CSC 412 Project Report
Facial Expression Prediction

Dhyey Sejpal
1000162986
dhyey.sejpal@mail.utoronto.ca

Shahin Yousefi
999811672
shahin.yousefi@mail.utoronto.ca

Abstract

This project undertakes the task of Facial Expression Prediction in a given image using various machine learning techniques. The idea is to create a web interface where anyone can visit and upload an image (not necessarily cropped to contain only a face). From that image, the face is extracted, preprocessed and classified using various techniques. The final result is provided as a majority vote of the different methods.

Introduction

Facial expression prediction is an important problem to solve, and it has its applications outside the technology industry as well. One of the most important application is the analysis of the facial images from a police interrogation of a criminal. There is a system called Facial Action Coding System, that is currently used by the police in assessing how a subject is reacting to certain questions. It includes some features like eye shape, eyebrow position, etc. to classify various micro expressions. This system, however, is too complex to implement here, in the given time and number of people working on it.

Hence, we try a very simple approach here. We take in the raw pixels of the image and try to learn their values to classify the various images into different classes, or expressions. We will be using the training data from Toronto Faces DataSet. This set has 2925 labeled, 32x32, grayscale images. We will be using the 2325 images as the training set, and 290 (10% of the set) images as validation and test sets.

The training images will be flattened to a 1x1024 dimensional vector, stacked into a matrix and the matrix will be scaled to have zero mean and unit variance for all its columns. This matrix will be used as the input to various classification models. We will be examining 5 models here, Logistic Regression, Linear SVM, k-Nearest Neighbors, Neural Networks, and Gaussian Mixture Models. The same preprocessing will be done on the validation and test sets before classifying them. We will conduct a number of experiments for each model, comparing how changing some parameters changes the result, and in the end, we will compare the models with each other.

As mentioned before, such type of sophisticated system is available to the police and maybe large companies, but the interface we build will allow anyone to classify any image. This is a very cool thing to do, and can amaze someone who is a non-programmer about how a computer can predict expressions from an image using machine learning.

Comparison to previous work

In [Liu et al, 2014], the authors created a Boosted Deep Belief Network framework that uses a three-stage training process: feature learning, feature selection, and classifier construction. The authors divided each training image into overlapping patches, and then used a DBN, an unsupervised learning method, for the patches at each location. They then used a boosted, supervised learning method on a subset of those patches that have high discriminative power,

and back-propagated to tune those selected features. The back-propagation minimized the classification error rate of all features, both the strong and weak classifiers.

In [Zhao et al, 2015], the authors combined a Deep Belief Network with a Multi-Layer Perceptron. A DBN was used to perform unsupervised learning on the raw pixels of a flattened image. Then a MLP was initialized with the same parameters as the trained DBN, and used as a classifier for the images.

In our approach, we used supervised or semi-supervised learning methods on the raw pixels of a flattened image. Since we were building a user interface that should respond quickly, we needed to choose high-performance methods that would inevitably be less accurate than a Deep Belief Network. There has been a method proposed in [Sabzevari et al, 2010] which trains a DBN quickly by utilizing face graphs extracted using a Constrained Local Model. However, we felt this method was too complicated to implement and beyond the scope of this course. We chose each model here for a specific reason. Our baseline method was the provided k-Nearest Neighbors classifier. Logistic regression is known to be a quick and simple classifier and Support Vector Machines work well with high-dimensional data. Gaussian Mixture Models seemed like a suitable Probabilistic Graphical Model for our task, especially in a semi-supervised manner. Finally, Multi-Layer Perceptrons were an obvious choice for a Neural Network classifier.

Experiments and Results

As mentioned before, we had 2925 labeled images, and we will be performing supervised and unsupervised learning on them.

The data had 7 possible classes or expressions, and the images were labeled from 1 to 7 in the order depicted in the examples here:

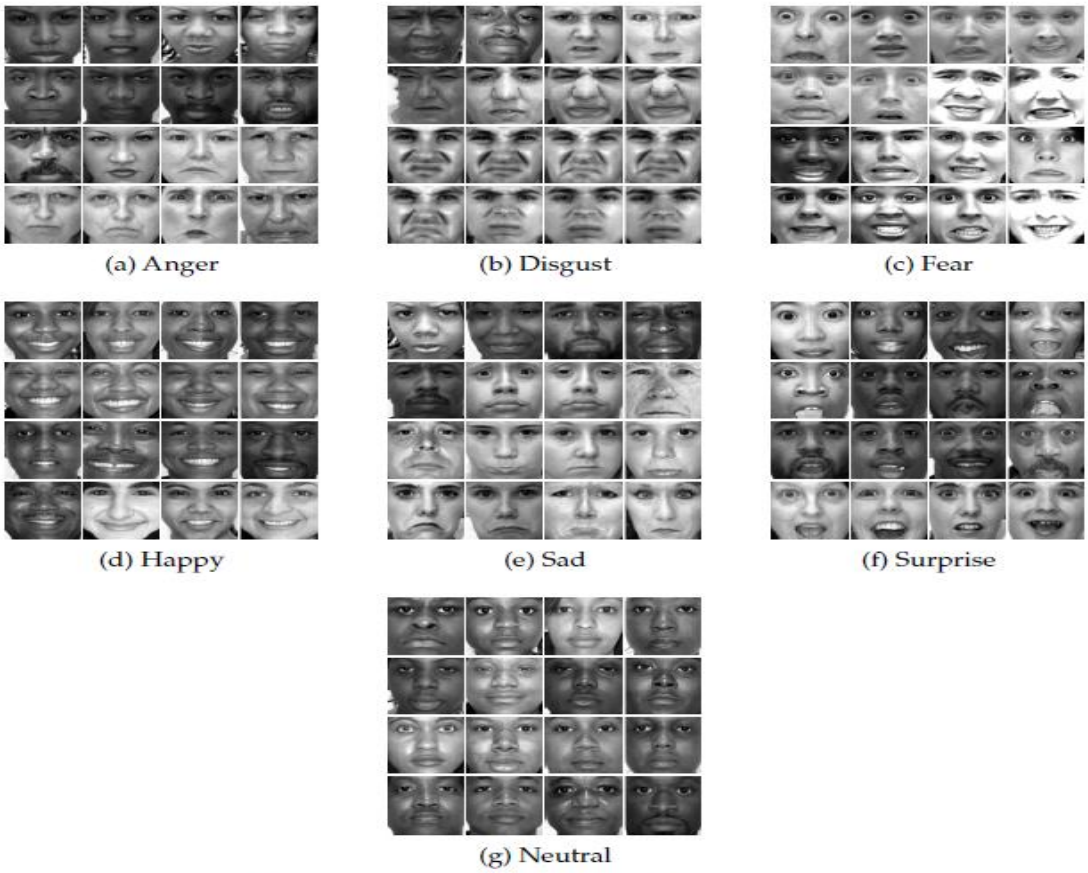


Figure 1: Training examples from the Toronto Faces Dataset.

We used 2345 images as training set, and the rest were equally divided into validation and test sets. We tried two ways to split the data:

1. The first thing we tried is to just split the data based on the index. We take the first 2345 images as the training set, the next 290 images as the validation set, and the last 290 images as the test set.

