Wavelet:Documentation

Objective: Develop a classification model to identify users based on biometric data

Pre-Processing:
The data set contains :
Number of Rows: 224375
Number of Columns: 26
Number of Features: 22
Number of Classes: 10

The data set contains a lot of missing data for individual features. I wrote a script to calculate in terms of percentage individual features. Here is the list. Statistically, It is advised that if a feature has more than 5% of missing instances it should be dropped. As the dataset increases it leads to less reliable results and bias into the dataset.

Feature Name	Percentage (Missing)	Feature Name	Percentage(Missing)
feat1	14%	feat12	6.69%
feat2	< 1%	feat13	6.69%
feat3	< 1%	feat14	6.69%
feat4	9%	feat15	6.69%
feat5	0%	**feat16**	**52.62%**
feat6	18%	feat17	18.78%
feat7	9.4%	feat18	18.78%
feat8	**47.0%**	**feat19**	**29.94%**
feat9	4.13%	feat20	12.54%
feat10	8.6%	feat21	0.96%
feat11	6.69%	**feat22**	**29.95%**

Since most of the features are missing more than 5% of the features. I prefered to take a different cutoff and Removed features with missing percentage of more 25%.

After the initial preprocessing **I removed 4 features (feat8, feat16, feat19, feat22).** The other rationale behind removing these is because they tend to create correlation among variables where none exist.

Removing Variables with High Correlation:

Since some of the classifiers have the assumption that each individual features is independent. Features with high correlation tend to void the assumptions and hence compromise the validity/ performance of the model. The other reason being that the increase the redundancy in the dataset. Third, they add to the computation and time taken in training the model.

Keeping with that I removed the features which had approximately 75% correlation. The following table records feature with high correlation. The cor.test in R does not take the na values and hence I had to use the with function to remove the na values. In sklearn we calculate among the features using the following command.

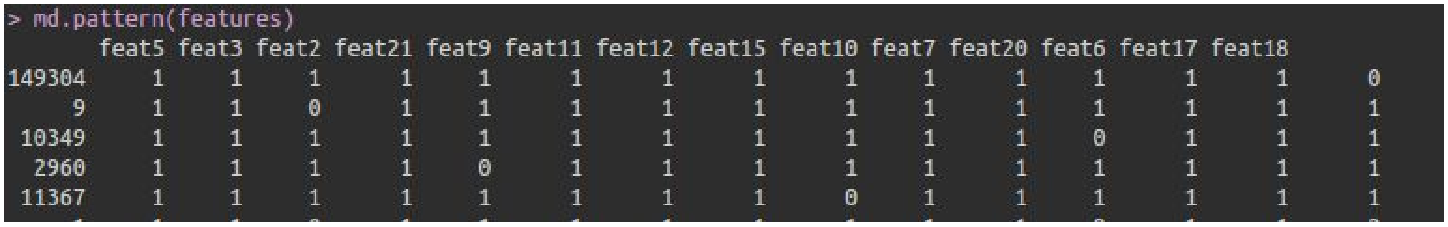
```
print clf.covariance_
```

Feature Name	Feature Name	Correlation Factor
feat1	feat2	0.83
feat3	feat4	-0.44
feat3	feat7	-0.42
feat3	feat8	-0.43
feat3	feat16	-0.77
feat4	feat5	0.54
feat4	feat7	0.79
feat4	feat17	0.51
feat4	feat18	0.67
feat4	feat21	0.674
feat6	feat3, feat4, feat5	0.30
feat7	feat21	0.548
feat11	feat13	0.99
feat14	feat15	0.73
feat17	feat21	0.59
feat18	feat7	0.50

After this step I further dropped **feat13, feat4, feat1 and feat14** because of extremely high correlation. I removed these features carefully based on their the max and min values and retained features which were closer to normalization.

