance, and all other algorithms except preceptron classifier perform similar to SVM on this dataset since the differences in their scores are significant. Perceptron classifier performs worst on AUS with a score of 0.48589, which is significantly lower than SVM's score of 0.74239.

For CHESS, which is well-balanced dataset, SVM again performs the best. Logistic regression performs similarly well, but all other algorithms perform significantly worse.

For another well-balanced dataset SHROOMS, KNN, logistic regression, random forests, and SVM all achieve scores of 100%, while decision tree and perceptron classifer achieves scores slightly lower than 100%. SHROOMS contains hypothetical samples that were generated without noise. This could be the reason why many algorithms achieve perfect performance on this dataset. Hence, the classification of edible and poisonous mushrooms is very easy for these algorithms.

For well-balanced AIRBNB dataset, SVM again achieves the best performance, while all other algorithms except logistic regression perform significantly worse.

For OLYMPIC, a slightly unbalanced dataset with 66% negative-class and 33% positive-class samples, even though decision tree performs best among all models, its score of 0.64124 is low relative to best scores on other datasets. Therefore, OLYMPIC is likely the most difficult dataset for prediction. In addition, logistic regression performs worst out of all models on OLYMPIC dataset.

Even though SVM performs best and perceptron classifier performs worst on almost all datasets, the asterisks around scores in Mean column indicate that all algorithms have just as good overall performance as SVM across all metrics and datasets.

Table 4 — Training Set Performance By Metric Per Algorithm

MODEL	ACC	PREC	REC	SPEC	F1	ROC	MEAN
DT	0.80141 (!)	0.82194 (!)	0.81676 (!)	0.90647 (!)	0.77365 (!)	0.80007 (!)	0.82005 (!)
KNN	0.78696 (!)	0.78117 (!)	0.87263 (!)	0.87924 (!)	0.86162 (!)	0.84251 (!)	0.83735 (!)
LOGREG	0.78626 (!)	0.77019 (!)	0.63927 (!)	0.88730 (!)	0.66742 (!)	0.74413 (!)	0.74910 (!)
PERC	0.70589 (!)	0.62259 (!)	0.61934 (!)	0.76421 (!)	0.58309 (!)	0.68313 (!)	0.66304 (!)
RF	**0.92529**	**0.93405**	**0.95525**	0.95569*	**0.94871**	**0.94952**	**0.94475**
SVM	0.81398 (!)	0.84597 (!)	0.81324 (!)	**0.98051**	0.77145 (!)	0.79959 (!)	0.83746 (!)

The mean training set performance across 7 trials and 6 datasets for each algorithm and metric combination are recorded in Table 4. Since bold-faced scores in almost all columns are produced by random forests, this model achieves the best score in almost every metric on training set, while SVM achieves the best score in specificity metric. Further observation suggests that all other models, including decision tree, KNN, logistic regression, perceptron classifier, and SVM, generally perform worse than random forests. Hence, random forests is clear winner in almost all scores on training set.

4. Discussion

Overall, this project is a replication of Caruana and Niculescu-Mizil (2006). According to CNM06, SVM and random forests have the best overall performance on testing set, while logistic regression and decision trees have the worst overall performance on testing set. Based on the analysis of the mean testing set performance across 7 trials and 6 datasets for each algorithm and metric combination, we reach a similar conclusion. SVM has the best score and random forest has the second best score in each metric. However, decision tree and logistic regression, the worst algorithms reported by CNM06, perform similarly well as SVM and random forests, while perceptron classifier, a model that is not used by CNM06, achieves the worst score in all metrics on test set.

According to CNM06, there is no a single algorithm that performs the best on all datasets. However, we reach a different conclusion. On almost all datasets in our project, SVM performs significantly better than all other algorithms, especially perceptron classifier. For OLYMPIC dataset that is slightly unbalanced with 66% samples being of negative-class, decision tree, one model with the worst overall performance reported by CNM06, is the most suitable algorithm. But its score is very low compared to the best scores on all other datasets. This suggests that OLYMPIC is likely the most difficult dataset for prediction. In addition, logistic regression performs the worst on this dataset. On the well-balanced SHROOMS dataset, all algorithms achieve nearly perfect scores. This suggests that the classification task of edible and poisonous created by SHROOMS is easy. Even though SVM has the best performance and perceptron classifier has the worst performance on almost all datasets, the asterisks around all scores in Mean column of Table 3 indicate that they have similar good overall performance as SVM.

After analyzing the mean training set performance across 7 trials and 6 datasets for each algorithm and metric combination, we find that random forest gives best performance, while all other algorithms perform significantly worse. However, the difference between random forest's scores on training and testing set is the largest among all models, which suggests that random forests is possibly more prone to overfitting or memorizing the labels of samples than all other models.

5. Bonus

Based on the instructions of final project, a single person is required to use 3 algorithms, 4 data sets, perform 5-fold cross validation for 5 trials, and measure the performance of each model by 3 metrics for a total of $3 * 4 * 5 = 60$ trials. Therefore, the minimum requirement for our 3-people group project is to use 6 algorithms, 6 data sets, perform 5-fold cross validation for 5 trials, and measure the performance of each model by 3 metrics, in total of $6 * 6 * 5 = 180$ trials.

To get extra credits, we perform 5-fold cross validation on each model and data set combination for 7 trials and use 6 metrics to evaluate performances of models, in total of $6 * 6 * 7 = 252$ trials. Based on the time complexity provided by (Kumar 2019), we analyze time complexity of 5 out of 6 algorithms explored in this project.

In the case of KNN, denote n as the number of training data samples, d as the dimension of the data, and k as the number of neighbors. In the process of training, KNN loops through

every observation, computes the distance d of all training data from the new data. Then, KNN selects k training samples whose distance from the new observation is one of those k smallest. Since the dimension of the data is d, distance computation of each training data requires $O(d)$ runtime, and distance computation of all training data needs $O(nd)$ runtime. Therefore, the overall selection of k closest training samples takes $O(k * n * d)$ runtime.

In the case of logistic regression, denote d as the dimension of the data. After logistic regression is fitted on the training set, a vector w of size d and a bias term b are obtained. Then, logistic regression computes $w^t * xi + b$ for each new data sample and compare result of operation with 0 to predict the class for the new observation. Since w is a vector of size d, computation of $w^t * xi$ takes $O(d)$ runtime. Therefore, given one new data, the runtime complexity of logistic regression is $O(d)$, and this algorithm is effective for predicting the labels of data with small dimension.

In the case of SVM, denote k as the number of support vectors and d as the dimension of the data. Given a new data, SVM selects k support vectors from all data and compute the distance between each pair as margin. Since the dimension of the data is d, distance computation of each support vector costs $O(d)$ runtime. Therefore, distance computation of k support vectors costs $O(k * n)$, which is also the runtime complexity of SVM.

In the case of decision tree, denote d as the maximum depth of the tree. Given a new observation, it starts from the root node of the tree, compares the attribute of the new data with the attribute of the root node, follows the corresponding branch based on the result of comparison, and obtain a label when the leaf node is reached. Therefore, for one new data sample, the runtime complexity of decision tree is $O(d)$.

In the case of random forests, denote k as the number of trees in the forest and d as the maximum depth of all trees in the forest. Given a new observation, random forests loops through each tree, starts from the root node of each tree, compares the attribute of the new data with the attribute of the root node, follows the corresponding branch based on the result of comparison, and obtain a label when the leaf of the tree is reached. The runtime complexity of reaching a leaf in a tree is approximately $O(d)$, therefore the runtime complexity of random forests algorithm is $O(k * d)$.

Acknowledgments

Acknowledgement We would like to acknowledge approval from Professor Fleischer and other COGS118A staff for our group final project. Also, the lecture notes of Professor Fleischer in week 9 and 10 clarify the goal of the project and details about how to conduct experiments, as well as help us write the codes. Thanks Professor Fleischer for addressing our specific questions about generating main tables and conducting two-sampled t-tests on scores during his office hours. In addition, we use the example final project paper on canvas as template for writing our final project paper. Lastly, thanks to classmates who contribute on Piazzas, provide pseudocode for running trials and calculating metric score on each data set, and help us solidify understanding of this project.

Appendix A. Training Performance

TRAINING table:

Appendix A — Training Set Performance By Metric Per Algorithm							
MODEL	ACC	PREC	REC	SPEC	F1	ROC	MEAN
DT	0.80141 (!)	0.82194 (!)	0.81676 (!)	0.90647 (!)	0.77365 (!)	0.80007 (!)	0.82005 (!)
KNN	0.78696 (!)	0.78117 (!)	0.87263 (!)	0.87924 (!)	0.86162 (!)	0.84251 (!)	0.83735 (!)
LOGREG	0.78626 (!)	0.77019 (!)	0.63927 (!)	0.88730 (!)	0.66742 (!)	0.74413 (!)	0.74910 (!)
PERC	0.70589 (!)	0.62259 (!)	0.61934 (!)	0.76421 (!)	0.58309 (!)	0.68313 (!)	0.66304 (!)
RF	**0.92529**	**0.93405**	**0.95525**	0.95569*	**0.94871**	**0.94952**	**0.94475**
SVM	0.81398 (!)	0.84597 (!)	0.81324 (!)	**0.98051**	0.77145 (!)	0.79959 (!)	0.83746 (!)

Appendix B. Trials

RAW trial scores:

Table 5 - Raw Testing Performance by Trial								
DATASET	MODEL	TRIAL	ACC	PREC	REC	SPEC	F1	ROC
CARDIO	DT	1	0.71809	0.76005	0.65176	0.80329	0.69803	0.71808
CARDIO	DT	2	0.71657	0.82352	0.75758	0.88833	0.71729	0.71658
CARDIO	DT	3	0.72398	0.75386	0.69187	0.80308	0.71147	0.72387
CARDIO	DT	4	0.72457	0.76149	0.68931	0.79707	0.71390	0.72452
CARDIO	DT	5	0.72688	0.77553	0.70600	0.80381	0.71186	0.72679
CARDIO	DT	6	0.73145	0.76172	0.67302	0.80442	0.72096	0.73148
CARDIO	DT	7	0.72263	0.77299	0.65914	0.81140	0.69508	0.72258
CARDIO	LOGREG	1	0.72665	0.75782	0.00000	1.00000	0.70887	0.72664
CARDIO	LOGREG	2	0.64778	0.64923	1.00000	0.66121	0.68156	0.64777
CARDIO	LOGREG	3	0.72677	0.74715	1.00000	0.76912	0.71448	0.72669
CARDIO	LOGREG	4	0.64978	0.65672	1.00000	0.67379	0.66971	0.64975
CARDIO	LOGREG	5	0.72562	0.74439	1.00000	1.00000	0.71311	0.72558
CARDIO	LOGREG	6	0.72531	0.75940	0.66029	0.00000	0.70639	0.72536
CARDIO	LOGREG	7	0.66548	0.67973	0.66062	1.00000	0.65118	0.66545
CARDIO	PERC	1	0.65883	0.82090	0.93094	0.64553	0.68236	0.65880
CARDIO	PERC	2	0.53218	0.82729	1.00000	0.97510	0.67475	0.53312
CARDIO	PERC	3	0.71365	0.76579	0.00000	0.76457	0.00000	0.71348
CARDIO	PERC	4	0.52314	0.80666	0.68645	0.41192	0.68710	0.52374
CARDIO	PERC	5	0.49968	0.82390	0.00000	0.96262	0.65575	0.50020
CARDIO	PERC	6	0.71418	0.76163	0.98162	0.80429	0.68612	0.71426
CARDIO	PERC	7	0.58442	0.76519	0.71229	0.94595	0.30889	0.58421
CARDIO	KNN	1	0.66245	0.68236	0.60843	0.74979	0.64285	0.66244
CARDIO	KNN	2	0.66438	0.67497	0.63137	0.69658	0.65244	0.66431
CARDIO	KNN	3	0.66226	0.68506	0.62905	0.71980	0.65026	0.66220
CARDIO	KNN	4	0.66166	0.67918	0.62051	0.71816	0.64169	0.66159
CARDIO	KNN	5	0.66351	0.67789	0.63001	0.71424	0.64677	0.66346
CARDIO	KNN	6	0.66058	0.70007	0.59343	0.75841	0.62816	0.66066
CARDIO	KNN	7	0.66515	0.69314	0.61616	0.73785	0.64599	0.66512

CARDIO	RF	1	0.72957	0.73791	0.71143	0.74635	0.72339	0.72957
CARDIO	RF	2	0.72749	0.73357	0.71274	0.74427	0.72114	0.72728
CARDIO	RF	3	0.72828	0.73543	0.71105	0.74234	0.72378	0.72862
CARDIO	RF	4	0.72497	0.72924	0.71587	0.73632	0.72102	0.72498
CARDIO	RF	5	0.72865	0.73117	0.72325	0.73597	0.72638	0.72875
CARDIO	RF	6	0.72925	0.74837	0.69233	0.76601	0.71875	0.72966
CARDIO	RF	7	0.72977	0.74422	0.69979	0.76118	0.72029	0.72956
CARDIO	SVM	1	0.72760	0.80839	0.80037	1.00000	0.71077	0.72759
CARDIO	SVM	2	0.72642	0.73343	0.99978	0.75793	0.72130	0.72638
CARDIO	SVM	3	0.72726	0.77307	0.99963	0.81371	0.71661	0.72719
CARDIO	SVM	4	0.72715	0.75205	0.99997	0.78674	0.71079	0.72709
CARDIO	SVM	5	0.72552	0.76599	1.00000	0.80202	0.71582	0.72549
CARDIO	SVM	6	0.72786	0.78436	0.81348	1.00000	0.70914	0.72792
CARDIO	SVM	7	0.72805	0.83155	1.00000	0.99517	0.71243	0.72801
AUS	DT	1	0.83213	0.79337	0.44963	0.98174	0.51483	0.68181
AUS	DT	2	0.82943	0.73455	0.49219	0.96503	0.52699	0.68752
AUS	DT	3	0.82901	0.78086	0.47615	0.97899	0.51492	0.69272
AUS	DT	4	0.82842	0.80148	0.45366	0.98384	0.53910	0.69482
AUS	DT	5	0.82998	0.80101	0.49774	0.98376	0.54223	0.70048
AUS	DT	6	0.82398	0.76804	0.46158	0.97608	0.54563	0.69947
AUS	DT	7	0.83320	0.73595	0.47661	0.96252	0.51588	0.68602
AUS	LOGREG	1	0.84479	0.82284	0.46654	0.99999	0.56829	0.70863
AUS	LOGREG	2	0.84299	0.79809	0.50414	0.99999	0.58453	0.72109
AUS	LOGREG	3	0.84141	0.82019	0.49362	0.99999	0.57718	0.71636
AUS	LOGREG	4	0.84359	0.79181	0.50059	0.99999	0.58354	0.72015
AUS	LOGREG	5	0.84331	0.79070	0.49625	0.99998	0.58231	0.71894
AUS	LOGREG	6	0.84312	0.82194	0.48505	0.99998	0.57932	0.71591
AUS	LOGREG	7	0.84334	0.80453	0.49158	0.99999	0.57897	0.71681
AUS	PERC	1	0.80807	0.58188	0.00000	1.00000	0.50402	0.50000
AUS	PERC	2	0.81950	0.66872	1.00000	0.95145	0.41619	0.60838
AUS	PERC	3	0.78071	0.00000	0.00000	1.00000	0.00000	0.50000
AUS	PERC	4	0.78109	0.00000	0.00000	1.00000	0.00000	0.55631
AUS	PERC	5	0.21900	0.00000	0.75037	0.00000	0.54480	0.75939
AUS	PERC	6	0.78067	0.21933	0.87019	1.00000	0.35975	0.75085
AUS	PERC	7	0.21911	0.21911	0.00000	0.00000	0.51373	0.72496
AUS	KNN	1	0.81552	0.88216	0.41007	0.99878	0.44179	0.64232
AUS	KNN	2	0.81718	0.89167	0.43042	0.99858	0.45701	0.65163
AUS	KNN	3	0.81568	0.87649	0.41921	0.99888	0.44811	0.64615
AUS	KNN	4	0.81916	0.84057	0.42653	0.99625	0.45148	0.64839
AUS	KNN	5	0.81830	0.87099	0.42495	0.99787	0.45237	0.64885
AUS	KNN	6	0.81605	0.85763	0.40718	0.99847	0.43948	0.64097
AUS	KNN	7	0.81760	0.85225	0.42393	0.99748	0.44789	0.64616
AUS	RF	1	0.84255	0.76881	0.40793	0.96640	0.53103	0.68684
AUS	RF	2	0.84438	0.73933	0.44907	0.95511	0.55823	0.70177