2. The next thing we try is divide the images according to their labels into different classes. Then, we take the first 80% data from each class into the training set, the next 10% from each class goes to validation set, and the last 10% goes to the test set.

After running a few early experiments, we found that the second way of splitting gives better accuracy. Naturally, the first way is close to random splitting, as we do not know how many samples from each class make it to the training set. However, taking data from the divided set would preserve the proportion of samples from each class in the original training set, and that way we could make sure that we were not leaving anything out, or a particular class did not have very few training examples. We could also be sure that the validation and test sets were not skewed.

After dividing the data, we trained several models on it, experimenting with hyperparameters and finally getting the accuracy on the test set.

Model 1: K-Nearest Neighbors

This is the standard k-Nearest Neighbors model, where the training data is projected into a d-dimensional space (d is the number of features in the data, 1024 here), and then each test example is projected to the same space. Then, we use a distance metric to get the training examples that are closest to the given test example. We then look at the class of these closest training examples, and take the majority vote to predict the class of the test example. The parameter that we experiment with is 'k', the number of closest training examples we look at to classify the test example. So if k = 1, then it just becomes the nearest neighbor classifier, and if k = 5, we look at the 5 closest examples and their classes to predict this example's label.

The code used was provided as the baseline in the project handout we linked in the proposal. It follows the procedure described above. It is used as a lower limit to compare the performance of other models. The results of our experiment are shown below:

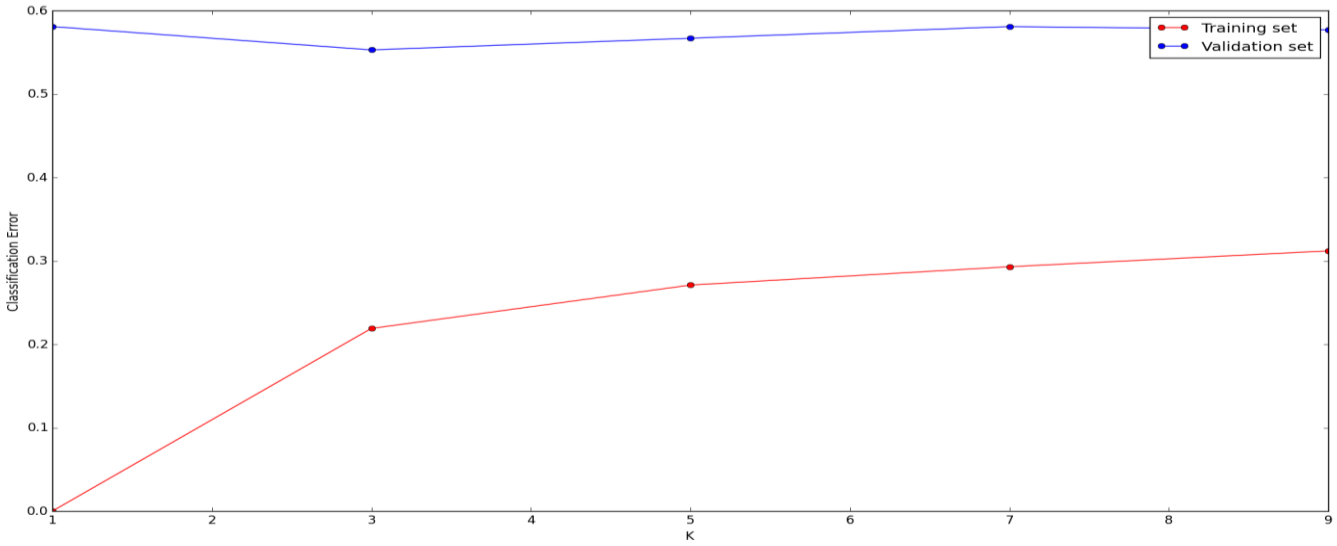


Figure 2: K vs Classification Error rates

As we see, k = 1 leads to a model that is severely overfitting. The training error is 0, but the validation error is very high. High values of k are also not good since there is a higher noise if the test example falls near a bunch of

examples from a different class. Values in the middle, like 3 and 5 usually perform well, which is also true here. The best value of k here was 3, with the validation and test error rates both being around 55%.

The model is not very powerful, which is shown by the high training error, and so it is not overfitting. But, this model was expected to do bad, as mentioned in the handout, and thus we try a few more models.

Model 2: Logistic Regression

The next model we try is another simple but common model. Logistic regression. It is often used for binary classification tasks. The formula used is $y(z) = \frac{1}{1+ e^{-z}}$, where z = $\sum w_i x_i$. Here, w_i represent the weights and x_i are the features in the input sample, which are the pixels in the image. We are using the Logistic Regression model form sklearn library in Python available at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html. We are using 5-fold cross validation.

We first experimented with how the multi class classification is handled. Minimizing the multinomial loss over the entire model, or dealing with label as binary one-vs-rest classification. The multinomial method is computationally less expensive as we only need to have a few boundaries, as compared to a binary problem for each label. The error rate on the validation set was 35% for the multinomial version, and 37% for the one-vs-rest version. Hence we will move forward with the multinomial case.

Then, we experimented with the maximum number of iterations of the optimization algorithm. Too many iterations would cause overfitting, and too few iterations would not allow the model to train properly. Sometimes increasing the limit would not help since the convergence criteria is met before that limit. The results we obtained are shown in the following graph:

