COGS 118A Final Project: Binary Classification Model Comparison

Isaac Douglas

Editor: Isaac Douglas

Abstract

At the time of publication, "An Empirical Comparison of Supervised Learning Algorithms" by Caruana et al. represented a comprehensive study of the most common supervised learning algorithms of the day examined on different problems and different performance criterias. This paper seeks to replicate some of their findings and corroborate the validity of their results by testing a subset of the algorithms and metrics used in the original paper.

1. Introduction

Performed in the mid 1990s, The Statlog Project, henceforth referred to as STATLOG, performed a thorough investigation into many of the popular supervised learning algorithms of the time on 12 contemporaneous data sets (King et al., 1995). STATLOG used a total of 17 algorithms that included symbolic learners (CART, C4.5, CN2, etc.), statistical learners (Naive Bayes, kNN, logistic regression, etc.), and neural networks (backpropagation and radial basis functions). This research provided a strong benchmark for the machine learning community as to the respective strengths and weaknesses of these algorithms for different problem sets. By 2006, however, many new algorithms and techniques had been invented that potentially rivaled those evaluated in STATLOG.

This lead to "An Empirical Comparison of Supervised Learning Algorithms", henceforth refereed to as CNMO6 (Caruana & Niculescu-Mizil, 2006):. CNM06 included these recently invented algorithms and techniques (bagging, boosting, SVMs, random forests, etc.) in its evaluations and used a different set of problems along with a more thorough evaluation procedure. Considerate that the most important algorithmic performance criteria is heavily domain-dependent (precision/recall is more informative for information retrieval tasks, ROC is more useful for medical learning problems, etc.), they evaluated the algorithms with an extensive array of performance criteria including F1-score, Lift, ROC AUC, and several others (8 metrics in total). CNM06 concluded that, with and without calibration, boosted trees have the best overall performance on all metrics with very good performances as well from bagged trees, random forests, and neural networks. They also found that, on average, memory-based learning algorithms (e.g. kNN), boosted stumps, single decision trees, logistic regression, and naive bayes performed better than the top algorithms on only a very small intersection of performance criteria and problems. Still, as CNM06 points out, for some applications, those problems and metrics are most relevant.

This paper seeks to replicate the procedure of CNM06 on a subset of performance metrics, algorithms, and problems from the original paper as well as two additional data sets not used in the original paper. This paper will examine the performance of SVMs, logistic regression, and random forests on data sets with different numbers of attributes, sizes, and binary class balances. The data sets used as binary classification problems for this study are SURGICAL and CHURN from Kaggle and two of the original data sets used in CNM06, ADULT and COV_TYPE from the UCI machine learning repository (Blake & Merz, 1998). The findings of this paper are useful to strengthen or make doubtful some of the algorithmic performance conclusions of CNM06.

2. Methods

2.1 Learning Algorithms

Three different learning algorithms are tested on all the binary classification problems: SVMs, logistic regression, and random forests. All hyperparameters not specified are kept at the default settings of Scikit-learn version 0.24.1 (Pedregosa et al., 2011). An exhaustive grid search with 5-fold cross-validation for each hyperparameter setting is performed to find the most optimal setting to maximize each metric. For SVMs and logistic regression, the maximum iterations of the algorithm is capped at 100,000.

Support Vector Machines (SVM) For Support Vector Machines, I test RBF and sigmoid kernels and vary the regularization parameter from 10^{-5} to 10^{4} by factors of 10 and set γ to either k^{-1} or $(k * var(training\ set))^{-1}$ (k = number of features in the data set; see table below for actual γ values used on each data set). As well, I test polynomial kernels with degree 2 and 3 and vary the regularization parameter from 10^{-5} to 10^{-4}, and set γ according to the table below. Lastly, I test with a linear kernel, varying the the regularization parameter from 10^{-5} to 10^{-4} by factors of 10.

SVM γ Hyperparameter Values Used By Data Set			
PROBLEM	#ATTR	$(k)^{-1}$	$(k * var(features))^{-1}$
SURGICAL	45	0.022	0.076
CHURN	13	0.077	0.155
ADULT	108	0.009	0.075
COV_TYPE	54	0.019	0.084

Logistic Regression (LOGREG) For logistic regression, both Ridge (L2) and LASSO (L1) regularization are used varying the regularization parameter from (10^{-5} to 10^{-4} by factors of 10). The unregularized model is also tested.

Random Forests (RF) All trees used 1024 individual estimators just like in CNM06. Both Gini and Entropy split quality criterion are tested and the max features considered while looking for the best split at each node is either $\sqrt{k}$ or $log_2(k)$ (k = the number of features in the current data set). See table below for the exact max features considered per data set.

Max Features Considered By Data Set			
PROBLEM	#ATTR	$\sqrt{k}$	$log_2(k)$
SURGICAL	45	7	5
CHURN	13	4	4
ADULT	108	10	7
COV_TYPE	54	7	6

2.2 Performance Metrics

For all training, testing, and 5-fold cross-validation, three metrics were used to evaluate the performance of the algorithms: accuracy, f1-score, and ROC AUC (receiver operator characteristic area under curve). All of these metrics are in the range [0, 1] with 0 as the worst score and 1 as the best score which makes them easy to compare, evaluate, and average.

Accuracy (ACC) What percentage of the dataset did the model correctly predict?

$$Accuracy = \frac{correctly\ predicted\ data\ points}{all\ data\ points\ predicted} \tag{1}$$

F1-Score (FSC) As the harmonic mean of precision ($\frac{true\ positives}{predicted\ positives}$) and recall ($\frac{true\ positives}{all\ positives}$), the F1-score functions as a composite metric for assessing both of these scores.

$$F_1 = 2\ \frac{precision\ *recall}{precision + recall} \tag{2}$$

ROC AUC (Receiver Operating Characteristic Area Under Curve) We use ROC AUC to gauge how well the algorithm is able to make strongly separating classification boundaries.

2.3 Data Sets

PROBLEM	#ATTR	TRAIN SIZE	TEST SIZE	%POZ
SURGICAL	24/45	5000	9635	25%
CHURN	10/13	5000	5000	20%
ADULT	14/108	5000	27561	24%
COV_TYPE	12/54	5000	576012	48%

Table 1: Description of problems

Four data sets (problems) are used for the training, testing, and evaluation of the algorithms. Two data sets were gathered from Kaggle: SURGICAL and CHURN. The other two data sets are gathered from the UCI Machine Learning Repository: ADULT and COV_TYPE. For both Kaggle datasets, the binary classification tasks were built in to the data set. The SURGICAL data set contains 14,635 data points and 24 attributes that were a combination of binary, categorical, and continuous features. The binary classification task is to predict whether or not a complication occurs during surgery based on information about the patient, the procedure, the hospital, and the static context. The CHURN data set contains 10,000 data points and 10 features that are binary, categorical, and continuous in nature. The binary classification task is to predict whether or not a bank customer will leave the bank and close their account or continue being a customer based on demographic and behavioral information about the customer. Both the Kaggle data sets are slightly unbalanced. SURGICAL has 25% of its data points belonging to the positive class and 75% belonging to the negative class while in CHURN 20% of the data points belong to the positive class and 80% belong to the negative class. In the results section we will see how this affects the performance of the algorithms.

The UCI Machine Learning repository data sets did not have implicit binary classification tasks like the Kaggle data sets but they could be transformed into ones. The ADULT data set has 32,561 data points and 14 attributes that are categorical and continuous in nature containing demographic and financial information about an individual with the target being that individual's income. ADULT can be transformed into a binary classification task by changing the target to a binary variable of whether or not the individual's income is greater than 50K or less than 50K. With 24% of data points belonging to the positive class and 76% belonging to the negative class, ADULT is similarly balanced to SURGICAL and CHURN. The COV_TYPE data set contains 581,012 data points and 12 attributes that are categorical and continuous in nature. As the target is the cover type of the tree (7 total cover types labeled 1-7), I follow the same procedure as Caruana to transform this multi-class problem into a binary classification problem by setting the most prevalent cover type (2) as the positive class and all the other classes as the negative class. COV_TYPE, unlike the the other data sets, represents a very balanced classification task with 48% of data points belonging to the positive class and 52% to the negative class. Including a balanced data set like COV_TYPE provided more complete information about performance of the algorithms on different problem types.

3. Experiments & Results

For each algorithm and data set combination, five trials were performed. In each trial the data was split according to Table 1 with 5000 data points randomly selected from the data set and remaining set aside as the test set. Critical to the significance tests performed later, Trial i for all algorithm-data set combinations used the same partitioning of training and test sets. Using the training set, an exhaustive grid search with 5-fold cross-validation of all hyperparameter settings was performed to find the hyperparameter setting that maximized each of the three metrics based on its average score over all 5 cross-validation folds. For each metric, the model was re-trained with the best hyperparameter setting for that metric using the entire training set and evaluated on the test set.

The results of all these trials are displayed in Tables 2 and 3. A bolded score indicates the model of that row scored the highest for that column (i.e. the bolded score is the highest score in the column). An asterisk appended to a score indicates that while the model of that row performed worse during the experiments than the column's bolded score, the difference was not significant (see p-values section in Appendix for exact p-value comparison scores).[1]

Algorithm Test Set Performance by Metric				
MODEL	ACC	FSC	ROC	MEAN
SVM	0.828	0.630	0.853	0.770
LOGREG	0.803	0.549	0.828	0.727
RF	**0.844**	**0.669**	**0.889**	**0.801**

Table 2: Normalized scores for each learning algorithm by metric (average over four problems)

The average performance of all models for each of the three metrics over all problems and trials can be seen in Table 2 (see Raw Test Set Scores section in Appendix for raw data). the far right column lists the mean score for every model over all metrics. For all metrics, random forests outperformed both logistic regression and SVMs. For accuracy, the performance difference between random forests and the other two algorithms is significant with random forests outperforming SVMs by 1.60% and logistic regression by 4.10% percent. Again, for F-1 score, the outperformance of random forests compared to the other two algorithms is significant with a 3.90% advantage over SVMs and a 12.90% advantange over logistic regression. Random forests dominate the ROC AUC as well with a significant difference over both other algorithms. Using the ROC AUC metric, random forests outperform SVMs by 3.60% and logistic regression by 6.10%.

From the mean performance score of all three metrics on all four problems, we can see that random forests have a clear and significant advantage over SVMs and logistic regression. Although the absolute mean performance advantage of random forests is only 3.10% over SVMs, this advantage leaps up to 7.40% for logistic regression. Based on these results and the scores of each algorithm on the individual metrics, the data indicates random forest is the best algorithm across the three metrics followed relatively closely by SVMs. Logistic regression, however, is the clear worst algorithm (at least on these metrics for these problems) with the worst scores across all metrics.

Algorithm Test Set Performance by Problem					
MODEL	SURGICAL	CHURN	ADULT	COV_TYPE	MEAN
SVM	0.699	0.753	0.8045*	0.826	0.770
LOGREG	0.696	0.631	0.8033*	0.777	0.727
RF	**0.786**	**0.765**	**0.8053**	**0.847**	**0.801**

Table 3: Normalized scores for each learning algorithm by problem (average over three metrics)

1. Significance is determined by a two-tailed paired t-test with the threshold for significance set at p=0.05.

Table 3 contains the average performance of each algorithm on each problem (see Raw Test Set Scores section in Appendix for raw data). For all problems, random forests are more successful than the SVMs and logistic regression while SVMs are more successful than logistic regression on all problems.

With approximately a 1:3 positive to negative class ratio and 45 attributes (after encoding), SURGICAL represent problems with moderately imbalanced data and a moderate number of attributes. For all algorithms, this data set is either the worst or second to worst performance by mean metric score. SVMs performs worst on this data set wtih a mean metric score of 69.90% while the second to worst performance of logistic regression and random forests is on this data set. See CHURN for their worst performances.

CHURN has the least number of attributes after encoding (13) with the highest class imbalance of a 1:4 positive to negative class ratio. CHURN therefore represents problems with high moderate class imbalance and minimal features. We see the worst performance of logistic regression and random forests on this data set while SVMs close the gap with random forests to just over 1%. Still CHURN represents the second worst performance by SVMs.

ADULT has the highest number of attributes (108 post-encoding) with a class imbalance very similar to SURGICAL of approximately a 1:3 positive to negative class ratio. All three models achieve a mean metric score of around 80% on this problem such that the advantage of random forests over SVMs and logistic regression is no longer significant. The most different property of this data set compared to the other problems is its large number of attributes. Though other reasons might be responsible for the more level playing (like easier class separability), further research should more comprehensively examine how the number of attributes in the data set affects the performance of different algorithms.

COV_TYPE has a similar number of attributes as SURGICAL with 54 after encoding. COV_TYPE differs from all other data sets, however, in its near perfect 1:1 positive to negative class balance. I incorrectly hypothesized the algorithms would perform the worst on COV_TYPE. Because of the high number of samples in the data set, the training set of 5000 data points only represented 0.86% of the whole data set. Therefore, I suspected there would a high generalization error from the training to the test set since the classifiers would need to correctly classify the other 99.14% of the data. My hypothesis was thoroughly incorrect as all classifiers achieved their best mean metric score on COV_TYPE except for logistic regression which achieved its second best. Likely, this is due to the even class balance.

Since the MEAN column is the average performance of the classifier over all trials, data sets, and metrics, the values are the same as the MEAN column in Table 2 (see Table 2 MEAN column anaylsis).

4. Discussion

The results from this subset of the algorithms evaluated in CNM06 corroborate their findings. Across accuracy, F1-sore, and ROC AUC, random forests perform best overall followed by SVMs then logistic regression. This order is the same for all data sets and metrics and matches the order of these algorithms' performances in CNM06.