AUS	RF	3	0.84423	0.75736	0.43151	0.96270	0.54671	0.69487
AUS	RF	4	0.84347	0.75022	0.43076	0.96008	0.54487	0.69553
AUS	RF	5	0.84465	0.74430	0.44696	0.95886	0.55639	0.70246
AUS	RF	6	0.84290	0.75477	0.42046	0.96243	0.53800	0.69019
AUS	RF	7	0.84348	0.75829	0.42569	0.96277	0.54459	0.69357
AUS	SVM	1	0.84502	0.84243	0.45810	1.00000	0.55211	0.69777
AUS	SVM	2	0.84251	0.87929	0.47253	1.00000	0.57443	0.71308
AUS	SVM	3	0.84334	0.86059	0.46755	1.00000	0.56140	0.70595
AUS	SVM	4	0.84343	0.86999	0.47876	1.00000	0.56148	0.70763
AUS	SVM	5	0.84487	0.85467	0.48084	1.00000	0.56953	0.71130
AUS	SVM	6	0.84518	0.90272	0.46061	1.00000	0.55775	0.70347
AUS	SVM	7	0.84472	0.84410	0.49396	1.00000	0.57415	0.71517
AIRBNB	DT	1	0.82073	0.80618	0.83847	0.79844	0.82003	0.82073
AIRBNB	DT	2	0.82201	0.83645	0.84680	0.84410	0.82687	0.82201
AIRBNB	DT	3	0.82631	0.84716	0.83025	0.85922	0.82755	0.82631
AIRBNB	DT	4	0.81936	0.83375	0.83901	0.85885	0.82330	0.81936
AIRBNB	DT	5	0.81504	0.81238	0.82603	0.81029	0.82031	0.81503
AIRBNB	DT	6	0.81882	0.80718	0.83850	0.80788	0.82255	0.81879
AIRBNB	DT	7	0.82014	0.82644	0.83883	0.83006	0.81824	0.82013
AIRBNB	LOGREG	1	0.82071	0.82589	0.83204	0.00000	0.82271	0.82066
AIRBNB	LOGREG	2	0.82062	0.82166	0.83500	1.00000	0.82202	0.82062
AIRBNB	LOGREG	3	0.82347	0.85493	0.83608	1.00000	0.82584	0.82345
AIRBNB	LOGREG	4	0.82160	0.82999	0.83145	1.00000	0.82226	0.82160
AIRBNB	LOGREG	5	0.82365	0.81513	0.83483	1.00000	0.82552	0.82365
AIRBNB	LOGREG	6	0.82146	0.86859	0.83063	1.00000	0.82331	0.82145
AIRBNB	LOGREG	7	0.82235	0.81615	0.83200	0.80866	0.82396	0.82235
AIRBNB	PERC	1	0.73758	0.68082	0.62390	0.55793	0.70390	0.73757
AIRBNB	PERC	2	0.79515	0.74855	0.77767	0.79936	0.79149	0.79515
AIRBNB	PERC	3	0.76965	0.59193	0.96892	0.74385	0.75521	0.76948
AIRBNB	PERC	4	0.75435	0.71809	0.59218	0.43524	0.70678	0.75433
AIRBNB	PERC	5	0.50028	0.81346	0.74028	0.81020	0.00000	0.50000
AIRBNB	PERC	6	0.75095	0.70179	0.93527	0.64961	0.77381	0.75079
AIRBNB	PERC	7	0.70685	0.69515	0.35803	0.55885	0.48348	0.70686
AIRBNB	KNN	1	0.82570	0.82529	0.84161	0.81976	0.82836	0.82570
AIRBNB	KNN	2	0.82807	0.82268	0.85714	0.81932	0.83119	0.82807
AIRBNB	KNN	3	0.82490	0.81430	0.85041	0.80859	0.82563	0.82487
AIRBNB	KNN	4	0.82834	0.83127	0.82607	0.83229	0.82756	0.82834
AIRBNB	KNN	5	0.83053	0.82488	0.86606	0.82049	0.83187	0.83053
AIRBNB	KNN	6	0.82752	0.82172	0.85656	0.81892	0.83062	0.82749
AIRBNB	KNN	7	0.82775	0.82634	0.84896	0.82747	0.82868	0.82809
AIRBNB	RF	1	0.83146	0.83013	0.83386	0.83234	0.83194	0.83139
AIRBNB	RF	2	0.83399	0.82542	0.85281	0.81895	0.83647	0.83342
AIRBNB	RF	3	0.83429	0.83008	0.84036	0.82692	0.83521	0.83389
AIRBNB	RF	4	0.83199	0.83792	0.82133	0.84036	0.83060	0.83139

AIRBNB	RF	5	0.83420	0.82908	0.84746	0.82819	0.83663	0.83459
AIRBNB	RF	6	0.83089	0.82575	0.84105	0.82216	0.83279	0.83056
AIRBNB	RF	7	0.83356	0.83360	0.83254	0.83512	0.83349	0.83351
AIRBNB	SVM	1	0.82470	0.86660	0.83851	1.00000	0.82685	0.82470
AIRBNB	SVM	2	0.82839	0.85251	0.84466	1.00000	0.83112	0.82839
AIRBNB	SVM	3	0.82693	0.77222	0.84841	0.99909	0.83074	0.82690
AIRBNB	SVM	4	0.82189	0.87583	0.83901	1.00000	0.82353	0.82265
AIRBNB	SVM	5	0.82902	0.86145	0.92788	1.00000	0.83375	0.82904
AIRBNB	SVM	6	0.82253	0.88128	0.84460	1.00000	0.82659	0.82250
AIRBNB	SVM	7	0.82531	0.86940	0.98226	1.00000	0.82760	0.82532
SHROOMS	DT	1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	DT	2	0.99840	1.00000	0.99670	1.00000	0.99835	0.99835
SHROOMS	DT	3	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	DT	4	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	DT	5	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	DT	6	1.00000	1.00000	1.00000	1.00000	0.99935	0.99937
SHROOMS	DT	7	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	LOGREG	1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	LOGREG	2	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	LOGREG	3	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	LOGREG	4	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	LOGREG	5	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	LOGREG	6	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	LOGREG	7	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	PERC	1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	PERC	2	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	PERC	3	0.99968	1.00000	0.99932	1.00000	0.99966	0.99966
SHROOMS	PERC	4	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	PERC	5	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	PERC	6	0.99936	1.00000	0.99870	1.00000	0.99935	0.99935
SHROOMS	PERC	7	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	KNN	1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	KNN	2	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	KNN	3	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	KNN	4	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	KNN	5	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	KNN	6	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	KNN	7	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	RF	1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	RF	2	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	RF	3	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	RF	4	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	RF	5	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	RF	6	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000

SHROOMS	RF	7	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	SVM	1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	SVM	2	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	SVM	3	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	SVM	4	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	SVM	5	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	SVM	6	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
SHROOMS	SVM	7	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
CHESS	DT	1	0.65188	0.58968	0.88380	0.47453	0.68898	0.65182
CHESS	DT	2	0.64690	0.62949	0.80551	0.61304	0.66232	0.64678
CHESS	DT	3	0.64949	0.64834	0.85617	0.86713	0.67965	0.64985
CHESS	DT	4	0.63740	0.63586	0.66161	0.59899	0.63314	0.63758
CHESS	DT	5	0.64836	0.63228	0.84960	0.64893	0.68989	0.64945
CHESS	DT	6	0.64650	0.66601	0.61390	0.65516	0.62494	0.64624
CHESS	DT	7	0.64730	0.66122	0.91095	0.90113	0.68950	0.64758
CHESS	LOGREG	1	0.67094	0.74896	0.68749	0.90398	0.67654	0.67085
CHESS	LOGREG	2	0.66928	0.66247	0.69489	0.74039	0.67810	0.66931
CHESS	LOGREG	3	0.66676	0.65953	0.71887	0.65061	0.68639	0.66688
CHESS	LOGREG	4	0.66749	0.66166	0.72817	0.72263	0.68602	0.66762
CHESS	LOGREG	5	0.66961	0.65114	0.00000	0.63251	0.68160	0.67010
CHESS	LOGREG	6	0.66695	0.65747	0.71406	0.65040	0.68256	0.66706
CHESS	LOGREG	7	0.66629	0.67247	0.70811	1.00000	0.67922	0.66638
CHESS	PERC	1	0.60665	0.52409	0.46001	1.00000	0.65140	0.60648
CHESS	PERC	2	0.62618	0.63189	0.93797	0.00000	0.68441	0.62644
CHESS	PERC	3	0.58175	0.56166	0.73140	0.37705	0.65147	0.58228
CHESS	PERC	4	0.57411	0.59090	0.76238	0.51842	0.64107	0.57454
CHESS	PERC	5	0.59849	0.56413	0.43991	0.43253	0.68484	0.60154
CHESS	PERC	6	0.64843	0.64856	0.00000	0.65331	0.64152	0.64840
CHESS	PERC	7	0.63534	0.54047	0.59510	0.36819	0.61955	0.63526
CHESS	KNN	1	0.62611	0.61993	0.66746	0.59769	0.64166	0.62607
CHESS	KNN	2	0.62585	0.61057	0.66534	0.59647	0.63954	0.62594
CHESS	KNN	3	0.62405	0.61086	0.70288	0.57239	0.64182	0.62424
CHESS	KNN	4	0.62830	0.61083	0.71991	0.58813	0.65803	0.62846
CHESS	KNN	5	0.62731	0.60537	0.73922	0.58153	0.65259	0.62817
CHESS	KNN	6	0.62923	0.61026	0.73126	0.56190	0.66344	0.62959
CHESS	KNN	7	0.63069	0.61419	0.71330	0.59231	0.65706	0.63082
CHESS	RF	1	0.66158	0.65750	0.66852	0.65075	0.66575	0.66315
CHESS	RF	2	0.66144	0.65667	0.67306	0.64895	0.66627	0.66366
CHESS	RF	3	0.66184	0.65413	0.68542	0.64068	0.66975	0.66125
CHESS	RF	4	0.66164	0.64878	0.70061	0.62656	0.67664	0.66272
CHESS	RF	5	0.66470	0.64676	0.72392	0.62094	0.68341	0.66623
CHESS	RF	6	0.66649	0.65492	0.70325	0.63770	0.67961	0.66596
CHESS	RF	7	0.66104	0.65200	0.68455	0.63685	0.66706	0.65863
CHESS	SVM	1	0.66788	0.70263	0.68152	1.00000	0.67616	0.66785

CHESS	SVM	2	0.66509	0.66572	0.68384	1.00000	0.66684	0.66511
CHESS	SVM	3	0.66330	0.78625	0.69941	1.00000	0.67425	0.66343
CHESS	SVM	4	0.66377	0.65257	0.76771	1.00000	0.68640	0.66393
CHESS	SVM	5	0.66343	0.63737	1.00000	0.61805	0.68533	0.66427
CHESS	SVM	6	0.66403	0.65294	1.00000	0.69788	0.69513	0.66426
CHESS	SVM	7	0.65381	0.66300	0.69919	1.00000	0.66300	0.65387
OLYMPIC	DT	1	0.67666	0.82206	0.37390	0.99699	0.41042	0.57293
OLYMPIC	DT	2	0.68103	0.82807	0.40278	0.99706	0.41263	0.56885
OLYMPIC	DT	3	0.68039	0.81787	0.40449	0.99682	0.42342	0.57597
OLYMPIC	DT	4	0.68325	0.81250	0.40522	0.99678	0.41945	0.56955
OLYMPIC	DT	5	0.67257	0.81522	0.37431	0.99694	0.39940	0.56604
OLYMPIC	DT	6	0.68409	0.82230	0.38593	0.99696	0.41309	0.57572
OLYMPIC	DT	7	0.68067	0.64309	0.39753	0.99724	0.41240	0.56936
OLYMPIC	LOGREG	1	0.66066	0.40641	0.05652	1.00000	0.10010	0.50947
OLYMPIC	LOGREG	2	0.66300	0.41225	0.12479	1.00000	0.19279	0.51804
OLYMPIC	LOGREG	3	0.66217	0.58790	0.11967	1.00000	0.18669	0.51854
OLYMPIC	LOGREG	4	0.67023	0.59110	0.17178	1.00000	0.25209	0.53776
OLYMPIC	LOGREG	5	0.66113	0.52326	0.07090	1.00000	0.12360	0.51566
OLYMPIC	LOGREG	6	0.66630	0.40647	0.10568	1.00000	0.16733	0.51341
OLYMPIC	LOGREG	7	0.66221	0.50777	0.09360	1.00000	0.15700	0.52123
OLYMPIC	PERC	1	0.66066	0.39159	0.04447	1.00000	0.07936	0.50000
OLYMPIC	PERC	2	0.57353	0.34082	0.70599	0.72716	0.47684	0.53401
OLYMPIC	PERC	3	0.66217	0.43166	0.54532	1.00000	0.43434	0.51785
OLYMPIC	PERC	4	0.66503	0.34040	0.31049	1.00000	0.32476	0.50372
OLYMPIC	PERC	5	0.65673	0.32252	0.47557	0.88671	0.38786	0.52240
OLYMPIC	PERC	6	0.66630	0.36953	0.36261	1.00000	0.36811	0.50351
OLYMPIC	PERC	7	0.66185	0.31914	0.54281	1.00000	0.43106	0.50432
OLYMPIC	KNN	1	0.67491	0.58378	0.44517	0.98119	0.45162	0.58743
OLYMPIC	KNN	2	0.67055	0.56325	0.49045	0.97203	0.47842	0.59826
OLYMPIC	KNN	3	0.67404	0.55114	0.46397	0.96492	0.45992	0.59529
OLYMPIC	KNN	4	0.67678	0.54758	0.49176	0.95820	0.47287	0.59850
OLYMPIC	KNN	5	0.67142	0.61209	0.46924	0.98414	0.46410	0.59292
OLYMPIC	KNN	6	0.67769	0.54653	0.49482	0.95935	0.47336	0.59820
OLYMPIC	KNN	7	0.67793	0.56190	0.47528	0.97198	0.46376	0.59090
OLYMPIC	RF	1	0.68691	0.63128	0.31387	0.94446	0.39833	0.59089
OLYMPIC	RF	2	0.69644	0.62235	0.36295	0.92327	0.43108	0.60190
OLYMPIC	RF	3	0.69255	0.62234	0.33972	0.92809	0.42015	0.59832
OLYMPIC	RF	4	0.69747	0.61157	0.37012	0.91383	0.44198	0.60877
OLYMPIC	RF	5	0.69100	0.61910	0.33177	0.93519	0.41564	0.59586
OLYMPIC	RF	6	0.68957	0.58210	0.36606	0.91006	0.43347	0.59971
OLYMPIC	RF	7	0.69231	0.61625	0.32954	0.93424	0.40956	0.59021
OLYMPIC	SVM	1	0.68615	0.71329	0.35974	1.00000	0.42656	0.59590
OLYMPIC	SVM	2	0.68667	0.61489	0.43967	1.00000	0.47435	0.61459
OLYMPIC	SVM	3	0.68738	0.64320	0.39873	1.00000	0.45040	0.60282