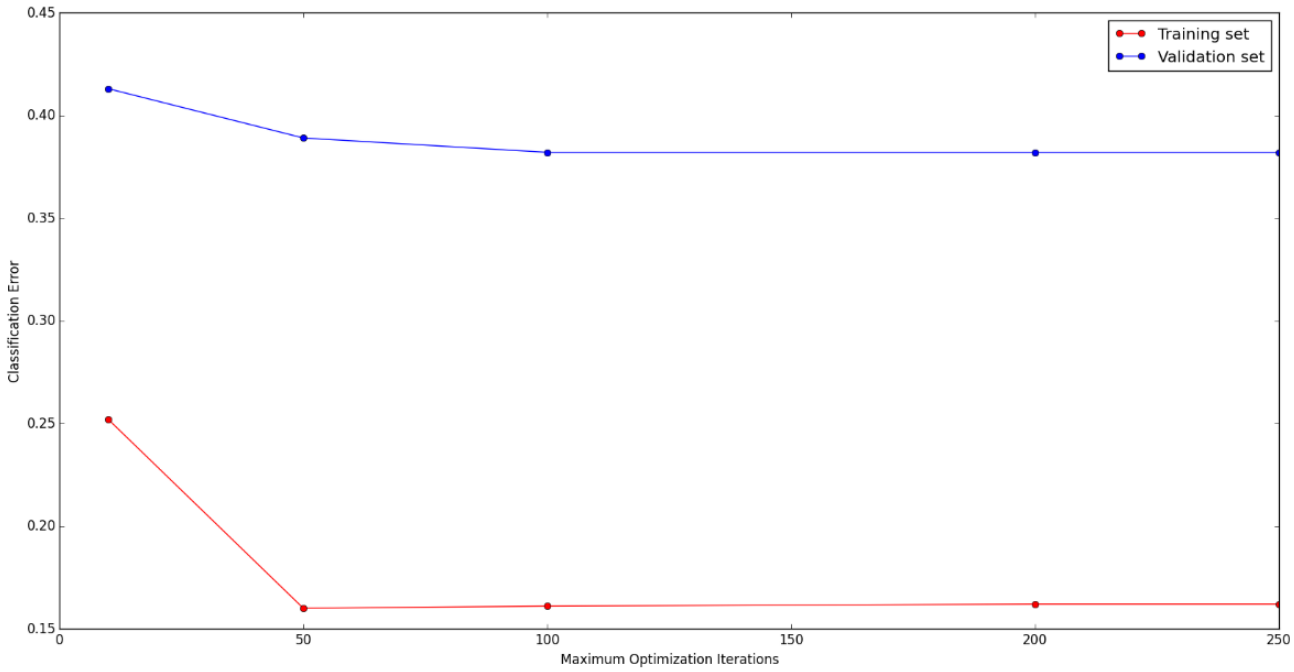


Figure 3: Maximum optimization iterations vs Classification Error Rates

We also tried experimenting with the tolerance level for stopping criteria, but it did not result in any significant change in accuracy even when the tolerance was changed by a scale of 100 (Each value was a 100 times more than

the next value). Hence we have excluded that experiment from this report to keep it concise.

We found that 100 iterations was a good limit, and with all other parameters as default, the training error was around 16%, validation error was around 38% and the test error rate was close to 39%.

This model is a significant improvement over the previous model, k-NN.

Model 3: Support Vector Machines

The next model we try is Support Vector Machines. This model tries to separate out examples form difference class by having a hyperplane between different classes. It tries to maximize the distance between the closest examples from each class, called margin. These closest examples are called support vectors since they support the choice of this hyperplane that maximizes the margin. SVMs are very effective in high dimensional data and hence we decided to use them here.

The model being used here is LinearSVC from sklearn library, available at http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html. It does classification using one-vs-rest technique.

We are using linear SVMs, as they performed better than any other kernels (e.g. Polynomial, rbf and sigmoid).

Here, we tried experimenting with the limit on maximum iterations of the optimization algorithm, and the tolerance level, but none of them resulted in significant changes. Hence we experimented with the regularization coefficient, since without it, the model was severely overfitting. The results are shown in the graph below:

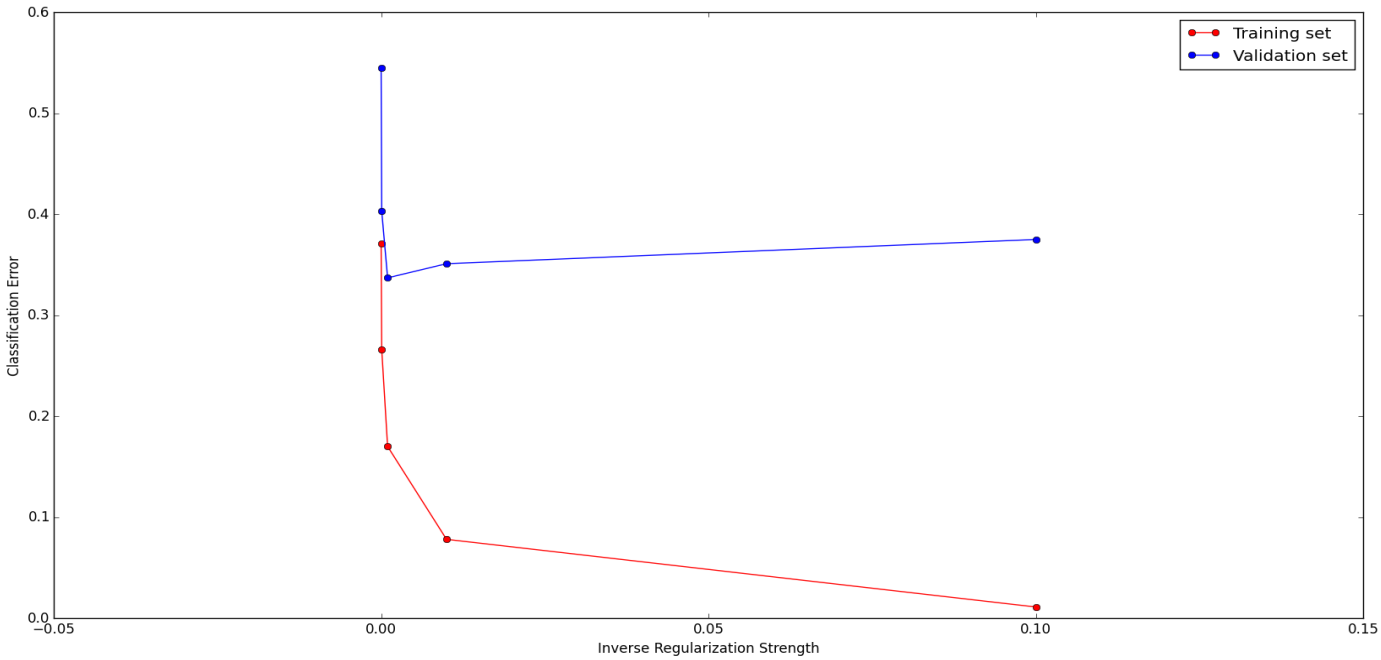


Figure 4: Inverse regularization constant vs Classification Error rates

The x-axis shows the inverse of the regularization strength, C. So smaller the value of C, the stronger is the regularization. A very big C might lead to less regularization and more overfitting, as shown in the last data point which has a very low training error but a very big validation error, while a very small C would cause very strong regularization, which makes the entire model bad, as shown by the first data point which has very high training and validation errors.

We found that the best value of C was 0.001, which gave us a training error of about 17%, validation error of about 33% and test error of about 35%.

We see that this model is a slight improvement from the previous model, logistic regression.

Model 4: Multi-Layer Perceptron (Artificial Neural Network)

Neural networks are one of the most commonly used models for image classification. It would have been an injustice not to include them here. The model is available from scikit-neuralnetwork as Multi-Layer Perceptron at https://scikit-neuralnetwork.readthedocs.org/en/latest/module_mlp.html#classifier. The same preprocessed data has been used as input to the neural network. The input layer takes in a 1024 dimensional vector. The output is a softmax over all the classes.

