Arad vazirpanah - 610399182

July 18, 2023

1 Data exploration

using .info() to see if there is a null data (missing data) in the DataFrame. and also to see the type of each column. as we can see, there is no missing data ("-" in service means something. probably it means that no service is given).

```
data.info()
```

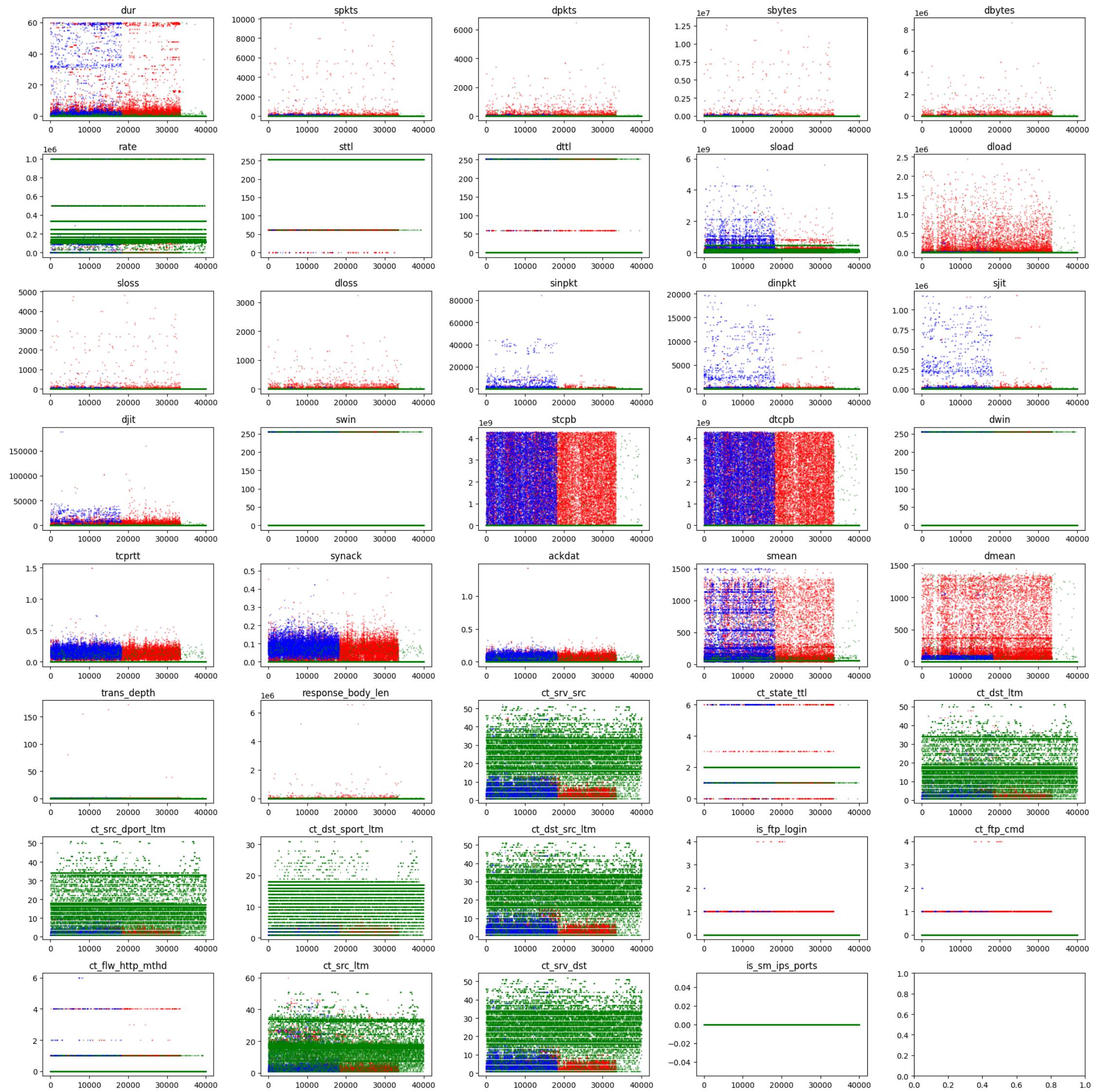
```
Data columns (total 44 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   id                 175341 non-null  int64
 1   dur                175341 non-null  float64
 2   proto              175341 non-null  object
 3   service            175341 non-null  object
 4   state              175341 non-null  object
 5   spkts              175341 non-null  int64
 6   dpkts              175341 non-null  int64
 7   sbytes             175341 non-null  int64
 8   dbytes             175341 non-null  int64
 9   rate               175341 non-null  float64
 10  sttl               175341 non-null  int64
 11  dttl               175341 non-null  int64
 12  sload              175341 non-null  float64
 13  dload              175341 non-null  float64
 14  sloss              175341 non-null  int64
 15  dloss              175341 non-null  int64
 16  sinpkt             175341 non-null  float64
 17  dinpkt             175341 non-null  float64
 18  sjit               175341 non-null  float64
 19  djit               175341 non-null  float64
 20  swin               175341 non-null  int64
 21  stcpb              175341 non-null  int64
 22  dtcpb              175341 non-null  int64
 23  dwin               175341 non-null  int64
 24  tcprtt             175341 non-null  float64
 25  synack             175341 non-null  float64
 26  ackdat             175341 non-null  float64
 27  smean              175341 non-null  int64
 28  dmean              175341 non-null  int64
 29  trans_depth        175341 non-null  int64
 30  response_body_len  175341 non-null  int64
 31  ct_srv_src         175341 non-null  int64
 32  ct_state_ttl       175341 non-null  int64
 33  ct_dst_ltm         175341 non-null  int64
 34  ct_src_dport_ltm   175341 non-null  int64
 35  ct_dst_sport_ltm   175341 non-null  int64
 36  ct_dst_src_ltm     175341 non-null  int64
 37  is_ftp_login       175341 non-null  int64
 38  ct_ftp_cmd         175341 non-null  int64
 39  ct_flw_http_mthd   175341 non-null  int64
 40  ct_src_ltm         175341 non-null  int64
 41  ct_srv_dst         175341 non-null  int64
 42  is_sm_ips_ports    175341 non-null  int64
 43  attack_cat         175341 non-null  object
dtypes: float64(11), int64(29), object(4)
```

As we can see, there is no null value in the data set, and comparing to the paper it shows that the null data has already been handled.

