

CHAPTER-1

INTRODUCTION

Face plays significant role in social communication. This is a 'window' to human personality, emotions and thoughts. According to the psychological research conducted by Mehrabian, nonverbal part is the most informative channel in social communication. Verbal part contributes about 7% of the message, vocal – 34% and facial expression about 55%. Due to that, face is a subject of study in many areas of science such as psychology, behavioral science, medicine and finally computer science.

In the field of computer science much effort is put to explore the ways of automation the process of face detection and segmentation. Several approaches addressing the problem of facial feature extraction have been proposed. The main issue is to provide appropriate face representation, which remains robust with respect to diversity of facial appearances.

The objective of this report is to outline the problem of facial expression recognition, and highlight the fact that it is a great challenge in the area of computer vision. Advantages of creating a fully automatic system for facial action analysis are constant motivation for exploring this field of science and will be mentioned in this report.

First Chapter is devoted to the psychological background of the facial expression recognition problem. The motivation of such study is outlined from the psychological aspect. Moreover, the techniques used by psychologists for facial action analysis are presented. Second chapter shows the idea of facial expression recognition system, the way such system is composed and its main features and requirements. Furthermore, approaches proposed in the literature will be described briefly. Finally, the application areas will be mentioned to show that automatic facial action recognition is widely used. In the third chapter, I will present the design and implementation of Facial Expression Recognition System. Techniques used at each stage of my system will be described and explained. Moreover, the performance of a system will be evaluated by testing the recognition accuracy on two training sets.

In 1978, Ekman introduced the system for measuring facial expressions called FACS – Facial Action Coding System. FACS was developed by analysis of the relations between muscle(s) contraction and changes in the face appearance caused by them. Contractions of muscles were responsible for the same action are marked as an Action Unit (AU).

The task of expression analysis with use of FACS is based on decomposing observed expression into the set of Action Units. There are 46 AUs that represent changes in facial expression and 12 AUs connected with eye gaze direction and head orientation. Action Units are highly descriptive in terms of facial movements, however, they do not provide any information about the message they represent. AUs are labeled with the description of the action (Fig.1.1).













Upper Face Action Units		
AU4	AU1+4	AU1+2
		
Brows lowered and drawn together	Medial portion of the brows is raised and pulled together	Inner and outer portions of the brows are raised
AU5	AU6	AU7
		
Upper eyelids are raised	Cheeks are raised and eye opening is narrowed	Lower eyelids are raised
Lower Face Action Units		
AU25	AU26	AU27
		
Lips are relaxed and parted	Lips are relaxed and parted; mandible is lowered	Mouth is stretched open and the mandible pulled down
AU12	AU12+25	AU20+25
		
Lip corners are pulled obliquely	AU12 with mouth opening	Lips are parted and pulled back laterally

Fig. 1.1: Examples of Action Units

Facial expression described by Action Units can be then analyzed on the semantic level in order to find the meaning of particular actions. According to the Ekman's theory, there are six basic emotion expressions that are universal for people of different nations and cultures. Those basic emotions are joy, sadness, anger, fear, disgust and surprise (Fig. 1.2).

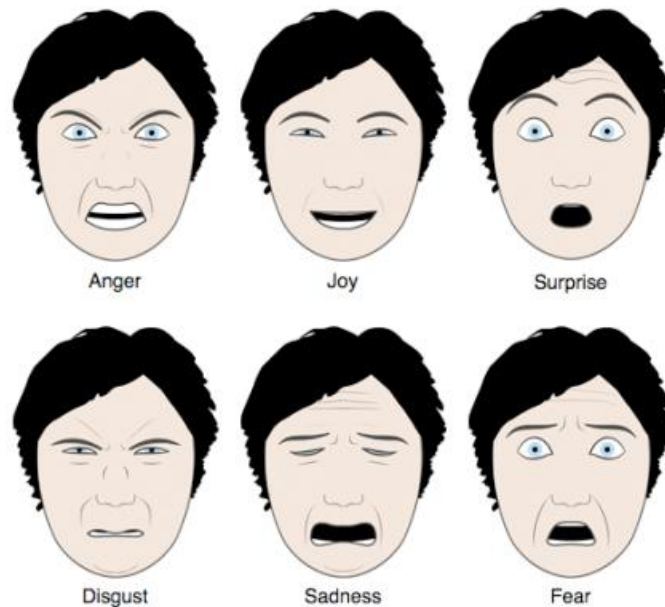


Fig. 1.2: Six universal emotions

The Facial Action Coding System was developed to help psychologists with face behavior analysis. Facial image was studied to detect the Action Units occurrences and then AU combinations were translated into emotion categories. This procedure required much effort, not only because the analysis was done manually, but also because about 100 hours of training were needed to become a FACS coder. That is why FACS was quickly automated and replaced by different types of computer software solutions.

CHAPTER-2

LITERATURE REVIEW

2.1 Introduction to Machine Learning

Machine Learning (ML) is the science of getting computers to act without being explicitly programmed. In the past decade, machine learning has given us self-driving cars, practical speech recognition, effective web search, and a vastly improved understanding of the human genome. Machine learning is so pervasive today that you probably use it dozens of times a day without knowing it. Many researchers also think it is the best way to make progress towards human-level AI.

Machine learning algorithms can figure out how to perform important tasks by generalizing from examples. This is often feasible and cost-effective where manual programming is not. As more data becomes available, more ambitious problems can be tackled. As a result, machine learning is widely used in computer science and other fields. Machine learning systems automatically learn programs from data. This is often a very attractive alternative to manually constructing them, and in the last decade the use of machine learning has spread rapidly throughout computer science and beyond. Machine learning is used in Web search, spam filters, recommender systems, ad placement, credit scoring, fraud detection, stock trading, drug design, and many other applications.

Among the different types of ML tasks, a crucial distinction is drawn between supervised and unsupervised learning:

- Supervised machine learning: The program is “trained” on a pre-defined set of “training examples”, which then facilitate its ability to reach an accurate conclusion when given new data. Supervised learning algorithms are trained using labeled examples, such as an input where the desired output is known. For example, a piece of equipment could have data points labeled either “F” (failed) or “R” (runs). The learning algorithm receives a set of inputs along with the corresponding correct outputs, and the algorithm learns by comparing its actual output with correct outputs to find errors. It then modifies the model accordingly. Through methods like

classification, regression, prediction and gradient boosting, supervised learning uses patterns to predict the values of the label on additional unlabeled data. Supervised learning is commonly used in applications where historical data predicts likely future events. For example, it can anticipate when credit card transactions are likely to be fraudulent or which insurance customer is likely to file a claim.

- **Unsupervised machine learning:** The program is given a bunch of data and must find patterns and relationships therein. Unsupervised learning is used against data that has no historical labels. The system is not told the "right answer." The algorithm must figure out what is being shown. The goal is to explore the data and find some structure within. Unsupervised learning works well on transactional data. For example, it can identify segments of customers with similar attributes who can then be treated similarly in marketing campaigns. Or it can find the main attributes that separate customer segments from each other. Popular techniques include self-organizing maps, nearest-neighbor mapping, k-means clustering and singular value decomposition. These algorithms are also used to segment text topics, recommend items and identify data outliers.