OLYMPIC	SVM	4	0.68298	0.68966	0.43106	1.00000	0.47074	0.61117
OLYMPIC	SVM	5	0.68329	0.69463	0.39072	1.00000	0.44572	0.60247
OLYMPIC	SVM	6	0.68135	0.61424	0.44079	1.00000	0.46957	0.60876
OLYMPIC	SVM	7	0.69076	0.67808	0.39859	1.00000	0.43930	0.59300

Appendix C. P-values

TESTING table supplements

Table 2 Supplemental — P-Values							
MODEL	ACC_P	PREC_P	REC_P	SPEC_P	F1_P	ROC_P	MEAN_P
DT	0.69659	0.80022	0.31565	0.00201	0.68746	0.67478	0.55129
KNN	0.35414	0.11755	0.15363	0.00014	0.41693	0.35134	0.27372
LOGREG	0.66722	0.08994	0.07496	0.05174	0.31664	0.56528	0.32717
PERC	0.00778	0.00011	0.05009	0.00008	0.01741	0.02482	0.01377
RF	**1.00000**	0.12536	0.12397	0.00001	0.85879	0.95038	0.39450
SVM	0.92429	**1.00000**	**1.00000**	**1.00000**	**1.00000**	**1.00000**	**1.00000**

Table 3 Supplemental — P-Values							
MODEL	CARDIO_P	AUS_P	CHESS_P	SHROOMS_P	AIRBNB_P	OLYMPIC_P	MEAN_P
DT	0.00242	0.43995	0.04005	0.02816	0.00014	**1.00000**	0.70374
KNN	0.00000	0.33182	0.00001	**1.00000**	0.00049	0.69982	0.51985
LOGREG	0.02332	0.94231	0.05363	**1.00000**	0.09783	0.01146	0.54549
PERC	0.00092	0.00010	0.00003	0.00762	0.00000	0.04638	0.16177
RF	0.00012	0.34639	0.00114	**1.00000**	0.00117	0.35473	0.61621
SVM	**1.00000**	**1.00000**	**1.00000**	**1.00000**	**1.00000**	0.90737	**1.00000**

TRAINING table supplements:

Appendix A Supplemental — P-Values							
MODEL	ACC	PREC	REC	SPEC	F1	ROC	MEAN
DT	8.29E-08	1.70E-08	1.85E-10	3.22E-04	5.99E-12	1.36E-11	7.16E-05
KNN	2.04E-08	6.79E-09	5.38E-04	1.76E-05	6.43E-04	4.33E-05	1.59E-04
LOGREG	1.85E-08	3.37E-10	3.70E-08	1.52E-02	3.85E-10	1.25E-12	3.34E-04
PERC	3.26E-10	4.45E-10	6.36E-08	1.54E-05	6.63E-12	6.72E-15	1.40E-06
RF	**1.00000**	**1.00000**	**1.00000**	0.05456	**1.00000**	**1.00000**	**1.00000**
SVM	2.84E-07	1.02E-05	4.51E-06	**1.00000**	8.88E-12	7.16E-11	5.75E-03

CODE is embedded after References.

References

Caruana, Rich and Alexandru Niculescu-Mizil (2006). "An Empirical Comparison of Supervised Learning Algorithms". In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, pp. 161–168. ISBN: 1595933832. DOI: 10.1145/1143844.1143865. URL: https://doi.org/10.1145/1143844.1143865.

Henery, R. J. and C. C. Taylor (1992). "StatLog: An Evaluation of Machine Learning and Statistical Algorithms". In: *Computational Statistics*. Ed. by Yadolah Dodge and Joe Whittaker. Heidelberg: Physica-Verlag HD, pp. 157–162. ISBN: 978-3-662-26811-7.

Huang, Shujun et al. (2018). "Applications of support vector machine (SVM) learning in cancer genomics". In: *Cancer Genomics-Proteomics* 15.1, pp. 41–51.

Kumar, Paritosh (Dec. 2019). *Time Complexity of ML Models*. URL: https://medium.com/@paritoshkumar_5426/time-complexity-of-ml-models-4ec39fad2770.

Cogs118A Final Project Code

Import Packages

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

from sklearn.base import clone
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV

# models
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import Perceptron
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier

# metrics
from sklearn.metrics import make_scorer
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import roc_auc_score
import warnings
warnings.filterwarnings('ignore')
```

Clean Chess Dataset

The dataset is from https://www.kaggle.com/datasnaek/chess (https://www.kaggle.com/datasnaek/chess), and contains 20058 games of chess. We predict whether white wins the match or not. White winning is the positive label, while black winning or a draw is the negative label. We use 8 features for prediction. The detailed description of features are specified below:

- rated: whether or not the game was a rated game
- created_at: time at game start
- turns: how many turns the game took
- victory_status: whether the game ended with a timeout, resignation, or checkmate
- white_rating: rating of the white player
- black_rating: rating of the black player
- opening_eco: code corresponding to the opening moves played
- opening_ply: the length of the opening

In [4]:
```
# https://www.kaggle.com/datasnaek/chess
chess = pd.read_csv('Data/games.csv')
chess.head()
```

Out[4]:

	id	rated	created_at	last_move_at	turns	victory_status	winner	increment_code
0	TZJHLljE	False	1.504210e+12	1.504210e+12	13	outoftime	white	15+2
1	l1NXvwaE	True	1.504130e+12	1.504130e+12	16	resign	black	5+10
2	mIICvQHh	True	1.504130e+12	1.504130e+12	61	mate	white	5+10
3	kWKvrqYL	True	1.504110e+12	1.504110e+12	61	mate	white	20+0
4	9tXo1AUZ	True	1.504030e+12	1.504030e+12	95	mate	white	30+3

In [5]:
```
# standarize numerical columns
chess_num_cols = ['created_at', 'last_move_at', 'turns',
                  'white_rating', 'black_rating', 'opening_ply']
for num_col in chess_num_cols:
    chess[num_col] = (chess[num_col] - chess[num_col].mean()) /chess[num_col].
std()

# create a new column called winner_white that is true
# if white wins, false otherwise
chess['winner_white'] = chess['winner'] == 'white'
chess = chess[['rated', 'created_at', 'last_move_at',
               'turns', 'victory_status',
                     'white_rating', 'black_rating',
               'opening_eco', 'opening_ply', 'winner_white']]
```

In [6]:
```
chess_df_X = chess.drop(columns=['winner_white'])
chess_df_y = chess['winner_white']
```

In [7]:
```
# one-hot encode categorical columns
chess_X_cat_col = ['rated', 'victory_status', 'opening_eco']
chess_X = pd.get_dummies(columns=chess_X_cat_col, data=chess_df_X)

chess_y = chess_df_y.replace({True: 1, False: 0})
```

In [8]:
```
# 50.1396% negative instances, 49.8604% positive instances
chess_y.value_counts(normalize = True)
```

Out[8]:
```
0    0.501396
1    0.498604
Name: winner_white, dtype: float64
```

Clean Mushrooms Dataset

The dataset is from https://www.kaggle.com/uciml/mushroom-classification (https://www.kaggle.com/uciml/mushroom-classification), and contains 8124 hypothetical descriptions of mushrooms from the UCI Machine Learning Repository. We predict whether the mushroom is edible or poisonous. Being edible is the negative label, while being poisonous is the positive label. We use 21 features for this prediction, which are all categorical data. The detailed description of features are specified below:

- class: edible(e), poisonous(p)
- cap-shape: bell(b), conical(c), convex(x), flat(f), knobbed(k), sunken(s)
- cap-surface: fibrous(f), grooves(g), scaly(y), smooth(s)
- cap-color: brown(n), buff(b), cinnamon(c), gray(g), green(r), pink(p), purple(u), red(e), white(w), yellow(y)
- bruises: bruises(t), no(f)
- odor: almond(a), anise(l), creosote(c), fishy(y), foul(f), musty(m), none(n), pungent(p), spicy(s)
- gill-attachment: attached(a), descending(d), free(f), notched(n)
- gill-spacing: close(c), crowded(w), distant(d)
- gill-size: broad(b), narrow(n)
- gill-color: black(k), brown(n), buff(b), chocolate(h), gray(g), green(r), orange(o), pink(p), purple(u), red(e), white(w), yellow(y)
- stalk-shape: enlarging(e), tapering(t)
- stalk-root: bulbous(b), club(c), cup(u), equal(e), rhizomorphs(z), rooted(r), missing(?)
- stalk-surface-above-ring: fibrous(f), scaly(y), silky(k), smooth(s)
- stalk-surface-below-ring: fibrous(f), scaly(y), silky(k), smooth(s)
- stalk-color-above-ring: brown(n), buff(b), cinnamon(c), gray(g), orange(o), pink(p), red(e), white(w), yellow(y)
- stalk-color-below-ring: brown(n), buff(b), cinnamon(c), gray(g), orange(o), pink(p), red(e), white(w), yellow(y)
- veil-type: partial(p), universal(u)
- veil-color: brown(n), orange(o), white(w), yellow(y)
- ring-number: none(n), one(o), two(t)
- ring-type: cobwebby(c), evanescent(e), flaring(f), large(l), none(n), pendant(p), sheathing(s), zone(z)
- spore-print-color: black(k), brown(n), buff(b), chocolate(h), green(r), orange(o), purple(u), white(w), yellow(y)
- population: abundant(a), clustered(c), numerous(n), scattered(s), several(v), solitary(y)
- habitat: grasses(g), leaves(l), meadows(m), paths(p), urban(u), waste(w), woods(d)

In [9]:
```
# https://www.kaggle.com/uciml/mushroom-classification
shrooms = pd.read_csv('Data/mushrooms.csv')
shrooms.head()
```

Out[9]:

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	a
0	p	x	s	n	t	p	f	c	n	k	...	s	
1	e	x	s	y	t	a	f	c	b	k	...	s	
2	e	b	s	w	t	l	f	c	b	n	...	s	
3	p	x	y	w	t	p	f	c	n	n	...	s	
4	e	x	s	g	f	n	f	w	b	k	...	s	

5 rows × 23 columns

In [10]:
```
shrooms_df_X = shrooms.drop(columns=['class'])
shrooms_df_y = shrooms['class']
```

In [11]:
```
shrooms_X = pd.get_dummies(data=shrooms_df_X)
shrooms_y = shrooms_df_y.replace({'e': 0, 'p': 1})
```

In [12]:
```
# 51.8% negative instances, 48.2% positive instances
shrooms_y.value_counts(normalize = True)
```

Out[12]:
```
0    0.517971
1    0.482029
Name: class, dtype: float64
```

Clean Cardiovascular Disease Dataset

Retrieved from the kaggle site https://www.kaggle.com/sulianova/cardiovascular-disease-dataset (https://www.kaggle.com/sulianova/cardiovascular-disease-dataset), this cardio dataset has 70000 samples and 12 variables, which were collected at the moment of medical examination. It contains a target variable that indicates the presence or absence of cardiovascular disease, as well as 11 features that might be associated with the presence of cardiovascular disease, such as age, gender, and blood pressure. There are 3 types of 11 input features:

- objective feature: factual information
- examination feature: results of medical examination
- subjective feature: information given by the patient

A more detailed description of 11 features are shown below:

- age: objective feature, int (days)
- height: objective feature, int (cm)
- weight: objective feature, float (kg)
- gender: objective feature, categorical code, 1: male, 2:female
- ap_hi: systolic blood pressure, examination feature, int
- ap_lo: diastolic blood pressure, examination feature, int
- cholesterol: examination feature, categorical code, 1: normal, 2: above normal, 3: well above normal
- gluc: glucose, examination feature, categorical code, 1: normal, 2: above normal, 3: well above normal
- smoke: subjective feature, binary, 0: do not smoke, 1: smoke
- alco: alcohol intake, subjective feature, binary, 0: do not drink alcohol, 1: drink alcohol
- active: physical activity, subjective feature, binary, 0: not physically active, 1: physically active

A detailed description of the target variable is shown below:

- cardio: presence or absence of cardiovascular disease, binary, 0: disease not present, 1: disease present

For this dataset, we want use those 11 input features and apply machine learning algorithms to predict whether a person has cardiovascular disease or not.

In [13]:
```
# load the cardio dataset
cardio = pd.read_csv('data/cardio.csv', delimiter = ';')
cardio.head()
```

Out[13]:

	id	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	card
0	0	18393	2	168	62.0	110	80	1	1	0	0	1	
1	1	20228	1	156	85.0	140	90	3	1	0	0	1	
2	2	18857	1	165	64.0	130	70	3	1	0	0	0	
3	3	17623	2	169	82.0	150	100	1	1	0	0	1	
4	4	17474	1	156	56.0	100	60	1	1	0	0	0	

In [14]:
```
# no missing values in cardio dataset
cardio.isnull().sum()
```

Out[14]:
```
id             0
age            0
gender         0
height         0
weight         0
ap_hi          0
ap_lo          0
cholesterol    0
gluc           0
smoke          0
alco           0
active         0
cardio         0
dtype: int64
```

In [15]:
```
# drop unnecessary column "id"
cardio = cardio.drop(columns = ['id'])
# convert age in days to age in years
cardio['age'] = cardio['age'].apply(lambda x: int(x/365))
```

In [16]:
```
# one hot encoding categorical input features stored in cate_cols
cardio_cate_cols = ['gender', 'cholesterol', 'gluc', 'smoke', 'alco', 'active'
]
cardio = pd.get_dummies(columns = cardio_cate_cols, data = cardio)
```

In [17]:
```
# scale numerical attributes to 0 mean 1 std
cardio_num_cols = ['age', 'height', 'weight', 'ap_hi', 'ap_lo']
for num_col in cardio_num_cols:
    cardio[num_col] = (cardio[num_col] - cardio[num_col].mean()) / cardio[num_
col].std()
```

In [18]:
```
# 50.03% negative labels, 49.97% positive labels
cardio['cardio'].value_counts(normalize = True)
```

Out[18]:
```
0    0.5003
1    0.4997
Name: cardio, dtype: float64
```

In [19]:
```
# split the cardio dataset into input features and labels
cardio_X = cardio.drop(columns=['cardio']) # input features
cardio_y = cardio['cardio'] # true lables
```

Clean Rain in Australia Dataset

Retrieved from the kaggle site https://www.kaggle.com/jsphyg/weather-dataset-rattle-package (https://www.kaggle.com/jsphyg/weather-dataset-rattle-package), this Rain in Australia dataset contains about 10 years of daily weather observations from many locations across Australia. There are 145460 samples and 23 variables in this dataset. It contains a target variable that indicates whether it rained the next day, as well as 22 features that might be associated with the target variable, such as minimum temperature, maximum temperature, rainfall of the day.

A more detailed description of 22 features are shown below:

- Date: the date of observation
- Location: the common name of the location of the weather station
- MinTemp: the minimum temperature in degrees celsius
- MaxTemp: the maximum temperature in degrees celsius
- Rainfall: the amount of rainfall recorded for the day in mm
- Evaporation: the so-called Class A pan evaporation (mm) in the 24 hours to 9am
- Sunshine: the number of hours of bright sunshine in the day
- WindGustDir: the direction of the strongest wind gust in the 24 hours to midnight
- WindGustSpeed: the speed (km/h) of the strongest wind gust in the 24 hours to midnight
- WindDir9am: direction of the wind at 9am
- WindDir3pm: direction of the wind at 3pm
- WindSpeed9am: wind speed (km/hr) averaged over 10 minutes prior to 9am
- WindSpeed3pm: wind speed (km/hr) averaged over 10 minutes prior to 3pm
- Humidity9am: humidity (percent) at 9am
- Humidity3pm: humidity (percent) at 3pm
- Pressure9am: atmospheric pressure (hpa) reduced to mean sea level at 9am
- Pressure3pm: atmospheric pressure (hpa) reduced to mean sea level at 3pm
- Cloud9am: fraction of sky obscured by cloud at 9am. This is measured in "oktas", which are a unit of eigths. It records how many eigths of the sky are obscured by cloud. A 0 measure indicates completely clear sky whilst an 8 indicates that it is completely overcast
- Cloud3pm: fraction of sky obscured by cloud (in "oktas": eighths) at 3pm. See Cload9am for a description of the values
- Temp9am: temperature (degrees C) at 9am
- Temp3pm: temperature (degrees C) at 3pm
- RainToday: whether the precipitation (mm) in the 24 hours to 9am exceeded 1mm, Yes: the precipitation exceeded 1mm, No: it did not exceed 1mm

A detailed description of the target variable is shown below:

- RainTomorrow: whether amount of next day rain exceeded 1mm, Yes: next day precipitation exceeded 1mm, No: it did not exceed 1mm

For this dataset, we want use those 22 input features and apply machine learning algorithms to predict whether it rained the next day or not.

Data source: http://www.bom.gov.au/climate/dwo/ (http://www.bom.gov.au/climate/dwo/) and http://www.bom.gov.au/climate/data (http://www.bom.gov.au/climate/data).

In [20]:
```
# load Australian rain dataset
aus = pd.read_csv('Data/weatherAUS.csv')
aus.head()
```

Out[20]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGus
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	

5 rows × 23 columns

In [21]:
```
# display the number of missing values in each column
aus.isnull().sum()
```

Out[21]:
```
Date                0
Location            0
MinTemp          1485
MaxTemp          1261
Rainfall         3261
Evaporation     62790
Sunshine        69835
WindGustDir     10326
WindGustSpeed   10263
WindDir9am      10566
WindDir3pm       4228
WindSpeed9am     1767
WindSpeed3pm     3062
Humidity9am      2654
Humidity3pm      4507
Pressure9am     15065
Pressure3pm     15028
Cloud9am        55888
Cloud3pm        59358
Temp9am          1767
Temp3pm          3609
RainToday        3261
RainTomorrow     3267
dtype: int64
```

In [22]:
```
# fill missing values in categorical columns with the mode
aus_cate_cols = aus.dtypes.index[aus.dtypes == "object"].tolist()
for cate_col in aus_cate_cols:
    aus[cate_col] = aus[cate_col].fillna(aus[cate_col].mode()[0])
```

In [23]:
```
# fill missing values in numerical columns with the mean
aus_num_cols = aus.dtypes.index[aus.dtypes == "float64"].tolist()
for num_col in aus_num_cols:
    aus[num_col] = aus[num_col].fillna(aus[num_col].mean())
    # scale numerical attributes to 0 mean 1 std
    aus[num_col] = (aus[num_col] - aus[num_col].mean()) / aus[num_col].std()
```

In [24]:
```
# all missing values are filled
aus.isnull().sum()
```

Out[24]:
```
Date             0
Location         0
MinTemp          0
MaxTemp          0
Rainfall         0
Evaporation      0
Sunshine         0
WindGustDir      0
WindGustSpeed    0
WindDir9am       0
WindDir3pm       0
WindSpeed9am     0
WindSpeed3pm     0
Humidity9am      0
Humidity3pm      0
Pressure9am      0
Pressure3pm      0
Cloud9am         0
Cloud3pm         0
Temp9am          0
Temp3pm          0
RainToday        0
RainTomorrow     0
dtype: int64
```

In [25]:
```
# split the date of each observation into year, month, and day
splitted_date = aus['Date'].str.split('-')

