Facial Expression Recognition (FER)

Ekaterina Chumbarova , Srinivas Jakka , Rui Li

Introduction

Facial Emotion Recognition (FER) is an important field of computer vision. Facial expressions are a form of nonverbal communication, as they reveal a person's inner feelings and emotions. Applications of FER can be found in a variety of actions such as biometrics, social media, video games testing, consumer research and tracking of audience response. The total addressable market is likely to grow to 56B in 2024.

Dataset

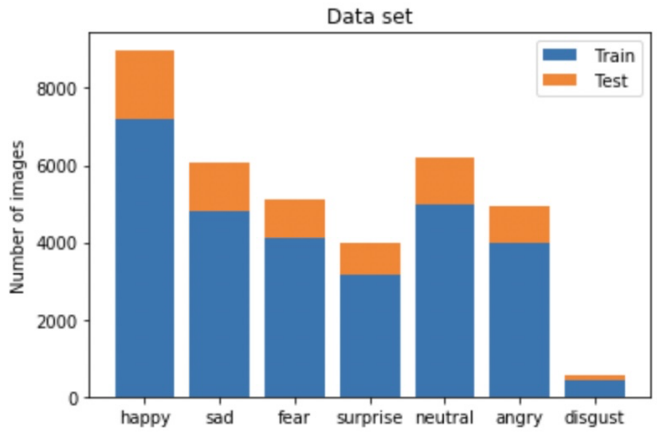
Description

The dataset we use is fer2013 which was published in the International Conference on Machine Learning in 2013. The dataset can be downloaded from Kaggle “FER-2013 Learn facial expressions from an image”.

The data consists of 48x48 pixel grayscale images of faces. The faces have been automatically registered so that the face is more or less centered and occupies about the same amount of space in each image. The training set consists of 28,709 examples and the public test set consists of 7,175 examples.

The task is to categorize each facial image into one of seven categories (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral).

The number of images in each category is shown below.



	happy	sad	fear	surprise	neutral	angry	disgust
train	7215	4830	4097	3171	4965	3995	436
test	1774	1247	1024	831	1233	958	111
total	8989	6077	5121	4002	6198	4953	547

Challenges

Building a model for detecting a facial expression from a given dataset is a difficult task with a multitude of challenges. Several contributing factors are:

- Presence of non-facial images in the dataset. Upon examining the dataset we found out that there is a need for a considerable amount of clean up before we can proceed to feature extraction.
- Variety of poses in the dataset. Any feature vector we choose would vary greatly depending on the position of face, angle etc. Some of the images include people looking down or sideways.
- Different backgrounds, different color or presence of clutter objects in the background
- Variability of facial textures and general illumination, for example some images are dim and some brightly lit. A feature vector we create would need to provide invariance to a broad range of illumination.
- Incorrect classification of images. Some images were tagged as angry when the pose suggested otherwise.
- Finally, facial occlusions due to hand gestures and other partial occlusions present a great challenge because only part of the face is visible for processing.

Data Preprocessing

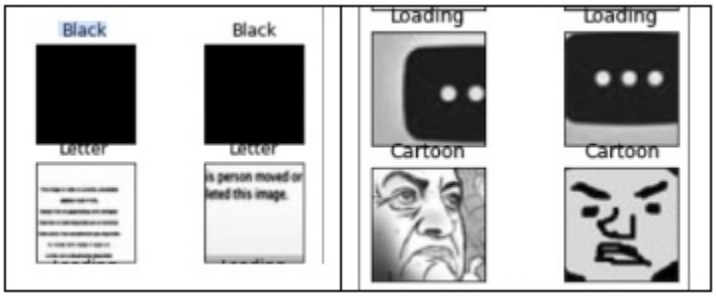
Non Facial images

We see that there is a disproportionate number of faces in each category. As a first pass, we would like to confirm that all images are that of real people. Since we do not have training labels for non facial images, we will attempt to create labels manually for non facial images by manually checking every image in a select category. We chose the “angry” dataset since this category contains about the mean number of images across all categories.

Upon investigating the images in the angry dataset, we found that there are 4 different types of non facial images - cartoon images, that appear to be “cartoon” like, fully black images without any faces, letter images that appear to be images of writings, and “loading” images that appear to be images synonymous with icons for loading data/pages.

We identified 42 cartoon images, 9 fully black images, 5 letter images and 2 loading images in the angry dataset. We used this for training our ML models.

Examples of non facial images

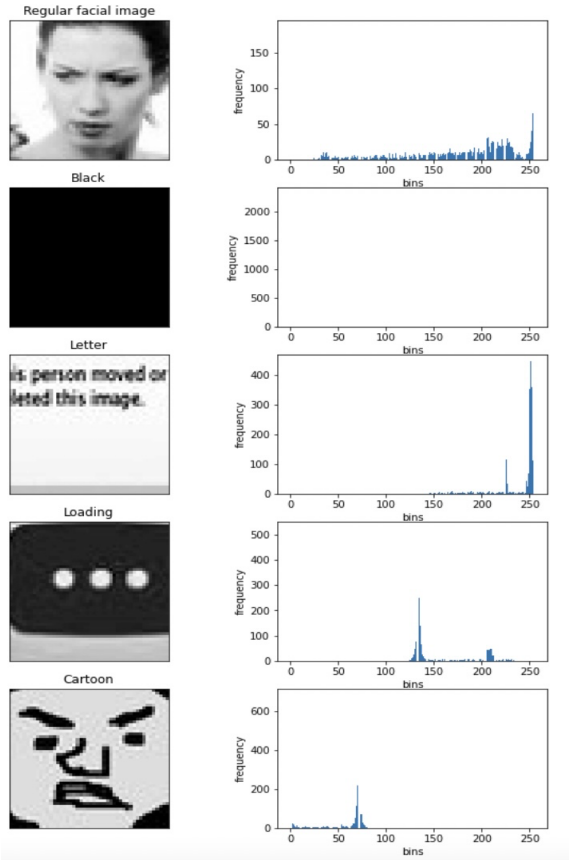


We then checked the histogram distribution of facial and non facial images to identify patterns. The following function was used to generate histograms of images

```
def convert_img_histograms():
    ang_hist = np.empty((angry_set.shape[0], len(np.arange(256))))
    X_hist = np.empty((X.shape[0], len(np.arange(256))))

    ## Convert images to histogram
    for i in range(angry_set.shape[0]):
        ang_hist[i,:] = np.histogram(angry_set[i,:], bins = np.arange(257))[0]

    for i in range(X.shape[0]):
        X_hist[i,:] = np.histogram(X[i,:], bins = np.arange(257))[0]

    return ang_hist, X_hist

ang_hist, X_hist = convert_img_histograms()
```

A sample facial and non facial image revealed the following histograms:

Sample images:



Based on the obtained histograms we made the following conclusions:

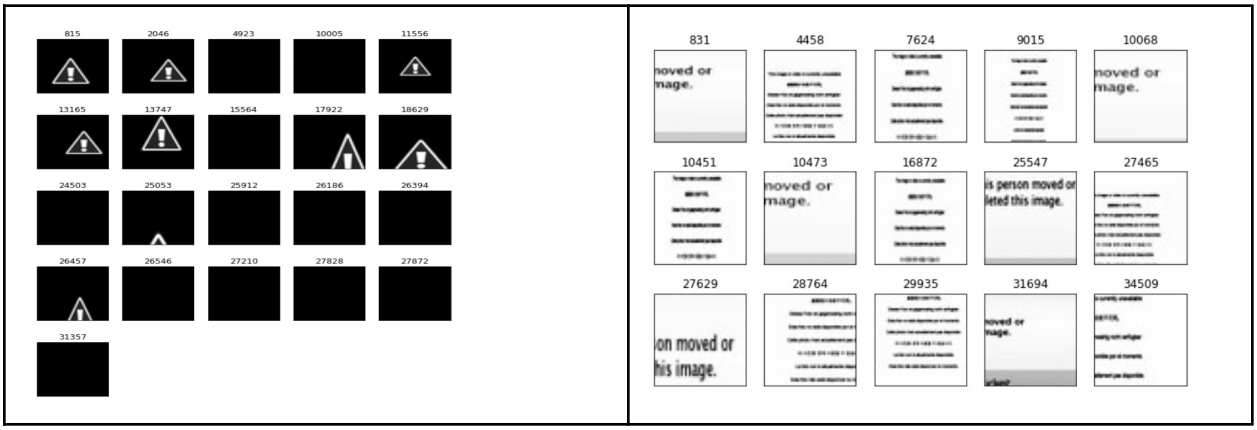
- Regular facial images have widely distributed histograms
- Black images have mostly black(0) and very little other tones
- Letter images have mostly white(255) and very little other tones
- Loading images have mostly bimodal distribution (some gray color and some black areas)
- Cartoon images have a wide distribution but not as wide as facial images