Machine Learning can be broadly classified into the following components

- **Representation.** A classifier must be represented in some formal language that the computer can handle. Conversely, choosing a representation for a learner is tantamount to choosing the set of classifiers that it can possibly learn. This set is called the hypothesis space of the learner. If a classifier is not in the hypothesis space, it cannot be learned. A related question, which we will address in a later section, is how to represent the input, i.e., what features to use.
- **Evaluation.** An evaluation function (also called objective function or scoring function) is needed to distinguish good classifiers from bad ones. The evaluation function used internally by the algorithm may differ from the external one that we want the classifier to optimize, for ease of optimization (see below) and due to the issues discussed in the next section.
- **Optimization.** Finally, we need a method to search among the classifiers in the language for the highest-scoring one. The choice of optimization technique is key to the efficiency of the learner, and also helps determine the classifier produced if the evaluation function

has more than one optimum. It is common for new learners to start out using off-the-shelf optimizers, which are later replaced by custom-designed ones.

There are five basic types of human expressions that have been considered in this report – **neutral, happy, surprise, disgust and anger**. This work investigates the application of Machine Learning algorithms such as Convolutional Neural Networks, Support Vector Machines and Logistic Regression to predict the emotion (mood) the user is displaying, along with further applications.

2.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standard multilayer neural network. The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units. In this article we will discuss the architecture of a CNN and the back propagation algorithm to compute the gradient with respect to the parameters of the model in order to use gradient based optimization.

A CNN consists of a number of convolutional and subsampling layers optionally followed by fully connected layers. The input to a convolutional layer is a $m \times m \times r$ image where m is the height and width of the image and r is the number of channels, e.g. an RGB image has $r=3$. The convolutional layer will have k filters (or kernels) of size $n \times n \times q$ where n is smaller than the dimension of the image and q can either be the same as the number of channels r or smaller and may vary for each kernel. The size of the filters gives rise to the locally connected structure which are each convolved with the image to produce k feature maps of size $m-n+1$. Each map is then subsampled typically with mean or max pooling over $p \times p$ contiguous regions where p ranges between 2 for small images (e.g. MNIST) and is usually not more than 5 for larger inputs. Either before or after the subsampling layer an additive bias and sigmoidal nonlinearity is

applied to each feature map. The figure below illustrates a full layer in a CNN consisting of convolutional and subsampling sublayers. Units of the same color have tied weights.

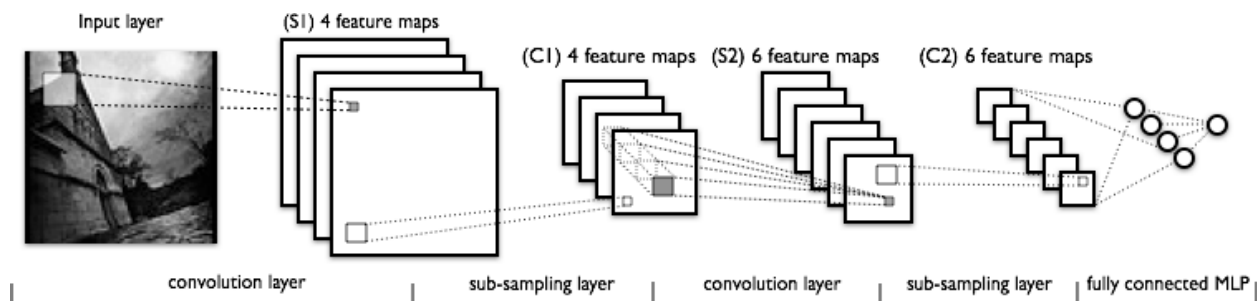


Fig 2.1: General architecture of a Convolutional Neural Network

After the convolutional layers there may be any number of fully connected layers. The densely connected layers are identical to the layers in a standard multilayer neural network.

2.2.1 Fully Connected Networks

In the sparse autoencoder, one design choice that we had made was to “fully connect” all the hidden units to all the input units. On the relatively small images that we were working with (e.g., 8x8 patches for the sparse autoencoder assignment, 28x28 images for the MNIST dataset), it was computationally feasible to learn features on the entire image. However, with larger images (e.g., 96x96 images) learning features that span the entire image (fully connected networks) is very computationally expensive—you would have about 104104 input units, and assuming you want to learn 100 features, you would have on the order of 106106 parameters to learn. The feedforward and backpropagation computations would also be about 102102 times slower, compared to 28x28 images.

2.2.2 Locally Connected Networks

One simple solution to this problem is to restrict the connections between the hidden units and the input units, allowing each hidden unit to connect to only a small subset of the input units. Specifically, each hidden unit will connect to only a small contiguous region of pixels in the input. (For input modalities different than images, there is often also a natural way to select “contiguous groups” of input units to connect to a single hidden unit as well; for example, for

audio, a hidden unit might be connected to only the input units corresponding to a certain time span of the input audio clip.)

This idea of having locally connected networks also draws inspiration from how the early visual system is wired up in biology. Specifically, neurons in the visual cortex have localized receptive fields (i.e., they respond only to stimuli in a certain location).

2.2.3 Convolutions

Natural images have the property of being “stationary”, meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations.

More precisely, having learned features over small (say 8x8) patches sampled randomly from the larger image, we can then apply this learned 8x8 feature detector anywhere in the image. Specifically, we can take the learned 8x8 features and “convolve” them with the larger image, thus obtaining a different feature activation value at each location in the image.

To give a concrete example, suppose you have learned features on 8x8 patches sampled from a 96x96 image. Suppose further this was done with an autoencoder that has 100 hidden units. To get the convolved features, for every 8x8 region of the 96x96 image, that is, the 8x8 regions starting at (1,1),(1,2),...,(89,89)(1,1),(1,2),...,(89,89), you would extract the 8x8 patch, and run it through your trained sparse autoencoder to get the feature activations. This would result in 100 sets 89x89 convolved features.

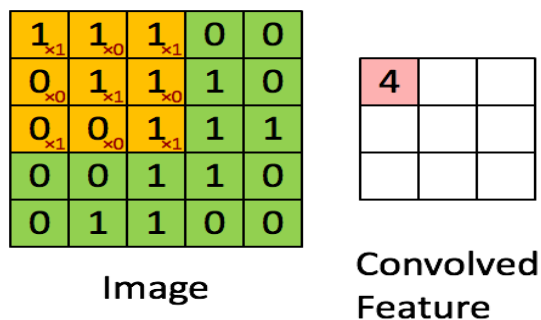


Fig 2.2: Generation of Features using Convolution

2.2.4 Pooling

After obtaining features using convolution, we would next like to use them for classification. In theory, one could use all the extracted features with a classifier such as a softmax classifier, but this can be computationally challenging. Consider for instance images of size 96x96 pixels, and suppose we have learned 400 features over 8x8 inputs. Learning a classifier with inputs having 3+ million features can be unwieldy, and can also be prone to over-fitting.