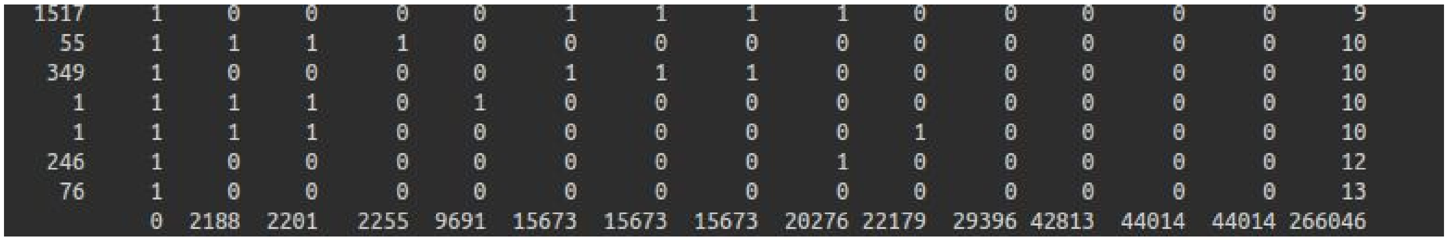
In addition to that a further preprocessing step could be taken and rows could be removed missing more than 5 of these features. But it turns out we have instances which are missing 13 features and less and doing that will results in an extremely small dataset.

R command (from mice package)
// md.patterns(data)



```
> md.pattern(features)
       feat5 feat3 feat2 feat21 feat9 feat11 feat12 feat15 feat10 feat7 feat20 feat6 feat17 feat18
149304     1     1     1      1     1      1      1      1      1     1      1     1      1      1      0
     9     1     1     0      1     1      1      1      1      1     1      1     1      1      1      1
 10349     1     1     1      1     1      1      1      1      1     1      1     0      1      1      1
  2960     1     1     1      1     0      1      1      1      1     1      1     1      1      1      1
 11367     1     1     1      1     1      1      1      1      0     1      1     1      1      1      1
```

.
.
.



```
  1517     1     0     0      0     0      1      1      1      1     0      0     0      0      0      9
    55     1     1     1      1     0      0      0      0      0     0      0     0      0      0     10
   349     1     0     0      0     0      1      1      1      0     0      0     0      0      0     10
     1     1     1     1      0     1      0      0      0      0     0      0     0      0      0     10
     1     1     1     1      0     0      0      0      0      0     1      0     0      0      0     10
   246     1     0     0      0     0      0      0      0      1     0      0     0      0      0     12
    76     1     0     0      0     0      0      0      0      0     0      0     0      0      0     13
           0  2188  2201   2255  9691  15673  15673  15673  20276 22179  29396 42813  44014  44014 266046
```

The above picture shows us the degree to which we are missing features. The last column tell us the number of feature we are missing and the first columns tell us the number of instances we are missing.

Prints rows with number of features missing in ascending to descending order in a table format.
This further gives us an idea of how many instances are missing a high number of features. Since imputing for these values create a great bias and reduce correlation among the variables(sounds like a paradox, a little correlation is indeed good).

Taking care of Missing Values: Imputation Techniques

1. **Replacing Missing values with mean/median grouped by individual user.**
   Pros: This takes advantage of the individual user biometric bias and would be one of more plausible Choice. `print clf.means_` (This command get all the individual means according to user's)
   Cons: We don't know the test labels and can't use this strategy to impute missing values for the same.

2. **Replacing missing values with mean of the entire dataset.**
   Pros: This is the most plausible and simple choice given the conditions.
   Cons: This will induce a strong bias of the whole dataset and increase existing correlation.

3. **Remove instances with missing values:**
   Pros: Will improve the classifier performance considerably.
   Cons: But on closer inspection we have so many instances missing an average of 6 features
   And doing this would remove most of the training data as I found it using the 'md.pattern(data)'

Command. Hence, I could not use this approach at all.

4. **Use Linear Regression to predict missing values**
   Pros: This approach is good in prediction tasks and sometimes in classification but given
   Cons: This would amplify the inherent bias of the whole dataset.That we are missing
   or less features, it would be really complicated and not worth it.

There were many more approaches but due to such huge amount of missing data for multiple features, they would not be plausible. The other reason is we don't know why is the data missing.

