# Facial Expression Analysis

***Introduction to the problem***
Over the last few years there has been an increased interest to automatic analysis of facial behavior and understanding facial features for making correct assumptions from them. The analysis of facial expression is based on Action Units, movements associated with changes in facial appearance. AU can be developed into a comprehensive system for distinguishing all possible visually distinguishable facial movements and evaluate what expression they form and how these expressions differ.

In our case, we have data that includes 52 feature vectors, split into training and test set. Half of the vectors (26) have been extracted from smiling faces, while the other half (26) have been extracted from faces of people displaying frown. Each vector has 17 components that account for the activation level of 17 Action Units, every value inside vector unit shows how intense particular muscle unit is, measuring from 0 to 1. Our task is to develop model capability to map every vector into its unit class (smile or frown).

***Description of the theory underlying the Gaussian Discriminant Functions***
To implement our model training, we must first decide how to distinguish between smiling and frowning classes. The key is to use probability in order to make decisions about data.

The discriminant functions in the special case that the likelihood *p(x|Ci)* assumes a Gaussian distribution are Gaussian DF. They represent the distance of a feature vector from the average of vectors belonging to specific class.

Returning to our case, we have 2 vector classes which represent smile and frown and the features inside vectors of each class are statistically independent. So, GDF provides that for every class and every feature we have a different Gaussian. The key is to calculate probabilities and make decisions about vector's belonging to class based on the likelihood criteria. Thus, multivariable Gaussian distribution is used, which parameters (mean and covariance) are calculated from the provided training data.

Gaussian Discriminant Function maximizes the probability of observing exactly one particular training set when we use our particular model and considered to be very effective in case with feature vectors.

***A description of the experimental setup***
I used Python numpy library to simplify certain calculations related with matrices which would be very time consuming to calculate manually, but GDF is estimated using right formula.

To carry the training, we must first extract feature vectors from our training and test files. Having training and test samples separately is the important condition to successfully train the model.
Create function *import_data* which gives us lists of vectors with features (AU) and separately list with class names for convenience.

Next step is to define function *calc_mean_cov* which calculates mean vectors for smiling and frowning vector classes and covariance matrices (using *np.cov()*), key components for implementing Gaussian Discriminant Function formula. Mention that covariance matrix is different for our smile class and frown class. Additionally, need to find out prior probabilities for each of our class (*calc_p_ci*), take into account that they are not equal as well.

Then having carried certain analysis of covariance and prior probabilities for our case we obtain Gaussian Discriminant Function, supposing that the likelihood *p(x|Ci)* has a normal distribution:

$$\gamma_i(\mathbf{x}) = -\frac{n}{2}\ln 2\pi - \frac{1}{2}\ln|\mathbf{\Sigma}| - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_i)^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_i) + \ln p(\mathcal{C}_i).$$

But before, we have to calculate components of this formula using numpy build in functions for estimating dot product of vectors, transpose of the difference from division of x_value (represent feature) on mean vector (*x_mean.T*), inverse of the covariance matrix (*cov_inv*). Thus, we calculate each component of the formula and add logarithm of prior probability to it.

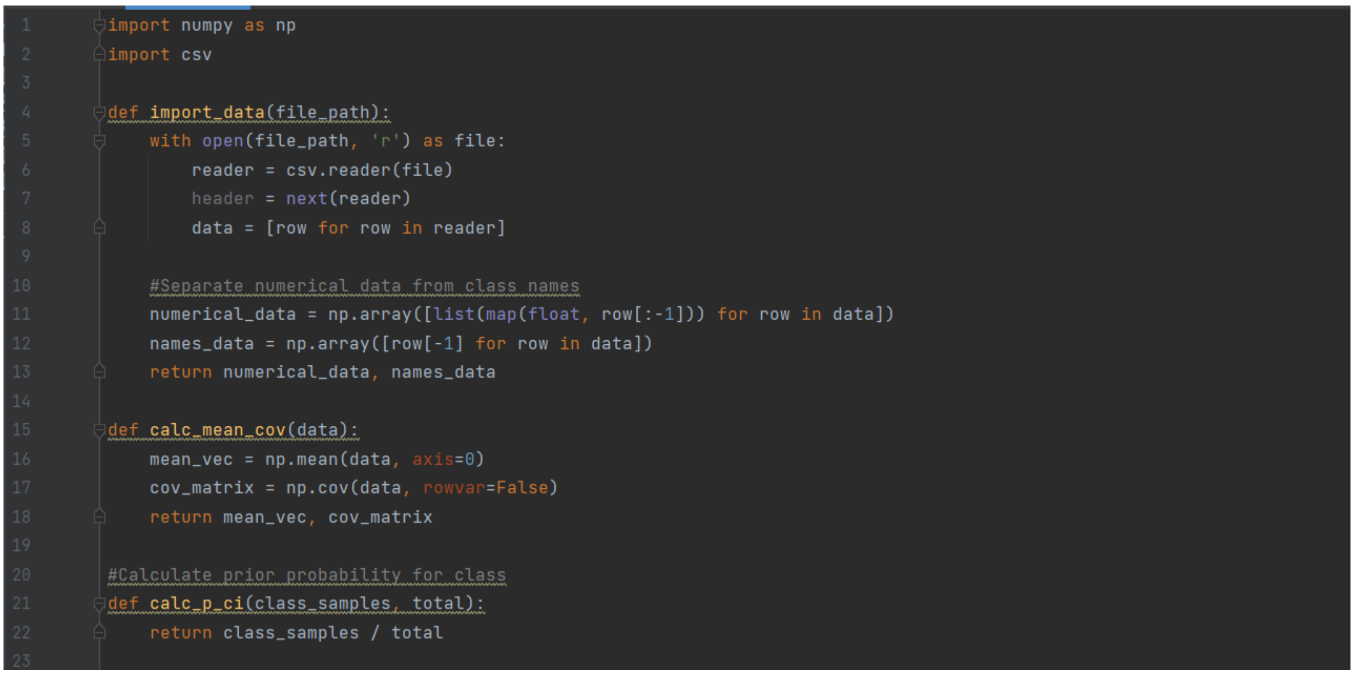
*train_model* is a function that calculates mean, covariance, prior probability for each of 2 classes; *test_model* carries test and implements QDF for smile and frown classes appending results in array *predictions*.