Looking at the individual data sets, CHURN has the lowest mean metric scores on average followed by SURGICAL, then ADULT, then COV_TYPE with the higest mean metric score. I believe COV_TYPE performed the best due to its high class balance and CHURN proved to be the hardest problem due to its large class imbalance. Future research should further examine the relationship between class imbalance and algorithmic performance and what strategies (downsampling, upsampling, stratification, etc.) are most effective at preventing the negative performance effect of class imbalances. This smaller-scale replication of CNM06 corroborates their findings and and strengthens the conclusions of Caruana et al.

References

[1] R. D. King, C. Feng & A. Sutherland (1995) STATLOG: Comparison of Classification Algorithms On Large Real-World Problems, Applied Artificial Intelligence, 9:3, 289-333, doi: 10.1080/0883

[2] R. Caruana and A. Niculescu-Mizil. "An Empirical Comparison of Supervised Learning Algorithms." *In Proceedings of the 23rd international conference on Machine learning*, 161-168. 2006.

[3] Blake, C., & Merz, C. (1998). UCI repository of machine learning databases.

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

[5] Mariappan, M. (2018, September). Dataset Surgical binary classification. Retrieved March 9, 2021 from https://www.kaggle.com/omnamahshivai/surgical-dataset-binary-classification.

[6] Iyyer, S. (2019, April). Churn Modeling. Retrieved March 9, 2021 from https://www.kaggle.com/shrutimechlearn/churn-modelling

5. Bonus

I took a digression off the totally beaten path and found two data sets that CNMO6 did not use. I selected SURGICAL and CHURN from Kaggle to add algorithms with different numbers of attributes, class balance, and data set size than the other two CNM06 data sets used in this report. Despite the extra cleaning required compared to only using Caruana data sets, selecting a wider variety of data set types furthered the goal of this paper to corroborate the findings of CNM06 in a different context and strengthen the conclusions this report reached.

Acknowledgements

Thank you to the COGS 118A Winter 2021 instructional staff for a great quarter! I learned a lot and had quite a bit of fun along the way. Thank you to my curious classmates for reminding me why I love learning and interacting with a community of fellow machine learning scholars. I had a lot of fun (and frustration) making this report and I'm proud of the result and I hope I get to work on similar issues in the future. I hope you enjoy the findings of my report.

Appendix

5.1 Training Set Performance

5.1.1 Table 2 Analog

Mean Training Set Performance by Metric				
MODEL	ACC	FSC	ROC	MEAN
SVM	0.862 (0.034)	0.771 (0.141)	0.887 (0.034)	0.840 (0.070)
LOGREG	0.807 (0.004)	0.557 (0.008)	0.834 (0.006)	0.733 (0.006)
RF	1.000 (0.156)	1.000 (0.331)	1.000 (0.111)	1.000 (0.199)

Table 4: Normalized scores for each learning algorithm by metric (average over four problems)

Table 4 records the mean training set performance of models across metrics. In order to more easily understand the generalizability of the algorithms, the generalization errors have been explicitly added in parentheses. For all model and metric pairs, the generalizability error is positive. We see results similar to the test set Table 2 results where random forests perform the best followed by

SVMs, then logistic regression. Interestingly, while the order of best to worst model by training set score is the same as the order by test set score for all metrics (1. random forests, 2. SVMs, 3. logistic regression), the ordering from best to worst model by generalization error is the opposite for all metrics (1. logistic regression, 2. SVMs, 3. random forests). This seems to imply that while logistic regression sacrificed in absolute lower scores, it made up with greater generalizability; in other words, logistic regression had the most bias while random forests had the most variance. For random forests, its perfect performance on the training set for all metrics indicates very obvious overfitting as its mean generalization error for all metrics is a 3217% increase of the generalization error of logistic regression and a 184% increase of the generalization error of SVM. Another perspective on this data is that SVMs and especially logistic regression are much less powerful compared to random forests and therefore have much less capability to overfit the data. If I were to further study this, I would try to change the maximum iterations of the SVMs and logistic regression models to 5,000,000 to see if that enabled them to increase their classification power and find a more optimal decision boundary.

5.1.2 Table 3 Analog

Mean Training Set Performance by Problem					
MODEL	SURGICAL	CHURN	ADULT	COV_TYPE	MEAN
SVM	0.830 (0.131)	0.789 (0.036)	0.844 (0.040)	0.897 (0.071)	0.840 (0.070)
LOGREG	0.705 (0.009)	0.627 (0.039)	0.813 (0.010)	0.786 (0.016)	0.733 (0.006)
RF	1.000 (0.214)	1.000 (0.235)	1.000 (0.195)	1.000 (0.153)	1.000 (0.199)

Table 5: Normalized scores for each learning algorithm by problem (average over three metrics)

Table 5 records the mean training set performance of models across problems. In order to more easily understand the generalizability of the algorithms, the generalization errors have been explicitly added in parentheses. In comparing the mean training set and test set performance across problems, we find similar results to comparing the training set and testing set performance by metric. Random forests achieve complete accuracy for all datasets, SVMs achieves a reasonable degree of accuracy, and logistic regression performs the worst on all data sets. Also similar to Table 4, for all data sets, the ordering of best to worst performing classifier (1. random forests, 2. SVMs, 3. logistic regression) is opposite that of the best to worst classifier by generalization error (1. logistic regression, 2. SVMs, 3. random forests) except on the CHURN data set where SVMs has slightly better generalization error compared to logistic regression. I also find differences in performance between the algorithms in examining the order of problems from least to most generalization error by algorithm:

- RF: COV_TYPE, ADULT, SURGICAL, CHURN
- LOGREG: SURGICAL, ADULT, COV_TYPE, CHURN.
- SVM: CHURN, ADULT, COV_TYPE, SURGICAL.

All problems except ADULT vary in the order they appear. This suggests random forests, logistic regression, and SVMs have different strengths and weaknesses when it comes to overfitting of binary classification problems. It also suggests a large number of attributes (as ADULT has post-encoding) may have an equalizing effect on generalization error across algorithms.

5.2 p-values

5.2.1 Table 2

Significance Test p-value for Metrics				
MODEL	ACC	FSC	ROC	MEAN
SVM	9.374×10^{-4}	2.210×10^{-3}	5.968×10^{-4}	3.175×10^{-5}
LOGREG	5.564×10^{-7}	3.209×10^{-5}	1.000×10^{-6}	2.839×10^{-7}
RF	**N/A**	**N/A**	**N/A**	**N/A**

Table 6: p-values for paired t-test score distributions over all trials and problems

5.2.2 Table 3

Significance Test p-value for Data Sets					
MODEL	SURGICAL	CHURN	ADULT	COV_TYPE	MEAN
SVM	1.675×10^{-7}	6.249×10^{-4}	0.624*	6.173×10^{-10}	3.175×10^{-5}
LOGREG	4.329×10^{-7}	1.026×10^{-4}	0.201*	2.696×10^{-16}	2.839×10^{-7}
RF	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**

Table 7: p-values for paired t-test score distributions over all trials and metrics

5.3 Raw Test Set Scores

	PROBLEM	MODEL	METRIC	TRIAL	SCORE
0	SURGICAL	SVM	ACC	1	0.800
1	SURGICAL	SVM	ACC	2	0.799
2	SURGICAL	SVM	ACC	3	0.797
3	SURGICAL	SVM	ACC	4	0.803
4	SURGICAL	SVM	ACC	5	0.805
5	SURGICAL	SVM	FSC	1	0.498
6	SURGICAL	SVM	FSC	2	0.494
7	SURGICAL	SVM	FSC	3	0.498
8	SURGICAL	SVM	FSC	4	0.496
9	SURGICAL	SVM	FSC	5	0.505
10	SURGICAL	SVM	ROC	1	0.801
11	SURGICAL	SVM	ROC	2	0.794
12	SURGICAL	SVM	ROC	3	0.792
13	SURGICAL	SVM	ROC	4	0.798
14	SURGICAL	SVM	ROC	5	0.800
15	SURGICAL	LOGREG	ACC	1	0.793
16	SURGICAL	LOGREG	ACC	2	0.795
17	SURGICAL	LOGREG	ACC	3	0.795
18	SURGICAL	LOGREG	ACC	4	0.797
19	SURGICAL	LOGREG	ACC	5	0.793
20	SURGICAL	LOGREG	FSC	1	0.480

Continued on next page

	PROBLEM	MODEL	METRIC	TRIAL	SCORE
21	SURGICAL	LOGREG	FSC	2	0.465
22	SURGICAL	LOGREG	FSC	3	0.476
23	SURGICAL	LOGREG	FSC	4	0.483
24	SURGICAL	LOGREG	FSC	5	0.489
25	SURGICAL	LOGREG	ROC	1	0.820
26	SURGICAL	LOGREG	ROC	2	0.813
27	SURGICAL	LOGREG	ROC	3	0.812
28	SURGICAL	LOGREG	ROC	4	0.814
29	SURGICAL	LOGREG	ROC	5	0.813
30	SURGICAL	RF	ACC	1	0.840
31	SURGICAL	RF	ACC	2	0.838
32	SURGICAL	RF	ACC	3	0.838
33	SURGICAL	RF	ACC	4	0.851
34	SURGICAL	RF	ACC	5	0.852
35	SURGICAL	RF	FSC	1	0.604
36	SURGICAL	RF	FSC	2	0.597
37	SURGICAL	RF	FSC	3	0.598
38	SURGICAL	RF	FSC	4	0.642
39	SURGICAL	RF	FSC	5	0.643
40	SURGICAL	RF	ROC	1	0.897
41	SURGICAL	RF	ROC	2	0.894
42	SURGICAL	RF	ROC	3	0.893
43	SURGICAL	RF	ROC	4	0.898
44	SURGICAL	RF	ROC	5	0.897
45	CHURN	SVM	ACC	1	0.860
46	CHURN	SVM	ACC	2	0.854
47	CHURN	SVM	ACC	3	0.861
48	CHURN	SVM	ACC	4	0.855
49	CHURN	SVM	ACC	5	0.865
50	CHURN	SVM	FSC	1	0.578
51	CHURN	SVM	FSC	2	0.561
52	CHURN	SVM	FSC	3	0.551
53	CHURN	SVM	FSC	4	0.561
54	CHURN	SVM	FSC	5	0.566
55	CHURN	SVM	ROC	1	0.846
56	CHURN	SVM	ROC	2	0.827
57	CHURN	SVM	ROC	3	0.835
58	CHURN	SVM	ROC	4	0.838
59	CHURN	SVM	ROC	5	0.836
60	CHURN	LOGREG	ACC	1	0.808
61	CHURN	LOGREG	ACC	2	0.815
62	CHURN	LOGREG	ACC	3	0.808
63	CHURN	LOGREG	ACC	4	0.809
64	CHURN	LOGREG	ACC	5	0.818
65	CHURN	LOGREG	FSC	1	0.312
66	CHURN	LOGREG	FSC	2	0.314
67	CHURN	LOGREG	FSC	3	0.278

Continued on next page

	PROBLEM	MODEL	METRIC	TRIAL	SCORE
68	CHURN	LOGREG	FSC	4	0.321
69	CHURN	LOGREG	FSC	5	0.326
70	CHURN	LOGREG	ROC	1	0.774
71	CHURN	LOGREG	ROC	2	0.772
72	CHURN	LOGREG	ROC	3	0.769
73	CHURN	LOGREG	ROC	4	0.769
74	CHURN	LOGREG	ROC	5	0.770
75	CHURN	RF	ACC	1	0.865
76	CHURN	RF	ACC	2	0.860
77	CHURN	RF	ACC	3	0.857
78	CHURN	RF	ACC	4	0.861
79	CHURN	RF	ACC	5	0.867
80	CHURN	RF	FSC	1	0.598
81	CHURN	RF	FSC	2	0.571
82	CHURN	RF	FSC	3	0.544
83	CHURN	RF	FSC	4	0.578
84	CHURN	RF	FSC	5	0.587
85	CHURN	RF	ROC	1	0.866
86	CHURN	RF	ROC	2	0.858
87	CHURN	RF	ROC	3	0.853
88	CHURN	RF	ROC	4	0.852
89	CHURN	RF	ROC	5	0.861
90	ADULT	SVM	ACC	1	0.853
91	ADULT	SVM	ACC	2	0.853
92	ADULT	SVM	ACC	3	0.851
93	ADULT	SVM	ACC	4	0.853
94	ADULT	SVM	ACC	5	0.853
95	ADULT	SVM	FSC	1	0.653
96	ADULT	SVM	FSC	2	0.663
97	ADULT	SVM	FSC	3	0.657
98	ADULT	SVM	FSC	4	0.660
99	ADULT	SVM	FSC	5	0.651
100	ADULT	SVM	ROC	1	0.905
101	ADULT	SVM	ROC	2	0.903
102	ADULT	SVM	ROC	3	0.904
103	ADULT	SVM	ROC	4	0.905
104	ADULT	SVM	ROC	5	0.905
105	ADULT	LOGREG	ACC	1	0.852
106	ADULT	LOGREG	ACC	2	0.849
107	ADULT	LOGREG	ACC	3	0.847
108	ADULT	LOGREG	ACC	4	0.852
109	ADULT	LOGREG	ACC	5	0.851
110	ADULT	LOGREG	FSC	1	0.654
111	ADULT	LOGREG	FSC	2	0.660
112	ADULT	LOGREG	FSC	3	0.646
113	ADULT	LOGREG	FSC	4	0.658
114	ADULT	LOGREG	FSC	5	0.659