To address this, first recall that we decided to obtain convolved features because images have the “stationarity” property, which implies that features that are useful in one region are also likely to be useful for other regions. Thus, to describe a large image, one natural approach is to aggregate statistics of these features at various locations. For example, one could compute the mean (or max) value of a particular feature over a region of the image. These summary statistics are much lower in dimension (compared to using all of the extracted features) and can also improve results (less over-fitting). We aggregation operation is called this operation “pooling”, or sometimes “mean pooling” or “max pooling” (depending on the pooling operation applied).

The following image shows how pooling is done over 4 non-overlapping regions of the image.

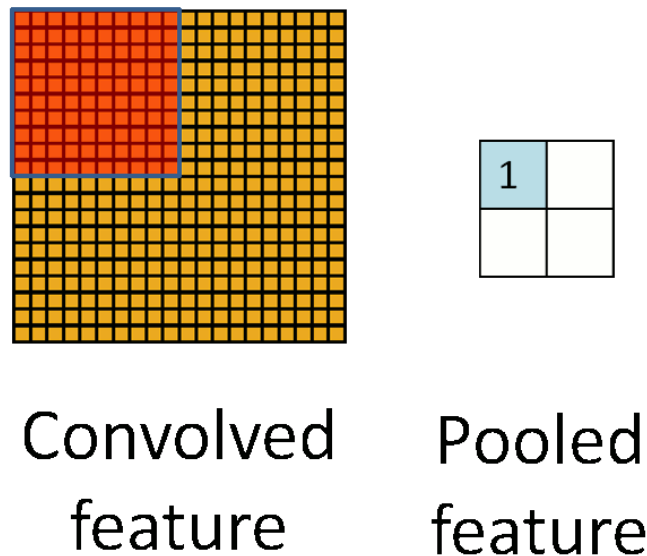


Fig 2.3 Using Pooling for Dimensionality Reduction

2.2.5 Pooling for Invariance

If one chooses the pooling regions to be contiguous areas in the image and only pools features generated from the same (replicated) hidden units. Then, these pooling units will then be ‘translation invariant’. This means that the same (pooled) feature will be active even when the image undergoes (small) translations. Translation-invariant features are often desirable; in many tasks (e.g., object detection, audio recognition), the label of the example (image) is the same even when the image is translated. For example, if you were to take an MNIST digit and translate it left or right, you would want your classifier to still accurately classify it as the same digit regardless of its final position.

2.2.6 Formal description

Formally, after obtaining our convolved features as described earlier, we decide the size of the region, say $m \times n$ to pool our convolved features over. Then, we divide our convolved features into disjoint $m \times n$ regions, and take the mean (or maximum) feature activation over these regions to obtain the pooled convolved features. These pooled features can then be used for classification.

CHAPTER 3

METHODOLOGY

Feature selection is concerned with choosing of a subset of features perfectly necessary to perform the classification task from a larger set of candidate features. The feature selection step has an effect on both the computational complexity and the quality of the classification results. It is essential that the information contained in the selected features is adequate to correctly verify the input class. Too many features may unnecessarily raise the complexity of the training and classification tasks, while a poor, inadequate selection of features may have a detrimental effect on the classification results. The process of selecting a sub set of features improves the efficiency of classifier and reduces execution time.

For the purpose of feature generation, OpenCV and DLib libraries have been used. The process of generating features can be broadly classified into 3 main categories-

1. Face Detection

The faces were detected from each face to remove unnecessary information from the images. To achieve this task, Haar Cascade Detection in OpenCV was used.

Object Detection using Haar feature-based cascade classifiers is an effective object detection. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images. OpenCV comes with a trainer as well as detector.

After successfully detecting the face from each image, the image of the face was cropped and saved and saved as a new image (150 X 150) for each detected face.

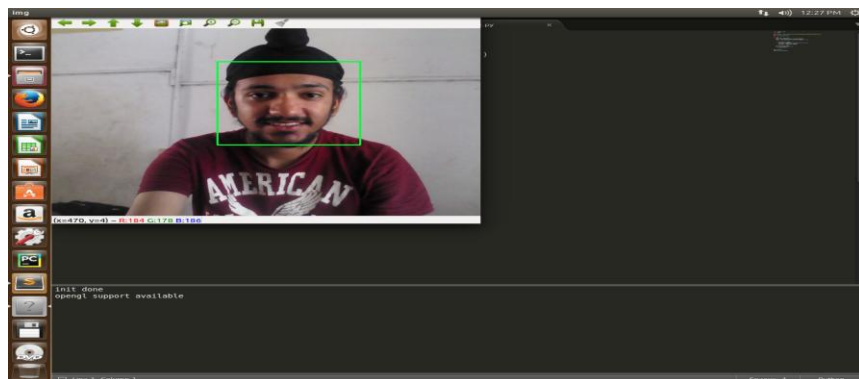


Fig 3.1: Face Detection implemented in OpenCV

2. Landmark Detection

The Landmark detection was achieved by training a Convolutional Neural Network to detect and track 68 Facial Landmarks for Pose and Gesture Estimation.

We find several effective ways to combine multiple convolutional networks. The first is multi-level regression. The face bounding box is the only prior knowledge for networks at the first level. The relative position of a facial point to the bounding box could vary in a large range due to large pose variations and the instability of face detectors. So the input regions of networks at the first level should be large in order to cover many possible predictions. But large input region is the major cause of inaccuracy because irrelevant areas included may degrade the final output of the network. The outputs of networks at the first level provide a strong prior for the following detections, i.e., the true position of a facial point should lie within a small region around the prediction at the first level. So the second level detection can be done within a small region, where the disruption from other areas is reduced significantly, and this process repeats. However, without context information, appearance of local regions is ambiguous and the prediction is unreliable. To avoid drifting, we should not cascade too many levels or trust the following levels too much. These networks are only allowed to adjust the initial prediction in a very small range.



Fig 3.2: Facial Landmarks

To further improve detection accuracy and reliability, we propose to jointly predict the position of each point with multiple networks at each level. These networks differ in input regions. The final predicted position of a facial point can be formally expressed as for an n-level cascade with I predictions at level i. Note that predictions at the first level are absolute positions while predictions at the following levels are adjustments.

This object is a tool that takes in an image region containing some object and outputs a set of point locations that define the pose of the object. The classic example of this is human face pose prediction, where we take an image of a human face as input and are expected to identify the locations of important facial landmarks such as the corners of the mouth and eyes, tip of the nose, and so forth.

The output of the shape_predictor function is 68 (x,y) values containing the values of the various facial landmarks such as eyes, eyebrows, nose etc in the Cartesian plane.

3. Feature Selection

After calculating the 68 (x,y) values of facial landmarks, their mean is calculated. This point is roughly the position of the nose tip in the face. After calculating the position of this central point, we calculate the distance and angle of each of the 68 points from this central point.

Faces may be tilted, which might confuse the classifier. We correct this by assuming that the bridge of the nose in most people is more or less straight, and offset all calculated angles by the angle of the nose bridge. This rotates the entire vector array so that tilted faces become similar to non-tilted faces with the same expression. Below are two images, the left one illustrates what happens in the code when the angles are calculated, the right one shows how we can calculate the face offset correction by taking the tip of the nose and finding the angle the nose makes relative to the image, and thus find the angular offset β we need to apply.