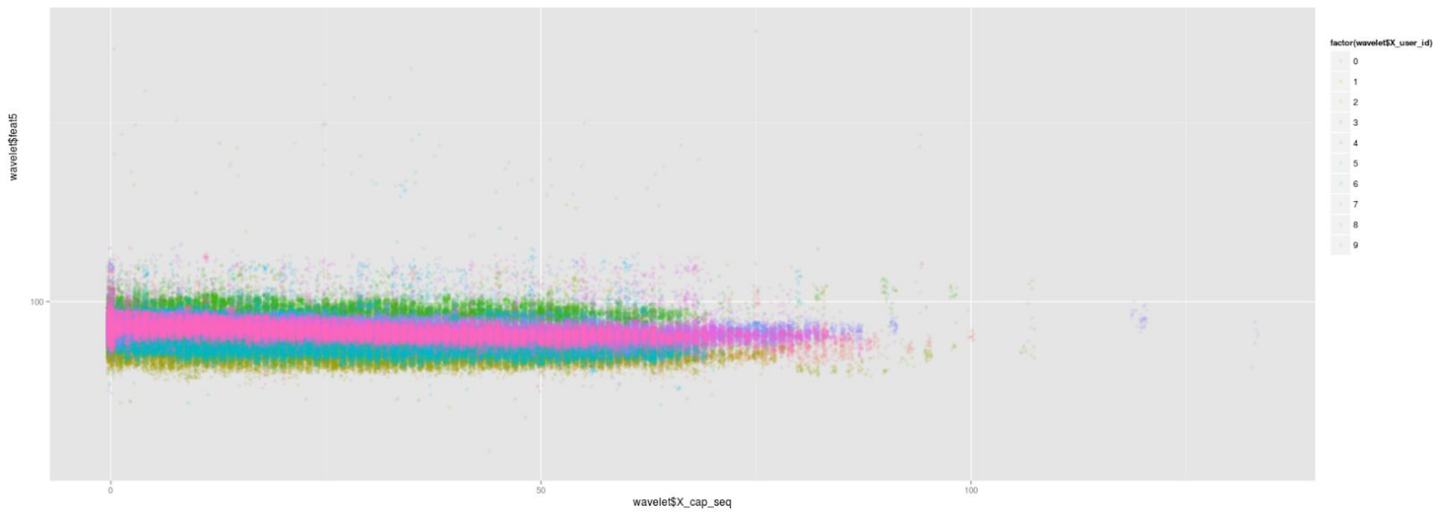
For, all the reasons above I chose the Second approach. Further, I did some exploratory analysis and we can actually visualize the features and how distinct they are for the users and it will give us an idea about the algorithms that we can choose based on the type of decision boundary they are able to produce.

Exploratory Analysis of the features: Plotting features factored by the user_id's to see which features are more distinguished for individual user ids. So plotted to see which features have a good visible boundary.

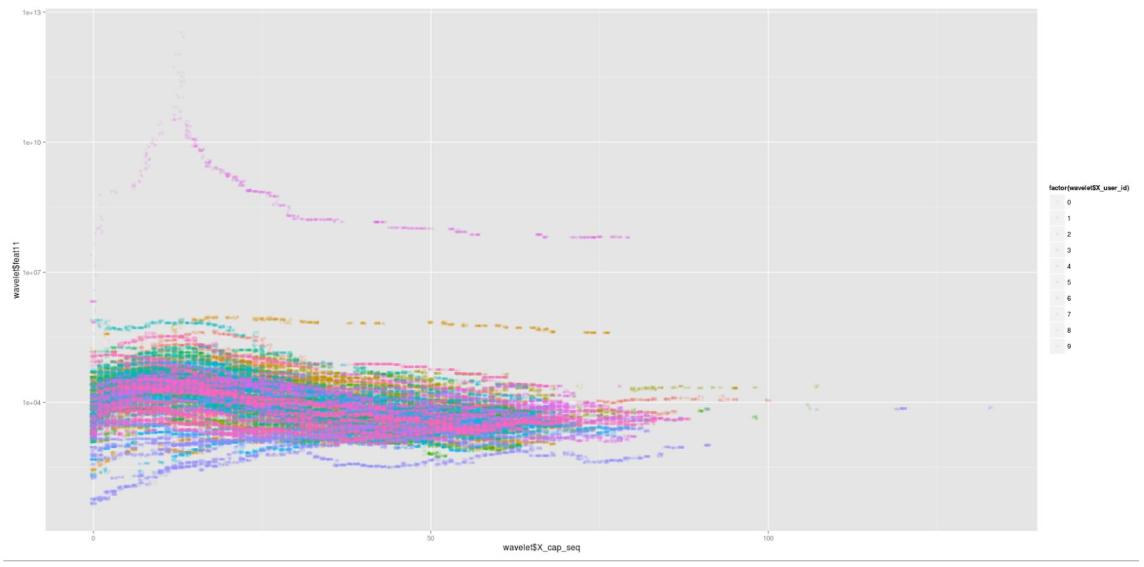
In addition to that the dataset is mostly balanced for individual user ids. We can see that by running the command. `print clf.priors_`

```
[ 0.11019729  0.09767568  0.06885463  0.08524655  0.11354824  0.0980171   0.11220359  0.11332057
0.07751304  0.12342324]
```