The same pre-processed data has been used here. Unlike other models, neural networks are very complex. We have a lot more parameters that we can tune to try to improve the accuracy of the result. We will conduct the experiments sequentially, by keeping the best parameters form the previous experiment in the successive experiments. The first thing we can change is the activation function used in the hidden layers. That is, the non-linearity in the hidden layers. We try a few different things, and get the following results:

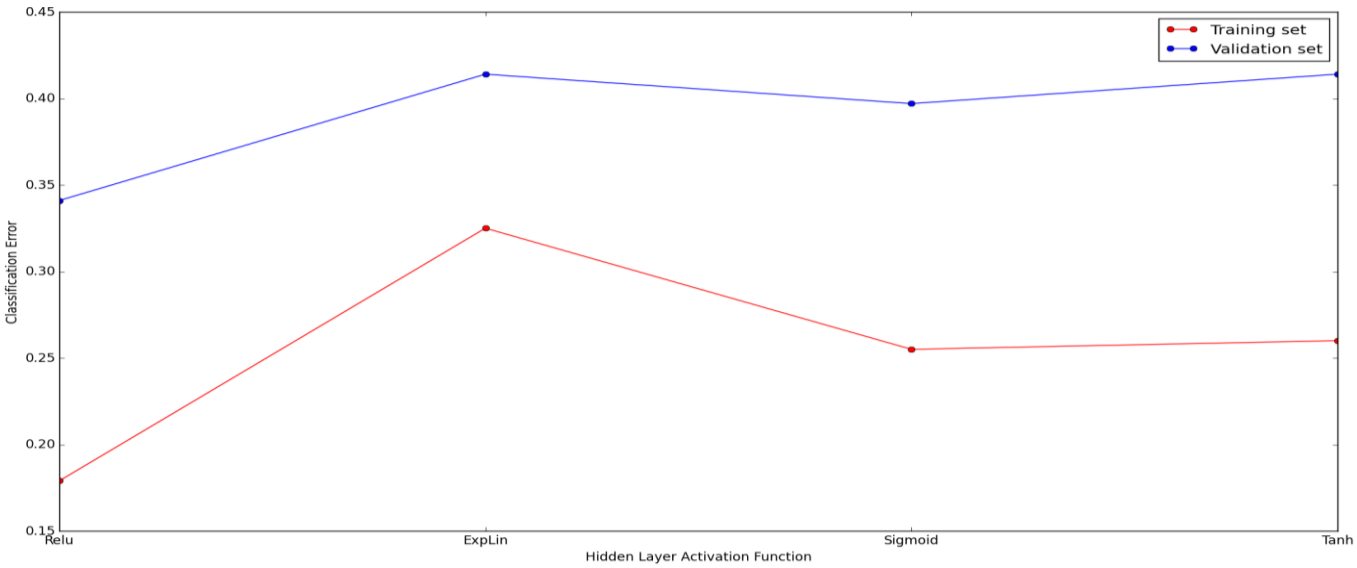


Figure 5: Hidden layer activation function vs Classification Error Rates

The x-axis shows the different functions used as hidden layer activation functions, Relu (f(x) = max(0, x)), ExpLin(f(x) = (x>=0)*x + (x<0) * alpha * (exp(x)-1)), Sigmoid (f(x) = 1 / (1 + exp(-x))) and Tanh(f(x) = tanh(x)). The best results were obtained with Relu, and we will be using that in our future experiments. The validation error was around 34%.

Next, we experiment with the learning rule. We try all the provided options in the API to select the best rule to use while updating the weights. After comparing all of them, we found the following results:

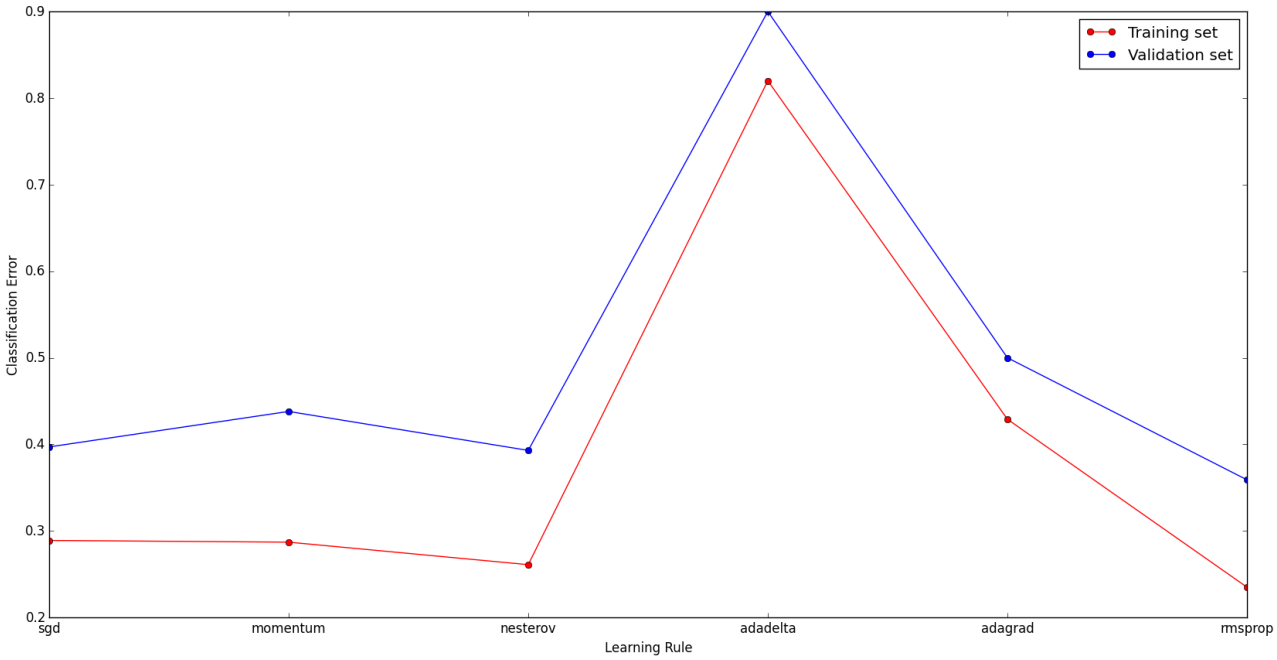


Figure 6: Learning Rule vs Classification Error Rates

We see that although the default rule, sgd does good, rmsprop brings down the error rates. The validation error rate was around 36%, not as good as previous experiment, but we will stick to rmsprop for now.

Next, we try to change the learning rate. If it is too big, then the model oscillates around the optimum solution, but if it is too small, learning is very slow. Hence we tried a few values to get the best learning rate.

