Sahil Shah

COGS 118A: Final Project

ABSTRACT

In this project for COGS 118A, we are replicating part of the study done by Caruana & Niculescu-Mizil in the early 2000's where we examine some supervised machine learning algorithms and test their training and testing accuracies under certain conditions. In the part of the study we are replicating, we will be examining the performance of three machine learning algorithms: Support Vector Machines, K-Nearest Neighbors, and Random Forest Classifiers. The main focus of this study is to examine each of their performance by testing certain hyperparameters and determining how long it took for each to run.

INTRODUCTION

In Caruana and Niculescu-Mizil's study, they analyzed and tested the performance of ten supervised machine learning algorithms: Support Vector Machines, Random Forest Classifiers, Decision Trees, Bagged Decision Trees, Logistic Regression Classifiers, K-Nearest Neighbors, Boosted Trees, Artificial Neural Networks, Boosted Stumps, and Naive Bayes. We only tested three out of these eleven algorithms. For their performance metrics, they had utilized F-score, accuracy, lift, and many more. Since the datasets we had chosen were relatively balanced, we only utilized the accuracy metric since it was a good scorer of our balanced datasets.

Our performances weren't that surprising. Overall, the KNN and RF algorithms were pretty consistent in their average test scores. The support vector machine didn't perform as well for a certain dataset but for the rest, it performed around similarly to what the paper had scored.

METHOD

For our analyses, we used Python's machine learning library, Sci-Kit Learn to code and develop our machine learning models.

As we start with the methodology, this part will explain what hyperparameters we used for our three algorithms. It was relatively similar to what Caruana & Niculescu-Mizil had done in their analyses.

1. SVM

For our SVM, we used the following kernels in Sklearn: linear, polynomial with the second and third degrees, and radial with width {0.001,0.005,0.01,0.05,0.1,0.5,1,2}. We tried to vary the regularization parameter, or C, from 10^−7 to 10^3 with each kernel. However, whenever we tried to test C from 10^1 to 10^3, the computer ended up hanging for an extremely long period of time, so we just tested from 10^-7 to 10^-1 for each of the kernels. For this algorithm, we would scale the training data and use that training scaler on the testing data.

2. **K-Nearest Neighbors**
   For our K-Nearest Neighbor hyperparameters, we pick 25 K values in a linear space from 1 to 500. Our program randomly picks 25 K values because if we were to use the paper's method of using K from 1 to 26, the model could end up overfitting extremely heavily. We also used Euclidean distance(minkowski) as a metric. We tried to use weighted minkowski(weighted Euclidean distance) as a metric, however, Sklearn removed its compatibility so we couldn't test weighted distance. And finally, we used uniform and distance weighting. For this algorithm, we would scale the training data and use that training scaler on the testing data.

3. **Random Forest Classifier**
   For our final algorithm, we used 1024 trees(n_estimators = 1024) and set the size of the feature set (max_features) varying from [1,2,4,6,8,12,16, 20]. For this algorithm, we don't need to scale the data. In the original study, they had tested the RF classifier twice, one with transformed nominal data and one with the original. I could not train the classifier on the original untransformed data since SKlearn does not support compatibility. "The decision trees implemented in scikit-learn use only numerical features and these features are interpreted always as continuous numeric variables." Therefore, I only trained the classifier on the transformed one-hot-encoded data.

PERFORMANCE METRIC

The paper had used eight different performance metrics, however we just decided to use accuracy as a performance metric since our data was relatively balanced. We didn't need to use lift, F-Score, or any other metric.

DATASETS

We used three different datasets: ADULT, COVERTYPE, AND LETTER-RECOGNITION, all taken from UCI's machine learning repository. The ADULT dataset was filled with a lot of text data so we had to convert into numerical data by one-hot-encoding. We converted the income column into the class label 1 or 0. Greater than 50K was labeled as 1; less than 50K was labeled as 0. The COVERTYPE dataset didn't have any text data so we simply took the largest class and labeled that as 1. The rest of the classes were labeled as 0. By class, we mean what forest cover each data point was designated as. There were 7 classes, and class 2 had the most data points designated. However, since the COVERTYPE dataset was too large and my computer memory

wasn't enough, I had to cut down the dataset to 30000 rows instead of 580000. I made sure to preserve class ratios so there wouldn't be any imbalances. Lastly, the LETTER RECOGNITION DATASET was balanced due to labeling letters "A-M" as 1 and the rest as 0.

PROCEDURE

Our procedure in running each algorithm was the same. We would run 3 trials for each algorithm and dataset. To start, 5000 random data points were chosen to be the training data. We would utilize 5-fold cross validation and use gridsearch on this training data to find the best hyperparameters out of our paramgrid. After finding the best combination of hyperparameters, we would run our model with the best_estimators on our training data to get the training accuracy. Then we would run our model on the rest of the testing dataset and find the test accuracy. We would do this 3 times (for each trial) and find the average test accuracy across the 3 trials for each algorithm and dataset pair. So, in total, there are 9 mean-valued test accuracies (1 for each algorithm and dataset) in a table and 3 mean-valued test accuracies(average algorithm performance across the three datasets) in a table.

EXPERIMENT

1. **BOLDED-** Bolded scores show the best performance.
2. Scores that are have * are not statistically significantly different in comparison to the best performance.
3. We are using a p-value of 0.05 to determine significance.

Table 1

		AVG TEST		
MODEL	ADULT	COV_TYPE	LETTER	**MEAN**
SVM	**0.82649**	0.7605	0.8991	0.8287
kNN	0.8228*	0.7776	**0.95257**	0.85098*
RF	0.82383	**0.8208267**	0.9453	**0.86332**

Table 1 shows the average test scores for each algorithm and each dataset pair. For the adult dataset, SVM performed the best with an average test score of 0.82649. SVM and RF were statistically significantly different and KNN was not. For the covertype dataset, RF performed the best with an average test score of 0.8208. Both SVM and kNN were statistically significantly different in comparison to RF's score. We can say that for that dataset, RF's performance was statistically significantly better than the other two algorithms. Finally for the letter dataset, kNN performed the best with an average score of 0.95257. It was statistically significantly greater than both SVM and kNN, so we can say that kNN performed the best for the letter dataset.