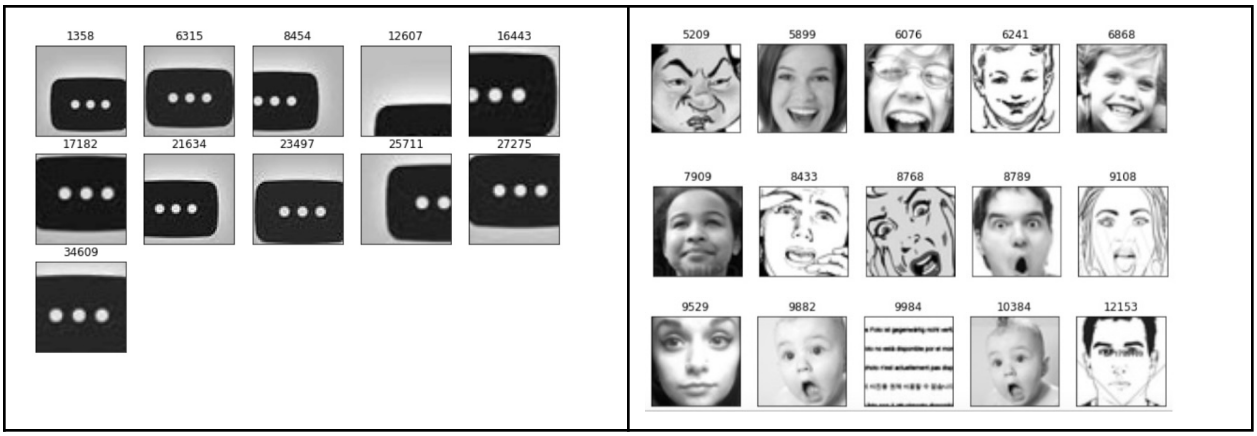
ML to predict Non Facial Images

We then used **KNN** to predict non facial images in other categories. The following function was used to predict labels across the dataset using training labels from the angry dataset.

```
hist_filt = KNeighborsClassifier(n_neighbors=1)
hist_filt.fit(ang_hist, angry_label)
prediction = hist_filt.predict(X_hist)

letter_KNN_hist_idx = []
black_KNN_hist_idx = []
loading_KNN_hist_idx = []
cartoon_KNN_hist_idx = []

i=0
for label in prediction:
    if label == 'letters':
        letter_KNN_hist_idx.append(i)
    elif label == 'black':
        black_KNN_hist_idx.append(i)
    elif label == 'loading':
        loading_KNN_hist_idx.append(i)
    elif label == 'cartoon':
        cartoon_KNN_hist_idx.append(i)
    i+=1

print("black",len(black_KNN_hist_idx))
print("letters",len(letter_KNN_hist_idx))
print("loading",len(loading_KNN_hist_idx))
print("cartoon",len(cartoon_KNN_hist_idx))
```

KNN identified 21 black, 15 letters, 11 loading and 138 cartoon images. Below are some of the non facial images that were predicted by KNN.





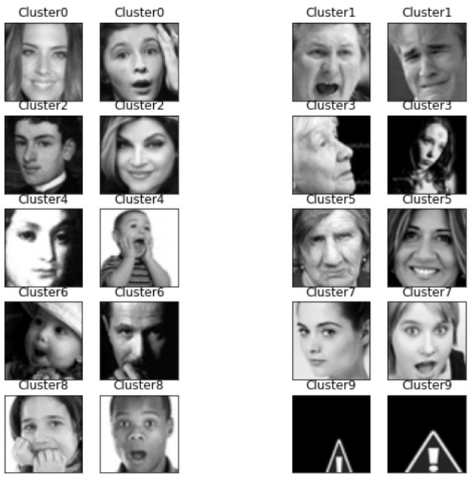
While KNN has correctly identified black, loading and label images, some images under the cartoon category were actually facial images. As such we did not mark the predicted cartoon images for deletion and looked for other ways to identify such images.

K Means Clustering

We then used **K Means** to identify the cluster for non facial images. The following function was used to identify the cluster for non facial images using training labels from the angry dataset.

```
km = KMeans(n_clusters=10,
            #init='random',
            init='k-means++',
            n_init=12,
            max_iter=300,
            tol=1e-04,
            random_state=0)

# predict k-means classes for histogram data
y_km_hist = km.fit_predict(X_hist)

# print clusters and the number of data in a cluster
unique, counts = np.unique(y_km_hist, return_counts=True)
dict(zip(unique, counts))
```

Cluster 9 contained all non facial images which were tagged for removal.



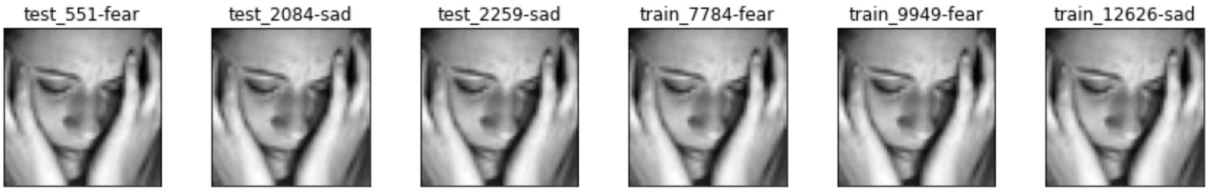
Duplicate and Cross Labeled images

A quick scan of images revealed that we have duplicate images across training and test datasets and cross labeled images as well. We used **DBSCAN** to identify such images. Below is the code snippet for the DBSCAN implementation.

```
# Tried to cluster with DB scan
db = DBSCAN(eps=3.0, min_samples=2, metric='euclidean')
y_db = db.fit_predict(X)

unique, counts = np.unique(y_db, return_counts=True)
dict(zip(unique, counts))
# Found a lot of duplicated images


# Histogram of duplicated images
# Mostly 2 images duplicated (probably one for the 'train' and one for the 'test')
# But there are 3 or more duplicated images
np.histogram(counts[1:])
```

Below are examples of duplicate images that were identified by DBSCAN:

Below are examples of cross labeled images.



We removed all cross labeled images and only kept a single copy from duplicate images.

In total, we removed 1962 images out of 35887, which is 5.47% of the total number of images in our dataset.

Detecting front face images

We found several images that would be difficult to use, for example some were side posed or tilted. We therefore decided to remove all images that were not front facing.

Haar Cascade is an Object Detection Algorithm used to identify faces in an image or a real time video. The algorithm uses edge or line detection features in images. We downloaded several pre-trained haar cascade files for frontal face detection and used them to filter out non frontal images.

The following Haar cascades were used to identify frontal face images:
Haarcascade_frontalface_default.xml
haarcascade_frontalface_alt.xml
haarcascade_frontalface_alt2.xml
haarcascade_frontalface_alt_tree.xml

Below is the function to detect frontal faces using the Haar classifiers.

```
def detect_front_face(img):
    dim = (img.shape[0], img.shape[1])
    resize = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
    resize = np.array(resize, dtype='uint8')

    # Detect the faces
    faces = face_classifier.detectMultiScale(resize, 1.03, 6) ##1.0485258, 6)
    # print(faces)
    if len(faces) == 1:
        return True
    return False
```

Using the 4 Haar classifiers that we downloaded, we identified 28479 images out of a total of 33925, which is 83.95% of the data, for removal.

Below are sample images that were considered for removal.

HOG + Linear SVM Classifier:

The get_frontal_face_detector function does not accept any parameters. A call to it returns the pre-trained HOG + Linear SVM face detector included in the dlib library.

Dlib's HOG + Linear SVM face detector is fast and efficient. By nature of how the Histogram of Oriented Gradients (HOG) descriptor works, it is not invariant to changes.

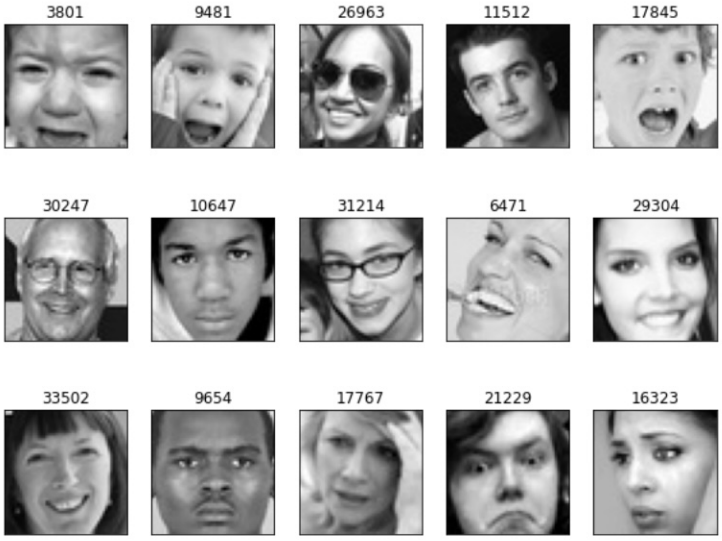
Below is the function to detect frontal faces using the HOG classifier.

```
def detect_front_facedlib(img):
    dim = (img.shape[0], img.shape[1])
    resize = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
    resize = np.array(resize, dtype='uint8')
    faces = detector(resize, 1)
    if len(faces) == 1:
        return True
    return False
```

Using this classifier, we may be removing 10542 images out of a total of 33925, which is 31.07% of the data.