According to our algorithm, the 26th and the 29th Landmark detected correspond to the topmost and the bottommost point of the nose. Using this knowledge, we calculate the offset angle that we need to incorporate.

$$\text{If } x[26]=x[29]: \text{anglenose} = 0$$

$$\text{Else anglenose} = \arctan (y[26]-y[29])/(x[26]-x[29]) * 180/\pi$$

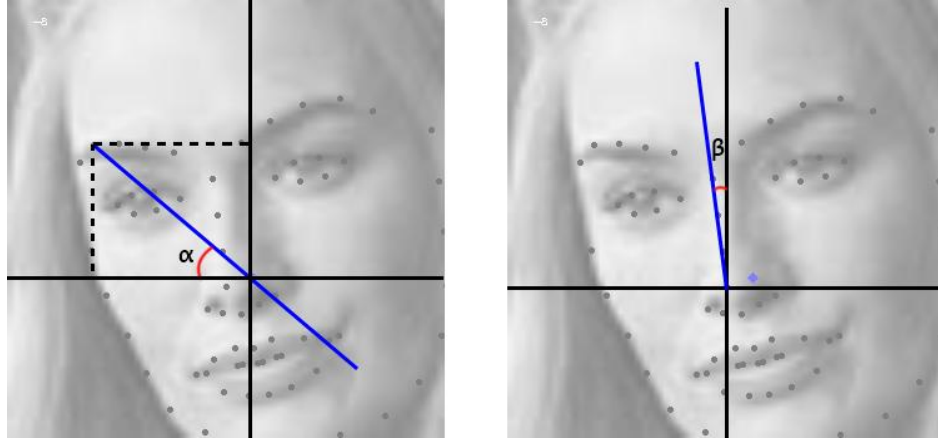


Fig 3.3: Computing the offset and calculating angles

Finally the features are selected as the angle between every detected point and the central point, after offsetting.

These features are then assigned labels according to their emotions. The extracted features and the corresponding labels are then used to train our classifier algorithm. The classifier, in turn, computes a mathematical model where all points are projected onto a feature space that has 68 different dimensions corresponding to the 68 Landmarks.

4. Dataset Generation

The final data is obtained using the feature generation process described above and consists of 565 rows and 67 columns. This is divided as 80% training data and 20% cross validation data.

CHAPTER 4

CLASSIFICATION ALGORITHMS

4.1 Support Vector Machines

In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

“Support Vector Machine” (SVM) is a supervised machine learning algorithm which can be used for both classification and regression challenges. However, it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n -dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiates the two classes very well.

In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

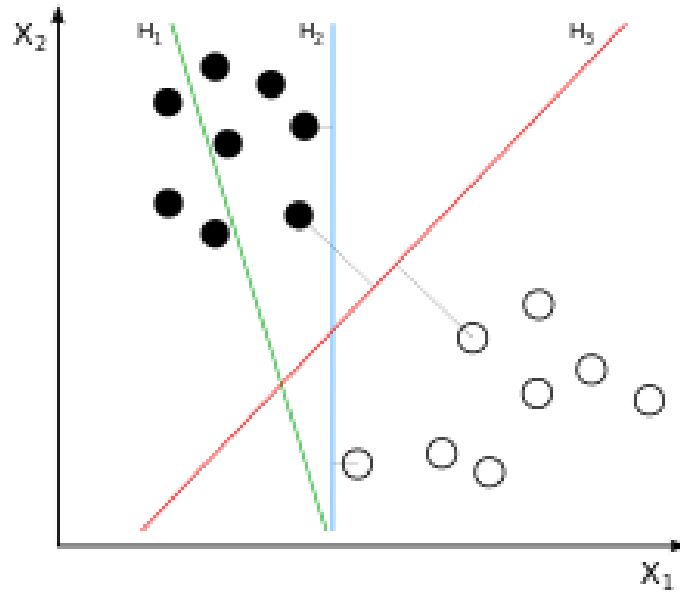


Fig 4.1: Possible separating hyperplanes

When data are not labeled, supervised learning is not possible, and an unsupervised learning approach is required, which attempts to find natural clustering of the data to groups, and then map new data to these formed groups. The clustering algorithm which provides an improvement to the support vector machines is called support vector clustering and is often used in industrial applications either when data are not labeled or when only some data are labeled as a preprocessing for a classification pass.

Working with neural networks for supervised and unsupervised learning showed good results while used for such learning applications. MLP's uses feed forward and recurrent networks. Multilayer perceptron (MLP) properties include universal approximation of continuous nonlinear functions and include learning with input-output patterns and also involve advanced network architectures with multiple inputs and outputs. There can be some issues noticed. Some of them are having many local minima and also finding how many neurons might be needed for a task is another issue which determines whether optimality of that NN is reached. Another thing to note is that even if the neural network solutions used tends to converge, this may not result in a unique solution.

There are many linear classifiers (hyper planes) that separate the data. However only one of these achieves maximum separation. The reason we need it is because if we use a hyper plane to classify, it might end up closer to one set of datasets compared to others and we do not want this

to happen and thus we see that the concept of maximum margin classifier or hyper plane as an apparent solution.

Support Vector Machine (SVM) is primarily a classifier method that performs classification tasks by constructing hyperplanes in a multidimensional space that separates cases of different class labels. SVM supports both regression and classification tasks and can handle multiple continuous and categorical variables. For categorical variables a dummy variable is created with case values as either 0 or 1.

In case of SVM, the cost function to be minimized is represented as

$$\min_{\theta} C \sum_{i=1}^m \left[y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

Unlike logistic, $h\theta(x)$ doesn't give us a probability, but instead we get a direct prediction of 1 or 0 i.e. if $\theta^T x$ is equal to or greater than 0 then $h\theta(x) = 1$, else $h\theta(x) = 0$.

For two-class, separable training data sets, there are lots of possible linear separators. Intuitively, a decision boundary drawn in the middle of the void between data items of the two classes seems better than one which approaches very close to examples of one or both classes. While some learning methods such as the perceptron algorithm find just any linear separator, others, like Naive Bayes, search for the best linear separator according to some criterion. The SVM in particular defines the criterion to be looking for a decision surface that is maximally far away from any data point. This distance from the decision surface to the closest data point determines the margin of the classifier. This method of construction necessarily means that the decision function for an SVM is fully specified by a (usually small) subset of the data which defines the position of the separator. These points are referred to as the support vectors (in a vector space, a point can be thought of as a vector between the origin and that point).

Maximizing the margin seems good because points near the decision surface represent very uncertain classification decisions: there is almost a 50% chance of the classifier deciding either way. A classifier with a large margin makes no low certainty classification decisions. This gives us a classification safety margin: a slight error in measurement or a slight document variation will not cause a misclassification. By construction, an SVM classifier insists on a large margin

around the decision boundary. Compared to a decision hyperplane, if you have to place a fat separator between classes, you have fewer choices of where it can be put. As a result of this, the memory capacity of the model has been decreased, and hence we expect that its ability to correctly generalize to test data is increased.

To implement the SVM algorithm in our project, we use the “kernel trick”.