Table 2

MODEL	ACC
SVM	0.8287
kNN	0.85098*
RF	**0.863317**

Table 2 shows that RF performed the highest on average out of the 3 algorithms with an average accuracy of 0.863317. RF's results were statistically significantly better than SVM but not different as compared to kNN.

Table 3

RAW TEST		ADULT			COVTYPE			LETTER P.2	
	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3
SVM	0.82759	0.82548	0.8264	0.76	0.7628	0.7587	0.8923	0.919	0.886
kNN	0.82413	0.82417	0.82	0.779	0.7756	0.7782	0.9505	0.9535	0.9537
RF	0.824	0.823	0.8245	0.8218	0.8146	0.8261	0.9449	0.94559	0.94539

Table 3 shows the raw test scores for each trial in each algorithm/dataset combo.

Table 4

		AVG TRAIN		
MODEL	ADULT	COV_TYPE	LETTER P.2	**MEAN**
SVM	0.8346	0.7691333	0.958	0.8539
kNN	1	1	1	1
RF	1	1	1	1

Table 4 shows the average training score for each algorithm and dataset combo. SVM was similar in it's training and test set performance, however, kNN and RF both had average training set accuracies of 1, so they both heavily overfitted the training set. We tried so many ways of stratifying the data, randomly splitting the data in different ways, but the training error always came to 1.

Table 5

Values that are *'d have a p-value greater than 0.05 and are not statistically significant.

p-values	ADULT	COVTYPE	LETTER
SVM vs. kNN	0.1276*	0.0155	0.0324
SVM vs. RF	0.0327	0.00566	0.0437
KNN vs. RF	0.603*	0.0035	0.01322

Table 6

p-values	All DATASETS
SVM vs. kNN	0.0361
SVM vs. RF	0.0088
KNN vs. RF	0.1556*

For finding the p-values, we used the ttest_rel method from the scipy.stats package library. It took in arrays as inputs and those arrays would be the test accuracies for each algorithm. Outputs would be the t-test score and the p-value.

CONCLUSION

In conclusion, SVM produced the best accuracy for the ADULT dataset, RF produced the best accuracy for COVERTYPE, and kNN produced the best accuracy for LETTER. All of these accuracies were statistically significant and so we can say this. Overall, kNN and RF were the best classifiers out of all the datasets and combos. They took significantly less time to run as compared to the SVM classifier and were more computationally efficient in my opinion due to this. Overall, I really enjoyed this challenging project and learned a lot.

CODE

ADULT DATASET CODE

```
import numpy as np
from sklearn import preprocessing
from sklearn.model_selection import GridSearchCV
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier, KernelDensity
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils import shuffle
import pandas as pd
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

```
#Loading Data into Dataframe
df1 = pd.read_csv('adult.data')
```

Adult Dataset

In [3]:

```
#Drop Rows with Missing Values
#Renaming Columns for Legibility
df1.replace({' ?': np.NaN}, inplace=True)
df1 = df1.dropna()
adultdf1 = df1
adultdf1 = adultdf1.rename(columns={' State-gov': 'State_gov'})
adultdf1 = adultdf1.rename(columns={' Bachelors': 'Bachelors'})
adultdf1 = adultdf1.rename(columns={' Never-married': 'Marital_status'})
adultdf1 = adultdf1.rename(columns={' Adm-clerical': 'Adm_clerical'})
adultdf1 = adultdf1.rename(columns={' Not-in-family': 'Family_status'})
adultdf1 = adultdf1.rename(columns={' 2174': 'Capital_gain'})
adultdf1 = adultdf1.rename(columns={' 0': 'Capital_loss'})
adultdf1 = adultdf1.rename(columns={' 40': 'Hours/week'})
adultdf1 = adultdf1.rename(columns={' United-States': 'Country'})
adultdf1 = adultdf1.rename(columns={' <=50K': 'Income'})

#First, label encoding the state-gov column and then converting into one-hot-encoded columns
labelencoder = LabelEncoder()
adultdf1['State_gov'] = labelencoder.fit_transform(adultdf1['State_gov'])
enc = OneHotEncoder(handle_unknown='ignore')
enc_df = pd.DataFrame(enc.fit_transform(adultdf1[['State_gov']]).toarray())

#Converted education year column into numerical column using .map.
#Since was ordinal data, did not one-hot-encode.
year_ord_map = {' Preschool': 1, ' 1st-4th': 2, ' 5th-6th': 3,
               ' 7th-8th': 4, ' 9th': 5, ' 10th': 6, ' 11th': 7, ' 12th': 8, ' HS-grad': 9, ' Some-college': 10, ' Assoc-voc': 1
                ' Assoc-acdm' : 12, ' Bachelors': 13, ' Prof-school': 14, ' Masters': 15, ' Doctorate': 16}
adultdf1['Bachelors'] = adultdf1['Bachelors'].map(year_ord_map)
```

```
#One-hot-encoded marital status; first put encoded data into seperate dataframe.
adultdf1['Marital_status'] = labelencoder.fit_transform(adultdf1['Marital_status'])
enc2 = OneHotEncoder(handle_unknown='ignore')
enc2_df = pd.DataFrame(enc2.fit_transform(adultdf1[['Marital_status']]).toarray())

#One-hot-encoded job; first put encoded data into seperate dataframe.
adultdf1['Adm_clerical'] = labelencoder.fit_transform(adultdf1['Adm_clerical'])
enc3 = OneHotEncoder(handle_unknown='ignore')
enc3_df = pd.DataFrame(enc3.fit_transform(adultdf1[['Adm_clerical']]).toarray())