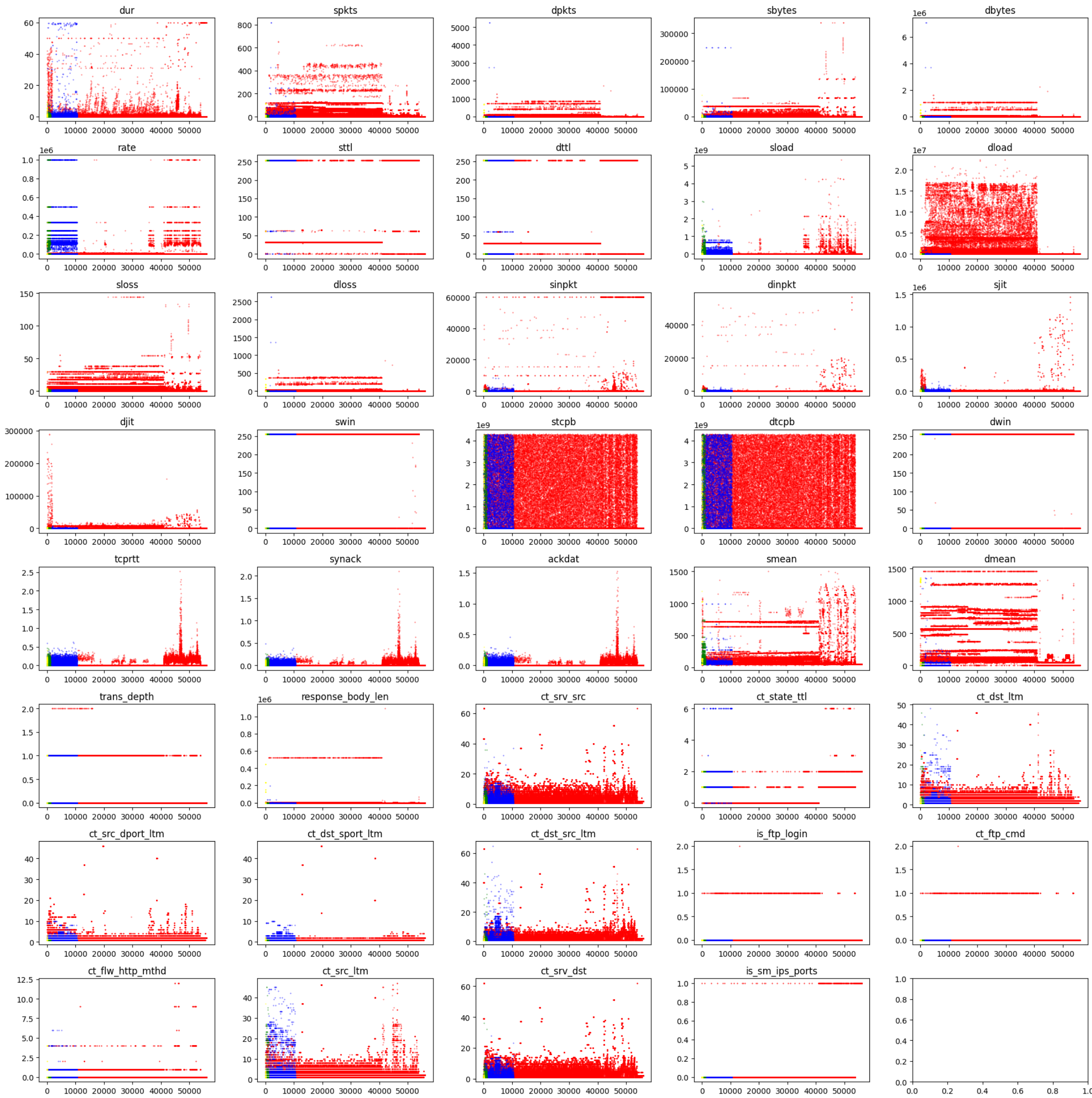
visualizing the distributions of different features to explore some pattern

visualizing the distribution of train data first three features.

visualizing the distribution of train data second three features.



visualizing the distribution of train data last four features.

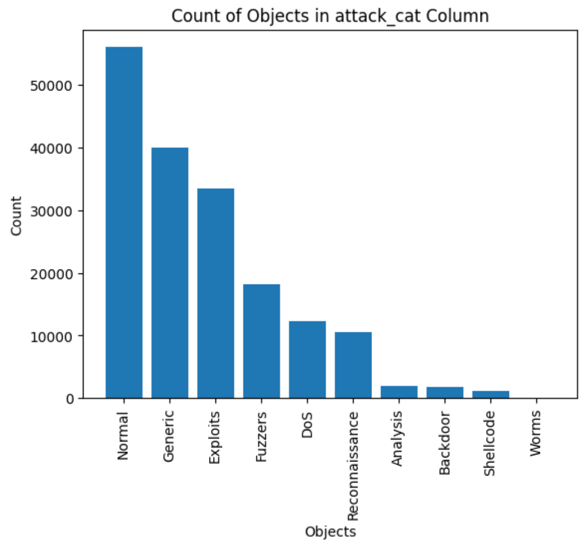


As we can see, there might be some pattern between features and attack category. For example in trans_depth we might use the pattern that separates some of classes from each other. Or in ct_srv_src, ct_dst_ltm, ct_dst_src_ltm and also ct_srv_dst, blue and red classes are separable from green class.

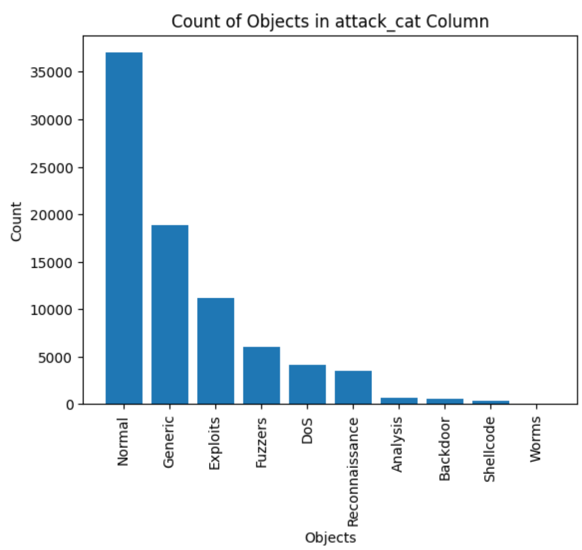
But there is no pattern in most of these plots. So we can see that the data for this problem, is not too suitable, and for sure the accuracy from different learning and classification algorithms is not so impressive as we are looking for.

visualizing the distribution of attack category pattern

Visualizing the attack category for train data to see how many of each label we have.



Visualizing the attack category for test data



As we can see, the ”Normal” category is the most common category in our data. Also the ”Analysis”, ”Backdoor”, ”shellcode” and ”worms” are rare in our data.

One thing that the paper did, is dropping these last four labels because there is not too many of them and dropping them, let us train better for different algorithms. We can do the same, but it doesn't really matter because there is not so many of these labels in our data set. so we leave it as it was.

The other thing that the paper has done, is over sampling. because number of "Normal" category in train data, is less than in test data compared to other categories, we add more data with "Normal" category to our data. This wish is done by adding "Normal" data from the data, to itself again. But we know that checking the test data is not the right thing to do, so I didn't do the same as paper. Because if I did, the new data wouldn't be pure as it has to be. (but we can use a below code for this)

```
# data = pd.concat([data, data[data["attack_cat"] == "Normal"]])
```

2 Feature Processing

from now on, we have X and y which is the separated data from "data" DataFrame. X is the features and y is attack_cat column.

these two functions are written to map output data from object to numbers. which the number is chosen by it's index in np.unique(y_train), array.

```
def convert_to_num(x, arr):
    for i in range(len(arr)):
        if (x == arr[i]):
            return i
```

```
def converter(array):
    map_list = np.unique(array)
    for i in range(len(array)):
        array[i] = convert_to_num(array[i], map_list)
```

calling the function to map

```
converter(y_train)
converter(y_test)
y_train = y_train.astype('int')
y_test = y_test.astype('int')
```

Here we drop the column Proto, Service and State so that we can scale other columns to have the scaled data for train data, and after that, do the same for test data.

```
proto = X_train["proto"]
X_train = X_train.drop("proto", axis=1)
service = X_train["service"]
X_train = X_train.drop("service", axis=1)
state = X_train["state"]
X_train = X_train.drop("state", axis=1)
X_train = X_train.drop('id', axis=1)
```

Then we scale the data using MinMaxScaler().

```
col_names = X_train.columns
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train = pd.DataFrame(scaler.transform(X_train), columns=col_names)
X_test = pd.DataFrame(scaler.transform(X_test), columns=col_names)
```

then again we add those columns that we dropped. so we can have scaled data for non-object columns and object columns also.

```
X_train = pd.DataFrame(pd.concat([X_train, proto, service, state], axis=1),
                        columns= col_names.values.tolist() + ["proto",
 ↪"service", "state"])
X_test = pd.DataFrame(pd.concat([X_test, proto_test, service_test, state_test],
 ↪axis=1),
                        columns= col_names.values.tolist() + ["proto",
 ↪"service", "state"])
```

and after that, we use get_dummies function to one_hot map the object columns (once dropped and then added columns).