In machine learning, kernel methods are a class of algorithms for pattern analysis, whose best known member is the support vector machine (SVM). The general task of pattern analysis is to find and study general types of relations (for example clusters, rankings, principal components, correlations, classifications) in datasets. For many algorithms that solve these tasks, the data in raw representation have to be explicitly transformed into feature vector representations via a user-specified feature map: in contrast, kernel methods require only a user-specified kernel, i.e., a similarity function over pairs of data points in raw representation.

Kernel methods owe their name to the use of kernel functions, which enable them to operate in a high-dimensional, implicit feature space without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between the images of all pairs of data in the feature space. This operation is often computationally cheaper than the explicit computation of the coordinates. This approach is called the "kernel trick". Kernel functions have been introduced for sequence data, graphs, text, images, as well as vectors.

Algorithms capable of operating with kernels include the kernel perceptron, support vector machines (SVM), Gaussian processes, principal components analysis (PCA), canonical correlation analysis, ridge regression, spectral clustering, linear adaptive filters and many others. Any linear model can be turned into a non-linear model by applying the kernel trick to the model: replacing its features (predictors) by a kernel function.

The idea of the kernel function is to enable operations to be performed in the input space rather than the potentially high dimensional feature space. Hence the inner product does not need to be evaluated in the feature space. We want the function to perform mapping of the attributes of the input space to the feature space. The kernel function plays a critical role in SVM and its performance.

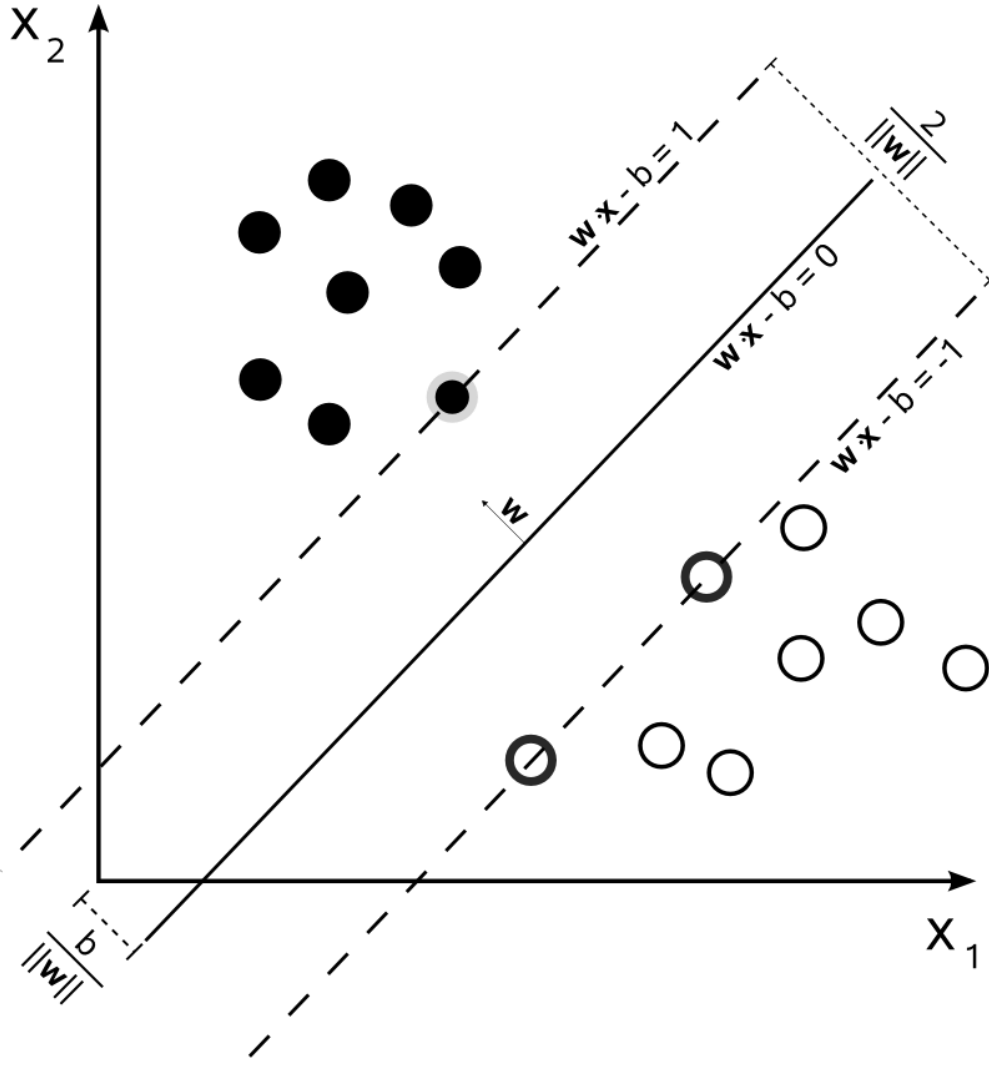


Fig 4.2: Computing the margin between two classes in order to maximize it.

Most kernel algorithms are based on convex optimization or eigenproblems and are statistically well-founded. Typically, their statistical properties are analyzed using statistical learning theory.

4.2 Deep Convolutional Neural Networks

In recent years by emerging powerful parallel processing hardware, Deep Neural Networks (DNN) become a hot topic in pattern recognition and machine learning science. Deep Learning (DL) scheme is based on the consecutive layers of signal processing units in order to mix and re-orient the input data to their most representative order correspond to a specific application.

Deep Neural Networks training also known as Deep Learning is one of the most advanced machine learning techniques trending in recent years due to appearance of extremely

powerful parallel processing hardware and Graphical Processing Units (GPU). Several consecutive signal processing units are set in serial/parallel architecture mixing and re-orienting the input data in order to result in most representative output considering a specific problem. The most popular image/sound processing structure of DNN is constructed by three main processing layers: Convolutional Layer, Pooling Layer and Fully Connected Layer. DNN units are described below:

Convolutional Layer: This layer convolves the (in general 3D) image “I” with (in general 4D) kernel “W” and adds a (in general 3D) bias term “b” to it. The output is given by:

$$P = I * W + b, \quad (1)$$

where * operator is nD convolution in general. In the training process the kernel and bias parameters are selected in a way to optimize the error function of the network output.

Pooling Layer: The pooling layers applies a non-linear transform on the input image which reduce the neuron numbers after the operation. It’s common to put a pooling layer between two consecutive convolutional layers. This operation also reduces the unit size which will lead to less computational load and also prevents the over-fitting problem.

Fully Connected Layer: Fully connected layers are exactly same as the classical Neural Network (NN) layers where all the neurons in a layer are connected to all the neurons in their subsequent layer. The neurons are triggered by the summation of their input multiplied by their weights passed from their activation functions.

The Network Architecture is shown in the table below

Layer	Kernel/Units	Size/Dropout Probability
Convolutional	8	3x3
Maxpool	N/A	2x2
Convolutional	8	3x3
Maxpool	N/A	2x2
Convolutional	8	3x3
Maxpool	N/A	2x2
Fully Connected	7	Dropout p =0.5

Table 4.1: Network Configuration for CK+ database.