# create 'Year', 'Month', 'Day' columns using splitted results of the date
aus['Year'] = splitted_date.str[0].astype(int)
aus['Month'] = splitted_date.str[1].astype(int)
aus['Day'] = splitted_date.str[2].astype(int)

# drop original 'Date' column
aus = aus.drop(columns = ['Date'])
```

In [26]:
```
# use 0 and 1 to indicate whether it rained or not
# 0: it rained, 1: it did not rain
aus['RainToday'] = aus['RainToday'].replace({'No': 0, 'Yes': 1})
aus['RainTomorrow'] = aus['RainTomorrow'].replace({'No': 0, 'Yes': 1})
```

In [27]:
```
# one hot encoding all categorical columns
cate_cols = aus.dtypes.index[aus.dtypes == "object"].tolist()
aus = pd.get_dummies(columns = cate_cols, data = aus)
```

In [28]:
```
# 78.0854% negative labels, 21.9146% positive labels
aus['RainTomorrow'].value_counts(normalize = True)
```

Out[28]:
```
0    0.780854
1    0.219146
Name: RainTomorrow, dtype: float64
```

In [29]:
```
# split the rain dataset into input features and labels
aus_X = aus.drop(columns=['RainTomorrow']) # input features
aus_y = aus['RainTomorrow'] # true lables
```

Clean AirBnB Dataset

The dataset is from https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data?select=AB_NYC_2019.csv (https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data?select=AB_NYC_2019.csv), and contains data from AirBnB listing and metrics in New York City, New York for the year 2019. There are 47900 unique values and 8 predictive variables in this dataset, in which we will be using different features to predict whether an AirBnB pricing is expensive or not. This makes the price variable the target variable of our data.

A more detailed description of 7 features are shown below:

- neighbourhood_group: location, str
- latitude: latitude coordinates, int
- longitude: longitude coordinates, int
- room_type: listing space type, str
- minimum_nights: amount of nights minimum, int
- number_of_reviews: number of reviews, int
- availability_365: number of days when listing is available for booking, int

A detailed description of the target variable is shown below:

- price: price in USD, int

For this dataset, we want use these 7 input features and apply machine learning algorithms to predict whether the AirBnb price is expensive or not.

In [30]:
```
# load the AirBnb dataset
airbnb = pd.read_csv('data/AB_NYC_2019.csv')
airbnb.head()
```

Out[30]:

	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude
0	2539	Clean & quiet apt home by the park	2787	John	Brooklyn	Kensington	40.64749
1	2595	Skylit Midtown Castle	2845	Jennifer	Manhattan	Midtown	40.75362
2	3647	THE VILLAGE OF HARLEM....NEW YORK !	4632	Elisabeth	Manhattan	Harlem	40.80902
3	3831	Cozy Entire Floor of Brownstone	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514
4	5022	Entire Apt: Spacious Studio/Loft by central park	7192	Laura	Manhattan	East Harlem	40.79851

In [31]:
```
# We will be looking at these features of the AirBnB for our predictive model
airbnb = airbnb[['neighbourhood_group','latitude','longitude','room_type', 'price',
                 'minimum_nights','number_of_reviews','availability_365']]
```

In [32]:
```
# The price median will be the threshold for our expensive classifier
airbnb['price'].median()
```

Out[32]: 106.0

In [33]:
```
# Creating a new variable called 'is_expensive' that
# labels prices greater than the median as True,
# lesser than the median as False
airbnb = airbnb.assign(
    is_expensive = airbnb.get('price') > 106.0
)
airbnb.head()
```

Out[33]:

	neighbourhood_group	latitude	longitude	room_type	price	minimum_nights	number_of_rev
0	Brooklyn	40.64749	-73.97237	Private room	149	1	
1	Manhattan	40.75362	-73.98377	Entire home/apt	225	1	
2	Manhattan	40.80902	-73.94190	Private room	150	3	
3	Brooklyn	40.68514	-73.95976	Entire home/apt	89	1	
4	Manhattan	40.79851	-73.94399	Entire home/apt	80	10	

In [34]:
```
# Drop the price column
airbnb = airbnb.drop(columns = ['price'])
```

In [35]:
```
# Scale numerical attributes to 0 mean 1 std
airbnb_num_cols = ['latitude', 'longitude', 'minimum_nights',
                   'number_of_reviews','availability_365']
for num_col in airbnb_num_cols:
    airbnb[num_col] = (airbnb[num_col] - airbnb[num_col].mean()) / airbnb[num_
col].std()
```

In [36]:
```
# 50.05% negative labels, 49.95% positive labels
airbnb['is_expensive'].value_counts(normalize = True)
```

Out[36]:
```
False    0.500501
True     0.499499
Name: is_expensive, dtype: float64
```

In [37]:
```
# split data into input features and labels
airbnb_df_X = airbnb.drop(columns=['is_expensive'])
airbnb_df_y = airbnb['is_expensive']
```

In [38]:
```
# one hot encoding for the categorical columns
airbnb_X_cat_col = ['neighbourhood_group', 'room_type']
airbnb_X = pd.get_dummies(columns=airbnb_X_cat_col, data=airbnb_df_X)

airbnb_y = airbnb_df_y.replace({True: 1, False: 0})
```

Clean Olympic Dataset

The dataset is from https://www.kaggle.com/heesoo37/120-years-of-olympic-history-athletes-and-results?select=athlete_events.csv (https://www.kaggle.com/heesoo37/120-years-of-olympic-history-athletes-and-results?select=athlete_events.csv), and includes historic data of all participants from the Olympic Games, from Athens 1896 to Rio 2016. There are 271116 unique values/observations. We want to see whether or not we can predict a gold medalist from all the medalists just by using participants' features. We use the 'Medal' column as our target variable for our dataset.

A more detailed description of 6 features are shown below:

- Sex: M or F, str
- Age: age, int
- Height: in centimeters, int
- Weight: in kilgrams, int
- NOC: National Olympic Committee 3-letter code, str
- Sport: sport, str

A detailed description of the target variable is shown below:

- Medal: Gold, Silver, Bronze, or NA, str

For this dataset, we want use these 6 input features and apply machine learning algorithms to predict whether an Olympian is a gold medalist or not.

In [39]:
```
# load the Olympic dataset
olympic = pd.read_csv('data/athlete_events.csv')
olympic.head()
```

Out[39]:

	ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	
0	1	A Dijiang	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Ba
1	2	A Lamusi	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	
2	3	Gunnar Nielsen Aaby	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Ant
3	4	Edgar Lindenau Aabye	M	34.0	NaN	NaN	Denmark/Sweden	DEN	1900 Summer	1900	Summer	
4	5	Christine Jacoba Aaftink	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	

In [40]:
```
# We only care about the medalists out of all Olympians, so we want to drop th
e NaN values in 'Medal'
olympic[olympic['Medal'].isnull()].shape
```

Out[40]: (231333, 15)

In [41]:
```
# drop null medal values from dataset
olympic = olympic[olympic['Medal'].notnull()]
olympic.shape
```

Out[41]: (39783, 15)

In [42]:
```
# These are the features we will be looking at for our classifier
olympic = olympic[['Sex', 'Age','Height','Weight','NOC','Sport','Medal']]
```

In [43]:
```
# This removes the NaN values from the numerical columns
olympic = olympic[olympic['Height'].notna()]
olympic = olympic[olympic['Weight'].notna()]
olympic = olympic[olympic['Age'].notna()]
olympic.head()
```

Out[43]:

	Sex	Age	Height	Weight	NOC	Sport	Medal
40	M	28.0	184.0	85.0	FIN	Ice Hockey	Bronze
41	M	28.0	175.0	64.0	FIN	Gymnastics	Bronze
42	M	28.0	175.0	64.0	FIN	Gymnastics	Gold
44	M	28.0	175.0	64.0	FIN	Gymnastics	Gold
48	M	28.0	175.0	64.0	FIN	Gymnastics	Gold

In [44]:
```
# After that cleaning, we are left with 30181 participants to work with
olympic.shape
```

Out[44]: (30181, 7)

In [45]:
```
# We want to make a column to see if the participant is a Gold Medalist
olympic['Gold'] = olympic['Medal']=='Gold'
```

In [46]:
```
# We no longer need the Medal column
olympic = olympic.drop(columns=['Medal'])
```

In [47]:
```
# Scale numerical attributes to 0 mean 1 std
olympic_num_cols = ['Age', 'Height', 'Weight']
for num_col in olympic_num_cols:
    olympic[num_col] = (olympic[num_col] -
                        olympic[num_col].mean()) / olympic[num_col].std()
```

In [48]:
```
# 66.31% negative labels, 33.68% positive labels
olympic['Gold'].value_counts(normalize = True)
```

Out[48]:
```
False    0.663132
True     0.336868
Name: Gold, dtype: float64
```

In [49]:
```
# split the olympic dataset into input features and labels
olympic_df_X = olympic.drop(columns=['Gold'])
olympic_df_y = olympic['Gold']
```

In [50]:
```
# one hot encode all the categorical columns
olympic_X_cat_col = ['Sex','NOC','Sport']
olympic_X = pd.get_dummies(columns=olympic_X_cat_col, data=olympic_df_X)
olympic_y = olympic_df_y.replace({True: 1, False: 0})
```

Perform Trials

In [51]:
```
# Parameters for models
tree_params = [
    {
        'max_depth': [2,3,4,5,7,10,13,15,18,None],
        'min_samples_split':[2,3,5,7,10,15,20],
        'min_samples_leaf':[2,3,5,7,10,15,20]
    }
]

log_reg_params = [
    {
        'solver': ['lbfgs'],
        'max_iter': [5000],
        'penalty': ['l2'],
        'C': 10**np.arange(-4, 5, 1, dtype='float32')
    },
    {
        'solver': ['saga'],
        'max_iter': [5000],
        'penalty': ['l1', 'l2'],
        'C': 10**np.arange(-4, 5, 1, dtype='float32')
    },
    {
        'solver': ['saga', 'lbfgs'],
        'max_iter': [5000],
        'penalty': ['none']
    }
]

perceptron_params = [
    {
        'penalty': ['l1', 'l2', 'elasticnet', 'none'],
        'alpha': [0.00001, 0.0001, 0.001, 0.01, 0.1]
    }
]

svc_params = [
    {
        'kernel': ['linear'],
        'C': 10 **np.array(np.arange(-3, 2, 2), dtype='float32')
    },
    {
        'kernel': ['poly'],
        'degree': [2, 3],
        'C': 10 **np.array(np.arange(-3, 2, 2), dtype='float32'),
    },
    {
        'kernel': ['rbf'],
        'C': 10 **np.array(np.arange(-3, 2, 2), dtype='float32'),
        'gamma': [0.001,0.01,0.1,1,2]
    }
]

knn_params = [
    {
        'n_neighbors': np.arange(1, 106, 4),
```

```
        'metric': ["euclidean", "manhattan", "minkowski"]
    }
]

forest_params = [
    {
        'n_estimators': [1024],
        'min_samples_split': [1, 2, 4, 6, 8, 12, 16, 20]
    }
]

# models that do not include SVM classifier
models_without_svm = {
    'DT': (DecisionTreeClassifier(), tree_params),
    'LOGREG': (LogisticRegression(), log_reg_params),
    'PERC': (Perceptron(), perceptron_params),
    'KNN': (KNeighborsClassifier(), knn_params),
    'RF': (RandomForestClassifier(), forest_params)
}

# SVM model
models_only_svm = {
    'SVM': (SVC(), svc_params)
}
```

In [52]:
```
models = np.sort(list(models_without_svm.keys()) + list(models_only_svm.keys
())).tolist()

scoring = {
        'ACC': make_scorer(accuracy_score),
        'PREC': make_scorer(precision_score),
        'REC': make_scorer(recall_score),
        'SPEC': make_scorer(recall_score, pos_label=0),
        'F1': make_scorer(f1_score),
        'ROC': make_scorer(roc_auc_score),
    }

results_columns = ['DATASET', 'MODEL', 'TRIAL'] + [
    "TRAIN_" + x for x in list(scoring.keys())] + ["TEST_" + x for x in list(s
coring.keys())]
```

In [53]:
```
# perform 7 trials using each of 6 algorithms on one dataset
def perform_trials(dataset_name, models, data_X, data_y):

    num_trials = 7

    data_results = pd.DataFrame(columns=results_columns)

    for model_name in models.keys():
        model = models[model_name][0]
        model_params_grid = models[model_name][1]
        model_results = pd.DataFrame(columns=results_columns)