Continued on next page

	PROBLEM	MODEL	METRIC	TRIAL	SCORE
115	ADULT	LOGREG	ROC	1	0.905
116	ADULT	LOGREG	ROC	2	0.905
117	ADULT	LOGREG	ROC	3	0.903
118	ADULT	LOGREG	ROC	4	0.906
119	ADULT	LOGREG	ROC	5	0.904
120	ADULT	RF	ACC	1	0.854
121	ADULT	RF	ACC	2	0.849
122	ADULT	RF	ACC	3	0.849
123	ADULT	RF	ACC	4	0.853
124	ADULT	RF	ACC	5	0.850
125	ADULT	RF	FSC	1	0.666
126	ADULT	RF	FSC	2	0.664
127	ADULT	RF	FSC	3	0.656
128	ADULT	RF	FSC	4	0.671
129	ADULT	RF	FSC	5	0.663
130	ADULT	RF	ROC	1	0.903
131	ADULT	RF	ROC	2	0.899
132	ADULT	RF	ROC	3	0.899
133	ADULT	RF	ROC	4	0.902
134	ADULT	RF	ROC	5	0.902
135	COV_TYPE	SVM	ACC	1	0.806
136	COV_TYPE	SVM	ACC	2	0.801
137	COV_TYPE	SVM	ACC	3	0.797
138	COV_TYPE	SVM	ACC	4	0.801
139	COV_TYPE	SVM	ACC	5	0.802
140	COV_TYPE	SVM	FSC	1	0.808
141	COV_TYPE	SVM	FSC	2	0.803
142	COV_TYPE	SVM	FSC	3	0.793
143	COV_TYPE	SVM	FSC	4	0.804
144	COV_TYPE	SVM	FSC	5	0.804
145	COV_TYPE	SVM	ROC	1	0.876
146	COV_TYPE	SVM	ROC	2	0.874
147	COV_TYPE	SVM	ROC	3	0.869
148	COV_TYPE	SVM	ROC	4	0.876
149	COV_TYPE	SVM	ROC	5	0.873
150	COV_TYPE	LOGREG	ACC	1	0.750
151	COV_TYPE	LOGREG	ACC	2	0.754
152	COV_TYPE	LOGREG	ACC	3	0.754
153	COV_TYPE	LOGREG	ACC	4	0.759
154	COV_TYPE	LOGREG	ACC	5	0.754
155	COV_TYPE	LOGREG	FSC	1	0.751
156	COV_TYPE	LOGREG	FSC	2	0.753
157	COV_TYPE	LOGREG	FSC	3	0.749
158	COV_TYPE	LOGREG	FSC	4	0.759
159	COV_TYPE	LOGREG	FSC	5	0.755
160	COV_TYPE	LOGREG	ROC	1	0.822
161	COV_TYPE	LOGREG	ROC	2	0.824

Continued on next page

	PROBLEM	MODEL	METRIC	TRIAL	SCORE
162	COV_TYPE	LOGREG	ROC	3	0.825
163	COV_TYPE	LOGREG	ROC	4	0.824
164	COV_TYPE	LOGREG	ROC	5	0.824
165	COV_TYPE	RF	ACC	1	0.821
166	COV_TYPE	RF	ACC	2	0.821
167	COV_TYPE	RF	ACC	3	0.816
168	COV_TYPE	RF	ACC	4	0.824
169	COV_TYPE	RF	ACC	5	0.816
170	COV_TYPE	RF	FSC	1	0.821
171	COV_TYPE	RF	FSC	2	0.823
172	COV_TYPE	RF	FSC	3	0.812
173	COV_TYPE	RF	FSC	4	0.824
174	COV_TYPE	RF	FSC	5	0.817
175	COV_TYPE	RF	ROC	1	0.901
176	COV_TYPE	RF	ROC	2	0.902
177	COV_TYPE	RF	ROC	3	0.899
178	COV_TYPE	RF	ROC	4	0.904
179	COV_TYPE	RF	ROC	5	0.896

CODE APPENDIX

In [ ]:
```
# utils
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from tqdm import tqdm
from collections import defaultdict
from tabulate import tabulate
from IPython.display import Latex, display
from scipy.stats import ttest_rel

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

# models
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

# metrics
from sklearn.metrics import accuracy_score, roc_auc_score, SCORERS
```

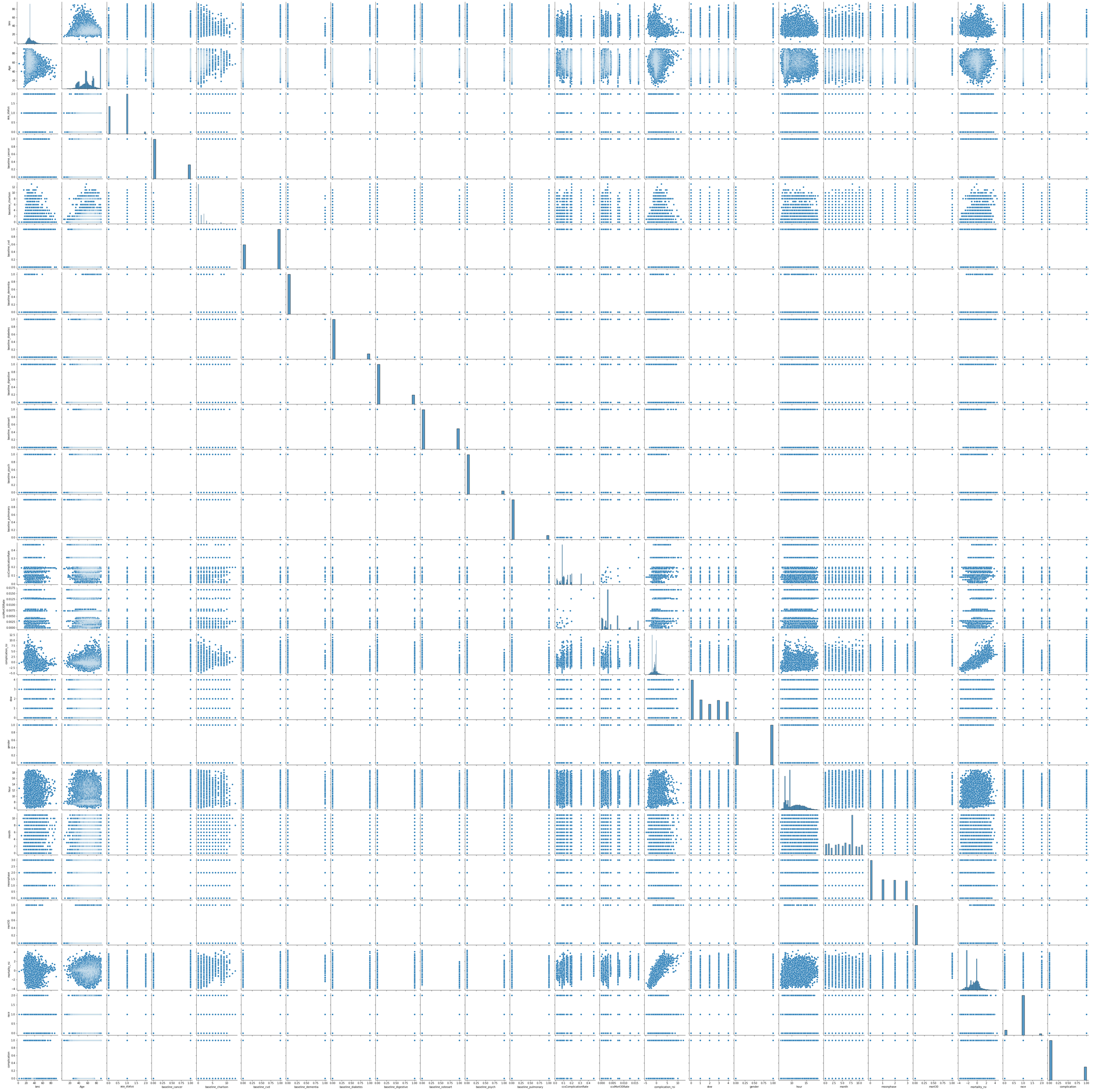
Cleaning up Surgical

In [ ]:
```
# Dataset 1: https://www.kaggle.com/omnamahshivai/surgical-dataset-binary-classification
# filename: Surgical-deepnet.csv
# Data dictionary: https://www.causeweb.org/tshs/datasets/Surgery%20Timing%20Data%20Dictionary.pdf
# target: complication
surgical = pd.read_csv('datasets/Surgical-deepnet.csv')
```

In [ ]:
```
surgical
```

In [211…
```
sns.pairplot(surgical)
```

Out[211…
```
<seaborn.axisgrid.PairGrid at 0x7ff2cc5d8670>
```



In [ ]:
```
#surgical.describe()
pd.set_option('display.max_columns', None)

print("all columns: ", list(surgical.columns))
for column in surgical.columns.values:
        if len(surgical[column].unique()) < 50:
            print(f"{column}: ", surgical[column].unique())
# drop ahrq_ccs because we can just ccsComplicationRate as a standin
surgical = surgical.drop(['ahrq_ccs'],axis=1)
# as we can see, surgical has no null values
print('ALL ROWS WITH MISSING VALUES:')
surgical[surgical.isnull().any(axis=1)]
```

In [ ]:
```
# already one-hot encoded: baseline_cancer, baseline_cvd, baseline_dementia, baseline_diabetes, baseline_digestive, baseline_osteoart
surgical_standardized_features = ["bmi", "Age", "ccsComplicationRate", "ccsMort30Rate", "complication_rsi", "hour", "mortality_rsi"]
surgical_categorical_features = ['asa_status', "dow", 'month', 'moonphase', 'race'] # to be one-hot encoded
surgical_minmax_features = ['baseline_charlson']

surgical_transformer = ColumnTransformer(
                transformers=[
                    ('std', StandardScaler(), surgical_standardized_features),
                    ('onehot', OneHotEncoder(), surgical_categorical_features),
                    ('minmax', MinMaxScaler(), surgical_minmax_features)
                ], remainder='passthrough')


surgical_trans = surgical_transformer.fit_transform(surgical)
print(f"surgical_trans.shape: {surgical_trans.shape}")
X_surgical = surgical_trans[:, :-1]
print(f"X_surgical.shape: {X_surgical.shape}")
y_surgical = surgical_trans[:, -1]
print(f"y_surgical.shape: {y_surgical.shape}")
print(f"all(y_surgical == surgical['complication']): {all(y_surgical == surgical['complication'])}")
```

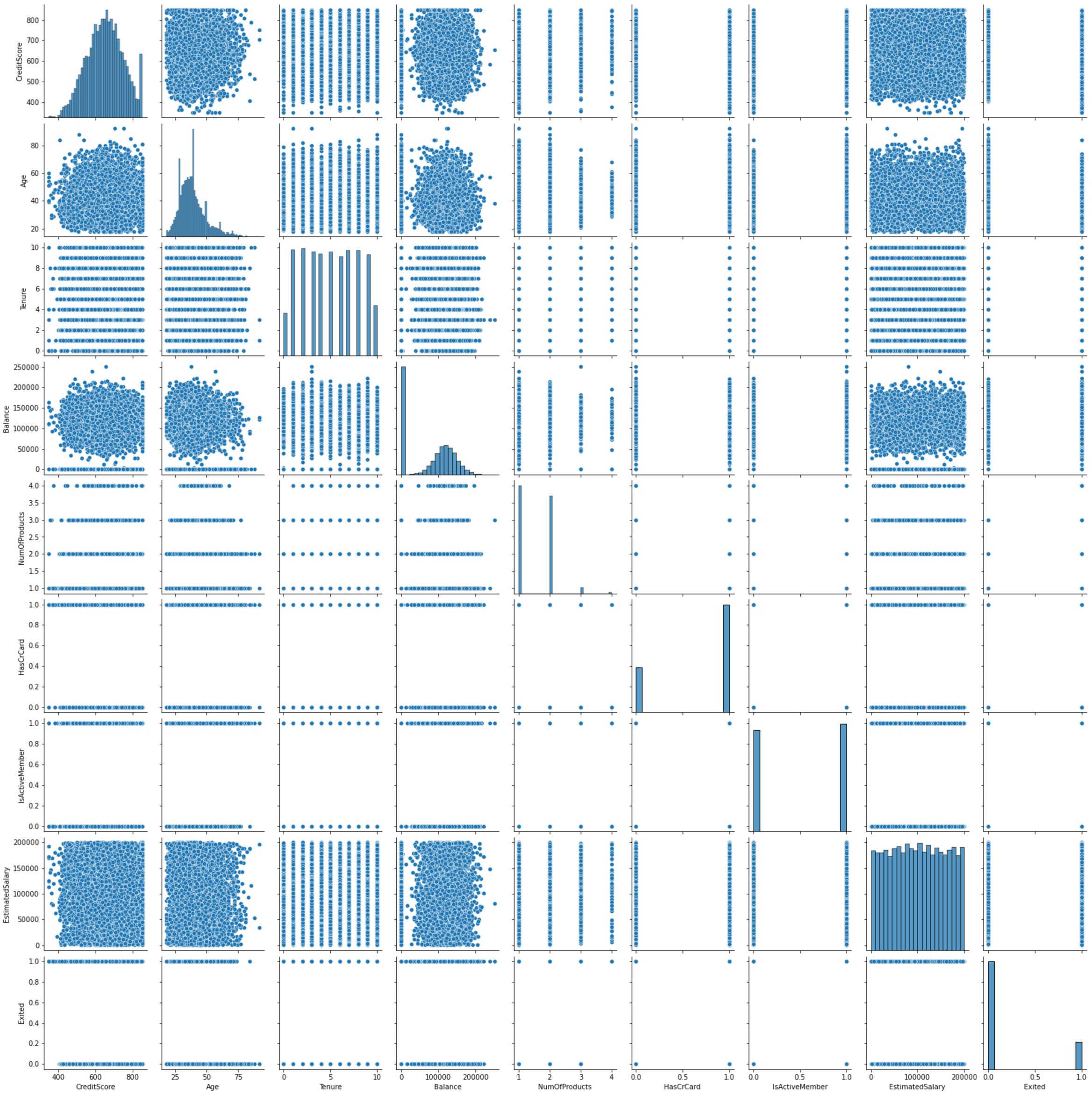
Cleaning up churn

In [ ]:
```
# Dataset 2: https://www.kaggle.com/shrutimechlearn/churn-modelling
# filename: Churn_Modeling.csv
# target: Exited
churn = pd.read_csv('datasets/Churn_Modeling.csv')
```