CHAPTER 5

RESULTS

Example 5.1

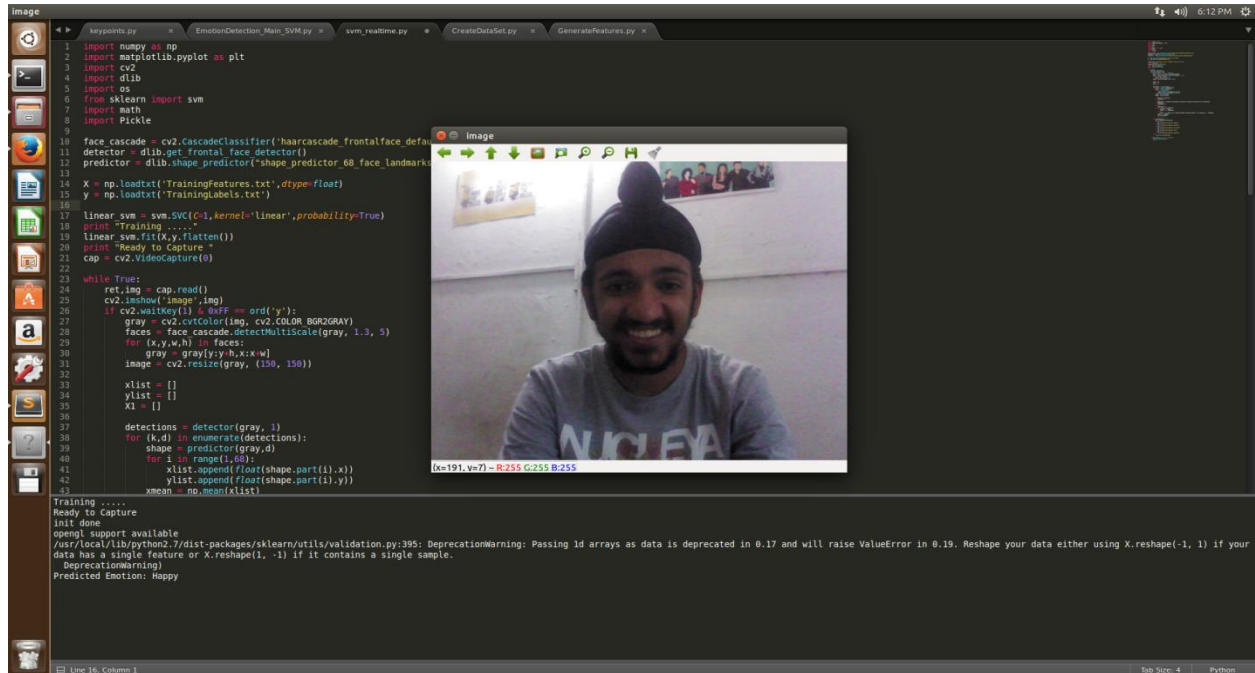


Fig 5.1 Classified as Happy

Example 5.2

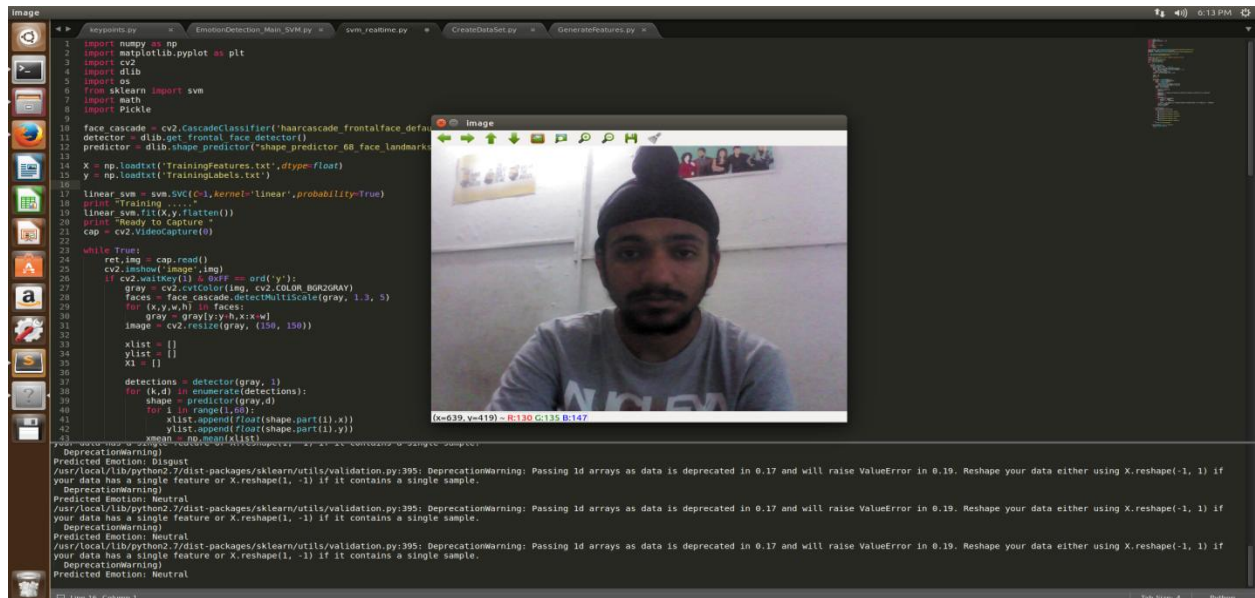


Fig 5.2 Classified as Neutral

Example 5.3

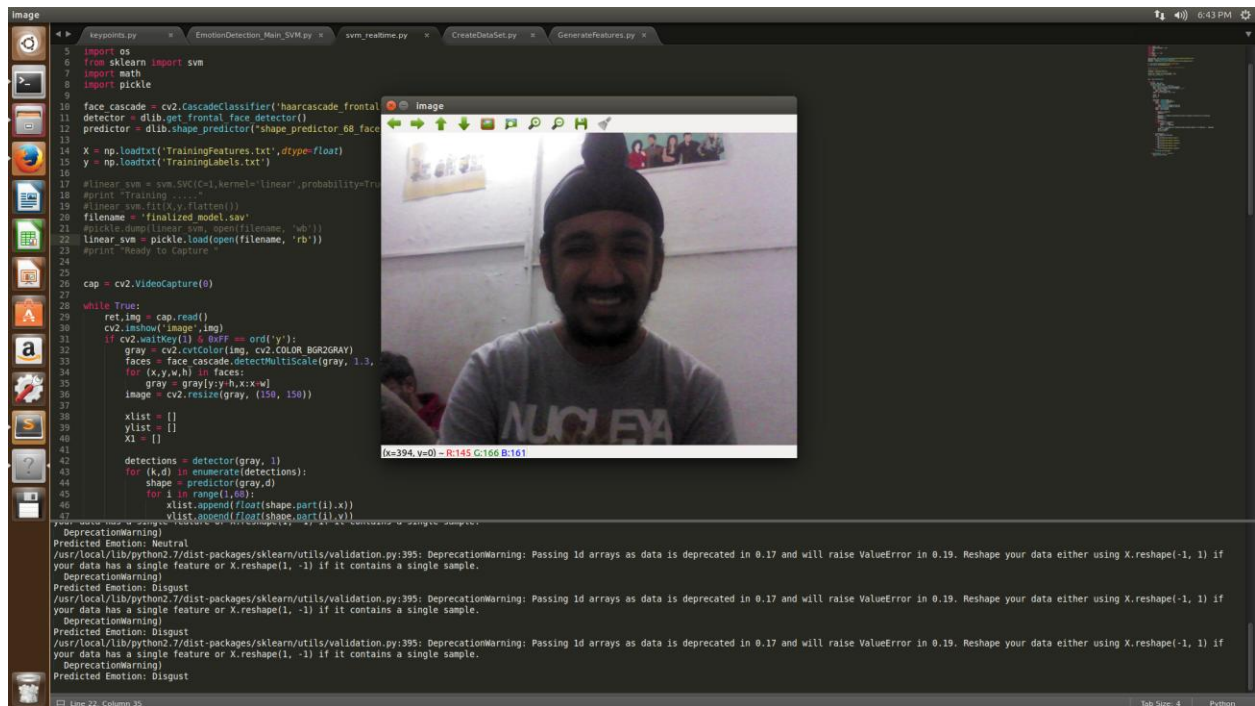


Fig 5.3 Classified as Disgust

Example 5.4

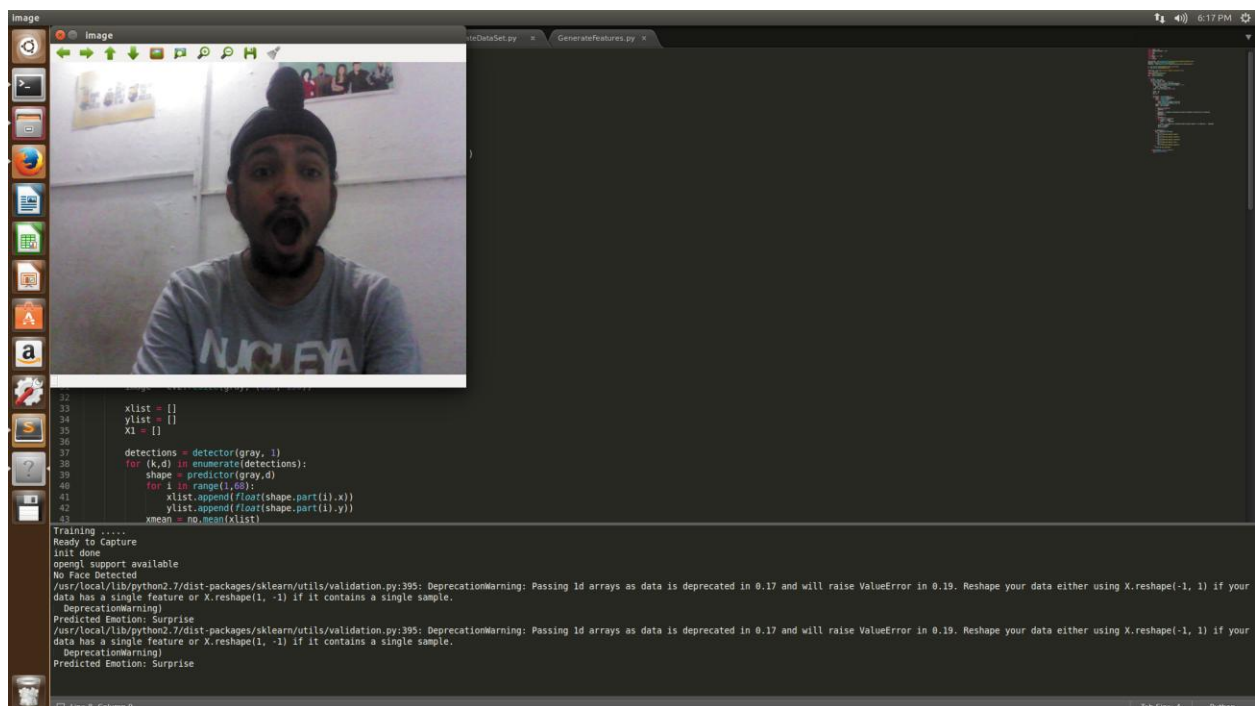


Fig 5.4 Classified as Surprise

CONCLUSION

In this project, a software implementation of the algorithm was implemented using Python. The face detection algorithm was implemented using the Haar-Cascades in OpenCV. Facial Landmarks were extracted using a trained Convolutional Neural Network that extracts 68 Keypoints on a face and tracks them. The training dataset was normalized to cropped 150 X 150 images that contained just the faces and profiles of the people. Finally, features were extracted using the dataset and a classifier was used to train on our training data and make predictions. The system proved to work with no lagging and under varying conditions of facial expressions, skin tones, and lighting.

REFERENCES

- [1] Tong Zhang, Wenming Zheng, Zhen Cui, Yuan Zong, Jingwei Yan and Keyu Yan, A Deep Neural Network Driven Feature Learning Method for Multi-view Facial Expression Recognition. IEEE Trans. on Multimedia, 2016.
- [2] Shabab Bazrafkan, Tudor Nedelcu, Pawel Filipczuk, Peter Corcoran, Deep Learning for Facial Expression Recognition. 2017 IEEE International Conference on Consumer Electronics (ICCE), 2017.
