|  |
| --- |
| Name: Samrat Bikram Shah |

**Machine Learning Solution for Biometric Recognition Task**

**Introduction:**

The development of Machine Learning and Artificial Intelligence over the years has led to its application in various fields. One of its applications is the facial recognition system. The Facial Recognition system allows the identification of different people using various distinct facial features. Features such as eyes, nose, lips, jaw, etc and overall structure of the face help to identify and recognize different types of faces. Facial Recognition algorithms must be effective enough to detect and classify different faces under various conditions such as lighting, angles, change in hairstyles, etc. with high accuracy. Facial Recognition systems have shown to have many uses in various sectors such as government facilities, airports, businesses and even in smartphones. For instance, some of the uses of Facial Recognition systems can be to prevent crimes and find missing persons. Criminal Databases can be updated and they can be identified easily if criminals enter any establishment. Similarly, if the database has the pictures of the missing person, it can easily identify if the person is found in any public place. (FaceFirst Face Recognition Software, 2020)

In this assignment, it is expected to create a facial recognition system efficient and effective enough to recognize and classify 30 different types of faces on the given 1500 images with the highest possible accuracy. Various techniques such as MLP, K-Fold, Adaboost, etc are expected to be used on the given scripts to achieve the optimal error-free results.

By providing various data of the same type under different conditions it allows machines to learn to identify the faces accurately. Here, using Principal component Analysis dataset are made more efficient for training and then trained using various classifying techniques and finally Adaboosting.

**Designing a Solution:**

In the given model, there are two files used to train the facial recognition system, one processes the images while the other classifies the images. Process image converts all the images in the given dataset into its respective matrices which can be used by the classify file to apply the necessary machine learning techniques.

Here, Principal Component Analysis is applied in the default scripts for the optimization of the dataset. It finds the patterns in the features of the images in the dataset and helps to remove redundancy. It makes data easy to be used by the machine to learn. PCA is a dimensionality reduction technique for classification problems. To apply optimal Principal Component Analysis, the number of principal components are changed throughout the whole program making sure there is no overfitting and the variance of the training and testing data are low as possible.

Firstly, the MultiLayer Perceptron model is applied in the solution for classification of images which provides multiple layers to train the machine. It consists of an input layer, the hidden layer and an output layer. Number of hidden neurons in the hidden layer are changed throughout the experiments for the most efficient accuracy.

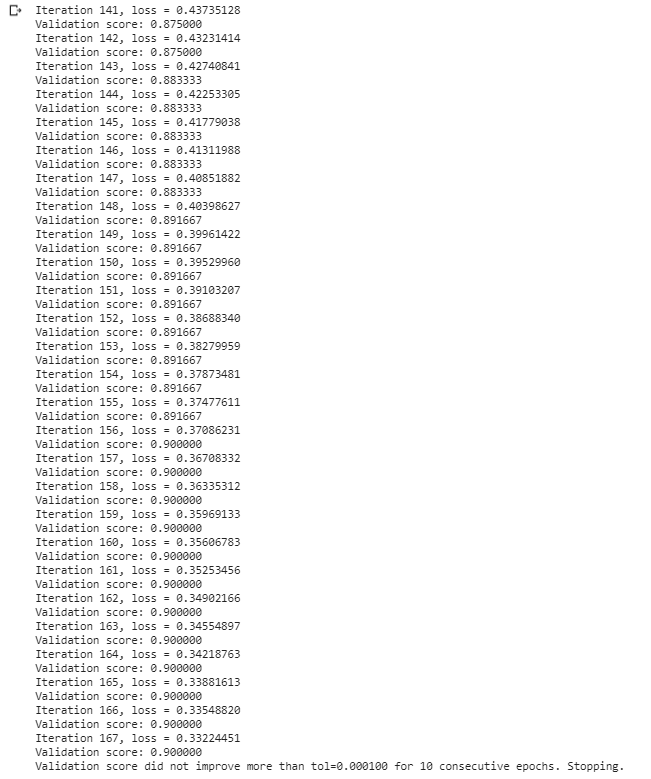
As for the testing, the default script splits the training and testing data but it fails to get high accuracy if data with new patterns are provided. Here, the ANN can recognize patterns given for training with very high accuracy but not those which might appear in the future so the model is **overfitted**. Hence, K-fold cross validation is applied in the experiments. This allows all the dataset to be used as both training and the testing data which helps to test the accuracy of the model with new dataset patterns. This takes error from every fold and calculates the mean error. However, it suffers from the problem of overlapping, i.e. the same data being used in multiple folds. Since it's an individual model, it still has a high chance to suffer from high biases and variances. The problem of overfitting also still exists in a way or other, so Adaboost, which uses multiple models to make predictions, is adapted into the experiments as well.