        # perform 7 trials using each model on the dataset
        for trial_count in range(num_trials):
            # pick 5000 samples with replacement to be in the training set
            X_train, X_test, y_train, y_test = train_test_split(data_X, data_y
,
                                                                train_size=500
0,
                                                                random_state=t
rial_count)

            # grid search with 5 k-folds
            search = GridSearchCV(model, model_params_grid, cv=5, verbose=3,
                                  n_jobs=-1, refit=False, scoring=scoring)

            # fit grid search model with training set
            search.fit(X_train, y_train)

            # store 7 metrics calculated in one trial
            model_result = {
                'DATASET': dataset_name,
                'MODEL': model_name,
                'TRIAL': trial_count + 1
            }

            for score_name in scoring.keys():
                # find the best parameters that make model achieves best score
of the metric
                best_params = search.cv_results_['params'][np.argmin(
                    search.cv_results_['rank_test_' + score_name])]
                # use best parameters to create the optimal model for the metr
ic
                best_model = clone(model).set_params(**best_params)
                # train the optimal model
                best_model.fit(X_train, y_train)

                # compute metrics
                train_score = scoring[score_name](best_model, X_train, y_train
)
                test_score = scoring[score_name](best_model, X_test, y_test)

                # append scores
                model_result['TRAIN_' + score_name] = train_score
                model_result['TEST_' + score_name] = test_score
```

```
            # append scores of one trial to the model_results dataframe
            model_results = model_results.append(model_result, ignore_index=Tr
ue)

        # append model_results to data_results
        data_results = data_results.append(model_results, ignore_index=True)

        # store scores averaged over 7 trials
        avg_result = {
            'DATASET': dataset_name,
            'MODEL': model_name,
            'TRIAL': 'avg',

            'TRAIN_ACC': model_results.train_accuracy.mean(),
            'TRAIN_PREC': model_results.train_precision.mean(),
            'TRAIN_REC': model_results.train_recall.mean(),
            'TRAIN_SPEC': model_results.train_specificity.mean(),
            'TRAIN_F1': model_results.train_f1.mean(),
            'TRAIN_ROC': model_results.train_roc_auc.mean(),

            'TEST_ACC': model_results.test_accuracy.mean(),
            'TEST_PREC': model_results.test_precision.mean(),
            'TEST_REC': model_results.test_recall.mean(),
            'TEST_SPEC': model_results.test_specificity.mean(),
            'TEST_F1': model_results.test_f1.mean(),
            'TEST_ROC': model_results.test_roc_auc.mean()
        }

        # append avg_result to the data_results dataframe
        data_results = data_results.append(avg_result, ignore_index=True)

    return data_results
```

Run algorithms on Datasets

Results of Chess Dataset

In [ ]:
```
chess_results_no_svm = perform_trials('chess', models_without_svm, chess_X, ch
ess_y)
chess_results_no_svm.to_csv('results/chess_no_svm.csv', index = False)
chess_results_no_svm
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    0.2s
[Parallel(n_jobs=-1)]: Done 208 tasks      | elapsed:    4.1s
[Parallel(n_jobs=-1)]: Done 528 tasks      | elapsed:   13.0s
[Parallel(n_jobs=-1)]: Done 976 tasks      | elapsed:   22.3s
[Parallel(n_jobs=-1)]: Done 1552 tasks      | elapsed:   46.1s
[Parallel(n_jobs=-1)]: Done 2256 tasks      | elapsed:  1.2min
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.2min finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    0.3s
[Parallel(n_jobs=-1)]: Done 208 tasks      | elapsed:    3.3s
[Parallel(n_jobs=-1)]: Done 528 tasks      | elapsed:    8.6s
[Parallel(n_jobs=-1)]: Done 976 tasks      | elapsed:   17.4s
[Parallel(n_jobs=-1)]: Done 1552 tasks      | elapsed:   32.1s
[Parallel(n_jobs=-1)]: Done 2256 tasks      | elapsed:   54.1s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.0min finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    0.4s
[Parallel(n_jobs=-1)]: Done 208 tasks      | elapsed:    3.7s
[Parallel(n_jobs=-1)]: Done 528 tasks      | elapsed:    9.0s
[Parallel(n_jobs=-1)]: Done 976 tasks      | elapsed:   17.8s
[Parallel(n_jobs=-1)]: Done 1552 tasks      | elapsed:   33.3s
[Parallel(n_jobs=-1)]: Done 2256 tasks      | elapsed:   54.9s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.0min finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    0.5s
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed:    2.5s
[Parallel(n_jobs=-1)]: Done 272 tasks      | elapsed:    5.4s
[Parallel(n_jobs=-1)]: Done 496 tasks      | elapsed:    9.7s
[Parallel(n_jobs=-1)]: Done 784 tasks      | elapsed:   17.6s
[Parallel(n_jobs=-1)]: Done 1136 tasks      | elapsed:   27.9s
[Parallel(n_jobs=-1)]: Done 1552 tasks      | elapsed:   41.2s
[Parallel(n_jobs=-1)]: Done 2032 tasks      | elapsed:   55.6s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.1min finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    0.3s
[Parallel(n_jobs=-1)]: Done 208 tasks      | elapsed:    4.2s
[Parallel(n_jobs=-1)]: Done 528 tasks      | elapsed:   11.1s
[Parallel(n_jobs=-1)]: Done 976 tasks      | elapsed:   20.6s
[Parallel(n_jobs=-1)]: Done 1552 tasks      | elapsed:   37.9s
[Parallel(n_jobs=-1)]: Done 2256 tasks      | elapsed:  1.0min
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.2min finished

Fitting 5 folds for each of 29 candidates, totalling 145 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    1.9s
```

In [ ]:
```
chess_results_svm = perform_trials('chess', models_only_svm, chess_X, chess_y)
chess_results_svm.to_csv('results/chess_svm.csv', index = False)
chess_results_svm
```

In [78]:
```
chess_results_no_svm = pd.read_csv('results/chess_no_svm.csv')
chess_results_svm = pd.read_csv('results/chess_svm.csv')
```

In [79]:
```
chess_results = chess_results_no_svm.append(chess_results_svm, ignore_index=True)
chess_results.to_csv('results/chess.csv', index = False)
```

Results of Mushrooms Dataset

In [59]:
```
shrooms_results_no_svm = perform_trials('shrooms', models_without_svm, shrooms_X, shrooms_y)
shrooms_results_no_svm.to_csv('results/shrooms_no_svm.csv', index = False)
shrooms_results_no_svm
```

```
Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    0.1s
[Parallel(n_jobs=-1)]: Done 368 tasks      | elapsed:    2.2s
[Parallel(n_jobs=-1)]: Done 1008 tasks      | elapsed:    6.4s
[Parallel(n_jobs=-1)]: Done 1904 tasks      | elapsed:   12.7s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:   16.6s finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:    0.1s
[Parallel(n_jobs=-1)]: Done 368 tasks      | elapsed:    2.3s
[Parallel(n_jobs=-1)]: Done 1008 tasks      | elapsed:    6.7s
[Parallel(n_jobs=-1)]: Done 1904 tasks      | elapsed:   12.9s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:   16.5s finished
```

Out[59]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_f1
0	shrooms	tree	1	1.0000	1.0	1.000000	1.0	1.000000
1	shrooms	tree	2	0.9994	1.0	0.998750	1.0	0.999375
2	shrooms	tree	avg	0.9997	1.0	0.999375	1.0	0.999687

In [ ]:
```
shrooms_results_svm = perform_trials('shrooms', models_only_svm, shrooms_X, shrooms_y)
shrooms_results_svm.to_csv('results/shrooms_svm.csv', index = False)
shrooms_results_svm
```

In [81]:
```
shrooms_results_no_svm = pd.read_csv('results/shrooms_no_svm.csv')
shrooms_results_svm = pd.read_csv('results/shrooms_svm.csv')
```

In [82]:
```
shrooms_results = shrooms_results_no_svm.append(shrooms_results_svm, ignore_in
dex=True)
shrooms_results.to_csv('results/shrooms.csv', index = False)
```

Results of Cardio Dataset

In [51]:
```
# running algorithms except SVM on cardio dataset
cardio_results_no_svm = perform_trials('cardio', models_without_svm, cardio_X,
cardio_y)
cardio_results_no_svm
```

```
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
```

Out[51]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_
0	cardio	tree	1	0.727200	0.769610	0.657407	0.814388	0.7054
1	cardio	tree	2	0.730800	0.825455	0.771586	0.884584	0.7361
2	cardio	tree	3	0.727600	0.773550	0.721763	0.817405	0.7389
3	cardio	tree	4	0.737200	0.787893	0.746233	0.823245	0.7244
4	cardio	tree	5	0.735000	0.797519	0.734182	0.807801	0.7436
5	cardio	tree	6	0.734800	0.747525	0.732354	0.799922	0.7202
6	cardio	tree	7	0.741400	0.784314	0.714229	0.819456	0.7480
7	cardio	tree	avg	0.733429	0.783695	0.725393	0.823829	0.7309
8	cardio	log_reg	1	0.729200	0.760739	0.000000	1.000000	0.7088
9	cardio	log_reg	2	0.652000	0.661729	1.000000	0.658646	0.6882
10	cardio	log_reg	3	0.728000	0.763263	1.000000	0.784059	0.7158
11	cardio	log_reg	4	0.652800	0.668092	1.000000	0.686844	0.6690
12	cardio	log_reg	5	0.727200	0.751857	1.000000	1.000000	0.7163
13	cardio	log_reg	6	0.721600	0.749176	0.649531	0.000000	0.6958
14	cardio	log_reg	7	0.658200	0.674133	0.640288	1.000000	0.6423
15	cardio	log_reg	avg	0.695571	0.718427	0.755688	0.732793	0.6909
16	cardio	perceptron	1	0.659600	0.813711	0.937198	0.652623	0.6798
17	cardio	perceptron	2	0.538600	0.807792	1.000000	0.970228	0.6808
18	cardio	perceptron	3	0.716800	0.785208	0.000000	0.783652	0.0000
19	cardio	perceptron	4	0.533800	0.814642	0.676447	0.424536	0.6892
20	cardio	perceptron	5	0.502400	0.842105	0.000000	0.966224	0.6398
21	cardio	perceptron	6	0.709800	0.747525	0.977152	0.799922	0.6754
22	cardio	perceptron	7	0.570600	0.768566	0.730216	0.946357	0.3010
23	cardio	perceptron	avg	0.604514	0.797079	0.617288	0.791935	0.5237
24	cardio	knn	1	0.688400	0.708935	0.665862	0.748808	0.6685
25	cardio	knn	2	0.676200	0.696905	0.645212	0.708401	0.6700
26	cardio	knn	3	0.690600	0.698636	0.679260	0.730378	0.6805
27	cardio	knn	4	0.672400	0.702938	0.683584	0.738095	0.6890
28	cardio	knn	5	0.683600	0.704795	0.668524	0.738641	0.6690
29	cardio	knn	6	0.668200	0.687437	0.658507	0.755983	0.6314
30	cardio	knn	7	0.679600	0.700187	0.639888	0.743395	0.6624
31	cardio	knn	avg	0.679857	0.699976	0.662977	0.737672	0.6673
32	cardio	forest	1	0.813800	0.822480	0.816425	0.829889	0.8259
33	cardio	forest	2	0.834000	0.847025	0.848901	0.834013	0.8570
34	cardio	forest	3	0.820000	0.845248	0.963400	0.850752	0.8167

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_
35	cardio	forest	4	0.844000	0.862226	0.992466	0.865214	0.8437
36	cardio	forest	5	0.812600	0.819579	0.823717	0.821070	0.8111
37	cardio	forest	6	0.827400	0.844618	0.773154	0.859553	0.8174
38	cardio	forest	7	0.823200	0.837083	0.921263	0.842274	0.8189
39	cardio	forest	avg	0.825000	0.839751	0.877047	0.843252	0.8273

In [52]:
```
# running SVM algorithm on cardio dataset, generally take longer time to run than other algorithms combined
cardio_results_svm = perform_trials('cardio', models_only_svm, cardio_X, cardio_y)
cardio_results_svm
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
```

Out[52]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_f1
0	cardio	svm	1	0.737200	0.805755	0.814412	1.000000	0.719471
1	cardio	svm	2	0.733200	0.748567	1.000000	0.760604	0.732665
2	cardio	svm	3	0.737000	0.785714	1.000000	0.824319	0.726668
3	cardio	svm	4	0.735000	0.777726	1.000000	0.802260	0.741597
4	cardio	svm	5	0.734400	0.770219	1.000000	0.805790	0.725506
5	cardio	svm	6	0.728400	0.776371	0.821297	1.000000	0.704654
6	cardio	svm	7	0.738600	0.903614	1.000000	0.996797	0.724668
7	cardio	svm	avg	0.734829	0.795424	0.947959	0.884253	0.725033

In [53]:
```
# combine results of svm and non-svm algorithms and save as a csv file
cardio_final_results = cardio_results_no_svm.append(cardio_results_svm, ignore_index=True)
cardio_final_results.to_csv('results/cardio_results.csv', index = False)
```

Results of Australian Rain Dataset

In [55]:
```
# running algorithms except SVM on Australian rain dataset
aus_results_no_svm = perform_trials('aus', models_without_svm, aus_X, aus_y)
aus_results_no_svm
```