- [3] Vahid Kazemi and Josephine Sullivan, One Millisecond Face Alignment with an Ensemble of Regression Trees. IEEE International Conference on Computer Vision and Pattern Recognition (CVPR), 2014.
- [4] Zhanpeng Zhang, Ping Luo, Chen Change Loy, and Xiaoou Tang, Facial Landmark Detection by Deep Multi-task Learning. IEEE ECCV, 2016.
- [5] Yi Sun, Xiaogang Wang and Xiaoou Tang, Deep Convolutional Network Cascade for Facial Point Detection. IEEE International Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

APPENDIX

Python Code

1. CreateDataset.py (Script to create a normalized dataset)

```
import cv2
import glob
import numpy as np
import matplotlib.pyplot as plt

face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

emotions = ["neutral", "anger", "contempt", "disgust", "fear", "happy", "sadness", "surprise"]

def ModifyImages(emotion):
    files = glob.glob("sorted_set/%s/*"%emotion)
    filenumber=0

    for f in files:
        img = cv2.imread(f)
        gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray,1.3,5)
        for (x,y,w,h) in faces:
            gray = gray[y:y+h,x:x+w]

            try:
                new_image = cv2.resize(gray,(300,350))
                cv2.imwrite("dataset/%s/%s.jpg"%(emotion,filenumber),new_image)
            except:
                pass

        filenumber+=1

    for emotion in emotions:
        ModifyImages(emotion)
```

2. GenerateFeatures.py (Script to extract features from Dataset)

```
import numpy as np
import matplotlib.pyplot as plt
```

```

import cv2
import dlib
import glob
import os
import math

X1 = open('TrainingFeatures.txt','w')
X2 = open('ValidationFeatures.txt','w')
Y1 = open('TrainingLabels.txt','w')
Y2 = open('ValidationLabels.txt','w')

detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

def BuildTrainingData(files,label):
    for myfile in files:
        img = cv2.imread(myfile)
        gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

        # clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
        # gray = clahe.apply(gray)

        xlist = []
        ylist = []

        detections = detector(gray,1)

        for k,d in enumerate(detections):
            shape = predictor(gray,d)
            for i in range(1,68):
                xlist.append( float(shape.part(i).x) )
                ylist.append( float(shape.part(i).y) )

            xmean = np.mean(xlist)
            ymean = np.mean(ylist)

            if xlist[26] == xlist[29]:
                anglenose = 0
            else:
                anglenose = int(math.atan((ylist[26]-ylist[29])/(xlist[26]-
xlist[29]))*(180/math.pi))

```

```

        if anglenose<0:
            anglenose += 90
        else:
            anglenose -= 90

    for i in range(0,len(xlist)):
        d = math.sqrt( (xlist[i]-xmean)*(xlist[i]-xmean) + (ylist[i]-ymean)*(ylist[i]-ymean) )

        if xlist[i] == xmean:
            angle = 90 - anglenose
        else:
            angle = int( math.atan( (ylist[i]-ymean)/(xlist[i]-xmean) )*(180/math.pi) ) - anglenose

        # d=str(d)
        angle=str(angle)
        # X1.write(d + " " + angle + " ")
        X1.write(angle + " ")

    X1.write("\n")
    Y1.write(label + "\n")

```

```

def BuildValidationData(files,label):
    for myfile in files:
        img = cv2.imread(myfile)
        gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

        # clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
        # gray = clahe.apply(gray)

        xlist = []
        ylist = []

        detections = detector(gray,1)

        for k,d in enumerate(detections):
            shape = predictor(gray,d)
            for i in range(1,68):
                xlist.append( float(shape.part(i).x) )
                ylist.append( float(shape.part(i).y) )

```

```

xmean = np.mean(xlist)
ymean = np.mean(ylist)

if xlist[26] == xlist[29]:
    anglenose = 0
else:
    anglenose = int(math.atan((ylist[26]-ylist[29])/(xlist[26]-
xlist[29]))*(180/math.pi))

if anglenose<0:
    anglenose += 90
else:
    anglenose -= 90

for i in range(0,len(xlist)):
    d = math.sqrt( (xlist[i]-xmean)*(xlist[i]-xmean) + (ylist[i]-ymean)*(ylist[i]-
ymean) )

    if xlist[i] == xmean:
        angle = 90 - anglenose
    else:
        angle = int( math.atan( (ylist[i]-ymean)/(xlist[i]-xmean)
)*(180/math.pi) ) - anglenose

    # d=str(d)
    angle=str(angle)
    # X2.write(d + " " + angle + " ")
    X2.write(angle + " ")

X2.write("\n")
Y2.write(label + "\n")

def BuildData(emotion,label):
    files = glob.glob('dataset/%s/*'%emotion)
    np.random.shuffle(files)
    training_data=[]
    cross_validation_data=[]
    tdl = int( 0.8*len(files) )
    for i in range(0,tdl):
        training_data.append(files[i])
    for i in range(tdl+1,len(files)):
        cross_validation_data.append(files[i])

```

```

BuildTrainingData(training_data,str(label))
BuildValidationData(cross_validation_data,str(label))

emotions = ['disgust','surprise','neutral','happy','anger']
np.random.shuffle(emotions)

labels=dict()
labels['disgust']=1
labels['surprise']=2
labels['neutral']=3
labels['happy']=4
labels['anger']=5

for emotion in emotions:
    BuildData(emotion,labels[emotion])

X1.close()
X2.close()
Y1.close()
Y2.close()

```

3. FacialLandmarks.py (Script to detect Facial Landmarks)

```

import cv2
import dlib

video_capture = cv2.VideoCapture(0)
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
x = []
frame = cv2.imread('/home/manjot/Desktop/Projects/0.jpg')
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
clahe_image = clahe.apply(gray)
cv2.imshow("Original Image", frame)

detections = detector(clahe_image, 1)

```

```

for k,d in enumerate(detections):
    shape = predictor(clahe_image, d)
    for i in range(1,68):
        cv2.circle(frame, (shape.part(i).x, shape.part(i).y), 1, (0,255,0), thickness=2)

cv2.imshow("image", frame)
cv2.waitKey(0)
cv2.destroyAllWindows()
if cv2.waitKey(1) & 0xFF == ord('w'):
    for k,d in enumerate(detections):
        shape = predictor(clahe_image, d)
        for i in range(1,68):
            x.append(float(shape.part(i).x))

if cv2.waitKey(1) & 0xFF == ord('q'):
    print(x)
    break

```

4. svm_realtime.py (Main Script that launches webcam and predicts emotions)

```

import numpy as np
import matplotlib.pyplot as plt
import cv2
import dlib
import os
from sklearn import svm
import math
import pickle

face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

X = np.loadtxt('TrainingFeatures.txt',dtype=float)
y = np.loadtxt('TrainingLabels.txt')

linear_svm = svm.SVC(C=1,kernel='linear',probability=True)
print "Training ....."

```

```

linear_svm.fit(X,y.flatten())
filename = 'finalized_model.sav'
pickle.dump(linear_svm, open(filename, 'wb'))
linear_svm = pickle.load(open(filename, 'rb'))
print "Ready to Capture "

```

```

cap = cv2.VideoCapture(0)

```

```

while True:

```

```

    ret,img = cap.read()
    cv2.imshow('image',img)
    if cv2.waitKey(1) & 0xFF == ord('y'):
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray, 1.3, 5)
        for (x,y,w,h) in faces:
            gray = gray[y:y+h,x:x+w]
            image = cv2.resize(gray, (150, 150))

```

```

        xlist = []
        ylist = []
        X1 = []

```

```

        detections = detector(gray, 1)
        for (k,d) in enumerate(detections):
            shape = predictor(gray,d)
            for i in range(1,68):
                xlist.append(float(shape.part(i).x))
                ylist.append(float(shape.part(i).y))
            xmean = np.mean(xlist)
            ymean = np.mean(ylist)

            if xlist[26]==xlist[29]:
                anglenose = 0
            else:
                anglenose = int(math.atan((ylist[26]-ylist[29])/(xlist[26]-
xlist[29]))*(180/math.pi))
            if anglenose<0:
                anglenose+=90
            else:
                anglenose-=90
            for i in range (0,len(xlist)):

```

```

        if xlist[i] == xmean:
            angle = 90 - anglenose
        else:
            angle = int( math.atan( (ylist[i]-ymean)/(xlist[i]-xmean)
)*(180/math.pi) ) - anglenose
        angle = str(angle)
        X1.append(angle)
        #print(angle)

    if len(detections) != 0 :
        val = linear_svm.predict(X1)
        if val == 4 :
            print("Predicted Emotion: Happy")
        if val == 3 :
            print("Predicted Emotion: Neutral")
        if val == 2 :
            print("Predicted Emotion: Surprise")
        if val == 5 :
            print("Predicted Emotion: Sad")
        if val == 1 :
            print("Predicted Emotion: Disgust")
    else:
        print("No Face Detected")

if cv2.waitKey(1) & 0xFF == ord('q'):
    cv2.destroyAllWindows()
    break

```