**Experiments:**

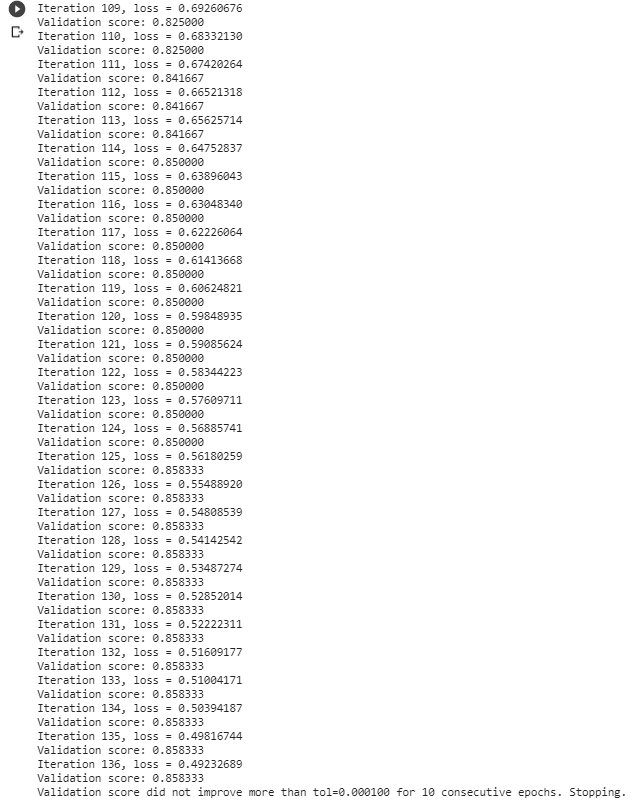
**Initial Experiment:**

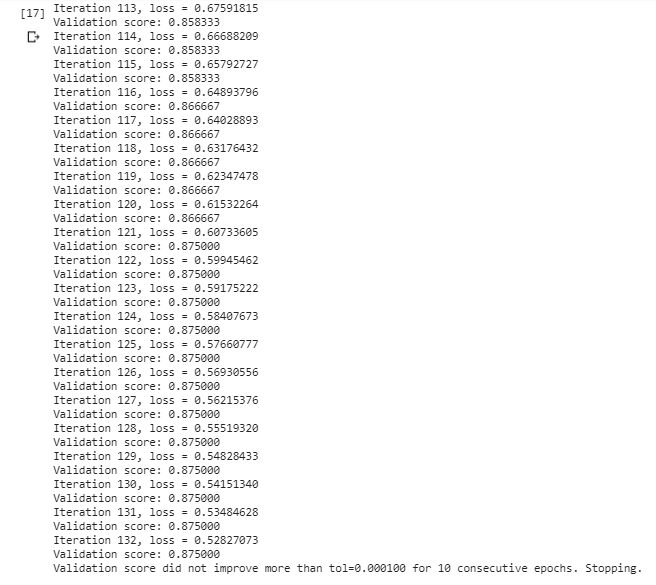
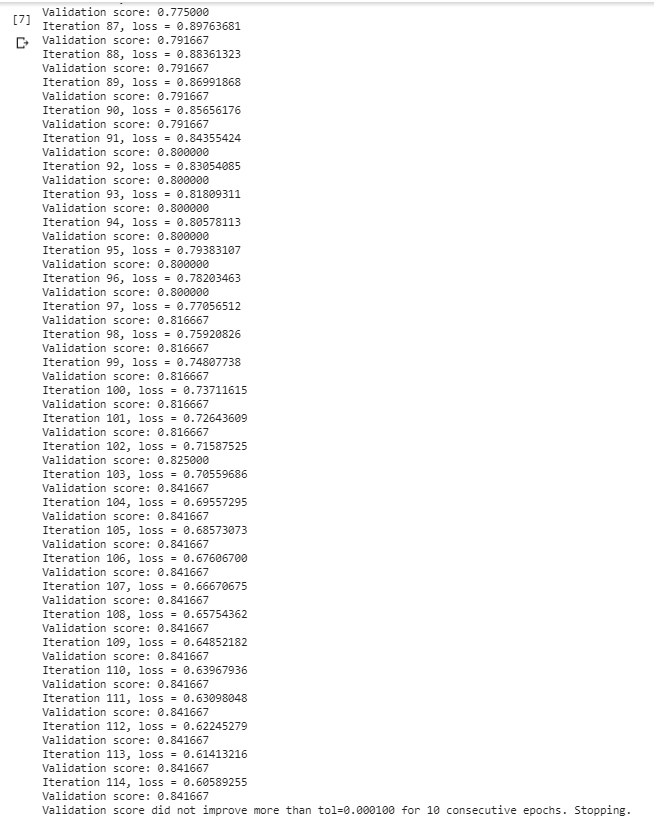
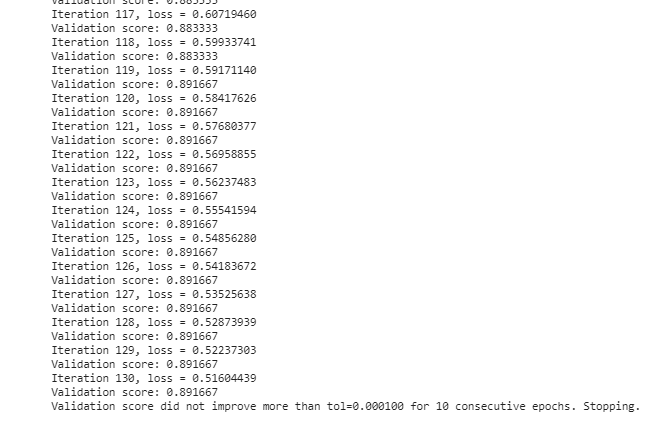
In the first experiment, the **default parameters** of the script were run **5** times and their corresponding training and testing results are compared. Here, MultiLayered Classification has been used with an activation function of **tanh**. The number of hidden neurons and principal components used are **200** for both.

Training dataset:



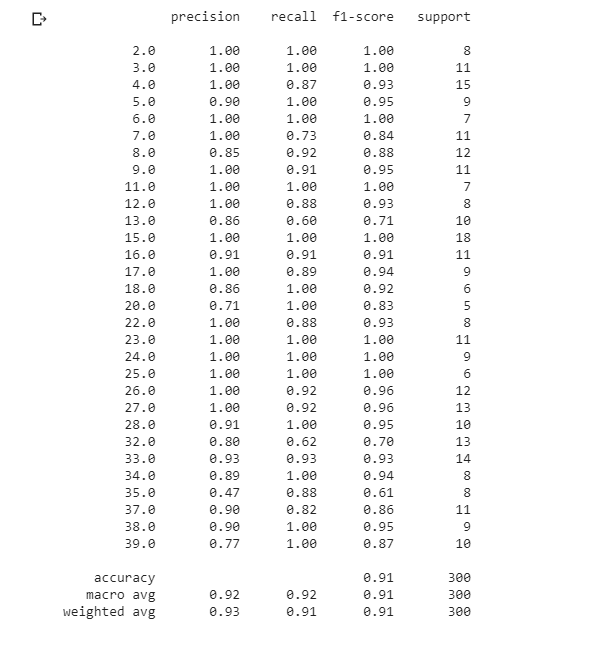
Here, an **final accuracy of 90%** was observed which was performed in 167 iterations, which is a lot. The trend remains similar in the realm of 80-92%.

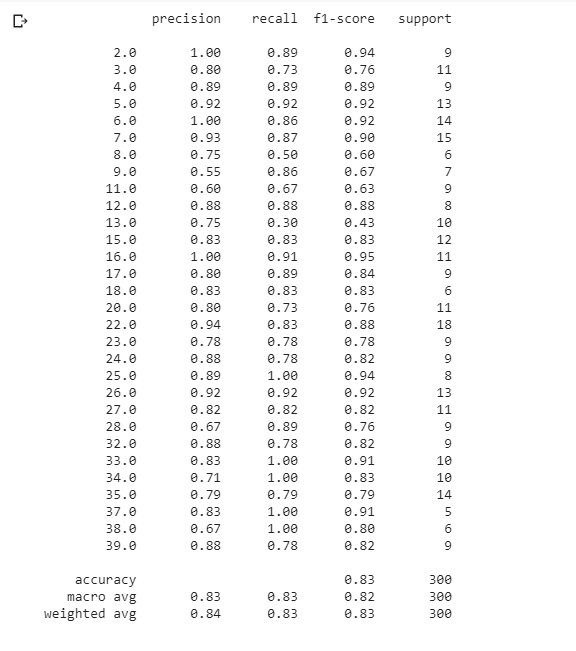
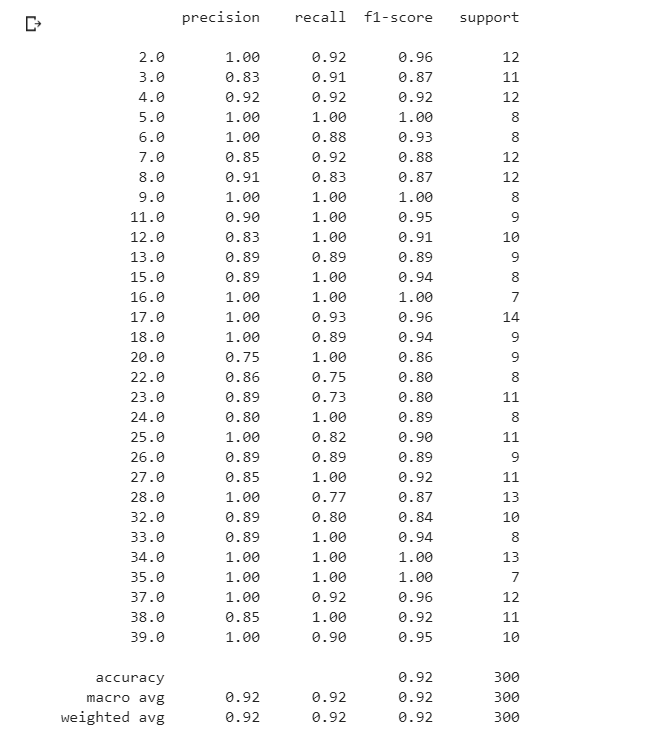