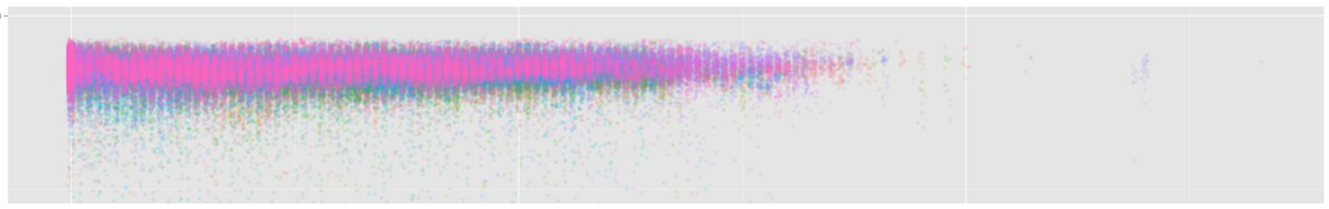
It's a pretty even distributions of all the userid's expect for userid 2 and userid 8.



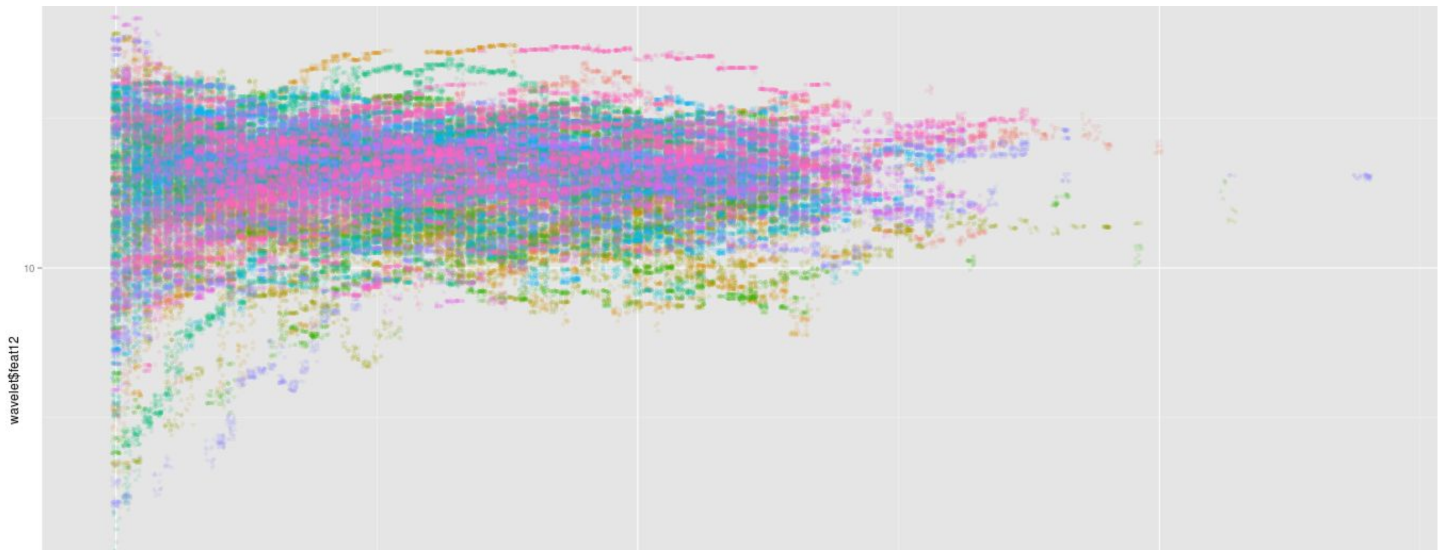
The graph is for "**feat5**" and we can see different boundaries in color according to user_id and we can see that feat5 will be good feature for classification.