This function is also is used in the paper. Using this function, we can convert object type columns to numeric column. This function concatenate some new columns. these columns are filled with 0 and 1s. new columns which were added, is for each different object in each column. 0 means that in this row, for example the service is not "-" (for service_- new column). and 1 means that it is.

```
dummy_X_train = pd.get_dummies(X_train)
dummy_X_test = pd.get_dummies(X_test)
dummy_X_test = dummy_X_test.reindex(columns= dummy_X_train.columns,
 ↪fill_value=0)
```

After scaling, we do the dimention reduction using PCA. So we can have less features with most importance in data set.

```
pca = PCA(n_components=0.98)

pca_train = pca.fit_transform(dummy_X_train)
pca_test = pca.transform(dummy_X_test)
```

Using PCA we extract some new features that contain 98% of the original data. After using PCA instead of 194 features, we will have only 55 features, which makes training the algorithms much easier, faster and also make the accuracy much better.

3 Model selection

Now we train different models. first for each model, we tune the parameters and then, train with the best hyper parameters. After that predict the test data and also the train data itself and calculate the accuracy of the trained classifier using confusion matrix and evaluation metrics.

3.1 SVM

SVM algorithm, tries to find some hyper planes that separates the different classes from each other. It is not so wisely to this classifier because finding some hyper planes that separates this data in 55 dimension is not a easy thing to do. So it takes a lot of time and probably won't have a results that we are looking for.

```
ker = ['rbf', 'sigmoid']
svm_scores = np.array([])
for a in ker:
    svm = SVC(kernel=a)
    score = cross_val_score(svm, pca_train, y_train, cv=5)
    svm_scores = np.append(svm_scores, score.mean())
    print(a, " is done with mean score: ", score.mean())
```

```
rbf  is done with mean score:  0.7570616428484193
sigmoid  is done with mean score:  0.662257851500584
```

So, the best hyper parameter for SVM is 'rbf'. but poly is much better in general. As this conclusion we train the main classifier with 'poly' kernel.

```
svm = SVC(kernel='poly')
score = cross_val_score(svm, pca_train, y_train, cv=5)
print(score.mean())
```

```
0.7567821867693232
```

Then we predict the test data using this classifier. results are not so good. 70% is not a high enough accuracy on predicting.

```
svm.fit(pca_train, y_train)
y_predict = svm.predict(pca_test)
acc = accuracy_score(y_test, y_predict)
accuracy_arr_test = np.append(accuracy_arr_test, acc)
acc
```

```
0.6996186173055434
```

Then we examine the confusion matrix. As we can see in the next page, the confusion matrix shows the reason of low accuracy on predicting test data. For example the fourth class has been predicted much, while it doesn't contains this much data and a lot of them belongs to other classes. In confusion matrix, row are truth labels and columns are predicted classes.

After that we will have precision, recall and F1-score for each category.

```python
print(confusion_matrix(y_test, y_predict))
```

```
[[    0     0     0   676     0     0     1     0     0     0]
 [    0     0     0   546    14     0     0    23     0     0]
 [    0     0   116  3602   229    20    14   108     0     0]
 [    8     0    63  9783   951     2    39   286     0     0]
 [    0     0     0  1358  4312     8   135   249     0     0]
 [    0     0    55   398   206 18161     2    49     0     0]
 [   55     0     7  1262 10950     1 22963  1762     0     0]
 [    0     0     4   800   417     7     2  2266     0     0]
 [    0     0     0     1   125     0     0   252     0     0]
 [    0     0     0    34     9     0     0     1     0     0]]
```

```python
print(precision_recall_fscore_support(y_test, y_predict))
```

```
(array([0.        , 0.        , 0.47346939, 0.52995666, 0.25050834,
       0.99791197, 0.99166523, 0.45356285, 0.        , 0.        ]),
 array([0.        , 0.        , 0.02836879, 0.87881782, 0.7113164 ,
       0.96237613, 0.62062162, 0.64816934, 0.        , 0.        ]),
 array([0.        , 0.        , 0.05353023, 0.66119221, 0.37052632,
       0.97982196, 0.76344837, 0.53367876, 0.        , 0.        ]),
 array([677 ,583 ,4089 ,11132 ,6062 , 18871, 37000, 3496, 378, 44], dtype=int64))
```

Now we predict the train data using SVM classifier. the accuracy is just 77.6% which again, is not high enough.

```python
y_predict = svm.predict(pca_train)
acc = accuracy_score(y_train, y_predict)
accuracy_arr_train = np.append(accuracy_arr_train, acc)
acc
```

```
0.776726492948027
```

Then we have confusion matrix, which the fourth category is not again, predicted very well (most of them, don't belong to this class).

```python
print(confusion_matrix(y_train, y_predict))
```

```
[[   83     0     6  1646     7     0   258     0     0     0]
 [    0     0     0  1581    54     0     4   107     0     0]
 [    1     0   298 11223   424    40    45   233     0     0]
 [    0     0   139 30379  2202    13   131   529     0     0]
 [    3     0     0  1866 14338    37   425  1515     0     0]
 [    0     0    57   627   152 39115     6    43     0     0]
 [   13     0     8   618  8852     0 45877   632     0     0]
 [    0     0    11  3398   951    20     9  6102     0     0]
 [    0     0     0     2   343     0     2   786     0     0]
 [    0     0     0   115    10     0     0     5     0     0]]
```

then we have other evaluation metrics.

```
print(precision_recall_fscore_support(y_train, y_predict))
```

```
(array([0.83      , 0.        , 0.57418112, 0.59039938, 0.52456737,
       0.99719567, 0.98117929, 0.61314309, 0.        , 0.        ]),
array([0.0415    , 0.        , 0.02429876, 0.90974156, 0.78849538,
       0.977875  , 0.81923214, 0.58164141, 0.        , 0.        ]),
array([0.07904762, 0.        , 0.04662442, 0.71608052, 0.63000637,
       0.98744083, 0.89292214, 0.59697696, 0.        , 0.        ]),
array([2000, 1746, 12264, 33393, 18184, 40000, 56000, 10491, 1133, 130],␣
 ↪dtype=int64))
```

3.2 KNN

This algorithm might have a weakness which is that it depends so much on distance between classes. some classes might be close and so they will be misclassified using this algorithm. but it should work much better than SVM.
KNN might be a good classifier for this data. So we train with this classifier.

```
n = [3, 5, 7]
for a in n:
    KNN = KNeighborsClassifier(n_neighbors= a)
    score = cross_val_score(KNN, pca_train, y_train, cv=5)
    knn_scores = np.append(knn_scores, score.mean())
    print("KNN with ", a, "neigbours is done.")
```

```
KNN with  3 neigbours is done.
KNN with  5 neigbours is done.
KNN with  7 neigbours is done.
[0.71318154 0.72706873 0.73208182]
```

So the best number of neighbors for KNN in this data is 7. So we train the main classifier with n_neighbors=7.

```
KNN = KNeighborsClassifier(n_neighbors= 7)
score = cross_val_score(KNN, pca_train, y_train, cv=5)
print(score.mean())
```

```
0.731642673886765
```