Below are sample images considered for removal by the HOG classifier:

Below are sample images considered for removal by the Haar classifier but not by HOG - most of these are non frontal faces too.



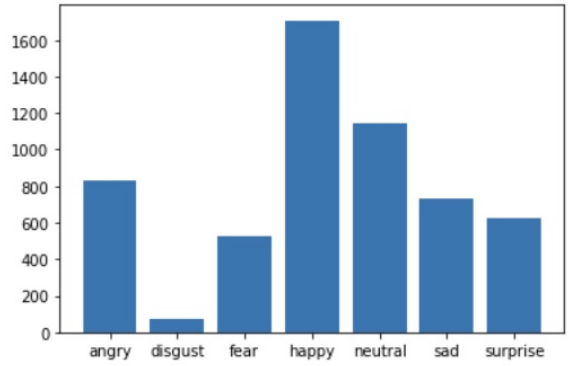
Below are sample images considered for removal by the HOG classifier but not by Haar - most of these are non frontal faces too.

We therefore decided to remove all images that were identified as non-facial by any of the HOG or Haar classifiers.
We therefore removed 28580 images out of a total of 33925, which is 84.24% of the data.

We decided to save all filtered images to gDrive so feature extraction and image classification could be run independently without the need to reprocess the images.

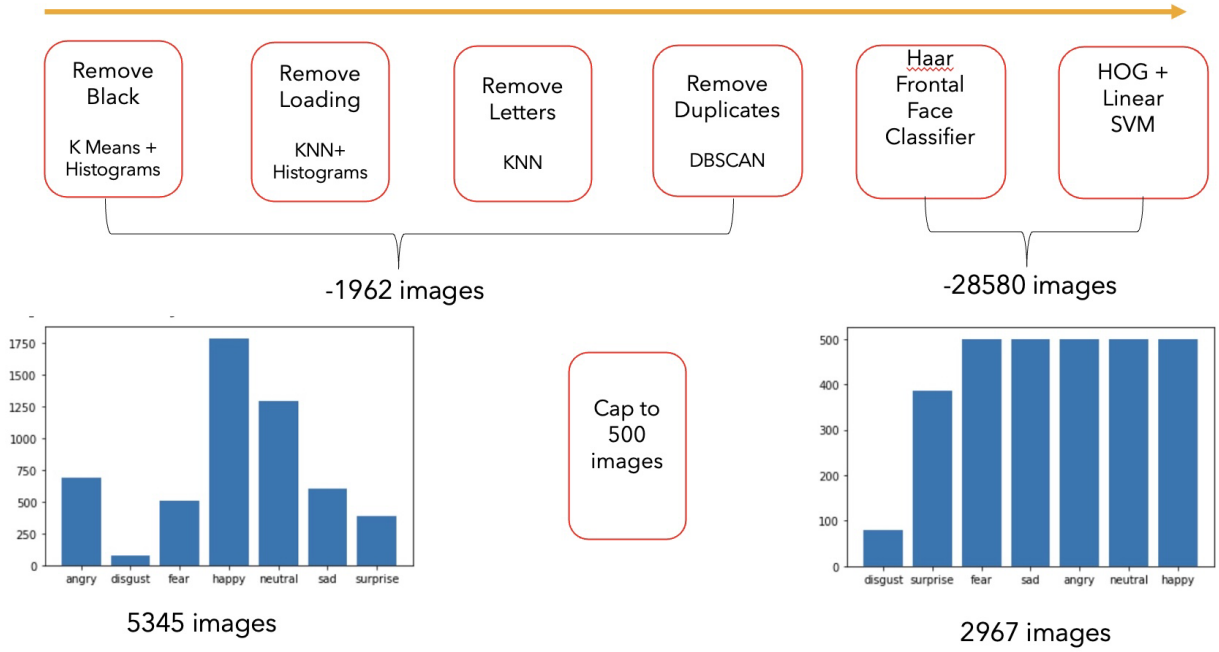
We now have 5345 facial images to process and classify under one of the 7 categories. Below is the distribution of all of these images.

```
angry : 828 images
disgust : 69 images
fear : 529 images
happy : 1707 images
neutral : 1147 images
sad : 736 images
surprise : 622 images
```



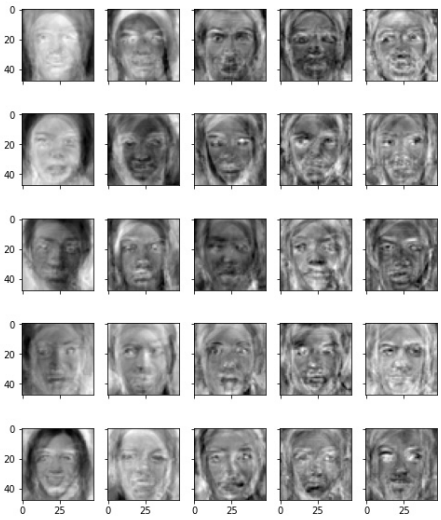
Class imbalance: There are only 69 images in the disgust category whereas the happy category has over 1700 images. Other categories have about 1000 or fewer images. We therefore decided to shuffle and cap all images to 500 to reduce the class imbalance in our dataset.

In Summary, we removed over 85% of the images in the dataset. Of the 5,345 images that remained after the cleanup, each category was capped to 500 images to reduce the class imbalance amongst the categories.



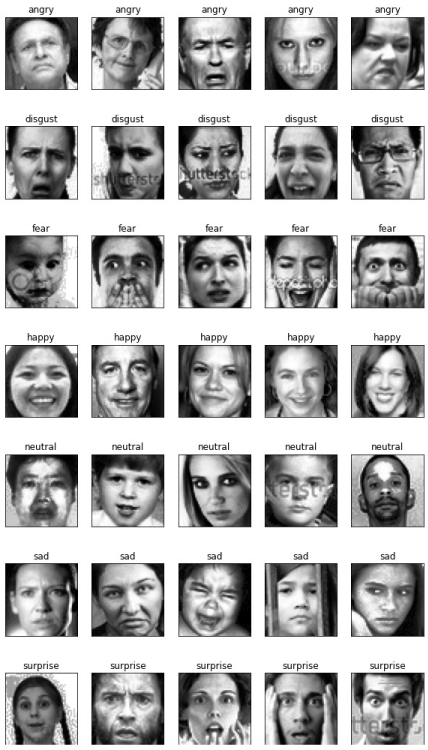
Illumination normalization

Below are Eigen faces for 25 sample images from the dataset:



We see that a lot of encoded parts of images are just areas of brightness and darkness which tells us that we need to normalize our images before attempting classification.

We used a histogram normalization and after applying it, the sample set looks like this:



We generated normalized Train and Test images:

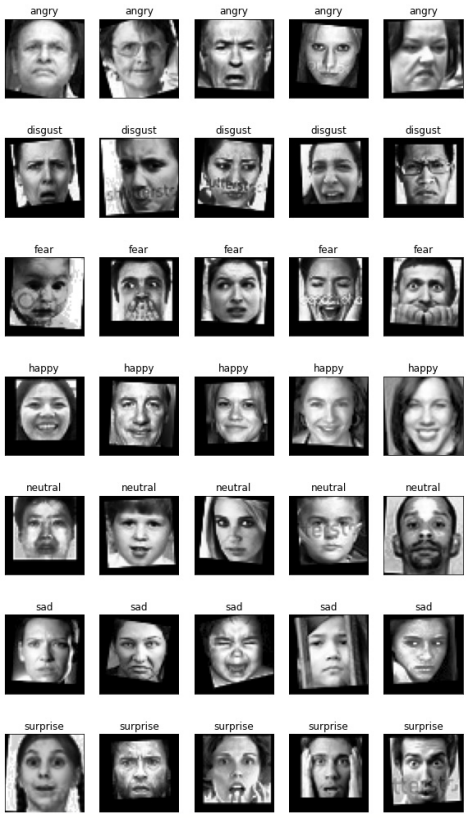
```
# Normalized images
X_train_norm = np.empty(shape=(len(X_train), 48*48))
for i in range(len(X_train)):
    img = X_train[i].reshape(48, 48).astype(np.uint8)
    equ = cv2.equalizeHist(img)
    X_train_norm[i] = equ.reshape(48*48)

X_test_norm = np.empty(shape=(len(X_test), 48*48))
for i in range(len(X_test)):
    img = X_test[i].reshape(48, 48).astype(np.uint8)
    equ = cv2.equalizeHist(img)
    X_test_norm[i] = equ.reshape(48*48)
```