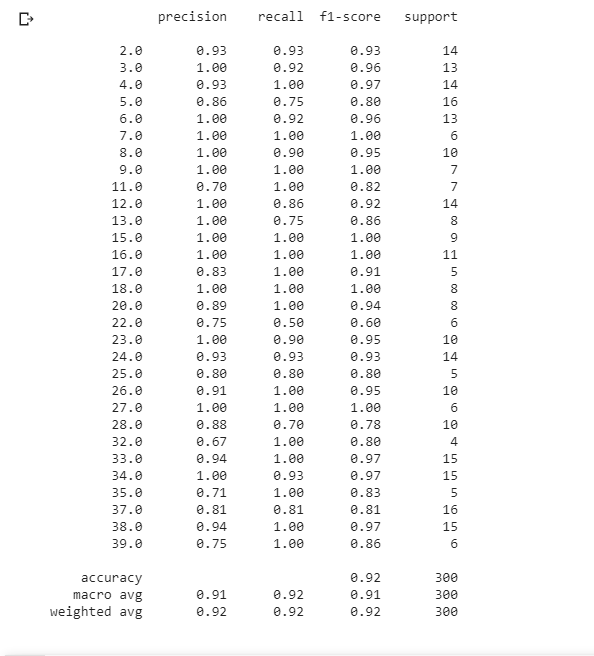
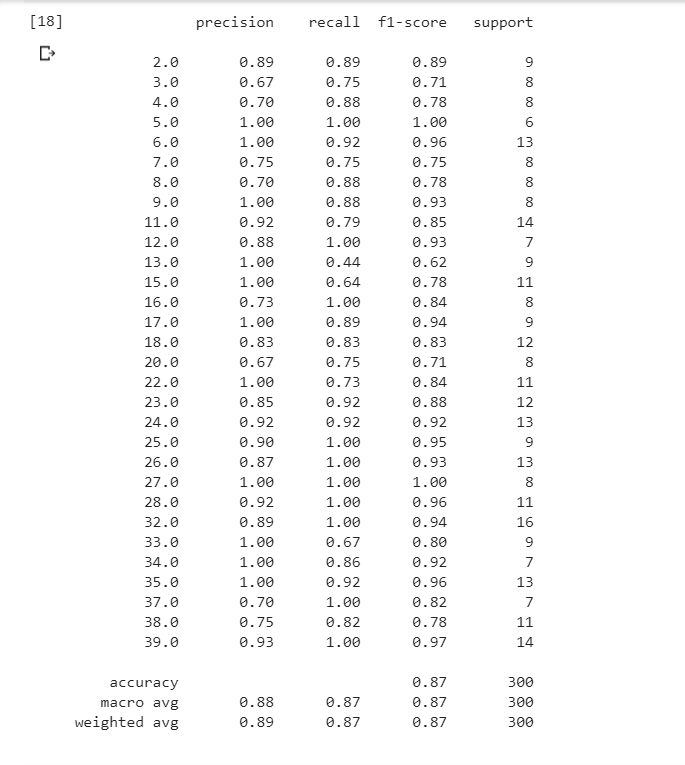


As observed in the rest of the training runs, the accuracy stays in the 80-92% however the number of iterations are comparatively less than in the first run. This is still a lot of iterations with a lot of fluctuations.

Average training accuracy: **87.36%**

Testing dataset: 





Average testing accuracy: **89%**

20% of the total dataset has been split separately beforehand for testing only. The results after testing gives off the similar predictions and accuracy. These are fairly high however this model still fails to achieve highest possible accuracy and the model is **overfitted** and isn’t able to predict if a data with new pattern is provided.

The ANN here varies with the same parameters i.e. we get varying results for the same parameters. This is because the weights are randomized as the training begins. As training continues, the weights are adjusted toward the desired values and the correct output. Another reason for varying accuracy can be because of the use of solver type as sgd (Stochastic Gradient Descent) which takes random data as batches for input rather than the whole dataset split for training.

From this, we can begin the approach of perfectly training our model with highest possible accuracy and efficiency.

**Basic experiment 1:**

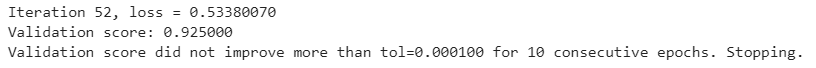
In this experiment, **only** the hidden neurons will be tweaked so that the accuracy is highest as possible with less variance.

For this **number of hidden neurons** was first increased to **1500** since more neurons equates to more logical operations.

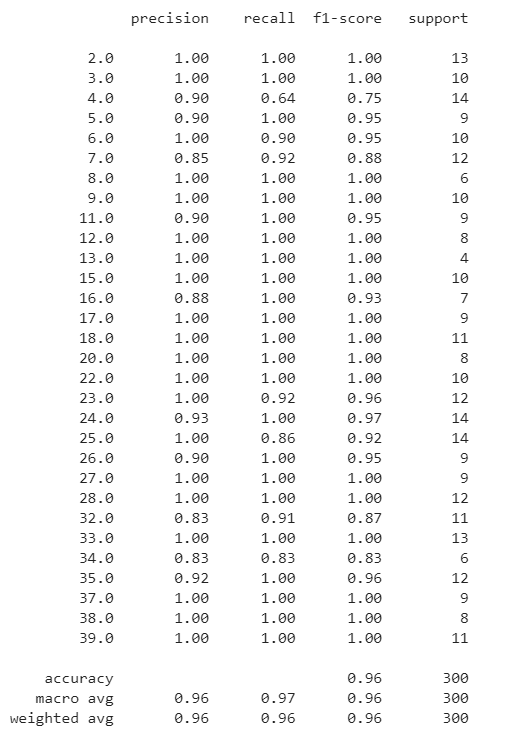
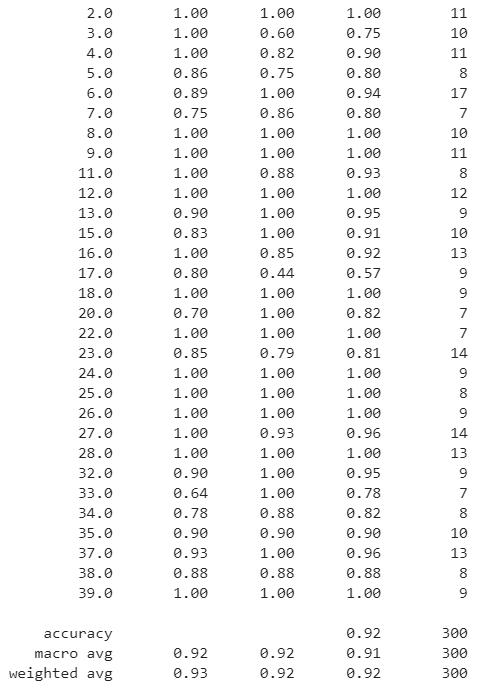
Training:

|  |  |
| --- | --- |
| **Training** | **Accuracy** |
| 1st Run | 96.7% |
| 2nd Run | 90.8% |
| 3rd Run | 89.2% |
| 4th Run | 91.7% |
| 5th Run | 92.5% |
| Average Accuracy | 92.2% |

Here, we can observe the accuracy range and average accuracy significantly improved from what it was in the initial run. The average accuracy increased from **87.36%** to **92.2%** and it showed only one case of accuracy in the range of **80-90%.**



Taking an example of the **5th run**, we can observe the **number of iterations** being significantly less than it was in the first experiment. However, it also takes a slightly **extra time** than earlier since the increasing number of hidden neurons makes more use of computational power.

Testing:

For testing, the outcomes can be interpreted the same. It still fails to recognize new patterns.