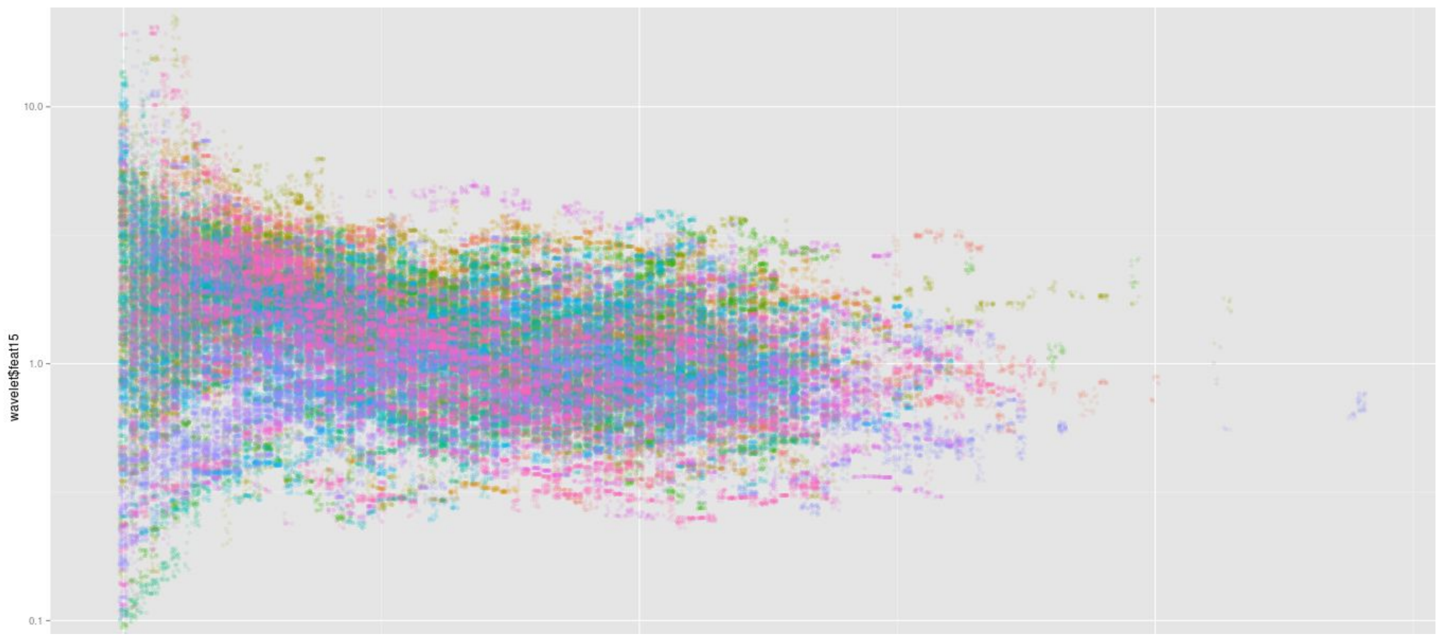
“feat11”