```
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 29 candidates, totalling 145 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Fitting 5 folds for each of 8 candidates, totalling 40 fits
```

Out[55]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_
0	aus	tree	1	0.849200	0.800000	0.628410	0.982220	0.5778
1	aus	tree	2	0.855000	0.777567	0.853636	0.970000	0.5987
2	aus	tree	3	0.842800	0.784884	0.853160	0.981142	0.5593
3	aus	tree	4	0.853800	0.823529	0.729849	0.984500	0.6233
4	aus	tree	5	0.862200	0.838608	0.867384	0.986869	0.6998
5	aus	tree	6	0.856800	0.772182	0.716822	0.975827	0.6564
6	aus	tree	7	0.838800	0.775701	0.702997	0.969223	0.6967
7	aus	tree	avg	0.851229	0.796067	0.764608	0.978540	0.6303
8	aus	log_reg	1	0.853600	0.788591	0.493885	1.000000	0.5922
9	aus	log_reg	2	0.854600	0.852665	0.532727	1.000000	0.6142
10	aus	log_reg	3	0.851800	0.826568	0.504647	1.000000	0.5944
11	aus	log_reg	4	0.851200	0.824607	0.534101	1.000000	0.6184
12	aus	log_reg	5	0.852400	0.808333	0.527778	1.000000	0.6154
13	aus	log_reg	6	0.853600	0.783607	0.512150	1.000000	0.6065
14	aus	log_reg	7	0.847800	0.798635	0.498638	1.000000	0.5906
15	aus	log_reg	avg	0.852143	0.811858	0.514846	1.000000	0.6045
16	aus	perceptron	1	0.813600	0.576074	0.000000	1.000000	0.5155
17	aus	perceptron	2	0.825200	0.704710	1.000000	0.958205	0.4177
18	aus	perceptron	3	0.784800	0.000000	0.000000	1.000000	0.0000
19	aus	perceptron	4	0.774200	0.000000	0.000000	1.000000	0.0000
20	aus	perceptron	5	0.223200	0.000000	0.748208	0.000000	0.5529
21	aus	perceptron	6	0.786000	0.214000	0.885047	1.000000	0.3525
22	aus	perceptron	7	0.220200	0.220200	0.000000	0.000000	0.5070
23	aus	perceptron	avg	0.632457	0.244998	0.376179	0.708315	0.3351
24	aus	knn	1	0.848400	0.862745	1.000000	0.998222	1.0000
25	aus	knn	2	0.834600	0.938776	1.000000	0.999231	1.0000
26	aus	knn	3	0.846200	0.869565	1.000000	0.998981	1.0000
27	aus	knn	4	0.847600	0.872549	1.000000	0.996642	1.0000
28	aus	knn	5	0.834600	0.910828	1.000000	0.997683	1.0000
29	aus	knn	6	0.844000	0.901961	1.000000	0.998728	1.0000
30	aus	knn	7	0.840600	0.842105	1.000000	0.997692	1.0000
31	aus	knn	avg	0.842286	0.885504	1.000000	0.998168	1.0000
32	aus	forest	1	1.000000	0.964286	0.985889	0.992634	1.0000
33	aus	forest	2	0.997400	1.000000	1.000000	1.000000	0.9940
34	aus	forest	3	0.997800	0.970506	1.000000	0.994139	1.0000

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_
35	aus	forest	4	0.966400	0.974607	0.980514	0.994575	0.9924
36	aus	forest	5	0.997600	0.955696	0.991039	0.990474	0.9945
37	aus	forest	6	0.996400	0.997778	0.984112	0.993893	0.9919
38	aus	forest	7	1.000000	0.969613	0.980926	0.993845	0.9903
39	aus	forest	avg	0.993657	0.976069	0.988926	0.994223	0.9947

In [56]:
```
# running SVM algorithm on Australian rain dataset, generally take
# longer time to run than other algorithms combined
aus_results_svm = perform_trials('aus', models_only_svm, aus_X, aus_y)
aus_results_svm
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Fitting 5 folds for each of 24 candidates, totalling 120 fits
```

Out[56]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_f1
0	aus	svm	1	0.857600	0.808743	0.484478	1.0	0.588915
1	aus	svm	2	0.859000	0.942529	0.700000	1.0	0.619383
2	aus	svm	3	0.853400	0.861272	0.471190	1.0	0.572881
3	aus	svm	4	0.856200	0.895161	0.719221	1.0	0.811189
4	aus	svm	5	0.855200	0.886667	0.511649	1.0	0.602321
5	aus	svm	6	0.856000	0.946667	0.490654	1.0	0.590551
6	aus	svm	7	0.847200	0.854460	0.506812	1.0	0.588918
7	aus	svm	avg	0.854943	0.885071	0.554858	1.0	0.624880

In [57]:
```
# combine results of svm and non-svm algorithms and save as a csv file
aus_final_results = aus_results_no_svm.append(aus_results_svm, ignore_index=True)
aus_final_results.to_csv('results/aus_results.csv', index = False)
```

Results of AirBnB Price Dataset

In [22]:
```
# running algorithms except SVM on airbnb dataset
airbnb_results_no_svm = perform_trials('airbnb', models_without_svm, airbnb_X,
airbnb_y)
airbnb_results_no_svm
```

```
Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:    3.6s
[Parallel(n_jobs=-1)]: Done 656 tasks      | elapsed:    6.5s
[Parallel(n_jobs=-1)]: Done 1936 tasks      | elapsed:   12.9s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:   15.1s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.4s
[Parallel(n_jobs=-1)]: Done 1528 tasks      | elapsed:    6.1s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:    9.8s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.4s
[Parallel(n_jobs=-1)]: Done 1528 tasks      | elapsed:    6.9s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:   10.5s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.4s
[Parallel(n_jobs=-1)]: Done 1528 tasks      | elapsed:    6.5s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:   10.1s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.5s
[Parallel(n_jobs=-1)]: Done 824 tasks      | elapsed:    4.4s
[Parallel(n_jobs=-1)]: Done 2104 tasks      | elapsed:   12.4s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:   13.9s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 52 candidates, totalling 260 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    1.3s
[Parallel(n_jobs=-1)]: Done 253 out of 260 | elapsed:    9.0s remaining:
0.2s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:    9.1s finished
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/_sag.py:330: ConvergenceWarning: The max_iter was reached which means th
e coef_ did not converge
  "the coef_ did not converge", ConvergenceWarning)
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 52 candidates, totalling 260 fits
```

```
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    1.1s
[Parallel(n_jobs=-1)]: Done 253 out of 260 | elapsed:    8.8s remaining:
0.2s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:    8.9s finished
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/_logistic.py:1505: UserWarning: Setting penalty='none' will ignore the C
and l1_ratio parameters
  "Setting penalty='none' will ignore the C and l1_ratio "
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/_sag.py:330: ConvergenceWarning: The max_iter was reached which means th
e coef_ did not converge
  "the coef_ did not converge", ConvergenceWarning)
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 52 candidates, totalling 260 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    1.1s
[Parallel(n_jobs=-1)]: Done 253 out of 260 | elapsed:    8.7s remaining:
0.2s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:    9.1s finished
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/_logistic.py:1505: UserWarning: Setting penalty='none' will ignore the C
and l1_ratio parameters
  "Setting penalty='none' will ignore the C and l1_ratio "
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/_sag.py:330: ConvergenceWarning: The max_iter was reached which means th
e coef_ did not converge
  "the coef_ did not converge", ConvergenceWarning)

Fitting 5 folds for each of 52 candidates, totalling 260 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.2s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    8.7s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:    9.8s finished
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/_logistic.py:1505: UserWarning: Setting penalty='none' will ignore the C
and l1_ratio parameters
  "Setting penalty='none' will ignore the C and l1_ratio "
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/_sag.py:330: ConvergenceWarning: The max_iter was reached which means th
e coef_ did not converge
  "the coef_ did not converge", ConvergenceWarning)

Fitting 5 folds for each of 52 candidates, totalling 260 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    1.3s
[Parallel(n_jobs=-1)]: Done 253 out of 260 | elapsed:    9.2s remaining:
0.3s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:    9.4s finished
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/_sag.py:330: ConvergenceWarning: The max_iter was reached which means th
e coef_ did not converge
  "the coef_ did not converge", ConvergenceWarning)
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 20 candidates, totalling 100 fits
```

```
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.5s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.9s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.6s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.9s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.6s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.8s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.5s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.8s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    0.5s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.8s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.4s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    8.4s
[Parallel(n_jobs=-1)]: Done 398 out of 405 | elapsed:   14.7s remaining:
0.3s
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:   14.9s finished

Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.3s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    8.3s
[Parallel(n_jobs=-1)]: Done 398 out of 405 | elapsed:   14.5s remaining:
0.3s
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:   14.8s finished

Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.3s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    8.5s
[Parallel(n_jobs=-1)]: Done 398 out of 405 | elapsed:   14.7s remaining:
0.3s
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:   15.0s finished

Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.3s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    8.8s
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:   15.5s finished
```

```
Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.4s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    8.7s
[Parallel(n_jobs=-1)]: Done 398 out of 405 | elapsed:   15.0s remaining:
0.3s
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:   15.3s finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   32.5s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:   56.0s finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   31.6s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:   55.0s finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   32.4s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:   55.6s finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   32.5s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:   55.5s finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   32.3s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:   55.5s finished
```

Out[22]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_
0	airbnb	tree	1	1.00000	1.000000	1.000000	1.000000	1.0000
1	airbnb	tree	2	1.00000	1.000000	1.000000	1.000000	1.0000
2	airbnb	tree	3	1.00000	1.000000	1.000000	1.000000	1.0000
3	airbnb	tree	4	1.00000	1.000000	1.000000	1.000000	1.0000
4	airbnb	tree	5	1.00000	1.000000	1.000000	1.000000	1.0000
5	airbnb	tree	avg	1.00000	1.000000	1.000000	1.000000	1.0000
6	airbnb	log_reg	1	0.98780	1.000000	0.975373	1.000000	0.9875
7	airbnb	log_reg	2	0.99060	1.000000	0.981033	1.000000	0.9904
8	airbnb	log_reg	3	0.98760	1.000000	0.974684	1.000000	0.9871
9	airbnb	log_reg	4	0.98840	1.000000	0.976585	1.000000	0.9881
10	airbnb	log_reg	5	0.98560	0.991857	0.979100	0.992038	0.9854
11	airbnb	log_reg	avg	0.98800	0.998371	0.977355	0.998408	0.9877
12	airbnb	perceptron	1	1.00000	1.000000	1.000000	1.000000	1.0000
13	airbnb	perceptron	2	1.00000	1.000000	1.000000	1.000000	1.0000
14	airbnb	perceptron	3	1.00000	1.000000	1.000000	1.000000	1.0000
15	airbnb	perceptron	4	0.84940	0.766873	1.000000	0.701546	0.8680
16	airbnb	perceptron	5	1.00000	1.000000	1.000000	1.000000	1.0000
17	airbnb	perceptron	avg	0.96988	0.953375	1.000000	0.940309	0.9736
18	airbnb	knn	1	0.99460	0.998779	0.990311	0.998811	0.9945
19	airbnb	knn	2	0.99460	0.998373	0.990718	0.998414	0.9945
20	airbnb	knn	3	0.99480	0.997945	0.991425	0.998040	0.9946
21	airbnb	knn	4	0.99280	0.998774	0.986677	0.998811	0.9926
22	airbnb	knn	5	0.99320	0.997567	0.988746	0.997611	0.9931
23	airbnb	knn	avg	0.99400	0.998288	0.989576	0.998337	0.9939
24	airbnb	forest	1	1.00000	1.000000	1.000000	1.000000	1.0000
25	airbnb	forest	2	1.00000	1.000000	1.000000	1.000000	1.0000
26	airbnb	forest	3	1.00000	1.000000	1.000000	1.000000	1.0000
27	airbnb	forest	4	1.00000	1.000000	1.000000	1.000000	1.0000
28	airbnb	forest	5	1.00000	1.000000	1.000000	1.000000	1.0000
29	airbnb	forest	avg	1.00000	1.000000	1.000000	1.000000	1.0000

In [23]:
```
# running SVM algorithm on AirBnB dataset, generally take longer time to run t
han other algorithms combined
airbnb_results_svm = perform_trials('airbnb', models_only_svm, airbnb_X, airbn
b_y)
airbnb_results_svm
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  39 tasks      | elapsed:    4.5s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:   27.2s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    3.8s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:   25.6s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    4.1s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:   26.3s finished

Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    3.9s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:   25.7s finished

Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    4.4s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:   26.2s finished
```

Out[23]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_f1
0	airbnb	svm	1	1.00000	1.000000	1.0	1.000000	1.000000
1	airbnb	svm	2	1.00000	1.000000	1.0	1.000000	1.000000
2	airbnb	svm	3	0.99940	0.998777	1.0	0.998824	0.999388
3	airbnb	svm	4	1.00000	1.000000	1.0	1.000000	1.000000
4	airbnb	svm	5	1.00000	1.000000	1.0	1.000000	1.000000
5	airbnb	svm	avg	0.99988	0.999755	1.0	0.999765	0.999878

In [24]:
```
airbnb_final_results = airbnb_results_no_svm.append(airbnb_results_svm, ignore
_index=True)
airbnb_final_results.to_csv('results/airbnb_results.csv', index = False)
```

Results of Olympic Gold Medal Dataset

In [26]:
```
# running algorithms except SVM on Olympic dataset
olympic_results_no_svm = perform_trials('olympic', models_without_svm, olympic
_X, olympic_y)
olympic_results_no_svm
```

```
Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:    1.0s
[Parallel(n_jobs=-1)]: Done 440 tasks      | elapsed:    6.4s
[Parallel(n_jobs=-1)]: Done 1080 tasks      | elapsed:   18.5s
[Parallel(n_jobs=-1)]: Done 1892 tasks      | elapsed:   45.4s
[Parallel(n_jobs=-1)]: Done 2180 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.4min finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.1s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    4.3s
[Parallel(n_jobs=-1)]: Done 552 tasks      | elapsed:   11.4s
[Parallel(n_jobs=-1)]: Done 1000 tasks      | elapsed:   20.5s
[Parallel(n_jobs=-1)]: Done 1576 tasks      | elapsed:   36.2s
[Parallel(n_jobs=-1)]: Done 2280 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.2min finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    0.7s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    3.4s
[Parallel(n_jobs=-1)]: Done 552 tasks      | elapsed:    9.2s
[Parallel(n_jobs=-1)]: Done 1000 tasks      | elapsed:   17.9s
[Parallel(n_jobs=-1)]: Done 1576 tasks      | elapsed:   33.9s
[Parallel(n_jobs=-1)]: Done 2280 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.3min finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.0s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    3.8s
[Parallel(n_jobs=-1)]: Done 552 tasks      | elapsed:    9.0s
[Parallel(n_jobs=-1)]: Done 1000 tasks      | elapsed:   22.2s
[Parallel(n_jobs=-1)]: Done 1576 tasks      | elapsed:   40.2s
[Parallel(n_jobs=-1)]: Done 2280 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.3min finished

Fitting 5 folds for each of 490 candidates, totalling 2450 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    0.7s
[Parallel(n_jobs=-1)]: Done 232 tasks      | elapsed:    3.4s
[Parallel(n_jobs=-1)]: Done 552 tasks      | elapsed:    8.5s
[Parallel(n_jobs=-1)]: Done 1000 tasks      | elapsed:   16.8s
[Parallel(n_jobs=-1)]: Done 1576 tasks      | elapsed:   32.5s
[Parallel(n_jobs=-1)]: Done 2280 tasks      | elapsed:   58.8s
[Parallel(n_jobs=-1)]: Done 2450 out of 2450 | elapsed:  1.1min finished

Fitting 5 folds for each of 52 candidates, totalling 260 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  32 tasks      | elapsed:    7.8s
[Parallel(n_jobs=-1)]: Done 128 tasks      | elapsed:   41.7s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:  1.8min finished

Fitting 5 folds for each of 52 candidates, totalling 260 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  32 tasks      | elapsed:    7.9s
[Parallel(n_jobs=-1)]: Done 128 tasks      | elapsed:   39.8s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:  1.7min finished

Fitting 5 folds for each of 52 candidates, totalling 260 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  32 tasks      | elapsed:    7.7s
[Parallel(n_jobs=-1)]: Done 128 tasks      | elapsed:   39.9s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:  1.7min finished

Fitting 5 folds for each of 52 candidates, totalling 260 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  32 tasks      | elapsed:    7.7s
[Parallel(n_jobs=-1)]: Done 128 tasks      | elapsed:   40.1s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:  1.7min finished

Fitting 5 folds for each of 52 candidates, totalling 260 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  32 tasks      | elapsed:    7.6s
[Parallel(n_jobs=-1)]: Done 128 tasks      | elapsed:   40.1s
[Parallel(n_jobs=-1)]: Done 260 out of 260 | elapsed:  1.7min finished

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.3s
[Parallel(n_jobs=-1)]: Done  93 out of 100 | elapsed:    2.9s remaining:
0.2s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    3.0s finished
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/metrics/
_classification.py:1272: UndefinedMetricWarning: Precision is ill-defined and
being set to 0.0 due to no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/metrics/
_classification.py:1272: UndefinedMetricWarning: Precision is ill-defined and
being set to 0.0 due to no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.4s
[Parallel(n_jobs=-1)]: Done  93 out of 100 | elapsed:    2.9s remaining:
0.2s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    3.0s finished
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.4s
[Parallel(n_jobs=-1)]: Done  93 out of 100 | elapsed:    2.9s remaining:
0.2s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    3.0s finished
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/metrics/
_classification.py:1272: UndefinedMetricWarning: Precision is ill-defined and
being set to 0.0 due to no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/sklearn/metrics/
_classification.py:1272: UndefinedMetricWarning: Precision is ill-defined and
being set to 0.0 due to no predicted samples. Use `zero_division` parameter t
o control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.2s
[Parallel(n_jobs=-1)]: Done  93 out of 100 | elapsed:    2.7s remaining:
0.2s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    2.8s finished