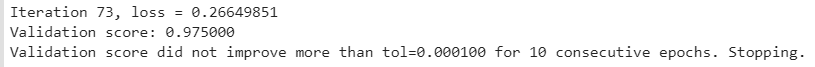
In theory, the number of hidden neurons should be at least ⅔ of the input size plus the size of the output layer. Keeping that in mind, continuing the same experiment by **increasing** the number of hidden neurons and keeping the **number of principal components constant**, in the values of 2500, 3500, 4000 and 8000.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Training** | **Accuracy** (nohn=2500) | **Accuracy** (nohn=3500) | **Accuracy** A(nohn=4000) | **Accuracy** (nohn=8000) |
| 1st Run | 96.7% | 95.8% | 92.5% | 97.5% |
| 2nd Run | 98.3% | 99.2% | 94.2% | 97.5% |
| 3rd Run | 97.5% | 95.0% | 95.0% | 100% |
| 4th Run | 98.3% | 97.5% | 96.7% | 96.7% |
| 5th Run | 91.7% | 96.7% | 95.8% | 95.8% |
| Average Accuracy | 96.5% | 96.8% | 94.9% | 97.5% |

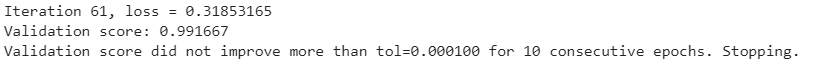
Here, the average training accuracy increased as the number of hidden neurons however, it also **decreased** in the case of **4000**. After many runs in the same parameters, the accuracy sometimes reached very high for instance, the accuracy was once noted to be 100%. However the general trend of accuracy was in the range of 94-99% for all.

Taking examples of iterations they all took:

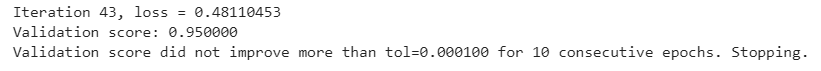
For 2500:



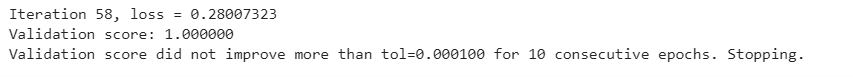
For 3500:



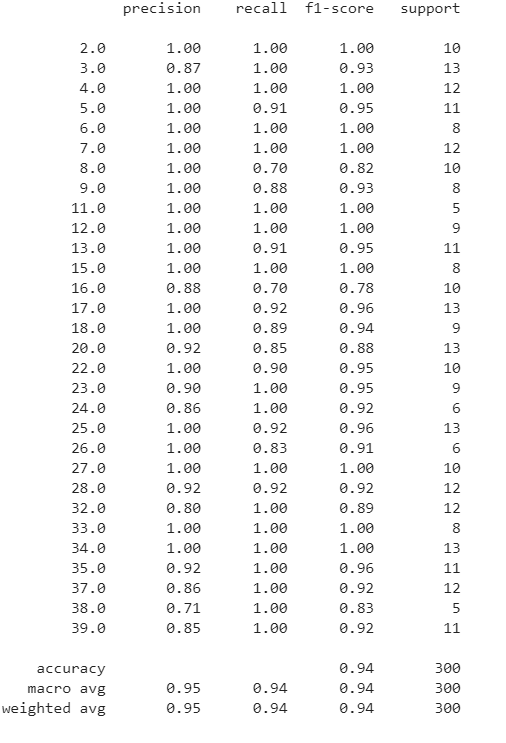
For 4000:

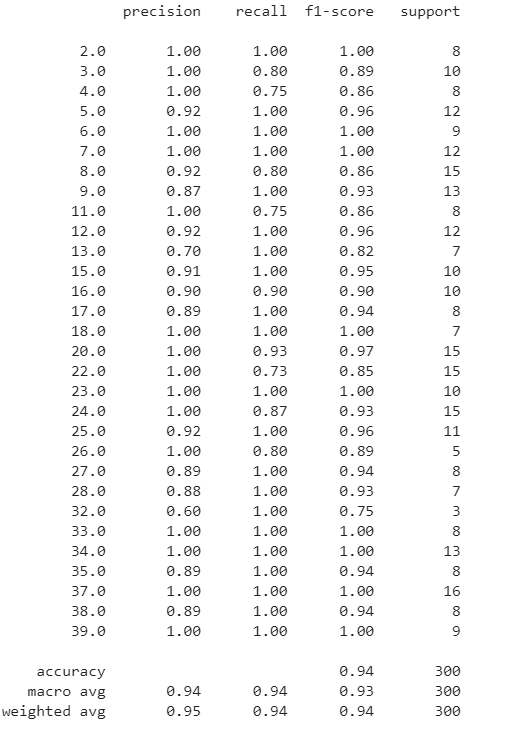


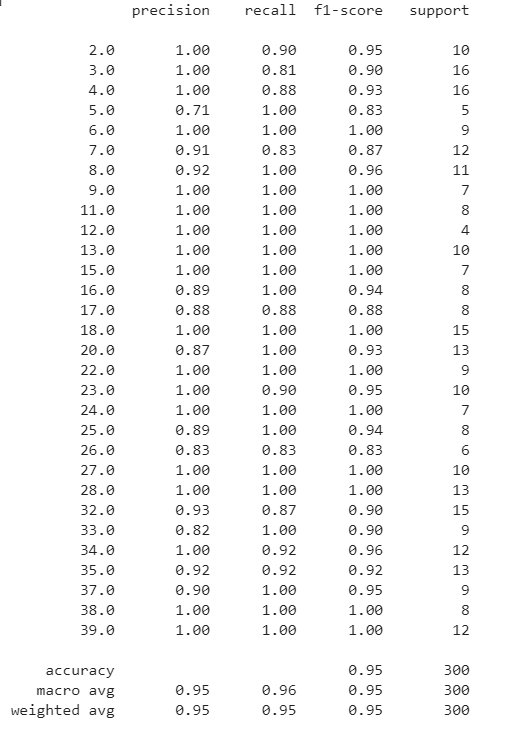
For 8000:



The trends here are similar, as the number of hidden neurons increases the computational time by a lot. It takes a lot of time to complete even one iteration for 8000 hidden neurons. The iterations kept decreasing in every increase of the hidden neuron.

Meanwhile in testing:





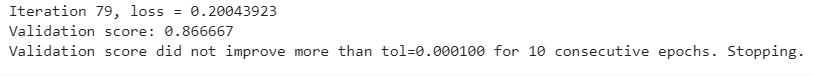
Here, there haven’t been significant changes in the average accuracy in the different changes of hidden neurons. The problem of **overfitting** hasn’t been solved either.

Looking at these, we can consider 2500 to be the optimal number of hidden neurons since it gives a high accuracy with a decent amount of iterations. It also takes less computing time to perform the training and iterations and produce results on the par of 8000 hidden neurons. Hence, **2500** is believed to perform better from all of the options chosen above.

**Basic Experiment 2:**

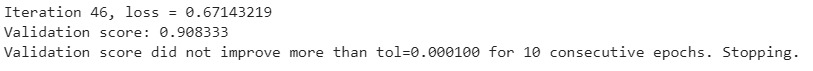
Keeping the **number of hidden neurons as 2500**, experiment 3 is based around **changing the number of principal components** value i.e. which decides the number of features it should take or discard from the image for optimization.

Increasing the number of principal components increased the number of features that had been discarded which increased the number of iterations while training the data. When it was **increased to 400**, the following could be observed.

