**CSC 412 Project Report  
Facial Expression Prediction**

Dhyey Sejpal Shahin Yousefi  
1000162986   
dhyey.sejpal@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.

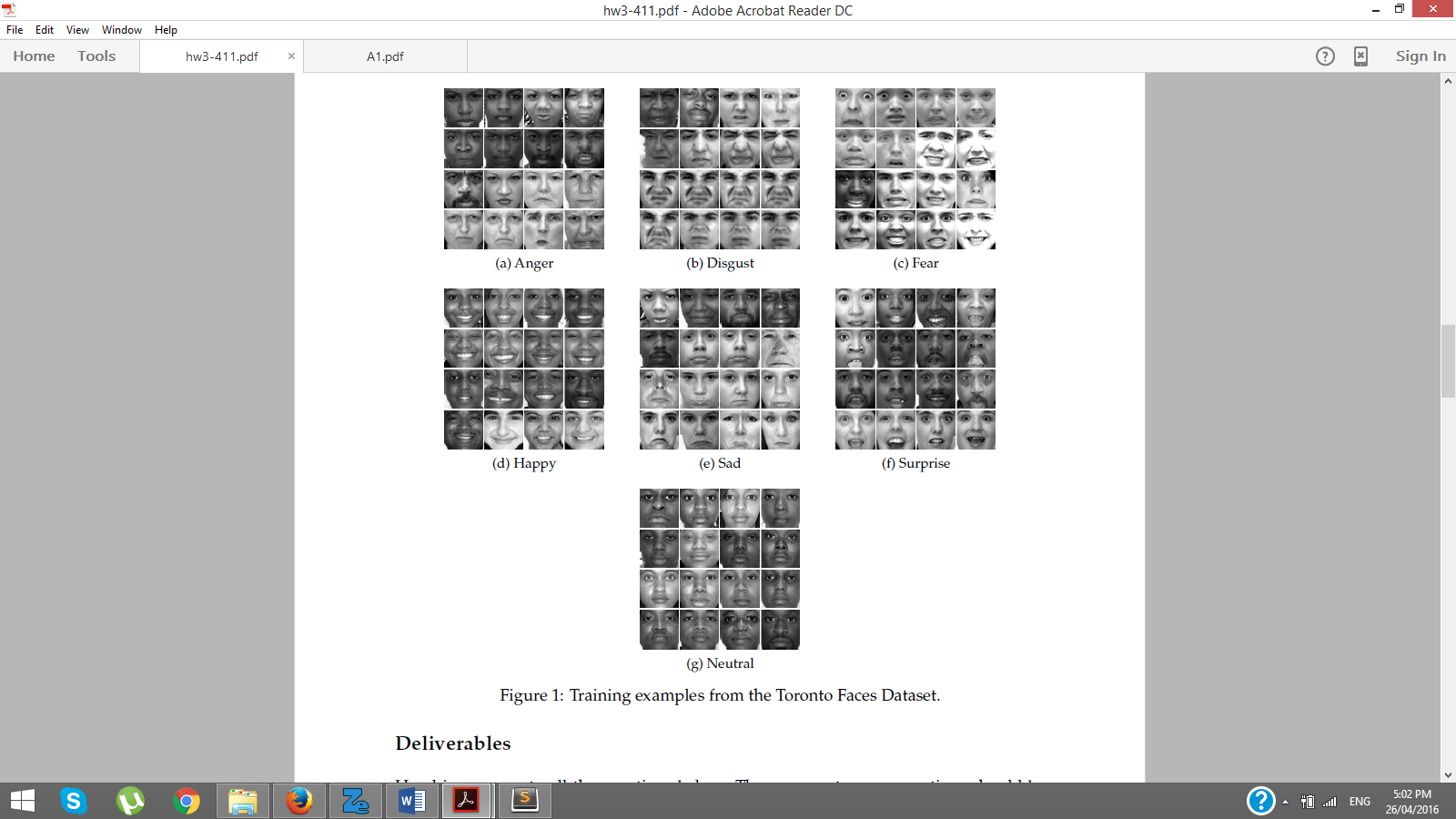
Hence, we try a very simple approach here. We take in the raw pixels of the image and try to learn their values to classify the various images into different classes, or expressions. We will be using the training data from Toronto Faces DataSet. This set has 2925 labeled, 32x32, grayscale images. We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

**Experiments and Results**

As mentioned before, we had 2925 labeled images, and we will be performing supervised 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:  
  
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 try is 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. What we definitely knew was that the data wasn’t ordered and stacked according to the classes, and hence leaving out the last few examples in the validation and test set won’t end up in having all of them from the same class, and excluding that entire class from training. 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, and if a model recognizes a particular class better, then the validation and test sets won’t favor it.  
  
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

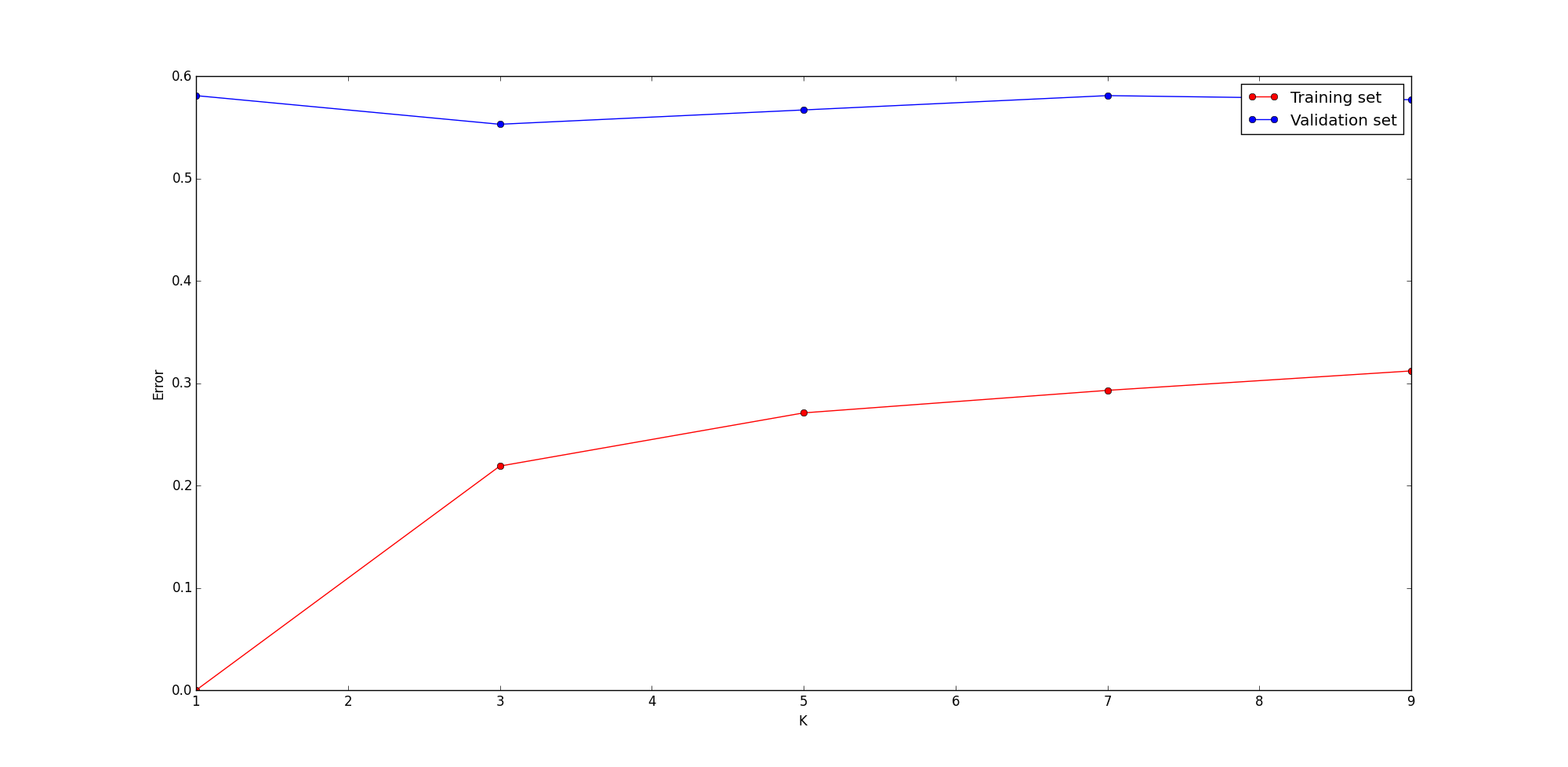
This is the standard k-Nearest Neighbors model, where the training data is projected into a d-dimensional space (d is the number of features in the data, 1024 here), and then each test example is projected to the same space. Then, we use a distance metric to get the training examples that are the closest 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 here 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, 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. We experimented with the values of k, and found the results shown below.

Figure 2: Different values of k in kNN vs Error rates

The x-axis shows the different values of k we tried, along with their classification error on the y-axis.   
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 influence 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 around 55%.   
  
The model is not very powerful, which is shown by the high training error as well, and around 45% accuracy on the test and validation set proves it true. 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

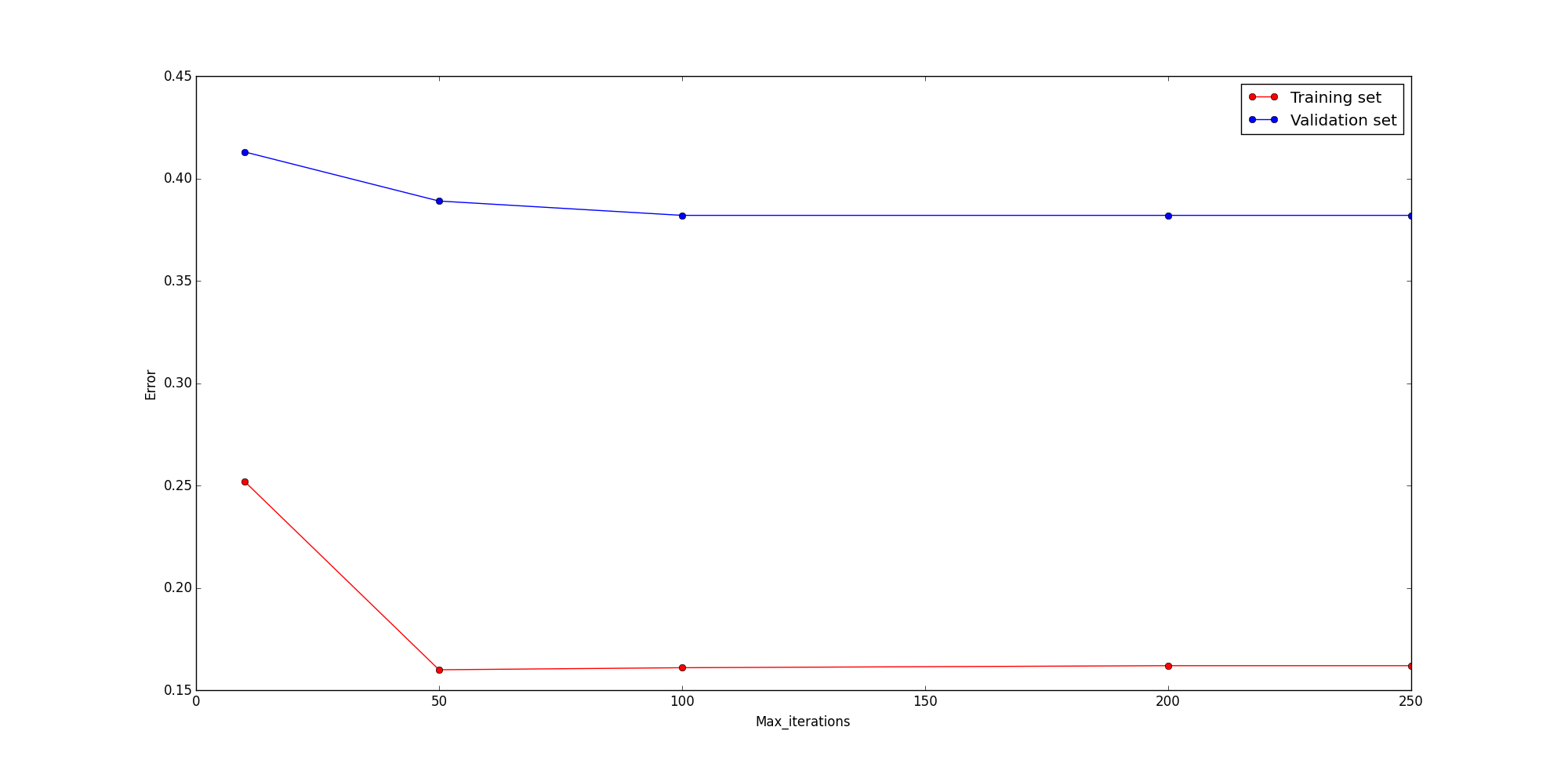
The next model we try is another simple but common model. Logistic regression. It is often used for binary classification tasks. The formula used is , where z = . Here, wi represent the weights and xi are the features in the input sample, which are the pixels in the image. We are using the sklearn library in Python to solve this problem using Logistic Regression, 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 is 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 over fitting, 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 shown in the following graph:

Figure 3: Maximum number of optimization iterations vs Error Rates

The x-axis shows the upper limit on number of iterations of the optimization algorithm, while the y-axis shows the 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 in 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.

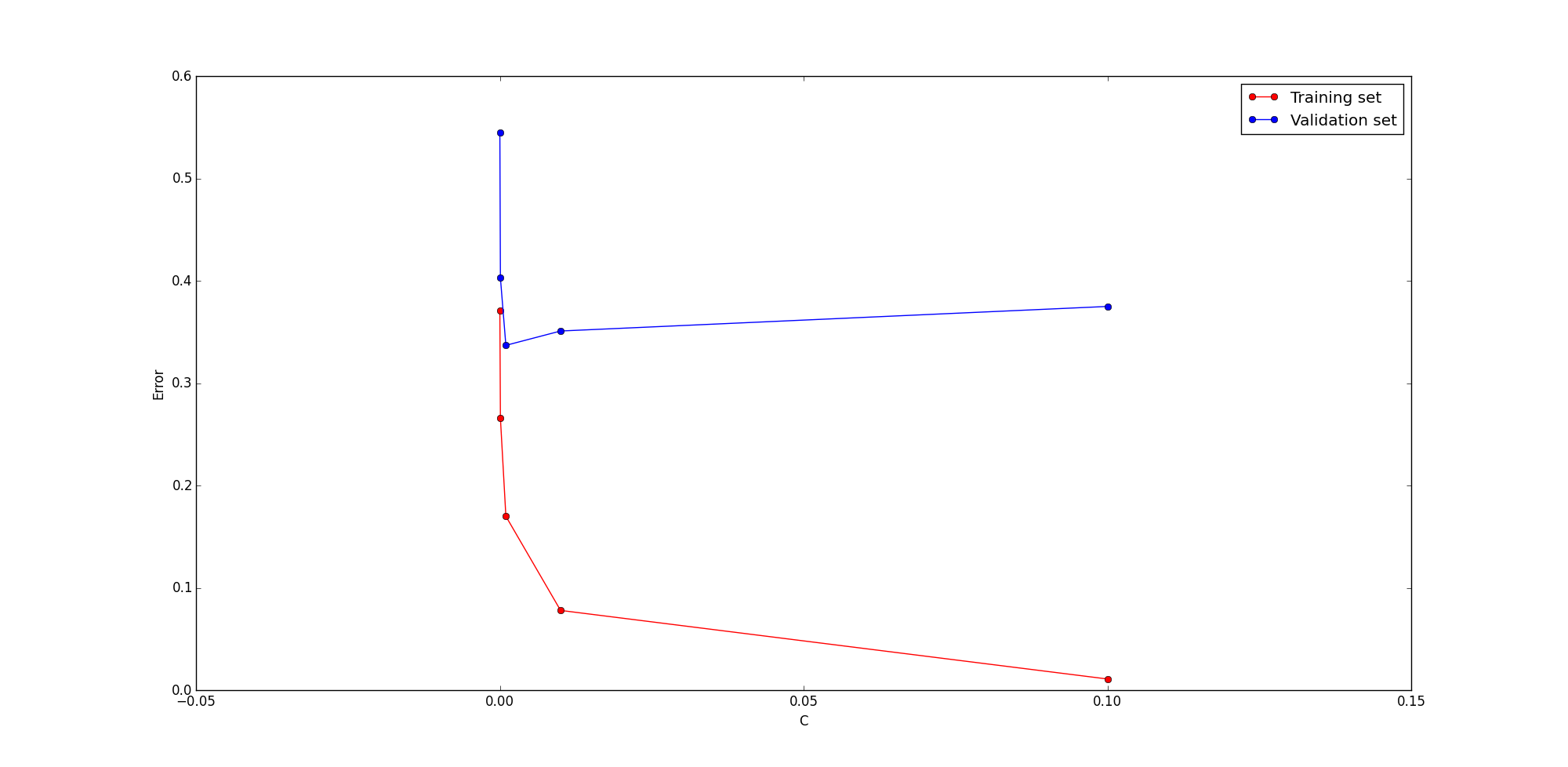
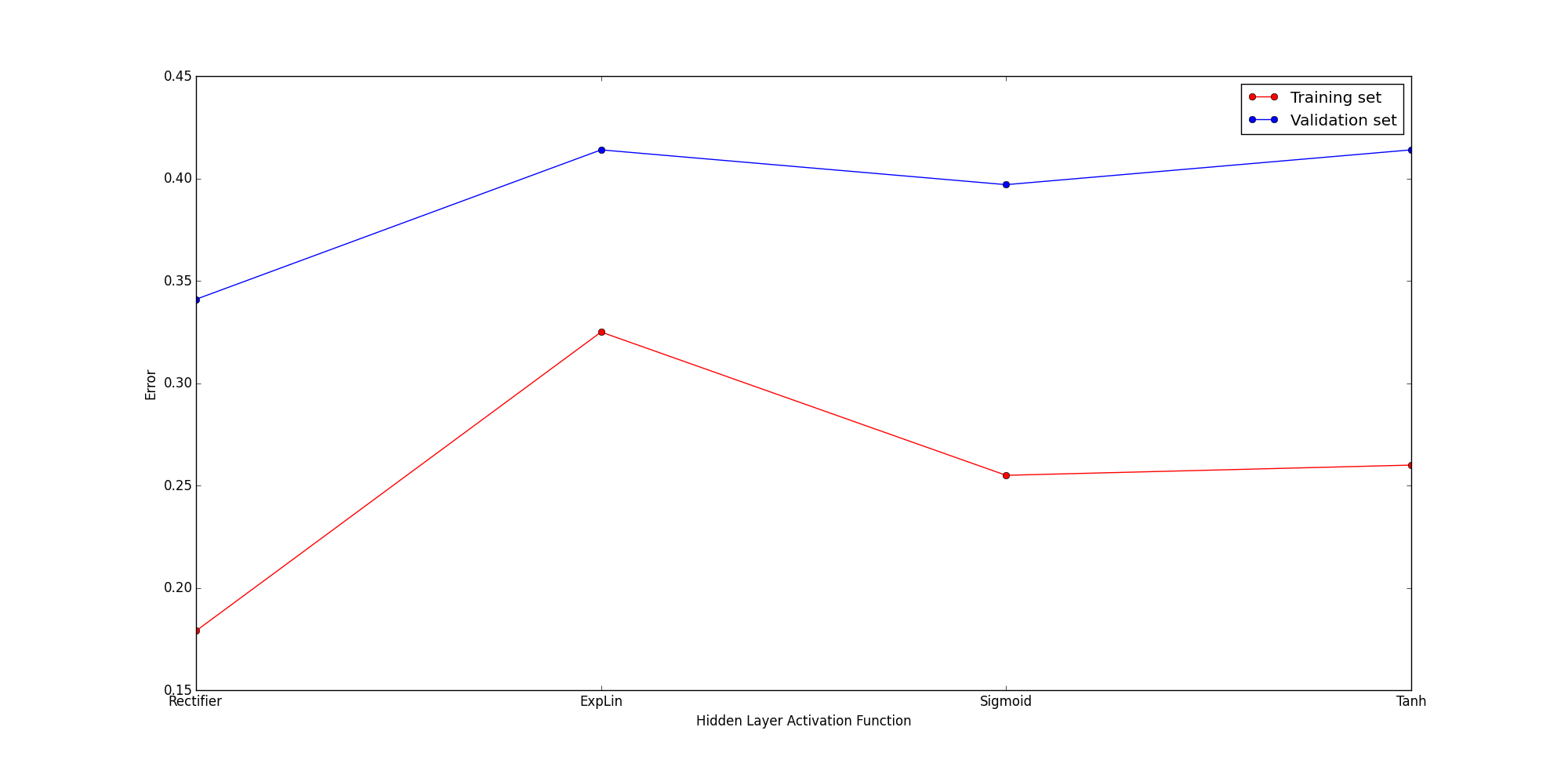
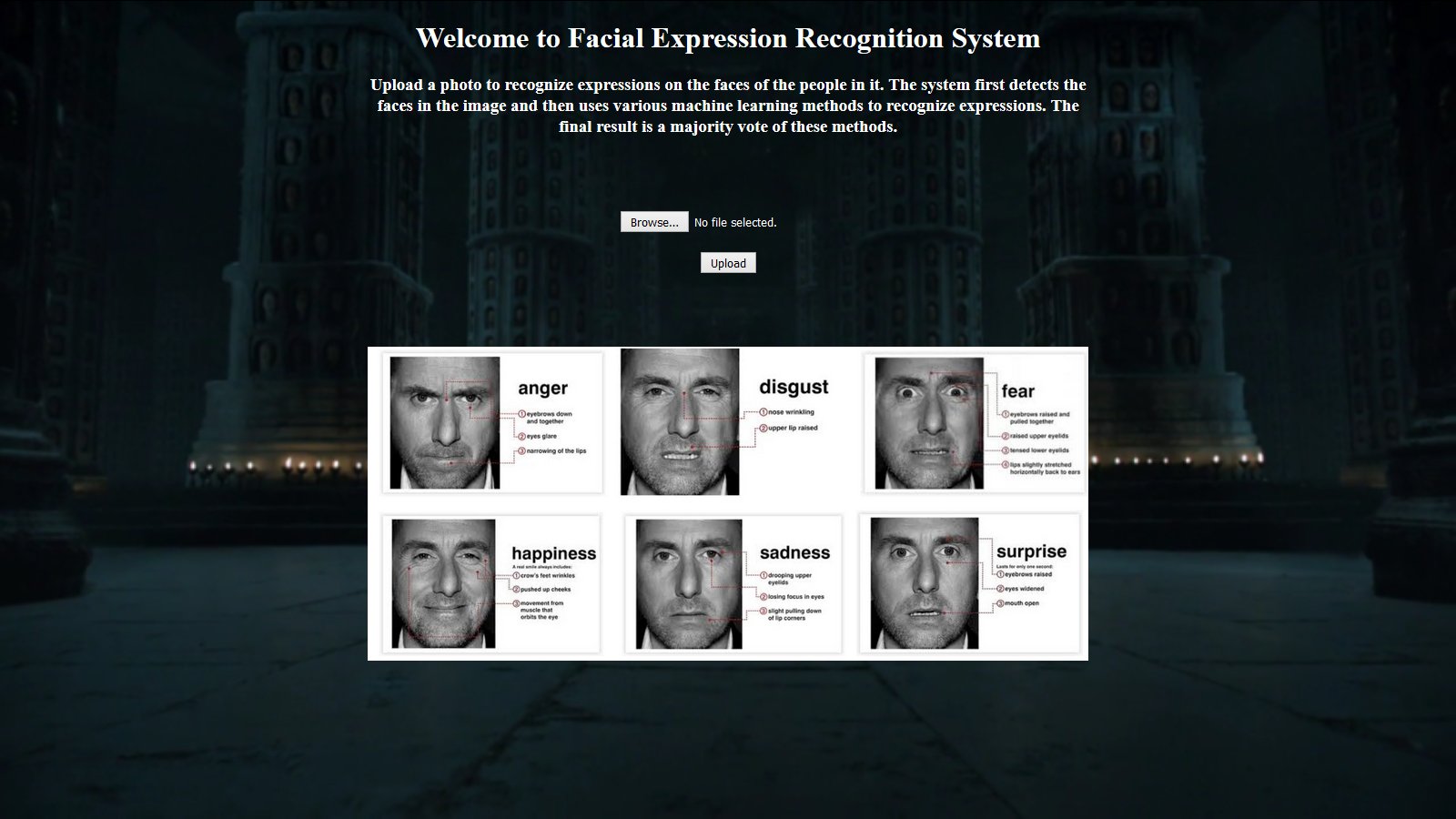
The model being used here is LinearSVC from sklearn library, available at http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html. It does classification using one-vs-rest technique.   
  
We are using linear SVMs, as they performed better than any other kernels (e.g. Polynomial, 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 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 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. 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 Error Rates

The x-axis shows the different functions used as hidden layer activation functions, Relu ((x) = max(0, x)), ExpLin, 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%, and the test error was around 45%.   
  
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:  
  
Model 5: Gaussian Mixture Models

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 Linear Regression. 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 saves a lot of 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 has been 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 the image. 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 that 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.   
  
The next screen shot shows an actually uploaded image and the face detections, with their expressions predicted as their labels.   
  
The classifier used to detect faces in the 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 ace 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 above.   
  
 **Conclusion**