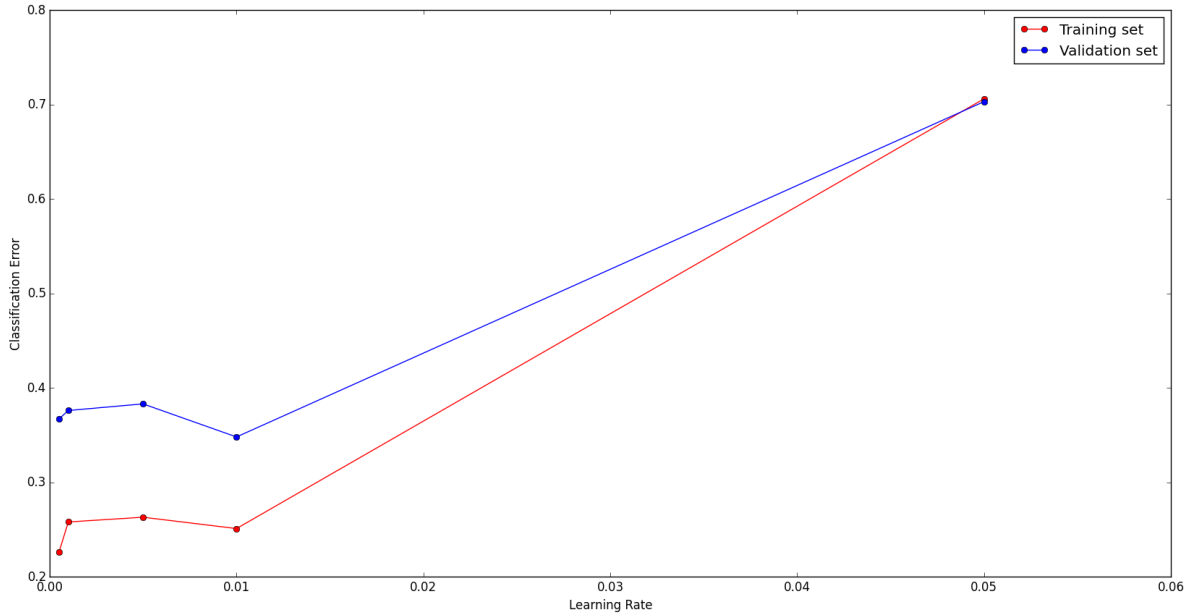


Figure 7: Learning Rate vs Classification Error Rates

As we discussed, a very high learning rate causes the model to be bad, considering the number of optimization iterations are same. Typically a good value is around 0.01, which we also see here, and it brings down the validation error to 35%.

Next, we try some learning momentum. This sometimes speeds up the learning, but sometimes causes the model to be too powerful and overfit the data. We tried a few values of momentum and obtained the following results:

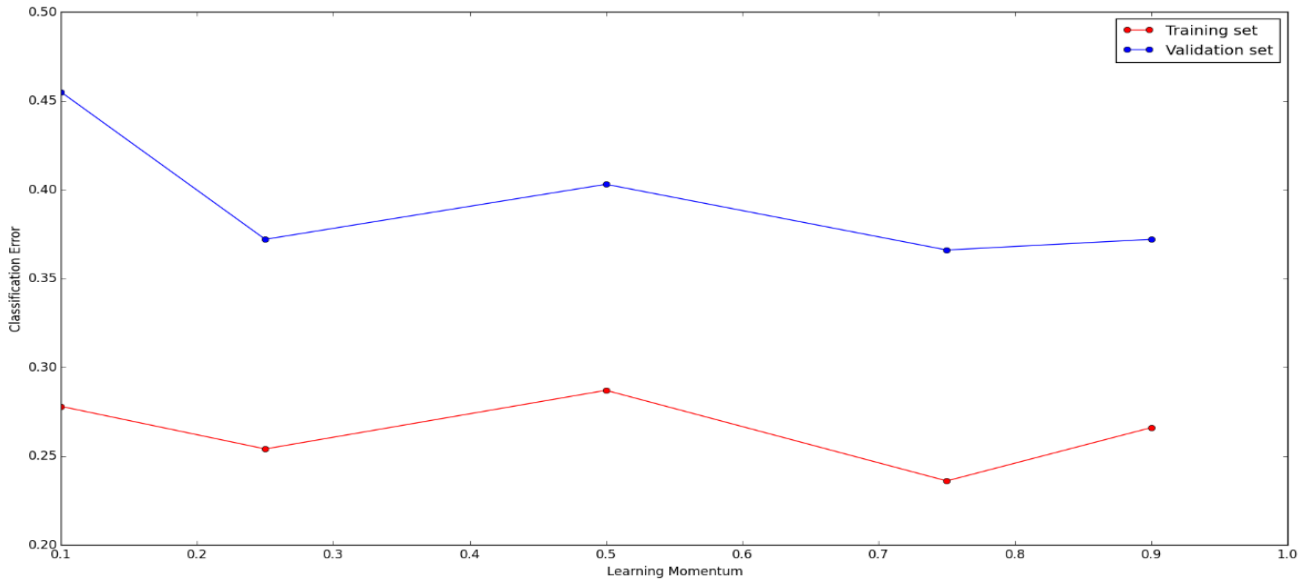


Figure 8: Learning Momentum vs Classification Error Rate

We see that the error rates are the lowest around 0.75. Typically momentum values are around 0.8-0.9, so this value is a feasible value. The training error went down by 2%, but the validation error increased to 36.6%. However, this is not a very big change to not include momentum in training.

Neural Networks are very powerful, and hence there are a lot of parameters that we can fine tune. One would be the batch size used for learning. If it is equal to 1, it becomes online learning. We tried a few values here, and got the following results:

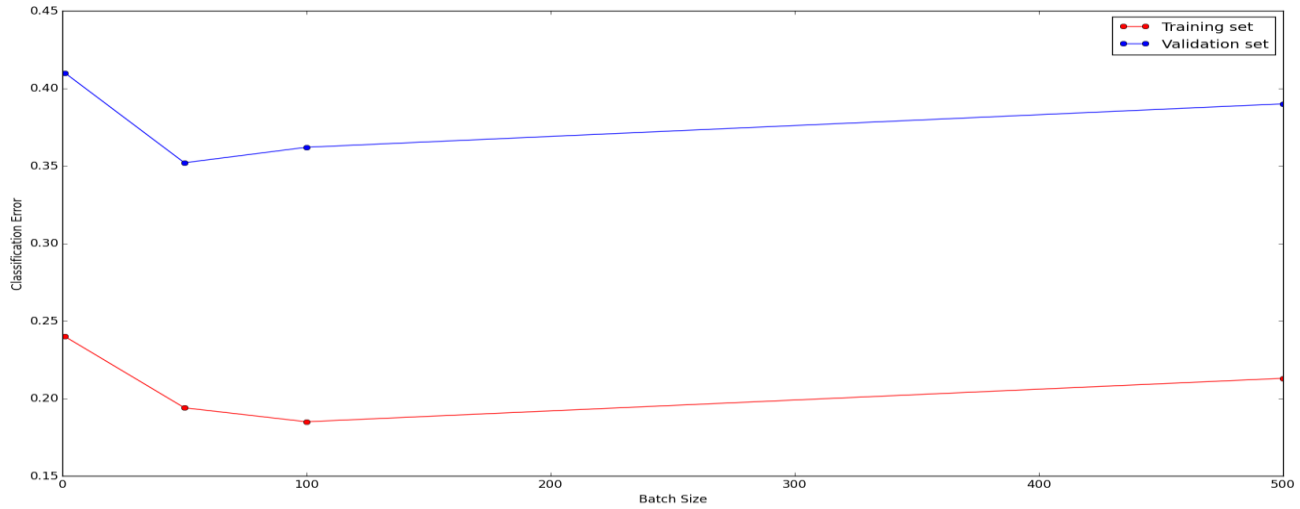


Figure 9: Learning Batch Size vs Classification Error Rates

We see that having a bigger batch size makes learning faster, but too big of a size might cause only a few updates to be made, and some changes in the model due to different samples are cancelled out before an actual update is made. We get the best results with a batch size of 50, where the training error went down to 19%, and validation error to 35%.