Now we predict the test data. As we can see, the accuracy is 71.5% which is much better than accuracy for SVM.

```
KNN.fit(pca_train, y_train)
y_predict = KNN.predict(pca_test)
acc = accuracy_score(y_test, y_predict)
accuracy_arr_test = np.append(accuracy_arr_test, acc)
```

Then we have the confusion matrix and evaluation metrics.

```
0.715201865617257
```

```
print(confusion_matrix(y_test, y_predict))
```

```
[[   24    71   284   297     0     0     1     0     0     0]
 [   23    39   257   231     9     0     6    18     0     0]
 [  397   641  1126  1612   142    17    52    91    10     1]
 [  392   610  1349  7620   495    10   203   424    27     2]
 [   91   143   539   689  3058     2  1362   174     4     0]
 [    1     3   101   383   106 18212    18    33    11     3]
 [  416     4   228  1249  7755     5 26431   873    38     1]
 [   47    78   115   495   329     4   133  2287     8     0]
 [    0     0     6    18    55     3    30   181    85     0]
 [    0     0     0    30     6     0     1     5     0     2]]
```

Here is the evaluation metrics. In the first row, it's precision of each label. Second row is recall and third row is F1-score. As we can see, the 6th label, was the best predicted label.

```
print(precision_recall_fscore_support(y_test, y_predict))
```

```
(array([0.01725377, 0.02454374, 0.28114856, 0.60361217, 0.25579256,
       0.99775379, 0.93604136, 0.5597161 , 0.46448087, 0.22222222]),
array([0.03545052, 0.06689537, 0.27537295, 0.68451312, 0.50445398,
       0.96507869, 0.71435135, 0.6541762 , 0.22486772, 0.04545455]),
array([0.02321083, 0.0359116 , 0.27823079, 0.64152214, 0.33945718,
       0.98114427, 0.81030703, 0.6032709 , 0.3030303 , 0.0754717 ]),
array([677, 583, 4089, 11132, 6062, 18871, 37000, 3496, 378, 44], dtype=int64))
```

Now we predict the train set.

```
y_predict = KNN.predict(pca_train)
acc = accuracy_score(y_train, y_predict)
accuracy_arr_train = np.append(accuracy_arr_train, acc)
acc
```

```
0.8080711299696021
```

```
print(confusion_matrix(y_train, y_predict))
```

```
[[  372    44   702   759     1     0   122     0     0     0]
 [   51   128   709   700    40     0    24    93     1     0]
 [   73    46  5631  5955   241    22    89   176    31     0]
 [  106    64  6225 24846   967    21   399   720    41     4]
 [   21    11   755  1278 13073     7  2479   524    36     0]
 [    5     3   240   458    73 39168     6    37     5     5]
 [  101     4    60   588  3688     3 51308   238    10     0]
 [    9     9   934  1843   629     8   260  6793     6     0]
 [    0     0    14    52   212     3    56   442   354     0]
 [    0     0     4    89    11     1     0    10     0    15]]
```

```python
print(precision_recall_fscore_support(y_train, y_predict))
```

```
(array([0.50406504, 0.41423948, 0.36866571, 0.67944651, 0.69041458,
       0.99834323, 0.93725225, 0.75202037, 0.73140496, 0.625     ]),
array([0.186     , 0.07331042, 0.45914873, 0.74404815, 0.71892873,
       0.9792    , 0.91621429, 0.64750739, 0.31244484, 0.11538462]),
array([0.27173119, 0.12457421, 0.40896216, 0.71028144, 0.7043832 ,
       0.98867896, 0.92661387, 0.6958615 , 0.43784787, 0.19480519]),
array([2000, 1746, 12264, 33393, 18184, 40000, 56000, 10491, 1133, 130],␣
 ↪dtype=int64))
```

3.3 Random Forest

Random forest is another classifier. this classifier is great for handling high-dimensional data. But on the other hand, it might over fit on data that has imbalanced number of each class.

```python
est = [75, 100]
max_feature = [5, 7]
cri = ["gini", "entropy"]
rf_scores = np.array([])
for a in est:
    for b in max_feature:
        for c in cri:
            RFC = RandomForestClassifier(n_estimators = a, max_features = b,␣
 ↪criterion = c)
            score = cross_val_score(RFC, pca_train, y_train, cv = 5)
            rf_scores = np.append(rf_scores, score.mean())
            print("n_estimator: ", a, " max_feature: ", b, " criterion: ", c, "␣
 ↪is done by score: ", score.mean())
```

```
n_estimator:  75  max_feature:  5  criterion:  gini  is done by score:
0.7540161758310149
n_estimator:  75  max_feature:  5  criterion:  entropy  is done by score:
0.7536055505858754
n_estimator:  75  max_feature:  7  criterion:  gini  is done by score:
0.7538336807454656
n_estimator:  75  max_feature:  7  criterion:  entropy  is done by score:
0.7544838367059861
n_estimator:  100  max_feature:  5  criterion:  gini  is done by score:
0.7537139101836756
n_estimator:  100  max_feature:  5  criterion:  entropy  is done by score:
0.7538165608842907
n_estimator:  100  max_feature:  7  criterion:  gini  is done by score:
0.7537424421472372
n_estimator:  100  max_feature:  7  criterion:  entropy  is done by score:
0.7546150087993355
[0.75401618 0.75360555 0.75383368 0.75448384 0.75371391 0.75381656
 0.75374244 0.75461501]
```

As we can see, n_estimator: 100 max_feature: 7 criterion: entropy, was the best hyper parameter for this data. So we train the classifier with these hyper parameters and then, test the accuracy of predicting.

```
RFC = RandomForestClassifier(n_estimators = 75, max_features = 7, criterion =␣
  ↪"entropy")
score = cross_val_score(RFC, pca_train, y_train, cv = 5)
print(score.mean())
```

```
0.7540960283473546
```

Now we predict the test data using RF classifier. As we can see, it has 72.7% accuracy on predicting the test data which is again better than KNN which was better than SVM.

```
RFC.fit(pca_train, y_train)
y_predict = RFC.predict(pca_test)
acc = accuracy_score(y_test, y_predict)
accuracy_arr_test = np.append(accuracy_arr_test, acc)
acc
```

```
0.7273721031919546
```

This is the confusion matrix. The first four classes were predicted in a wrong way a lot more than the others, especially the fourth class. And this is the reason for 72.7% accuracy.

```
print(confusion_matrix(y_test, y_predict))
```

```
[[   77   212     9   378     0     0     1     0     0     0]
 [   77   110    17   357    11     0     3     3     5     0]
 [  395  1439   337  1651   143    13    28    50    32     1]
 [  472  1428   333  8092   415    10   124   200    57     1]
 [  190   365    19   826  2940     4  1402   291    25     0]
 [    0     1    52   456    81 18241    11    15    13     1]
 [  504     0    47   852  7717     8 27187   614    69     2]
 [   43   177    11   400    87     4    38  2718    18     0]
 [    0     0     8    51    59     1    13    69   177     0]
 [    0     0     0    31     5     0     0     0     1     7]]
```

Here are the evaluation metrics. It is shown that precision, recall and F1-score for the sixth class was very high. we had the same thing in KNN. this was also a shown in the visualizing the distributions in the first pages.

```
print(precision_recall_fscore_support(y_test, y_predict))
```

```
(array([0.04379977, 0.02947481, 0.40456182, 0.61799297, 0.25658928,
       0.99781194, 0.94376367, 0.68636364, 0.44584383, 0.58333333]),
array([0.11373708, 0.18867925, 0.08241624, 0.7269134 , 0.48498845,
       0.96661544, 0.73478378, 0.77745995, 0.46825397, 0.15909091]),
array([0.06324435, 0.05098494, 0.1369362 , 0.6680426 , 0.33561644,
       0.98196598, 0.82626468, 0.72907725, 0.45677419, 0.25      ]),
array([677, 583, 4089, 11132, 6062, 18871, 37000, 3496, 378, 44], dtype=int64))
```