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:    1.2s
[Parallel(n_jobs=-1)]: Done  93 out of 100 | elapsed:    2.9s remaining:
0.2s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    3.0s finished

Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:    3.5s
[Parallel(n_jobs=-1)]: Done 120 tasks      | elapsed:   23.9s
[Parallel(n_jobs=-1)]: Done 280 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:  1.5min finished

Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:    3.5s
[Parallel(n_jobs=-1)]: Done 120 tasks      | elapsed:   23.0s
[Parallel(n_jobs=-1)]: Done 280 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:  1.6min finished

Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:    3.5s
[Parallel(n_jobs=-1)]: Done 120 tasks      | elapsed:   22.9s
[Parallel(n_jobs=-1)]: Done 280 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:  1.6min finished

Fitting 5 folds for each of 81 candidates, totalling 405 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:    3.5s
[Parallel(n_jobs=-1)]: Done 120 tasks      | elapsed:   23.0s
[Parallel(n_jobs=-1)]: Done 280 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:  1.5min finished

Fitting 5 folds for each of 81 candidates, totalling 405 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:    3.6s
[Parallel(n_jobs=-1)]: Done 120 tasks      | elapsed:   23.3s
[Parallel(n_jobs=-1)]: Done 280 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:  1.5min finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:  2.6min finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:  2.6min finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:  2.7min finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:  1.5min
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:  2.6min finished

Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:  1.5min
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:  2.6min finished
```

Out[26]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_
0	olympic	tree	1	0.95220	0.954655	0.997053	0.096386	0.9753
1	olympic	tree	2	0.95680	0.958232	0.998108	0.144628	0.9777
2	olympic	tree	3	0.95400	0.954911	0.998737	0.104000	0.9763
3	olympic	tree	4	0.95000	0.952678	0.996839	0.074803	0.9742
4	olympic	tree	5	0.96040	0.961701	0.998326	0.140271	0.9796
5	olympic	tree	avg	0.95468	0.956436	0.997813	0.112018	0.9766
6	olympic	log_reg	1	0.95020	0.950200	1.000000	0.000000	0.9744
7	olympic	log_reg	2	0.95160	0.951600	1.000000	0.000000	0.9752
8	olympic	log_reg	3	0.95000	0.950000	1.000000	0.000000	0.9743
9	olympic	log_reg	4	0.94920	0.949200	1.000000	0.000000	0.9739
10	olympic	log_reg	5	0.95580	0.955800	1.000000	0.000000	0.9774
11	olympic	log_reg	avg	0.95136	0.951360	1.000000	0.000000	0.9750
12	olympic	perceptron	1	0.04980	0.000000	0.000000	1.000000	0.0000
13	olympic	perceptron	2	0.92880	0.955694	0.970156	0.115702	0.9628
14	olympic	perceptron	3	0.05000	0.000000	0.000000	1.000000	0.0000
15	olympic	perceptron	4	0.94920	0.949200	1.000000	0.000000	0.9739
16	olympic	perceptron	5	0.95580	0.955800	1.000000	0.000000	0.9774
17	olympic	perceptron	avg	0.58672	0.572139	0.594031	0.423140	0.5828
18	olympic	knn	1	0.95020	0.950200	1.000000	0.000000	0.9744
19	olympic	knn	2	0.95180	0.951790	1.000000	0.004132	0.9753
20	olympic	knn	3	0.95180	0.951894	0.999789	0.040000	0.9752
21	olympic	knn	4	0.94920	0.949200	1.000000	0.000000	0.9739
22	olympic	knn	5	0.95580	0.955800	1.000000	0.000000	0.9774
23	olympic	knn	avg	0.95176	0.951777	0.999958	0.008826	0.9752
24	olympic	forest	1	0.96000	0.959604	1.000000	0.196787	0.9793
25	olympic	forest	2	0.95940	0.959081	1.000000	0.161157	0.9791
26	olympic	forest	3	0.97060	0.969982	1.000000	0.412000	0.9847
27	olympic	forest	4	0.99940	0.999368	1.000000	0.988189	0.9996
28	olympic	forest	5	0.96180	0.961755	0.999791	0.140271	0.9804
29	olympic	forest	avg	0.97024	0.969958	0.999958	0.379681	0.9846

In [27]:
```
# running SVM algorithm on the Olympic dataset, generally take longer time to
 run than other algorithms combined
olympic_results_svm = perform_trials('olympic', models_only_svm, olympic_X, ol
ympic_y)
olympic_results_svm
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   17.4s
/Users/melchisedec/opt/anaconda3/lib/python3.7/site-packages/joblib/external
s/loky/process_executor.py:706: UserWarning: A worker stopped while some jobs
were given to the executor. This can be caused by a too short worker timeout
or by a memory leak.
  "timeout or by a memory leak.", UserWarning
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:  2.2min finished

Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   19.0s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:  2.1min finished

Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   18.0s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:  2.1min finished

Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   17.6s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:  2.1min finished

Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   15.5s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:  2.0min finished
```

Out[27]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_f1
0	olympic	svm	1	0.95020	0.95020	1.0	0.0	0.974464
1	olympic	svm	2	0.95160	0.95160	1.0	0.0	0.975200
2	olympic	svm	3	0.95000	0.95000	1.0	0.0	0.974359
3	olympic	svm	4	0.94920	0.94920	1.0	0.0	0.973938
4	olympic	svm	5	0.95580	0.95580	1.0	0.0	0.977401
5	olympic	svm	avg	0.95136	0.95136	1.0	0.0	0.975072

In [28]:
```
olympic_final_results = olympic_results_no_svm.append(olympic_results_svm, ign
ore_index=True)
olympic_final_results.to_csv('results/olympic_results.csv', index = False)
```

In [29]:
```
# display performance
pd.read_csv('results/olympic_results.csv')
```

Out[29]:

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_
0	olympic	tree	1	0.95220	0.954655	0.997053	0.096386	0.9753
1	olympic	tree	2	0.95680	0.958232	0.998108	0.144628	0.9777
2	olympic	tree	3	0.95400	0.954911	0.998737	0.104000	0.9763
3	olympic	tree	4	0.95000	0.952678	0.996839	0.074803	0.9742
4	olympic	tree	5	0.96040	0.961701	0.998326	0.140271	0.9796
5	olympic	tree	avg	0.95468	0.956436	0.997813	0.112018	0.9766
6	olympic	log_reg	1	0.95020	0.950200	1.000000	0.000000	0.9744
7	olympic	log_reg	2	0.95160	0.951600	1.000000	0.000000	0.9752
8	olympic	log_reg	3	0.95000	0.950000	1.000000	0.000000	0.9743
9	olympic	log_reg	4	0.94920	0.949200	1.000000	0.000000	0.9739
10	olympic	log_reg	5	0.95580	0.955800	1.000000	0.000000	0.9774
11	olympic	log_reg	avg	0.95136	0.951360	1.000000	0.000000	0.9750
12	olympic	perceptron	1	0.04980	0.000000	0.000000	1.000000	0.0000
13	olympic	perceptron	2	0.92880	0.955694	0.970156	0.115702	0.9628
14	olympic	perceptron	3	0.05000	0.000000	0.000000	1.000000	0.0000
15	olympic	perceptron	4	0.94920	0.949200	1.000000	0.000000	0.9739
16	olympic	perceptron	5	0.95580	0.955800	1.000000	0.000000	0.9774
17	olympic	perceptron	avg	0.58672	0.572139	0.594031	0.423140	0.5828
18	olympic	knn	1	0.95020	0.950200	1.000000	0.000000	0.9744
19	olympic	knn	2	0.95180	0.951790	1.000000	0.004132	0.9753
20	olympic	knn	3	0.95180	0.951894	0.999789	0.040000	0.9752
21	olympic	knn	4	0.94920	0.949200	1.000000	0.000000	0.9739
22	olympic	knn	5	0.95580	0.955800	1.000000	0.000000	0.9774
23	olympic	knn	avg	0.95176	0.951777	0.999958	0.008826	0.9752
24	olympic	forest	1	0.96000	0.959604	1.000000	0.196787	0.9793
25	olympic	forest	2	0.95940	0.959081	1.000000	0.161157	0.9791
26	olympic	forest	3	0.97060	0.969982	1.000000	0.412000	0.9847
27	olympic	forest	4	0.99940	0.999368	1.000000	0.988189	0.9996
28	olympic	forest	5	0.96180	0.961755	0.999791	0.140271	0.9804
29	olympic	forest	avg	0.97024	0.969958	0.999958	0.379681	0.9846
30	olympic	svm	1	0.95020	0.950200	1.000000	0.000000	0.9744
31	olympic	svm	2	0.95160	0.951600	1.000000	0.000000	0.9752
32	olympic	svm	3	0.95000	0.950000	1.000000	0.000000	0.9743
33	olympic	svm	4	0.94920	0.949200	1.000000	0.000000	0.9739
34	olympic	svm	5	0.95580	0.955800	1.000000	0.000000	0.9774

	dataset	model	trial	train_accuracy	train_precision	train_recall	train_specificity	train_
35	olympic	svm	avg	0.95136	0.951360	1.000000	0.000000	0.975(

Create Main Tables and Appendix

In [54]:
```
datasets = ['CARDIO', 'AUS', 'CHESS', 'SHROOMS', 'AIRBNB', 'OLYMPIC']
```

In [55]:
```
# load datasets
cardio_results = pd.read_csv('results/cardio_results.csv')

aus_results = pd.read_csv('results/aus_results.csv')

olympic_results = pd.read_csv('results/olympic_results.csv')

airbnb_results = pd.read_csv('results/airbnb_results.csv')

shrooms_results = pd.read_csv('results/shrooms_results.csv')

chess_results = pd.read_csv('results/chess_results.csv')