Finally, we try some regularization. There are two options, Dropout and Weight decay. We tried both options, and obtained the following results:

Regularization	Training Error	Validation Error
L2 Weight Decay	0.164	0.372
Dropout	0.248	0.414

Table 1: Regularization in Neural Networks

We have a good validation error, but not the best that we have had until now. Hence, we take the better option, L2 weight decay, and experiment with the various weight decay coefficients. We obtained the following results:

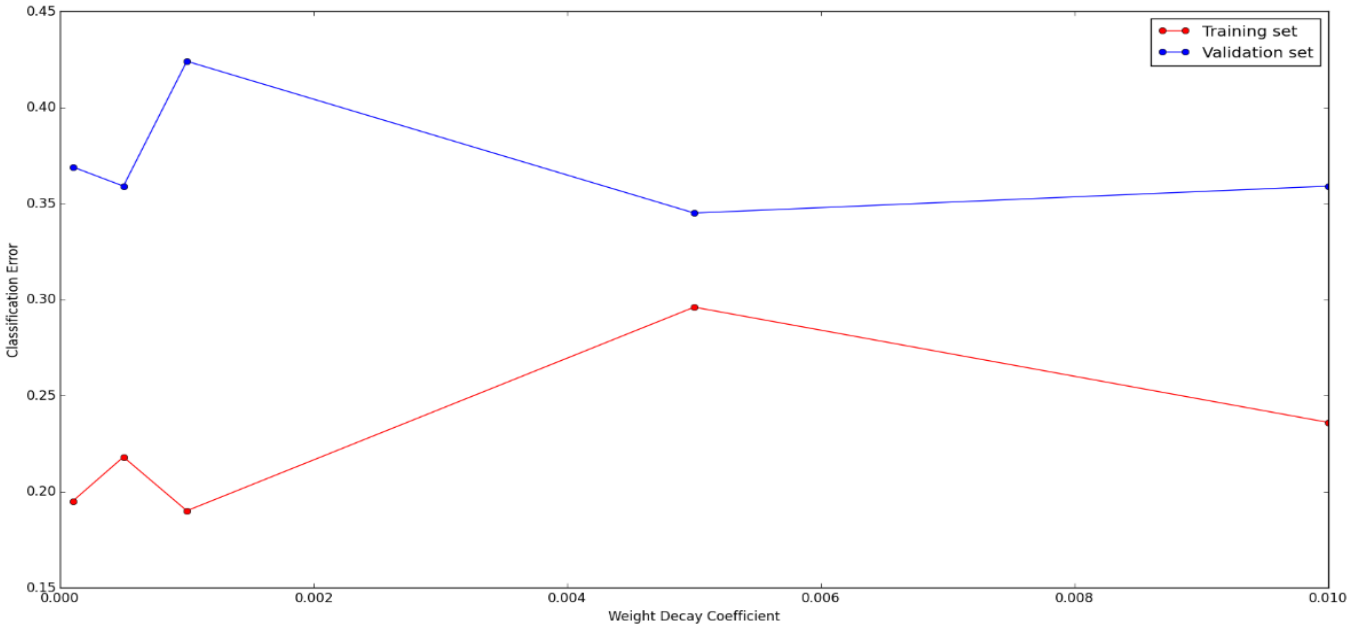


Figure 10: Weight Decay Coefficient vs Classification Error Rates

We find that the best value is around 0.005. The training and validation errors are close, and so there is very less overfitting. The validation error was brought down to 34%.

Finally, we combine all the best parameters from the above experiments to get a test error rate of about 37%. When compared to the previous models like Logistic regression and SVM, this is not very good. Considering the complexity of this model, those models literally perform better. Hence, we can conclude that this model is not ideal to solve this problem, although neural networks are famous to be good at image classification, like objects in the image, different scenes depicted in the image, etc. This model needs more than just the pixels to classify the images better.

Model 5: Gaussian Mixture Models

The last model we try is Gaussian Mixture Models/Mixture of Gaussians. This is the Probabilistic Graphical Model that we will be using in this project. Typically, we have k components trying to cluster the data, and each cluster is modeled by a Gaussian. Each input sample is distributed between the components. That is, each component has a probability that it generated the sample. The probabilities sum up to 1 and the component with the maximum probability is assigned to the sample.

The model we used is available as scikit learn's GMM, at http://scikit-learn.org/stable/modules/generated/sklearn.mixture.GMM.html#sklearn.mixture.GMM. We followed the example given in the documentation to create a mixture model. We had 7 clusters in the model, one for each class/expression. We set their means as the means images from each class, as shown in the example. So, assigning a component would result in assigning a class label to a sample. We then performed unsupervised learning on the model.

There are various ways how the different components are related to each other, which are captured by the covariance matrices between them. We had three types of covariance matrices that we could use in the model, and hence we tried all of them to choose the best one. We got the following results:

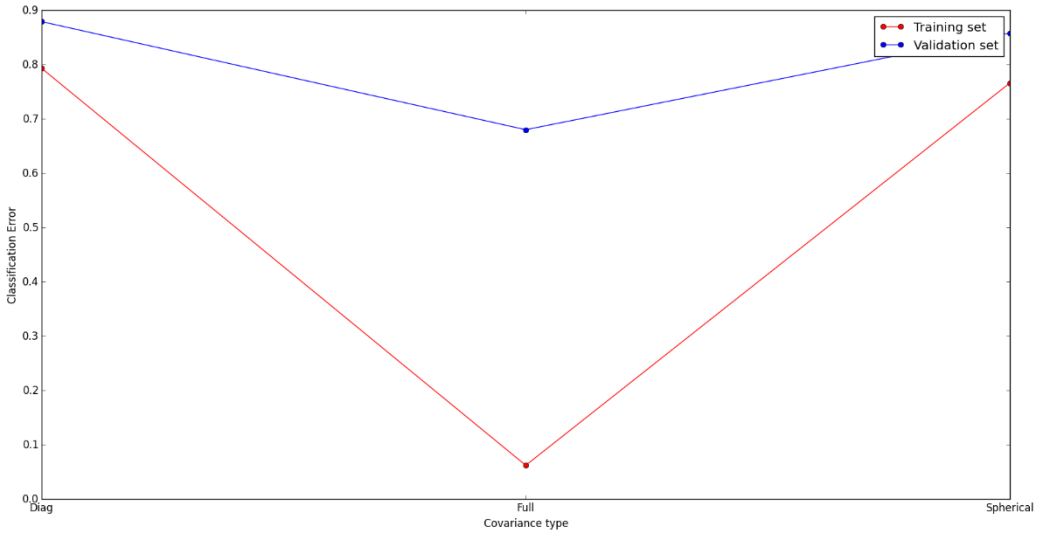


Figure 11: Covariance Matrix Type vs Classification Error Rates

We see that the model is performing terribly. Changing other parameters or using regularization techniques like weight decay, early stopping and dropout did not help. We thought we were doing something wrong. Hence, we tried to use another library, WEKA. Unfortunately, the results were not much different.

We were pretty sure that this model is not the right choice for the data. Hence, we tried something else. We did not set the means explicitly, but initialized those using k-Means, varying the 'k' a few times, which controls the number of components in the model. That did not help either.

We then thought maybe the input is not good for this type of model. Hence, we passed the input through a Canny Edge detector, to keep just the edges, which would be easier to differentiate. However, this method also failed to capture the differences between the classes, and we got around 85% validation error.

Lastly, we tried a completely different approach with GMMs. Something we used in another course (CSC401), to train one GMM for each class. We used 200 images from each class to train its corresponding GMM. We would then predict the log probability of each test sample under each GMM, and choose the class with the maximum value.

We got the following results when we tried to experiment with the number of components in the GMM of each class:

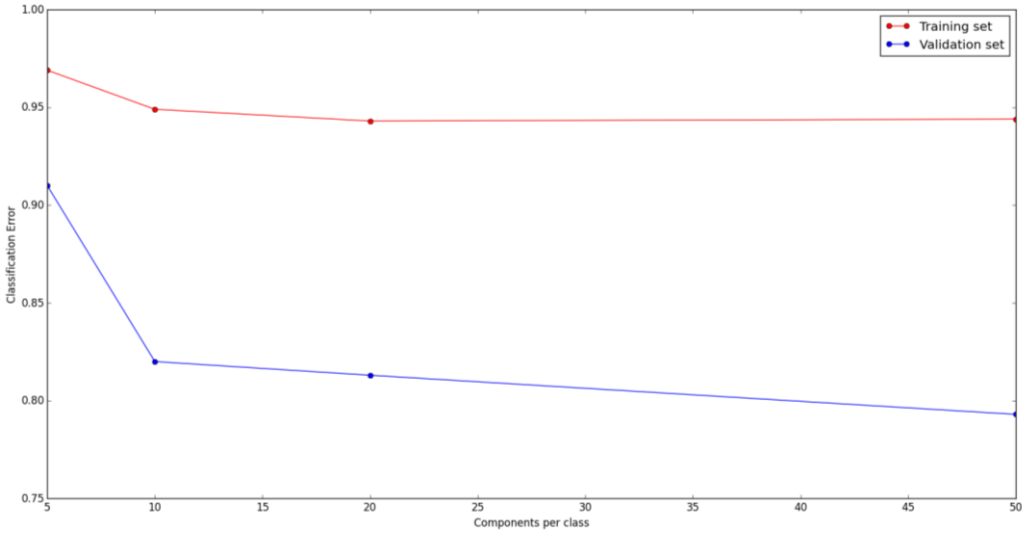


Figure 12: Components per class vs Classification Error Rates

We see that as we have more components for each class, the error decreases, because we can capture more features in the image, but it is still very bad. The models were not able to capture the differences in the classes that efficiently, and it seems that something more than just the raw pixel values are needed. Hence, we can conclude that GMM is definitely not the model we should use while predicting facial expressions based on the pixel values in the image.

Excluded Models

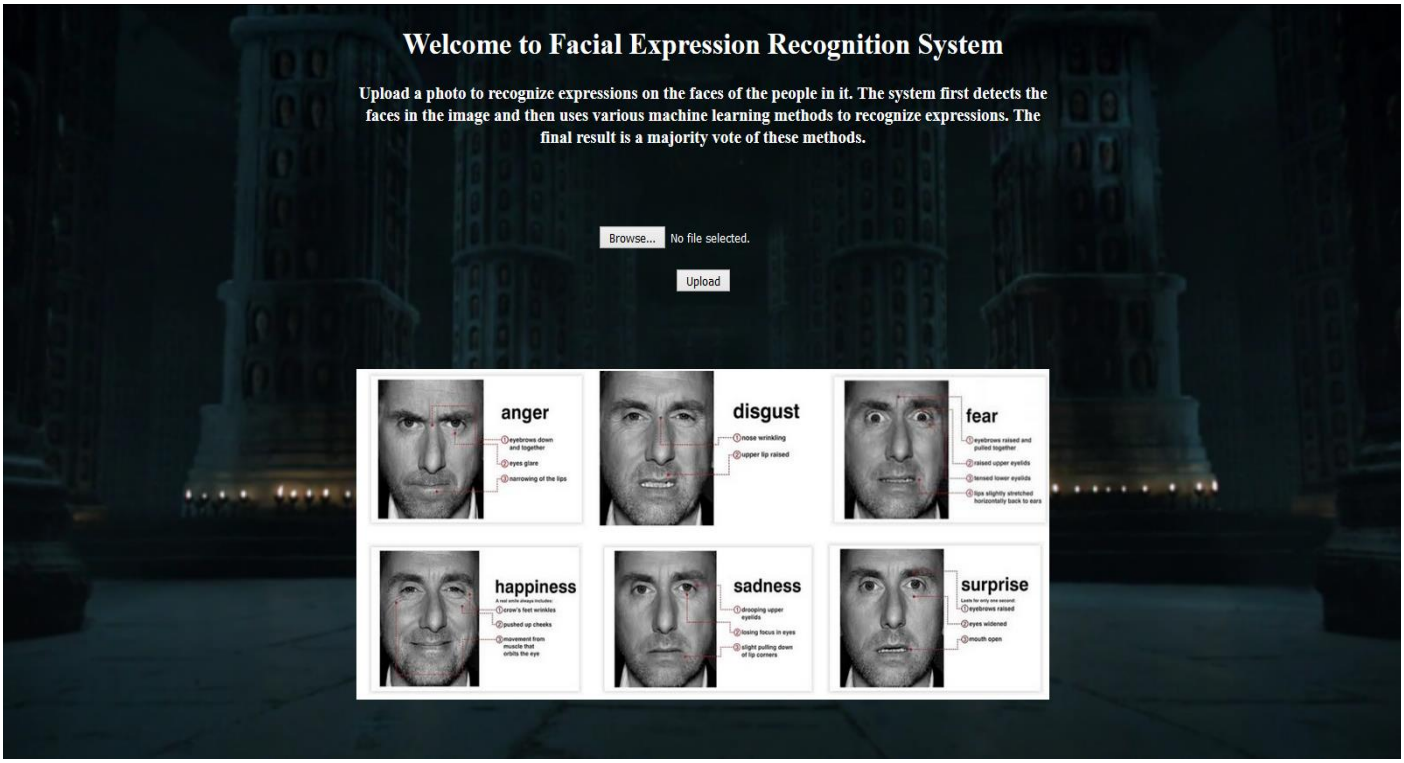
Although we have a lot of models to compare here, we have excluded some of the models and experiments we conducted because their results were not significantly better and a similar model was present in the above mentioned list.

One such model was linear regression, whose performance was like logistic regression and the linear SVM. Since these two similar models were present, we chose to exclude it. Another model was the general SVM, which we also excluded because Linear SVM was present and performing similarly.

One preprocessing step that we took out was PCA. We tried to reduce the dimension of the data, but it hardly brought about any changes in performance. In some cases, it even reduced the accuracy. Hence, it was better to not perform PCA on the data before passing it into these models. It also saved some computational power.

The Interface

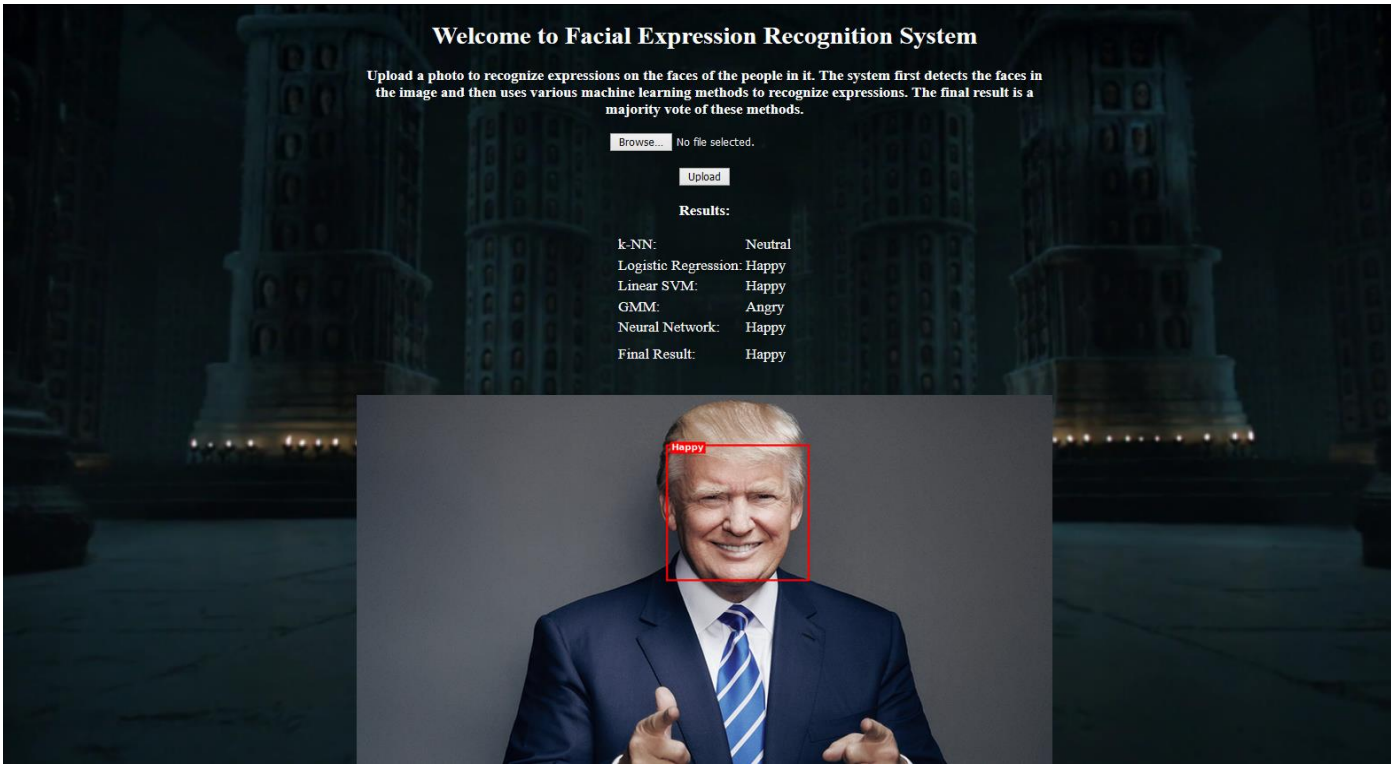
As mentioned before, a very important aspect of this project was to provide an interface to non-technical people to use this system based on various complex machine learning models. For that, a website was built where the users can visit and upload an image of their liking to predict the expressions on the faces of the people present in it. Below is a screen shot of the interface where the user uploads the image:



It provides an option to browse the computer to upload an image. On the page is another image, which shows the different expressions, and the markers of each expression on the face. It should be noted that these markers are to help the user identify the expressions in the photo, and are not used in prediction by the methods.

On a side note, the photos with the markers are of an actor who plays the role of an expert in lie detection for the police in a TV series called ‘Lie To Me’, who detects lies by observing a person’s facial expressions. It was one of the motivations behind this project.

The classifier used to detect and extract faces in the uploaded image is Haar Cascade Classifier. This classifier is available from OpenCV, at http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html. It best detects frontal poses. Once the face patch has been extracted from the image, it is preprocessed to be a 32x32 grayscale image. Then, it is passed into each of the 5 models listed above one by one, to get their prediction. The final prediction is determined by a majority vote. Lastly, a box is drawn around the detected face patch, labeled with the prediction from the models, and displayed to the user, as shown in the image below:



Conclusion

We chose a project that was interesting for us, while being a cool thing to demo to the non-technical people we know. We tried to use a new approach to recognize emotions in images by analyzing the raw pixels in the images. As discussed before, this is very different technique as compared to the one listed in the papers we used as reference, mainly on the speed metric.

We started with some basic models, like k-NN, Logistic regression and SVMs. Then, we moved on to the complex models like Neural Network and GMMs. The basic models did quite good on the data, but the complex models did not perform as well as we expected them to. While the Neural Network still did a decent job, GMM was clearly a bad choice for this task. We thought each Gaussian, or their mixture would be able to capture an expression, but it did not turn out to be like that. We can say that the final ranking of the models can be SVM > Logistic Regression > Neural Network > kNN > GMM.

We also made the interface which is a great way to interact with this complex system, even for our non-technical friends.

In the end, we can say that we had fun working on this project, trying to implement what we saw only in TV and movies. We learnt a lot of stuff, like comparing different models, producing graphs, writing a (semi-) formal report, and most importantly, not all models that perform well on some data are suited for all tasks, like our GMMs here. We just worked on them in CSC401 for speaker identification, and got excellent results, but they did not do well at all in this task.

We would like to continue working on this project in the future, where we have more time, and resources. We would try to change the input to these models, by performing feature extractions. Some good features might be the position of the eyebrows, shape of the eyes, shape of the lips, nose width, etc. as mentioned in the introduction.

References

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

2. Xiaoming Zhao, Xugan Shi, Shiqing Zhang. (2015) Facial Expression Recognition via Deep Learning. IETE Technical Review 32347-355.
   Online publication date: 3-Sep-2015.

3. Ping Liu, Shizhong Han, Zibo Meng, Yan Tong. (2014) Facial Expression Recognition via a Boosted Deep Belief Network. 2014 IEEE Conference on Computer Vision and Pattern Recognition pp. 1805-1812.

4. Maryam Sabzevari, Saeed Toosizadeh, Saeed Rahati Quchani, Vahid Abrishami. (2010) A fast and accurate facial expression synthesis system for color face images using face graph and deep belief network. 2010 International Conference on Electronics and Information Engineering V2-354-V2-358.

5. The OpenCV Library Dr. Dobb's Journal of Software Tools (2000) by G. Bradski

6. Toronto Faces Dataset (http://aclab.ca/users/josh/TFD.html) by Josh Susskind