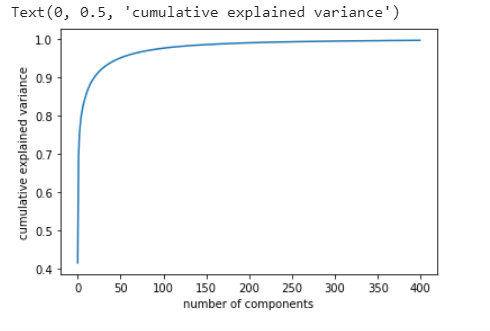


The accuracy was observed to be relatively lower with more iteration. This is because increasing the components gives more values to be required training. So, by trying to reduce the number of features it might be more possible to improve the results.

Now, keeping **number of principal components as 100,**

****

The **iteration** was found to **decrease** by a huge margin however the accuracy was still lower comparatively on many tests. This might have been because it discarded some of the important image features.

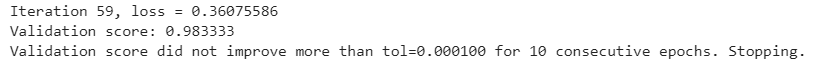


By plotting a graph of variance with the number of components, it could be observed the line began becoming straight around **175-225.** So, taking that into consideration the optimal value can be chosen.

From this, we can also conclude changing PCA but in a small amount around 200 might be helpful to improve accuracy, By keeping the values **180, 190, 195, 205** and **210** respectively the following results are observed.

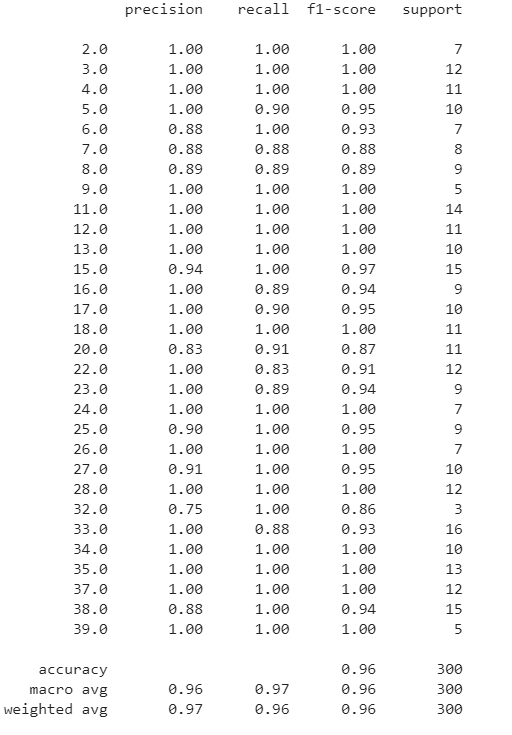
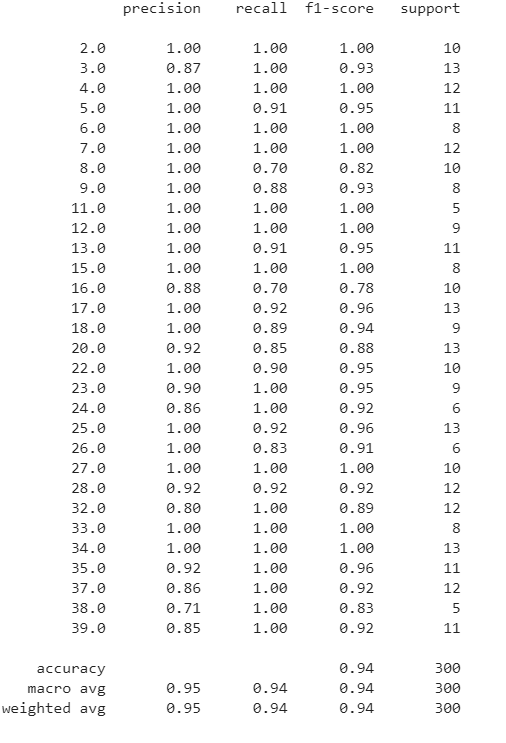
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Training** | **Accuracy** (principal components=180) | **Accuracy** (principal components=190) | **Accuracy** (principal components=195) | **Accuracy** (principal components=205) | **Accuracy** (principal components=210) |
| 1st Run | 95.0% | 96.7% | 98.3% | 94.2% | 96.7% |
| 2nd Run | 94.2% | 96.7% | 96.7% | 94.2% | 95.8% |
| 3rd Run | 95.8% | 97.5% | 98.3% | 96.7% | 95.0% |
| 4th Run | 95.0% | 97.5% | 95.8% | 95.8% | 94.2% |
| 5th Run | 95.8% | 95.0% | 95.0% | 95.0% | 93.3% |
| Average Accuracy | 96.2% | 96.7% | 96.8% | 95.2% | 95.0% |

The accuracy was somewhat stable throughout all the change however, when the number of principal components were 190 and 195 it performed better than when it was increased to 205 and 210. Choosing the better set of accuracy, I think **195** will provide the most effective and efficient results.



Theiterations also decreased on average in comparison to how it performed on default settings.

As for the testing portion, the accuracy did go up from **94% to 96%** compared to how it was with 200 PCA.



In conclusion, for the basic parameters,

**number of hidden neurons: 2500**  
**number of principal components: 195** are chosen.

**Advanced Experiments:**

So far, the parameters used in the MLP Classifier can be tweaked and changed according to the need. Here, the model has been based on a default constant **learning rate** of ‘**0.01**’. It’s solver type is **‘sgd’** which as explained before select badges of samples which are chosen by **batch size** of ‘256’ rather than the whole dataset for each iteration. SGD compares the output of the network to the expected output and calculates the error. This error is then propagated back through the network, one layer at a time, and the weights are updated according to the amount that they contributed to the error. This is called the backpropagation algorithm. The process is repeated for all of the examples in the training data. (Brownlee, 2020)

Similarly, the **tanh** activation function has been used on this classifier. Activation function determines whether the neuron should be activated or not based on the threshold value. The activation function changes the output of each neuron to a range between 1 and 0 or between -1 and 1. There are many types of activation functions, some of which that can be used in the MLP classifiers are ‘identity’, ‘logistic’, ‘tanh’, ‘relu’ . They each have their own uses however, in the provided script ‘tanh’ also known as hyperbolic tangent function have been used. It is basically a sigmoid function but better. Lastly, **early stopping** has been set to **True** which terminates the training when validation score is not improving. This is only applied when solver is of type ’sgd’ or ‘adam’. (sklearn.neural\_network.MLPClassifier — scikit-learn 0.22.2 documentation, 2020) The model has been set from the above parameters and **fit** into the PCA applied X dataset and Y labels which trains the model using MLP classifier.

**K-fold cross validation:**

The dataset above has been trained and tested by a fixed dataset, however, it has not undergone any proper validation or testing. If a new pattern of data is provided, it fails to identify the images with high accuracy. So, the data is overfitted with the training and testing dataset it receives. With K-fold cross validation, we aim to fix just that and validate. We can split the data into a number of folds denoted by K.

Firstly using **K=5** the dataset will be divided into testing and training in five folds. After running 3 tests for it, the accuracy for each tests are as follows:

****

Average Accuracy: **79.1%**

****

Average accuracy: **79.3%**

****

Average accuracy: **80.1%**

**Overall Average accuracy: 79.5%**

As we can notice there is fluctuation in accuracy in each fold. Some have very high accuracy while others have a relatively low accuracy. This is because the data is **overfitted** and fails to efficiently recognize new patterns. The k-fold cross validation trains and tests the data in different folds so that it can familiarize itself with different patterns. Here, we achieve an overall accuracy of 79.5% which is decent for our model.

Now, keeping the value **k=10,**



Average accuracy: **92.6%**

****

Average accuracy: **90.9%**

****

Average accuracy: **90.3%**

**Overall Accuracy: 91.3%**

Having K=10, gives a **more stable** and **very high accuracy** results as compared to k=5. This is because the model is trained and tested in 10 different folds which reduces the **overfitting.**

**Final Experiment**:

The final experiment will be performed using Adaboosting which combines multiple weak classifiers to make a high accuracy strong classifier. Adaboost is an ensemble model which works with multiple models to make predictions. Multiple individual models can be combined sequentially to make the ensemble model more flexible(less bias) and less data-sensitive (less variance). Each individual model also learns from its mistakes from the previous model. (Chen, 2020) Boosting algorithms in general are less affected by overfitting and adaboosting reduces overfitting by a huge margin though it is not prone to it. (Navlani, 2020)

In the adaboost model created, the parameters that are set and altered are the **base\_estimator**, **n\_estimator, max\_depth** and the **learning rate.**

Talking about the **learning rate**, the lower the learning rate, the steps taken for gradient descent are low so it is able to successfully reach the minimum point though it might take more time. However, if the learning rate is high, it will quickly descend but with a risk of skipping the minimum point completely. In this case, the learning rate is the weight of weak learners. So, low learning rates which do not take up a lot of time must be considered.

As for the **base\_estimator**, it is the classifier used to model the weak learner and train the overall model. Here, I will be altering between a **Decision Tree** and a **Support Vector Classifier** to choose the better model. **max\_depth** for the decision tree is basically the depth of the tree i.e n\_estimators on one level of the decision stump. High max depth can cause overfitting while low max depth causes underfitting. This is due to the nodes required for the decision trees increase and might overtrain.

Finally, **n\_estimator** are the number of weak learners that are used to train iteratively.

Using a **learning rate** of **0.1**, **base estimator** as **Decision Tree**, **n\_estimator** as **500** and the **max\_depth** as **1,** the following accuracy can be achieved on the predictions and testing.

**Changing the Learning rate:**

As mentioned above, learning rate should be kept low but not too low according to the nature of the model and the dataset. Firstly, **0.1** is chosen as the initial learning rate as the standard is usually 0.1 or 0.01. The testing accuracy is then measured. Then, the value of learning rate is **increased and decreased** to check the improvement in accuracy or a drop for the optimal learning rate.

All the experiments are conducted on a dataset which are split and PCA applied to it using the final parameters from the experiments above.

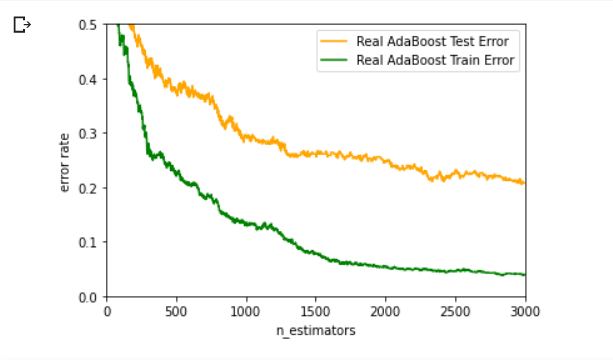
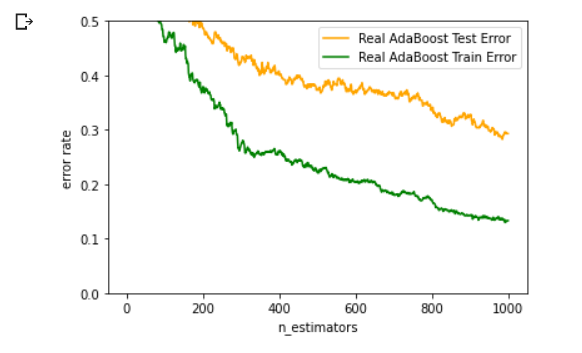
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Learning rate 0.1** | **Learning rate 0.01** | **Learning rate 0.5** |
| **1st Run Accuracy** | 57.9% | 46.9% | 26.9% |
| **2nd Run Accuracy** | 56.3% | 45.9% | 18.9% |
| **3rd Run Accuracy** | 65.3% | 47.7% | 20.5% |
| **Average Accuracy** | 59.8% | 46.8% | 22.1% |

As it can be observed, the accuracy is **significantly lower** than the one observed from K-fold; however, it can be improved by tweaking the parameters.

In this case, the learning rate of **0.1** can be seen performing way better than the rest.

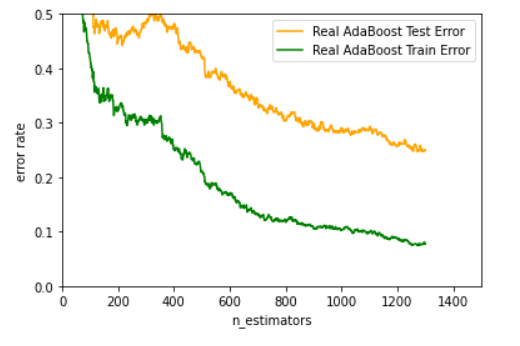
**N\_estimation:**

By keeping **n\_estimation as 1000** and **3000 respectively** and plotting a graph against the errors, the following graph can be achieved.



As we can observe from the graph of error rate and n\_estimator. The error rate gradually decreases as the number of estimators increase for both training and testing data. We can say **overfitting** is happening as the difference of accuracy between training and testing data is huge especially when the number of estimators is high. The training accuracy does improve however at the cost of being overtrained and unable to predict future patterns well.   
  
The accuracy is also observed to be **69.4%** for n\_estimation as **1000** and **79.2%** for n\_estimation of **3000.**

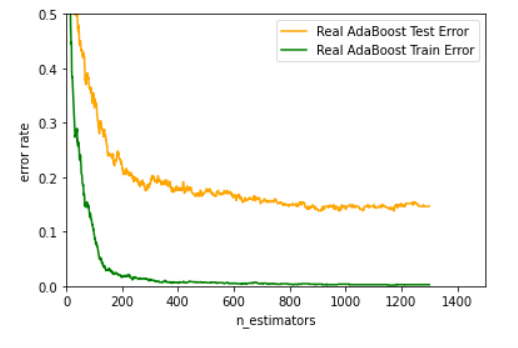
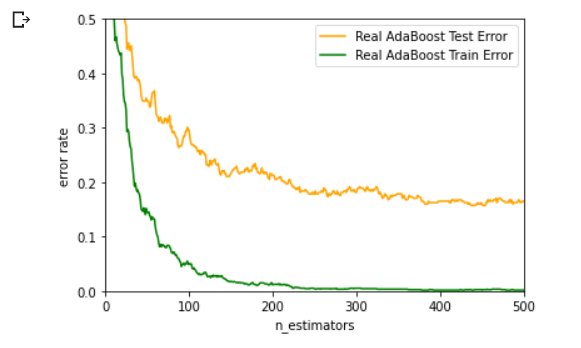
Looking at the graph the optimal number of n\_estimators can be seen around 1000-1500 as the model isn’t overfitting and giving decent results.



Finally, choosing an **n\_estimator of 1300** gives a **decent accuracy** and **low overfitting**. This also gave an accuracy of **74.9%**

**Max depth and Base estimator:**

Increasing the max depth value by a lot can cause high overfitting, so increasing max depth by **2** and **3** the following accuracy and graph can be observed.

**max\_depth = 2 max\_depth = 3**

By increasing the depth it can be seen the number of estimators can be dropped to 250 and still give similar results.

Compared to max\_depth =2, max\_depth = 3 gives a far better and smoother graph. So, from this we can decrease the **number of estimates** as **250** and **max depth** as **3.**

Noting the testing accuracy when **number of estimates = 250 and max\_depth = 3**

|  |  |
| --- | --- |
| **Test** | **Accuracy** |
| **1st Run** | 88.5% |
| **2nd Run** | 85.7% |
| **3rd Run** | 87.7% |
| **Average Accuracy** | 87.3% |

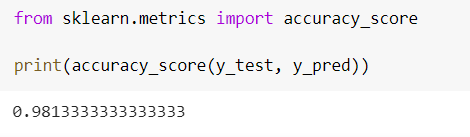
The average accuracy obtained from the current parameters is 87.3%. It does give a **fairly high accuracy** on testing which shows the model hasn't been overfitted by a lot. Now in order to obtain the highest possible accuracy.

Changing the **base\_estimator** classifier from Decision Tree to Support Vector Classifier, the weak model will now not work on the basis of trees but rather hyperplanes i.e. multidimensional planes to classify the images. The following accuracy can be obtained by implementing SVC:

|  |  |
| --- | --- |
| **Test** | **Accuracy** |
| **1st Run** | 97.6% |
| **2nd Run** | 97.3% |
| **3rd Run** | 97.1% |
| **4th Run** | 98.1% |
| **5th Run** | 97.9% |
| **Average Accuracy** | 97.6% |

After applying the SVC model the accuracy can be seen significantly high and consistent throughout the 5 runs which shows the great potential of hyperplanes used by SVC for classifying images for the weak learners.

The final average accuracy using Adaboost is noted to be **97.6%** which is very high and the highest noted accuracy for the entire model is found to be **98.1%.**

****

**Conclusion:**

In conclusion, in order to classify and recognize different faces, various classifiers were used. Firstly, MLP Classifier was used which had a very high average accuracy of **96.8%** but overfitted and failed to recognize new patterns. Hence, K-fold was applied to the classifier which helped to minimize the overfitting and better at recognizing new patterns. An average overall accuracy of **91.3%** was obtained in the end after many sets of experiments. Then, in order to solve problem of high variance and bias and to reduce overfitting even further, adaboosting was used along with Support Vector Classifier as the weak learner classifier which helped to elevate the average final accuracy to **97.6%** and the highest final accuracy to a whole **98.1%**.

As for my group, each member of the team worked on their own solutions and interacted throughout on what model they were gonna apply. The group activities were coordinated well and our tutor conveyed and helped us with most of our doubts regarding the assignment.

Putting together all the results concluded by the team:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Student ID** | **Name** | **Final Model Used** | **Final Average Accuracy** | **Final Highest Accuracy** |
| 1815674 | Samrat | Adaboost | **97.6%** | **98.1%** |
| 1811043 | Praveen | MLP | 94.0% | 97.0% |
| **1816249** | Prazit | MLP and K-Fold | 88.8% | 94.3% |
| 1811037 | Amrit | MLP and Stratified K-Fold | 92.3% | 97.3% |

The **highest possible results** obtained from the whole experiment is the Adaboost model with an final average of **97.6%** and the highest accuracy of **98.1%**. So from our group, we will be choosing my solution for our final conclusive accuracy.

**References**

FaceFirst Face Recognition Software. (2020). *21 Amazing Uses for Face Recognition – Facial Recognition Use Cases*. [online] Available at: https://www.facefirst.com/blog/amazing-uses-for-face-recognition-facial-recognition-use-cases/ [Accessed 6 Mar. 2020].

Brownlee, J., 2020. *Crash Course On Multi-Layer Perceptron Neural Networks*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/neural-networks-crash-course/> [Accessed 12 March 2020].

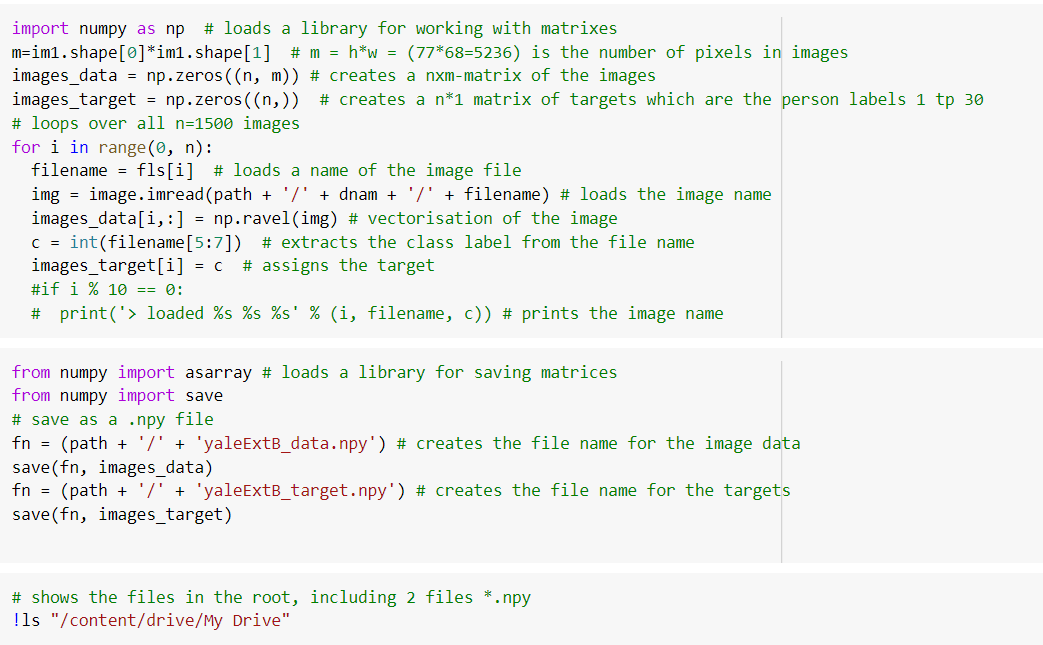
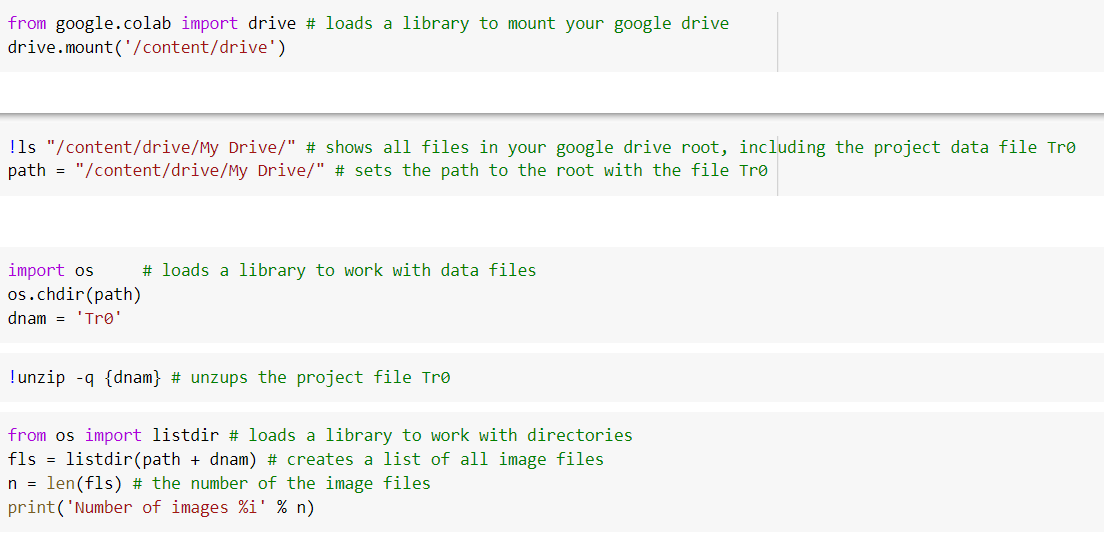
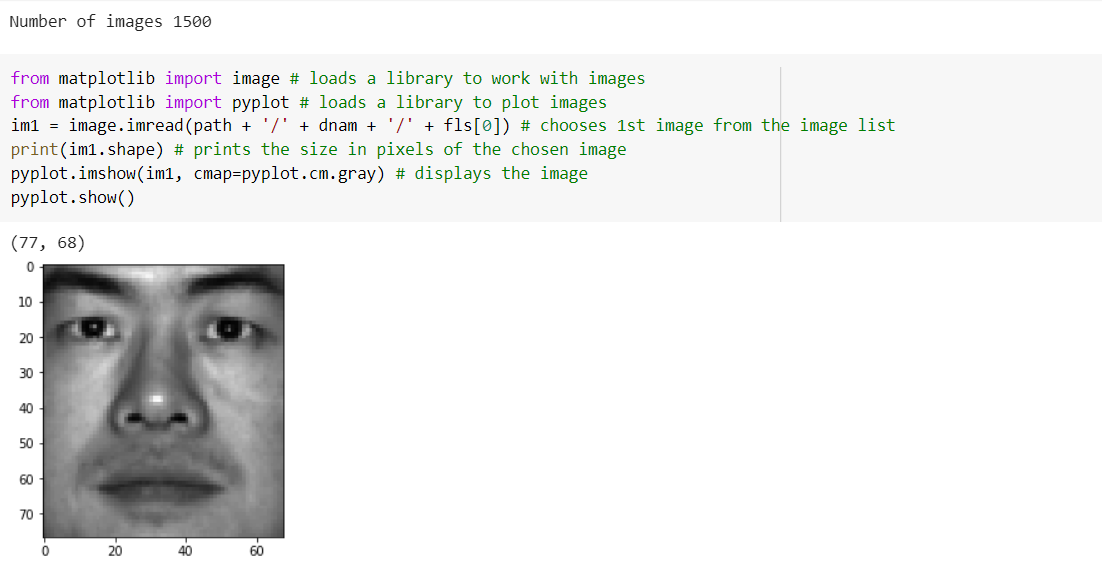
Scikit-learn.org. 2020. *Sklearn.Neural\_Network.Mlpclassifier — Scikit-Learn 0.22.2 Documentation*. [online] Available at: <https://scikit-learn.org/stable/modules/generated/sklearn.neural\_network.MLPClassifier.html> [Accessed 12 March 2020].

Navlani, A., 2020. *Adaboost Classifier In Python*. [online] DataCamp Community. Available at: <https://www.datacamp.com/community/tutorials/adaboost-classifier-python> [Accessed 18 March 2020].

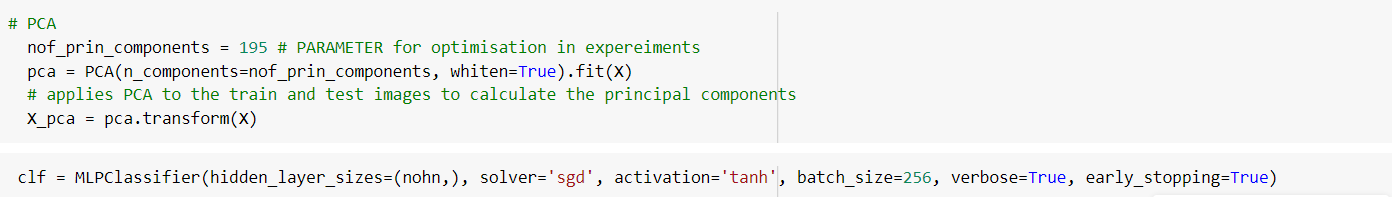
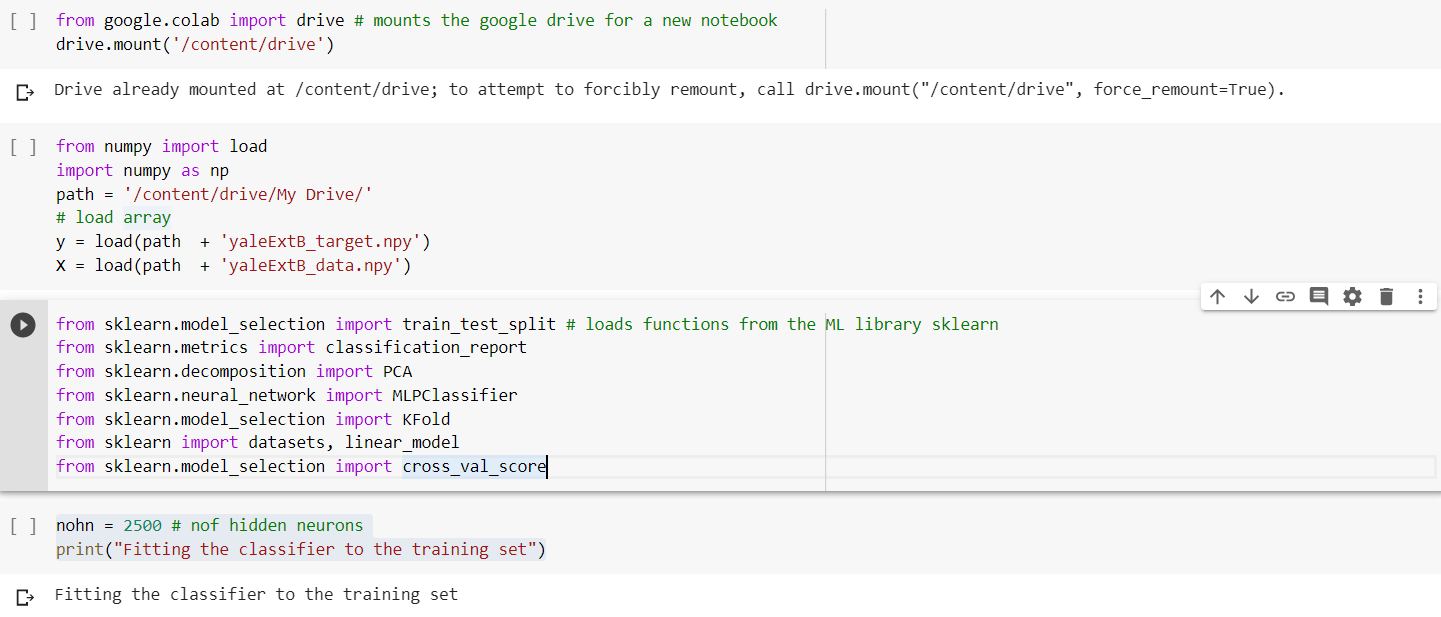
Chen, L., 2020. *Basic Ensemble Learning (Random Forest, Adaboost, Gradient Boosting)- Step By Step Explained*. [online] Medium. Available at: <https://towardsdatascience.com/basic-ensemble-learning-random-forest-adaboost-gradient-boosting-step-by-step-explained-95d49d1e2725> [Accessed 18 March 2020].

**Appendix:**

Code to process the images (default script)



Final K fold integrated MLP classifier code:



Final code: (Adaboost with SVC)

