Supervised Binary Classification Model Comparison

Jack Steussie

University of California San Diego Cognitive Science Department
jsteussi@ucsd.edu

Abstract

Machine learning has seen a lot of popularity and growth in the industry and university environments within the last decade. In this paper, I present a replication of Caruana and Niculescu-Mizil's 2006 paper and I empirically evaluate the performances of five supervised machine learning algorithms: KNN, neural nets, boosted decision trees, random forest, and logistic regression. The results of my analysis of these algorithms resemble Caruana's results, but differ slightly in which algorithm ended up giving the best results overall. I will be using three metrics to evaluate these algorithms so that we can see how they perform with respect to these different metrics, and the performances will be measured across four data sets as well. The main purpose of this study is to gain a better general understanding of the algorithms evaluated while taking into account their individual advantages and disadvantages.

1 Introduction

STATLOG (King et al., 1995) was the first well-known empirical comparison of supervised learning models, and Caruana and Niculescu-Mizil (2006), henceforth referred to as CNM06, followed up STATLOG in order to provide a more thorough and up-to-date analysis of the various algorithms available. CNM06 furthered the research done by STATLOG by evaluating more algorithms and evaluating said algorithms with respect to a larger variety of metrics. Caruana experiments using SVMs, neural nets, logistic regression, naive bayes, kNN, random forests, and decision trees, on 11 different data sets, and argues that on average boosted trees and random forests perform best out of all of the algorithms while logistic regression, decision trees, stumps, and naive bayes perform worst. However, they also state that certain algorithms still have poor performance on particular data sets and metrics. CNM06 is, in the end, a great example of how algorithms within the machine learning sphere should be compared and evaluated. Having said that, this paper will follow in the footsteps of CNM06 and attempt to replicate their results with five algorithms (random forests, KNNs, boosted decision trees, logistic regression, and artificial neural networks), four datasets that were not used in CNM06 (HTRU2, HIGGS, SUSY, and BIT), and three metrics (accuracy, F-score, and AUC-ROC). It is important to do the same type of analysis as CNM06, as this paper is doing, in slightly different ways while keeping the main goal of comparison because it brings insight into how we might be able to generalize the results found in CNM06.

2 Methodology

2.1 Learning Algorithms

For each learning algorithm being evaluated, I perform a grid search that covers as much of the parameter combinations for each algorithm within the limits of computation as possible. I also use the sklearn implementation of each algorithm exclusively. It is also worth noting that I use validation sets for the hyper parameter search. All algorithms within each dataset use features that have been scaled to 0 mean 1 std.

KNN: I use 26 values of K at increments of four ranging from K = 1 to K = 105. I use KNN with Euclidean distance and Euclidean distance weighted by gain ratio.

ANNs: I train neural nets using stochastic gradient descent back propagation and vary the number of hidden layers {1, 2, 4, 8, 32, 128}. I use the rectified linear activation function because of how widely used it is in many situations. I also vary the maximum amount of training epochs allowed {2, 4, 8, 16, 32, 64, 128, 256, 512}. I also use constant and inverse scaling of the learning rate, varying the initial starting value for the learning rate by factors of ten from 10^{-3} to 10^{0}.

Logistic Regression: I train both unregularized and regularized models, varying the regularization parameter by factors of ten from 10^{-8} to 10^{4}. The regularization types used are L1 and L2 regularizations.

Boosted Decision Trees: We use the sklearn AdaBoost implementation of boosted trees. I chose to use the SAMME.R boosting algorithm as it seemed like the best fit for our datasets and it is known to converge faster and give lower test error than the SAMME solver. I also chose to vary the estimators and chose the values {2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048}. The final parameter I adjusted was the learning rate, which took the values {1e-3, 1e-2, 1e-1, 1e0, 1e1, 2e1, 5e1}.

Random Forest: The forests have 1024 trees, and the size of the feature set considered at each split is 1, 2, 4, 6, 8, 12, 16, or 20.

2.2 Performance Metrics

I use three metrics to evaluate model performance: accuracy (ACC), area under the ROC curve (ROC), and F-score (FSC). FSC and ACC are referred to as threshold metrics, while ROC is what is referred to as a rank metric. Threshold metrics are evaluated whether or not they are above a certain threshold of correct predictions. Ranking metrics show how the model performs in its ordering of positive cases before negative cases.

Table 1. Description of Problems (Pre-Data Sampling)

PROBLEM	# ATTR	TRAIN SIZE	VAL SIZE	TEST SIZE	%POZ
HTRU2	9	n/a	n/a	n/a	9.16%
HIGGS	28	n/a	n/a	n/a	52.99%
SUSY	18	n/a	n/a	n/a	45.76%
BIT	10	n/a	n/a	n/a	1.42%
Description of Problems (Post-Data Sampling)					
HTRU2	9	5000	5000	2000	50%
HIGGS	28	5000	5000	2000	50%
SUSY	18	5000	5000	2000	50%
BIT	10	5000	5000	2000	50%

2.3 Data Sets

The algorithms are trained and tested on four data sets available from the UCI Machine Learning repository: HTRU2, HIGGS, SUSY, and Bitcoin Heist Ransomware Address Data Set (BIT) (See

Table 1). When cleaning the data in each set, I down sampled the majority class randomly without replacement and up sampled the minority class without replacement to match the amount of samples in the majority class (after the down sampling). The HTRU2 is the only data set where I up sampled the minority class with replacement because of how few samples there actually were in the minority class. After down sampling and up sampling each data set, the data sets we used each ended up containing a total of 12000 samples with 6000 samples in each class. When looking at the original BIT data set, it is a *very* imbalanced dataset with only around 1.42% of the data set representing the positive class. Down sampling and up sampling the data changed this in the end, but it is definitely still worth noting. When we exclude the target column, this data set contains a total of nine attributes that are to be used as features. I removed the Bitcoin address, year, and day attributes as there is nothing that can be predicted from these. We are then left with a total of six attributes to work with. In the original HTRU2 data set, we see that there is eight attributes if we exclude the target label, and it is a rather unbalanced dataset with 9.16% of the data representing the positive target class. No data cleaning except the up and down sampling was done on this since all of the values are numerical in nature. Similarly, the SUSY data set had no cleaning done, but it has a total of 17 attributes excluding the target label. SUSY is a pretty balanced data set. The HIGGS data set contains a total of 27 attributes if we exclude the target label and had no cleaning done to it. It is also a very balanced dataset.

3 Experiments

3.1 Performances by Metric

Table 2. Scores by Metric Averaged Over 4 Problems

MODEL	ACC	ROC	FSC	MEAN
RF	**0.795**	**0.852**	**0.790**	**0.812**
KNN	0.744*	0.794*	0.750	0.762
BST-DT	0.785	0.843*	0.780*	0.802
LR	0.725*	0.766*	0.745*	0.745
ANN	0.750*	0.803*	0.756*	0.770

For each test problem, I randomly select 5000 samples for training, 5000 for validation and 2000 for testing each trial (of which there are five). I use k-fold cross validation on the 5000 validation samples in order to find the best hyper parameters. We use a validation set in order to make sure that no information from the training data is getting leaked to the classifier, which could potentially provide artificially better performances within each metric.

Table 2 shows the score for each algorithm averaged over the set of three metrics we are measuring. Similar to what was done in CNM06, we find the best set of hyper parameters for each algorithm with the validation set, then report that model's score on the test set. Each entry in the table averages the scores of its corresponding algorithm across the five trials and four test problems. Higher scores indicate better performance in a metric. The last column in Table 2 is the mean of all the metrics with respect to a given algorithm. A **boldfaced** entry indicates that the algorithm it corresponds to is the best score out of all the other algorithms for a given metric. A * next to any entry in the table indicates that the algorithm that entry corresponds with is statistically indistinguishable from the score of the best algorithm with respect to the metric that entry corresponds to. In order to determine whether an algorithm was statistically distinguishable from the best or not, we took related paired t-tests at a value of $p = 0.05$ (refer to appendix to see the p-values of each test done).

Looking at the mean scores of each algorithm over the four problem sets with respect to each metric (see Table 2), random forests has a better score than the rest of the algorithms across the board, which corresponds to the findings of CNM06, but statistically it is only better than a couple of other algorithms with respect to a couple of the metrics (which does not correspond with CNM06). Random forests turns out to be statistically better than boosted decision trees with respect to accuracy and it is also statistically better than KNN with respect to F-score. The latter is correspondent with CNM06 while the former is not.

3.2 Performances by Problem

If we refer to the mean scores of each algorithm across all metrics with respect to each problem set (see Table 3), we can see that random forest numerically out performs all other algorithms across each problem except for SUSY, where ANN performs best. For all problems, the best classifier was statistically better than all of the other algorithms. In fact, the p-values from the t-tests conducted were all very low (all of them are below 10^{-3}. Numerically, logistic regression performed the worst over all datasets except for SUSY, where KNN performed the worst.

Table 3. Scores by Problem Averaged Over 3 Metrics

MODEL	HTRU2	HIGGS	SUSY	BIT
RF	**0.984**	**0.730**	0.803	**0.731**
KNN	0.977	0.641	0.765	0.666
BST-DT	0.975	0.716	0.805	0.713
LR	0.956	0.607	0.799	0.618
ANN	0.961	0.669	**0.810**	0.639

4 Discussion

Looking at my experimental results overall, they generally point to the same conclusion that was reached in CNM06: random forests generally performs better than the other algorithms presented. This comes with the caveat, however, that in my study, even though random forests does perform *numerically* better than most algorithms with respect to each problem over the three metrics, it does not perform *statistically* better than the rest (more often than not). When we look at the results with respect to each metric over the four problems, we do see that it performs statistically better than the rest more often than not. In the case of when the ANN performed better than random forests in the SUSY problem, this could be indicative of why neural networks have seen a growth in popularity over the past years.When we compare the performance of KNN and random forests in the test set to their respective training performances (not shown but they had near perfect training performance) we can see that this corresponds with random forests' and KNNs' tendency to over fit. Boosted decision trees performed worse on average than random forests, which coincides with the findings in CNM06. Almost all of the results in this study match those of CNM06, giving the study more merit than it did before I conducted my own version.

5 Bonus

In addition to the three required algorithm, this study explores an additional two algorithms.

References

(1) Caruana , Rich, and Alexandru Niculescu-Mizil. "An Empirical Comparison of Supervised Learning Algorithms." In Proceedings of the 23rd International Conference on Machine Learning, 2006.

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

(3) Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

A P-Values

A.1 Table 2 P-Values

P-Values Table 2			
MODEL	ACC(RF)	ROC(RF)	FSC(RF)
RF	1.000	1.000	1.000
KNN	0.085	0.104	0.034
BST-DT	0.046	0.254	0.144
LR	0.102	0.160	0.086
ANN	0.117	0.189	0.148

A.2 Table 3 P-Values

P-Values Table 3				
MODEL	HTRU2(RF)	HIGGS(RF)	SUSY(ANN)	BIT(RF)
RF	1.000	1.000	4.405e-4	1.000
KNN	3.264e-7	3.346e-8	3.211e-9	1.963e-9
BST-DT	7.970e-6	1.440e-3	1.191e-4	2.749e-3
LR	1.512e-8	2.822e-7	8.739e-5	5.412e-6
ANN	2.529e-7	6.249e-6	1.000	5.224e-7

HIGGS Dataset

March 19, 2021

1 HIGGS Dataset

See https://archive.ics.uci.edu/ml/datasets/HIGGS for dataset information and feature descriptions.

[1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import csv
import numpy as np
import pandas as pd
import pandas_profiling
import matplotlib.pyplot as plt
from scipy import stats
import pickle
import operator
import glob
from scipy.io.arff import loadarff
from scipy.stats import ttest_rel

import seaborn as sns; sns.set_style('white')

from sklearn.utils import resample
from sklearn.metrics import accuracy_score, plot_confusion_matrix, f1_score,␣
↪plot_roc_curve, roc_auc_score, make_scorer
from sklearn.model_selection import KFold, GridSearchCV, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.pipeline import Pipeline
from sklearn.neural_network import MLPClassifier

from sklearn.model_selection import cross_val_score
```

[19]:
```
df = pd.read_csv('HIGGS.csv', header=None)
df.columns = ['class', 'lepton_pT', 'lepton_eta', 'lepton_phi',␣
 ↪'missing_energy_magnitude',
              'missing_energy_phi', 'jet_1_pt', 'jet_1_eta', 'jet_1_phi',␣
 ↪'jet_1_b-tag',
              'jet_2_pt', 'jet_2_eta', 'jet_2_phi', 'jet_2_b-tag', 'jet_3_pt',␣
 ↪'jet_3_eta',
              'jet_3_phi', 'jet_3_b-tag', 'jet_4_pt', 'jet_4_eta', 'jet_4_phi',␣
 ↪'jet_4_b-tag',
              'm_jj', 'm_jjj', 'm_lv', 'm_jlv', 'm_bb', 'm_wbb', 'm_wwbb']
```

[21]:
```
df
```

[21]:
```
          class  lepton_pT  lepton_eta  lepton_phi  missing_energy_magnitude  \
0           1.0   0.869293   -0.635082    0.225690                  0.327470
1           1.0   0.907542    0.329147    0.359412                  1.497970
2           1.0   0.798835    1.470639   -1.635975                  0.453773
3           0.0   1.344385   -0.876626    0.935913                  1.992050
4           1.0   1.105009    0.321356    1.522401                  0.882808
…           …         …           …          …                         …
10999995    1.0   1.159912    1.013847    0.108615                  1.495524
10999996    1.0   0.618388   -1.012982    1.110139                  0.941023
10999997    1.0   0.700559    0.774251    1.520182                  0.847112
10999998    0.0   1.178030    0.117796   -1.276980                  1.864457
10999999    0.0   0.464477   -0.337047    0.229019                  0.954596

          missing_energy_phi  jet_1_pt  jet_1_eta  jet_1_phi  jet_1_b-tag  \
0                  -0.689993  0.754202  -0.248573  -1.092064     0.000000
1                  -0.313010  1.095531  -0.557525  -1.588230     2.173076
2                   0.425629  1.104875   1.282322   1.381664     0.000000
3                   0.882454  1.786066  -1.646778  -0.942383     0.000000
4                  -1.205349  0.681466  -1.070464  -0.921871     0.000000
…                        …         …         …          …          …
10999995           -0.537545  2.342396  -0.839740   1.320683     0.000000
10999996           -0.379199  1.004656   0.348535  -1.678593     2.173076
10999997            0.211230  1.095531   0.052457   0.024553     2.173076
10999998           -0.584370  0.998519  -1.264549   1.276333     0.000000
10999999           -0.868466  0.430004  -0.271348  -1.252278     2.173076

          …  jet_4_eta  jet_4_phi  jet_4_b-tag      m_jj     m_jjj  \
0         …  -0.010455  -0.045767     3.101961  1.353760  0.979563
1         …  -1.138930  -0.000819     0.000000  0.302220  0.833048
2         …   1.128848   0.900461     0.000000  0.909753  1.108330
3         …  -0.678379  -1.360356     0.000000  0.946652  1.028704
4         …  -0.373566   0.113041     0.000000  0.755856  1.361057
…        …        …          …           …         …         …
10999995  …  -0.097068   1.190680     3.101961  0.822136  0.766772
```

```
10999996  …  -0.216995   1.049177     3.101961  0.826829  0.989809
10999997  …   1.585235   1.713962     0.000000  0.337374  0.845208
10999998  …   1.399515  -1.313189     0.000000  0.838842  0.882890
10999999  …  -1.652782  -0.586254     0.000000  0.752535  0.740727

              m_lv     m_jlv      m_bb     m_wbb    m_wwbb
0         0.978076  0.920005  0.721657  0.988751  0.876678
1         0.985700  0.978098  0.779732  0.992356  0.798343
2         0.985692  0.951331  0.803252  0.865924  0.780118
3         0.998656  0.728281  0.869200  1.026736  0.957904
4         0.986610  0.838085  1.133295  0.872245  0.808487
…              …         …         …         …         …
10999995  1.002191  1.061233  0.837004  0.860472  0.772484
10999996  1.029104  1.199679  0.891481  0.938490  0.865269
10999997  0.987610  0.883422  1.888438  1.153766  0.931279
10999998  1.201380  0.939216  0.339705  0.759070  0.719119
10999999  0.986917  0.663952  0.576084  0.541427  0.517420

[11000000 rows x 29 columns]
```

[4]:
```python
# Separate majority and minority classes
df_majority = df[df['class']==0]
df_minority = df[df['class']==1]

# Downsample majority and minority class
df_majority_downsampled = resample(df_majority,
                                   replace=False,    # sample without replacement
                                   n_samples=6000,     # to match minority class
                                   random_state=123) # reproducible results

df_minority_downsampled = resample(df_minority,
                                   replace=False,    # sample with replacement
                                   n_samples=6000,     # to match majority class
                                   random_state=123) # reproducible results

# Combine minority class with downsampled majority class
df_downsampled = pd.concat([df_majority_downsampled, df_minority_downsampled])

# Display new class counts
df_downsampled['class'].value_counts()
```

[4]:
```
1.0    6000
0.0    6000
Name: class, dtype: int64
```

[5]:
```python
df_downsampled
```

[5]:

	class	lepton_pT	lepton_eta	lepton_phi	missing_energy_magnitude	\
2984264	0.0	0.558361	0.125588	-0.383650	0.499787	
10764116	0.0	1.296253	0.516149	-0.117317	0.636398	
9530004	0.0	1.062368	-0.338021	1.577887	0.534542	
5455217	0.0	1.355182	0.181104	0.414898	0.758879	
6082563	0.0	0.659565	-1.691838	0.469274	0.860520	
…	…	…	…	…	…	
521000	1.0	1.165585	1.037223	-1.130496	1.259846	
1672790	1.0	0.706049	0.738214	1.374808	0.757621	
7951892	1.0	0.741004	-0.879548	0.831599	0.560017	
10444720	1.0	1.245926	-0.610733	-0.588952	0.719146	
5729388	1.0	0.383404	-1.080185	-0.242160	1.824291	

	missing_energy_phi	jet_1_pt	jet_1_eta	jet_1_phi	jet_1_b-tag	\
2984264	-1.464681	0.656732	-0.140638	0.583364	0.000000	
10764116	-0.455267	0.988075	0.144548	0.907119	2.173076	
9530004	-0.183894	1.081973	-1.038777	-0.608094	2.173076	
5455217	1.102309	0.513184	-0.193120	1.042387	2.173076	
6082563	-0.639793	0.850298	-1.143741	1.576804	2.173076	
…	…	…	…	…	…	
521000	-0.187387	0.397484	-0.461473	-0.793256	0.000000	
1672790	-0.172691	0.392995	0.581240	0.354407	2.173076	
7951892	-0.590536	1.122463	-0.517916	-1.707975	0.000000	
10444720	0.448291	1.028657	0.864445	1.571815	2.173076	
5729388	1.061444	2.387559	1.132798	-0.476151	0.000000	

	…	jet_4_eta	jet_4_phi	jet_4_b-tag	m_jj	m_jjj	\
2984264	…	0.719931	0.224578	3.101961	1.373681	0.927890	
10764116	…	1.291248	1.573014	0.000000	0.901899	0.875995	
9530004	…	0.164438	0.940969	0.000000	1.120079	0.906208	
5455217	…	0.516723	1.498656	3.101961	0.959572	0.989931	
6082563	…	-1.006511	-1.012426	0.000000	0.901754	1.060937	
…	…	…	…	…	…	…	
521000	…	0.586680	1.607974	3.101961	1.355938	1.000538	
1672790	…	-0.128715	-0.656173	0.000000	0.957407	1.225450	
7951892	…	-0.693370	0.101387	3.101961	0.807574	1.053625	
10444720	…	-0.700865	1.082472	3.101961	0.727640	0.912113	
5729388	…	-0.177019	-0.231108	0.000000	1.058214	0.788499	

	m_lv	m_jlv	m_bb	m_wbb	m_wwbb
2984264	0.994193	1.126345	1.159692	0.833733	0.796250
10764116	0.975326	0.882177	1.255846	0.909731	0.763828
9530004	1.043838	0.997955	1.427019	0.901307	0.789998
5455217	0.984073	0.778193	0.565135	0.838409	0.813519
6082563	0.985931	0.842516	0.874739	0.762080	0.752349
…	…	…	…	…	…
521000	1.261941	1.162553	0.474319	0.771688	1.002877

```
1672790   0.997393  0.571817  1.134243  0.831666  0.737647
7951892   0.986507  0.659439  0.759948  0.887652  0.748781
10444720  1.055416  0.650828  0.919102  0.956839  0.801805
5729388   1.069115  1.305123  0.820215  1.079734  1.016867

[12000 rows x 29 columns]
```

[6]:
```
df = df_downsampled.copy()
scaler = StandardScaler()
X, y = df.iloc[:,1:].to_numpy(), df.iloc[:,0].to_numpy()
```

1.1 Hyperparameter Search & Experimentation

[10]:
```
def experiment():
    pipeline1 = Pipeline((
    ('clf', RandomForestClassifier()),
    ))

    pipeline2 = Pipeline((
    ('clf', KNeighborsClassifier()),
    ))

    pipeline3 = Pipeline((
    ('clf', AdaBoostClassifier()),
    ))

    pipeline4 = Pipeline((
    ('clf', LogisticRegression()),
    ))

    pipeline5 = Pipeline((
    ('clf', MLPClassifier()),
    ))

    # Random Forest
    parameters1 = {
    'clf__n_estimators': [1024],
    'clf__max_features': [1, 2, 4, 6, 8, 12, 16, 20]
    }

    # KNN
    parameters2 = {
    'clf__n_neighbors':␣
↪[1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61,65,69,73,77,81,85,89,93,97,101,105],
    'clf__weights': ['uniform', 'distance']
    }
```

```
    # AdaBoost (Boosted Decision Tree)
    parameters3 = {
        'clf__algorithm': ['SAMME.R'],
        'clf__n_estimators': [2,4,8,16,32,64,128,256,512,1024,2048],
        'clf__learning_rate': [1e-3, 1e-2, 1e-1, 1e0, 1e1, 2e1, 5e1]
    }

    # Logistic
    parameters4 = {
    'clf__penalty':['l1', 'l2', None],
    'clf__C':[1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 0, 1e0, 1e1, 1e2,␣
↪1e3, 1e4],
    'clf__max_iter':[5000]
    }

    # Multi-layer Perceptron
    parameters5 = {
        'clf__hidden_layer_sizes':[(1,), (2,), (4,), (8,), (32,), (128,)],
        'clf__solver':['sgd'],
        'clf__activation':['relu'],
        'clf__learning_rate':['constant', 'invscaling'],
        'clf__learning_rate_init': [1e-3, 1e-2, 1e-1, 1e0],
        'clf__max_iter': [2, 4, 8, 16, 32, 64, 128, 256, 512]
    }

    pars = [parameters1, parameters2, parameters3, parameters4, parameters5]
    pips = [pipeline1, pipeline2, pipeline3, pipeline4, pipeline5]

    # List of dictionaries to hold the scores of the various metrics for each␣
↪type of classifier
    best_clf_list = []
    trial_storage = {}
    training_storage = {}

    print("starting Gridsearch")
    for i in range(len(pars)):
        trial_averages = []
        train_performance = []
        for t in range(5):
            # split and scale data
            X_train, X_test, y_train, y_test = train_test_split(X, y,␣
↪test_size=1/6, random_state=t)
            X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,␣
↪test_size=0.5, random_state=t)
            X_train = scaler.fit_transform(X_train)
            X_val = scaler.transform(X_val)
            X_test = scaler.transform(X_test)
```

```
        clf = GridSearchCV(pips[i], pars[i], refit=False, n_jobs=8, cv=5,␣
↪verbose=3, scoring=('accuracy', 'roc_auc', 'f1'))
        clf = clf.fit(X_val, y_val)

        print("finished Gridsearch trial " + str(t + 1) + " classifier " +␣
↪str(i + 1))
        print("")
        print("")

        # find the best params for each metric in a given trial
        best_index_acc = np.argmin(clf.cv_results_['rank_test_accuracy'])
        best_params_acc = clf.cv_results_['params'][best_index_acc]
        best_index_roc = np.argmin(clf.cv_results_['rank_test_roc_auc'])
        best_params_roc = clf.cv_results_['params'][best_index_roc]
        best_index_f1 = np.argmin(clf.cv_results_['rank_test_f1'])
        best_params_f1 = clf.cv_results_['params'][best_index_f1]

        # train and test models for given metric with their corresponding␣
↪best parameter settings
        pipe = pips[i]
        clf_acc = pipe.set_params(**best_params_acc)
        clf_acc = clf_acc.fit(X_train, y_train)
        clf_roc = pipe.set_params(**best_params_roc)
        clf_roc = clf_roc.fit(X_train, y_train)
        clf_f1 = pipe.set_params(**best_params_f1)
        clf_f1 = clf_f1.fit(X_train, y_train)

        # get training set performance
        train_acc = accuracy_score(y_train, clf_acc.predict(X_train))
        train_roc = roc_auc_score(y_train, clf_roc.predict_proba(X_train)[:
↪, 1])
        train_f1 = f1_score(y_train, clf_f1.predict(X_train))

        train_performance.append({
            'Model #': i + 1,
            'average': (train_f1 + train_acc + train_roc)/3,
            'accuracy': train_acc,
            'roc_auc_score': train_roc,
            'f1 score': train_f1
        })

        # get test set performances
        trial_acc = clf_acc.score(X_test, y_test)
        trial_roc = roc_auc_score(y_test, clf_roc.predict_proba(X_test)[:,␣
↪1])
        trial_f1 = f1_score(y_test, clf_f1.predict(X_test))
```

```
            # store scores and their averages in list containing averages for␣
↪each trial
            trial_averages.append({
                'Model #': i + 1, # model number corresponds to the numbers␣
↪used in pipeline above (i.e. 1 = Random Forest)
                'average':(trial_acc + trial_roc + trial_f1) / 3,
                'accuracy': trial_acc,
                'roc_auc_score': trial_roc,
                'f1_score': trial_f1
            })

            train_performance.append({
                'Model #': i + 1, # model number corresponds to the numbers␣
↪used in pipeline above (i.e. 1 = Random Forest)
                'average':(train_acc + train_roc + train_f1) / 3,
                'accuracy': train_acc,
                'roc_auc_score': train_roc,
                'f1_score': train_f1
            })

        # find the trial with the best average metric scores and append those␣
↪scores as a dict to best clf list
        max_average = 0
        for trial in trial_averages:
            if trial['average'] > max_average:
                max_average = trial['average']
                best_trial = trial

        best_clf_list.append(best_trial)
        training_storage[str(i + 1)]=train_performance
        trial_storage[str(i + 1)]=trial_averages

    return best_clf_list, trial_storage, training_storage
```

[11]:
```
%%capture --no-stdout --no-display
best_clf_list, trial_storage, training_perf = experiment()
```

```
starting Gridsearch
Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 1 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 2 classifier 1
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 3 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 4 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 5 classifier 1


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 1 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 2 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 3 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 4 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 5 classifier 2


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 1 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 2 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 3 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 4 classifier 3
```

```
Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 5 classifier 3


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 1 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 2 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 3 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 4 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 5 classifier 4


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 1 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 2 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 3 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 4 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 5 classifier 5
```

1.2 Calculating and Organizing Results

[22]:
```python
print('Best Models On Average For Test Set:')
for element in best_clf_list:
    print(element)

print()

print('Train Set Data')
for i in range(len(training_perf)):
    print(training_perf[str(i + 1)])

print()

alg_avg = {}
alg_acc = {}
alg_roc = {}
alg_f1 = {}

for i in range(len(trial_storage)):
    alg_avg[str(i + 1)]=[]
    alg_acc[str(i + 1)]=[]
    alg_roc[str(i + 1)]=[]
    alg_f1[str(i + 1)]=[]
    for entry in trial_storage[str(i + 1)]:
        alg_avg[str(i + 1)].append(entry['average'])
        alg_acc[str(i + 1)].append(entry['accuracy'])
        alg_roc[str(i + 1)].append(entry['roc_auc_score'])
        alg_f1[str(i + 1)].append(entry['f1_score'])

print('set of averages of algorithms over 5 trials:')
print(alg_avg)
print()
print('set of acc values of algorithms over 5 trials')
print(alg_acc)
print()
print('set of roc values of algorithms over 5 trials')
print(alg_roc)
print()
print('set of f1 values of algorithms over 5 trials')
print(alg_f1)
```

```
Best Models On Average For Test Set:
{'Model #': 1, 'average': 0.7561565748411576, 'accuracy': 0.735,
'roc_auc_score': 0.8038778877887788, 'f1_score': 0.7295918367346939}
{'Model #': 2, 'average': 0.6618301953002318, 'accuracy': 0.627,
'roc_auc_score': 0.6856835683568356, 'f1_score': 0.6728070175438597}
{'Model #': 3, 'average': 0.7288081064743643, 'accuracy': 0.705,
```

'roc_auc_score': 0.7714931493149316, 'f1_score': 0.7099311701081613}
{'Model #': 4, 'average': 0.6415731580835212, 'accuracy': 0.62, 'roc_auc_score': 0.668008193179818, 'f1_score': 0.6367112810707457}
{'Model #': 5, 'average': 0.6889371606935969, 'accuracy': 0.671, 'roc_auc_score': 0.7281347144040231, 'f1_score': 0.6676767676767676}

Train Set Data

[{'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}]
[{'Model #': 2, 'average': 0.6709448781975818, 'accuracy': 0.6284, 'roc_auc_score': 0.6967035421557706, 'f1 score': 0.6877310924369747}, {'Model #': 2, 'average': 0.6709448781975818, 'accuracy': 0.6284, 'roc_auc_score': 0.6967035421557706, 'f1_score': 0.6877310924369747}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}]
[{'Model #': 3, 'average': 0.74206384576397, 'accuracy': 0.7158, 'roc_auc_score': 0.7922800514756035, 'f1 score': 0.7181114858163063}, {'Model #': 3, 'average': 0.7420638457639699, 'accuracy': 0.7158, 'roc_auc_score': 0.7922800514756035, 'f1_score': 0.7181114858163063}, {'Model #': 3, 'average': 0.7295420633943038, 'accuracy': 0.7058, 'roc_auc_score': 0.7799378847900615, 'f1 score': 0.7028883053928499}, {'Model #': 3, 'average': 0.7295420633943038, 'accuracy': 0.7058, 'roc_auc_score': 0.7799378847900615, 'f1_score': 0.7028883053928499}, {'Model #': 3, 'average': 0.7206933964368417, 'accuracy': 0.6992, 'roc_auc_score': 0.7648882214390389, 'f1 score': 0.697991967871486}, {'Model #': 3, 'average': 0.7206933964368417, 'accuracy': 0.6992, 'roc_auc_score': 0.7648882214390389, 'f1_score': 0.697991967871486}, {'Model #': 3, 'average': 0.7296980079045815, 'accuracy': 0.7068, 'roc_auc_score': 0.7843129771542059, 'f1 score': 0.6979810465595385}, {'Model #': 3, 'average': 0.7296980079045814, 'accuracy': 0.7068, 'roc_auc_score': 0.7843129771542059, 'f1_score': 0.6979810465595385}, {'Model #': 3, 'average': 0.7606319834637382, 'accuracy': 0.7332, 'roc_auc_score': 0.8176475632944405, 'f1 score':

0.7310483870967741}, {'Model #': 3, 'average': 0.7606319834637382, 'accuracy':
0.7332, 'roc_auc_score': 0.8176475632944405, 'f1_score': 0.7310483870967741}]
[{'Model #': 4, 'average': 0.6485412946540371, 'accuracy': 0.6282,
'roc_auc_score': 0.6751302530211838, 'f1 score': 0.6422936309409274}, {'Model
#': 4, 'average': 0.6485412946540371, 'accuracy': 0.6282, 'roc_auc_score':
0.6751302530211838, 'f1_score': 0.6422936309409274}, {'Model #': 4, 'average':
0.621709304049512, 'accuracy': 0.599, 'roc_auc_score': 0.6315370610459298, 'f1
score': 0.6345908511026063}, {'Model #': 4, 'average': 0.621709304049512,
'accuracy': 0.599, 'roc_auc_score': 0.6315370610459298, 'f1_score':
0.6345908511026063}, {'Model #': 4, 'average': 0.6229264883660783, 'accuracy':
0.5864, 'roc_auc_score': 0.6273911434432465, 'f1 score': 0.6549883216549883},
{'Model #': 4, 'average': 0.6229264883660783, 'accuracy': 0.5864,
'roc_auc_score': 0.6273911434432465, 'f1_score': 0.6549883216549883}, {'Model
#': 4, 'average': 0.5847138914489138, 'accuracy': 0.5942, 'roc_auc_score':
0.6390443898130933, 'f1 score': 0.5208972845336481}, {'Model #': 4, 'average':
0.5847138914489138, 'accuracy': 0.5942, 'roc_auc_score': 0.6390443898130933,
'f1_score': 0.5208972845336481}, {'Model #': 4, 'average': 0.6176748467700118,
'accuracy': 0.5864, 'roc_auc_score': 0.621706957892453, 'f1 score':
0.6449175824175823}, {'Model #': 4, 'average': 0.6176748467700118, 'accuracy':
0.5864, 'roc_auc_score': 0.621706957892453, 'f1_score': 0.6449175824175823}]
[{'Model #': 5, 'average': 0.7072981889433686, 'accuracy': 0.6814,
'roc_auc_score': 0.7490361124431215, 'f1 score': 0.6914584543869843}, {'Model
#': 5, 'average': 0.7072981889433686, 'accuracy': 0.6814, 'roc_auc_score':
0.7490361124431215, 'f1_score': 0.6914584543869843}, {'Model #': 5, 'average':
0.7110489140544232, 'accuracy': 0.6858, 'roc_auc_score': 0.7442660390825663, 'f1
score': 0.7030807030807031}, {'Model #': 5, 'average': 0.7110489140544232,
'accuracy': 0.6858, 'roc_auc_score': 0.7442660390825663, 'f1_score':
0.7030807030807031}, {'Model #': 5, 'average': 0.69320483576881, 'accuracy':
0.6374, 'roc_auc_score': 0.7274119274731756, 'f1 score': 0.7148025798332546},
{'Model #': 5, 'average': 0.69320483576881, 'accuracy': 0.6374, 'roc_auc_score':
0.7274119274731756, 'f1_score': 0.7148025798332546}, {'Model #': 5, 'average':
0.7316799684696855, 'accuracy': 0.7096, 'roc_auc_score': 0.7809222008913519, 'f1
score': 0.7045177045177046}, {'Model #': 5, 'average': 0.7316799684696855,
'accuracy': 0.7096, 'roc_auc_score': 0.7809222008913519, 'f1_score':
0.7045177045177046}, {'Model #': 5, 'average': 0.6572762499863596, 'accuracy':
0.6324, 'roc_auc_score': 0.6671748269918892, 'f1 score': 0.6722539229671897},
{'Model #': 5, 'average': 0.6572762499863596, 'accuracy': 0.6324,
'roc_auc_score': 0.6671748269918892, 'f1_score': 0.6722539229671897}]

set of averages of algorithms over 5 trials:
{'1': [0.7216360618212688, 0.725602211140024, 0.6982611604798219,
0.7494914597389917, 0.7561565748411576], '2': [0.6472725902848482,
0.633537417952973, 0.6193329486231925, 0.6450118701002678, 0.6618301953002318],
'3': [0.7225491368446826, 0.7246900498659006, 0.6791173744342903,
0.7250845238499908, 0.7288081064743643], '4': [0.6415731580835212,
0.61816618165116, 0.5956137193263026, 0.5684022811113622, 0.6131959825460319],
'5': [0.6788812159815212, 0.6634056600353703, 0.6545530031986111,
0.6889371606935969, 0.659882205570595]}

```
set of acc values of algorithms over 5 trials
{'1': [0.7015, 0.7055, 0.6765, 0.734, 0.735], '2': [0.6115, 0.604, 0.5845,
0.6155, 0.627], '3': [0.7, 0.7015, 0.66, 0.705, 0.705], '4': [0.62, 0.597,
0.5655, 0.5825, 0.5805], '5': [0.652, 0.6425, 0.6115, 0.671, 0.6375]}

set of roc values of algorithms over 5 trials
{'1': [0.7690917492118866, 0.7763299374076432, 0.752650406504065,
0.7953613908326033, 0.8038778877887788], '2': [0.6602540765870285,
0.6667057118028441, 0.6420132582864291, 0.6698772959500061, 0.6856835683568356],
'3': [0.7646771135043446, 0.7721185690658401, 0.7200940587867417,
0.7734395941604555, 0.7714931493149316], '4': [0.668008193179818,
0.6341340589721718, 0.5881626016260163, 0.6242083448355878, 0.6211941194119412],
'5': [0.7151279784288938, 0.6922952933591231, 0.6671164477798623,
0.7281347144040231, 0.6724427442744274]}

set of f1 values of algorithms over 5 trials
{'1': [0.6943164362519201, 0.6949766960124287, 0.6656330749354005,
0.7191129883843718, 0.7295918367346939], '2': [0.670063694267516,
0.6299065420560747, 0.6314855875831487, 0.6496583143507972, 0.6728070175438597],
'3': [0.702970297029703, 0.7004515805318615, 0.657258064516129,
0.696813977389517, 0.7099311701081613], '4': [0.6367112810707457,
0.6233644859813084, 0.6331785563528916, 0.49849849849849853,
0.6378938282261546], '5': [0.6695156695156694, 0.655421686746988,
0.6850425618159709, 0.6676767676767676, 0.6697038724373576]}
```

[23]:
```python
# calculate average acc metric scores per algorithm over 5 trials
alg_acc_averages = {}
for i in range(len(alg_acc)):
    alg_acc_averages[str(i + 1)] = sum(alg_acc[str(i + 1)])/5

print(alg_acc_averages)
```

```
{'1': 0.7104999999999999, '2': 0.6085, '3': 0.6943, '4': 0.5891, '5': 0.6429}
```

[14]:
```python
# calculate average roc metric scores per algorithm over 5 trials
alg_roc_averages = {}
for i in range(len(alg_roc)):
    alg_roc_averages[str(i + 1)] = sum(alg_roc[str(i + 1)])/5

print(alg_roc_averages)
```

```
{'1': 0.7794622743489954, '2': 0.6649067821966286, '3': 0.7603644969664627, '4':
0.627141463605107, '5': 0.6950234356492659}
```

[15]:
```python
# calculate average f1 metric scores per algorithm over 5 trials
alg_f1_averages = {}
for i in range(len(alg_f1)):
```

```
    alg_f1_averages[str(i + 1)] = sum(alg_f1[str(i + 1)])/5

print(alg_f1_averages)
```

{'1': 0.7007262064637629, '2': 0.6507842311602793, '3': 0.6934850179150744, '4': 0.6059293300259198, '5': 0.6694721116385507}

[16]:
```
averages = {}
for i in range(len(alg_acc_averages)):
    averages[str(i + 1)] = (alg_acc_averages[str(i + 1)] +␣
 ↪alg_roc_averages[str(i + 1)] + alg_f1_averages[str(i + 1)])/3

print(averages)
```

{'1': 0.7302294936042527, '2': 0.6413970044523026, '3': 0.7160498382938458, '4': 0.6073902645436755, '5': 0.6691318490959389}

[17]:
```
# t-test best against rest mean of metrics (RF against rest)
combined_metrics_1 = []
combined_metrics_2 = []
combined_metrics_3 = []
combined_metrics_4 = []
combined_metrics_5 = []

for item in alg_acc['1']:
    combined_metrics_1.append(item)
for item in alg_roc['1']:
    combined_metrics_1.append(item)
for item in alg_f1['1']:
    combined_metrics_1.append(item)

for item in alg_acc['2']:
    combined_metrics_2.append(item)
for item in alg_roc['2']:
    combined_metrics_2.append(item)
for item in alg_f1['2']:
    combined_metrics_2.append(item)

for item in alg_acc['3']:
    combined_metrics_3.append(item)
for item in alg_roc['3']:
    combined_metrics_3.append(item)
for item in alg_f1['3']:
    combined_metrics_3.append(item)

for item in alg_acc['4']:
    combined_metrics_4.append(item)
for item in alg_roc['4']:
```

```
    combined_metrics_4.append(item)
for item in alg_f1['4']:
    combined_metrics_4.append(item)

for item in alg_acc['5']:
    combined_metrics_5.append(item)
for item in alg_roc['5']:
    combined_metrics_5.append(item)
for item in alg_f1['5']:
    combined_metrics_5.append(item)

print(ttest_rel(combined_metrics_1, combined_metrics_2))
print(ttest_rel(combined_metrics_1, combined_metrics_3))
print(ttest_rel(combined_metrics_1, combined_metrics_4))
print(ttest_rel(combined_metrics_1, combined_metrics_5))
```

```
Ttest_relResult(statistic=10.85608253750064, pvalue=3.3462524215480636e-08)
Ttest_relResult(statistic=3.9539094923767744, pvalue=0.0014406524366184668)
Ttest_relResult(statistic=9.13710780210442, pvalue=2.822386600659466e-07)
Ttest_relResult(statistic=6.999851511777664, pvalue=6.249422937916668e-06)
```

Bitcoin Ransomware Detection Dataset

March 19, 2021

1 Bitcoin Ransomware Dataset

See https://archive.ics.uci.edu/ml/datasets/BitcoinHeistRansomwareAddressDataset for dataset information and feature descriptions.

[1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import csv
import numpy as np
import pandas as pd
import pandas_profiling
import matplotlib.pyplot as plt
from scipy import stats
import pickle
import operator
import glob
from scipy.io.arff import loadarff
from scipy.stats import ttest_rel

import seaborn as sns; sns.set_style('white')

from sklearn.utils import resample
from sklearn.metrics import accuracy_score, plot_confusion_matrix, f1_score,␣
↪plot_roc_curve, roc_auc_score, make_scorer
from sklearn.model_selection import KFold, GridSearchCV, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.pipeline import Pipeline
from sklearn.neural_network import MLPClassifier

from sklearn.model_selection import cross_val_score
```

[2]:
```
df = pd.read_csv('BitCoinHeistData.csv')
df = df.rename(columns={'count':'cnt'})
df
```

[2]:
```
                                     address  year  day  length     weight  \
0          111K8kZAEnJg245r2cM6y9zgJGHZtJPy6  2017   11      18   0.008333
1         1123pJv8jzeFQaCV4w644pzQJzVWay2zcA  2016  132      44   0.000244
2         112536im7hy6wtKbpH1qYDWtTyMRAcA2p7  2016  246       0   1.000000
3         1126eDRw2wqSkWosjTCre8cjjQW8sSeWH7  2016  322      72   0.003906
4         1129TSjKtx65E35GiUo4AYVeyo48twbrGX  2016  238     144   0.072848
…                                        …     …    …       …          …
2916692   12D3trgho1vJ4mGtWBRPyHdMJK96TRYSry  2018  330       0   0.111111
2916693   1P7PputTcVkhXBmXBvSD9MJ3UYPsiou1u2  2018  330       0   1.000000
2916694   1KYiKJEfdJtap9QX2v9BXJMpz2SfU4pgZw  2018  330       2  12.000000
2916695   15iPUJsRNZQZHmZZVwmQ63srsmughCXV4a  2018  330       0   0.500000
2916696   3LFFBxp15h9KSFtaw55np8eP5fv6kdK17e  2018  330     144   0.073972

          cnt  looped  neighbors        income           label
0           1       0          2  1.000500e+08  princetonCerber
1           1       0          1  1.000000e+08   princetonLocky
2           1       0          2  2.000000e+08  princetonCerber
3           1       0          2  7.120000e+07  princetonCerber
4         456       0          1  2.000000e+08   princetonLocky
…         …       …          …             …               …
2916692     1       0          1  1.255809e+09            white
2916693     1       0          1  4.409699e+07            white
2916694     6       6         35  2.398267e+09            white
2916695     1       0          1  1.780427e+08            white
2916696  6800       0          2  1.123500e+08            white

[2916697 rows x 10 columns]
```

[3]:
```
# 1 = ransomware; 0 = whitelist
df['new_label'] = np.where(df['label']=='white', 0, 1)
df = df.drop(columns=['year', 'day', 'address', 'label'])
df
```

[3]:
```
         length     weight   cnt  looped  neighbors        income  new_label
0            18   0.008333     1       0          2  1.000500e+08          1
1            44   0.000244     1       0          1  1.000000e+08          1
2             0   1.000000     1       0          2  2.000000e+08          1
3            72   0.003906     1       0          2  7.120000e+07          1
4           144   0.072848   456       0          1  2.000000e+08          1
…           …          …     …     …          …             …        …
2916692       0   0.111111     1       0          1  1.255809e+09          0
2916693       0   1.000000     1       0          1  4.409699e+07          0
2916694       2  12.000000     6       6         35  2.398267e+09          0
```

```
2916695       0   0.500000     1       0          1  1.780427e+08          0
2916696     144   0.073972  6800       0          2  1.123500e+08          0

[2916697 rows x 7 columns]
```

[4]:
```
df['new_label'].value_counts()
```

[4]:
```
0    2875284
1      41413
Name: new_label, dtype: int64
```

[5]:
```
# Separate majority and minority classes
df_majority = df[df.new_label==0]
df_minority = df[df.new_label==1]

# Downsample majority and minority class
df_majority_downsampled = resample(df_majority,
                                 replace=False,    # sample without replacement
                                 n_samples=6000,     # to match minority class
                                 random_state=123) # reproducible results

df_minority_downsampled = resample(df_minority,
                                 replace=False,    # sample without replacement
                                 n_samples=6000,     # to match majority class
                                 random_state=123) # reproducible results

# Combine minority class with downsampled majority class
df_downsampled = pd.concat([df_majority_downsampled, df_minority_downsampled])

# Display new class counts
df_downsampled.new_label.value_counts()
```

[5]:
```
1    6000
0    6000
Name: new_label, dtype: int64
```

[6]:
```
df_downsampled
```

[6]:

	length	weight	cnt	looped	neighbors	income	new_label
2488416	2	1.000000	1	0	2	6.089024e+08	0
2873770	2	0.500000	1	0	2	3.969468e+08	0
2537321	50	0.022457	6	0	2	1.228130e+08	0
2868318	144	0.077579	4825	0	2	2.000000e+08	0
636908	0	1.000000	1	0	2	5.605605e+09	0
…	…	…	…	…	…	…	…
23778	2	0.500000	1	0	1	3.000000e+08	1
6969	0	0.500000	1	0	2	1.220000e+08	1

```
36643          0  1.000000     1       0          1  5.000000e+07        1
31367          4  1.588816     4       0          8  8.068000e+08        1
3945           0  0.500000     1       0          2  1.200000e+08        1

[12000 rows x 7 columns]
```

[7]:
```
df = df_downsampled.copy()
scaler = StandardScaler()
X, y = df.iloc[:,0:6].to_numpy(), df.iloc[:,6:].to_numpy()
```

1.1 Hyperparameter Search & Experimentation

[11]:
```
def experiment():
    pipeline1 = Pipeline((
    ('clf', RandomForestClassifier()),
    ))

    pipeline2 = Pipeline((
    ('clf', KNeighborsClassifier()),
    ))

    pipeline3 = Pipeline((
    ('clf', AdaBoostClassifier()),
    ))

    pipeline4 = Pipeline((
    ('clf', LogisticRegression()),
    ))

    pipeline5 = Pipeline((
    ('clf', MLPClassifier()),
    ))

    # Random Forest
    parameters1 = {
    'clf__n_estimators': [1024],
    'clf__max_features': [1, 2, 4, 6, 8, 12, 16, 20]
    }

    # KNN
    parameters2 = {
    'clf__n_neighbors':␣
↪[1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61,65,69,73,77,81,85,89,93,97,101,105],
    'clf__weights': ['uniform', 'distance']
    }

    # AdaBoost (Boosted Decision Tree)
```

```
    parameters3 = {
        'clf__algorithm': ['SAMME.R'],
        'clf__n_estimators': [2,4,8,16,32,64,128,256,512,1024,2048],
        'clf__learning_rate': [1e-3, 1e-2, 1e-1, 1e0, 1e1, 2e1, 5e1]
    }

    # Logistic
    parameters4 = {
    'clf__penalty':['l1', 'l2', None],
    'clf__C':[1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 0, 1e0, 1e1, 1e2,␣
↪1e3, 1e4],
    'clf__max_iter':[5000]
    }

    # Multi-layer Perceptron
    parameters5 = {
        'clf__hidden_layer_sizes':[(1,), (2,), (4,), (8,), (32,), (128,)],
        'clf__solver':['sgd'],
        'clf__activation':['relu'],
        'clf__learning_rate':['constant', 'invscaling'],
        'clf__learning_rate_init': [1e-3, 1e-2, 1e-1, 1e0],
        'clf__max_iter': [2, 4, 8, 16, 32, 64, 128, 256, 512]
    }

    pars = [parameters1, parameters2, parameters3, parameters4, parameters5]
    pips = [pipeline1, pipeline2, pipeline3, pipeline4, pipeline5]

    # List of dictionaries to hold the scores of the various metrics for each␣
↪type of classifier
    best_clf_list = []
    trial_storage = {}
    training_storage = {}

    print("starting Gridsearch")
    for i in range(len(pars)):
        trial_averages = []
        train_performance = []
        for t in range(5):
            # split and scale data
            X_train, X_test, y_train, y_test = train_test_split(X, y,␣
↪test_size=1/6, random_state=t)
            X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,␣
↪test_size=0.5, random_state=t)
            X_train = scaler.fit_transform(X_train)
            X_val = scaler.transform(X_val)
            X_test = scaler.transform(X_test)
```

```
            clf = GridSearchCV(pips[i], pars[i], refit=False, n_jobs=8, cv=5,␣
↪verbose=3, scoring=('accuracy', 'roc_auc', 'f1'))
            clf = clf.fit(X_val, y_val)

            print("finished Gridsearch trial " + str(t + 1) + " classifier " +␣
↪str(i + 1))
            print("")
            print("")

            # find the best params for each metric in a given trial
            best_index_acc = np.argmin(clf.cv_results_['rank_test_accuracy'])
            best_params_acc = clf.cv_results_['params'][best_index_acc]
            best_index_roc = np.argmin(clf.cv_results_['rank_test_roc_auc'])
            best_params_roc = clf.cv_results_['params'][best_index_roc]
            best_index_f1 = np.argmin(clf.cv_results_['rank_test_f1'])
            best_params_f1 = clf.cv_results_['params'][best_index_f1]

            # train and test models for given metric with their corresponding␣
↪best parameter settings
            pipe = pips[i]
            clf_acc = pipe.set_params(**best_params_acc)
            clf_acc = clf_acc.fit(X_train, y_train)
            clf_roc = pipe.set_params(**best_params_roc)
            clf_roc = clf_roc.fit(X_train, y_train)
            clf_f1 = pipe.set_params(**best_params_f1)
            clf_f1 = clf_f1.fit(X_train, y_train)

            # get training set performance
            train_acc = accuracy_score(y_train, clf_acc.predict(X_train))
            train_roc = roc_auc_score(y_train, clf_roc.predict_proba(X_train)[:
↪, 1])
            train_f1 = f1_score(y_train, clf_f1.predict(X_train))

            train_performance.append({
                'Model #': i + 1,
                'average': (train_f1 + train_acc + train_roc)/3,
                'accuracy': train_acc,
                'roc_auc_score': train_roc,
                'f1 score': train_f1
            })

            # get test set performances
            trial_acc = clf_acc.score(X_test, y_test)
            trial_roc = roc_auc_score(y_test, clf_roc.predict_proba(X_test)[:,␣
↪1])
            trial_f1 = f1_score(y_test, clf_f1.predict(X_test))
```

```
            # store scores and their averages in list containing averages for␣
↪each trial
            trial_averages.append({
                'Model #': i + 1, # model number corresponds to the numbers␣
↪used in pipeline above (i.e. 1 = Random Forest)
                'average':(trial_acc + trial_roc + trial_f1) / 3,
                'accuracy': trial_acc,
                'roc_auc_score': trial_roc,
                'f1_score': trial_f1
            })

            train_performance.append({
                'Model #': i + 1, # model number corresponds to the numbers␣
↪used in pipeline above (i.e. 1 = Random Forest)
                'average':(train_acc + train_roc + train_f1) / 3,
                'accuracy': train_acc,
                'roc_auc_score': train_roc,
                'f1_score': train_f1
            })

        # find the trial with the best average metric scores and append those␣
↪scores as a dict to best clf list
        max_average = 0
        for trial in trial_averages:
            if trial['average'] > max_average:
                max_average = trial['average']
                best_trial = trial

        best_clf_list.append(best_trial)
        training_storage[str(i + 1)]=train_performance
        trial_storage[str(i + 1)]=trial_averages

    return best_clf_list, trial_storage, training_storage
```

[12]:
```
%%capture --no-stdout --no-display
best_clf_list, trial_storage, training_perf = experiment()
```

```
starting Gridsearch
Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 1 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 2 classifier 1


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

```
finished Gridsearch trial 3 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 4 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 5 classifier 1


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 1 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 2 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 3 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 4 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 5 classifier 2


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 1 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 2 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 3 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 4 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
```

```
finished Gridsearch trial 5 classifier 3


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 1 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 2 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 3 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 4 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 5 classifier 4


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 1 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 2 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 3 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 4 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 5 classifier 5
```

1.2 Calculating and Organizing Results

[14]:
```
print('Best Models On Average For Test Set:')
for element in best_clf_list:
    print(element)

print()

print('Train Set Data')
for i in range(len(training_perf)):
    print(training_perf[str(i + 1)])

print()

alg_avg = {}
alg_acc = {}
alg_roc = {}
alg_f1 = {}

for i in range(len(trial_storage)):
    alg_avg[str(i + 1)]=[]
    alg_acc[str(i + 1)]=[]
    alg_roc[str(i + 1)]=[]
    alg_f1[str(i + 1)]=[]
    for entry in trial_storage[str(i + 1)]:
        alg_avg[str(i + 1)].append(entry['average'])
        alg_acc[str(i + 1)].append(entry['accuracy'])
        alg_roc[str(i + 1)].append(entry['roc_auc_score'])
        alg_f1[str(i + 1)].append(entry['f1_score'])

print('set of averages of algorithms over 5 trials:')
print(alg_avg)
print()
print('set of acc values of algorithms over 5 trials')
print(alg_acc)
print()
print('set of roc values of algorithms over 5 trials')
print(alg_roc)
print()
print('set of f1 values of algorithms over 5 trials')
print(alg_f1)
```

```
Best Models On Average For Test Set:
{'Model #': 1, 'average': 0.7397841712938319, 'accuracy': 0.7155,
'roc_auc_score': 0.779130694820054, 'f1_score': 0.7247218190614417}
{'Model #': 2, 'average': 0.6740052911361613, 'accuracy': 0.654,
'roc_auc_score': 0.7072315596829936, 'f1_score': 0.6607843137254902}
{'Model #': 3, 'average': 0.730361630281657, 'accuracy': 0.711, 'roc_auc_score':
```

0.7731883391208333, 'f1_score': 0.7068965517241379}
{'Model #': 4, 'average': 0.6369932174616385, 'accuracy': 0.597,
'roc_auc_score': 0.6237413741374138, 'f1_score': 0.690238278247502}
{'Model #': 5, 'average': 0.6760133373701054, 'accuracy': 0.65, 'roc_auc_score':
0.712968241775388, 'f1_score': 0.6650717703349281}

Train Set Data

[{'Model #': 1, 'average': 0.9896706077254435, 'accuracy': 0.986,
'roc_auc_score': 0.9968458152711918, 'f1 score': 0.9861660079051385}, {'Model
#': 1, 'average': 0.9896706077254435, 'accuracy': 0.986, 'roc_auc_score':
0.9968458152711918, 'f1_score': 0.9861660079051385}, {'Model #': 1, 'average':
0.9885347265654086, 'accuracy': 0.9844, 'roc_auc_score': 0.9966191994590718, 'f1
score': 0.9845849802371541}, {'Model #': 1, 'average': 0.9885347265654086,
'accuracy': 0.9844, 'roc_auc_score': 0.9966191994590718, 'f1_score':
0.9845849802371541}, {'Model #': 1, 'average': 0.9911563027776232, 'accuracy':
0.988, 'roc_auc_score': 0.9973783962248906, 'f1 score': 0.9880905121079794},
{'Model #': 1, 'average': 0.9911563027776232, 'accuracy': 0.988,
'roc_auc_score': 0.9973783962248906, 'f1_score': 0.9880905121079794}, {'Model
#': 1, 'average': 0.9890587078721288, 'accuracy': 0.985, 'roc_auc_score':
0.9972152252807137, 'f1 score': 0.9849608983356727}, {'Model #': 1, 'average':
0.9890587078721288, 'accuracy': 0.985, 'roc_auc_score': 0.9972152252807137,
'f1_score': 0.9849608983356727}, {'Model #': 1, 'average': 0.9893414100594601,
'accuracy': 0.9856, 'roc_auc_score': 0.996692957883493, 'f1 score':
0.9857312722948871}, {'Model #': 1, 'average': 0.9893414100594601, 'accuracy':
0.9856, 'roc_auc_score': 0.996692957883493, 'f1_score': 0.9857312722948871}]
[{'Model #': 2, 'average': 0.72469402050915, 'accuracy': 0.6968,
'roc_auc_score': 0.7646209546282842, 'f1 score': 0.7126611068991658}, {'Model
#': 2, 'average': 0.72469402050915, 'accuracy': 0.6968, 'roc_auc_score':
0.7646209546282842, 'f1_score': 0.7126611068991658}, {'Model #': 2, 'average':
0.7291600932540728, 'accuracy': 0.6988, 'roc_auc_score': 0.7734760437561671, 'f1
score': 0.7152042360060515}, {'Model #': 2, 'average': 0.7291600932540728,
'accuracy': 0.6988, 'roc_auc_score': 0.7734760437561671, 'f1_score':
0.7152042360060515}, {'Model #': 2, 'average': 0.7335154491888453, 'accuracy':
0.7012, 'roc_auc_score': 0.7766291983460455, 'f1 score': 0.7227171492204899},
{'Model #': 2, 'average': 0.7335154491888453, 'accuracy': 0.7012,
'roc_auc_score': 0.7766291983460455, 'f1_score': 0.7227171492204899}, {'Model
#': 2, 'average': 0.6731406149895767, 'accuracy': 0.6436, 'roc_auc_score':
0.7069255238984957, 'f1 score': 0.6688963210702341}, {'Model #': 2, 'average':
0.6731406149895767, 'accuracy': 0.6436, 'roc_auc_score': 0.7069255238984957,
'f1_score': 0.6688963210702341}, {'Model #': 2, 'average': 0.7282117122283416,
'accuracy': 0.6944, 'roc_auc_score': 0.7776692977083506, 'f1 score':
0.7125658389766741}, {'Model #': 2, 'average': 0.7282117122283416, 'accuracy':
0.6944, 'roc_auc_score': 0.7776692977083506, 'f1_score': 0.7125658389766741}]
[{'Model #': 3, 'average': 0.7826221272322312, 'accuracy': 0.7572,
'roc_auc_score': 0.841596518116702, 'f1 score': 0.7490698635799917}, {'Model #':
3, 'average': 0.7826221272322312, 'accuracy': 0.7572, 'roc_auc_score':
0.841596518116702, 'f1_score': 0.7490698635799917}, {'Model #': 3, 'average':
0.7513417504283962, 'accuracy': 0.7274, 'roc_auc_score': 0.805300368848059, 'f1

score': 0.7213248824371296}, {'Model #': 3, 'average': 0.7513417504283962,
'accuracy': 0.7274, 'roc_auc_score': 0.805300368848059, 'f1_score':
0.7213248824371296}, {'Model #': 3, 'average': 0.7759264666244775, 'accuracy':
0.7522, 'roc_auc_score': 0.8346218418554523, 'f1 score': 0.7409575580179804},
{'Model #': 3, 'average': 0.7759264666244775, 'accuracy': 0.7522,
'roc_auc_score': 0.8346218418554523, 'f1_score': 0.7409575580179804}, {'Model
#': 3, 'average': 0.7474944978213438, 'accuracy': 0.7214, 'roc_auc_score':
0.804732444797772, 'f1 score': 0.7163510486662594}, {'Model #': 3, 'average':
0.7474944978213438, 'accuracy': 0.7214, 'roc_auc_score': 0.804732444797772,
'f1_score': 0.7163510486662594}, {'Model #': 3, 'average': 0.6902500562681393,
'accuracy': 0.666, 'roc_auc_score': 0.7171332589652858, 'f1 score':
0.687616909839132}, {'Model #': 3, 'average': 0.6902500562681393, 'accuracy':
0.666, 'roc_auc_score': 0.7171332589652858, 'f1_score': 0.687616909839132}]
[{'Model #': 4, 'average': 0.6322166932706624, 'accuracy': 0.5952,
'roc_auc_score': 0.6096108593491373, 'f1 score': 0.6918392204628502}, {'Model
#': 4, 'average': 0.6322166932706624, 'accuracy': 0.5952, 'roc_auc_score':
0.6096108593491373, 'f1_score': 0.6918392204628502}, {'Model #': 4, 'average':
0.6276307861869385, 'accuracy': 0.5938, 'roc_auc_score': 0.601121856179497, 'f1
score': 0.6879705023813181}, {'Model #': 4, 'average': 0.6276307861869383,
'accuracy': 0.5938, 'roc_auc_score': 0.601121856179497, 'f1_score':
0.6879705023813181}, {'Model #': 4, 'average': 0.6170462399940608, 'accuracy':
0.575, 'roc_auc_score': 0.6036159892070245, 'f1 score': 0.672522730775158},
{'Model #': 4, 'average': 0.6170462399940608, 'accuracy': 0.575,
'roc_auc_score': 0.6036159892070245, 'f1_score': 0.672522730775158}, {'Model #':
4, 'average': 0.6108843825400015, 'accuracy': 0.5884, 'roc_auc_score':
0.6069602325688944, 'f1 score': 0.6372929150511104}, {'Model #': 4, 'average':
0.6108843825400015, 'accuracy': 0.5884, 'roc_auc_score': 0.6069602325688944,
'f1_score': 0.6372929150511104}, {'Model #': 4, 'average': 0.6307999905904254,
'accuracy': 0.59, 'roc_auc_score': 0.6164318345163742, 'f1 score':
0.685968137254902}, {'Model #': 4, 'average': 0.6307999905904254, 'accuracy':
0.59, 'roc_auc_score': 0.6164318345163742, 'f1_score': 0.685968137254902}]
[{'Model #': 5, 'average': 0.6813977289707274, 'accuracy': 0.6492,
'roc_auc_score': 0.6913771003964672, 'f1 score': 0.7036160865157147}, {'Model
#': 5, 'average': 0.6813977289707274, 'accuracy': 0.6492, 'roc_auc_score':
0.6913771003964672, 'f1_score': 0.7036160865157147}, {'Model #': 5, 'average':
0.6415233205897778, 'accuracy': 0.617, 'roc_auc_score': 0.6696421871427499, 'f1
score': 0.6379277746265835}, {'Model #': 5, 'average': 0.6415233205897778,
'accuracy': 0.617, 'roc_auc_score': 0.6696421871427499, 'f1_score':
0.6379277746265835}, {'Model #': 5, 'average': 0.6511876039768523, 'accuracy':
0.6126, 'roc_auc_score': 0.6584517481705174, 'f1 score': 0.6825110637600393},
{'Model #': 5, 'average': 0.6511876039768523, 'accuracy': 0.6126,
'roc_auc_score': 0.6584517481705174, 'f1_score': 0.6825110637600393}, {'Model
#': 5, 'average': 0.699344575447375, 'accuracy': 0.6722, 'roc_auc_score':
0.7397582618717176, 'f1 score': 0.6860754644704079}, {'Model #': 5, 'average':
0.699344575447375, 'accuracy': 0.6722, 'roc_auc_score': 0.7397582618717176,
'f1_score': 0.6860754644704079}, {'Model #': 5, 'average': 0.6085182302322812,
'accuracy': 0.6004, 'roc_auc_score': 0.6939486841271578, 'f1 score':
0.5312060065696856}, {'Model #': 5, 'average': 0.6085182302322812, 'accuracy':

```
0.6004, 'roc_auc_score': 0.6939486841271578, 'f1_score': 0.5312060065696856}]

set of averages of algorithms over 5 trials:
{'1': [0.7397841712938319, 0.7295932348050758, 0.7364385320712664,
0.7257956155589397, 0.7257140550019482], '2': [0.6676176613693542,
0.6740052911361613, 0.6641355502586234, 0.6507480056613185, 0.672099431323765],
'3': [0.730361630281657, 0.725492978749458, 0.7160258784928986,
0.7146393832181008, 0.6791102285801099], '4': [0.6284974285229111,
0.6210954610550077, 0.6115011492605995, 0.5917756804544217, 0.6369932174616385],
'5': [0.6559063204500494, 0.6234998640953213, 0.6398691869128804,
0.6760133373701054, 0.6003038535357116]}

set of acc values of algorithms over 5 trials
{'1': [0.7155, 0.708, 0.7125, 0.697, 0.6975], '2': [0.644, 0.654, 0.633, 0.622,
0.651], '3': [0.711, 0.708, 0.696, 0.693, 0.657], '4': [0.594, 0.588, 0.566,
0.568, 0.597], '5': [0.623, 0.605, 0.602, 0.65, 0.597]}

set of roc values of algorithms over 5 trials
{'1': [0.779130694820054, 0.7724879961235188, 0.7850361475922452,
0.7815796896191654, 0.7781128112811282], '2': [0.6924987198156535,
0.7072315596829936, 0.6970331457160724, 0.6805683914047154, 0.6999099909990999],
'3': [0.7731883391208333, 0.7814478537188991, 0.7744530331457161,
0.7610191597553122, 0.7076207620762076], '4': [0.5947036373237746,
0.5966592380792196, 0.604416760475297, 0.5862744097843178, 0.6237413741374138],
'5': [0.6623263749979996, 0.6489947379170321, 0.6454494058786743,
0.712968241775388, 0.684818481848185]}

set of f1 values of algorithms over 5 trials
{'1': [0.7247218190614417, 0.7082917082917084, 0.711779448621554,
0.6988071570576541, 0.7015293537247164], '2': [0.6663542642924087,
0.6607843137254902, 0.6623735050597975, 0.6496756255792401, 0.6653883029721955],
'3': [0.7068965517241379, 0.6870310825294748, 0.6776246023329798,
0.68989898989899, 0.6727099236641222], '4': [0.6967886482449589,
0.6786271450858035, 0.6640866873065016, 0.6210526315789474, 0.690238278247502],
'5': [0.6823925863521483, 0.616504854368932, 0.6721581548599671,
0.6650717703349281, 0.5190930787589499]}
```

[15]:

```python
# calculate average acc metric scores per algorithm over 5 trials
alg_acc_averages = {}
for i in range(len(alg_acc)):
    alg_acc_averages[str(i + 1)] = sum(alg_acc[str(i + 1)])/5

print(alg_acc_averages)
```

```
{'1': 0.7061, '2': 0.6407999999999999, '3': 0.6930000000000001, '4': 0.5826,
'5': 0.6154}
```

[16]:
```
# calculate average roc metric scores per algorithm over 5 trials
alg_roc_averages = {}
for i in range(len(alg_roc)):
    alg_roc_averages[str(i + 1)] = sum(alg_roc[str(i + 1)])/5

print(alg_roc_averages)
```

```
{'1': 0.7792694678872223, '2': 0.695448361523707, '3': 0.7595458295633938, '4':
0.6011590839600045, '5': 0.6709114484834557}
```

[17]:
```
# calculate average f1 metric scores per algorithm over 5 trials
alg_f1_averages = {}
for i in range(len(alg_f1)):
    alg_f1_averages[str(i + 1)] = sum(alg_f1[str(i + 1)])/5

print(alg_f1_averages)
```

```
{'1': 0.7090258973514149, '2': 0.6609152023258265, '3': 0.686832230029941, '4':
0.6701586780927427, '5': 0.6310440889349851}
```

[18]:
```
averages = {}
for i in range(len(alg_acc_averages)):
    averages[str(i + 1)] = (alg_acc_averages[str(i + 1)] +␣
 ↪alg_roc_averages[str(i + 1)] + alg_f1_averages[str(i + 1)])/3

print(averages)
```

```
{'1': 0.7314651217462124, '2': 0.6657211879498445, '3': 0.7131260198644448, '4':
0.6179725873509158, '5': 0.6391185124728136}
```

[19]:
```
# t-test best against rest mean of metrics (RF against rest)
combined_metrics_1 = []
combined_metrics_2 = []
combined_metrics_3 = []
combined_metrics_4 = []
combined_metrics_5 = []

for item in alg_acc['1']:
    combined_metrics_1.append(item)
for item in alg_roc['1']:
    combined_metrics_1.append(item)
for item in alg_f1['1']:
    combined_metrics_1.append(item)

for item in alg_acc['2']:
    combined_metrics_2.append(item)
for item in alg_roc['2']:
    combined_metrics_2.append(item)
for item in alg_f1['2']:
```

```python
    combined_metrics_2.append(item)

for item in alg_acc['3']:
    combined_metrics_3.append(item)
for item in alg_roc['3']:
    combined_metrics_3.append(item)
for item in alg_f1['3']:
    combined_metrics_3.append(item)

for item in alg_acc['4']:
    combined_metrics_4.append(item)
for item in alg_roc['4']:
    combined_metrics_4.append(item)
for item in alg_f1['4']:
    combined_metrics_4.append(item)

for item in alg_acc['5']:
    combined_metrics_5.append(item)
for item in alg_roc['5']:
    combined_metrics_5.append(item)
for item in alg_f1['5']:
    combined_metrics_5.append(item)

print(ttest_rel(combined_metrics_1, combined_metrics_2))
print(ttest_rel(combined_metrics_1, combined_metrics_3))
print(ttest_rel(combined_metrics_1, combined_metrics_4))
print(ttest_rel(combined_metrics_1, combined_metrics_5))
```

```
Ttest_relResult(statistic=13.536554913411994, pvalue=1.9634473848890334e-09)
Ttest_relResult(statistic=3.626745938180636, pvalue=0.002748710807162354)
Ttest_relResult(statistic=7.091129109155958, pvalue=5.411836315428835e-06)
Ttest_relResult(statistic=8.681118334491316, pvalue=5.224431932512865e-07)
```

Misc Calculations

March 19, 2021

[1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import csv
import numpy as np
import pandas as pd
import pandas_profiling
import matplotlib.pyplot as plt
from scipy import stats
import pickle
import operator
import glob
from scipy.io.arff import loadarff
from scipy.stats import ttest_rel, ttest_ind

import seaborn as sns; sns.set_style('white')

from sklearn.utils import resample
from sklearn.metrics import accuracy_score, plot_confusion_matrix, f1_score,␣
↪plot_roc_curve, roc_auc_score, make_scorer
from sklearn.model_selection import KFold, GridSearchCV, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.pipeline import Pipeline
from sklearn.neural_network import MLPClassifier

from sklearn.model_selection import cross_val_score
```

[2]:
```
# 1 = RF, 2 = KNN, 3 = Boost-DT, 4 = Log, 5 = ANN
htru_acc = {
    '1': [0.98, 0.98, 0.9805, 0.9755, 0.9765],
```

```
    '2': [0.9725, 0.97, 0.976, 0.968, 0.9665],
    '3': [0.968, 0.963, 0.9695, 0.966, 0.966],
    '4': [0.95, 0.947, 0.9405, 0.9445, 0.946],
    '5': [0.9555, 0.951, 0.945, 0.9475, 0.951]
}

htru_roc = {
    '1': [0.9961039389672113, 0.9950482758068327, 0.9944005003126954, 0.
 ↪9939938227892666, 0.9943374337433742],
    '2': [0.9933840473028117, 0.991702407983629, 0.9901558474046278, 0.
 ↪9891478729831047, 0.9916826682668267],
    '3': [0.9907416668000194, 0.9910426452979646, 0.9928655409631019, 0.
 ↪992091449924185, 0.9924912491249125],
    '4': [0.9819744043142212, 0.9793541333781851, 0.9739577235772358, 0.
 ↪975645226464387, 0.980974097409741],
    '5': [0.9853428893760702, 0.9848344686457544, 0.9826161350844278, 0.
 ↪9814243591743982, 0.9849294929492949]
}

htru_f1 = {
    '1': [0.980295566502463, 0.9791449426485923, 0.9799897383273473, 0.
 ↪9750889679715302, 0.9763462506290891],
    '2': [0.9731051344743277, 0.9692622950819672, 0.9755102040816327, 0.
 ↪9675126903553299, 0.9663823381836427],
    '3': [0.968410661401777, 0.9614984391259105, 0.9687339825730394, 0.
 ↪9655521783181358, 0.9657258064516129],
    '4': [0.949443882709808, 0.9433155080213904, 0.9367357788410421, 0.
 ↪9418543740178104, 0.9442148760330578],
    '5': [0.9552988448016073, 0.9478168264110757, 0.9413020277481323, 0.
 ↪9455676516329704, 0.9496402877697842]
}

susy_acc = {
    '1': [0.792, 0.771, 0.7865, 0.7855, 0.7955],
    '2': [0.7555, 0.7505, 0.7485, 0.7575, 0.7585],
    '3': [0.788, 0.771, 0.779, 0.7945, 0.7915],
    '4': [0.782, 0.7735, 0.775, 0.7825, 0.791],
    '5': [0.795, 0.7695, 0.7875, 0.802, 0.7965]
}

susy_roc = {
    '1': [0.8553576715046967, 0.8360234431002238, 0.8509038148843027, 0.
 ↪8539913823109329, 0.8675452545254526],
    '2': [0.8160450104815093, 0.8134052965227804, 0.8217005628517823, 0.
 ↪8281623198146837, 0.8328402840284028],
```

```
    '3': [0.8613070282120626, 0.8457236565469683, 0.8582243902439024, 0.
↪8625510600077615, 0.8665706570657066],
    '4': [0.8531308508425213, 0.8401872564684776, 0.8512280175109443, 0.
↪8634232309532668, 0.871051105110511],
    '5': [0.8644909867020851, 0.8489939369911618, 0.8609450906816759, 0.
↪8677230737224496, 0.874019901990199]
}

susy_f1 = {
    '1': [0.7835587929240375, 0.7458379578246392, 0.769811320754717, 0.
↪7684835402050728, 0.783483324510323],
    '2': [0.7352463454250134, 0.7103888566453859, 0.7007733491969066, 0.
↪7131874630396215, 0.7281935846933033],
    '3': [0.7832310838445808, 0.7524324324324324, 0.7613390928725702, 0.
↪7791509940891993, 0.7849406910778751],
    '4': [0.7710084033613446, 0.7473508087005019, 0.7483221476510067, 0.
↪7590027700831025, 0.7728260869565217],
    '5': [0.7880041365046535, 0.7517501346257404, 0.7716281569048898, 0.
↪7836065573770491, 0.7859021567596003]
}

higgs_acc = {
    '1': [0.7015, 0.7055, 0.6765, 0.734, 0.735],
    '2': [0.6115, 0.604, 0.5845, 0.6155, 0.627],
    '3': [0.7, 0.7015, 0.66, 0.705, 0.705],
    '4': [0.62, 0.597, 0.5655, 0.5825, 0.5805],
    '5': [0.652, 0.6425, 0.6115, 0.671, 0.6375]
}

higgs_roc = {
    '1': [0.7690917492118866, 0.7763299374076432, 0.752650406504065, 0.
↪7953613908326033, 0.8038778877887788],
    '2': [0.6602540765870285, 0.6667057118028441, 0.6420132582864291, 0.
↪6698772959500061, 0.6856835683568356],
    '3': [0.7646771135043446, 0.7721185690658401, 0.7200940587867417, 0.
↪7734395941604555, 0.7714931493149316],
    '4': [0.668008193179818, 0.6341340589721718, 0.5881626016260163, 0.
↪6242083448355878, 0.6211941194119412],
    '5': [0.7151279784288938, 0.6922952933591231, 0.6671164477798623, 0.
↪7281347144040231, 0.6724427442744274]
}

higgs_f1 = {
    '1': [0.6943164362519201, 0.6949766960124287, 0.6656330749354005, 0.
↪7191129883843718, 0.7295918367346939],
```

```
    '2': [0.670063694267516, 0.6299065420560747, 0.6314855875831487, 0.
↪6496583143507972, 0.6728070175438597],
    '3': [0.702970297029703, 0.7004515805318615, 0.657258064516129, 0.
↪696813977389517, 0.7099311701081613],
    '4': [0.6367112810707457, 0.6233644859813084, 0.6331785563528916, 0.
↪49849849849849853, 0.6378938282261546],
    '5': [0.6695156695156694, 0.655421686746988, 0.6850425618159709, 0.
↪6676767676767676, 0.6697038724373576]
}

bit_acc = {
    '1': [0.7155, 0.708, 0.7125, 0.697, 0.6975],
    '2': [0.644, 0.654, 0.633, 0.622, 0.651],
    '3': [0.711, 0.708, 0.696, 0.693, 0.657],
    '4': [0.594, 0.588, 0.566, 0.568, 0.597],
    '5': [0.623, 0.605, 0.602, 0.65, 0.597]
}

bit_roc = {
    '1': [0.779130694820054, 0.7724879961235188, 0.7850361475922452, 0.
↪7815796896191654, 0.7781128112811282],
     '2': [0.6924987198156535, 0.7072315596829936, 0.6970331457160724, 0.
↪6805683914047154, 0.6999099909990999],
     '3': [0.7731883391208333, 0.7814478537188991, 0.7744530331457161, 0.
↪7610191597553122, 0.7076207620762076],
     '4': [0.5947036373237746, 0.5966592380792196, 0.604416760475297, 0.
↪5862744097843178, 0.6237413741374138],
     '5': [0.6623263749979996, 0.6489947379170321, 0.6454494058786743, 0.
↪712968241775388, 0.684818481848185]
}

bit_f1 = {
    '1': [0.7247218190614417, 0.7082917082917084, 0.711779448621554, 0.
↪6988071570576541, 0.7015293537247164],
    '2': [0.6663542642924087, 0.6607843137254902, 0.6623735050597975, 0.
↪6496756255792401, 0.6653883029721955],
    '3': [0.7068965517241379, 0.6870310825294748, 0.6776246023329798, 0.
↪68989898989899, 0.6727099236641222],
    '4': [0.6967886482449589, 0.6786271450858035, 0.6640866873065016, 0.
↪6210526315789474, 0.690238278247502],
    '5': [0.6823925863521483, 0.616504854368932, 0.6721581548599671, 0.
↪6650717703349281, 0.5190930787589499]
}
```

[3]:
```
all_acc = {}
all_acc_1 = []
```

```
all_acc_2 = []
all_acc_3 = []
all_acc_4 = []
all_acc_5 = []

all_acc_1.append(sum(htru_acc['1']))
all_acc_1.append(sum(susy_acc['1']))
all_acc_1.append(sum(higgs_acc['1']))
all_acc_1.append(sum(bit_acc['1']))
all_acc_2.append(sum(htru_acc['2']))
all_acc_2.append(sum(susy_acc['2']))
all_acc_2.append(sum(higgs_acc['2']))
all_acc_2.append(sum(bit_acc['2']))
all_acc_3.append(sum(htru_acc['3']))
all_acc_3.append(sum(susy_acc['3']))
all_acc_3.append(sum(higgs_acc['3']))
all_acc_3.append(sum(bit_acc['3']))
all_acc_4.append(sum(htru_acc['4']))
all_acc_4.append(sum(susy_acc['4']))
all_acc_4.append(sum(higgs_acc['4']))
all_acc_4.append(sum(bit_acc['4']))
all_acc_5.append(sum(htru_acc['5']))
all_acc_5.append(sum(susy_acc['5']))
all_acc_5.append(sum(higgs_acc['5']))
all_acc_5.append(sum(bit_acc['5']))

all_acc['1']=sum(all_acc_1)/20
all_acc['2']=sum(all_acc_2)/20
all_acc['3']=sum(all_acc_3)/20
all_acc['4']=sum(all_acc_4)/20
all_acc['5']=sum(all_acc_5)/20

print(all_acc)
```

```
{'1': 0.7953, '2': 0.7435, '3': 0.7846500000000001, '4': 0.7245250000000001,
'5': 0.7495999999999999}
```

[4]:

```
all_roc = {}
all_roc_1 = []
all_roc_2 = []
all_roc_3 = []
all_roc_4 = []
all_roc_5 = []

all_roc_1.append(sum(htru_roc['1']))
all_roc_1.append(sum(susy_roc['1']))
all_roc_1.append(sum(higgs_roc['1']))
```

```
all_roc_1.append(sum(bit_roc['1']))
all_roc_2.append(sum(htru_roc['2']))
all_roc_2.append(sum(susy_roc['2']))
all_roc_2.append(sum(higgs_roc['2']))
all_roc_2.append(sum(bit_roc['2']))
all_roc_3.append(sum(htru_roc['3']))
all_roc_3.append(sum(susy_roc['3']))
all_roc_3.append(sum(higgs_roc['3']))
all_roc_3.append(sum(bit_roc['3']))
all_roc_4.append(sum(htru_roc['4']))
all_roc_4.append(sum(susy_roc['4']))
all_roc_4.append(sum(higgs_roc['4']))
all_roc_4.append(sum(bit_roc['4']))
all_roc_5.append(sum(htru_roc['5']))
all_roc_5.append(sum(susy_roc['5']))
all_roc_5.append(sum(higgs_roc['5']))
all_roc_5.append(sum(bit_roc['5']))

all_roc['1']=sum(all_roc_1)/20
all_roc['2']=sum(all_roc_2)/20
all_roc['3']=sum(all_roc_3)/20
all_roc['4']=sum(all_roc_4)/20
all_roc['5']=sum(all_roc_5)/20

print(all_roc)
```

```
{'1': 0.8515682124563039, '2': 0.7935001018120917, '3': 0.8426580488417933, '4':
0.7656214391927525, '5': 0.8032497377990563}
```

[5]:

```
all_f1 = {}
all_f1_1 = []
all_f1_2 = []
all_f1_3 = []
all_f1_4 = []
all_f1_5 = []


all_f1_1.append(sum(htru_f1['1']))
all_f1_1.append(sum(susy_f1['1']))
all_f1_1.append(sum(higgs_f1['1']))
all_f1_1.append(sum(bit_f1['1']))
all_f1_2.append(sum(htru_f1['2']))
all_f1_2.append(sum(susy_f1['2']))
all_f1_2.append(sum(higgs_f1['2']))
all_f1_2.append(sum(bit_f1['2']))
all_f1_3.append(sum(htru_f1['3']))
all_f1_3.append(sum(susy_f1['3']))
```

```
all_f1_3.append(sum(higgs_f1['3']))
all_f1_3.append(sum(bit_f1['3']))
all_f1_4.append(sum(htru_f1['4']))
all_f1_4.append(sum(susy_f1['4']))
all_f1_4.append(sum(higgs_f1['4']))
all_f1_4.append(sum(bit_f1['4']))
all_f1_5.append(sum(htru_f1['5']))
all_f1_5.append(sum(susy_f1['5']))
all_f1_5.append(sum(higgs_f1['5']))
all_f1_5.append(sum(bit_f1['5']))

all_f1['1']=sum(all_f1_1)/20
all_f1['2']=sum(all_f1_2)/20
all_f1['3']=sum(all_f1_3)/20
all_f1['4']=sum(all_f1_4)/20
all_f1['5']=sum(all_f1_5)/20

print(all_f1)
```

```
{'1': 0.7895400460686851, '2': 0.749902971430383, '3': 0.7796300800956105, '4':
0.744725733848445, '5': 0.756154889170159}
```

[6]:
```
mean_metric_all={}

mean_metric_all['1']=(all_acc['1'] + all_roc['1'] + all_f1['1'])/3
mean_metric_all['2']=(all_acc['2'] + all_roc['2'] + all_f1['2'])/3
mean_metric_all['3']=(all_acc['3'] + all_roc['3'] + all_f1['3'])/3
mean_metric_all['4']=(all_acc['4'] + all_roc['4'] + all_f1['4'])/3
mean_metric_all['5']=(all_acc['5'] + all_roc['5'] + all_f1['5'])/3

print(mean_metric_all)
```

```
{'1': 0.8121360861749963, '2': 0.7623010244141583, '3': 0.8023127096458014, '4':
0.7449573910137325, '5': 0.7696682089897383}
```

[7]:
```
# t-tests for means - NH: two sets are statistically indistinguishable, p < 0.
 ↪05 means reject the null, p > 0.05 retain null (aka *)
for i in range(len(mean_metric_all)):
    print(ttest_rel([all_acc['1'], all_roc['1'], all_f1['1']], [all_acc[str(i +␣
 ↪1)], all_roc[str(i + 1)], all_f1[str(i + 1)]]))
```

```
Ttest_relResult(statistic=nan, pvalue=nan)
Ttest_relResult(statistic=9.210753953534981, pvalue=0.011582777803603988)
Ttest_relResult(statistic=19.48656978747843, pvalue=0.002623117698521593)
Ttest_relResult(statistic=5.593889107774433, pvalue=0.030502822365633616)
Ttest_relResult(statistic=9.224451883802589, pvalue=0.011548996639156055)
```

[14]:
```python
# t-tests for accs
print(ttest_rel(all_acc_1, all_acc_2))
print(ttest_rel(all_acc_1, all_acc_3))
print(ttest_rel(all_acc_1, all_acc_4))
print(ttest_rel(all_acc_1, all_acc_5))
```

```
Ttest_relResult(statistic=2.5322228273312066, pvalue=0.08525791606921863)
Ttest_relResult(statistic=3.2860094328622647, pvalue=0.046216415404471044)
Ttest_relResult(statistic=2.330818102706705, pvalue=0.10207165574406177)
Ttest_relResult(statistic=2.1806163610255695, pvalue=0.11728121875132183)
```

[9]:
```python
# t-tests for rocs
print(ttest_rel(all_roc_1, all_roc_2))
print(ttest_rel(all_roc_1, all_roc_3))
print(ttest_rel(all_roc_1, all_roc_4))
print(ttest_rel(all_roc_1, all_roc_5))
```

```
Ttest_relResult(statistic=2.308283636461942, pvalue=0.10419456275379478)
Ttest_relResult(statistic=1.40572787759656, pvalue=0.25447193965142945)
Ttest_relResult(statistic=1.8587846005434372, pvalue=0.1600227970397546)
Ttest_relResult(statistic=1.694069135807133, pvalue=0.1888255662612093)
```

[10]:
```python
# t-tests for f1s
print(ttest_rel(all_f1_1, all_f1_2))
print(ttest_rel(all_f1_1, all_f1_3))
print(ttest_rel(all_f1_1, all_f1_4))
print(ttest_rel(all_f1_1, all_f1_5))
```

```
Ttest_relResult(statistic=3.7226356569347976, pvalue=0.03374571596010911)
Ttest_relResult(statistic=1.9667492834436537, pvalue=0.1439019185469673)
Ttest_relResult(statistic=2.5170293609360184, pvalue=0.08640192348389514)
Ttest_relResult(statistic=1.941028217112582, pvalue=0.14756312449890074)
```

SUSY Dataset

March 19, 2021

1 SUSY Dataset

See https://archive.ics.uci.edu/ml/datasets/SUSY for dataset information and feature descriptions.

[1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import csv
import numpy as np
import pandas as pd
import pandas_profiling
import matplotlib.pyplot as plt
from scipy import stats
import pickle
import operator
import glob
from scipy.io.arff import loadarff
from scipy.stats import ttest_rel

import seaborn as sns; sns.set_style('white')

from sklearn.utils import resample
from sklearn.metrics import accuracy_score, plot_confusion_matrix, f1_score,␣
↪plot_roc_curve, roc_auc_score, make_scorer
from sklearn.model_selection import KFold, GridSearchCV, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.pipeline import Pipeline
from sklearn.neural_network import MLPClassifier

from sklearn.model_selection import cross_val_score
```

[2]:

```
df = pd.read_csv('SUSY.csv', header=None)
df.columns = ['class', 'lepton_1_pT', 'lepton_1_eta', 'lepton_1_phi',␣
↪'lepton_2_pT', 'lepton_2_eta',
              'lepton_2_phi', 'missing_energy_magnitude', 'missing_energy_phi',␣
↪'MET_rel', 'axial_MET',
              'M_R', 'M_TR_2', 'R', 'MT2', 'S_R', 'M_Delta_R', 'dPhi_r_b',␣
↪'cos(theta_r1)']
df
```

[2]:

	class	lepton_1_pT	lepton_1_eta	lepton_1_phi	lepton_2_pT	\
0	0.0	0.972861	0.653855	1.176225	1.157156	
1	1.0	1.667973	0.064191	-1.225171	0.506102	
2	1.0	0.444840	-0.134298	-0.709972	0.451719	
3	1.0	0.381256	-0.976145	0.693152	0.448959	
4	1.0	1.309996	-0.690089	-0.676259	1.589283	
…	…	…	…	…	…	
4999995	1.0	0.853325	-0.961783	-1.487277	0.678190	
4999996	0.0	0.951581	0.139370	1.436884	0.880440	
4999997	0.0	0.840389	1.419162	-1.218766	1.195631	
4999998	1.0	1.784218	-0.833565	-0.560091	0.953342	
4999999	0.0	0.761500	0.680454	-1.186213	1.043521	

	lepton_2_eta	lepton_2_phi	missing_energy_magnitude	\
0	-1.739873	-0.874309	0.567765	
1	-0.338939	1.672543	3.475464	
2	-1.613871	-0.768661	1.219918	
3	0.891753	-0.677328	2.033060	
4	-0.693326	0.622907	1.087562	
…	…	…	…	
4999995	0.493580	1.647969	1.843867	
4999996	-0.351948	-0.740852	0.290863	
4999997	1.695645	0.663756	0.490888	
4999998	-0.688969	-1.428233	2.660703	
4999999	-0.316755	0.246879	1.120280	

	missing_energy_phi	MET_rel	axial_MET	M_R	M_TR_2	\
0	-0.175000	0.810061	-0.252552	1.921887	0.889637	
1	-1.219136	0.012955	3.775174	1.045977	0.568051	
2	0.504026	1.831248	-0.431385	0.526283	0.941514	
3	1.533041	3.046260	-1.005285	0.569386	1.015211	
4	-0.381742	0.589204	1.365479	1.179295	0.968218	
…	…	…	…	…	…	
4999995	0.276954	1.025105	-1.486535	0.892879	1.684429	
4999996	-0.732360	0.001360	0.257738	0.802871	0.545319	
4999997	-0.509186	0.704289	0.045744	0.825015	0.723530	
4999998	-0.861344	2.116892	2.906151	1.232334	0.952444	
4999999	0.998479	1.640881	-0.797688	0.854212	1.121858	

```
                R       MT2       S_R  M_Delta_R  dPhi_r_b  cos(theta_r1)
0        0.410772  1.145621  1.932632   0.994464  1.367815       0.040714
1        0.481928  0.000000  0.448410   0.205356  1.321893       0.377584
2        1.587535  2.024308  0.603498   1.562374  1.135454       0.180910
3        1.582217  1.551914  0.761215   1.715464  1.492257       0.090719
4        0.728563  0.000000  1.083158   0.043429  1.154854       0.094859
…             …         …         …          …         …              …
4999995  1.674084  3.366298  1.046707   2.646649  1.389226       0.364599
4999996  0.602730  0.002998  0.748959   0.401166  0.443471       0.239953
4999997  0.778236  0.752942  0.838953   0.614048  1.210595       0.026692
4999998  0.685846  0.000000  0.781874   0.676003  1.197807       0.093689
4999999  1.165438  1.498351  0.931580   1.293524  1.539167       0.187496

[5000000 rows x 19 columns]
```

[3]:
```
df['class'].value_counts()
```

[3]:
```
0.0    2712173
1.0    2287827
Name: class, dtype: int64
```

[4]:
```
# Separate majority and minority classes
df_majority = df[df['class']==0.0]
df_minority = df[df['class']==1.0]

# Downsample majority and minority class
df_majority_downsampled = resample(df_majority,
                                   replace=False,    # sample without replacement
                                   n_samples=6000,     # to match minority class
                                   random_state=123) # reproducible results

df_minority_downsampled = resample(df_minority,
                                   replace=False,    # sample without replacement
                                   n_samples=6000,     # to match majority class
                                   random_state=123) # reproducible results

# Combine minority class with downsampled majority class
df_downsampled = pd.concat([df_majority_downsampled, df_minority_downsampled])

# Display new class counts
df_downsampled['class'].value_counts()
```

[4]:
```
1.0    6000
0.0    6000
Name: class, dtype: int64
```

[5]:
```
df = df_downsampled.copy()
scaler = StandardScaler()
X, y = df.iloc[:,1:].to_numpy(), df.iloc[:,0].to_numpy()
```

1.1 Hyperparameter Search & Experimentation

[18]:
```
def experiment():
    pipeline1 = Pipeline((
    ('clf', RandomForestClassifier()),
    ))

    pipeline2 = Pipeline((
    ('clf', KNeighborsClassifier()),
    ))

    pipeline3 = Pipeline((
    ('clf', AdaBoostClassifier()),
    ))

    pipeline4 = Pipeline((
    ('clf', LogisticRegression()),
    ))

    pipeline5 = Pipeline((
    ('clf', MLPClassifier()),
    ))

    # Random Forest
    parameters1 = {
    'clf__n_estimators': [1024],
    'clf__max_features': [1, 2, 4, 6, 8, 12, 16, 20]
    }

    # KNN
    parameters2 = {
    'clf__n_neighbors':␣
 ↪[1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61,65,69,73,77,81,85,89,93,97,101,105],
    'clf__weights': ['uniform', 'distance']
    }

    # AdaBoost (Boosted Decision Tree)
    parameters3 = {
        'clf__algorithm': ['SAMME.R'],
        'clf__n_estimators': [2,4,8,16,32,64,128,256,512,1024,2048],
        'clf__learning_rate': [1e-3, 1e-2, 1e-1, 1e0, 1e1, 2e1, 5e1]
    }
```

```
    # Logistic
    parameters4 = {
    'clf__penalty':['l1', 'l2', None],
    'clf__C':[1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 0, 1e0, 1e1, 1e2,␣
↪1e3, 1e4],
    'clf__max_iter':[5000]
    }

    # Multi-layer Perceptron
    parameters5 = {
        'clf__hidden_layer_sizes':[(1,), (2,), (4,), (8,), (32,), (128,)],
        'clf__solver':['sgd'],
        'clf__activation':['relu'],
        'clf__learning_rate':['constant', 'invscaling'],
        'clf__learning_rate_init': [1e-3, 1e-2, 1e-1, 1e0],
        'clf__max_iter': [2, 4, 8, 16, 32, 64, 128, 256, 512]
    }

    pars = [parameters1, parameters2, parameters3, parameters4, parameters5]
    pips = [pipeline1, pipeline2, pipeline3, pipeline4, pipeline5]

    # List of dictionaries to hold the scores of the various metrics for each␣
↪type of classifier
    best_clf_list = []
    trial_storage = {}
    training_storage = {}

    print("starting Gridsearch")
    for i in range(len(pars)):
        trial_averages = []
        train_performance = []
        for t in range(5):
            # split and scale data
            X_train, X_test, y_train, y_test = train_test_split(X, y,␣
↪test_size=1/6, random_state=t)
            X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,␣
↪test_size=0.5, random_state=t)
            X_train = scaler.fit_transform(X_train)
            X_val = scaler.transform(X_val)
            X_test = scaler.transform(X_test)

            clf = GridSearchCV(pips[i], pars[i], refit=False, n_jobs=8, cv=5,␣
↪verbose=3, scoring=('accuracy', 'roc_auc', 'f1'))
            clf = clf.fit(X_val, y_val)

            print("finished Gridsearch trial " + str(t + 1) + " classifier " +␣
↪str(i + 1))
```

```
            print("")
            print("")

            # find the best params for each metric in a given trial
            best_index_acc = np.argmin(clf.cv_results_['rank_test_accuracy'])
            best_params_acc = clf.cv_results_['params'][best_index_acc]
            best_index_roc = np.argmin(clf.cv_results_['rank_test_roc_auc'])
            best_params_roc = clf.cv_results_['params'][best_index_roc]
            best_index_f1 = np.argmin(clf.cv_results_['rank_test_f1'])
            best_params_f1 = clf.cv_results_['params'][best_index_f1]

            # train and test models for given metric with their corresponding␣
↪best parameter settings
            pipe = pips[i]
            clf_acc = pipe.set_params(**best_params_acc)
            clf_acc = clf_acc.fit(X_train, y_train)
            clf_roc = pipe.set_params(**best_params_roc)
            clf_roc = clf_roc.fit(X_train, y_train)
            clf_f1 = pipe.set_params(**best_params_f1)
            clf_f1 = clf_f1.fit(X_train, y_train)

            # get training set performance
            train_acc = accuracy_score(y_train, clf_acc.predict(X_train))
            train_roc = roc_auc_score(y_train, clf_roc.predict_proba(X_train)[:
↪, 1])
            train_f1 = f1_score(y_train, clf_f1.predict(X_train))

            train_performance.append({
                'Model #': i + 1,
                'average': (train_f1 + train_acc + train_roc)/3,
                'accuracy': train_acc,
                'roc_auc_score': train_roc,
                'f1 score': train_f1
            })

            # get test set performances
            trial_acc = clf_acc.score(X_test, y_test)
            trial_roc = roc_auc_score(y_test, clf_roc.predict_proba(X_test)[:,␣
↪1])
            trial_f1 = f1_score(y_test, clf_f1.predict(X_test))

            # store scores and their averages in list containing averages for␣
↪each trial
            trial_averages.append({
                'Model #': i + 1, # model number corresponds to the numbers␣
↪used in pipeline above (i.e. 1 = Random Forest)
                'average':(trial_acc + trial_roc + trial_f1) / 3,
```

```
                'accuracy': trial_acc,
                'roc_auc_score': trial_roc,
                'f1_score': trial_f1
            })

            train_performance.append({
                'Model #': i + 1, # model number corresponds to the numbers␣
↪used in pipeline above (i.e. 1 = Random Forest)
                'average':(train_acc + train_roc + train_f1) / 3,
                'accuracy': train_acc,
                'roc_auc_score': train_roc,
                'f1_score': train_f1
            })

        # find the trial with the best average metric scores and append those␣
↪scores as a dict to best clf list
        max_average = 0
        for trial in trial_averages:
            if trial['average'] > max_average:
                max_average = trial['average']
                best_trial = trial

        best_clf_list.append(best_trial)
        training_storage[str(i + 1)]=train_performance
        trial_storage[str(i + 1)]=trial_averages

    return best_clf_list, trial_storage, training_storage
```

[19]:
```
%%capture --no-stdout --no-display
best_clf_list, trial_storage, training_perf = experiment()
```

```
starting Gridsearch
Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 1 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 2 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 3 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 4 classifier 1
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 5 classifier 1


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 1 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 2 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 3 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 4 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 5 classifier 2


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 1 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 2 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 3 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 4 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 5 classifier 3


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 1 classifier 4
```

```
Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 2 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 3 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 4 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 5 classifier 4


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 1 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 2 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 3 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 4 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 5 classifier 5
```

1.2 Calculating and Organizing Results

[20]:
```
print('Best Models On Average For Test Set:')
for element in best_clf_list:
    print(element)

print()

print('Train Set Data')
for i in range(len(training_perf)):
```

```python
    print(training_perf[str(i + 1)])

print()

alg_avg = {}
alg_acc = {}
alg_roc = {}
alg_f1 = {}

for i in range(len(trial_storage)):
    alg_avg[str(i + 1)]=[]
    alg_acc[str(i + 1)]=[]
    alg_roc[str(i + 1)]=[]
    alg_f1[str(i + 1)]=[]
    for entry in trial_storage[str(i + 1)]:
        alg_avg[str(i + 1)].append(entry['average'])
        alg_acc[str(i + 1)].append(entry['accuracy'])
        alg_roc[str(i + 1)].append(entry['roc_auc_score'])
        alg_f1[str(i + 1)].append(entry['f1_score'])

print('set of averages of algorithms over 5 trials:')
print(alg_avg)
print()
print('set of acc values of algorithms over 5 trials')
print(alg_acc)
print()
print('set of roc values of algorithms over 5 trials')
print(alg_roc)
print()
print('set of f1 values of algorithms over 5 trials')
print(alg_f1)
```

```
Best Models On Average For Test Set:
{'Model #': 1, 'average': 0.8155095263452585, 'accuracy': 0.7955,
'roc_auc_score': 0.8675452545254526, 'f1_score': 0.783483324510323}
{'Model #': 2, 'average': 0.7731779562405686, 'accuracy': 0.7585,
'roc_auc_score': 0.8328402840284028, 'f1_score': 0.7281935846933033}
{'Model #': 3, 'average': 0.8143371160478606, 'accuracy': 0.7915,
'roc_auc_score': 0.8665706570657066, 'f1_score': 0.7849406910778751}
{'Model #': 4, 'average': 0.8116257306890109, 'accuracy': 0.791,
'roc_auc_score': 0.871051105110511, 'f1_score': 0.7728260869565217}
{'Model #': 5, 'average': 0.8188073529165999, 'accuracy': 0.7965,
'roc_auc_score': 0.874019901990199, 'f1_score': 0.7859021567596003}

Train Set Data
[{'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1
score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score':
1.0, 'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0,
```

'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0,
'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 1,
'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model
#': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0},
{'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1
score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score':
1.0, 'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0,
'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0,
'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}]
[{'Model #': 2, 'average': 0.8119888062465478, 'accuracy': 0.7908,
'roc_auc_score': 0.8757660660059574, 'f1 score': 0.769400352733686}, {'Model #':
2, 'average': 0.8119888062465478, 'accuracy': 0.7908, 'roc_auc_score':
0.8757660660059574, 'f1_score': 0.769400352733686}, {'Model #': 2, 'average':
1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 2,
'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model
#': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0},
{'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0,
'f1_score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0,
'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 2, 'average': 1.0,
'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 2,
'average': 0.7869677586833174, 'accuracy': 0.7692, 'roc_auc_score':
0.8525531856340389, 'f1 score': 0.7391500904159133}, {'Model #': 2, 'average':
0.7869677586833174, 'accuracy': 0.7692, 'roc_auc_score': 0.8525531856340389,
'f1_score': 0.7391500904159133}]
[{'Model #': 3, 'average': 0.8309274077230716, 'accuracy': 0.8074,
'roc_auc_score': 0.8863002285937922, 'f1 score': 0.7990819945754226}, {'Model
#': 3, 'average': 0.8309274077230716, 'accuracy': 0.8074, 'roc_auc_score':
0.8863002285937922, 'f1_score': 0.7990819945754226}, {'Model #': 3, 'average':
0.8248132249523804, 'accuracy': 0.7996, 'roc_auc_score': 0.8830691012910563, 'f1
score': 0.7917705735660848}, {'Model #': 3, 'average': 0.8248132249523804,
'accuracy': 0.7996, 'roc_auc_score': 0.8830691012910563, 'f1_score':
0.7917705735660848}, {'Model #': 3, 'average': 0.8418045411090489, 'accuracy':
0.8174, 'roc_auc_score': 0.8996995355673312, 'f1 score': 0.8083140877598152},
{'Model #': 3, 'average': 0.8418045411090488, 'accuracy': 0.8174,
'roc_auc_score': 0.8996995355673312, 'f1_score': 0.8083140877598152}, {'Model
#': 3, 'average': 0.8311218595332234, 'accuracy': 0.808, 'roc_auc_score':
0.8898819825195678, 'f1 score': 0.7954835960801023}, {'Model #': 3, 'average':
0.8311218595332234, 'accuracy': 0.808, 'roc_auc_score': 0.8898819825195678,
'f1_score': 0.7954835960801023}, {'Model #': 3, 'average': 0.8335477262592926,
'accuracy': 0.8086, 'roc_auc_score': 0.8910450502688323, 'f1 score':
0.8009981285090456}, {'Model #': 3, 'average': 0.8335477262592926, 'accuracy':
0.8086, 'roc_auc_score': 0.8910450502688323, 'f1_score': 0.8009981285090456}]
[{'Model #': 4, 'average': 0.8067827633249033, 'accuracy': 0.7878,
'roc_auc_score': 0.8614587538366301, 'f1 score': 0.7710895361380797}, {'Model
#': 4, 'average': 0.8067827633249033, 'accuracy': 0.7878, 'roc_auc_score':
0.8614587538366301, 'f1_score': 0.7710895361380797}, {'Model #': 4, 'average':
0.8036118350354551, 'accuracy': 0.7832, 'roc_auc_score': 0.860153737624598, 'f1
score': 0.7674817674817674}, {'Model #': 4, 'average': 0.8036118350354552,

'accuracy': 0.7832, 'roc_auc_score': 0.860153737624598, 'f1_score':
0.7674817674817674}, {'Model #': 4, 'average': 0.8142920919095308, 'accuracy':
0.7954, 'roc_auc_score': 0.8677130895068489, 'f1 score': 0.7797631862217437},
{'Model #': 4, 'average': 0.8142920919095308, 'accuracy': 0.7954,
'roc_auc_score': 0.8677130895068489, 'f1_score': 0.7797631862217437}, {'Model
#': 4, 'average': 0.8080586788953648, 'accuracy': 0.7906, 'roc_auc_score':
0.8644470399937016, 'f1 score': 0.7691289966923925}, {'Model #': 4, 'average':
0.8080586788953648, 'accuracy': 0.7906, 'roc_auc_score': 0.8644470399937016,
'f1_score': 0.7691289966923925}, {'Model #': 4, 'average': 0.8036892496357865,
'accuracy': 0.7854, 'roc_auc_score': 0.8576677489073593, 'f1 score':
0.7680000000000001}, {'Model #': 4, 'average': 0.8036892496357865, 'accuracy':
0.7854, 'roc_auc_score': 0.8576677489073593, 'f1_score': 0.7680000000000001}]
[{'Model #': 5, 'average': 0.8160180985920991, 'accuracy': 0.7932,
'roc_auc_score': 0.8726302014207792, 'f1 score': 0.782224094355518}, {'Model #':
5, 'average': 0.8160180985920991, 'accuracy': 0.7932, 'roc_auc_score':
0.8726302014207792, 'f1_score': 0.782224094355518}, {'Model #': 5, 'average':
0.8222340800148724, 'accuracy': 0.7992, 'roc_auc_score': 0.8771055803368928, 'f1
score': 0.7903966597077244}, {'Model #': 5, 'average': 0.8222340800148724,
'accuracy': 0.7992, 'roc_auc_score': 0.8771055803368928, 'f1_score':
0.7903966597077244}, {'Model #': 5, 'average': 0.8322117669504517, 'accuracy':
0.8104, 'roc_auc_score': 0.8844787136493476, 'f1 score': 0.8017565872020075},
{'Model #': 5, 'average': 0.8322117669504517, 'accuracy': 0.8104,
'roc_auc_score': 0.8844787136493476, 'f1_score': 0.8017565872020075}, {'Model
#': 5, 'average': 0.8225718797427476, 'accuracy': 0.8024, 'roc_auc_score':
0.8817924139872262, 'f1 score': 0.7835232252410167}, {'Model #': 5, 'average':
0.8225718797427476, 'accuracy': 0.8024, 'roc_auc_score': 0.8817924139872262,
'f1_score': 0.7835232252410167}, {'Model #': 5, 'average': 0.8216220211479742,
'accuracy': 0.7998, 'roc_auc_score': 0.8758472805422596, 'f1 score':
0.7892187829016635}, {'Model #': 5, 'average': 0.8216220211479742, 'accuracy':
0.7998, 'roc_auc_score': 0.8758472805422596, 'f1_score': 0.7892187829016635}]

set of averages of algorithms over 5 trials:
{'1': [0.8103054881429115, 0.784287133641621, 0.8024050452130066,
0.8026583075053352, 0.8155095263452585], '2': [0.768930451968841,
0.7580980510560553, 0.7569913040162296, 0.7662832609514352, 0.7731779562405686],
'3': [0.8108460373522145, 0.7897186963264669, 0.7995211610388241,
0.8120673513656537, 0.8143371160478606], '4': [0.8020464180679553,
0.7870126883896598, 0.7915167217206504, 0.8016420003454564, 0.8116257306890109],
'5': [0.8158317077355797, 0.7900813572056341, 0.8066910825288552,
0.817776543699833, 0.8188073529165999]}

set of acc values of algorithms over 5 trials
{'1': [0.792, 0.771, 0.7865, 0.7855, 0.7955], '2': [0.7555, 0.7505, 0.7485,
0.7575, 0.7585], '3': [0.788, 0.771, 0.779, 0.7945, 0.7915], '4': [0.782,
0.7735, 0.775, 0.7825, 0.791], '5': [0.795, 0.7695, 0.7875, 0.802, 0.7965]}

set of roc values of algorithms over 5 trials
{'1': [0.8553576715046967, 0.8360234431002238, 0.8509038148843027,

```
0.8539913823109329, 0.8675452545254526], '2': [0.8160450104815093,
0.8134052965227804, 0.8217005628517823, 0.8281623198146837, 0.8328402840284028],
'3': [0.8613070282120626, 0.8457236565469683, 0.8582243902439024,
0.8625510600077615, 0.8665706570657066], '4': [0.8531308508425213,
0.8401872564684776, 0.8512280175109443, 0.8634232309532668, 0.871051105110511],
'5': [0.8644909867020851, 0.8489939369911618, 0.8609450906816759,
0.8677230737224496, 0.874019901990199]}

set of f1 values of algorithms over 5 trials
{'1': [0.7835587929240375, 0.7458379578246392, 0.769811320754717,
0.7684835402050728, 0.783483324510323], '2': [0.7352463454250134,
0.7103888566453859, 0.7007733491969066, 0.7131874630396215, 0.7281935846933033],
'3': [0.7832310838445808, 0.7524324324324324, 0.7613390928725702,
0.7791509940891993, 0.7849406910778751], '4': [0.7710084033613446,
0.7473508087005019, 0.7483221476510067, 0.7590027700831025, 0.7728260869565217],
'5': [0.7880041365046535, 0.7517501346257404, 0.7716281569048898,
0.7836065573770491, 0.7859021567596003]}
```

[21]:
```python
# calculate average acc metric scores per algorithm over 5 trials
alg_acc_averages = {}
for i in range(len(alg_acc)):
    alg_acc_averages[str(i + 1)] = sum(alg_acc[str(i + 1)])/5

print(alg_acc_averages)
```

```
{'1': 0.7861, '2': 0.7540999999999999, '3': 0.7848, '4': 0.7807999999999999,
'5': 0.7901}
```

[22]:
```python
# calculate average roc metric scores per algorithm over 5 trials
alg_roc_averages = {}
for i in range(len(alg_roc)):
    alg_roc_averages[str(i + 1)] = sum(alg_roc[str(i + 1)])/5

print(alg_roc_averages)
```

```
{'1': 0.8527643132651217, '2': 0.8224306947398317, '3': 0.8588753584152802, '4':
0.8558040921771444, '5': 0.8632345980175143}
```

[23]:
```python
# calculate average f1 metric scores per algorithm over 5 trials
alg_f1_averages = {}
for i in range(len(alg_f1)):
    alg_f1_averages[str(i + 1)] = sum(alg_f1[str(i + 1)])/5

print(alg_f1_averages)
```

```
{'1': 0.7702349872437579, '2': 0.717557919800046, '3': 0.7722188588633315, '4':
0.7597020433504955, '5': 0.7761782284343866}
```

[26]:
```
averages = {}
for i in range(len(alg_acc_averages)):
    averages[str(i + 1)] = (alg_acc_averages[str(i + 1)] +␣
 ↪alg_roc_averages[str(i + 1)] + alg_f1_averages[str(i + 1)])/3

print(averages)
```

```
{'1': 0.8030331001696265, '2': 0.7646962048466258, '3': 0.8052980724262039, '4':
0.7987687118425466, '5': 0.8098376088173002}
```

[25]:
```
# t-test best against rest mean of metrics (ANN against rest)
combined_metrics_1 = []
combined_metrics_2 = []
combined_metrics_3 = []
combined_metrics_4 = []
combined_metrics_5 = []

for item in alg_acc['1']:
    combined_metrics_1.append(item)
for item in alg_roc['1']:
    combined_metrics_1.append(item)
for item in alg_f1['1']:
    combined_metrics_1.append(item)

for item in alg_acc['2']:
    combined_metrics_2.append(item)
for item in alg_roc['2']:
    combined_metrics_2.append(item)
for item in alg_f1['2']:
    combined_metrics_2.append(item)

for item in alg_acc['3']:
    combined_metrics_3.append(item)
for item in alg_roc['3']:
    combined_metrics_3.append(item)
for item in alg_f1['3']:
    combined_metrics_3.append(item)

for item in alg_acc['4']:
    combined_metrics_4.append(item)
for item in alg_roc['4']:
    combined_metrics_4.append(item)
for item in alg_f1['4']:
    combined_metrics_4.append(item)

for item in alg_acc['5']:
    combined_metrics_5.append(item)
```

```
for item in alg_roc['5']:
    combined_metrics_5.append(item)
for item in alg_f1['5']:
    combined_metrics_5.append(item)

print(ttest_rel(combined_metrics_5, combined_metrics_1))
print(ttest_rel(combined_metrics_5, combined_metrics_2))
print(ttest_rel(combined_metrics_5, combined_metrics_3))
print(ttest_rel(combined_metrics_5, combined_metrics_4))
```

```
Ttest_relResult(statistic=4.5653650438780184, pvalue=0.0004405925267108117)
Ttest_relResult(statistic=13.035113141942922, pvalue=3.2114263925488175e-09)
Ttest_relResult(statistic=5.266886492081862, pvalue=0.00011912737913248329)
Ttest_relResult(statistic=5.438233464673745, pvalue=8.738872162370697e-05)
```

HTRU2 Dataset

March 19, 2021

1 HTRU2 Pulsar Dataset

See https://archive.ics.uci.edu/ml/datasets/HTRU2 for dataset information and feature descriptions.

[1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import csv
import numpy as np
import pandas as pd
import pandas_profiling
import matplotlib.pyplot as plt
from scipy import stats
import pickle
import operator
import glob
from scipy.io.arff import loadarff
from scipy.stats import ttest_rel

import seaborn as sns; sns.set_style('white')

from sklearn.utils import resample
from sklearn.metrics import accuracy_score, plot_confusion_matrix, f1_score,␣
↪plot_roc_curve, roc_auc_score, make_scorer
from sklearn.model_selection import KFold, GridSearchCV, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.pipeline import Pipeline
from sklearn.neural_network import MLPClassifier

from sklearn.model_selection import cross_val_score
```

[2]:
```
raw_data = loadarff('HTRU2/HTRU_2.arff')
df = pd.DataFrame(raw_data[0])
df['class'] = np.where(df['class']==b'1', 1, 0)
df['class'].value_counts()
```

[2]:
```
0    16259
1     1639
Name: class, dtype: int64
```

[3]:
```
# Separate majority and minority classes
df_majority = df[df['class']==0]
df_minority = df[df['class']==1]

# Downsample majority and minority class
df_majority_downsampled = resample(df_majority,
                                   replace=False,    # sample without replacement
                                   n_samples=6000,     # to match minority class
                                   random_state=123) # reproducible results

df_minority_downsampled = resample(df_minority,
                                   replace=True,    # sample with replacement
                                   n_samples=6000,     # to match majority class
                                   random_state=123) # reproducible results

# Combine minority class with downsampled majority class
df_downsampled = pd.concat([df_majority_downsampled, df_minority_downsampled])

# Display new class counts
df_downsampled['class'].value_counts()
```

[3]:
```
1    6000
0    6000
Name: class, dtype: int64
```

[4]:
```
df = df_downsampled.copy()
scaler = StandardScaler()
X, y = df.iloc[:,0:8].to_numpy(), df.iloc[:,8].to_numpy()
```

1.1 Hyperparameter Search & Experimentation

[5]:
```
def experiment():
    pipeline1 = Pipeline((
    ('clf', RandomForestClassifier()),
    ))

    pipeline2 = Pipeline((
    ('clf', KNeighborsClassifier()),
```

```
))

pipeline3 = Pipeline([
('clf', AdaBoostClassifier()),
])

pipeline4 = Pipeline([
('clf', LogisticRegression()),
])

pipeline5 = Pipeline([
('clf', MLPClassifier()),
])

# Random Forest
parameters1 = {
'clf__n_estimators': [1024],
'clf__max_features': [1, 2, 4, 6, 8, 12, 16, 20]
}

# KNN
parameters2 = {
'clf__n_neighbors':␣
↪[1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61,65,69,73,77,81,85,89,93,97,101,105],
'clf__weights': ['uniform', 'distance']
}

# AdaBoost (Boosted Decision Tree)
parameters3 = {
    'clf__algorithm': ['SAMME.R'],
    'clf__n_estimators': [2,4,8,16,32,64,128,256,512,1024,2048],
    'clf__learning_rate': [1e-3, 1e-2, 1e-1, 1e0, 1e1, 2e1, 5e1]
}

# Logistic
parameters4 = {
'clf__penalty':['l1', 'l2', None],
'clf__C':[1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 0, 1e0, 1e1, 1e2,␣
↪1e3, 1e4],
'clf__max_iter':[5000]
}

# Multi-layer Perceptron
parameters5 = {
    'clf__hidden_layer_sizes':[(1,), (2,), (4,), (8,), (32,), (128,)],
    'clf__solver':['sgd'],
    'clf__activation':['relu'],
```

```
        'clf__learning_rate':['constant', 'invscaling'],
        'clf__learning_rate_init': [1e-3, 1e-2, 1e-1, 1e0],
        'clf__max_iter': [2, 4, 8, 16, 32, 64, 128, 256, 512]
    }

    pars = [parameters1, parameters2, parameters3, parameters4, parameters5]
    pips = [pipeline1, pipeline2, pipeline3, pipeline4, pipeline5]

    # List of dictionaries to hold the scores of the various metrics for each␣
↪type of classifier
    best_clf_list = []
    trial_storage = {}
    training_storage = {}

    print("starting Gridsearch")
    for i in range(len(pars)):
        trial_averages = []
        train_performance = []
        for t in range(5):
            # split and scale data
            X_train, X_test, y_train, y_test = train_test_split(X, y,␣
↪test_size=1/6, random_state=t)
            X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,␣
↪test_size=0.5, random_state=t)
            X_train = scaler.fit_transform(X_train)
            X_val = scaler.transform(X_val)
            X_test = scaler.transform(X_test)

            clf = GridSearchCV(pips[i], pars[i], refit=False, n_jobs=8, cv=5,␣
↪verbose=3, scoring=('accuracy', 'roc_auc', 'f1'))
            clf = clf.fit(X_val, y_val)

            print("finished Gridsearch trial " + str(t + 1) + " classifier " +␣
↪str(i + 1))
            print("")
            print("")

            # find the best params for each metric in a given trial
            best_index_acc = np.argmin(clf.cv_results_['rank_test_accuracy'])
            best_params_acc = clf.cv_results_['params'][best_index_acc]
            best_index_roc = np.argmin(clf.cv_results_['rank_test_roc_auc'])
            best_params_roc = clf.cv_results_['params'][best_index_roc]
            best_index_f1 = np.argmin(clf.cv_results_['rank_test_f1'])
            best_params_f1 = clf.cv_results_['params'][best_index_f1]

            # train and test models for given metric with their corresponding␣
↪best parameter settings
```

```
            pipe = pips[i]
            clf_acc = pipe.set_params(**best_params_acc)
            clf_acc = clf_acc.fit(X_train, y_train)
            clf_roc = pipe.set_params(**best_params_roc)
            clf_roc = clf_roc.fit(X_train, y_train)
            clf_f1 = pipe.set_params(**best_params_f1)
            clf_f1 = clf_f1.fit(X_train, y_train)

            # get training set performance
            train_acc = accuracy_score(y_train, clf_acc.predict(X_train))
            train_roc = roc_auc_score(y_train, clf_roc.predict_proba(X_train)[:
↪, 1])
            train_f1 = f1_score(y_train, clf_f1.predict(X_train))

            train_performance.append({
                'Model #': i + 1,
                'average': (train_f1 + train_acc + train_roc)/3,
                'accuracy': train_acc,
                'roc_auc_score': train_roc,
                'f1 score': train_f1
            })

            # get test set performances
            trial_acc = clf_acc.score(X_test, y_test)
            trial_roc = roc_auc_score(y_test, clf_roc.predict_proba(X_test)[:,␣
↪1])
            trial_f1 = f1_score(y_test, clf_f1.predict(X_test))

            # store scores and their averages in list containing averages for␣
↪each trial
            trial_averages.append({
                'Model #': i + 1, # model number corresponds to the numbers␣
↪used in pipeline above (i.e. 1 = Random Forest)
                'average':(trial_acc + trial_roc + trial_f1) / 3,
                'accuracy': trial_acc,
                'roc_auc_score': trial_roc,
                'f1_score': trial_f1
            })

            train_performance.append({
                'Model #': i + 1, # model number corresponds to the numbers␣
↪used in pipeline above (i.e. 1 = Random Forest)
                'average':(train_acc + train_roc + train_f1) / 3,
                'accuracy': train_acc,
                'roc_auc_score': train_roc,
                'f1_score': train_f1
            })
```

```
        # find the trial with the best average metric scores and append those␣
↪scores as a dict to best clf list
        max_average = 0
        for trial in trial_averages:
            if trial['average'] > max_average:
                max_average = trial['average']
                best_trial = trial

        best_clf_list.append(best_trial)
        training_storage[str(i + 1)]=train_performance
        trial_storage[str(i + 1)]=trial_averages

    return best_clf_list, trial_storage, training_storage
```