Face Alignment

To improve the robustness of our features, we decided to pre-process our dataset by aligning faces so that the eyes end up in the same pixel location. Obviously, this algorithm does not guarantee a complete alignment because of natural variation of facial shapes and features.

```
from imutils.face_utils import FaceAligner, shape_to_np, FACIAL_LANDMARKS_68_IDXS
from google.colab.patches import cv2_imshow
predictor = dlib.shape_predictor("/content/drive/MyDrive/281/shape_predictor_68_face_landmarks.dat")
detector = dlib.get_frontal_face_detector()

fa = FaceAligner(predictor, desiredFaceWidth=48, desiredFaceHeight=48)
# img = example_images[3].reshape(48,48).astype(np.uint8)
def align_face(img):
  # cv2_imshow(img)
  gray = img
  rects = detector(gray, 2)
  faceAligned = img
  # loop over the face detections
  for rect in rects:
      shape = shape_to_np(predictor(gray, rect))
      (lStart, lEnd) = FACIAL_LANDMARKS_68_IDXS["left_eye"]
      (rStart, rEnd) = FACIAL_LANDMARKS_68_IDXS["right_eye"]
      leftEyePts = shape[lStart:lEnd]
      rightEyePts = shape[rStart:rEnd]

      # compute the center of mass for each eye
      leftEyeCenter = leftEyePts.mean(axis=0).astype("int")
      rightEyeCenter = rightEyePts.mean(axis=0).astype("int")

      # compute the angle between the eye centroids
      dY = rightEyeCenter[1] - leftEyeCenter[1]
      dX = rightEyeCenter[0] - leftEyeCenter[0]
      angle = np.degrees(np.arctan2(dY, dX)) - 180

      # compute the desired right eye x-coordinate based on the
      # desired x-coordinate of the left eye
      desiredRightEyeX = 1.0 - fa.desiredLeftEye[0]

      # determine the scale of the new resulting image by taking
      # the ratio of the distance between eyes in the *current*
      # image to the ratio of distance between eyes in the
      # *desired* image
      dist = np.sqrt((dX ** 2) + (dY ** 2))
      desiredDist = (desiredRightEyeX - fa.desiredLeftEye[0])
      desiredDist *= fa.desiredFaceWidth
      scale = desiredDist / dist

      # compute center (x, y)-coordinates (i.e., the median point)
      # between the two eyes in the input image
      eyesCenter = (int((leftEyeCenter[0] + rightEyeCenter[0]) // 2),
          int((leftEyeCenter[1] + rightEyeCenter[1]) // 2))
      M = cv2.getRotationMatrix2D(eyesCenter, angle, scale)

      # update the translation component of the matrix
      tX = fa.desiredFaceWidth * 0.5
      tY = fa.desiredFaceHeight * fa.desiredLeftEye[1]
      M[0, 2] += (tX - eyesCenter[0])
      M[1, 2] += (tY - eyesCenter[1])

      # apply the affine transformation
      (w, h) = (fa.desiredFaceWidth, fa.desiredFaceHeight)
      faceAligned = cv2.warpAffine(img, M, (w, h),
        flags=cv2.INTER_CUBIC)
  return cv2.equalizeHist(faceAligned)
```

Below is the sample dataset with facial images aligned using the algorithm we developed:



Transform train and test images

```
X_train_aligned = np.empty(shape=(len(X_train), 48*48))
for i in range(len(X_train)):
    img = X_train[i].reshape(48, 48).astype(np.uint8)
    X_train_aligned[i] = align_face(img).reshape(48*48)


X_test_aligned = np.empty(shape=(len(X_test), 48*48))
for i in range(len(X_test)):
    img = X_test[i].reshape(48, 48).astype(np.uint8)
    X_test_aligned[i] = align_face(img).reshape(48*48)
```

Data Split for Training and Testing

We used stratified split to ensure equal proportion of facial images by category in test, train and validation splits. This way we could ensure that each facial expression is equally presented in the training, validation and testing datasets.
The validation split data was used for hyperparameter tuning of the best performing ML model.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_data, Y_data, test_size=0.20, stratify=Y_data)
X_rem, X_valid, y_rem, y_valid = train_test_split(X_train, y_train, test_size=0.20, stratify=y_train)
```

Feature Engineering

Sample Images for Visualization

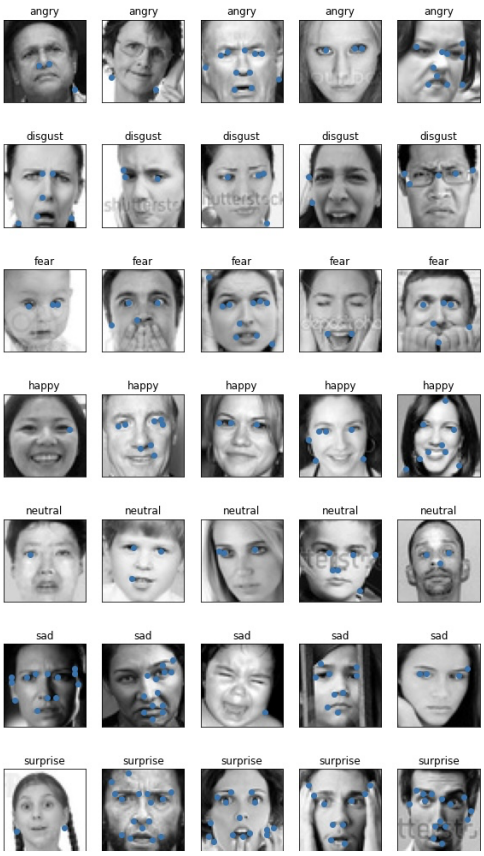
We build a list of 5 test images from each category. These images were then visualized as each feature set is identified and applied to these samples.

```
## Get 5 examples for each category
import random
from more_itertools import locate
angry_ind = random.sample(list(locate(y_train, lambda x: x == 'angry') ),5)
disgust_ind = random.sample(list(locate(y_train, lambda x: x == 'disgust') ),5)
fear_ind = random.sample(list(locate(y_train, lambda x: x == 'fear') ),5)
happy_ind = random.sample(list(locate(y_train, lambda x: x == 'happy') ),5)
neutral_ind = random.sample(list(locate(y_train, lambda x: x == 'neutral') ),5)
sad_ind = random.sample(list(locate(y_train, lambda x: x == 'sad') ),5)
surprise_ind = random.sample(list(locate(y_train, lambda x: x == 'surprise') ),5)

example_index = list(angry_ind+disgust_ind+fear_ind+happy_ind+neutral_ind+sad_ind+surprise_ind)
#print(example_index)
#print(angry_ind)
example_images = X_train[example_index]
example_labels = [y_train[i] for i in example_index]
print(example_images.shape)
```

Corner Peaks

Our first attempt was to use Harris algorithm for corner detection to create a vector of corners and use it for classification of facial expressions. The idea was that the algorithm would detect, for example, corners of mouth and we would be able to distinguish between expressions based on their position, or if we added the angle of corner then we would be able to detect downturned corners of mouth for angry expression, upturned for happy, rounded mouth for surprise etc.

```
def plot_corner_peaks(example_images, example_labels):

    plt.rc('image', cmap='gray')
    plt.figure(figsize=(10, 3*len(example_images)//5))
    i = 0
    for k in range(example_images.shape[0]):
        plt.subplot(len(example_images)//5+1,5,i+1)
        ax = plt.gca()
        ax.set_title(example_labels[k])
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        img = example_images[k].reshape(48,48).astype(np.uint8)
        keypoints = corner_peaks(corner_harris(img), min_distance=2, threshold_rel=0.05)
        ax.imshow(img, cmap="gray")
        ax.scatter(keypoints[:,1], keypoints[:,0])
        #subpix = corner_subpix(img, keypoints, window_size=11)
        #orientations = corner_orientations(img, keypoints, octagon(3, 2))
        i = i+1

plot_corner_peaks(example_images, example_labels)
```



Based on the plotted corners on sample images we realized that the corner detection does not provide a consistent feature vector. Depending on the image, it detects only a subset of corners that are of interest for facial expression detection.
To see if sharpening helps, we create corner peaks for sharpened images:

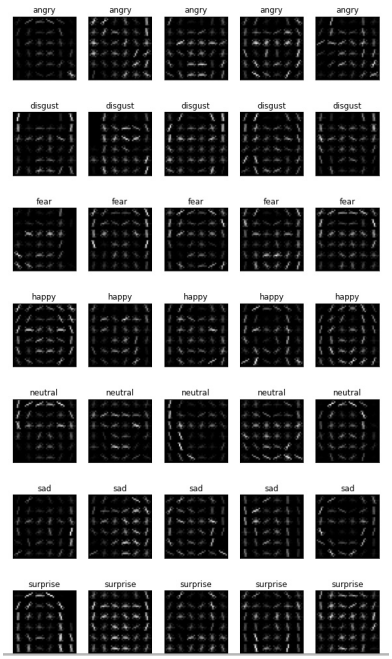
```
def plot_sharp_corner_peaks(example_images, example_labels):

    plt.rc('image', cmap='gray')
    plt.figure(figsize=(10, 3*len(example_images)//5))
    i = 0
    for k in range(example_images.shape[0]):
        plt.subplot(len(example_images)//5+1,5,i+1)
        ax = plt.gca()
        ax.set_title(example_labels[k])
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        img = example_images[k].reshape(48,48).astype(np.uint8)
        blurred_f = ndimage.gaussian_filter(img, 3)
        alpha = 2
        sharpened = img+alpha*(img-blurred_f)

        keypoints = corner_peaks(corner_harris(sharpened), min_distance=2, threshold_rel=0.05)
        ax.imshow(img, cmap="gray")
        ax.scatter(keypoints[:,1], keypoints[:,0])
        #subpix = corner_subpix(img, keypoints, window_size=11)
        #orientations = corner_orientations(img, keypoints, octagon(3, 2))
        i = i+1

plot_sharp_corner_peaks(example_images, example_labels)
```

But even though sharpened images yield more corners, they are still not usable for classification. One problem is that the presence of other objects like glasses or hands skews corner datapoints.

HOG Vectors

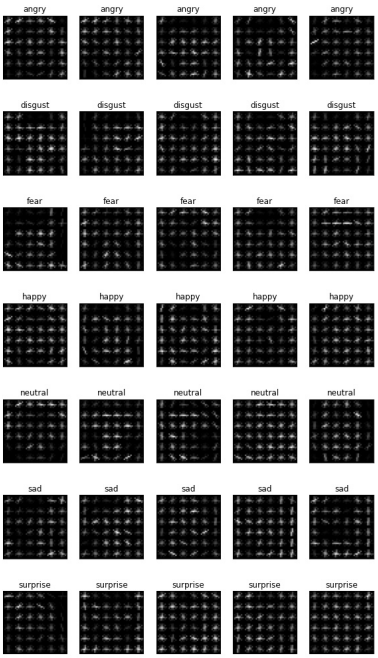
HOG features are used for facial recognition in many research papers, so we thought that it might prove useful as a feature for facial expression as well.

```
def get_hog_image(img):
  hog_vec, hog_vis = hog(img, pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=True)
  return hog_vis

plot_images(example_images, example_labels, get_hog_image)
```



HOG features for sharpened images

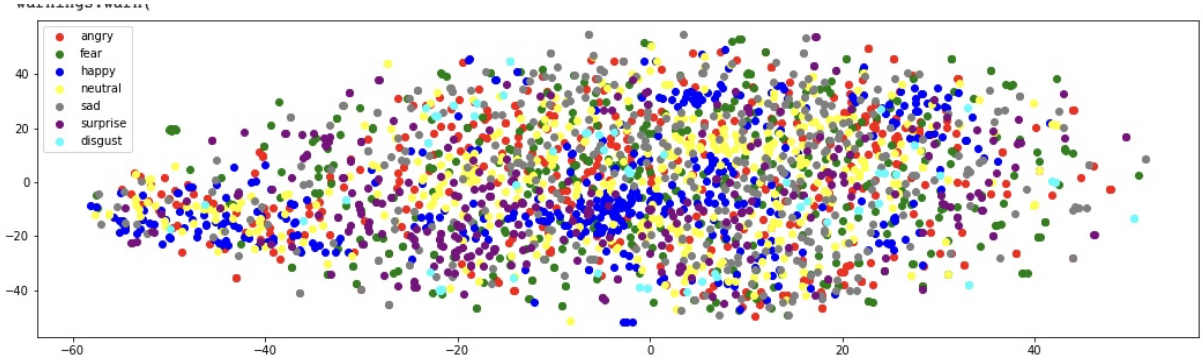
```
def get_hog_sharp_image(img):
    blurred_f = ndimage.gaussian_filter(img, 3)
    alpha = 2
    sharpened = img+alpha*(img-blurred_f)

    hog_vec, hog_vis = hog(sharpened, pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=True)
    return hog_vis

plot_images(example_images, example_labels, get_hog_sharp_image)
```



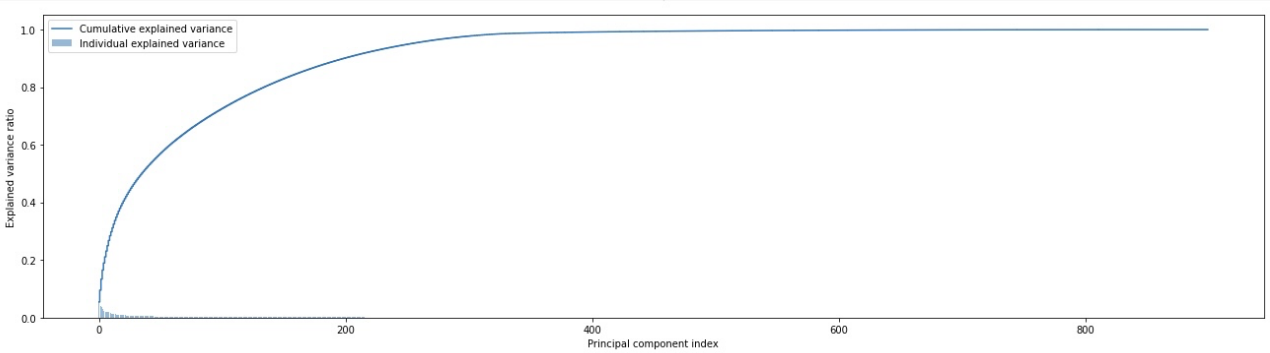
We calculated the HOG vector for each image in our dataset and created a train and test vectors.

```
# Calculate the hog features for training and test data
def identify_hog_features(X_data):
    hog_features = np.empty(shape=(X_data.shape[0], 900))
    for i in range(X_data.shape[0]):
        img = X_data[i,:].reshape(48,48)
        blurred_f = ndimage.gaussian_filter(img, 3)
        alpha = 2
        sharpened = img+alpha*(img-blurred_f)
        hog_vec = hog(img, pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=False)
        hog_features[i] = hog_vec

    return hog_features
```

tSNE visualization of the HOG feature vector does not look great. Ideally we would like to see a better label space separation.



To reduce the dimensionality of our HOG feature vector, we conducted a PCA analysis. The graph below shows the amount of explained variance vs amount of components.

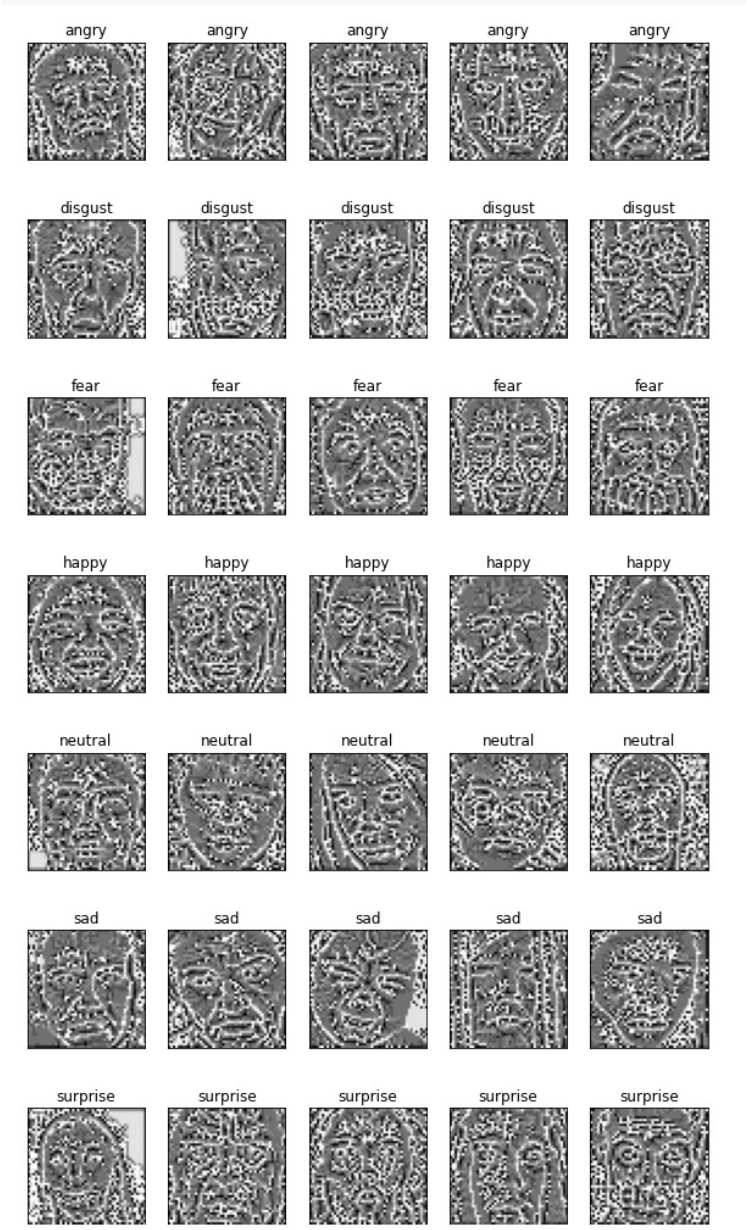


It is obvious from the graph that 300 components is the appropriate number of components for our HOG feature vector.

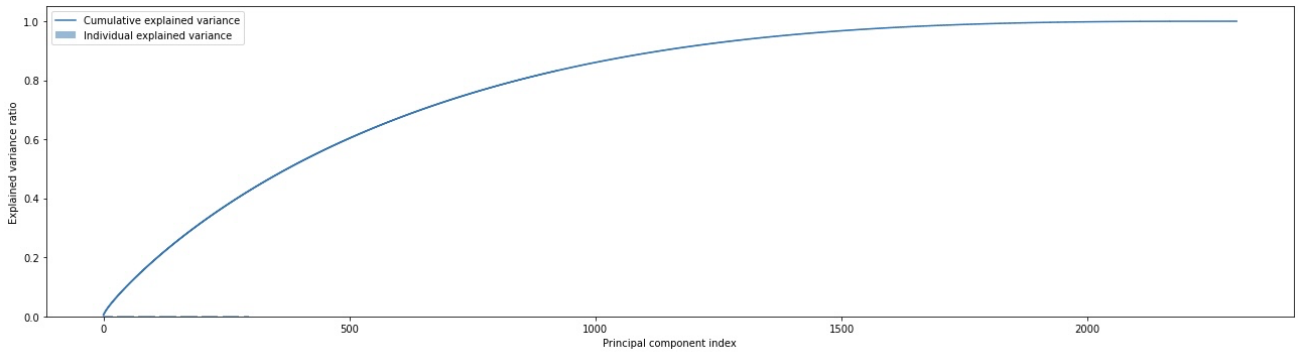
LBP (Local Binary Pattern) features

Several research papers (for example Niu B, Gao Z, Guo B. Facial Expression Recognition with LBP and ORB Features) show promise in using Local Binary Patterns for facial expression recognition systems.
We use local_binary_pattern function from sklearn library to build LBP features for sample images:

```
# Visualize LBP images
def get_lbp_image(img):
    # settings for LBP
    radius = 1
    n_points = 8 * radius
    METHOD = 'uniform'
    lbp = local_binary_pattern(img, n_points, radius, METHOD)
    return lbp

plot_images(example_images, example_labels, get_lbp_image)
```



We conducted PCA analysis for LBP Features:



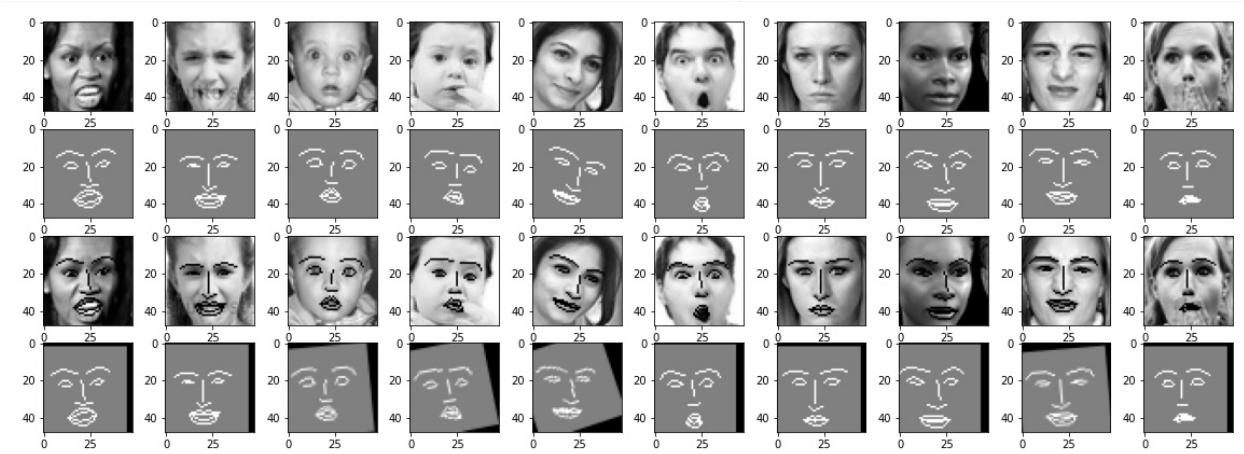
The below function was used to convert our raw images into a LBP feature vector:

```
def identify_lbp_features(X_data):
    features = np.empty(shape=(X_data.shape[0], 48*48))
    for i in range(X_data.shape[0]):
        img = X_data[i,:].reshape(48,48)
        lbp = local_binary_pattern(img, 8, 1, 'uniform')
        features[i] = lbp.reshape(48*48)
    return features
```

Face Landmarks

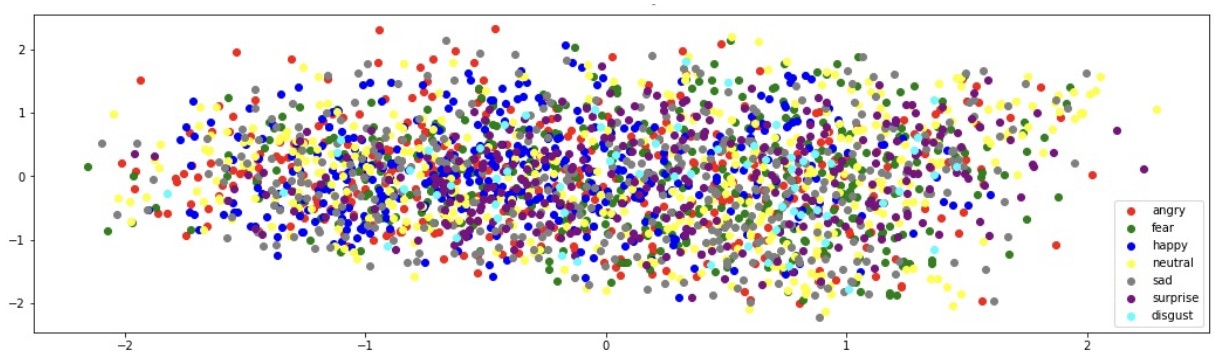
Finally we calculated face landmarks using a “face-recognition” library that is based on HOG and SVM classification. This library outputs a set of locations of 68 facial points. To create the feature vector we mark those locations as 1 in our image matrix, and everything else as 0.

Sample images



Function to generate facial features

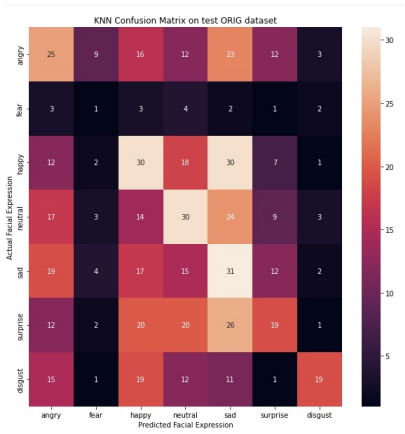
```
# Create a vector of facial features for training and test dataset
def identify_facial_features(X_data):
    num_face_landmarks = 55
    face_landmark_features = np.empty(shape=(X_data.shape[0], 48*48))
    for i in range(X_data.shape[0]):
        img = convert_to_uint(X_data[i,:].reshape(48, 48))
        face_landmarks_list = face_recognition.face_landmarks(img)
        if len(face_landmarks_list) == 0:
            face_landmark_features[i] = np.zeros(shape=(48*48))
        else:
            face_landmarks_list[0].pop('chin')
            M = get_rectify_matrix(face_landmarks_list[0])
            face_points = np.array([[element[0], element[1], 1] for sublist in face_landmarks_list[0].values() for element in sublist])
            rectified_face_points = (M @ face_points.T).T
            feature_vector = np.zeros(shape=(48, 48))
            for point in rectified_face_points:
                x, y = point
                if x < 48 and y < 48:
                    feature_vector[int(x - 1), int(y - 1)] = 1
            face_landmark_features[i] = feature_vector.flatten()

    return face_landmark_features
```



Classification Models and Results

KNN

Using Raw pixel data



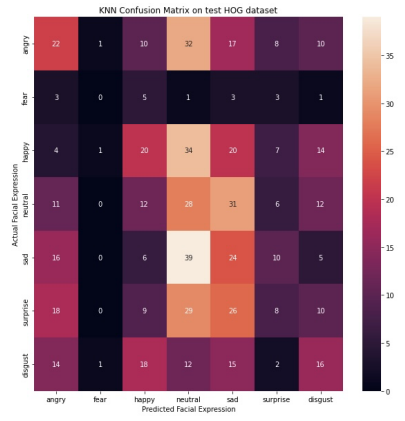
KNN multiclass accuracy score on test ORIG dataset: 0.260943

KNN multiclass recall score on test ORIG dataset: 0.236584

KNN multiclass precision score on test ORIG dataset: 0.277972

KNN multiclass f1 score on test ORIG dataset: 0.241847

Using HOG features



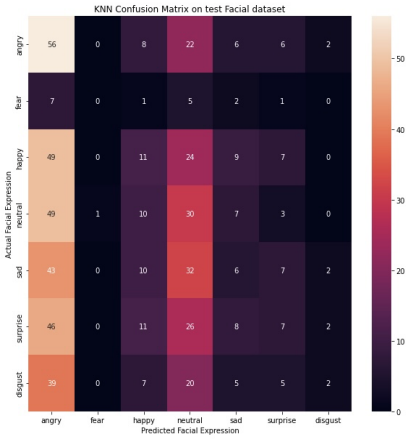
KNN multiclass accuracy score on test HOG dataset: 0.198653

KNN multiclass recall score on test HOG dataset: 0.175018

KNN multiclass precision score on test HOG dataset: 0.179083

KNN multiclass f1 score on test HOG dataset: 0.170511

Using Facial Features



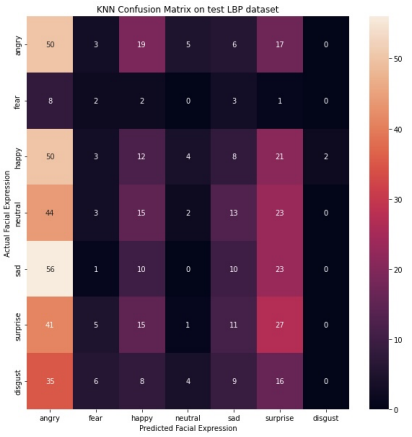
KNN multiclass accuracy score on test Facial dataset: 0.188552

KNN multiclass recall score on test Facial dataset: 0.160806

KNN multiclass precision score on test Facial dataset: 0.165155

KNN multiclass f1 score on test Facial dataset: 0.127455

Using LBP Features



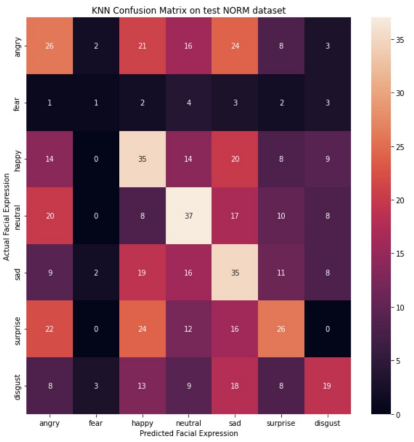
KNN multiclass accuracy score on test LBP dataset: 0.173401

KNN multiclass recall score on test LBP dataset: 0.162143

KNN multiclass precision score on test LBP dataset: 0.130538

KNN multiclass f1 score on test LBP dataset: 0.127415

Using normalized images



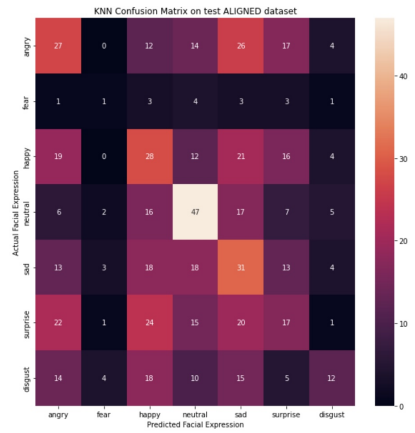
KNN multiclass accuracy score on test NORM dataset: 0.301347

KNN multiclass recall score on test NORM dataset: 0.270870

KNN multiclass precision score on test NORM dataset: 0.287686

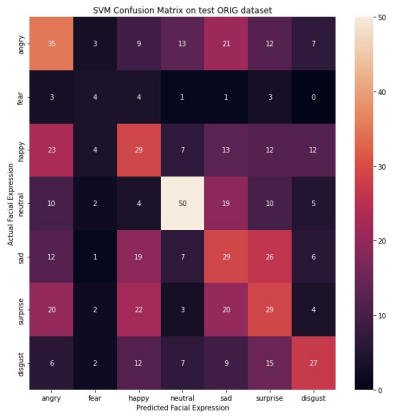
KNN multiclass f1 score on test NORM dataset: 0.273186

Using face aligned images

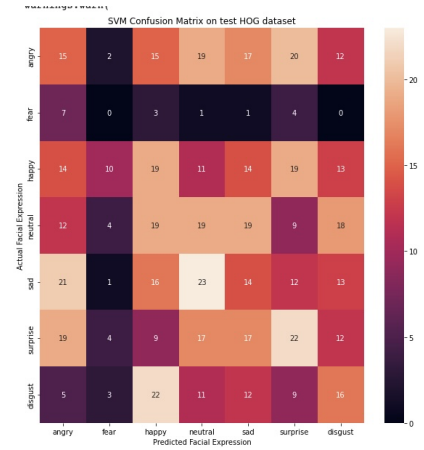
	KNN multiclass accuracy score on test ALIGNED dataset: 0.274411 KNN multiclass recall score on test ALIGNED dataset: 0.245192 KNN multiclass precision score on test ALIGNED dataset: 0.260101 KNN multiclass f1 score on test ALIGNED dataset: 0.243096

SVM

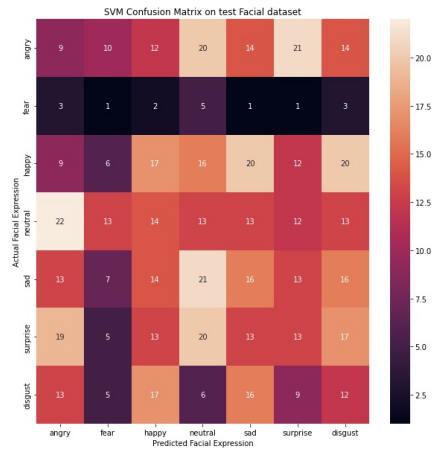
Using pixel values

	SVM multiclass accuracy score on test ORIG dataset: 0.341751 SVM multiclass recall score on test ORIG dataset: 0.330879 SVM multiclass precision score on test ORIG dataset: 0.339573 SVM multiclass f1 score on test ORIG dataset: 0.333695

Using HOG Features

	SVM multiclass accuracy score on test HOG dataset: 0.176768 SVM multiclass recall score on test HOG dataset: 0.156447 SVM multiclass precision score on test HOG dataset: 0.157838 SVM multiclass f1 score on test HOG dataset: 0.157027

Using Facial Features



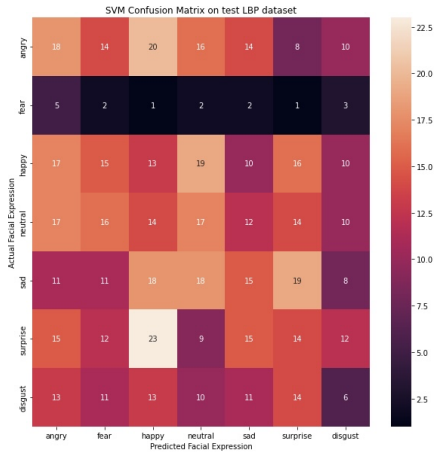
SVM multiclass accuracy score on test Facial dataset: 0.136364

SVM multiclass recall score on test Facial dataset: 0.128049

SVM multiclass precision score on test Facial dataset: 0.128875

SVM multiclass f1 score on test Facial dataset: 0.126417

Using LBP Features



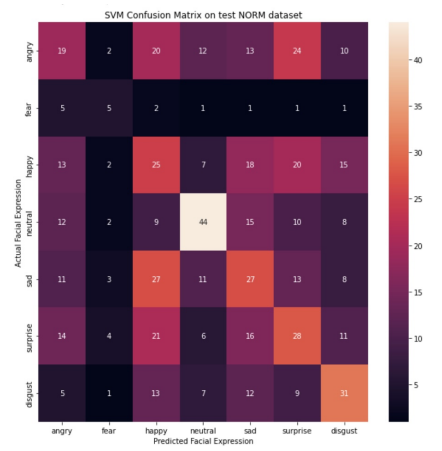
SVM multiclass accuracy score on test LBP dataset: 0.143098

SVM multiclass recall score on test LBP dataset: 0.138846

SVM multiclass precision score on test LBP dataset: 0.140116

SVM multiclass f1 score on test LBP dataset: 0.133909

Using normalized images



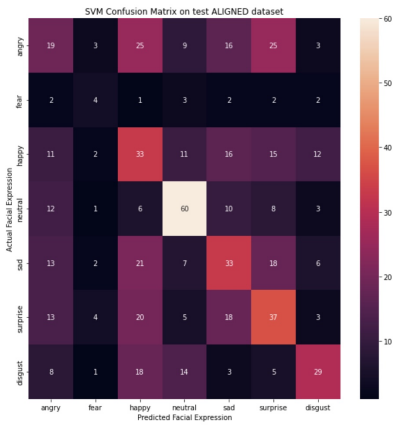
SVM multiclass accuracy score on test NORM dataset: 0.301347

SVM multiclass recall score on test NORM dataset: 0.305705

SVM multiclass precision score on test NORM dataset: 0.302537

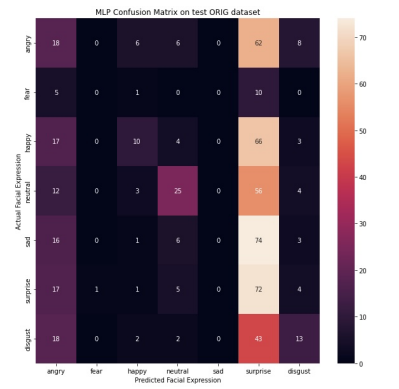
SVM multiclass f1 score on test NORM dataset: 0.302817

Using face aligned images

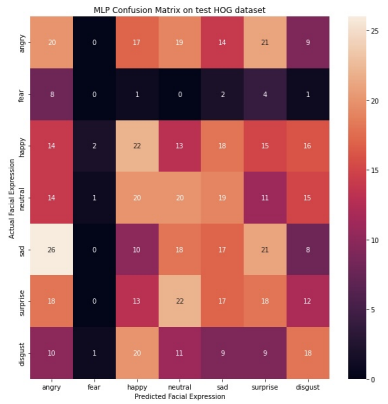
	SVM multiclass accuracy score on test ALIGNED dataset: 0.361953 SVM multiclass recall score on test ALIGNED dataset: 0.348828 SVM multiclass precision score on test ALIGNED dataset: 0.352653 SVM multiclass f1 score on test ALIGNED dataset: 0.348128

MLP

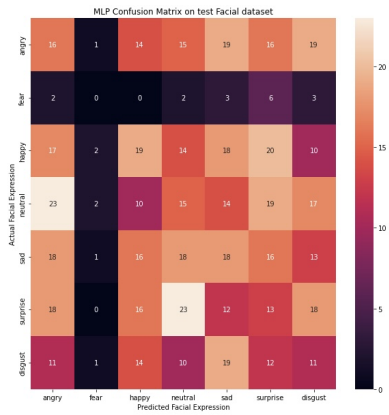
Using Pixel values

	MLP multiclass accuracy score on test ORIG set: 0.232323 MLP multiclass recall score on test ORIG set: 0.202381 MLP multiclass precision score on test ORIG set: 0.238811 MLP multiclass f1 score on test ORIG set: 0.172099

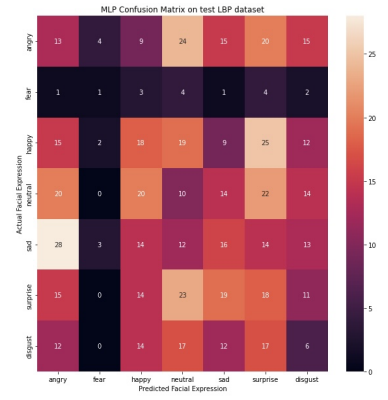
Using HOG Features

	MLP multiclass accuracy score on test HOG set: 0.193603 MLP multiclass recall score on test HOG set: 0.171538 MLP multiclass precision score on test HOG set: 0.168048 MLP multiclass f1 score on test HOG set: 0.1697

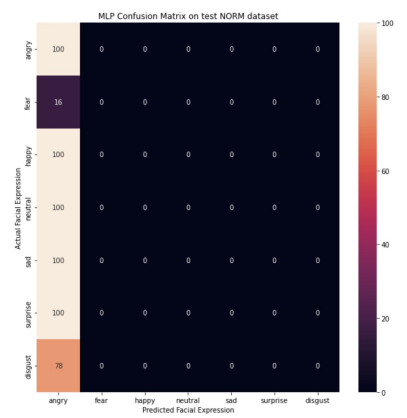
Using Facial Features

	MLP multiclass accuracy score on test Facial set: 0.154882 MLP multiclass recall score on test Facial set: 0.135861 MLP multiclass precision score on test Facial set: 0.134799 MLP multiclass f1 score on test Facial set: 0.135096

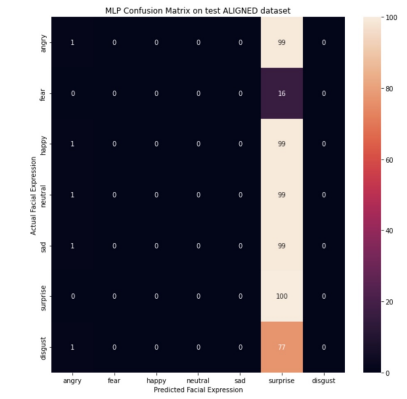
Using LBP Features

	MLP multiclass accuracy score on test LBP set: 0.138047 MLP multiclass recall score on test LBP set: 0.127060 MLP multiclass precision score on test LBP set: 0.132948 MLP multiclass f1 score on test LBP set: 0.128960

Using normalized images

	MLP multiclass accuracy score on test NORM set: 0.168350 MLP multiclass recall score on test NORM set: 0.142857 MLP multiclass precision score on test NORM set: 0.024050 MLP multiclass f1 score on test NORM set: 0.041169

Using face aligned images

	MLP multiclass accuracy score on test ALIGNED set: 0.170034 MLP multiclass recall score on test ALIGNED set: 0.144286 MLP multiclass precision score on test ALIGNED set: 0.052826 MLP multiclass f1 score on test ALIGNED set: 0.044189

We tried running KNN, SVM, MLP classifiers with the combination of different features. The winner is using SVM multiclass accuracy score on the test ALIGNED dataset. We see the classifier performance is very poor for fear of facial expressions across all models.

Hyperparameter tuning

We executed the SVM model using various hyperparameters and Gridsearch. Since the experiments were taking a lot of time, we grouped the hyperparameters and executed Gridsearch in two batches, the second batch using the results from the previous run. Below is the code snippet for these experiments.

The experiments were run using the validation dataset.

Batch 1 run:

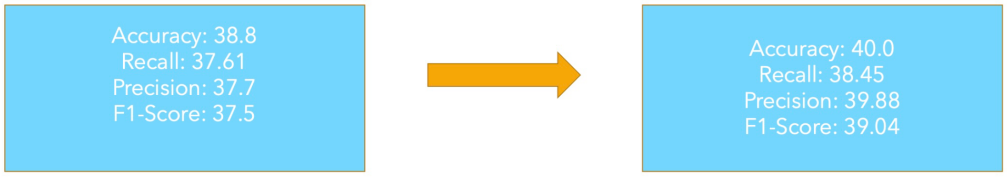
```
# identifying hyperparameters for SVC using the Aligned dataset

from sklearn.model_selection import GridSearchCV

penalty = ['l1', 'l2']
loss = ['squared_hinge']
C=[1.0, 2.0, 3.0]
max_iter = [1000, 2000, 3000]
dual =[False]
#Convert to dictionary
hyperparameters = dict(penalty=penalty,loss=loss,C=C, max_iter=max_iter, dual=dual)

#Create new SVC object
svm_model = LinearSVC()

#Use GridSearch to identify best model
clf = GridSearchCV(svm_model, hyperparameters, cv=5, scoring="accuracy", n_jobs = -1)

#Fit the model on validation dataset
best_model = clf.fit(X_valid_aligned,y_valid)
```

Batch 2 run:

```
tol = [0.0001,0.00001,0.000001]
loss = ['squared_hinge', 'hinge']
multi_class = ['ovr', 'crammer_singer']
dual = [True,False]

#Convert to dictionary
hyperparameters = dict(penalty=penalty,loss=loss,C=C,max_iter=max_iter,dual=dual,tol=tol,multi_class=multi_class)
#Create new SVC object
svm_model = LinearSVC()

#Use GridSearch to identify best model
clf = GridSearchCV(svm_model, hyperparameters, cv=5, scoring="accuracy", n_jobs = -1)

#Fit the model on validation dataset
best_model = clf.fit(X_valid_aligned,y_valid)
```

Below are the results that were observed after hyperparameter tuning. Accuracy, Recall, Precision and F1-score improved by 1-2% over the baseline model with default hyperparameters.

penalty	loss	C	max Iter	tol	multi_class
l1 l2	squared_hinge hinge	1 2 3	1000 2000 3000	0.0001 (1e-4) 0.00001 (1e-5) 0.000001 (1e-6)	ovr crammer_singer



Conclusion

Distinguishing between human emotions is a difficult problem, even humans sometimes cannot agree on the meaning behind a facial expression from one image alone. Not to mention that not all emotions can be fully expressed and understood from a person's face only. Technical aspects such as image quality, how blurry or noisy the image is, lighting conditions can greatly affect the quality of the interpretation.

Our best model was SVM on aligned images. The other features did not prove to be useful for facial expression detection. An interesting observation was that practically all of the classifiers showed a very poor performance with images classified as "fear". It's worth further investigating this fact. When pulling out fear impressions, the image often faces with open mouths and wide eyes, which the classifier got confused with surprising and angry facial expressions.

Next direction in the development of a better algorithm should be trying to reproduce the success of several papers that use a Multi Cascade Convolutional Network. We have also tried CNN, and haven't got a better result. The next step is to try VGG and ResNet, some more deep learning algorithms to learn more nuances from facial expressions. Gathering a dataset with images of a better quality would increase chances of training a robust and valuable model.