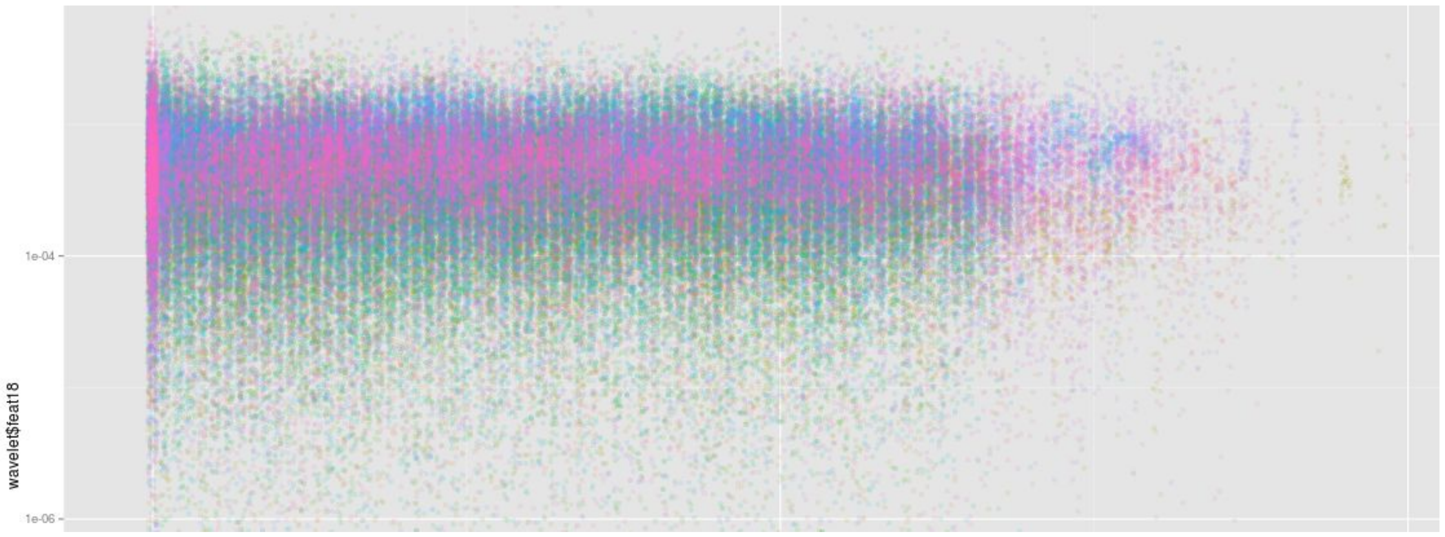
“feat6”



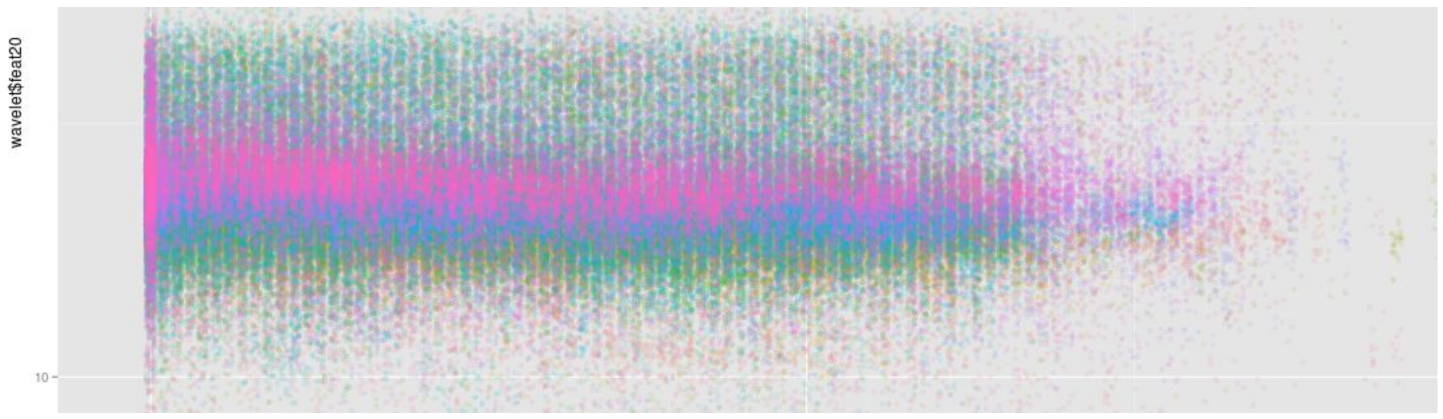
“feat12”



“feat15”



“feat18”



“feat20”

“ ggplot(data = wavelet, aes(x = wavelet$X_cap_seq ,y = wavelet$feat3, color = factor(wavelet$X_user_id))) + geom_point(alpha = 0.1, position = position_jitter(h = 0)) + scale_y_log10() “

Statistically given a huge dataset all the features are shown statistically significant with high p-value but on closer analysis of looking at t-value and also using decision tree classifier feature importance I got to know that the most important features are (feat2, feat5, feat11, feat12, feat15)

This process further helps us recognizing and get a feel for the data.

Standardization of the data:
I used the sklearn MinMaxScaler to normalize the data. This step in important since it helps in converging of algorithms and also avoid certain features with high/extreme values to dominate other features.

Sampling data:
I didn’t used all the data at once and used small amount of training data and observed that for all the algorithms that as the data increased the performance of the classifier increased.

Splitting training and test data:
I used the train_test_split with a random seed of 100 to split into training and test data.

Classifier Performance:

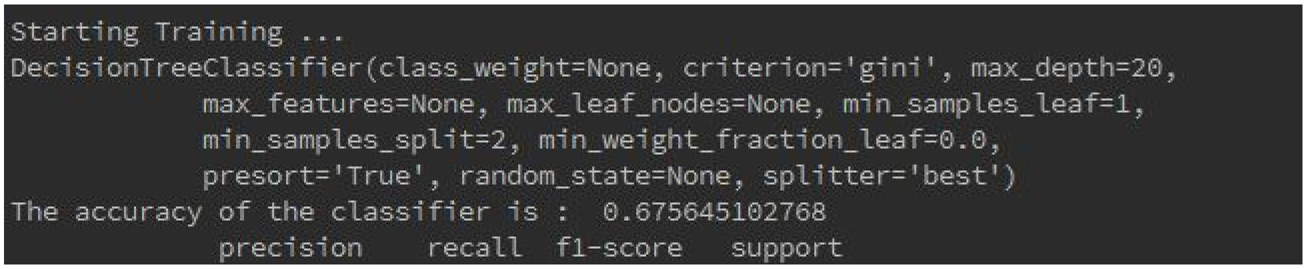
I started with all the simple classifiers and then moved on to use GridSearchCV to tune the classifier parameters and then to using ensemble methods and finally to pipelining classifier.

Simple Classifiers → GridSearchCV → Ensemble Methods → Pipelining Classifiers.

Here is the order I started with.

Simple classifiers which crossed the cutoff accuracy, the parameters for them were tuned using GridSearchCV.

For example for the DecisionTreeClassifer:



```
Starting Training ...
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=20,
            max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort='True', random_state=None, splitter='best')
The accuracy of the classifier is :  0.675645102768
             precision    recall  f1-score   support
```

I did this GridSearchCV for all the remaining algorithms for the pipelining steps to build the best pipeline. In code I have done this step myself and is hidden and have put the best classifier in the code.

The following table contains the simple classifiers and the ensemble methods whose parameters have been fine tuned using the GridSearchCV. This step in not in the code but I ran it for the individual classifiers and the parameter grid for each classifier will be commented in the code just for your reference.

Results:

Classifier	Precision	Recall	Accuracy	F-Score
GaussinaNB	0.15	0.13	0.12	0.04
LogisticRegression	0.35	0.33	0.351	0.32
KNearNegClassifier	0.45	0.45	0.448	0.48
AdaboostClassifier	0.48	0.49	0.485	0.48
DecisionTreeClassifier	0.68	0.68	0.69	0.68
RandomForestClassifier	0.78	0.78	0.774	0.78
ExtraTreeClassifier	0.80	0.80	0.803	0.80
SupportVectorMachine	0.38	0.37	0.368	0.37
LDA	0.41	0.42	0.416	0.40
QDA	0.44	0.40	0.397	0.38

Some classifiers such as logistic regression and Support Vector Machines work well with scaled features but most of the tree classifiers and LDA and QDA have better performance with the original features. Hence, by default I used the original features but created the scaled and pca transformed features.

I used the pipelining for pca features and used the transformed pca features and plug them into the SVM. The code for pipelining code is commented since it takes more than an hour to train and doesn't give any better accuracy. The pipelining and the ensemble step could have been better explored. The current version of sklearn does not support neural network. This is the only one I wanted to try but I think that, it would have given comparable or slightly less performance than tree classifiers.

The code is pretty much self contained.
wavelet.R: preprocesses the file, feature selection, outputs test.csv used by classifiers.
Wavelet.py: training the classifiers, parameter tuning, pipelining and generate the results.
Test.csv: preprocessed file the the wavelet.R produces.