Now predicting for train data. It has a great accuracy of 90.7%. One thing that we can see is that, the classifier has over fitted. That's the reason for low accuracy on test data while high accuracy on train data.

```
y_predict = RFC.predict(pca_train)
acc = accuracy_score(y_train, y_predict)
accuracy_arr_train = np.append(accuracy_arr_train, acc)
acc
```

```
0.9075002423848387
```

Here is the confusion matrix. The classifier has wrongly predicted the third and fourth class. This shows that how close 3th and 4th classes are to other classes. So they predict most wrong.(other classes predict to be in 4th or 3th)

```
print(confusion_matrix(y_train, y_predict))
```

```
[[  601    54   368   972     5     0     0     0     0     0]
 [   39   467   368   863     8     0     0     1     0     0]
 [    0     0  5263  6980     4     4     0    11     1     1]
 [    0     2  2506 30831    14    11     0    23     2     4]
 [    6     9   366  1115 16569     2   111     6     0     0]
 [    0     0   204    99     1 39696     0     0     0     0]
 [    0     0     0     0   209     0 55790     0     1     0]
 [    0     0   506  1331     2     0     0  8652     0     0]
 [    0     0     0     0     1     1     0     0  1131     0]
 [    0     0     0     8     0     0     0     0     0   122]]
```

```
print(precision_recall_fscore_support(y_train, y_predict))
```

```
(array([0.93034056, 0.87781955, 0.54931636, 0.73060973, 0.98548742,
       0.99954676, 0.99801435, 0.99528356, 0.99647577, 0.96062992]),
array([0.3005    , 0.2674685 , 0.4291422 , 0.92327733, 0.91118566,
       0.9924    , 0.99625   , 0.82470689, 0.99823477, 0.93846154]),
array([0.4542706 , 0.41000878, 0.48184939, 0.81572124, 0.94688116,
       0.99596056, 0.99713139, 0.90200167, 0.9973545 , 0.94941634]),
array([2000, 1746, 12264, 33393, 18184, 40000, 56000, 10491, 1133, 130],
 dtype=int64))
```

Between these three classifiers, SVM was the worst, Random Forest was the best, but it did over fit. So combining KNN and Random Forest, might be a good technique to build an ensemble classifier.

3.4 MLP

MLP is capable of learning and representing complex nonlinear relationships between features and the target variable. By using activation functions and multiple layers, MLP can capture intricate patterns in the data, making it suitable for tasks where linear models are insufficient.

MLP requires careful selection and tuning of hyperparameters, such as the number of hidden layers, the number of neurons in each layer, learning rate, and regularization parameters. The performance of MLP can vary significantly depending on these hyperparameters, and finding the optimal configuration can be a time-consuming process.

```
layers = [(30, 50), (60, 40), (50, 40, 30)]
# learning_rate = ['invscaling', 'adaptive']
# activation = ['tanh', 'relu']
mlp_scores = np.array([])
for l in layers:
    mlp = MLPClassifier(hidden_layer_sizes= l, learning_rate= 'adaptive' ,␣
 ↪activation= 'relu')
    score = cross_val_score(mlp, pca_train, y_train, cv=5)
    mlp_scores = np.append(mlp_scores, score.mean())
    print("layer: ", l, "is done by score: ", np.mean(score))
print(mlp_scores)
```

```
layer:  (30, 50) is done by score:  0.7692949664521465
layer:  (60, 40) is done by score:  0.7719241549460399
layer:  (50, 40, 30) is done by score:  0.7771539432130545
```

So the best combination of layers is (50, 40, 30). Now we train the MLP with this hidden layers.

```
mlp = MLPClassifier(hidden_layer_sizes= (60, 45, 35, 27), learning_rate=␣
 ↪'adaptive' , activation= 'relu')
score = cross_val_score(mlp, pca_train, y_train, cv=5)
print(score.mean())
```

```
0.7776216562916491
```

Now predicting the test data using trained MLP classifier:

```
mlp.fit(pca_train, y_train)
y_predict = mlp.predict(pca_test)
acc = accuracy_score(y_test, y_predict)
accuracy_arr_test = np.append(accuracy_arr_test, acc)
acc
```

```
0.7365908759656027
```

So, this classifier has 73.6% accuracy on predicting test data. which is the highest accuracy till now between the classifiers that we have trained. (there is a possibility that if we use a better combination of layers, the accuracy will increase)

```
print(confusion_matrix(y_test, y_predict))
```

```
[[    6    55   107   475    19     1     8     6     0     0]
 [    6    73   111   347    25     0     5    12     4     0]
 [   22   473   455  2913    83    40    18    44    41     0]
 [   38   478   554  9374   231    20    63   321    52     1]
 [   29   169   231  1239  3400     3   700   112   179     0]
 [    0     7    44   454    60 17737    19   538    11     1]
 [  437    23    79  1531  8203     9 26484   123   111     0]
 [    3    56    10   497    28     4     4  2882    12     0]
 [    1     1     9    64    57     0     3    15   228     0]
 [    0     0     1    32     1     1     1     0     2     6]]
```

```
print(precision_recall_fscore_support(y_test, y_predict))
```

```
(array([0.01107011, 0.05468165, 0.28419738, 0.55382252, 0.28082927,
       0.99562167, 0.96993225, 0.71107821, 0.35625   , 0.75      ]),
array([0.00886263, 0.12521441, 0.11127415, 0.8420769 , 0.560871  ,
       0.9399078 , 0.71578378, 0.82437071, 0.6031746 , 0.13636364]),
array([0.00984413, 0.07612096, 0.1599297 , 0.66818733, 0.37426386,
       0.96696287, 0.82369956, 0.76354484, 0.44793713, 0.23076923]),
array([677, 583, 4089, 11132, 6062, 18871, 37000, 3496, 378, 44], dtype=int64))
```

And predicting train data... . which has a medium accuracy.

```
y_predict = mlp.predict(pca_train)
acc = accuracy_score(y_train, y_predict)
accuracy_arr_train = np.append(accuracy_arr_train, acc)
acc
```

```
0.8279295772238096
```

This is the confusion matrix. And again, the fourth class is predicting much more than it has to be.

```
print(confusion_matrix(y_train, y_predict))
```

```
[[  258    48   120  1380     1     0   193     0     0     0]
 [    2   243   131  1329    10     1     3    15    12     0]
 [    6    38  1786 10084   114    38    36    86    75     1]
 [   21    47  1209 30415   477    47   158   917   101     1]
 [   14     2   160  1685 14120    10  2033    88    72     0]
 [    0     6    40   622    47 39260    10     6     7     2]
 [   78     2    27   671  4931     3 50195    68    25     0]
 [    2     5   237  1969    27     9    23  8214     5     0]
 [    0     2    39   167   205     0    15    50   655     0]
 [    0     1     2    85     7     2     3     2     4    24]]
```

```
print(precision_recall_fscore_support(y_train, y_predict))
```

```
(array([0.67716535, 0.61675127, 0.4761397 , 0.62831822, 0.70815989,
       0.99720599, 0.9530274 , 0.86957442, 0.68514644, 0.85714286]),
array([0.129     , 0.13917526, 0.14562948, 0.91081963, 0.77650682,
       0.9815    , 0.89633929, 0.78295682, 0.57811121, 0.18461538]),
array([0.21671567, 0.2271028 , 0.2230409 , 0.74364303, 0.74076017,
       0.98929066, 0.92381452, 0.82399559, 0.6270943 , 0.30379747]),
array([2000, 1746, 12264, 33393, 18184, 40000, 56000, 10491, 1133, 130],
 dtype=int64))
```

3.5 ADAboost

This classifier is much like Random Forest. But the differences is that in ADAboost, trees have depth equal to one (one node with two leaves which here it is 10 leaves(number of classes)) and also trees have priorities. Each tree also have an amount of say (how much they vote worth) while in Random Forest, all tree have only one vote and there is no priority.

```
adaboost = AdaBoostClassifier()
score = cross_val_score(mlp, pca_train, y_train, cv=5)
print(score.mean())
```

```
0.7786881420035352
```

```
adaboost.fit(pca_train, y_train)
y_predict = adaboost.predict(pca_test)
acc = accuracy_score(y_test, y_predict)
accuracy_arr_test = np.append(accuracy_arr_test, acc)
acc
```

```
0.6014307924014963
```

```
print(confusion_matrix(y_test, y_predict))
```

```
[[    4     0    14   601    57     0     1     0     0     0]
 [    7     0    17   513    40     0     6     0     0     0]
 [   67     0    41  3421   486     6    68     0     0     0]
 [   96     0    66  8127  2699     0   144     0     0     0]
 [   12     0   123  1529  4396     0     2     0     0     0]
 [    4     0    15   326   360 18141    25     0     0     0]
 [    0     0   109  3459 14624     0 18808     0     0     0]
 [   17     0     2   802  2665     5     5     0     0     0]
 [    0     0     0     0   378     0     0     0     0     0]
 [    0     0     0    34    10     0     0     0     0     0]]
```

It is the worst confusion matrix ever. No data was predicted to be 2nd, 8th, 9th or 10th class at all. How is that even possible...

```python
print(precision_recall_fscore_support(y_test, y_predict))
```

```
(array([0.01932367, 0.        , 0.10594315, 0.43201148, 0.17095081,
       0.99939401, 0.98683037, 0.        , 0.        , 0.        ]),
array([0.00590842, 0.        , 0.0100269 , 0.73005749, 0.72517321,
       0.96131631, 0.50832432, 0.        , 0.        , 0.        ]),
array([0.00904977, 0.        , 0.01831993, 0.54281325, 0.2766781 ,
       0.97998541, 0.67100733, 0.        , 0.        , 0.        ]),
array([677, 583, 4089, 11132, 6062, 18871, 37000, 3496, 378, 44], dtype=int64))
```

As we can see, it has the worst accuracy on test data between the classifiers from before.

```python
y_predict = adaboost.predict(pca_train)
acc = accuracy_score(y_train, y_predict)
accuracy_arr_train = np.append(accuracy_arr_train, acc)
acc
```

```
0.7042391682492971
```

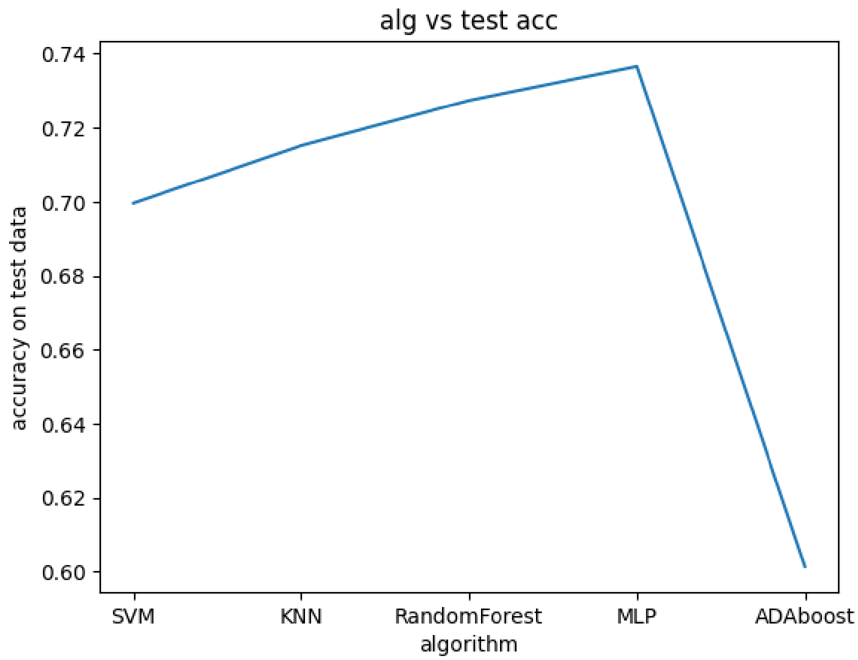
```python
print(confusion_matrix(y_train, y_predict))
```

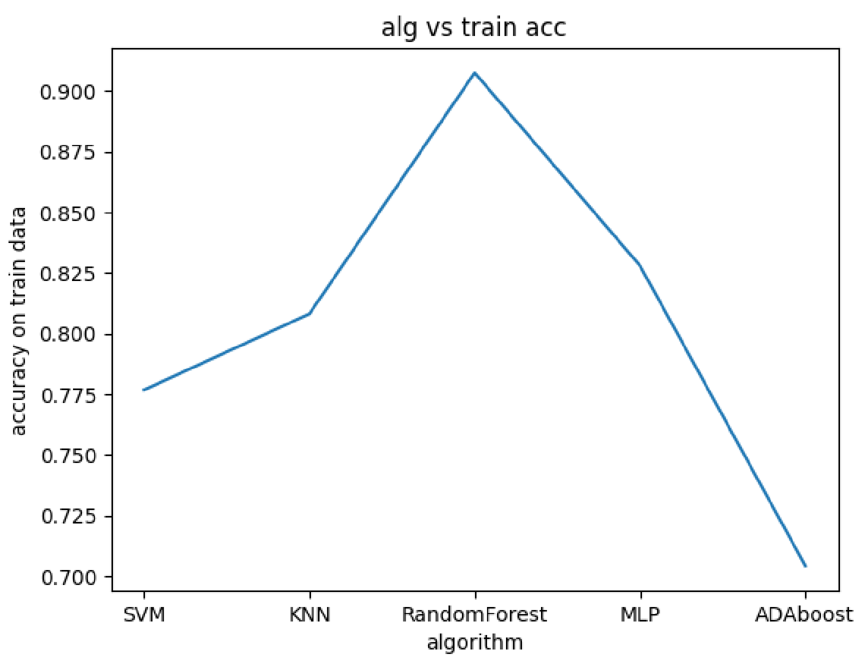
```
[[   14     0     1  1985     0     0     0     0     0     0]
 [   33     0     7  1493   184     0    29     0     0     0]
 [  176     0    62 11012   828    14   172     0     0     0]
 [  309     0   145 28398  4123    13   405     0     0     0]
 [   35     0   141  2469 15295    81   163     0     0     0]
 [    8     0    22   633   222 39098    17     0     0     0]
 [    0     0   121  4355 10909     0 40615     0     0     0]
 [   46     0     6  3431  6982     6    20     0     0     0]
 [    0     0     0     0  1133     0     0     0     0     0]
 [    0     0     0   114    16     0     0     0     0     0]]
```

```python
print(precision_recall_fscore_support(y_train, y_predict))
```

```
(array([0.02254428, 0.        , 0.12277228, 0.52696233, 0.38534213,
       0.99709273, 0.98054127, 0.        , 0.        , 0.        ]),
array([0.007     , 0.        , 0.00505545, 0.85041775, 0.84112407,
       0.97745   , 0.72526786, 0.        , 0.        , 0.        ]),
array([0.01068295, 0.        , 0.00971102, 0.65071091, 0.52854378,
       0.98717366, 0.8338038 , 0.        , 0.        , 0.        ]),
array([2000, 1746, 12264, 33393, 18184, 40000, 56000, 10491, 1133, 130],
 ↪dtype=int64))
```

Plotting the results till now.





So the best accuracy on test data is for MLP and the best accuracy on train data is for Random Forest (because it was probably over fitted).

So MLP model was the best model. Let's try it with a different hidden layers.

```
mlp = MLPClassifier(hidden_layer_sizes= (60, 45, 35, 27, 20, 15),␣
 ↪learning_rate= 'adaptive' , activation= 'relu')
score = cross_val_score(mlp, pca_train, y_train, cv=5)
print(score.mean())
```

```
0.77828888430068
```

```
mlp.fit(pca_train, y_train)
y_predict = mlp.predict(pca_test)
acc = accuracy_score(y_test, y_predict)
accuracy_arr_test = np.append(accuracy_arr_test, acc)
acc
```

```
0.7492226594762669
```

```
print(confusion_matrix(y_test, y_predict))
```

```
[[    0    39    31   602     0     2     3     0     0     0]
 [    0    33    31   509     3     0     0     3     4     0]
 [  283   346   518  2758    69    18     8    32    57     0]
 [  261   323   485  9396   222    24    83   234   101     3]
 [   34    58   115  1527  3318     0   592   149   269     0]
 [    0     3    74   503    77 18181     1     9    14     9]
 [  254     5   137  1023  8037     5 27118   273   147     1]
 [   36    43    35   483    19     3     4  2853    20     0]
 [    0     0     2    47    35     0     5    28   261     0]
 [    0     0     2    32     2     0     0     0     1     7]]
```

```
y_predict = mlp.predict(pca_train)
acc = accuracy_score(y_train, y_predict)
accuracy_arr_train = np.append(accuracy_arr_train, acc)
acc
```

```
0.8285113008366555
```

```
print(confusion_matrix(y_train, y_predict))
```

```
[[  202     8   201  1271     1     0   317     0     0     0]
 [   35   209   214  1234    13     0     3    17    19     2]
 [    1    19  2114  9742   121    35    56    76   100     0]
 [    8    18  1634 29892   481    48   343   798   163     8]
 [    5     2   248  1567 13955     1  2134   165   104     3]
 [    0     1    65   645    45 39202    10     7    12    13]
 [    7     2    41   437  4571     3 50794   111    34     0]
 [    0     5   345  1969    33     6    16  8111     6     0]
 [    0     0     8   131   150     1     9    64   770     0]
 [    0     0     5    88     6     0     0     3     5    23]]
```

conclusion: MLP classifier gets better by tuning it's layers.

3.6 implementing an stacking method

stacking method is a method that adds some new columns to data set which these columns are the votes that some classifiers have given to each data. After building our new data set, there will be a classifier that classify this new data. It's like combining classifiers together.

First of all, we again reduce the dimension of the 55 dimensional data that we had using PCA. Reducing dimension in such a way that 98% of data will remain and only 2% will be lost (at most).

```
pca = PCA(n_components=0.98)
pca_train = pca.fit_transform(pca_train)
pca_test = pca.transform(pca_test)
```

First of all, we classify the data using KNN and add a new column to our data. KNN might be a good choice of classifier in first step of stacking method because it will separate data kinda well for the first time and the label it gives to it might be useful in next classifiers. So train the KNN classifier:

```
KNN = KNeighborsClassifier(n_neighbors= 7)
score = cross_val_score(KNN, pca_train, y_train, cv=5)
print(score.mean())
```

```
0.7340665172324664
```

```
KNN.fit(pca_train, y_train)
y_predict = KNN.predict(pca_test)
pca_test = np.concatenate((pca_test, y_predict.reshape(-1, 1)), axis=1)
print(accuracy_score(y_test, y_predict))
```

```
0.7114123305640577
```

```
y_predict = KNN.predict(pca_train)
pca_train = np.concatenate((pca_train, y_predict.reshape(-1, 1)), axis=1)
print(accuracy_score(y_train, y_predict))
```

```
0.8074437809753566
```

So, as we can see, new column was added. This new feature predicted the test data 71.1% right. It is not a great accuracy but let's see what happens.

Also the accuracy of prediction for train data is 80.7%, which still is not high enough.

first KNN was the same. But in the road we are going through, we are looking for some improvement in predicting accuracy for our test data as for train data.

Let's train another classifier with our new data...

The next classifier that might be good for our new data, is Random Forest. because of it's good separating the data. And of course this new feature which is KNN's vote for label, would help a lot to Decision trees for labeling the data. So we train a Random Forest classifier...

```
RFC = RandomForestClassifier(n_estimators = 80, max_features = 7, criterion =␣
 ↪"entropy")
score = cross_val_score(RFC, pca_train, y_train, cv = 5)
print(score.mean())
```

```
0.7660898353843465
```

Predicting test data...

As we can see the accuracy hasn't improved much. why is that for??

```
RFC.fit(pca_train, y_train)
y_predict = RFC.predict(pca_test)
pca_test = np.concatenate((pca_test, y_predict.reshape(-1, 1)), axis=1)
print(accuracy_score(y_test, y_predict))
```

```
0.7276757518340378
```

Predicting the train set...

accuracy hasn't improved! :|

```
y_predict = RFC.predict(pca_train)
pca_train = np.concatenate((pca_train, y_predict.reshape(-1, 1)), axis=1)
print(accuracy_score(y_train, y_predict))
```

```
0.9075002423848387
```

Okay, let's be patient. there might be something good at the end! Let's keep going...

So this is our last shot... using best classifier of all the time. For sure you know which one I am talking about. It is time for MLP...

So we train a MLP classifier for our new data which has two new columns, that one of them is the vote from KNN and the other one is the vote from Random Forest for label of each data.

```
mlp = MLPClassifier(hidden_layer_sizes= (60, 45, 35, 27, 20, 15, 25),␣
 ↪learning_rate= 'adaptive' , activation= 'relu')
score = cross_val_score(mlp, pca_train, y_train, cv=5)
print(score.mean())
```

```
0.9019453302969277
```

```
mlp.fit(pca_train, y_train)
y_predict = mlp.predict(pca_test)
print(accuracy_score(y_test, y_predict))
```

```
0.728489530194821
```

```
y_predict = mlp.predict(pca_train)
print(accuracy_score(y_train, y_predict))
```

0.9069983631894423

The accuracy for test and train was different. Accuracy for test has decreased a little bit, on the other hand, accuracy for train data has improved much... it might be over fitted...

now what if we use KNN for last classifier?

```
KNN = KNeighborsClassifier(n_neighbors= 7)
score = cross_val_score(KNN, pca_train, y_train, cv=5)
print(score.mean())
```

0.884105819567891

```
KNN.fit(pca_train, y_train)
y_predict = KNN.predict(pca_test)
print(accuracy_score(y_test, y_predict))
```

0.7242141573142885

```
y_predict = KNN.predict(pca_train)
print(accuracy_score(y_train, y_predict))
```

0.899093765861949

So, these accuracy for sure was lower than when we used MLP for the last classifier. But that one itself wasn't high enough that we expected. The best classifier for now, was MLP alone itself.