In [ ]:
```
churn
```

In [212…
```
sns.pairplot(churn)
```

Out[212…
```
<seaborn.axisgrid.PairGrid at 0x7ff2fbdc82e0>
```



In [ ]:
```
# as we can see, churn has no null values
print('# OF ROWS WITH MISSING VALUES:', len(churn[churn.isnull().any(axis=1)]))
```

In [9]:
```
print("all columns: ", list(churn.columns))

for column in churn.columns.values:
        if len(churn[column].unique()) < 50:
            print(f"{column}: ", churn[column].unique())

# drop id's and rownumber
churn = churn.drop(['CustomerId', 'Surname', 'RowNumber'], axis=1)
```

```
all columns:  ['RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProduct
s', 'HasCrCard', 'IsActiveMember', 'EstimatedSalary', 'Exited']
Geography:  ['France' 'Spain' 'Germany']
Gender:  ['Female' 'Male']
Tenure:  [ 2  1  8  7  4  6  3 10  5  9  0]
NumOfProducts:  [1 3 2 4]
HasCrCard:  [1 0]
IsActiveMember:  [1 0]
Exited:  [1 0]
```

In [10]:
```
# already one-hot encoded: HasCrCard, IsActiveMember, Exited
churn_standardized_features = ["CreditScore", "Age", "Balance", 'EstimatedSalary']
churn_categorical_features = ['Geography', "Gender"] # to be one-hot encoded
churn_minmax_features = ['Tenure', 'NumOfProducts']

churn_transformer = ColumnTransformer(
                transformers=[
                    ('std', StandardScaler(), churn_standardized_features),
                    ('onehot', OneHotEncoder(), churn_categorical_features),
                    ('minmax', MinMaxScaler(), churn_minmax_features)
                ], remainder='passthrough')

churn_trans = churn_transformer.fit_transform(churn)
print(f"churn_trans.shape: {churn_trans.shape}")
X_churn = churn_trans[:, :-1]
print(f"X_churn.shape: {X_churn.shape}")
y_churn = churn_trans[:, -1]
print(f"y_churn.shape: {y_churn.shape}")
print(f"all(y_churn == churn['Exited']): {all(y_churn == churn['Exited'])}")
```

```
churn_trans.shape: (10000, 14)
X_churn.shape: (10000, 13)
y_churn.shape: (10000,)
all(y_churn == churn['Exited']): True
```

Cleaning up ADULT

In [213…
```
# Dataset 7: https://archive.ics.uci.edu/ml/datasets/Adult
# filenames: adult.data + adult.test
adult = pd.read_csv('datasets/ADULT/adult.data')
```

In [214…
```
adult
```

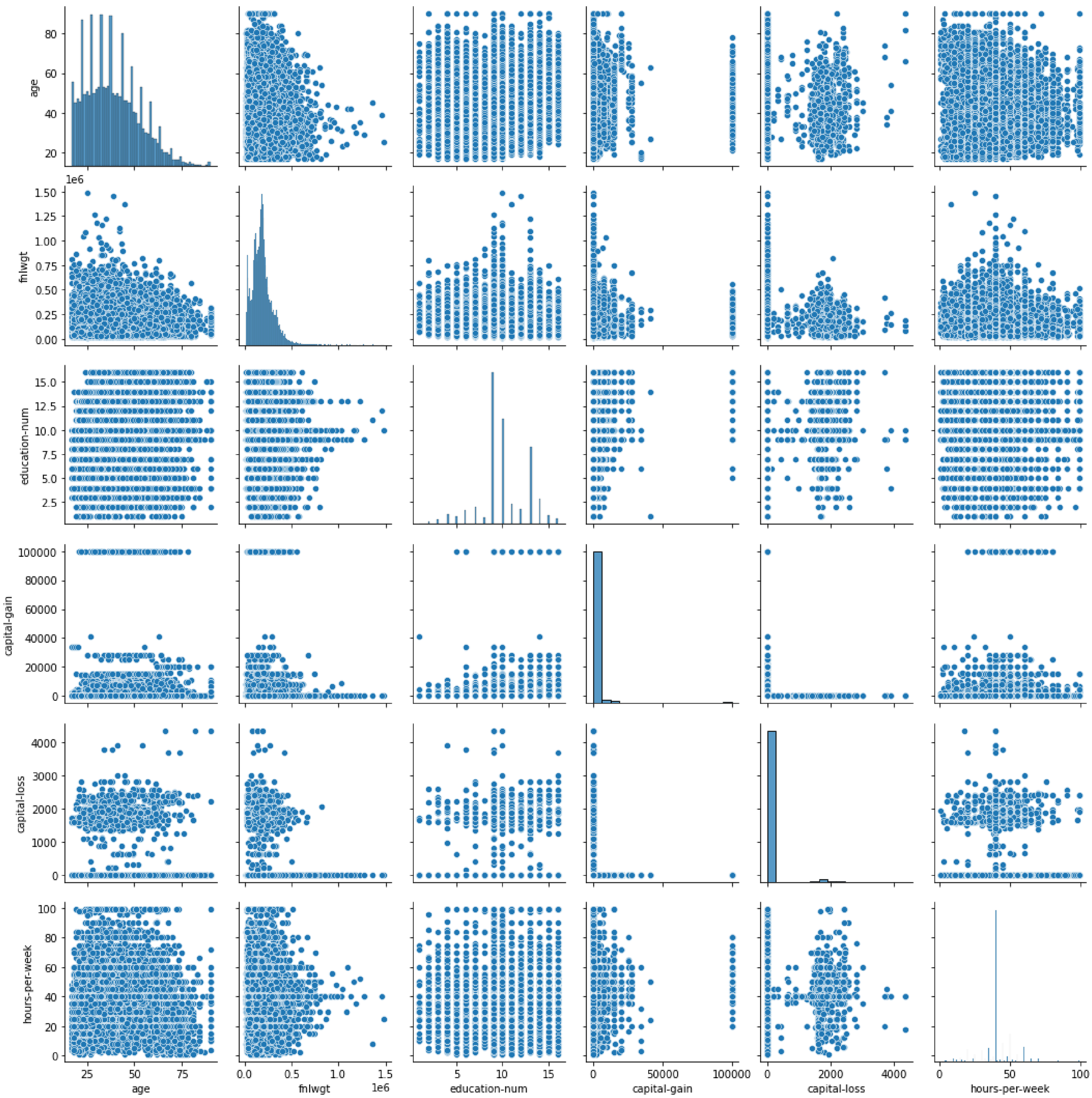
Out[214…

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income_group
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
32556	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K
32557	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K
32558	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K
32559	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K
32560	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K

32561 rows × 15 columns

In [215…
```
sns.pairplot(adult)
```

Out[215… <seaborn.axisgrid.PairGrid at 0x7ff2fe494e20>



In [13]:
```
# as we can see, adult has no null values
print('# OF ROWS WITH MISSING VALUES:', len(adult[adult.isnull().any(axis=1)]))
```

```
# OF ROWS WITH MISSING VALUES: 0
```

In [14]:
```
print("all columns: ", list(adult.columns))

for column in adult.columns.values:
        if len(adult[column].unique()) < 50:
            print(f"{column}: ", adult[column].unique())

 # drop id's and rownumber
```

```
all columns:  ['age', 'workclass', 'fnlwgt', 'education', 'education-num', 'marital-status', 'occupation', 'relationship', 'race', 's
ex', 'capital-gain', 'capital-loss', 'hours-per-week', 'native-country', 'income_group']
workclass:  [' State-gov' ' Self-emp-not-inc' ' Private' ' Federal-gov' ' Local-gov'
 ' ?' ' Self-emp-inc' ' Without-pay' ' Never-worked']
education:  [' Bachelors' ' HS-grad' ' 11th' ' Masters' ' 9th' ' Some-college'
 ' Assoc-acdm' ' Assoc-voc' ' 7th-8th' ' Doctorate' ' Prof-school'
 ' 5th-6th' ' 10th' ' 1st-4th' ' Preschool' ' 12th']
education-num:  [13  9  7 14  5 10 12 11  4 16 15  3  6  2  1  8]
marital-status:  [' Never-married' ' Married-civ-spouse' ' Divorced'
 ' Married-spouse-absent' ' Separated' ' Married-AF-spouse' ' Widowed']
occupation:  [' Adm-clerical' ' Exec-managerial' ' Handlers-cleaners' ' Prof-specialty'
 ' Other-service' ' Sales' ' Craft-repair' ' Transport-moving'
 ' Farming-fishing' ' Machine-op-inspct' ' Tech-support' ' ?'
 ' Protective-serv' ' Armed-Forces' ' Priv-house-serv']
relationship:  [' Not-in-family' ' Husband' ' Wife' ' Own-child' ' Unmarried'
 ' Other-relative']
race:  [' White' ' Black' ' Asian-Pac-Islander' ' Amer-Indian-Eskimo' ' Other']
sex:  [' Male' ' Female']
native-country:  [' United-States' ' Cuba' ' Jamaica' ' India' ' ?' ' Mexico' ' South'
 ' Puerto-Rico' ' Honduras' ' England' ' Canada' ' Germany' ' Iran'
 ' Philippines' ' Italy' ' Poland' ' Columbia' ' Cambodia' ' Thailand'
 ' Ecuador' ' Laos' ' Taiwan' ' Haiti' ' Portugal' ' Dominican-Republic'
 ' El-Salvador' ' France' ' Guatemala' ' China' ' Japan' ' Yugoslavia'
 ' Peru' ' Outlying-US(Guam-USVI-etc)' ' Scotland' ' Trinadad&Tobago'
```

```
 ' Greece' ' Nicaragua' ' Vietnam' ' Hong' ' Ireland' ' Hungary'
 ' Holand-Netherlands']
income_group:  [' <=50K' ' >50K']
```

In [15]:
```
# already one-hot encoded: N/A
adult_standardized_features = ['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week']
adult_categorical_features = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-countr
#adult_label_feature = ['income_group']


adult_transformer = ColumnTransformer(
                transformers=[
                    ('std', StandardScaler(), adult_standardized_features),
                    ('onehot', OneHotEncoder(), adult_categorical_features),
#                     ('label', LabelEncoder(), adult_label_feature)
                ], remainder='passthrough', sparse_threshold=0)

adult['income_group'] = LabelEncoder().fit_transform(adult['income_group'])

adult_trans = adult_transformer.fit_transform(adult)
print(f"adult_trans.shape: {adult_trans.shape}")
X_adult = adult_trans[:, :-1]
print(f"X_adult.shape: {X_adult.shape}")
y_adult = adult_trans[:, -1]
print(f"y_adult.shape: {y_adult.shape}")
print(f"all(y_adult == adult['income_group']): {all(y_adult == adult['income_group'])}")
```

```
adult_trans.shape: (32561, 109)
X_adult.shape: (32561, 108)
y_adult.shape: (32561,)
all(y_adult == adult['income_group']): True
```

cleaning up COV_TYPE

In [216…
```
# Dataset 4: https://archive.ics.uci.edu/ml/datasets/covertype
# filename: COV_TYPE/covtype.data
cov_type = pd.read_csv('datasets/COV_TYPE/covtype.data')
```

In [217…
```
cov_type
```

Out[217…

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hillshade_N
0	2596	51	3	258	0	510	221	
1	2590	56	2	212	-6	390	220	
2	2804	139	9	268	65	3180	234	
3	2785	155	18	242	118	3090	238	
4	2595	45	2	153	-1	391	220	
…	…	…	…	…	…	…	…	
581007	2396	153	20	85	17	108	240	
581008	2391	152	19	67	12	95	240	
581009	2386	159	17	60	7	90	236	
581010	2384	170	15	60	5	90	230	
581011	2383	165	13	60	4	67	231	

581012 rows × 55 columns

In [ ]:
```
# cov_type is too big to pairplot
# sns.pairplot(cov_type)
```

In [202…
```
# as we can see, cov_type has no null values
print('# OF ROWS WITH MISSING VALUES:', len(cov_type[cov_type.isnull().any(axis=1)]))
```

```
# OF ROWS WITH MISSING VALUES: 0
```

In [203…
```
# Just like Caruana, let's turn this into a binary classification problem and
# only keep the rows of the most prevalent cover type in the dataset
most_prevalent_cover_type = -1; most_prevalence = -1
for cover_type in np.sort(cov_type['Cover_Type'].unique()):
    curr_prevalence = len(cov_type[cov_type['Cover_Type'] == cover_type])
    if curr_prevalence > most_prevalence:
        most_prevalent_cover_type = cover_type; most_prevalence = curr_prevalence
    print(f"% of rows with cover type {cover_type}: ", f"{100*curr_prevalence/len(cov_type):.2f}%")

print(f"Setting all rows with cover type {most_prevalent_cover_type} as the positive class (1) and the rest as the negative class (0)
cov_type['Cover_Type'] = np.array(cov_type['Cover_Type'] == most_prevalent_cover_type, dtype="int")
cov_type
```

```
% of rows with cover type 1:  36.46%
% of rows with cover type 2:  48.76%
% of rows with cover type 3:  6.15%
% of rows with cover type 4:  0.47%
% of rows with cover type 5:  1.63%
% of rows with cover type 6:  2.99%
% of rows with cover type 7:  3.53%
Setting all rows with cover type 2 as the positive class (1) and the rest as the negative class (0).
```

Out[203…

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hillshade_N
0	2596	51	3	258	0	510	221	
1	2590	56	2	212	-6	390	220	
2	2804	139	9	268	65	3180	234	
3	2785	155	18	242	118	3090	238	

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hillshade_N
4	2595	45	2	153	-1	391	220	
...	...	...	...	...	...	...	...	
581007	2396	153	20	85	17	108	240	
581008	2391	152	19	67	12	95	240	
581009	2386	159	17	60	7	90	236	
581010	2384	170	15	60	5	90	230	
581011	2383	165	13	60	4	67	231	

581012 rows × 55 columns

In [20]:
```
print("all columns: ", list(cov_type.columns))

for column in cov_type.columns.values:
        if len(cov_type[column].unique()) < 50:
            vals = cov_type[column].unique()
            vals.sort()
            print(f"{column}: ", vals)
```

```
all columns:  ['Elevation', 'Aspect', 'Slope', 'Horizontal_Distance_To_Hydrology', 'Vertical_Distance_To_Hydrology', 'Horizontal_Dist
ance_To_Roadways', 'Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm', 'Horizontal_Distance_To_Fire_Points', 'Wilderness_Area1', 'Wil
derness_Area2', 'Wilderness_Area3', 'Wilderness_Area4', 'soil_type1', 'soil_type2', 'soil_type3', 'soil_type4', 'soil_type5', 'soil_t
ype6', 'soil_type7', 'soil_type8', 'soil_type9', 'soil_type10', 'soil_type11', 'soil_type12', 'soil_type13', 'soil_type14', 'soil_typ
e15', 'soil_type16', 'soil_type17', 'soil_type18', 'soil_type19', 'soil_type20', 'soil_type21', 'soil_type22', 'soil_type23', 'soil_t
ype24', 'soil_type25', 'soil_type26', 'soil_type27', 'soil_type28', 'soil_type29', 'soil_type30', 'soil_type31', 'soil_type32', 'soil
_type33', 'soil_type34', 'soil_type35', 'soil_type36', 'soil_type37', 'soil_type38', 'soil_type39', 'soil_type40', 'Cover_Type']
Wilderness_Area1:  [0 1]
Wilderness_Area2:  [0 1]
Wilderness_Area3:  [0 1]
Wilderness_Area4:  [0 1]
soil_type1:  [0 1]
soil_type2:  [0 1]
soil_type3:  [0 1]
soil_type4:  [0 1]
soil_type5:  [0 1]
soil_type6:  [0 1]
soil_type7:  [0 1]
soil_type8:  [0 1]
soil_type9:  [0 1]
soil_type10:  [0 1]
soil_type11:  [0 1]
soil_type12:  [0 1]
soil_type13:  [0 1]
soil_type14:  [0 1]
soil_type15:  [0 1]
soil_type16:  [0 1]
soil_type17:  [0 1]
soil_type18:  [0 1]
soil_type19:  [0 1]
soil_type20:  [0 1]
soil_type21:  [0 1]
soil_type22:  [0 1]
soil_type23:  [0 1]
soil_type24:  [0 1]
soil_type25:  [0 1]
soil_type26:  [0 1]
soil_type27:  [0 1]
soil_type28:  [0 1]
soil_type29:  [0 1]
soil_type30:  [0 1]
soil_type31:  [0 1]
soil_type32:  [0 1]
soil_type33:  [0 1]
soil_type34:  [0 1]
soil_type35:  [0 1]
soil_type36:  [0 1]
soil_type37:  [0 1]
soil_type38:  [0 1]
soil_type39:  [0 1]
soil_type40:  [0 1]
Cover_Type:  [0 1]
```

In [21]:
```
# most columns are already one-hot encoded
cov_type_standardized_features = ['Elevation', 'Aspect', 'Slope', 'Horizontal_Distance_To_Hydrology',
                                  'Vertical_Distance_To_Hydrology', 'Horizontal_Distance_To_Roadways',
                                  'Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm',
                                  'Horizontal_Distance_To_Fire_Points']


cov_type_transformer = ColumnTransformer(
                transformers=[
                    ('std', StandardScaler(), cov_type_standardized_features),
                ], remainder='passthrough')


cov_type_trans = cov_type_transformer.fit_transform(cov_type)
print(f"cov_type_trans.shape: {cov_type_trans.shape}")
X_cov_type = cov_type_trans[:, :-1]
print(f"X_cov_type.shape: {X_cov_type.shape}")
y_cov_type = cov_type_trans[:, -1]
print(f"y_cov_type.shape: {y_cov_type.shape}")
print(f"all(y_cov_type == cov_type['Cover_Type']): {all(y_cov_type == cov_type['Cover_Type'])}")
```

```
cov_type_trans.shape: (581012, 55)
X_cov_type.shape: (581012, 54)
y_cov_type.shape: (581012,)
all(y_cov_type == cov_type['Cover_Type']): True
```

Testing Algorithms

In [22]:
```
# helper function for analyzing test set performance
def getClfBestMetricScore(clf, metric_name, cv_results_, X_train, y_train, X_test, y_test):
    metric_best_index = np.argmin(cv_results_[f'rank_test_{metric_name}'])
    metric_best_params = cv_results_['params'][metric_best_index]
    clf.set_params(**metric_best_params)
    print(f'best {clf} parameters for {metric_name}: {metric_best_params}')
    clf.fit(X_train, y_train)
    clf_best_metric_score = SCORERS[metric_name](clf, X_test, y_test)
    print(f'test set {metric_name} score: {clf_best_metric_score}')
    return clf_best_metric_score
```

In [23]:
```
# analysis variables

NUM_TRIALS = 5
MAX_ITER = 100000

# SVC param search spaces
test_svc_search_space = [{'kernel': ['linear', 'rbf'],
                          'C': np.logspace(-1, 1, 3)},
                         ]
full_svc_search_space = [{'kernel': ['rbf', 'sigmoid'],
                          'gamma': ['scale', 'auto'],
                          'C': np.logspace(-5, 4, 10)},
                         {'kernel': ['poly'],
                          'gamma': ['scale', 'auto'],
                          'C': np.logspace(-5, 4, 10),
                          'degree': [2, 3]},
                        {'kernel': ['linear'],
                         'C': np.logspace(-5, 4, 10)}
                        ]

# LOGREG param search spaces
test_logreg_search_space = [{'penalty': ['l1', 'l2'],
                             'C': np.logspace(-1, 1, 3)}
                            ]
full_logreg_search_space = [{'penalty': ['l1', 'l2'],
                             'C': np.logspace(-5, 4, 10)},
                            {'penalty': ['none']}
                            ]

# RF param search spaces
full_rf_search_space = [{'max_features': ['auto', 'sqrt', 'log2'],
                         'criterion': ['gini', 'entropy']}]

data_tups = [(X_surgical, y_surgical, 'SURGICAL'), (X_churn, y_churn, 'CHURN'), (X_adult, y_adult, 'ADULT'), (X_cov_type, y_cov_type,
classifiers_tups = [(SVC(max_iter=MAX_ITER), full_svc_search_space, 'SVM'),
                    (LogisticRegression(max_iter=MAX_ITER, solver='saga'), full_logreg_search_space, 'LOGREG'),
                    (RandomForestClassifier(n_estimators=1024), full_rf_search_space, 'RF' )]

best_scores = {}
all_cv_results = {}
metrics = ['accuracy', 'f1', 'roc_auc']
```

In [ ]:
```
%%time

for X_data, y_data, data_name in data_tups:
    for clf, clf_search_space, clf_name in classifiers_tups:
        for trial in range(NUM_TRIALS):
            print(f"Tuning hyperparameters for {clf_name} on {data_name} dataset [trial {trial}]...")
            X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, train_size=5000, random_state=trial) # downsample maj
            grid = GridSearchCV(clf, clf_search_space, cv=StratifiedKFold(n_splits=5),
                    scoring=metrics, refit=False, verbose=0).fit(X_train, y_train)

            # save grid search results
            if data_name not in all_cv_results:

                all_cv_results[data_name] = {}
            if clf_name not in all_cv_results[data_name]:
                all_cv_results[data_name][clf_name] = [None]*NUM_TRIALS
            all_cv_results[data_name][clf_name][trial] = grid.cv_results_


            # save best scores
            if data_name not in best_scores:
                best_scores[data_name] = {}
            if clf_name not in best_scores[data_name]:
                best_scores[data_name][clf_name] = {}
            for metric in metrics:
                if metric not in best_scores[data_name][clf_name]:
                    best_scores[data_name][clf_name][metric] = [0]*NUM_TRIALS
                clf_best_metric_score = getClfBestMetricScore(clf, metric, grid.cv_results_, X_train, y_train, X_test, y_test)
                best_scores[data_name][clf_name][metric][trial] = clf_best_metric_score
```

In [25]:
```
# save results
np.save('all_cv_results.npy', all_cv_results)
np.save('best_scores.npy', best_scores)
#temp = np.load('best_scores.npy',allow_pickle='TRUE').item()
```

In [ ]:
```
all_cv_results
```

In [27]:
```
# pretty print a nested dictionary
def pretty(d, indent=0):
    for key, value in d.items():
        print('\t' * indent + str(key))
        if isinstance(value, dict):
            pretty(value, indent+1)
        else:
            print('\t' * (indent+1) + str(value))
pretty(best_scores)
```

```
SURGICAL
        SVM
                accuracy
                        [0.8, 0.7990659055526725, 0.7970939283860924, 0.8029060716139076, 0.8052932018681889]
                f1
                        [0.4976481655691439, 0.49411764705882355, 0.4981751824817517, 0.4956231763234681, 0.5049573437860272]
                roc_auc
                        [0.8013164879982685, 0.7941297956952895, 0.7924771842638257, 0.797931176322326, 0.8001148892057983]
        LOGREG
                accuracy
                        [0.7925272444213803, 0.7949143746756616, 0.7949143746756616, 0.7973015049299429, 0.7932537623248573]
                f1
                        [0.4799999999999999, 0.4646194926568759, 0.476215644820296, 0.48292295472597296, 0.48888888888888893]
                roc_auc
                        [0.819725902550724, 0.8131976934480281, 0.8115785909035851, 0.8144507553180074, 0.8131537831537832]
        RF
                accuracy
                        [0.8400622729631552, 0.8384016606123508, 0.8377789309807991, 0.8512714063310846, 0.8518941359626362]
                f1
                        [0.6038994356080041, 0.5967947087255151, 0.5983975963945919, 0.6422662321383806, 0.6430884184308842]
                roc_auc
                        [0.8973450654484558, 0.8940556624124507, 0.8933214553346723, 0.8978034129103456, 0.8973485278030732]
CHURN
        SVM
                accuracy
                        [0.8596, 0.8542, 0.8608, 0.8554, 0.865]
                f1
                        [0.578125, 0.5605786618444847, 0.5514612452350699, 0.5611077664057796, 0.5664811379097093]
                roc_auc
                        [0.8455552076596914, 0.8273735420584677, 0.8352937904395175, 0.8382577647574889, 0.8358604247158465]
        LOGREG
                accuracy
                        [0.8082, 0.8146, 0.8078, 0.809, 0.8178]
                f1
                        [0.3121881682109765, 0.31407407407407406, 0.27756939234808703, 0.320863309352518, 0.326440177252585]
                roc_auc
                        [0.7744076585210264, 0.772230511973049, 0.7694798109041088, 0.7685716620029005, 0.7697154752877644]
        RF
                accuracy
                        [0.8654, 0.8604, 0.857, 0.8614, 0.867]
                f1
                        [0.597950572634117, 0.5708978328173374, 0.5435897435897437, 0.5782396088019559, 0.586833855799373]
                roc_auc
                        [0.8656657019872006, 0.8581453178604398, 0.8534457982992193, 0.851925299011126, 0.8607396619444811]
ADULT
        SVM
                accuracy
                        [0.8529443779253293, 0.8526541126954755, 0.8508036718551576, 0.8529080947715976, 0.8525815463880121]
                f1
                        [0.6533871383188805, 0.662624938554809, 0.6567287784679088, 0.6600374928682045, 0.6511129431162407]
                roc_auc
                        [0.9052181684494194, 0.9030829581439591, 0.9042858296515394, 0.9046148308105365, 0.9051965215239024]
        LOGREG
                accuracy
                        [0.8523275643118899, 0.8486992489387177, 0.8465222597148144, 0.8517833170059141, 0.8505496897790356]
                f1
                        [0.6535406455974312, 0.6599409448818897, 0.6459658520254437, 0.6579013482957876, 0.6588254783400977]
                roc_auc
                        [0.9047524994904877, 0.9049963316559452, 0.9033740885204125, 0.9059388302863854, 0.9044353322458085]
        RF
                accuracy
                        [0.8538514567686223, 0.8488080983999129, 0.8485178331700591, 0.8533797757701099, 0.8503319908566452]
                f1
                        [0.6656177542232528, 0.663581750284137, 0.6563196565673244, 0.6705344010459225, 0.6632919660352711]
                roc_auc
                        [0.9032429786336142, 0.8993862554550104, 0.8991537737594484, 0.9020705030349222, 0.9015671887130629]
COV_TYPE
        SVM
                accuracy
                        [0.8057696714651779, 0.8009850489225919, 0.7970997132004194, 0.8009433831239627, 0.8019798198648639]
                f1
                        [0.8077419443532713, 0.8031337852203078, 0.7925676263295865, 0.8040362124275555, 0.8041901138856749]
                roc_auc
                        [0.8755575611637495, 0.8735610173110151, 0.8693740993640505, 0.8755905436766463, 0.873035558681097]
        LOGREG
                accuracy
                        [0.7499027798031985, 0.7541822045374055, 0.7541822045374055, 0.7592237661715381, 0.7536856871037408]
                f1
                        [0.751264302721294, 0.7531481710708227, 0.7487302888659377, 0.7589676746611053, 0.755067655050737]
                roc_auc
                        [0.8222331622015479, 0.8244847914569645, 0.8247172298247956, 0.8238643133494374, 0.8243948564584195]
        RF
                accuracy
                        [0.8211408790094651, 0.8212710846301813, 0.8163163267431929, 0.8237224224495323, 0.8163961861905654]
                f1
                        [0.8212779106788836, 0.8227943014246438, 0.8121986203856594, 0.8244929554588715, 0.8167130919220055]
                roc_auc
                        [0.9013735595573694, 0.9020952401984305, 0.8985431865786819, 0.9038809555438783, 0.8963870832199677]
```

Analyzing Results

In [42]:

```
# why did I use a nested dictionary?? Bad Isaac. Baaaaaad Isaac. I should have just made a 2D pandas dataframe...
# sigh... Guess I'll just convert it.
# columns: dataset | classifier | metric | trial | score
data = {'dataset':[], 'classifier':[], 'metric':[], 'trial':[], 'score':[]}
print('dataset\tclassifier\tmetric\ttrial\tscore')
for data_name in best_scores:
    for clf_name in best_scores[data_name]:
        for metric in best_scores[data_name][clf_name]:
            for trial, score in enumerate(best_scores[data_name][clf_name][metric]):
                #print(f"{data_name}\t{clf_name}\t{metric}\t{trial}\t{score:.4f}")
                data['dataset'].append(data_name)
                data['classifier'].append(clf_name)
                data['metric'].append(metric)
                data['trial'].append(trial+1)
                data['score'].append(score)
```

```
best_score_df = pd.DataFrame(data, columns=list(data.keys()))
best_score_df
```

Out[42]:

	dataset	classifier	metric	trial	score
0	SURGICAL	SVM	accuracy	1	0.800000
1	SURGICAL	SVM	accuracy	2	0.799066
2	SURGICAL	SVM	accuracy	3	0.797094
3	SURGICAL	SVM	accuracy	4	0.802906
4	SURGICAL	SVM	accuracy	5	0.805293
...	...	...	...	...	...
175	COV_TYPE	RF	roc_auc	1	0.901374
176	COV_TYPE	RF	roc_auc	2	0.902095
177	COV_TYPE	RF	roc_auc	3	0.898543
178	COV_TYPE	RF	roc_auc	4	0.903881
179	COV_TYPE	RF	roc_auc	5	0.896387

180 rows × 5 columns

Generate Table 1 (description of problems)

In [100…

```
# Generate table 1
table1 = [["PROBLEM", "#ATTR", "TRAIN SIZE", "TEST SIZE", "%POZ"]]
for X_data, Y_data, data_name in data_tups:
    #               PROBLEM       #ATTR              TRAIN SIZE   TEST SIZE              %POZ
    dataset_row = [data_name, X_data.shape[1], 5000, len(X_data) - 5000, f"{int(100*np.mean(Y_data))}%"]
    table1.append(dataset_row)

#latexify the table
table1_latex = tabulate(table1, tablefmt="latex")

# add in the other horizontal line
table1_latex_list = table1_latex.split("\n")
table1_latex_list.insert(3, "\hline")
table1_latex = "\n".join(table1_latex_list)
display(Latex(rf"LaTex for Table 1: {table1_latex}"))
```

LaTex for Table 1:

```
\begin{tabular}{lllll}
\hline
 PROBLEM  & \#ATTR & TRAIN SIZE & TEST SIZE & \%POZ \\
\hline
 SURGICAL & 45    & 5000       & 9635      & 25\%  \\
 CHURN    & 13    & 5000       & 5000      & 20\%  \\
 ADULT    & 108   & 5000       & 27561     & 24\%  \\
 COV\_TYPE & 54    & 5000       & 576012    & 48\%  \\
\hline
\end{tabular}
```

Print table 2 data

In [48]:

```
pd.pivot_table(best_score_df.drop(['trial'],axis=1), columns=['classifier', 'metric'])
```

Out[48]:

classifier	LOGREG			RF			SVM		
metric	accuracy	f1	roc_auc	accuracy	f1	roc_auc	accuracy	f1	roc_auc
score	0.802568	0.549357	0.828485	0.844217	0.668939	0.888875	0.828401	0.630192	0.852891

MEAN classifier score over all metrics and data data (MEAN column in Table 2 and Table 3)

In [51]:

```
pd.pivot_table(best_score_df.drop(['trial'],axis=1), columns=['classifier'])
```

Out[51]:

classifier	LOGREG	RF	SVM
score	0.726803	0.800677	0.770495

Check for statistically insignificant differences in table 2

In [208…

```
for metric in best_score_df['metric'].unique():
    print(f"Checking for statistically insignificant differences between classifiers for {metric.upper()[:3]}...")
    # RF is highest scoring classifier
    best_scores_dist = np.array(best_score_df[(best_score_df['classifier'] == 'RF') & (best_score_df['metric'] == metric)]['score'])
    print(f"\tbest (RF) score distribution: \n{best_scores_dist}")
    for clf_name in best_score_df['classifier'].unique():
        if clf_name == 'RF': continue # don't need to check RF against itself
        curr_clf_scores_dist = np.array(best_score_df[(best_score_df['classifier'] == clf_name) & (best_score_df['metric'] == metric)
        t, p = ttest_rel(curr_clf_scores_dist, best_scores_dist)
        print(f"\tchecking against {clf_name} scores dist: p = {p}", ("(\033[1mINSIGNIFICANT DIFFERENCE\033[0m)" if p>0.05 else ""))
```

```
Checking for statistically insignificant differences between classifiers for ACC...
        best (RF) score distribution:
[0.84006227 0.83840166 0.83777893 0.85127141 0.85189414 0.8654
 0.8604     0.857      0.8614     0.867      0.85385146 0.8488081
```

```
 0.84851783 0.85337978 0.85033199 0.82114088 0.82127108 0.81631633
 0.82372242 0.81639619]
        checking against SVM scores dist: p = 0.0009374284375567621
        checking against LOGREG scores dist: p = 5.563756920870249e-07
Checking for statistically insignificant differences between classifiers for FSC...
        best (RF) score distribution:
[0.60389944 0.59679471 0.5983976  0.64226623 0.64308842 0.59795057
 0.57089783 0.54358974 0.57823961 0.58683386 0.66561775 0.66358175
 0.65631966 0.6705344  0.66329197 0.82127791 0.8227943  0.81219862
 0.82449296 0.81671309]
        checking against SVM scores dist: p = 0.0022100417911829545
        checking against LOGREG scores dist: p = 3.2093545999099624e-05
Checking for statistically insignificant differences between classifiers for ROC...
        best (RF) score distribution:
[0.89734507 0.89405566 0.89332146 0.89780341 0.89734853 0.8656657
 0.85814532 0.8534458  0.8519253  0.86073966 0.90324298 0.89938626
 0.89915377 0.9020705  0.90156719 0.90137356 0.90209524 0.89854319
 0.90388096 0.89638708]
        checking against SVM scores dist: p = 0.0005967990931783062
        checking against LOGREG scores dist: p = 1.0002672492034258e-06
```

Checking for statistically insignificant differences between classifiers for MEAN column

In [91]:
```
pd.pivot_table(best_score_df, columns=['classifier', 'trial'])
```

Out[91]:

classifier	LOGREG					RF					SVM				
trial	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
score	0.726756	0.726591	0.721754	0.729232	0.729684	0.803069	0.798053	0.792882	0.805082	0.804299	0.773572	0.768792	0.76718	0.77078	0.77215

In [207…
```
logreg_means = [0.726756, 0.726591, 0.721754, 0.729232, 0.729684]
rf_means = [0.803069, 0.798053, 0.792882, 0.805082, 0.804299]
svm_means = [0.773572, 0.768792, 0.76718, 0.77078, 0.77215]

t, p = ttest_rel(svm_means, rf_means)
print(f'SVM v. RF MEAN p-value: {p}')
t, p = ttest_rel(logreg_means, rf_means)
print(f'LOGREG v. RF MEAN p-value: {p}')
t, p = ttest_rel(rf_means, rf_means)
print(f'RF v. RF MEAN p-value: {p}')
```

```
SVM v. RF MEAN p-value: 3.1745064444648126e-05
LOGREG v. RF MEAN p-value: 2.839359222293215e-07
RF v. RF MEAN p-value: nan
```

No insignificant differences in any columns for table 2! (so no asterisks in table 3)

Print table 3 data

In [114…
```
pd.pivot_table(best_score_df.drop(['trial'],axis=1), columns=['classifier', 'dataset'])
```

Out[114…

classifier	LOGREG				RF				SVM			
dataset	ADULT	CHURN	COV_TYPE	SURGICAL	ADULT	CHURN	COV_TYPE	SURGICAL	ADULT	CHURN	COV_TYPE	SURGICAL
score	0.803304	0.630863	0.777203	0.695844	0.80531	0.765242	0.846574	0.785582	0.804545	0.753006	0.825704	0.698723

MEAN classifier score over all metrics and data data (same as table 2 MEAN data)

In [115…
```
pd.pivot_table(best_score_df.drop(['trial'],axis=1), columns=['classifier'])
```

Out[115…

classifier	LOGREG	RF	SVM
score	0.726803	0.800677	0.770495

Check for statistically insignificant differences in table 3

In [120…
```
for data_name in best_score_df['dataset'].unique():
    print(f"Checking for statistically insignificant differences between classifiers for {data_name}...")
    # RF is highest scoring classifier
    best_scores_dist = np.array(best_score_df[(best_score_df['classifier'] == 'RF') & (best_score_df['dataset'] == data_name)]['score
    print(f"\tbest (RF) score distribution: \n{best_scores_dist}")
    for clf_name in best_score_df['classifier'].unique():
        if clf_name == 'RF': continue # don't need to check RF against itself
        curr_clf_scores_dist = np.array(best_score_df[(best_score_df['classifier'] == clf_name) & (best_score_df['dataset'] == data_n
        t, p = ttest_rel(curr_clf_scores_dist, best_scores_dist)
        print(f"\tchecking against {clf_name} scores dist: p = {p}", ("(\033[1mINSIGNIFICANT DIFFERENCE\033[0m)" if p>0.05 else ""))
```

```
Checking for statistically insignificant differences between classifiers for SURGICAL...
        best (RF) score distribution:
[0.84006227 0.83840166 0.83777893 0.85127141 0.85189414 0.60389944
 0.59679471 0.5983976  0.64226623 0.64308842 0.89734507 0.89405566
 0.89332146 0.89780341 0.89734853]
        checking against SVM scores dist: p = 1.67451870897372e-07
        checking against LOGREG scores dist: p = 4.3289703704382915e-07
Checking for statistically insignificant differences between classifiers for CHURN...
        best (RF) score distribution:
[0.8654     0.8604     0.857      0.8614     0.867      0.59795057
 0.57089783 0.54358974 0.57823961 0.58683386 0.8656657  0.85814532
 0.8534458  0.8519253  0.86073966]
        checking against SVM scores dist: p = 0.00062488232704701
        checking against LOGREG scores dist: p = 0.00010255111223237967
Checking for statistically insignificant differences between classifiers for ADULT...
        best (RF) score distribution:
[0.85385146 0.8488081  0.84851783 0.85337978 0.85033199 0.66561775
 0.66358175 0.65631966 0.6705344  0.66329197 0.90324298 0.89938626
 0.89915377 0.9020705  0.90156719]
        checking against SVM scores dist: p = 0.624345018707828 (INSIGNIFICANT DIFFERENCE)
        checking against LOGREG scores dist: p = 0.20074009754414918 (INSIGNIFICANT DIFFERENCE)
```

```
Checking for statistically insignificant differences between classifiers for COV_TYPE...
        best (RF) score distribution:
[0.82114088 0.82127108 0.81631633 0.82372242 0.81639619 0.82127791
 0.8227943  0.81219862 0.82449296 0.81671309 0.90137356 0.90209524
 0.89854319 0.90388096 0.89638708]
        checking against SVM scores dist: p = 6.173104581489482e-10
        checking against LOGREG scores dist: p = 2.6957326289809 79e-16
```

Since MEAN column for Table 3 is the same as MEAN column for Table 2, we can use the results of our significance testing for the MEAN column of Table 2 so no further significance testing is required

Table 3 has two insignificant differences in the ADULT column for both the SVM and logistic regression classifiers

Appendix tables

Except for p-values which I already calculated to create Table 2 and Table 3

Generating mean training set performance table

In [210…

```python
# helper function for calculating training set performance
def getClfMetricBestTrainScore(clf, metric_name, cv_results_, X_train, y_train):
    metric_best_index = np.argmin(cv_results_[f'rank_test_{metric_name}'])
    metric_best_params = cv_results_['params'][metric_best_index]
    clf.set_params(**metric_best_params)
    print(f'best {clf} parameters for {metric_name}: {metric_best_params}')
    clf.fit(X_train, y_train)
    clf_best_train_metric_score = SCORERS[metric_name](clf, X_train, y_train)
    print(f'training set {metric_name} score: {clf_best_train_metric_score}')
    return clf_best_train_metric_score
```

In [ ]:

```python
%%time

# Since I didn't store the training set performance when I initially testing the algorithms, I can use results of
# the gridsearches I already performed to quickly get the training set performances of the algorithms
training_scores_data = {'dataset':[], 'classifier':[], 'metric':[], 'trial':[], 'score':[]}

for X_data, y_data, data_name in data_tups:
    for clf, clf_search_space, clf_name in classifiers_tups:
        for trial in range(NUM_TRIALS):
            print(f"Evaluating training set performance for {clf_name} on {data_name} dataset [trial {trial}]...")

            X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, train_size=5000, random_state=trial) # downsample maj
            cv_results_ = all_cv_results[data_name][clf_name][trial]


            for metric in metrics:
                clf_best_train_metric_score = getClfMetricBestTrainScore(clf, metric, cv_results_, X_train, y_train)
                training_scores_data['dataset'].append(data_name)
                training_scores_data['classifier'].append(clf_name)
                training_scores_data['metric'].append(metric)
                training_scores_data['trial'].append(trial + 1)
                training_scores_data['score'].append(clf_best_train_metric_score)
```

In [187…

```python
train_best_scores_df = pd.DataFrame(training_scores_data, columns=list(data.keys()))
train_best_scores_df
```

Out[187…

	dataset	classifier	metric	trial	score
0	SURGICAL	SVM	accuracy	1	0.833600
1	SURGICAL	SVM	f1	1	0.905660
2	SURGICAL	SVM	roc_auc	1	0.797776
3	SURGICAL	SVM	accuracy	2	0.832600
4	SURGICAL	SVM	f1	2	0.522013
…	…	…	…	…	…
175	COV_TYPE	RF	f1	4	1.000000
176	COV_TYPE	RF	roc_auc	4	1.000000
177	COV_TYPE	RF	accuracy	5	1.000000
178	COV_TYPE	RF	f1	5	1.000000
179	COV_TYPE	RF	roc_auc	5	1.000000

180 rows × 5 columns

Training Set Table 2 Data

In [188…

```python
pd.pivot_table(train_best_scores_df.drop(['trial'],axis=1), columns=['classifier', 'metric'])
```

Out[188…

classifier	LOGREG			RF			SVM		
metric	**accuracy**	**f1**	**roc_auc**	**accuracy**	**f1**	**roc_auc**	**accuracy**	**f1**	**roc_auc**
score	0.8072	0.557456	0.833891	1.0	1.0	1.0	0.86181	0.771315	0.886564

Training Set MEAN column data

In [190…

```python
pd.pivot_table(train_best_scores_df.drop(['trial'],axis=1), columns=['classifier'])
```

Out[190…

classifier	LOGREG	RF	SVM
score	0.732849	1.0	0.839896

Training Set Table 3 Data

In [189…
```
pd.pivot_table(train_best_scores_df.drop(['trial'],axis=1), columns=['classifier', 'dataset'])
```

Out[189…

classifier	LOGREG				RF				SVM			
dataset	ADULT	CHURN	COV_TYPE	SURGICAL	ADULT	CHURN	COV_TYPE	SURGICAL	ADULT	CHURN	COV_TYPE	SURGICAL
score	0.813353	0.627368	0.785881	0.704795	1.0	1.0	1.0	1.0	0.843761	0.789278	0.896822	0.829723

Appendix table with raw test set scores:

In [ ]:
```
# convert metric names
metric2short = {'accuracy': 'ACC', 'f1': 'FSC', 'roc_auc': 'ROC'}
best_score_df['metric'] = best_score_df['metric'].apply(lambda name: metric2short[name])
```

In [151…
```
''' Failed attempt to put the raw test score table as two tables side by side
i = 1
table = ""
while 20*(i-1) < len(best_score_df):
    print(f'40*(i-1): {20*(i-1)}')
    #print(f'table: {table}')
    table += "\n" + best_score_df[(i-1)*20:i*20].to_latex(longtable=True, float_format="{:0.3f}".format) + "\quad"
    i += 1
Latex(table)
'''
Latex(best_score_df.to_latex(longtable=True, float_format="{:0.3f}".format))
```

Out[151…

```
\begin{longtable}{lllrr}
\toprule
{} &   dataset & classifier & metric &  trial &  score \\
\midrule
\endhead
\midrule
\multicolumn{6}{r}{{Continued on next page}} \\
\midrule
\endfoot

\bottomrule
\endlastfoot
0   &  SURGICAL &       SVM &    ACC &      1 &  0.800 \\
1   &  SURGICAL &       SVM &    ACC &      2 &  0.799 \\
2   &  SURGICAL &       SVM &    ACC &      3 &  0.797 \\
3   &  SURGICAL &       SVM &    ACC &      4 &  0.803 \\
4   &  SURGICAL &       SVM &    ACC &      5 &  0.805 \\
5   &  SURGICAL &       SVM &    FSC &      1 &  0.498 \\
6   &  SURGICAL &       SVM &    FSC &      2 &  0.494 \\
7   &  SURGICAL &       SVM &    FSC &      3 &  0.498 \\
8   &  SURGICAL &       SVM &    FSC &      4 &  0.496 \\
9   &  SURGICAL &       SVM &    FSC &      5 &  0.505 \\
10  &  SURGICAL &       SVM &    ROC &      1 &  0.801 \\
11  &  SURGICAL &       SVM &    ROC &      2 &  0.794 \\
12  &  SURGICAL &       SVM &    ROC &      3 &  0.792 \\
13  &  SURGICAL &       SVM &    ROC &      4 &  0.798 \\
14  &  SURGICAL &       SVM &    ROC &      5 &  0.800 \\
15  &  SURGICAL &    LOGREG &    ACC &      1 &  0.793 \\
16  &  SURGICAL &    LOGREG &    ACC &      2 &  0.795 \\
17  &  SURGICAL &    LOGREG &    ACC &      3 &  0.795 \\
18  &  SURGICAL &    LOGREG &    ACC &      4 &  0.797 \\
19  &  SURGICAL &    LOGREG &    ACC &      5 &  0.793 \\
20  &  SURGICAL &    LOGREG &    FSC &      1 &  0.480 \\
21  &  SURGICAL &    LOGREG &    FSC &      2 &  0.465 \\
22  &  SURGICAL &    LOGREG &    FSC &      3 &  0.476 \\
23  &  SURGICAL &    LOGREG &    FSC &      4 &  0.483 \\
24  &  SURGICAL &    LOGREG &    FSC &      5 &  0.489 \\
25  &  SURGICAL &    LOGREG &    ROC &      1 &  0.820 \\
26  &  SURGICAL &    LOGREG &    ROC &      2 &  0.813 \\
27  &  SURGICAL &    LOGREG &    ROC &      3 &  0.812 \\
28  &  SURGICAL &    LOGREG &    ROC &      4 &  0.814 \\
29  &  SURGICAL &    LOGREG &    ROC &      5 &  0.813 \\
30  &  SURGICAL &        RF &    ACC &      1 &  0.840 \\
31  &  SURGICAL &        RF &    ACC &      2 &  0.838 \\
32  &  SURGICAL &        RF &    ACC &      3 &  0.838 \\
33  &  SURGICAL &        RF &    ACC &      4 &  0.851 \\
34  &  SURGICAL &        RF &    ACC &      5 &  0.852 \\
35  &  SURGICAL &        RF &    FSC &      1 &  0.604 \\
36  &  SURGICAL &        RF &    FSC &      2 &  0.597 \\
37  &  SURGICAL &        RF &    FSC &      3 &  0.598 \\
38  &  SURGICAL &        RF &    FSC &      4 &  0.642 \\
39  &  SURGICAL &        RF &    FSC &      5 &  0.643 \\
40  &  SURGICAL &        RF &    ROC &      1 &  0.897 \\
41  &  SURGICAL &        RF &    ROC &      2 &  0.894 \\
42  &  SURGICAL &        RF &    ROC &      3 &  0.893 \\
43  &  SURGICAL &        RF &    ROC &      4 &  0.898 \\
44  &  SURGICAL &        RF &    ROC &      5 &  0.897 \\
45  &    CHURN &       SVM &    ACC &      1 &  0.860 \\
46  &    CHURN &       SVM &    ACC &      2 &  0.854 \\
47  &    CHURN &       SVM &    ACC &      3 &  0.861 \\
48  &    CHURN &       SVM &    ACC &      4 &  0.855 \\
49  &    CHURN &       SVM &    ACC &      5 &  0.865 \\
50  &    CHURN &       SVM &    FSC &      1 &  0.578 \\
51  &    CHURN &       SVM &    FSC &      2 &  0.561 \\
52  &    CHURN &       SVM &    FSC &      3 &  0.551 \\
```

53 & CHURN & SVM & FSC & 4 & 0.561 \\
54 & CHURN & SVM & FSC & 5 & 0.566 \\
55 & CHURN & SVM & ROC & 1 & 0.846 \\
56 & CHURN & SVM & ROC & 2 & 0.827 \\
57 & CHURN & SVM & ROC & 3 & 0.835 \\
58 & CHURN & SVM & ROC & 4 & 0.838 \\
59 & CHURN & SVM & ROC & 5 & 0.836 \\
60 & CHURN & LOGREG & ACC & 1 & 0.808 \\
61 & CHURN & LOGREG & ACC & 2 & 0.815 \\
62 & CHURN & LOGREG & ACC & 3 & 0.808 \\
63 & CHURN & LOGREG & ACC & 4 & 0.809 \\
64 & CHURN & LOGREG & ACC & 5 & 0.818 \\
65 & CHURN & LOGREG & FSC & 1 & 0.312 \\
66 & CHURN & LOGREG & FSC & 2 & 0.314 \\
67 & CHURN & LOGREG & FSC & 3 & 0.278 \\
68 & CHURN & LOGREG & FSC & 4 & 0.321 \\
69 & CHURN & LOGREG & FSC & 5 & 0.326 \\
70 & CHURN & LOGREG & ROC & 1 & 0.774 \\
71 & CHURN & LOGREG & ROC & 2 & 0.772 \\
72 & CHURN & LOGREG & ROC & 3 & 0.769 \\
73 & CHURN & LOGREG & ROC & 4 & 0.769 \\
74 & CHURN & LOGREG & ROC & 5 & 0.770 \\
75 & CHURN & RF & ACC & 1 & 0.865 \\
76 & CHURN & RF & ACC & 2 & 0.860 \\
77 & CHURN & RF & ACC & 3 & 0.857 \\
78 & CHURN & RF & ACC & 4 & 0.861 \\
79 & CHURN & RF & ACC & 5 & 0.867 \\
80 & CHURN & RF & FSC & 1 & 0.598 \\
81 & CHURN & RF & FSC & 2 & 0.571 \\
82 & CHURN & RF & FSC & 3 & 0.544 \\
83 & CHURN & RF & FSC & 4 & 0.578 \\
84 & CHURN & RF & FSC & 5 & 0.587 \\
85 & CHURN & RF & ROC & 1 & 0.866 \\
86 & CHURN & RF & ROC & 2 & 0.858 \\
87 & CHURN & RF & ROC & 3 & 0.853 \\
88 & CHURN & RF & ROC & 4 & 0.852 \\
89 & CHURN & RF & ROC & 5 & 0.861 \\
90 & ADULT & SVM & ACC & 1 & 0.853 \\
91 & ADULT & SVM & ACC & 2 & 0.853 \\
92 & ADULT & SVM & ACC & 3 & 0.851 \\
93 & ADULT & SVM & ACC & 4 & 0.853 \\
94 & ADULT & SVM & ACC & 5 & 0.853 \\
95 & ADULT & SVM & FSC & 1 & 0.653 \\
96 & ADULT & SVM & FSC & 2 & 0.663 \\
97 & ADULT & SVM & FSC & 3 & 0.657 \\
98 & ADULT & SVM & FSC & 4 & 0.660 \\
99 & ADULT & SVM & FSC & 5 & 0.651 \\
100 & ADULT & SVM & ROC & 1 & 0.905 \\
101 & ADULT & SVM & ROC & 2 & 0.903 \\
102 & ADULT & SVM & ROC & 3 & 0.904 \\
103 & ADULT & SVM & ROC & 4 & 0.905 \\
104 & ADULT & SVM & ROC & 5 & 0.905 \\
105 & ADULT & LOGREG & ACC & 1 & 0.852 \\
106 & ADULT & LOGREG & ACC & 2 & 0.849 \\
107 & ADULT & LOGREG & ACC & 3 & 0.847 \\
108 & ADULT & LOGREG & ACC & 4 & 0.852 \\
109 & ADULT & LOGREG & ACC & 5 & 0.851 \\
110 & ADULT & LOGREG & FSC & 1 & 0.654 \\
111 & ADULT & LOGREG & FSC & 2 & 0.660 \\
112 & ADULT & LOGREG & FSC & 3 & 0.646 \\
113 & ADULT & LOGREG & FSC & 4 & 0.658 \\
114 & ADULT & LOGREG & FSC & 5 & 0.659 \\
115 & ADULT & LOGREG & ROC & 1 & 0.905 \\
116 & ADULT & LOGREG & ROC & 2 & 0.905 \\
117 & ADULT & LOGREG & ROC & 3 & 0.903 \\
118 & ADULT & LOGREG & ROC & 4 & 0.906 \\
119 & ADULT & LOGREG & ROC & 5 & 0.904 \\
120 & ADULT & RF & ACC & 1 & 0.854 \\
121 & ADULT & RF & ACC & 2 & 0.849 \\
122 & ADULT & RF & ACC & 3 & 0.849 \\
123 & ADULT & RF & ACC & 4 & 0.853 \\
124 & ADULT & RF & ACC & 5 & 0.850 \\
125 & ADULT & RF & FSC & 1 & 0.666 \\
126 & ADULT & RF & FSC & 2 & 0.664 \\
127 & ADULT & RF & FSC & 3 & 0.656 \\
128 & ADULT & RF & FSC & 4 & 0.671 \\
129 & ADULT & RF & FSC & 5 & 0.663 \\
130 & ADULT & RF & ROC & 1 & 0.903 \\
131 & ADULT & RF & ROC & 2 & 0.899 \\
132 & ADULT & RF & ROC & 3 & 0.899 \\
133 & ADULT & RF & ROC & 4 & 0.902 \\
134 & ADULT & RF & ROC & 5 & 0.902 \\
135 & COV_TYPE & SVM & ACC & 1 & 0.806 \\
136 & COV_TYPE & SVM & ACC & 2 & 0.801 \\
137 & COV_TYPE & SVM & ACC & 3 & 0.797 \\
138 & COV_TYPE & SVM & ACC & 4 & 0.801 \\
139 & COV_TYPE & SVM & ACC & 5 & 0.802 \\
140 & COV_TYPE & SVM & FSC & 1 & 0.808 \\
141 & COV_TYPE & SVM & FSC & 2 & 0.803 \\
142 & COV_TYPE & SVM & FSC & 3 & 0.793 \\
143 & COV_TYPE & SVM & FSC & 4 & 0.804 \\
144 & COV_TYPE & SVM & FSC & 5 & 0.804 \\
145 & COV_TYPE & SVM & ROC & 1 & 0.876 \\
146 & COV_TYPE & SVM & ROC & 2 & 0.874 \\
147 & COV_TYPE & SVM & ROC & 3 & 0.869 \\
148 & COV_TYPE & SVM & ROC & 4 & 0.876 \\
149 & COV_TYPE & SVM & ROC & 5 & 0.873 \\

```
150 &  COV\_TYPE &    LOGREG &    ACC &     1 &  0.750 \\
151 &  COV\_TYPE &    LOGREG &    ACC &     2 &  0.754 \\
152 &  COV\_TYPE &    LOGREG &    ACC &     3 &  0.754 \\
153 &  COV\_TYPE &    LOGREG &    ACC &     4 &  0.759 \\
154 &  COV\_TYPE &    LOGREG &    ACC &     5 &  0.754 \\
155 &  COV\_TYPE &    LOGREG &    FSC &     1 &  0.751 \\
156 &  COV\_TYPE &    LOGREG &    FSC &     2 &  0.753 \\
157 &  COV\_TYPE &    LOGREG &    FSC &     3 &  0.749 \\
158 &  COV\_TYPE &    LOGREG &    FSC &     4 &  0.759 \\
159 &  COV\_TYPE &    LOGREG &    FSC &     5 &  0.755 \\
160 &  COV\_TYPE &    LOGREG &    ROC &     1 &  0.822 \\
161 &  COV\_TYPE &    LOGREG &    ROC &     2 &  0.824 \\
162 &  COV\_TYPE &    LOGREG &    ROC &     3 &  0.825 \\
163 &  COV\_TYPE &    LOGREG &    ROC &     4 &  0.824 \\
164 &  COV\_TYPE &    LOGREG &    ROC &     5 &  0.824 \\
165 &  COV\_TYPE &       RF &    ACC &     1 &  0.821 \\
166 &  COV\_TYPE &       RF &    ACC &     2 &  0.821 \\
167 &  COV\_TYPE &       RF &    ACC &     3 &  0.816 \\
168 &  COV\_TYPE &       RF &    ACC &     4 &  0.824 \\
169 &  COV\_TYPE &       RF &    ACC &     5 &  0.816 \\
170 &  COV\_TYPE &       RF &    FSC &     1 &  0.821 \\
171 &  COV\_TYPE &       RF &    FSC &     2 &  0.823 \\
172 &  COV\_TYPE &       RF &    FSC &     3 &  0.812 \\
173 &  COV\_TYPE &       RF &    FSC &     4 &  0.824 \\
174 &  COV\_TYPE &       RF &    FSC &     5 &  0.817 \\
175 &  COV\_TYPE &       RF &    ROC &     1 &  0.901 \\
176 &  COV\_TYPE &       RF &    ROC &     2 &  0.902 \\
177 &  COV\_TYPE &       RF &    ROC &     3 &  0.899 \\
178 &  COV\_TYPE &       RF &    ROC &     4 &  0.904 \\
179 &  COV\_TYPE &       RF &    ROC &     5 &  0.896 \\
\end{longtable}
```

SVC parameter table generator

In [178…

```python
# make SVC parameter table
table_svc_parameters = [['PROBLEM', '#ATTR', r'(num_features)^{-1}', r'(num_features * var(features))^{-1}']]
for X_data, y_data, data_name in data_tups:
    num_features = X_data.shape[1]
                #  PROBLEM        #ATTR  (num_features)^{-1}   (num_features * var(features))^{-1}
    #print(f"{data_name}\t{num_features}\t{int(num_features ** (-1))}\t{int((num_features * X_data.var()) ** (-1))}")
    table_svc_parameters.append([data_name, num_features, f"{num_features ** (-1):.3f}", f"{(num_features * X_data.var()) ** (-1):.3f

#latexify the table
table_svc_parameters_latex = tabulate(table_svc_parameters, tablefmt="latex")

# add in the other horizontal line
table_svc_parameters_latex_list = table_svc_parameters_latex.split("\n")
table_svc_parameters_latex_list.insert(3, "\hline")
table_svc_parameters_latex = "\n".join(table_svc_parameters_latex_list)

display(Latex(table_svc_parameters_latex))
```

```
\begin{tabular}{llll}
\hline
 PROBLEM  & \#ATTR & (num\_features)\^{}\{-1\} & (num\_features * var(features))\^{}\{-1\} \\
\hline
 SURGICAL & 45    & 0.022              & 0.076                          \\
 CHURN    & 13    & 0.077              & 0.155                          \\
 ADULT    & 108   & 0.009              & 0.075                          \\
 COV\_TYPE & 54    & 0.019              & 0.084                          \\
\hline
\end{tabular}
```

FAILED ATTEMPT TO AUTO-GENERATE TABLE 2 AND TABLE 3

After 8 hours of coding it, I realized I had very erroneously assumed autogenerating the tables would mean less work for me... I was wrong, quite wrong. I leave this code graveyard here so you can gaze upon my attempt.

(I realized way later that it would have been much easier to autogenerate tables 2 and 3 using pandas)

In [ ]:

```python
# generate the tables
tables = []
```

In [ ]:

```python
# GENERATE TABLE 2

def getClfMetricScores(this_clf_name, metric):
    '''
    get a list of all the scores on this metric for this classifier on all datasets
    '''

    scores = [] # len(scores) should = # of datasets x # of trials

    for data_name in best_scores:
        for trial in best_scores[data_name][clf_name]:
            scores.append(trial[metric])

    if len(scores) != len(data_tups) * NUM_TRIALS:
        print('ERROR: len(scores) should = # of datasets x # of trials')

    return scores
```

```
# enter scores into table 2

#create header
table2 = [["MODEL"]]
table2[0].extend([metric.upper() for metric in metrics])
table2[0].append('MEAN')

# init metric_trial_scores
metric_trial_scores = {}
for metric in metrics:
    metric_trial_scores[metric] = {}


# add scores for all metrics and classifiers to table2
for clf_name in best_scores['surgical']:
                # MODEL
    clf_row = [clf_name.upper()]
    for metric in metrics:
        metric_trial_scores[metric][clf_name] = getClfMetricScores(clf_name, metric)

        # average score for this metric over all trials and datasets
        clf_row.append(f"{np.mean(metric_trial_scores[metric][clf_name]):.3f}")

    # mean score for this clf across metrics, trials, and datasets
    print(f"clf_row: {clf_row}")
    clf_row.append(f"{np.mean([float(score) for score in clf_row[1:]]):.3f}")
    table2.append(clf_row)


#latexify the table
table2_latex = tabulate(table2, tablefmt="latex")

# add in the other horizontal line
table2_latex_list = table2_latex.split("\n")
table2_latex_list.insert(3, "\hline")
table2_latex = "\n".join(table2_latex_list)

tables.append(table2_latex)


# add asterisks for non-significant differences and save boldface rows and cols
# boldface_table[2][3] = 1 => for clf 2 and and metric 3, score should be bold (0 = not bold)
boldface_table = np.zeros((len(table2), len(table2[0])))

#validation
if (len(table2[0]) != len(metrics) + 2):
    print("ERROR: len(tables[0]) should equal len(metrics) + 2")

for col, metric in enumerate(metrics):
    col += 1 # don't want to consider the column with classifier names
    num_clfs = best_scores['surgical']
    # loop over all clfs and calculate which ones are not statistically different from top scorer

    # find top scorer
    top_clf_score = 0; top_clf_scores = []; top_clf_row = 0
    for row,clf_name in enumerate(best_scores['surgical']):
        row += 1 # dont want to consider the title row
        clf_metric_score_mean = np.mean(metric_trial_scores[metric][clf_name])
        if clf_metric_score_mean > top_clf_score:
            top_clf_score = clf_metric_score_mean
            top_clf_scores = metric_trial_scores[metric][clf_name]
            top_clf_row = row


    # save top score index to boldface later index
    boldface_table[top_clf_row][col] = 1

    # add asterisks to scores with non-significant differences
    for row, clf_name in enumerate(best_scores['surgical']):
        row += 1 # dont want to consider the title row
        # all the scores (for all trials) on this metrics for this row's clf
        curr_clf_scores = metric_trial_scores[metric][clf_name]

        # compare for statistical difference
        t, p = ttest_rel(curr_clf_scores, top_clf_scores)
        #print(f'top_clf_scores: {top_clf_scores}\ncurr_clf_scores: {curr_clf_scores}\np: {p}')
        if p >= 0.05:
            # add asterisk
            table2[row][col] += "*"


#boldface the top scores
# validation
if (len(table2) != len(boldface_table) or len(table2[0]) != len(boldface_table[0])):
    print("ERROR 3")

print(table2)
print(boldface_table)
table2_latex_list = table2_latex.split("\n")
COL_OFFSET = 2; ROW_OFFSET = 2
for row in range(len(table2)):
    for col in range(len(table2[0])):
        if boldface_table[row][col] == 1:
            print(f"row: {row} | col: {col}")
            row_list = table2_latex_list[row + ROW_OFFSET].split(" ")
            print(f"row_list: {row_list}")
            print(f"before: {row_list[col + MODEL_AMP_OFFSET]}")
            row_list[col + MODEL_AMP_OFFSET] = "\textbf{" + row_list[col + MODEL_AMP_OFFSET] + "}"
            print(f"after: {row_list[col + MODEL_AMP_OFFSET]}")
            table2_latex_list[row + ROW_OFFSET] = " ".join(row_list)
```

```
# put the table back together again
table2_latex = "\n".join(table2_latex_list)
```

In [ ]:

```
# GENERATE TABLE 3
def getClfDataScores(this_clf_name, this_data_name):

    '''
    get a list of all the scores on this metric for this classifier on all datasets
    get the best model for this classifier
    '''

    scores = [] # len(scores) should = # of datasets x # of trials x # of metrics

    for trial in best_scores[data_name][clf_name]:
        #print(f"trial.values(): {np.array(list(trial.values()))}")
        trial_metric_scores = list(trial.values())
        scores.append(np.mean(trial_metric_scores))

    '''
    for data_name in best_scores:
        if data_name == this_data_name:
            for clf_name in best_scores[data_name]:
                if clf_name == this_clf_name:
                    for i,trial in enumerate(best_scores[data_name][clf_name]):
                        print(f'trial {i}: {trial}')
                    scores = [np.mean(trial.values()) for trial in best_scores[data_name][clf_name]]
                    if len(best_scores[data_name][clf_name][0]) != NUM_TRIALS:
                        print('ERROR 4')
    '''
    if len(scores) != NUM_TRIALS:
        print('ERROR1')

    return scores


# init data_clf_scores: each data_name/clf_name key maps to the list of length
# NUM_TRIALS containing the average metric for all trials
data_clf_scores = {}
for data_name in best_scores:
    data_clf_scores[data_name] = {}

table3 = []
# each element = mean across metrics for that clf and data combo
for data_name in best_scores:
    #print('STARTING LOOP')
    # set the value of all rows (clfs) and find top scoring row values
    data_col = []; top_clf_score = 0; top_clf_scores = []; top_clf_row = 0
    for row, clf_name in enumerate(best_scores[data_name]):
        data_clf_scores[data_name][clf_name] = getClfDataScores(clf_name, data_name)
        clf_data_score_mean = np.mean(data_clf_scores[data_name][clf_name])
        if clf_data_score_mean > top_clf_score:
            top_clf_score = clf_data_score_mean
            top_clf_scores = data_clf_scores[data_name][clf_name]
            top_clf_row = row

    # append ALL the scores for this clf/data combo (will mean over trials later)
    data_col.append(data_clf_scores[data_name][clf_name])

print(f"{data_name} data_col: {data_col}")

# find rows that should be asterisked and save the row indices
to_be_asterisked = []
for row, curr_clf_scores in enumerate(data_col):
    #compare to top score distribution
    t, p = ttest_rel(curr_clf_scores, top_clf_scores)
    print(f'top_clf_scores: {top_clf_scores}\ncurr_clf_scores: {curr_clf_scores}\np: {p}')
    if p > 0.05:
        to_be_asterisked.append(row)

# convert rows to strings and add asterisks
temp = []
for row, curr_clf_scores in data_col:
    curr_clf_avg_score = np.mean(curr_clf_scores) # average over all trials (for this dataset)
    temp.append(f"{curr_clf_avg_score:.3f}")
    if row in to_be_asterisked:
        temp[-1] += "*"
    data_col = temp

    # add the column to the table
    table3.append(data_col)

'''
what table3 looks like right now:
[[avg_c1d1, avg_c2d1], ...]
         ...
 [avg_c1dn], [avg_c2dn], ...]]

'''
print(f"what table3 looks like right now: \n{table3}")

# transpose table3
table3 = np.matrix(table3).T.tolist()

# add MEAN value (first convert from string to float, calc mean, then add mean to row as string)
for i,row in enumerate(table3):
    row_as_num = [float(score[:-1]) for score in row]
    table3[i] = row + [f"{np.mean(row_as_num):.3f}"]

print(f"table 3 after adding all numerical values: {table3}")

# add in algo names and dataset names (and MODEL and MEAN)
```

```
for i,name in enumerate(classifier_names):
    table3[i].insert(0, name.upper())


#add header
header = ["MODEL"]
header.extend([data_name.upper() for data_name in dataset_names])
header.append('MEAN')
table3.insert(0, header)

print(f"table 3 after adding all heading values: {table3}")


#latexify the table and add in the other horizontal line
table3_latex = tabulate(table3, tablefmt="latex")
table3_latex_list = table3_latex.split("\n")
table3_latex_list.insert(3, "\hline")
table3_latex = "\n".join(table3_latex_list)

tables.append(table3_latex)


# add scores for all metrics and classifiers to table2
for clf_name in best_scores['surgical']:
                # MODEL
    clf_row = [clf_name.upper()]
    for metric in metrics:
        metric_trial_scores[metric][clf_name] = getClfMetricScores(clf_name, metric)

        # average score for this metric over all trials and datasets
        clf_row.append(f"{np.mean(metric_trial_scores[metric][clf_name]):.3f}")

    # mean score for this clf across metrics, trials, and datasets
    print(f"clf_row: {clf_row}")
    clf_row.append(f"{np.mean([float(score) for score in clf_row[1:]]):.3f}")
    table2.append(clf_row)
```

In [ ]:

```
# print latex for all tables
for i, latex_table in enumerate(tables):
    print(f"LaTex for table {i}:")
    display(Latex(rf"{latex_table}"))
```