#One-hot-encoded family status; first put encoded data into seperate dataframe.
adultdf1['Family_status'] = labelencoder.fit_transform(adultdf1['Family_status'])
enc4 = OneHotEncoder(handle_unknown='ignore')
enc4_df = pd.DataFrame(enc4.fit_transform(adultdf1[['Family_status']]).toarray())

#One-hot-encoded race; first put encoded data into seperate dataframe.
adultdf1[' White'] = labelencoder.fit_transform(adultdf1[' White'])
enc5 = OneHotEncoder(handle_unknown='ignore')
enc5_df = pd.DataFrame(enc5.fit_transform(adultdf1[[' White']]).toarray())

#One-hot-encoded Gender; first put encoded data into seperate dataframe.
adultdf1[' Male'] = labelencoder.fit_transform(adultdf1[' Male'])
enc6 = OneHotEncoder(handle_unknown='ignore')
enc6_df = pd.DataFrame(enc6.fit_transform(adultdf1[[' Male']]).toarray())

#one-hot-encoded Country; first put encoded data into seperate dataframe.
adultdf1['Country'] = labelencoder.fit_transform(adultdf1['Country'])
enc7 = OneHotEncoder(handle_unknown='ignore')
enc7_df = pd.DataFrame(enc7.fit_transform(adultdf1[['Country']]).toarray())

#Dropped all the numerical columns
adultdf1 = adultdf1.drop(['State_gov'], axis = 1)
adultdf1 = adultdf1.drop(['Marital_status'], axis = 1)
adultdf1 = adultdf1.drop(['Adm_clerical'], axis = 1)
adultdf1 = adultdf1.drop(['Family_status'], axis = 1)
adultdf1 = adultdf1.drop([' White'], axis = 1)
adultdf1 = adultdf1.drop([' Male'], axis = 1)
adultdf1 = adultdf1.drop(['Country'], axis = 1)
```

```
#Dropped the columns which had no significance to the data. If we kept this data, it would skew some data points.
adultdf1 = adultdf1.drop(columns=[' 13'])
adultdf1 = adultdf1.drop(columns=['Capital_gain'])
adultdf1 = adultdf1.drop(columns=['Capital_loss'])

#Combined all the one-hot-encoded data into one list so easier to join later on.
dfs = [enc_df, enc2_df, enc3_df, enc4_df, enc5_df, enc6_df, enc7_df]

#Changed income to class label 1 and 0 using .map method.
income_ord_map = {' <=50K': 0, ' >50K': 1}
adultdf1['Income'] = adultdf1['Income'].map(income_ord_map)

#Combined original dataframe with one-hot-encoded data. Now all data is numerical and stored in adultdf1.
dfss = pd.concat(dfs, axis = 1)
adultdf1 = adultdf1.reset_index(drop=True)
adultdf1 = dfss.join(adultdf1)
adultdf1 = adultdf1.dropna()

#Converted dataframe into array.
adult_arr = adultdf1.to_numpy()
```

In [4]:
```
#Accuracy function. Compares predicted labels vs actual labels and counts accuracy.
def prediction(predictions, Y_given):
    wrong = 0
    counter = 0
    for test, train in zip(predictions, Y_given):
        if test == train:
            wrong = wrong
        else:
            wrong = wrong + 1
        counter = counter + 1
    accuracy = 1 - (wrong/counter)
    return accuracy
```

SVM

In [5]:
```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
SVM_train_accuracy = []
SVM_test_accuracy = []
SVM_best_params = []

#Each trial
for i in range(3):
    data_svm = shuffle(adult_arr)
#Data Splitting, train-test-split
    X_svm = data_svm[:, 0:-1]
    Y_svm = data_svm[:, -1]
    X_train_svm, X_test_svm, Y_train_svm, Y_test_svm = train_test_split(X_svm, Y_svm, test_size = 25161/30161, random_state=42,
                                                                        stratify = Y_svm)
#Scaling training data using StandardScaler
    scaler_svm = preprocessing.StandardScaler().fit(X_train_svm)
    X_train_svm = scaler_svm.transform(X_train_svm)
#Param-grid
    parameters = [{'kernel': ['rbf'], 'gamma': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 2.0], 'C': [10**-7, 10**-6, 10**-5,
                                                                                                      10**-4, 10**-3, 10**-2,
                                                                                                      10**-1]},
                  {'kernel': ['poly'], 'degree': [2, 3], 'C': [10**-7, 10**-6, 10**-5, 10**-4, 10**-3, 10**-2, 10**-1]},
                  {'kernel': ['linear'], 'C': [10**-7, 10**-6, 10**-5, 10**-4, 10**-3, 10**-2, 10**-1]}]
#SVC class, grid search, and fit training data and training labels
    svc = svm.SVC(gamma = 'auto')
    grid_search_svm = GridSearchCV(svc, parameters, cv = 5, error_score = np.nan)
    grid_search2_svm = grid_search_svm.fit(X_train_svm, Y_train_svm)
#Printing and Storing best params into list.
    best_params_svm = grid_search2_svm.best_params_
    print("Best params: {}".format(best_params_svm))
    SVM_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_svm))
#Best estimator is refitted so just used .predict on training data to find predicted training values.
#Used prediction pre-defined function to count accuracy on predicted training set.
    train_predictions_svm = grid_search2_svm.best_estimator_.predict(X_train_svm) #.best_estimator_
    train_accuracy_svm = prediction(train_predictions_svm, Y_train_svm)
```

```
#Printing and storing training accuracy into list.
    print("Train accuracy: {}".format(train_accuracy_svm))
    SVM_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_svm))
#Scaling testing data with my training data scaler. This ensures training and testing data are scaled the same.
#Printing and storing test accuracy into list.
    X_test_svm = scaler_svm.transform(X_test_svm)
    test_predictions_svm = grid_search2_svm.best_estimator_.predict(X_test_svm) #.best_estimator_
    test_accuracy_svm = prediction(test_predictions_svm, Y_test_svm)
    print("Test accuracy: {}".format(test_accuracy_svm))
    SVM_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_svm))
```

```
Best params: {'C': 0.1, 'kernel': 'linear'}
Train accuracy: 0.8344
Test accuracy: 0.8275903183498271
Best params: {'C': 0.1, 'kernel': 'linear'}
Train accuracy: 0.8314
Test accuracy: 0.8254838837884027
Best params: {'C': 0.1, 'kernel': 'linear'}
Train accuracy: 0.838
Test accuracy: 0.826437740948293
```

kNN

In [6]:
```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
kNN_train_accuracy = []
kNN_test_accuracy = []
kNN_best_params = []

#Setting K parameters
p = (np.linspace(1,500,25))
p = p.astype('int64')

#For Each Trial
for i in range(3):
    data_knn = shuffle(adult_arr)
#Data Splitting, Train-test-split
    X_knn = data_knn[:, 0:-1]
    Y_knn = data_knn[:, -1]
    X_train_knn, X_test_knn, Y_train_knn, Y_test_knn = train_test_split(X_knn, Y_knn, test_size = 25161/30161, random_state=42,
                                                                        stratify = Y_knn)
#Scaling training data using StandardScaler
    scaler_knn = preprocessing.StandardScaler().fit(X_train_knn)
    X_train_knn = scaler_knn.transform(X_train_knn)
#Param-grid, initializing KNN Class, and fitting with gridsearch
#Printing and storing best params in list
    params = [{'weights' : ['uniform', 'distance'], 'metric' : ['minkowski'],'n_neighbors': p}]
    neighbor = KNeighborsClassifier()
    grid_search_knn = GridSearchCV(neighbor, params, cv=5, error_score = np.nan)
    grid_search_knn2 = grid_search_knn.fit(X_train_knn, Y_train_knn)
    best_params_knn = grid_search_knn2.best_params_
    print("Best params: {}".format(best_params_knn))
    kNN_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_knn))
#Predicting training data with best_estimator
#Calculating training accuracy with defined prediction function
#Printing and storing training accuracy in list.
    train_predictions_knn = grid_search_knn2.best_estimator_.predict(X_train_knn)
    train_accuracy_knn = prediction(train_predictions_knn, Y_train_knn)
    print("Train accuracy: {}".format(train_accuracy_knn))
    kNN_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_knn))
#Scaling testing data with my training data scaler. This ensures training and testing data are scaled the same.
#Printing and storing test accuracy into list.
    X_test_knn = scaler_knn.transform(X_test_knn)
    test_predictions_knn = grid_search_knn2.best_estimator_.predict(X_test_knn)
    test_accuracy_knn = prediction(test_predictions_knn, Y_test_knn)
    print("Test accuracy: {}".format(test_accuracy_knn))
    kNN_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_knn))
```

```
Best params: {'metric': 'minkowski', 'n_neighbors': 416, 'weights': 'distance'}
Train accuracy: 1.0
Test accuracy: 0.8241325861452248
Best params: {'metric': 'minkowski', 'n_neighbors': 458, 'weights': 'distance'}
Train accuracy: 1.0
Test accuracy: 0.8241723301935535
Best params: {'metric': 'minkowski', 'n_neighbors': 312, 'weights': 'distance'}
Train accuracy: 1.0
Test accuracy: 0.8200786932156909
```

Random Forest-Transformed Data

In [9]:
```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
rf_train_accuracy = []
rf_test_accuracy = []
rf_best_params = []

#For each trial
for i in range(3):
    data_rf = shuffle(adult_arr)
#Data splitting, train-test-split
    X_rf = data_rf[:, 0:-1]
    Y_rf = data_rf[:, -1]
    X_train_rf, X_test_rf, Y_train_rf, Y_test_rf = train_test_split(X_rf, Y_rf, test_size = 25161/30161, random_state=42)
#Param-grid, initialize random forest class, and fitting training data with grid search.
    params = [{'n_estimators' : [1024], 'max_features' : [1, 2, 4, 6, 8, 12, 16, 20]}]
    forest = RandomForestClassifier()
    grid_search_rf = GridSearchCV(forest, params, cv=5, error_score = np.nan)
    grid_search_rf2 = grid_search_rf.fit(X_train_rf, Y_train_rf)
#Printing and storing best params into list.
    best_params_rf = grid_search_rf2.best_params_
    print("Best params: {}".format(best_params_rf))
    rf_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_rf))
#Using best_estimator to predict training data.
#Storing and printing training accuracy into list.
#Using defined prediction function to calculate total accuracy.
    train_predictions_rf = grid_search_rf2.best_estimator_.predict(X_train_rf)
    train_accuracy_rf = prediction(train_predictions_rf, Y_train_rf)
    print("Train accuracy: {}".format(train_accuracy_rf))
    rf_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_rf))
#Predicting Test data with best_estimator.predict.
#Using prediction function to calculate accuracy.
#Printing and storing Test accuracy into list.
    test_predictions_rf = grid_search_rf2.best_estimator_.predict(X_test_rf)
    test_accuracy_rf = prediction(test_predictions_rf, Y_test_rf)
    print("Test accuracy: {}".format(test_accuracy_rf))
```

```
    rf_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_rf))
```

```
Best params: {'max_features': 16, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.8247684909184849
Best params: {'max_features': 8, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.8231787289853345
Best params: {'max_features': 20, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.8244902825801836
```

COVERTYPE DATASET CODE

In [1]:
```
import numpy as np
from sklearn import preprocessing
from sklearn.model_selection import GridSearchCV
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier, KernelDensity
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils import shuffle
import pandas as pd
```

In [2]:
```
#Importing Data into Pandas Dataframe
cov_df2 = pd.read_csv('covtype.data.gz')
```

COVERTYPE DATASET ¶

In [3]:
```
#Renaming Column
#Creating Seperate Data Columns
cov_df2 = cov_df2.rename(columns={'5': 'Forest_cov_desig.'})
class_2 = cov_df2[cov_df2['Forest_cov_desig.'] == 2]
class_1 = cov_df2[cov_df2['Forest_cov_desig.'] == 1]
class_3 = cov_df2[cov_df2['Forest_cov_desig.'] == 3]
class_7 = cov_df2[cov_df2['Forest_cov_desig.'] == 7]
class_6 = cov_df2[cov_df2['Forest_cov_desig.'] == 6]
class_5 = cov_df2[cov_df2['Forest_cov_desig.'] == 5]
class_4 = cov_df2[cov_df2['Forest_cov_desig.'] == 4]

#Since the Dataset was way too big and my computer couldn't handle it, I decided to make
#it the same 30000 row set like the paper. I kept the same percentage of each class and
# cut each class down by the same ratio. The ratio g is below.
# This code picks the number of random samples and cuts down dataset by the ratio.
g = 30000/581011
class_2 = class_2.sample(frac=g, replace = False).reset_index(drop=True)
class_1 = class_1.sample(frac=g, replace = False).reset_index(drop=True)
class_3 = class_3.sample(frac=g, replace = False).reset_index(drop=True)
class_7 = class_7.sample(frac=g, replace = False).reset_index(drop=True)
class_6 = class_6.sample(frac=g, replace = False).reset_index(drop=True)
class_5 = class_5.sample(frac=g, replace = False).reset_index(drop=True)
class_4 = class_4.sample(frac=g, replace = False).reset_index(drop=True)
frames = [class_2, class_1, class_3, class_7, class_6, class_5, class_4]

#Combined data columns into a dataframe.
cov_df2 = pd.concat(frames)
#Changed class to 1 and 0 using .map to have largest class as 1 and the rest as 0.
class_map = {1: 0, 2: 1, 3: 0,
             4: 0, 5: 0, 6: 0, 7: 0}
cov_df2['Class'] = cov_df2['Forest_cov_desig.'].map(class_map)
cov_df2 = cov_df2.drop(['Forest_cov_desig.'], axis = 1)
#Shuffling data and converting into numpy array.
cov_df2 = shuffle(cov_df2)
cov_arr = cov_df2.to_numpy()
```

In [4]:
```
#Accuracy function. Compares predicted labels vs actual labels and counts accuracy.
def prediction(predictions, Y_given):
    wrong = 0
    counter = 0
    for test, train in zip(predictions, Y_given):
        if test == train:
            wrong = wrong
        else:
            wrong = wrong + 1
        counter = counter + 1
    accuracy = 1 - (wrong/counter)
    return accuracy
```

SVM

In [5]:
```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
SVM_train_accuracy = []
SVM_test_accuracy = []
SVM_best_params = []

#For each Trial
for i in range(3):
    data_svm = shuffle(cov_arr)
#Data splitting, train-test-split.
    X_svm = data_svm[:, 0:-1]
    Y_svm = data_svm[:, -1]
    X_train_svm, X_test_svm, Y_train_svm, Y_test_svm = train_test_split(X_svm, Y_svm, test_size = 5/6, random_state=42)
#Scaling training data using StandardScaler
    scaler_svm = preprocessing.StandardScaler().fit(X_train_svm)
    X_train_svm = scaler_svm.transform(X_train_svm)
#Param-grid, setting SVM class,grid search parameters, and fitting training data.
    parameters = [{'kernel': ['rbf'], 'gamma': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 2.0], 'C': [10**-7, 10**-6, 10**-5,
                                                                                                      10**-4, 10**-3, 10**-2,
                                                                                                      10**-1]},
                  {'kernel': ['poly'], 'degree': [2, 3], 'C': [10**-7, 10**-6, 10**-5, 10**-4, 10**-3, 10**-2, 10**-1]},
                  {'kernel': ['linear'], 'C': [10**-7, 10**-6, 10**-5, 10**-4, 10**-3, 10**-2, 10**-1]}]
    svc = svm.SVC(gamma = 'auto')
    grid_search_svm = GridSearchCV(svc, parameters, cv = 5, error_score = np.nan)
    grid_search2_svm = grid_search_svm.fit(X_train_svm, Y_train_svm)
    best_params_svm = grid_search2_svm.best_params_
#Printing and storing best params into list.
    print("Best params: {}".format(best_params_svm))
    SVM_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_svm))
#Using best_estimator to predict training data labels.
#Using prediction function to calculate training accuracy.
#Printing and storing training accuracies into list.
    train_predictions_svm = grid_search2_svm.best_estimator_.predict(X_train_svm) #.best_estimator_
    train_accuracy_svm = prediction(train_predictions_svm, Y_train_svm)
    print("Train accuracy: {}".format(train_accuracy_svm))
    SVM_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_svm))
```

```
#Scaling testing data with my training data scaler. This ensures training and testing data are scaled the same.
#Using prediction function to calculate testing accuracy.
#Printing and storing test accuracy into list.
    X_test_svm = scaler_svm.transform(X_test_svm)
    test_predictions_svm = grid_search2_svm.best_estimator_.predict(X_test_svm) #.best_estimator_
    test_accuracy_svm = prediction(test_predictions_svm, Y_test_svm)
    print("Test accuracy: {}".format(test_accuracy_svm))
    SVM_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_svm))
```

```
Best params: {'C': 0.1, 'kernel': 'linear'}
Train accuracy: 0.764
Test accuracy: 0.76076
Best params: {'C': 0.1, 'kernel': 'linear'}
Train accuracy: 0.7668
Test accuracy: 0.7628
Best params: {'C': 0.1, 'kernel': 'linear'}
Train accuracy: 0.7766
Test accuracy: 0.7587200000000001
```

kNN

In [6]:
```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
kNN_train_accuracy = []
kNN_test_accuracy = []
kNN_best_params = []

#Setting K parameters
p = (np.linspace(1,500,25))
p = p.astype('int64')

#For each trial
for i in range(3):
    data_knn = shuffle(cov_arr)
#Data splitting, train-test-split
    X_knn = data_knn[:, 0:-1]
    Y_knn = data_knn[:, -1]
    X_train_knn, X_test_knn, Y_train_knn, Y_test_knn = train_test_split(X_knn, Y_knn, test_size = 5/6, random_state=42)
# Scaling training data using StandardScaler()
    scaler_knn = preprocessing.StandardScaler().fit(X_train_knn)
    X_train_knn = scaler_knn.transform(X_train_knn)
#Param-grid, initializing kNN class, grid search parameters and fitting training data.
# Printing and storing best parameters in list.
    params = [{'weights' : ['uniform', 'distance'], 'metric' : ['minkowski'],'n_neighbors': p}]
    neighbor = KNeighborsClassifier()
    grid_search_knn = GridSearchCV(neighbor, params, cv=5, error_score = np.nan)
    grid_search_knn2 = grid_search_knn.fit(X_train_knn, Y_train_knn)
    best_params_knn = grid_search_knn2.best_params_
    print("Best params: {}".format(best_params_knn))
    kNN_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_knn))
#Using best_estimator to predict training data labels.
#Using prediction function to calculate training accuracy
#Printing and storing training accuracy into list.
    train_predictions_knn = grid_search_knn2.best_estimator_.predict(X_train_knn)
    train_accuracy_knn = prediction(train_predictions_knn, Y_train_knn)
    print("Train accuracy: {}".format(train_accuracy_knn))
    kNN_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_knn))
#Scaling testing data with my training data scaler. This ensures training and testing data are scaled the same.
#Using prediction function to calculate testing accuracy.
#Printing and storing test accuracy into list.
    X_test_knn = scaler_knn.transform(X_test_knn)
    test_predictions_knn = grid_search_knn2.best_estimator_.predict(X_test_knn)
    test_accuracy_knn = prediction(test_predictions_knn, Y_test_knn)
    print("Test accuracy: {}".format(test_accuracy_knn))
    kNN_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_knn))
```

```
Best params: {'metric': 'minkowski', 'n_neighbors': 1, 'weights': 'uniform'}
Train accuracy: 1.0
Test accuracy: 0.77932
Best params: {'metric': 'minkowski', 'n_neighbors': 1, 'weights': 'uniform'}
Train accuracy: 1.0
Test accuracy: 0.77568
Best params: {'metric': 'minkowski', 'n_neighbors': 1, 'weights': 'uniform'}
Train accuracy: 1.0
Test accuracy: 0.7782
```

Random Forest ¶

In [7]:
```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
rf_train_accuracy = []
rf_test_accuracy = []
rf_best_params = []

#For each trial
for i in range(3):
    data_rf = shuffle(cov_arr)
#Data splitting, train-test-split
    X_rf = data_rf[:, 0:-1]
    Y_rf = data_rf[:, -1]
    X_train_rf, X_test_rf, Y_train_rf, Y_test_rf = train_test_split(X_rf, Y_rf, test_size = 5/6, random_state=42)
#Param-grid, initialize random forest class, and fitting training data with grid search.
    params = [{'n_estimators' : [1024], 'max_features' : [1, 2, 4, 6, 8, 12, 16, 20]}]
    forest = RandomForestClassifier()
    grid_search_rf = GridSearchCV(forest, params, cv=5, error_score = np.nan)
    grid_search_rf2 = grid_search_rf.fit(X_train_rf, Y_train_rf)
#Printing and storing best params into list.
    best_params_rf = grid_search_rf2.best_params_
    print("Best params: {}".format(best_params_rf))
    rf_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_rf))
#Using best_estimator to predict training data.
#Storing and printing training accuracy into list.
#Using defined prediction function to calculate total accuracy.
    train_predictions_rf = grid_search_rf2.best_estimator_.predict(X_train_rf)
    train_accuracy_rf = prediction(train_predictions_rf, Y_train_rf)
    print("Train accuracy: {}".format(train_accuracy_rf))
    rf_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_rf))
#Predicting Test data with best_estimator.predict.
#Using prediction function to calculate accuracy.
#Printing and storing Test accuracy into list.
    test_predictions_rf = grid_search_rf2.best_estimator_.predict(X_test_rf)
    test_accuracy_rf = prediction(test_predictions_rf, Y_test_rf)
    print("Test accuracy: {}".format(test_accuracy_rf))
    rf_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_rf))
```

```
Best params: {'max_features': 6, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.8218
Best params: {'max_features': 2, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.81464
Best params: {'max_features': 12, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.82608
```

LETTER P.2 DATASET CODE

In [1]:
```
import numpy as np
from sklearn import preprocessing
from sklearn.model_selection import GridSearchCV
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier, KernelDensity
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils import shuffle
import pandas as pd
```

In [2]:
```
#Load dataset into Pandas Dataframe
df3 = pd.read_csv('letter-recognition.data')
```

Letter Dataset P.2

In [3]:
```
#Rename Column
letter_df = df3
letter_df = letter_df.rename(columns={'T': 'Letter'})
#Initialize Column
letter_df['Class'] = 0
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M']
#For each row in the dataframe, if the alphabet letter in that row is not in the list above,
# set the class of that data point to 0. Otherwise, it's 1.
for index, row in letter_df.iterrows():
    if row['Letter'] in letters:
        letter_df.at[index, 'Class'] = 1
letter_df = letter_df.drop(['Letter'], axis = 1)
#Converting dataframe into array.
letter_arr = letter_df.to_numpy()
```

In [4]:
```
#Accuracy function. Compares predicted labels vs actual labels and counts accuracy.
def prediction(predictions, Y_given):
    wrong = 0
    counter = 0
    for test, train in zip(predictions, Y_given):
        if test == train:
            wrong = wrong
        else:
            wrong = wrong + 1
        counter = counter + 1
    accuracy = 1 - (wrong/counter)
    return accuracy
```

SVM

In [5]:
```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
SVM_train_accuracy = []
SVM_test_accuracy = []
SVM_best_params = []

#For each Trial
for i in range(3):
    data_svm = shuffle(letter_arr)
#Data splitting, train-test-split.
    X_svm = data_svm[:, 0:-1]
    Y_svm = data_svm[:, -1]
    X_train_svm, X_test_svm, Y_train_svm, Y_test_svm = train_test_split(X_svm, Y_svm, test_size = 14999/19999, random_state=42)
#Scaling training data using StandardScaler
    scaler_svm = preprocessing.StandardScaler().fit(X_train_svm)
    X_train_svm = scaler_svm.transform(X_train_svm)
#Param-grid, setting SVM class,grid search parameters, and fitting training data.
    parameters = [{'kernel': ['rbf'], 'gamma': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 2.0], 'C': [10**-7, 10**-6, 10**-5,
                                                                                                      10**-4, 10**-3, 10**-2,
                                                                                                      10**-1]},
                  {'kernel': ['poly'], 'degree': [2, 3], 'C': [10**-7, 10**-6, 10**-5, 10**-4, 10**-3, 10**-2, 10**-1]},
                  {'kernel': ['linear'], 'C': [10**-7, 10**-6, 10**-5, 10**-4, 10**-3, 10**-2, 10**-1]}]
    svc = svm.SVC(gamma = 'auto')
    grid_search_svm = GridSearchCV(svc, parameters, cv = 5, error_score = np.nan)
    grid_search2_svm = grid_search_svm.fit(X_train_svm, Y_train_svm)
    best_params_svm = grid_search2_svm.best_params_
#Printing and storing best params into list.
    print("Best params: {}".format(best_params_svm))
    SVM_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_svm))
#Using best_estimator to predict training data labels.
#Using prediction function to calculate training accuracy.
#Printing and storing training accuracies into list.
    train_predictions_svm = grid_search2_svm.best_estimator_.predict(X_train_svm) #.best_estimator_
    train_accuracy_svm = prediction(train_predictions_svm, Y_train_svm)
    print("Train accuracy: {}".format(train_accuracy_svm))
```

```
    SVM_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_svm))
#Scaling testing data with my training data scaler. This ensures training and testing data are scaled the same.
#Using prediction function to calculate testing accuracy.
#Printing and storing test accuracy into list.
    X_test_svm = scaler_svm.transform(X_test_svm)
    test_predictions_svm = grid_search2_svm.best_estimator_.predict(X_test_svm) #.best_estimator_
    test_accuracy_svm = prediction(test_predictions_svm, Y_test_svm)
    print("Test accuracy: {}".format(test_accuracy_svm))
    SVM_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_svm))
```

```
Best params: {'C': 0.1, 'gamma': 0.5, 'kernel': 'rbf'}
Train accuracy: 0.9566
Test accuracy: 0.8923261550770052
Best params: {'C': 0.1, 'gamma': 0.5, 'kernel': 'rbf'}
Train accuracy: 0.9672
Test accuracy: 0.9193279551970132
Best params: {'C': 0.1, 'gamma': 0.5, 'kernel': 'rbf'}
Train accuracy: 0.9502
Test accuracy: 0.8862590839389293
```

kNN

In [6]:

```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
kNN_train_accuracy = []
kNN_test_accuracy = []
kNN_best_params = []

#Setting K parameters
p = (np.linspace(1,500,25))
p = p.astype('int64')

#For each trial
for i in range(3):
    data_knn = shuffle(letter_arr)
#Data splitting, train-test-split
    X_knn = data_knn[:, 0:-1]
    Y_knn = data_knn[:, -1]
    X_train_knn, X_test_knn, Y_train_knn, Y_test_knn = train_test_split(X_knn, Y_knn, test_size = 14999/19999, random_state=42)
# Scaling training data using StandardScaler()
    scaler_knn = preprocessing.StandardScaler().fit(X_train_knn)
    X_train_knn = scaler_knn.transform(X_train_knn)
#Param-grid, initializing kNN class, grid search parameters and fitting training data.
# Printing and storing best parameters in list.
    params = [{'weights' : ['uniform', 'distance'], 'metric' : ['minkowski'],'n_neighbors': p}]
    neighbor = KNeighborsClassifier()
    grid_search_knn = GridSearchCV(neighbor, params, cv=5, error_score = np.nan)
    grid_search_knn2 = grid_search_knn.fit(X_train_knn, Y_train_knn)
    best_params_knn = grid_search_knn2.best_params_
    print("Best params: {}".format(best_params_knn))
    kNN_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_knn))
#Using best_estimator to predict training data labels.
#Using prediction function to calculate training accuracy
#Printing and storing training accuracy into list.
    train_predictions_knn = grid_search_knn2.best_estimator_.predict(X_train_knn)
    train_accuracy_knn = prediction(train_predictions_knn, Y_train_knn)
    print("Train accuracy: {}".format(train_accuracy_knn))
```

```
    kNN_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_knn))
#Scaling testing data with my training data scaler. This ensures training and testing data are scaled the sam
#Using prediction function to calculate testing accuracy.
#Printing and storing test accuracy into list.
    X_test_knn = scaler_knn.transform(X_test_knn)
    test_predictions_knn = grid_search_knn2.best_estimator_.predict(X_test_knn)
    test_accuracy_knn = prediction(test_predictions_knn, Y_test_knn)
    print("Test accuracy: {}".format(test_accuracy_knn))
    kNN_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_knn))
```

```
Best params: {'metric': 'minkowski', 'n_neighbors': 1, 'weights': 'uniform'}
Train accuracy: 1.0
Test accuracy: 0.950530035335689
Best params: {'metric': 'minkowski', 'n_neighbors': 1, 'weights': 'uniform'}
Train accuracy: 1.0
Test accuracy: 0.9535302353490233
Best params: {'metric': 'minkowski', 'n_neighbors': 1, 'weights': 'uniform'}
Train accuracy: 1.0
Test accuracy: 0.9537302486832455
```

Random Forest

In [9]:

```
#Three lists to store training accuracy for each trial, test accuracy for each trial, and best parameters for each trial.
rf_train_accuracy = []
rf_test_accuracy = []
rf_best_params = []

#For each trial
for i in range(3):
    data_rf = shuffle(letter_arr)
#Data splitting, train-test-split
    X_rf = data_rf[:, 0:-1]
    Y_rf = data_rf[:, -1]
    X_train_rf, X_test_rf, Y_train_rf, Y_test_rf = train_test_split(X_rf, Y_rf, test_size = 14999/19999, random_state=42)
#Param-grid, initialize random forest class, and fitting training data with grid search.
    params = [{'n_estimators' : [1024], 'max_features' : [1, 2, 4, 6, 8, 12, 16, 20]}]
    forest = RandomForestClassifier()
    grid_search_rf = GridSearchCV(forest, params, cv=5, error_score = np.nan)
    grid_search_rf2 = grid_search_rf.fit(X_train_rf, Y_train_rf)
#Printing and storing best params into list.
    best_params_rf = grid_search_rf2.best_params_
    print("Best params: {}".format(best_params_rf))
    rf_best_params.append('Trial ' + str(i + 1) + ': ' + str(best_params_rf))
#Using best_estimator to predict training data.
#Storing and printing training accuracy into list.
#Using defined prediction function to calculate total accuracy.
    train_predictions_rf = grid_search_rf2.best_estimator_.predict(X_train_rf)
    train_accuracy_rf = prediction(train_predictions_rf, Y_train_rf)
    print("Train accuracy: {}".format(train_accuracy_rf))
    rf_train_accuracy.append('Trial ' + str(i + 1) + ': ' + str(train_accuracy_rf))
#Predicting Test data with best_estimator.predict.
#Using prediction function to calculate accuracy.
#Printing and storing Test accuracy into list.
    test_predictions_rf = grid_search_rf2.best_estimator_.predict(X_test_rf)
    test_accuracy_rf = prediction(test_predictions_rf, Y_test_rf)
    print("Test accuracy: {}".format(test_accuracy_rf))
```

```
    rf_test_accuracy.append('Trial ' + str(i + 1) + ': ' + str(test_accuracy_rf))
```

```
Best params: {'max_features': 4, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.9449963330888725

Best params: {'max_features': 4, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.9455963730915394

Best params: {'max_features': 2, 'n_estimators': 1024}
Train accuracy: 1.0
Test accuracy: 0.9453963597573172
```

T-TESTING CODE

In [1]:
```
import numpy as np
from scipy.stats import ttest_rel
```

In [2]:
```
#Lists with test performance for each trial
#Adult
adult_svm = [0.82759, 0.82548, 0.8264]
adult_knn = [0.82413, 0.82417, 0.82]
adult_rf = [0.824, 0.823, 0.82449]
#Covertype
cov_svm = [0.76,0.7628, 0.7587]
cov_knn = [0.779, 0.7756, 0.7782]
cov_rf = [0.8218, 0.8146, 0.82608]
#Letter
letter_svm = [0.8923, 0.919, 0.886]
letter_knn = [0.9505, 0.9535, 0.9537]
letter_rf = [0.9449, 0.94559, 0.94539]
```

In [3]:
```
#Dataset Alg. Comparison
#Adult Dataset Algorithm Comparison
print(ttest_rel(adult_svm, adult_knn))
print(ttest_rel(adult_svm, adult_rf))
print(ttest_rel(adult_knn, adult_rf))

#Covertype Dataset Algorithm Comparison
print(ttest_rel(cov_svm, cov_knn))
print(ttest_rel(cov_svm, cov_rf))
print(ttest_rel(cov_knn, cov_rf))

#Letter Dataset Algorithm Comparison
print(ttest_rel(letter_svm, letter_knn))
print(ttest_rel(letter_svm, letter_rf))
print(ttest_rel(letter_knn, letter_rf))
```

```
Ttest_relResult(statistic=2.5238765169739845, pvalue=0.12761818975674855)
Ttest_relResult(statistic=5.392753658967343, pvalue=0.03270808658168681)
Ttest_relResult(statistic=-0.6113085680021051, pvalue=0.6032218613239739)
Ttest_relResult(statistic=-7.935625778523586, pvalue=0.015511030659130757)
Ttest_relResult(statistic=-13.236819490442409, pvalue=0.005658926170328615)
Ttest_relResult(statistic=-16.804695997589523, pvalue=0.0035224051776294603)
Ttest_relResult(statistic=-5.4160452507636, pvalue=0.03244088226443116)
Ttest_relResult(statistic=-4.62130525421406, pvalue=0.043772729144864124)
Ttest_relResult(statistic=8.611600015911057, pvalue=0.013217664942198835)
```

In [4]:

```
#Overall Alg. Comparison
all_svm = adult_svm + cov_svm + letter_svm
all_knn = adult_knn + cov_knn + letter_knn
all_rf = adult_rf + cov_rf + letter_rf

print(ttest_rel(all_svm, all_knn))
print(ttest_rel(all_svm, all_rf))
print(ttest_rel(all_knn, all_rf))
```

```
Ttest_relResult(statistic=-2.5146834648665815, pvalue=0.03610654286797922)
Ttest_relResult(statistic=-3.4430929984688876, pvalue=0.008783689706310284)
Ttest_relResult(statistic=-1.5678078846655759, pvalue=0.15556241874665216)
```

REFERENCES

Caruana, R & Niculescu-Mizil, A.(2006). An Empirical Comparison of Supervised Learning Algorithms. *In Proceedings of the 23rd international conference on Machine Learning.*

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.

Fleisher. J(2020). COGS 118A. Supervised Machine Learning Algorithms. University of California, San Diego, La Jolla, CA.

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