3.7 MLP'

So again, we train the MLP model with different layers to check for better results

```
mlp = MLPClassifier(hidden_layer_sizes= (50, 30, 20, 10), learning_rate=␣
 ↪'adaptive' , activation= 'relu')
score = cross_val_score(mlp, pca_train, y_train, cv=5)
print(score.mean())
```

0.7744677181135609

```
mlp.fit(pca_train, y_train)
y_predict = mlp.predict(pca_test)
print(accuracy_score(y_test, y_predict))
```

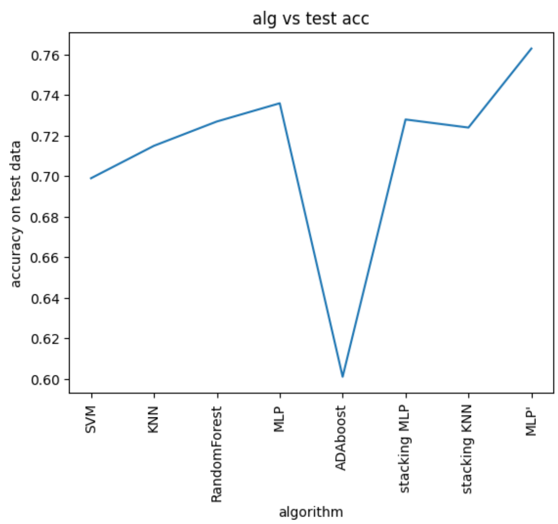
0.76629985910703

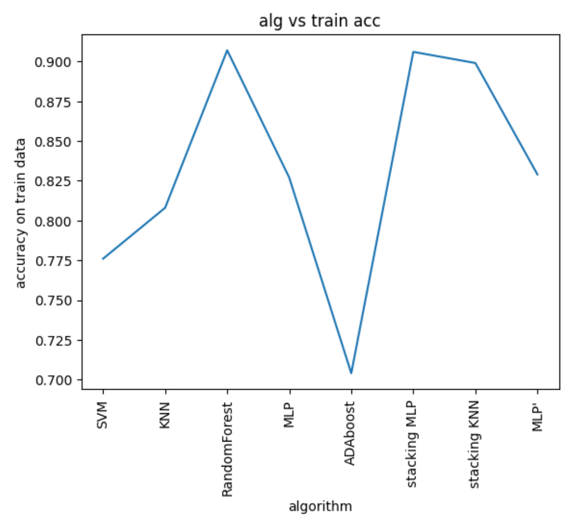
```
y_predict = mlp.predict(pca_train)
print(accuracy_score(y_train, y_predict))
```

0.8213595223022567

So it was the highest accuracy that we get, 76.6% on test and 82.1% on train data. again there might be better results using MLP with different layers and learning rate possibly. But as it is shown, it probably won't get higher than nearly 78%.

The article did some sort of things that we didn't (because I still think it's not right to do), which were deleting the rare labels, that this one doesn't change anything much I guess. The other thing was over sampling, means adding some classes again to the data set. This decision was made by exploring test set, which we now is not a right thing to do. We can't explore test set. The other one was using a test set for validation while training, again this decision was made by exploring test set. which again is not a right thing to do. But these two last things, for sure would improve the results, but this improving is not real, I mean in real world, this might even label the categories worst than other classifiers (because of the data, none of them would classify very good, only like 80% of them would be true which is not a high enough accuracy). So the results of the article wasn't true in real world in my opinion and those are not a great thing to do if we like to build a useful classifier for real world.





So by looking at the figures, we conclude that MLP' was better than other classifiers in predicting a test data while it's accuracy on predicting train data wasn't that high (it didn't over fitted and was just in the best place).

The second best was the first MLP. And the third best was stacking MLP (as I said that MLP is the best). It over fitted a little bit compared to those alone MLPs.

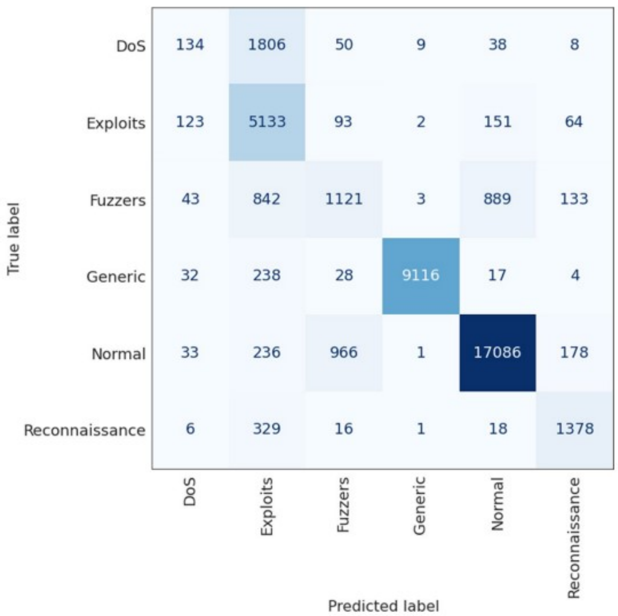
The fourth best classifier was Random Forest itself, it over fitted a little bit but still wasn't that bad on predicting test data.

And then, it is the stacking KNN, which was close to stacking MLP.

As we can see, the MLP' is the best classifier for this data. We may change the layers again and get some better results. But let's leave it here and compare our results with the article's.

4 Comparison

So let's first compare the confusion matrix for last MLP (MLP') and the articles results.




```
[[    0     0    11   664     0     0     2     0     0     0]
 [    0    24    17   533     1     0     1     1     6     0]
 [    0    14   301  3589    64    13    26    37    45     0]
 [   12    11   412 10161   138    28    95   198    77     0]
 [    0    31    28  1873  2413    16  1188   208   304     1]
 [    0     4    97   447    39 18253     6    13    10     2]
 [  396     2    17  1231  5827    12 28894   471   150     0]
 [    0     2     8   589    31     1    16  2817    32     0]
 [    0     1     6    48    45     4     3    48   223     0]
 [    0     0     1    36     0     0     1     0     1     5]]
```

As we can see, because we didn't drop those rare labels, our matrix is 10*10 while the articles is 6*6. So let's compare them as the accuracy for articles is 84.24% and ours is just 76.6%, the articles matrix is much better (look at Normal). our fourth class was the most predicted class why the predicted data, don't belong to this class.

	Precision	Recall	F1 Score	FPR	Accuracy
DoS	0.3612	0.0655	0.1109	0.0062	84.24%
Expl.	0.5980	0.9222	0.7255	0.0993	
Fuzz.	0.4930	0.3698	0.4226	0.0309	
Gene.	0.9982	0.9662	0.9820	0.0005	
Norm.	0.9388	0.9236	0.9311	0.0510	
Recon.	0.7807	0.7883	0.7845	0.0100	
Weighted Avg.	0.8360	0.8424	0.8285	0.0403	

```
(array([0.        , 0.26966292, 0.33518931, 0.5300193 , 0.2819584 ,
        0.99596224, 0.95574226, 0.74268389, 0.2629717 , 0.625     ]),
 array([0.        , 0.04116638, 0.07361213, 0.91277398, 0.39805345,
        0.96725134, 0.78091892, 0.80577803, 0.58994709, 0.11363636]),
 array([0.        , 0.07142857, 0.12071386, 0.67062667, 0.33009576,
        0.98139685, 0.85953118, 0.77294553, 0.36378467, 0.19230769]),
 array([  677,   583,  4089, 11132,  6062, 18871, 37000,  3496,   378,
           44], dtype=int64))
```

the first row, is precision for our 10 labels, the second row is recall and the third row is F1 score. And accuracy for our classifier is 76.6% while it is 84.24% for articles.

The 6th, 7th and 8th label, was the best predicted labels, using our classifier. while Expl, Gene and Norm was best predicted labels using articles.