[6]:
```
%%capture --no-stdout --no-display
best_clf_list, trial_storage, training_perf = experiment()
```

```
starting Gridsearch
Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 1 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 2 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 3 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 4 classifier 1


Fitting 5 folds for each of 8 candidates, totalling 40 fits
finished Gridsearch trial 5 classifier 1


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 1 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 2 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
```

```
finished Gridsearch trial 3 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 4 classifier 2


Fitting 5 folds for each of 54 candidates, totalling 270 fits
finished Gridsearch trial 5 classifier 2


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 1 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 2 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 3 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 4 classifier 3


Fitting 5 folds for each of 77 candidates, totalling 385 fits
finished Gridsearch trial 5 classifier 3


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 1 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 2 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 3 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
finished Gridsearch trial 4 classifier 4


Fitting 5 folds for each of 42 candidates, totalling 210 fits
```

```
finished Gridsearch trial 5 classifier 4


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 1 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 2 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 3 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 4 classifier 5


Fitting 5 folds for each of 432 candidates, totalling 2160 fits
finished Gridsearch trial 5 classifier 5
```

1.2 Calculating and Organizing Results

[7]:
```
print('Best Models On Average For Test Set:')
for element in best_clf_list:
    print(element)

print()

print('Train Set Data')
for i in range(len(training_perf)):
    print(training_perf[str(i + 1)])

print()

alg_avg = {}
alg_acc = {}
alg_roc = {}
alg_f1 = {}

for i in range(len(trial_storage)):
    alg_avg[str(i + 1)]=[]
    alg_acc[str(i + 1)]=[]
    alg_roc[str(i + 1)]=[]
```

```
    alg_f1[str(i + 1)]=[]
    for entry in trial_storage[str(i + 1)]:
        alg_avg[str(i + 1)].append(entry['average'])
        alg_acc[str(i + 1)].append(entry['accuracy'])
        alg_roc[str(i + 1)].append(entry['roc_auc_score'])
        alg_f1[str(i + 1)].append(entry['f1_score'])

print('set of averages of algorithms over 5 trials:')
print(alg_avg)
print()
print('set of acc values of algorithms over 5 trials')
print(alg_acc)
print()
print('set of roc values of algorithms over 5 trials')
print(alg_roc)
print()
print('set of f1 values of algorithms over 5 trials')
print(alg_f1)
```

```
Best Models On Average For Test Set:
{'Model #': 1, 'average': 0.985648124035749, 'accuracy': 0.9815,
'roc_auc_score': 0.9944090056285179, 'f1_score': 0.9810353664787289}
{'Model #': 2, 'average': 0.9805553504954202, 'accuracy': 0.976,
'roc_auc_score': 0.9901558474046278, 'f1_score': 0.9755102040816327}
{'Model #': 3, 'average': 0.9770331745120471, 'accuracy': 0.9695,
'roc_auc_score': 0.9928655409631019, 'f1_score': 0.9687339825730394}
{'Model #': 4, 'average': 0.960472762341343, 'accuracy': 0.95, 'roc_auc_score':
0.9819744043142212, 'f1_score': 0.949443882709808}
{'Model #': 5, 'average': 0.9657670237746814, 'accuracy': 0.9565,
'roc_auc_score': 0.9846297866892832, 'f1_score': 0.9561712846347608}

Train Set Data
[{'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1
score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score':
1.0, 'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0,
'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0,
'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 1,
'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 0.9999999999999999, 'f1
score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score':
0.9999999999999999, 'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy':
1.0, 'roc_auc_score': 0.9999999999999999, 'f1 score': 1.0}, {'Model #': 1,
'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 0.9999999999999999,
'f1_score': 1.0}, {'Model #': 1, 'average': 1.0, 'accuracy': 1.0,
'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 1, 'average': 1.0,
'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}]
[{'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1
score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score':
1.0, 'f1_score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0,
```

'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 2, 'average': 1.0,
'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}, {'Model #': 2,
'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model
#': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0},
{'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1
score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0, 'roc_auc_score':
1.0, 'f1_score': 1.0}, {'Model #': 2, 'average': 1.0, 'accuracy': 1.0,
'roc_auc_score': 1.0, 'f1 score': 1.0}, {'Model #': 2, 'average': 1.0,
'accuracy': 1.0, 'roc_auc_score': 1.0, 'f1_score': 1.0}]
[{'Model #': 3, 'average': 0.9978536525424969, 'accuracy': 0.9968,
'roc_auc_score': 0.9999583996738535, 'f1 score': 0.9968025579536371}, {'Model
#': 3, 'average': 0.9978536525424969, 'accuracy': 0.9968, 'roc_auc_score':
0.9999583996738535, 'f1_score': 0.9968025579536371}, {'Model #': 3, 'average':
0.9975866125627301, 'accuracy': 0.9964, 'roc_auc_score': 0.9999627199940352, 'f1
score': 0.9963971176941553}, {'Model #': 3, 'average': 0.9975866125627301,
'accuracy': 0.9964, 'roc_auc_score': 0.9999627199940352, 'f1_score':
0.9963971176941553}, {'Model #': 3, 'average': 0.9979891596250382, 'accuracy':
0.997, 'roc_auc_score': 0.9999692799557632, 'f1 score': 0.9969981989193516},
{'Model #': 3, 'average': 0.9979891596250382, 'accuracy': 0.997,
'roc_auc_score': 0.9999692799557632, 'f1_score': 0.9969981989193516}, {'Model
#': 3, 'average': 0.9975678262181606, 'accuracy': 0.9964, 'roc_auc_score':
0.9999531136909783, 'f1 score': 0.9963503649635037}, {'Model #': 3, 'average':
0.9975678262181606, 'accuracy': 0.9964, 'roc_auc_score': 0.9999531136909783,
'f1_score': 0.9963503649635037}, {'Model #': 3, 'average': 0.995702767845874,
'accuracy': 0.9936, 'roc_auc_score': 0.9999185599478784, 'f1 score':
0.9935897435897436}, {'Model #': 3, 'average': 0.995702767845874, 'accuracy':
0.9936, 'roc_auc_score': 0.9999185599478784, 'f1_score': 0.9935897435897436}]
[{'Model #': 4, 'average': 0.9534485904479814, 'accuracy': 0.9428,
'roc_auc_score': 0.9764177351150433, 'f1 score': 0.9411280362289007}, {'Model
#': 4, 'average': 0.9534485904479814, 'accuracy': 0.9428, 'roc_auc_score':
0.9764177351150433, 'f1_score': 0.9411280362289007}, {'Model #': 4, 'average':
0.953048929015495, 'accuracy': 0.9428, 'roc_auc_score': 0.9756091160974585, 'f1
score': 0.940737670949026}, {'Model #': 4, 'average': 0.9530489290154948,
'accuracy': 0.9428, 'roc_auc_score': 0.9756091160974585, 'f1_score':
0.940737670949026}, {'Model #': 4, 'average': 0.9554394460119063, 'accuracy':
0.9454, 'roc_auc_score': 0.9772650872617256, 'f1 score': 0.9436532507739938},
{'Model #': 4, 'average': 0.9554394460119066, 'accuracy': 0.9454,
'roc_auc_score': 0.9772650872617256, 'f1_score': 0.9436532507739938}, {'Model
#': 4, 'average': 0.9528958494569627, 'accuracy': 0.9418, 'roc_auc_score':
0.9779322305609444, 'f1 score': 0.9389553178099435}, {'Model #': 4, 'average':
0.9528958494569627, 'accuracy': 0.9418, 'roc_auc_score': 0.9779322305609444,
'f1_score': 0.9389553178099435}, {'Model #': 4, 'average': 0.9522701525724996,
'accuracy': 0.942, 'roc_auc_score': 0.9749265439529881, 'f1 score':
0.9398839137645107}, {'Model #': 4, 'average': 0.9522701525724996, 'accuracy':
0.942, 'roc_auc_score': 0.9749265439529881, 'f1_score': 0.9398839137645107}]
[{'Model #': 5, 'average': 0.9591559820342345, 'accuracy': 0.9486,
'roc_auc_score': 0.9816942564829708, 'f1 score': 0.9471736896197328}, {'Model
#': 5, 'average': 0.9591559820342345, 'accuracy': 0.9486, 'roc_auc_score':

0.9816942564829708, 'f1_score': 0.9471736896197328}, {'Model #': 5, 'average':
0.9577066357573972, 'accuracy': 0.9466, 'roc_auc_score': 0.9814468770315004, 'f1
score': 0.9450730302406912}, {'Model #': 5, 'average': 0.9577066357573972,
'accuracy': 0.9466, 'roc_auc_score': 0.9814468770315004, 'f1_score':
0.9450730302406912}, {'Model #': 5, 'average': 0.9623856962903033, 'accuracy':
0.9516, 'roc_auc_score': 0.9848041381179589, 'f1 score': 0.9507529507529509},
{'Model #': 5, 'average': 0.9623856962903033, 'accuracy': 0.9516,
'roc_auc_score': 0.9848041381179589, 'f1_score': 0.9507529507529509}, {'Model
#': 5, 'average': 0.960378713120653, 'accuracy': 0.9482, 'roc_auc_score':
0.9861298136277217, 'f1 score': 0.9468063257342371}, {'Model #': 5, 'average':
0.9603787131206528, 'accuracy': 0.9482, 'roc_auc_score': 0.9861298136277217,
'f1_score': 0.9468063257342371}, {'Model #': 5, 'average': 0.9557201629625229,
'accuracy': 0.9428, 'roc_auc_score': 0.9827516689610681, 'f1 score':
0.9416088199265007}, {'Model #': 5, 'average': 0.9557201629625229, 'accuracy':
0.9428, 'roc_auc_score': 0.9827516689610681, 'f1_score': 0.9416088199265007}]

set of averages of algorithms over 5 trials:
{'1': [0.984837386263412, 0.983515278023449, 0.985648124035749,
0.9824066698220754, 0.9827570359987039], '2': [0.9796630605923798,
0.9769882343551988, 0.9805553504954202, 0.974886854446145, 0.9748550021501564],
'3': [0.9757174427339321, 0.9718470281412918, 0.9770331745120471,
0.9745478760807735, 0.9743968466116134], '4': [0.960472762341343,
0.9565565471331917, 0.950397834139426, 0.9539998668273991, 0.9570629911475995],
'5': [0.9657670237746814, 0.9589732947290605, 0.9573872507852422,
0.9570526344734825, 0.9596680077282943]}

set of acc values of algorithms over 5 trials
{'1': [0.979, 0.9785, 0.9815, 0.9765, 0.977], '2': [0.9725, 0.97, 0.976, 0.968,
0.9665], '3': [0.968, 0.963, 0.9695, 0.966, 0.9655], '4': [0.95, 0.947, 0.9405,
0.9445, 0.946], '5': [0.9565, 0.9475, 0.947, 0.9455, 0.9475]}

set of roc values of algorithms over 5 trials
{'1': [0.9961814501288185, 0.9945001421643419, 0.9944090056285179,
0.994662953938972, 0.9944099409940994], '2': [0.9933840473028117,
0.991702407983629, 0.9901558474046278, 0.9891478729831047, 0.9916826682668267],
'3': [0.9907416668000194, 0.9910426452979646, 0.9928655409631019,
0.992091449924185, 0.9924512451245125], '4': [0.9819744043142212,
0.9793541333781851, 0.9739577235772358, 0.975645226464387, 0.980974097409741],
'5': [0.9846297866892832, 0.9851228019590648, 0.9808340212632896,
0.981872446999612, 0.9847224722472248]}

set of f1 values of algorithms over 5 trials
{'1': [0.9793307086614172, 0.9775456919060052, 0.9810353664787289,
0.9760570555272542, 0.9768611670020121], '2': [0.9731051344743277,
0.9692622950819672, 0.9755102040816327, 0.9675126903553299, 0.9663823381836427],
'3': [0.968410661401777, 0.9614984391259105, 0.9687339825730394,
0.9655521783181358, 0.9652392947103275], '4': [0.949443882709808,
0.9433155080213904, 0.9367357788410421, 0.9418543740178104, 0.9442148760330578],

'5': [0.9561712846347608, 0.9442970822281167, 0.9443277310924371, 0.9437854564208354, 0.9467815509376584]}

[8]:
```
# calculate average acc metric scores per algorithm over 5 trials
alg_acc_averages = {}
for i in range(len(alg_acc)):
    alg_acc_averages[str(i + 1)] = sum(alg_acc[str(i + 1)])/5

print(alg_acc_averages)
```

{'1': 0.9785, '2': 0.9705999999999999, '3': 0.9664000000000001, '4': 0.9456, '5': 0.9488}

[9]:
```
# calculate average roc metric scores per algorithm over 5 trials
alg_roc_averages = {}
for i in range(len(alg_roc)):
    alg_roc_averages[str(i + 1)] = sum(alg_roc[str(i + 1)])/5

print(alg_roc_averages)
```

{'1': 0.9948326985709499, '2': 0.9912145687881999, '3': 0.9918385096219566, '4': 0.9783811170287539, '5': 0.983436305831695}

[10]:
```
# calculate average f1 metric scores per algorithm over 5 trials
alg_f1_averages = {}
for i in range(len(alg_f1)):
    alg_f1_averages[str(i + 1)] = sum(alg_f1[str(i + 1)])/5

print(alg_f1_averages)
```

{'1': 0.9781659979150834, '2': 0.9703545324353801, '3': 0.9658869112258381, '4': 0.9431128839246217, '5': 0.9470726210627618}

[11]:
```
# calculate average of all 3 metric scores for each algorithm
averages = {}
for i in range(len(alg_acc_averages)):
    averages[str(i + 1)] = (alg_acc_averages[str(i + 1)] +␣
↪alg_roc_averages[str(i + 1)] + alg_f1_averages[str(i + 1)])/3

print(averages)
```

{'1': 0.9838328988286777, '2': 0.9773897004078599, '3': 0.9747084736159316, '4': 0.9556980003177918, '5': 0.9597696422981522}

[12]:
```
# t-test best against rest mean of metrics (RF against rest)
combined_metrics_1 = []
combined_metrics_2 = []
combined_metrics_3 = []
combined_metrics_4 = []
```

```
combined_metrics_5 = []

for item in alg_acc['1']:
    combined_metrics_1.append(item)
for item in alg_roc['1']:
    combined_metrics_1.append(item)
for item in alg_f1['1']:
    combined_metrics_1.append(item)

for item in alg_acc['2']:
    combined_metrics_2.append(item)
for item in alg_roc['2']:
    combined_metrics_2.append(item)
for item in alg_f1['2']:
    combined_metrics_2.append(item)

for item in alg_acc['3']:
    combined_metrics_3.append(item)
for item in alg_roc['3']:
    combined_metrics_3.append(item)
for item in alg_f1['3']:
    combined_metrics_3.append(item)

for item in alg_acc['4']:
    combined_metrics_4.append(item)
for item in alg_roc['4']:
    combined_metrics_4.append(item)
for item in alg_f1['4']:
    combined_metrics_4.append(item)

for item in alg_acc['5']:
    combined_metrics_5.append(item)
for item in alg_roc['5']:
    combined_metrics_5.append(item)
for item in alg_f1['5']:
    combined_metrics_5.append(item)

print(ttest_rel(combined_metrics_1, combined_metrics_2))
print(ttest_rel(combined_metrics_1, combined_metrics_3))
print(ttest_rel(combined_metrics_1, combined_metrics_4))
print(ttest_rel(combined_metrics_1, combined_metrics_5))
```

```
Ttest_relResult(statistic=9.488391242469374, pvalue=1.783103793622407e-07)
Ttest_relResult(statistic=7.314255242662729, pvalue=3.82457095450185e-06)
Ttest_relResult(statistic=11.394423748363014, pvalue=1.812873909233341e-08)
Ttest_relResult(statistic=9.31379364409074, pvalue=2.2366945324128233e-07)
```