# combine datasets
results = cardio_results.append([aus_results, airbnb_results, shrooms_results,
                                 chess_results, olympic_results], ignore_index
= True)
```

In [56]:
```
# generate standard column names
train_metrics = ["TRAIN_" + x for x in list(scoring.keys())]
test_metrics = ["TEST_" + x for x in list(scoring.keys())]
```

In [57]:
```
test_avg_results = results.loc[results['TRIAL'] == 'avg',
                               ['DATASET', 'MODEL', 'TRIAL'] + test_metrics]
test_avg_results = test_avg_results.reset_index(drop = True)
```

In [58]:
```
test_results = results.loc[results['TRIAL'] != 'avg',
                           ['DATASET', 'MODEL', 'TRIAL'] + test_metrics]
test_results = test_results.reset_index(drop = True)
```

In [59]:
```
train_avg_results = results.loc[results['TRIAL'] == 'avg',
                                ['DATASET', 'MODEL', 'TRIAL'] + train_metrics]
train_avg_results = train_avg_results.reset_index(drop = True)
```

In [60]:
```
train_results = results.loc[results['TRIAL'] != 'avg',
                            ['DATASET', 'MODEL', 'TRIAL'] + train_metrics]
train_results = train_results.reset_index(drop = True)
```

Table 1 - Dataset Description

In [61]:
```
# Create main table 1
table1 = pd.DataFrame(columns = ['Name', '#ATTR', 'TRAIN SIZE', 'TEST SIZE',
'%POS'])
table1['Name'] = np.array(datasets)
table1['#ATTR'] = np.array(['11', '22', '9/15', '22', '7/15', '6/14'])
table1['TRAIN SIZE'] = np.array([5000] * 6)
table1['TEST SIZE'] = ['65000', '140460', '15058', '3124', '43895', '25181']
table1['%POS'] = ['49.97%', '21.9146%', '49.8604%', '48.2%', '49.95%', '33.6
9%']
```

In [62]:
```
table1
```

Out[62]:

	Name	#ATTR	TRAIN SIZE	TEST SIZE	%POS
0	CARDIO	11	5000	65000	49.97%
1	AUS	22	5000	140460	21.9146%
2	CHESS	9/15	5000	15058	49.8604%
3	SHROOMS	22	5000	3124	48.2%
4	AIRBNB	7/15	5000	43895	49.95%
5	OLYMPIC	6/14	5000	25181	33.69%

In [63]:
```
# save it to a csv file
table1.to_csv('results/table1.csv', index = False)
pd.read_csv('results/table1.csv')
```

Out[63]:

	Name	#ATTR	TRAIN SIZE	TEST SIZE	%POS
0	CARDIO	11	5000	65000	49.97%
1	AUS	22	5000	140460	21.9146%
2	CHESS	9/15	5000	15058	49.8604%
3	SHROOMS	22	5000	3124	48.2%
4	AIRBNB	7/15	5000	43895	49.95%
5	OLYMPIC	6/14	5000	25181	33.69%

Table 2 - Model & Metrics (test avg trials & datasets)

In [64]:
```
# group mean scores across 7 trials of all metrics on testing set by models
# average them across all dataset
table2 = test_avg_results.groupby(by='MODEL').mean()
# create MEAN column by averaging scores of all metrics
table2['MEAN'] = table2.mean(axis = 1)
table2 = table2.reset_index()
table2
```

Out[64]:

	MODEL	TEST_ACC	TEST_PREC	TEST_REC	TEST_SPEC	TEST_F1	TEST_ROC	MEAN
0	DT	0.783277	0.800428	0.698014	0.883093	0.690094	0.742250	0.766192
1	KNN	0.768266	0.759028	0.678289	0.850243	0.673057	0.726480	0.742560
2	LOGREG	0.782066	0.752993	0.632982	0.886029	0.657502	0.736934	0.741418
3	PERC	0.701554	0.613989	0.615241	0.761425	0.579136	0.678610	0.658326
4	RF	0.793308	0.764779	0.669086	0.850390	0.699296	0.752849	0.754951
5	SVM	0.790895	0.806914	0.748617	0.963586	0.706468	0.754641	0.795187

In [65]:
```
# save it to a csv file
table2.to_csv('results/table2.csv', index = False)
pd.read_csv('results/table2.csv')
```

Out[65]:

	MODEL	TEST_ACC	TEST_PREC	TEST_REC	TEST_SPEC	TEST_F1	TEST_ROC	MEAN
0	DT	0.783277	0.800428	0.698014	0.883093	0.690094	0.742250	0.766192
1	KNN	0.768266	0.759028	0.678289	0.850243	0.673057	0.726480	0.742560
2	LOGREG	0.782066	0.752993	0.632982	0.886029	0.657502	0.736934	0.741418
3	PERC	0.701554	0.613989	0.615241	0.761425	0.579136	0.678610	0.658326
4	RF	0.793308	0.764779	0.669086	0.850390	0.699296	0.752849	0.754951
5	SVM	0.790895	0.806914	0.748617	0.963586	0.706468	0.754641	0.795187

Table 3 - Model & Dataset (test avg trials & metrics)

In [66]:
```
# create empty dataframe, and specify column names
table3 = pd.DataFrame(columns = ['MODEL'] + datasets + ['MEAN'])
table3['MODEL'] = models
```

In [67]:
```
for data in datasets:
    for model in models:
        # for each combinatin of algorithm and dataset
        # average across all metrics
        score = test_avg_results.loc[(test_avg_results['MODEL'] == model) &
                                     (test_avg_results['DATASET'] == data)][
            test_metrics].mean(axis = 1).tolist()[0]
        table3.loc[table3['MODEL'] == model, data] = score
    table3[data] = table3[data].astype('float64')
# create MEAN column by averaging scores across all datasets for each algorith
m
table3['MEAN'] = table3[datasets].mean(axis = 1)
table3
```

Out[67]:

	MODEL	CARDIO	AUS	CHESS	SHROOMS	AIRBNB	OLYMPIC	MEAN
0	DT	0.739188	0.711984	0.679259	0.999774	0.825712	0.641237	0.766192
1	KNN	0.666772	0.699582	0.634543	1.000000	0.829761	0.624705	0.742560
2	LOGREG	0.714271	0.739579	0.675994	1.000000	0.827292	0.491369	0.741418
3	PERC	0.655749	0.485895	0.582336	0.999883	0.684399	0.541694	0.658326
4	RF	0.728808	0.704868	0.663366	1.000000	0.832898	0.599768	0.754951
5	SVM	0.795096	0.742391	0.729506	1.000000	0.868146	0.635981	0.795187

In [68]:
```
# save it as a csv file
table3.to_csv('results/table3.csv', index = False)
pd.read_csv('results/table3.csv')
```

Out[68]:

	MODEL	CARDIO	AUS	CHESS	SHROOMS	AIRBNB	OLYMPIC	MEAN
0	DT	0.739188	0.711984	0.679259	0.999774	0.825712	0.641237	0.766192
1	KNN	0.666772	0.699582	0.634543	1.000000	0.829761	0.624705	0.742560
2	LOGREG	0.714271	0.739579	0.675994	1.000000	0.827292	0.491369	0.741418
3	PERC	0.655749	0.485895	0.582336	0.999883	0.684399	0.541694	0.658326
4	RF	0.728808	0.704868	0.663366	1.000000	0.832898	0.599768	0.754951
5	SVM	0.795096	0.742391	0.729506	1.000000	0.868146	0.635981	0.795187

Appendix 1 - Model & Metrics (train avg trials & datasets)

In [69]:
```
# group mean scores across 7 trials of all metrics on training set by models
# average them across all datasets
appendix1 = train_avg_results.groupby(by='MODEL').mean()
# create MEAN column by averaging scores of all metrics
appendix1['MEAN'] = appendix1.mean(axis = 1)
appendix1 = appendix1.reset_index()
appendix1
```

Out[69]:

	MODEL	TRAIN_ACC	TRAIN_PREC	TRAIN_REC	TRAIN_SPEC	TRAIN_F1	TRAIN_ROC	M
0	DT	0.801410	0.821938	0.816756	0.906471	0.773649	0.800073	0.82
1	KNN	0.786957	0.781165	0.872633	0.879235	0.861624	0.842512	0.83
2	LOGREG	0.786262	0.770186	0.639267	0.887305	0.667421	0.744131	0.74
3	PERC	0.705890	0.622589	0.619338	0.764205	0.583095	0.683130	0.66
4	RF	0.925290	0.934050	0.955250	0.955693	0.948712	0.949518	0.94
5	SVM	0.813976	0.845974	0.813235	0.980515	0.771450	0.799586	0.83

In [70]:
```
# save it as a csv file
appendix1.to_csv('results/appendix1.csv', index = False)
pd.read_csv('results/appendix1.csv')
```

Out[70]:

	MODEL	TRAIN_ACC	TRAIN_PREC	TRAIN_REC	TRAIN_SPEC	TRAIN_F1	TRAIN_ROC	M
0	DT	0.801410	0.821938	0.816756	0.906471	0.773649	0.800073	0.82
1	KNN	0.786957	0.781165	0.872633	0.879235	0.861624	0.842512	0.83
2	LOGREG	0.786262	0.770186	0.639267	0.887305	0.667421	0.744131	0.74
3	PERC	0.705890	0.622589	0.619338	0.764205	0.583095	0.683130	0.66
4	RF	0.925290	0.934050	0.955250	0.955693	0.948712	0.949518	0.94
5	SVM	0.813976	0.845974	0.813235	0.980515	0.771450	0.799586	0.83

Appendix 2 - Raw Test Scores

In [88]:
```
test_results.to_csv('results/appendix2.csv', index = False)
pd.read_csv('results/appendix2.csv').head()
```

Out[88]:

	DATASET	MODEL	TRIAL	TEST_ACC	TEST_PREC	TEST_REC	TEST_SPEC	TEST_F1	TEST
0	CARDIO	DT	1	0.718092	0.760048	0.651762	0.803292	0.698032	0.7
1	CARDIO	DT	2	0.716569	0.823521	0.757578	0.888329	0.717287	0.7
2	CARDIO	DT	3	0.723985	0.753861	0.691874	0.803083	0.711474	0.7
3	CARDIO	DT	4	0.724569	0.761494	0.689312	0.797068	0.713895	0.7
4	CARDIO	DT	5	0.726877	0.775533	0.706000	0.803805	0.711856	0.7

Appendix 3 - P values of Table 2 (Model & Metrics (test avg trials & datasets))

In [72]:
```
# get the model with the best score in each metric
best_models = table2.set_index('MODEL').idxmax()
```

In [73]:
```
# create empty dataframe, and specify column names
tl2_pvals = pd.DataFrame(columns = ['MODEL'] + test_metrics + ['MEAN'])
tl2_pvals['MODEL'] = table2['MODEL']
```

In [74]:
```
# Calculate p-values for six metrics on testing set
for metric in test_metrics:
    best_model = best_models[metric]
    # group individual scores by models
    grp = test_results.groupby(by='MODEL')
    for model in models:
        # individual scores of the best model
        dist1 = grp.get_group(best_model)[metric].tolist()
        # individual scores of other model
        dist2 = grp.get_group(model)[metric].tolist()
        # perform 2-sampled t-tests and obtain p-value
        pval = stats.ttest_ind(dist1, dist2).pvalue
        tl2_pvals.loc[tl2_pvals['MODEL'] == model, metric] = pval
    tl2_pvals[metric] = tl2_pvals[metric].astype('float64')
```

In [75]:
```
# Calculate p-values for mean on testing set
for model in models:
    # model with the best score in MEAN column of table 2
    best_model = best_models['MEAN']
    # scores on the row whose model is the best model in table 2
    dist1 = table2.loc[table2['MODEL'] == best_model, test_metrics].iloc[0].to
list()
    # scores on the other model's row in table 2
    dist2 = table2.loc[table2['MODEL'] == model, test_metrics].iloc[0].tolist
()
    # perform 2-sampled t-tests and obtain p-value
    pval = stats.ttest_ind(dist1, dist2).pvalue
    tl2_pvals.loc[tl2_pvals['MODEL'] == model, 'MEAN'] = pval
tl2_pvals['MEAN'] = tl2_pvals['MEAN'].astype('float64')
```

In [76]:
```
tl2_pvals
```

Out[76]:

	MODEL	TEST_ACC	TEST_PREC	TEST_REC	TEST_SPEC	TEST_F1	TEST_ROC	MEAN
0	DT	0.696592	0.800221	0.315647	0.002014	0.687457	0.674781	0.551291
1	KNN	0.354139	0.117555	0.153632	0.000137	0.416928	0.351340	0.273719
2	LOGREG	0.667217	0.089935	0.074958	0.051738	0.316641	0.565281	0.327170
3	PERC	0.007785	0.000115	0.050094	0.000078	0.017409	0.024818	0.013765
4	RF	1.000000	0.125360	0.123967	0.000015	0.858790	0.950381	0.394504
5	SVM	0.924287	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

In [77]:
```
# save it as a csv file
tl2_pvals.to_csv('results/tl2_pvals.csv', index = False)
pd.read_csv('results/tl2_pvals.csv')
```

Out[77]:

	MODEL	TEST_ACC	TEST_PREC	TEST_REC	TEST_SPEC	TEST_F1	TEST_ROC	MEAN
0	DT	0.696592	0.800221	0.315647	0.002014	0.687457	0.674781	0.551291
1	KNN	0.354139	0.117555	0.153632	0.000137	0.416928	0.351340	0.273719
2	LOGREG	0.667217	0.089935	0.074958	0.051738	0.316641	0.565281	0.327170
3	PERC	0.007785	0.000115	0.050094	0.000078	0.017409	0.024818	0.013765
4	RF	1.000000	0.125360	0.123967	0.000015	0.858790	0.950381	0.394504
5	SVM	0.924287	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Appendix 3 - P values of Table 3 (Model & Dataset (test avg trials & metrics))

In [78]:
```
# get the model with the best score in each dataset
best_models = table3.set_index('MODEL').idxmax()
```

In [79]:
```
# create empty dataframe, and specify column names
tl3_pvals = pd.DataFrame(columns = ['MODEL'] + datasets + ['MEAN'])
tl3_pvals['MODEL'] = models
```

In [80]:
```
# Calculate p-values for six datasets on testing set

# group individual scores by models and datasets
grp = test_results.groupby(by=['MODEL', 'DATASET'])
for data in datasets:
    # model with best score in each dataset
    best_model = best_models[data]
    for model in models:
        # individual scores of the best model
        dist1 = grp.get_group((best_model, data))[test_metrics].values.flatten
()
        # individual scores of the other model on that dataset
        dist2 = grp.get_group((model, data))[test_metrics].values.flatten()
        # perform 2-sampled t-tests and obtain p-value
        pval = stats.ttest_ind(dist1, dist2).pvalue
        tl3_pvals.loc[tl3_pvals['MODEL'] == model, data] = pval
    tl3_pvals[data] = tl3_pvals[data].astype('float64')
```

In [81]:
```
# Calculate p-values for mean on testing set
for model in models:
    # model with the best score in MEAN column of table 3
    best_model = best_models['MEAN']
    # scores on the row whose model is the best model in table 3
    dist1 = table3.loc[table3['MODEL'] == best_model, datasets].iloc[0].tolist
()
    # scores on the other model's row in table3
    dist2 = table3.loc[table3['MODEL'] == model, datasets].iloc[0].tolist()
    # perform 2-sampled t-tests and obtain p-value
    pval = stats.ttest_ind(dist1, dist2).pvalue
    tl3_pvals.loc[tl3_pvals['MODEL'] == model, 'MEAN'] = pval
tl3_pvals['MEAN'] = tl3_pvals['MEAN'].astype('float64')
```

In [82]:
```
tl3_pvals
```

Out[82]:

	MODEL	CARDIO	AUS	CHESS	SHROOMS	AIRBNB	OLYMPIC	MEAN
0	DT	2.417368e-03	0.439951	0.040054	0.028161	1.426333e-04	1.000000	0.703738
1	KNN	7.965780e-11	0.331824	0.000012	NaN	4.928899e-04	0.699824	0.519854
2	LOGREG	2.331690e-02	0.942313	0.053629	NaN	9.782886e-02	0.011458	0.545491
3	PERC	9.214726e-04	0.000102	0.000030	0.007619	2.208079e-09	0.046378	0.161766
4	RF	1.191210e-04	0.346393	0.001136	NaN	1.167387e-03	0.354726	0.616209
5	SVM	1.000000e+00	1.000000	1.000000	NaN	1.000000e+00	0.907375	1.000000

In [83]:
```
tl3_pvals.to_csv('results/tl3_pvals.csv', index = False)
pd.read_csv('results/tl3_pvals.csv')
```

Out[83]:

	MODEL	CARDIO	AUS	CHESS	SHROOMS	AIRBNB	OLYMPIC	MEAN
0	DT	2.417368e-03	0.439951	0.040054	0.028161	1.426333e-04	1.000000	0.703738
1	KNN	7.965780e-11	0.331824	0.000012	NaN	4.928899e-04	0.699824	0.519854
2	LOGREG	2.331690e-02	0.942313	0.053629	NaN	9.782886e-02	0.011458	0.545491
3	PERC	9.214726e-04	0.000102	0.000030	0.007619	2.208079e-09	0.046378	0.161766
4	RF	1.191210e-04	0.346393	0.001136	NaN	1.167387e-03	0.354726	0.616209
5	SVM	1.000000e+00	1.000000	1.000000	NaN	1.000000e+00	0.907375	1.000000

Appendix 3 - P value of appendix 1 (Model & Metrics (train avg trials & datasets))

In [84]:
```
# get the model with the best score in each metric
best_models = appendix1.set_index('MODEL').idxmax()

# create empty dataframe, and specify column names
appendix1_pvals = pd.DataFrame(columns = ['MODEL'] + train_metrics + ['MEAN'])
appendix1_pvals['MODEL'] = appendix1['MODEL']

# Calculate p-values for six metrics on training set
for metric in train_metrics:
    best_model = best_models[metric]
    # group individual scores by models
    grp = train_results.groupby(by='MODEL')
    for model in models:
        # individual scores of the best model
        dist1 = grp.get_group(best_model)[metric].tolist()
        # individual scores of other model
        dist2 = grp.get_group(model)[metric].tolist()
        # perform 2-sampled t-tests and obtain p-value
        pval = stats.ttest_ind(dist1, dist2).pvalue
        appendix1_pvals.loc[appendix1_pvals['MODEL'] == model, metric] = pval
    appendix1_pvals[metric] = appendix1_pvals[metric].astype('float64')
```

In [85]:
```
# Calculate p-values for mean on training set
for model in models:
    # model with the best score in MEAN column of appendix 1
    best_model = best_models['MEAN']
    # scores on the row whose model is the best model in appendix 1
    dist1 = appendix1.loc[appendix1['MODEL'] == best_model,
                          train_metrics].iloc[0].tolist()
    # scores on the other model's row in appendix 1
    dist2 = appendix1.loc[appendix1['MODEL'] == model, train_metrics].iloc[0].
tolist()
     # perform 2-sampled t-tests and obtain p-value
    pval = stats.ttest_ind(dist1, dist2).pvalue
    appendix1_pvals.loc[appendix1_pvals['MODEL'] == model, 'MEAN'] = pval
appendix1_pvals['MEAN'] = appendix1_pvals['MEAN'].astype('float64')
appendix1_pvals
```

Out[85]:

	MODEL	TRAIN_ACC	TRAIN_PREC	TRAIN_REC	TRAIN_SPEC	TRAIN_F1	TRAIN_RC
0	DT	8.287021e-08	1.701569e-08	1.853618e-10	0.000322	5.985106e-12	1.361701e-
1	KNN	2.040673e-08	6.789643e-09	5.382399e-04	0.000018	6.427847e-04	4.332876e-(
2	LOGREG	1.854892e-08	3.371046e-10	3.695047e-08	0.015201	3.854539e-10	1.250001e-
3	PERC	3.257160e-10	4.448309e-10	6.357903e-08	0.000015	6.634743e-12	6.718829e-
4	RF	1.000000e+00	1.000000e+00	1.000000e+00	0.054564	1.000000e+00	1.000000e+(
5	SVM	2.842700e-07	1.015314e-05	4.506495e-06	1.000000	8.881075e-12	7.155815e-

In [86]:
```
# save it to a csv file
appendix1_pvals.to_csv('results/appendix1_pvals.csv', index = False)
pd.read_csv('results/appendix1_pvals.csv')
```

Out[86]:

	MODEL	TRAIN_ACC	TRAIN_PREC	TRAIN_REC	TRAIN_SPEC	TRAIN_F1	TRAIN_RC
0	DT	8.287021e-08	1.701569e-08	1.853618e-10	0.000322	5.985106e-12	1.361701e-
1	KNN	2.040673e-08	6.789643e-09	5.382399e-04	0.000018	6.427847e-04	4.332876e-(
2	LOGREG	1.854892e-08	3.371046e-10	3.695047e-08	0.015201	3.854539e-10	1.250001e-
3	PERC	3.257160e-10	4.448309e-10	6.357903e-08	0.000015	6.634743e-12	6.718829e-
4	RF	1.000000e+00	1.000000e+00	1.000000e+00	0.054564	1.000000e+00	1.000000e+(
5	SVM	2.842700e-07	1.015314e-05	4.506495e-06	1.000000	8.881075e-12	7.155815e-

In [ ]: