COGS 118A: Classification Model Comparison

Rem C. Talde Jr rtalde@ucsd.edu

Abstract

This paper describes the replication of Caruana and Niculescu-Mizil's 2006 paper titled *An Empirical Comparison of Supervise Learning Algorithms*, in which the authors compared various supervised machine learning algorithms across different data sets and different metrics. This paper focuses specifically on the learning algorithms of random forests, support-vector machines, and logistic regression and compares how well they do on binary classification tasks. Tuning of the algorithms is done by using grid search, so that the best model would be used for training. The metrics used to compared the three algorithms across data sets are accuracy, F-score, and ROC AUC. Results found in this paper are similar to Caruana and Niculescu-Mizil's results and the widespread "no free lunch" theorem, that there is no single algorithm that is the best choice for all problems. For the data gathered in this paper, support-vector machines and random forests perform quite similar when averaged, but even then, performance really depends on the problem.

Keywords: Binary Classification, Random Forests, Support-Vector Machines, Logistic Regression, Supervise Learning, Machine Learning

1. Introduction

Caruana and Niculescu-Mizil's research (2006), henceforth referred to as CNM06, is a result of updating comparisons between different supervised learning algorithms. Prior to CNM06, STATLOG was the most well-known study to do these comparisons (King et al., 1995). CNM06 compares previous algorithms with new algorithms at the time (e.g., SVMs, random forests).

This paper focuses on four data sets, three algorithms, and three performance metrics. CNM06 looks at eleven different data sets, eight different algorithms, measured across eight different metrics. These eight metrics are divided into three groups – threshold metrics, ordering/rank metrics, and probability metrics. This paper uses accuracy, F-score, and ROC AUC to measure performance. Accuracy and F-score are examples of threshold metrics, where what is important is if the score is above a specified threshold. ROC AUC is an example of a rank metric, and it describes the model's performance in correctly classifying classes.

Caruana and Niculescu-Mizil's paper describes how specifically calibrated boosted trees were overall the best learning algorithm, followed by random forests and SVMs (2006). Some of the worst performers outlined by CNM06 included logistic regression as well as boosted stumps.

This paper is a small-scale replication of Caruana and Niculescu-Mizil's paper. This paper evaluates three algorithms (random forests, support-vector machines, and logistic regression) along three performance metrics of accuracy, F-score, and ROC AUC. Four data sets are used, three of which are used by the original paper (ADULT, LETTER, COVTYPE) and one new data set, focusing on classification of COVID-19 positive and negative patients. By introducing a new data set, comparisons can be made between the performance of certain algorithms for specific problem sets.

2. Methodology

2.1 Learning Algorithms

Three different learning algorithms are compared in this study: random forests, support-vector machines, and logistic regression. Scikit learn functions were used, and parameters remained set to their defaults, except the hyper parameters tuned in grid search.

Logistic Regression (LR) implements two different versions. One logistic regression classifier had the "saga" solver and either a penalty of L1 or L2. The other classifier uses the "lbfgs" solver with L2 penalty. Both classifiers have a range of C values from 1e-04 to 1e+04.

Support-Vector Machines (SVMs) implements two different kernels: "rbf" and "sigmoid". Both C values and gamma values are in the range from 1e-04 to 1e+04.

Random Forests (RF) have 1024 trees, the same as CNM06, with the max features and max depth being tuned. The max features being tuned were [1, 2, 4, 6] while max depth was composed of [1, 2, 4, 6, 8, 12, 16, 20]. The difference in range of hyperparameters in this paper and in CNM06 is a result of one data set, the COVID-19 data set, not having enough features to be evaluated on the same scale as max depth.

Random forests and support-vector machines are algorithms that were not present in STATLOG. When tuned properly, random forests and SVMs can perform well on classification tasks. Logistic regression, while still good, usually under-performs compared to these two algorithms. This paper serves to see if CNM06's findings hold true, and further investigate how these algorithms approach different problems.

2.2 Performance Metric

Algorithms are scored on accuracy, F-score and ROC AUC (AUC).

$$Accuracy\ (ACC) = (\sum True_{positive} + \sum True_{negative}) \frac{1}{\sum Total_{population}}$$

$$F - Score\ (FSC) = 2 * \frac{Precision * Recall}{Precision + Recall}$$

$$AUC = \int TPR\ d(FPR)$$

2.3 Data Sets

Three of the data sets used in this paper are the same data sets from CNM06, which are from the U.C.I. Machine Learning Repository. These data sets include ADULT, COVTYPE, and LETTER. The new data set, from GitHub, is the data set used in a paper about machine learning based

predictions and will be referred to as COVID (Zoabi et al., 2021). Prior to conducting experiments. All data sets are scaled with Standard Scaler (Pedregosa et al., 2011).

ADULT is set to a binary classification problem by setting adults that earn $50,000/year or more as the positive class and less than that amount as the negative class. The ADULT data set also contains categorical data, so these features are one-hot encoded. Other changes to the data set includes dropping columns that are difficult to understand. These columns were used prior to being uploaded to the repository and are discarded as they would not contribute to this study's classification. ADULT is slightly unbalanced with approximately 75.9% of the class labels belonging to the negative class and 24.1% belonging to the positive class.

LETTER is not originally fit for a binary classification. However, following CNM06, letters A through M are given a positive class label while letters N through Z are given a negative class label. Splitting the data in this manner leads to a balanced problem with 50.3% of samples belonging to the negative class and the remaining 49.7% belonging to the positive class.

COVTYPE is structured for a multi-class classification. To do binary classification, in line with CNM06, the largest class (7) is designated the positive class and all the other labels are the negative class. This leads to a very unbalanced data set with approximately 3.5% of samples belonging to the positive class and 96.5% belonging to the negative class.

The COVID data set used in this paper is specifically created to tackle the problem of classifying positive and negative patients. Not much additional cleaning is needed. The changes that made include dropping gender, due to the original source of this data did not provide many gender options, and test date, as no other dates were being recorded. If other dates were included, such as date of recovery, perhaps test date would have influence on infection rate. Similarly, in this paper, "test indication" is being interpreted as knowing you had contact with a COVID-19 positive individual. COVID is a very unbalanced data set with approximately 92.3% of negative class labels and 7.7% samples with a positive class label.

Table 1. Problem Set Summary

	#ATTR	TRAIN SIZE	TEST SIZE	%POS
ADULT	14/105	5000	27561	24%
LETTER	16	5000	15000	50%
COVTYPE	13	5000	576012	4%
COVID	9/7	5000	144292	8%

3. Experiments & Results

For each data set and algorithm combination, 5000 data samples are randomly chosen. These data samples are then scaled. Afterwards, these 5000 samples are used to conduct a 5-fold cross

validation. Each trial tunes hyperparameters via grid search. The hyperparameters vary per algorithm. After cross validating, the best hyperparameters are chosen for each of the three metrics. These specific hyperparameters are then used to train three individual models. After training, each of the three models are evaluated on the remaining test set. Afterwards, the mean is taken from the three models, and this score is saved. This procedure constitutes one trial. Each data set/algorithm combination runs through five trials. After the five trials, the mean of the five trials is computed and recorded in the tables below.

The results of this study are presented in Table 2 and Table 3. Values that are bolded represent the highest value in that column. Values with an * signify a value that is not significantly different from the bold value. Values appended with an (!) signify that this value is significantly lower than the bold value. These values are calculated with unpaired t-tests. The p-value threshold is set at p = 0.05.

Table 2. Normalized score for each learning algorithm by metric (average over four problems)

	ACC	FSC	AUC
LR	0.854817 (!)	0.854817 (!)	0.703512 (!)
SVM	**0.937775**	**0.937775**	**0.814420**
RF	0.933528*	0.933528*	0.813413*

Table 2 indicates that SVMs are the top performers in this study. However, random forests did not perform poorly. Random forests achieve scores not significantly different compared to SVMs. Logistic regression did not perform well for these problems. This falls in line with CNM06's analysis that when calibrated, random forests and SVMs can perform quite well.

Table 3. Normalized scores for each learning algorithm by problem (averaged over three metrics)

	ADULT	LETTER	COVTYPE	COVID	MEAN
LR	0.735944 (!)	0.724300 (!)	0.910689	0.846595	0.804382
SVM	**0.897055**	**0.958048**	**0.927624**	0.866029	**0.912189**
RF	0.813289	0.946322	0.919773	**0.894576**	0.893490

Table 3 further supports this claim. Across three out of the four data sets, SVMs score the highest when averaged across the three metrics. In only the COVID data set did random forests perform better than SVMS. One factor that is consistent is that logistic regression performs the worst across all data sets. However, for COVTYPE and COVID, despite being the lowest, it is not significantly lower, such as in the ADULT data set.

4. Discussion

Overall, the study presented in this paper seems to replicate the study done by Caruana and Niculescu-Mizil (2006). As found in the previous study, SVMs and random forests perform well when tuned. Additionally, logistic regression does not perform as well compared to calibrated SVMs and random forests. This is illustrated when comparing algorithms by metrics as well as comparing algorithms by data set.

Looking across data sets, ADULT has the lowest mean scores, followed by COVID, LETTER, and then COVTYPE with the highest mean scores. This is interesting because COVTYPE and LETTER are examples of a very unbalanced data set, and an almost perfectly balanced data set, respectively. Across all the variances of the data sets, SVMs perform significantly better compared to logistic regression and random forests perform slightly worse when compared to SVMs. Though this study represents a small subset compared to CNM06, the results are similar. These results continue to show that there is no ideal algorithm (Wolpert & Macready, 1997). Different algorithms can perform better than others depending on the problem at hand.

Appendix

Supplement 1. Mean test scores across five trials per algorithm/data set combination.

Algo/Dataset Combo	Average Accuracy	Average F1	Average AUC
LR/ADULT	0.775973	0.775973	0.655887
LR/LETTER	0.724280	0.724280	0.724339
LR/COVERTYPE	0.976884	0.976884	0.778298
LR/COVID	0.942130	0.942130	0.655526
SVM/ADULT	0.864865	0.864865	0.775048
SVM/LETTER	0.958058	0.958058	0.958028
SVM/COVERTYPE	0.980208	0.980208	0.822456
SVM/COVID	0.947970	0.947970	0.702148
RF/ADULT	0.849326	0.849326	0.741216
RF/LETTER	0.946316	0.946316	0.946335
RF/COVERTYPE	0.982728	0.982728	0.793862
RF/COVID	0.955744	0.955744	0.772240

CODE is shown after References.

References

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

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapea, M. Brucher, M. Perrot & E. Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *JMLR*, 2825-2830.

R. Caruana and A. Niculescu-Mizil. 2006. An empirical comparison of supervised learning algorithms. *Proceedings of the 23rd international conference on Machine learning*, 161-168.

D. H. Wolpert and W. G. Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, **1**(1), 67-82. doi: 10.1109/4235.585893.

Y. Zoabi, S. Deri-Rozov & N. Shomron. 2021. Machine learning-based prediction of COVID-19 diagnosis based on symptoms. *npj Digit. Med.* **4**(3) https://doi.org/10.1038/s41746-020-00372-6

Import statements.

```
import pandas as pd
import numpy as np

from sklearn import svm
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score,make_scorer
from sklearn.model_selection import KFold,StratifiedKFold,GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler

import warnings
warnings.filterwarnings('always')
warnings.simplefilter(action='ignore', category=FutureWarning)
from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)
```

Reading data sets into DataFrames.

```
# Imports adult dataset
adult = pd.read_csv('adult.data', header = None)

# One hot encodes categorical variables
# Drops irrelevant columns
# Renames columns to be more user friendly
adult = pd.get_dummies(adult,prefix = '', prefix_sep = '')
adult = adult.drop(columns = [2,' ?'])
adult = adult.rename(columns = {0:'age', 4:'education-num',10:'capital-gain',
    11:'capital-loss',12:'hours-per-week'})
```

```
letter = pd.read_csv('letter-recognition.data', header = None)
```

```
covtype = pd.read_csv('covtype.data.gz', header = None)
```

```
# Covid-19 dataset provided from
# https://github.com/nshomron/covidpred
covid = pd.read_csv('covid.english.csv', dtype='unicode')

# Drop test date as we are not concerned with turnaround time, gender
# Drop samples that do not have test results reported
# Turn into binary classification of +1 v -1
covid = covid.drop(columns = ['test_date', 'gender'])
covid = covid[covid["age_60_and_above"].str.contains("None")==False]
covid = covid[covid["corona_result"].str.contains("other")==False]
covid = covid.replace(['Yes','positive','Contact with confirmed','1'],1)
covid = covid.replace(['No','negative','None','Other','Abroad','0'],-1)
covid = covid.reset_index(drop = True)
```

Function called to extract the best logistic regression model per metric found in grid search.

```
def ranked_classifier(fitted):
'''

Function called to return the model associated with the best parameters associated with each metric

Input - fit GridSearchCV classifier
Outputs - three Logistic Regression models

'''
    # Gets the index of the highest ranked metric (best)
    rank_acc = pd.Series(fitted.cv_results_['rank_test_accuracy']).argmin()
    rank_f1 = pd.Series(fitted.cv_results_['rank_test_f1_micro']).argmin()
    rank_auc = pd.Series(fitted.cv_results_['rank_test_roc_auc_ovr_weighted']).argmin()
    # Gets the associated logistic regression model of index
    acc_class = fitted.cv_results_['params'][rank_acc]['classifier']
    f_class = fitted.cv_results_['params'][rank_f1]['classifier']
    auc_class = fitted.cv_results_['params'][rank_auc]['classifier']
    return acc_class,f_class,auc_class
```

Function called to extract the best C values per metric found in grid search.

```
def ranked_Cval(fitted):
'''

Function called to return the C Values associated with the best parameters associated with each metric

Input - fit GridSearchCV classifier
Outputs - three C values

'''
    # Gets the index of the highest ranked metric (best)
    rank_acc = pd.Series(fitted.cv_results_['rank_test_accuracy']).argmin()
    rank_f1 = pd.Series(fitted.cv_results_['rank_test_f1_micro']).argmin()
    rank_auc = pd.Series(fitted.cv_results_['rank_test_roc_auc_ovr_weighted']).argmin()
    # Gets the associated C value of the index
    acc_c = fitted.cv_results_['params'][rank_acc]['classifier__C']
    f_c = fitted.cv_results_['params'][rank_f1]['classifier__C']
    auc_c = fitted.cv_results_['params'][rank_auc]['classifier__C']
    return acc_c,f_c,auc_c
```

Function called to extract the best penalty per metric found in grid search.

```
def ranked_penalty(fitted):
'''

Function called to return the penalty associated with the best parameters associated with each metric

Input - fit GridSearchCV classifier
Outputs - three penalties

'''
    # Gets the index of the highest ranked metic (best)
    rank_acc = pd.Series(fitted.cv_results_['rank_test_accuracy']).argmin()
    rank_f1 = pd.Series(fitted.cv_results_['rank_test_f1_micro']).argmin()
    rank_auc = pd.Series(fitted.cv_results_['rank_test_roc_auc_ovr_weighted']).argmin()
    # Gets the associated penalty of the index
    acc_pen = fitted.cv_results_['params'][rank_acc]['classifier__penalty']
    f_pen = fitted.cv_results_['params'][rank_f1]['classifier__penalty']
    auc_pen = fitted.cv_results_['params'][rank_auc]['classifier__penalty']
    return acc_pen,f_pen,auc_pen
```

Function called to make a logistic regression model using the hyperparameters found.

```
def make_model(model,c,penalty,X_train,Y_train,X_test,Y_test):
'''

Function called to make a logistic regression model, train on training set, and compared metrics
                           against ground truth and predicted

Inputs - logistic regression model, c value, penalty, X-train set, Y-train set, X-test set, Y-test set
Output - DataFrame with accuracy, f-score, and roc-auc scores

'''


    model_res = []
    predicteds=[]
    trueys=[]

    # Sets the C value and penalty for input logistic regression model
    model = model.set_params(**{'C':c, 'penalty':penalty})
    model.fit(X_train,Y_train)

    predicteds.append( model.predict(X_test))
    trueys.append(Y_test)
    predicteds = np.concatenate(predicteds)
    trueys = np.concatenate(trueys)

    # Gets the score for each metric, comparing the ground truth to the models predicted results
    model_res.append([accuracy_score(trueys,predicteds), f1_score(trueys,predicteds, average = 'micro'),
        roc_auc_score(trueys,predicteds, average = 'weighted')])
    model_results = pd.DataFrame( model_res, columns=['accuracy', 'F score', 'AUC'])
    return model_results
```

Example of setting up pipeline and parameter grid for one logistic regression/data set combination.

```
# Sets up pipeline and parameter grid to conduct GridSearch
pipe = Pipeline([('classifier', LogisticRegression())])
param_grid = [{'classifier':[LogisticRegression(solver = 'saga', max_iter=5000)],
               'classifier__penalty':['l1','l2'],
               'classifier__C':np.logspace(-4,4,9)},
              {'classifier':[LogisticRegression(solver = 'lbfgs', max_iter=5000)],
               'classifier__penalty':['l2'],
               'classifier__C': np.logspace(-4,4,9)},
             ]

clf = GridSearchCV(pipe,param_grid,cv=StratifiedKFold(n_splits = 5),
                   scoring = ['accuracy','f1_micro','roc_auc_ovr_weighted'], refit = False)

# Sets up empty list to compile the scores per trial
three_res_df = []
```

Example of executing five trials of logistic regression/data set combination.

```
for trial in range(5):
    # Shuffles the data
    adult = adult.sample(frac=1).reset_index(drop=True)

    # Gets training and test sets
    X_train = StandardScaler().fit(adult.iloc[:5000,:-2])
    X_train = X_train.transform(adult.iloc[:5000,:-2])
    X_train = adult.iloc[:5000,:-2].values
    Y = adult.iloc[:5000,-1:]
    Y = Y.replace({0:-1})
    Y_train = Y.values.ravel()

    X_test = StandardScaler().fit(adult.iloc[5000:,:-2])
    X_test = X_test.transform(adult.iloc[5000:,:-2])
    #X_test = adult.iloc[5000:,:-2].values
    Y = adult.iloc[5000:,-1:]
    Y = Y.replace({0:-1})
    Y_test = Y.values.ravel()

    # Conducts a grid search to fnd the best hyperparameters
    best_clf = clf.fit(X_train,Y_train)

    # Gets the models, c values, and penalties for each of the metrics
    acc_class,f_class,auc_class = ranked_classifier(best_clf)
    acc_c,f_c,auc_c = ranked_Cval(best_clf)
    acc_pen,f_pen,auc_pen = ranked_penalty(best_clf)

    # Creates, trains, and evaluates models using the outputs returned from above
    model_acc = make_model(acc_class,acc_c,acc_pen,X_train,Y_train,X_test,Y_test)
    model_f1 = make_model(f_class,f_c,f_pen,X_train,Y_train,X_test,Y_test)
    model_auc = make_model(auc_class,auc_c,auc_pen,X_train,Y_train,X_test,Y_test)

    # Adds scores to lists to calculate means afterwards
    three_res = [model_acc,model_f1,model_auc]
    three_results = pd.concat(three_res)
    three_res_df.append(three_results)
five_trials_res = pd.concat(three_res_df)
scores_dataset_alg.append(['LR/ADULT',five_trials_res['accuracy'].mean(),five_trials_res['F score'].mean(),
    five_trials_res['AUC'].mean()])
```

Function called to extract the best gamma values per metric found in grid search.

```
def ranked_gamma(fitted):
'''

Function called to return the gamma values associated with the best parameters associated with each metric

Input - fit GridSearchCV classifier
Outputs - three gamma values

'''
    # Gets the index of the highest ranked metic (best)
    rank_acc = pd.Series(fitted.cv_results_['rank_test_accuracy']).argmin()
    rank_f1 = pd.Series(fitted.cv_results_['rank_test_f1_micro']).argmin()
    rank_auc = pd.Series(fitted.cv_results_['rank_test_roc_auc_ovr_weighted']).argmin()
    # Gets the associated gamma value of the index
    acc_gamma = fitted.cv_results_['params'][rank_acc]['classifier__gamma']
    f_gamma = fitted.cv_results_['params'][rank_f1]['classifier__gamma']
    auc_gamma = fitted.cv_results_['params'][rank_auc]['classifier__gamma']
    return acc_gamma,f_gamma,auc_gamma
```

Function called to extract the best kernel per metric found in grid search.

```
def ranked_kernel(fitted):
'''

Function called to return the kernel associated with the best parameters associated with each metric

Input - fit GridSearchCV classifier
Outputs - three kernels

'''
    # Gets the index of the highest ranked metic (best)
    rank_acc = pd.Series(fitted.cv_results_['rank_test_accuracy']).argmin()
    rank_f1 = pd.Series(fitted.cv_results_['rank_test_f1_micro']).argmin()
    rank_auc = pd.Series(fitted.cv_results_['rank_test_roc_auc_ovr_weighted']).argmin()
    # Gets the associated kernel of the index
    acc_kernel = fitted.cv_results_['params'][rank_acc]['classifier__kernel']
    f_kernel = fitted.cv_results_['params'][rank_f1]['classifier__kernel']
    auc_kernel = fitted.cv_results_['params'][rank_auc]['classifier__kernel']
    return acc_kernel,f_kernel,auc_kernel
```

Function called to create a SVM model using the hyperparameters found.

```
def make_model_svm(model,c,gamma,kernel,X_train,Y_train,X_test,Y_test):
'''

Function called to make a SVM model, train on training set, and compared metrics against ground truth and predicted

Inputs - SVM model, gamma, kernel, X-train set, Y-train set, X-test set, Y-test set
Output - DataFrame with accuracy, f-score, and roc-auc scores

'''
    model_res = []
    predicteds=[]
    trueys=[]

    # Sets the parameters of the SVM model
    model = model.set_params(**{'C':c,'gamma':gamma,'kernel':kernel})
    model.fit(X_train,Y_train)

    predicteds.append( model.predict(X_test))
    trueys.append(Y_test)
    predicteds = np.concatenate(predicteds)
    trueys = np.concatenate(trueys)

    # Gets the score for each metric, comparing the ground truth to the models predicted results
    model_res.append([accuracy_score(trueys,predicteds), f1_score(trueys,predicteds, average = 'micro'),
        roc_auc_score(trueys,predicteds, average = 'weighted')])
    model_results = pd.DataFrame( model_res, columns=['accuracy', 'F score', 'AUC'])
    return model_results
```

Example of setting up pipeline and parameter grid for one SVM/data set combination.

```
# Sets up the pipeline and parameter grid for use in GridSearchCV
pipe = Pipeline([('classifier', svm.SVC())])
param_grid = [{'classifier':[svm.SVC(probability=True)],
               'classifier__kernel':['rbf'],
               'classifier__C':np.logspace(-4,4,9),
               'classifier__gamma':np.logspace(-4,4,9)},
              {'classifier':[svm.SVC(probability=True)],
               'classifier__kernel':['sigmoid'],
               'classifier__C':np.logspace(-4,4,9),
               'classifier__gamma':np.logspace(-4,4,9)},
                ]

clf = GridSearchCV(pipe,param_grid,cv=StratifiedKFold(n_splits = 5),
                   scoring = ['accuracy','f1_micro','roc_auc_ovr_weighted'], refit = False)

# Sets up empty list to compile the scores per trial
three_res_df = []
```

Example of executing five trials of SVM/data set combination.

```
for trial in range(5):
    letter = letter.sample(frac=1).reset_index(drop=True)

    # Gets training and test sets
    X_train = StandardScaler().fit(letter.iloc[:5000,1:])
    X_train = X_train.transform(letter.iloc[:5000,1:])
    Y = letter.iloc[:5000,0]
    Y = Y.replace(pos_let,1)
    Y = Y.replace(neg_let,-1)
    Y_train = Y.values.ravel()

    X_test = StandardScaler().fit(letter.iloc[5000:,1:])
    X_test = X_test.transform(letter.iloc[5000:,1:])
    Y = letter.iloc[5000:,0]
    Y = Y.replace(pos_let,1)
    Y = Y.replace(neg_let,-1)
    Y_test = Y.values.ravel()

    # Conducts a grid search to fnd the best hyperparameters
    best_clf = clf.fit(X_train,Y_train)

    # Gets the classifier, c values, gamma values, and kernel from the best model
    acc_class,f_class,auc_class = ranked_classifier(best_clf)
    acc_c,f_c,auc_c = ranked_Cval(best_clf)
    acc_gamma,f_gamma,auc_gamma = ranked_gamma(best_clf)
    acc_kernel,f_kernel,auc_kernel = ranked_kernel(best_clf)

    # Creates, trains, and evaluates models using the outputs returned from above
    model_acc = make_model_svm(acc_class,acc_c,acc_gamma,acc_kernel,X_train,Y_train,X_test,Y_test)
    model_f1 = make_model_svm(f_class,f_c,f_gamma,f_kernel,X_train,Y_train,X_test,Y_test)
    model_auc = make_model_svm(auc_class,auc_c,auc_gamma,auc_kernel,X_train,Y_train,X_test,Y_test)

    # Adds scores to lists to calculate means afterwards
    three_res = [model_acc,model_f1,model_auc]
    three_results = pd.concat(three_res)
    three_res_df.append(three_results)
five_trials_res = pd.concat(three_res_df)
scores_dataset_alg.append(['SVM/LETTER',five_trials_res['accuracy'].mean(),five_trials_res['F score'].mean(),
    five_trials_res['AUC'].mean()])
```

Function called to extract the best max depth per metric found in grid search.

```
def ranked_depth(fitted):
'''

Function called to return the max depth associated with the best parameters associated with each metric

Input - fit GridSearchCV classifier
Outputs - three max depth values

'''
    # Gets the index of the highest ranked metic (best)
    rank_acc = pd.Series(fitted.cv_results_['rank_test_accuracy']).argmin()
    rank_f1 = pd.Series(fitted.cv_results_['rank_test_f1_micro']).argmin()
    rank_auc = pd.Series(fitted.cv_results_['rank_test_roc_auc_ovr_weighted']).argmin()
    # Gets the associated max_depth of the index
    acc_depth = fitted.cv_results_['params'][rank_acc]['max_depth']
    f_depth = fitted.cv_results_['params'][rank_f1]['max_depth']
    auc_depth = fitted.cv_results_['params'][rank_auc]['max_depth']
    return acc_depth,f_depth,auc_depth
```

Function called to extract the best max features per metric found in grid search.

```
def ranked_features(fitted):
'''

Function called to return the max features  associated with the best parameters associated with each metric

Input - fit GridSearchCV classifier
Outputs - three max features

'''
    # Gets the index of the highest ranked metic (best)
    rank_acc = pd.Series(fitted.cv_results_['rank_test_accuracy']).argmin()
    rank_f1 = pd.Series(fitted.cv_results_['rank_test_f1_micro']).argmin()
    rank_auc = pd.Series(fitted.cv_results_['rank_test_roc_auc_ovr_weighted']).argmin()
    # Gets the associated max features of the index
    acc_features = fitted.cv_results_['params'][rank_acc]['max_features']
    f_features = fitted.cv_results_['params'][rank_f1]['max_features']
    auc_features = fitted.cv_results_['params'][rank_auc]['max_features']
    return acc_features,f_features,auc_features
```

Function called to create a random forest model using the hyperparameters found.

```
def make_model_rf(depth,features,X_train,Y_train,X_test,Y_test):
    model_res = []
    predicteds=[]
    trueys=[]

    # Makes model with depth and features
    model = RandomForestClassifier(n_jobs = -1, n_estimators = 1024,
        oob_score = True, max_features = features, max_depth = depth)
    model.fit(X_train,Y_train)

    predicteds.append( model.predict(X_test))
    trueys.append(Y_test)
    predicteds = np.concatenate(predicteds)
    trueys = np.concatenate(trueys)

    # Gets the score for each metric, comparing the ground truth to the models predicted results
    model_res.append([accuracy_score(trueys,predicteds), f1_score(trueys,predicteds, average = 'micro'),
        roc_auc_score(trueys,predicteds, average = 'weighted')])
    model_results = pd.DataFrame( model_res, columns=['accuracy', 'F score', 'AUC'])
    return model_results
```

Example of setting up classifier and parameters for one RF/data set combination.

```
# Initializes the max features, max depth, and Random Forest classifier to use for grid search
max_features = [1,2,4,6]
max_depth = [1,2,4,6,8,12,16,20]
classifier = RandomForestClassifier(n_jobs = -1, n_estimators = 1024, oob_score = True)

clf =  GridSearchCV(estimator = classifier,param_grid = {'max_features':max_features, 'max_depth':max_depth},
    scoring = ['accuracy','f1_micro','roc_auc_ovr_weighted'], refit = False  )

# Sts up empty list to compile the scores per trial
three_res_df = []
```

Example of executing five trials of RF/data set combination.

```
for trial in range(5):
    # Randomly shuffles the data
    covid = covid.sample(frac=1).reset_index(drop=True)

    # Gets training and test sets
    X_train = StandardScaler().fit(covid.loc[:5000,covid.columns != 'corona_result'])
    X_train = X_train.transform(covid.loc[:5000,covid.columns != 'corona_result'])
    Y = covid.loc[:5000,covid.columns == 'corona_result']
    Y_train = Y.values.ravel()

    X_test = StandardScaler().fit(covid.loc[5000:,covid.columns != 'corona_result'])
    X_test = X_test.transform(covid.loc[5000:,covid.columns != 'corona_result'])
    Y = covid.loc[5000:,covid.columns == 'corona_result']
    Y_test = Y.values.ravel()

    # Conducts a grid search to find the best hyperparameters
    best_clf = clf.fit(X_train,Y_train)

    # Gets the max depth and max features from the best models
    acc_depth,f_depth,auc_depth = ranked_depth(best_clf)
    acc_features,f_features,auc_features = ranked_features(best_clf)

    # Creates, trains, and evaluates models using the outputs returned from above
    model_acc = make_model_rf(acc_depth,acc_features,X_train,Y_train,X_test,Y_test)
    model_f1 = make_model_rf(f_depth,f_features,X_train,Y_train,X_test,Y_test)
    model_auc = make_model_rf(auc_depth, auc_features,X_train,Y_train,X_test,Y_test)

    # Adds scores to lists to calculate means afterwards
    three_res = [model_acc,model_f1,model_auc]
    three_results = pd.concat(three_res)
    three_res_df.append(three_results)
five_trials_res = pd.concat(three_res_df)
scores_dataset_alg.append(['RF/COVID',five_trials_res['accuracy'].mean(),five_trials_res['F score'].mean(),
    five_trials_res['AUC'].mean()])
```

Compiling all the kept scores into one DataFrame.

```
res1 = pd.DataFrame(scores_dataset_alg, columns = ['Algo/Dataset Combo','Average Accuracy','Average F1', 'Average AUC'])
res1
```

	Algo/Dataset Combo	Average Accuracy	Average F1	Average AUC
0	LR/ADULT	0.775973	0.775973	0.655887
1	LR/LETTER	0.724280	0.724280	0.724339
2	LR/COVERTYPE	0.976884	0.976884	0.778298
3	LR/COVID	0.942130	0.942130	0.655526
4	SVM/ADULT	0.864865	0.864865	0.775048
5	SVM/LETTER	0.958058	0.958058	0.958028
6	SVM/COVERTYPE	0.980208	0.980208	0.822456
7	SVM/COVID	0.947970	0.947970	0.702148
8	RF/ADULT	0.849326	0.849326	0.741216
9	RF/LETTER	0.946316	0.946316	0.946335
10	RF/COVERTYPE	0.982728	0.982728	0.793862
11	RF/COVID	0.955744	0.955744	0.772240

Condensing the above table to show algorithms against metrics.

```
avg_acc_lr = np.mean([res1.iloc[0,1],res1.iloc[1,1],res1.iloc[2,1],res1.iloc[3,1]])
avg_f1_lr = np.mean([res1.iloc[0,2],res1.iloc[1,2],res1.iloc[2,2],res1.iloc[3,2]])
avg_auc_lr = np.mean([res1.iloc[0,3],res1.iloc[1,3],res1.iloc[2,3],res1.iloc[3,3]])

avg_acc_svm = np.mean([res1.iloc[4,1],res1.iloc[5,1],res1.iloc[6,1],res1.iloc[7,1]])
avg_f1_svm = np.mean([res1.iloc[4,2],res1.iloc[5,2],res1.iloc[6,2],res1.iloc[7,2]])
avg_auc_svm = np.mean([res1.iloc[4,3],res1.iloc[5,3],res1.iloc[6,3],res1.iloc[7,3]])

avg_acc_rf = np.mean([res1.iloc[8,1],res1.iloc[9,1],res1.iloc[10,1],res1.iloc[11,1]])
avg_f1_rf = np.mean([res1.iloc[8,2],res1.iloc[9,2],res1.iloc[10,2],res1.iloc[11,2]])
avg_auc_rf = np.mean([res1.iloc[8,3],res1.iloc[9,3],res1.iloc[10,3],res1.iloc[11,3]])
```

```
res2 = []
res2.append(['LR',avg_acc_lr,avg_f1_lr,avg_auc_lr])
res2.append(['SVM',avg_acc_svm,avg_f1_svm,avg_auc_svm])
res2.append(['RF',avg_acc_rf,avg_f1_rf,avg_auc_rf])
res2 = pd.DataFrame(res2,columns = ['Algorithm','Average Accuracy','Average F Score', 'Average AUC'])
res2
```

	Algorithm	Average Accuracy	Average F Score	Average AUC
0	LR	0.854817	0.854817	0.703512
1	SVM	0.937775	0.937775	0.814420
2	RF	0.933528	0.933528	0.813413