Last important function is for interpreting results comparing eliminated predicted class names with actual names, then calculating *error rate*, the percentage of times our approach maps a vector into the wrong class.

Finally, we carry on the process of training our model, consequently implementing each of the steps described above on loaded training and testing samples of sets of feature vectors.

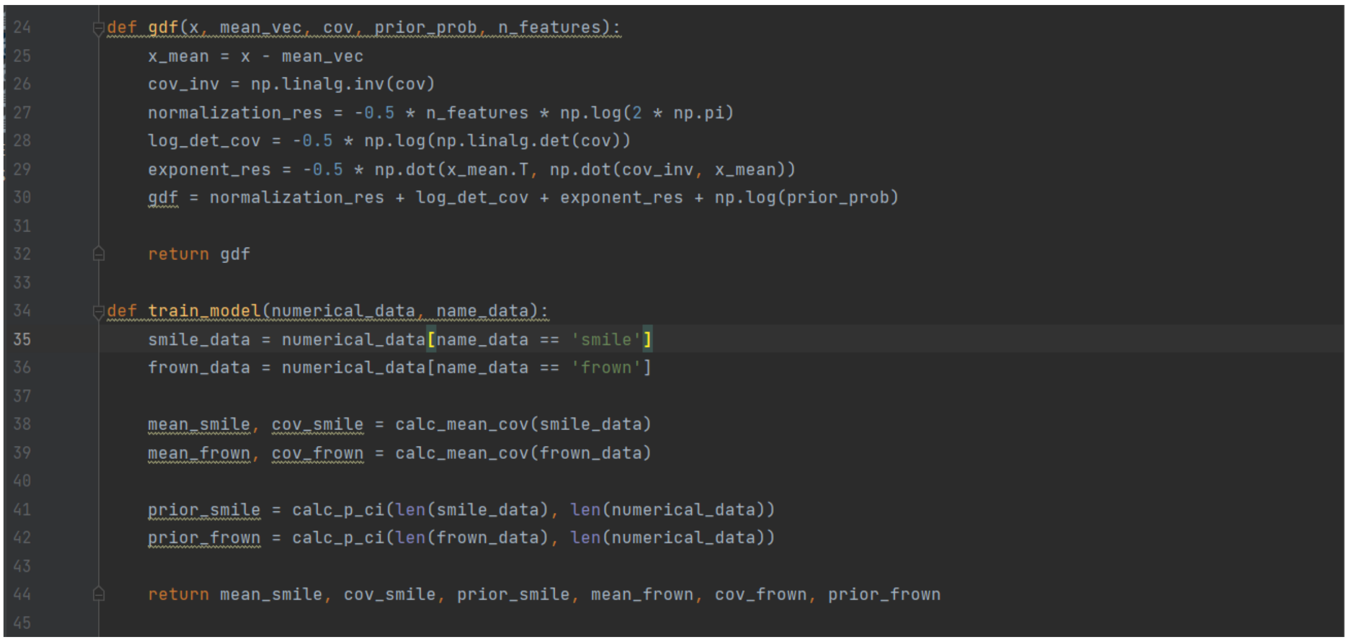
***Description of the results***

Results of training our model using Gaussian Discriminant Functions show that classifier incorrectly predicted the class in 6.25% of the time, making correct predictions in 93.75% of the test samples. We can conclude that our model is reasonably accurate and we succeeded in developing classifier capability to automatically map every vector into its class of smile or frown based on provided Action Units.

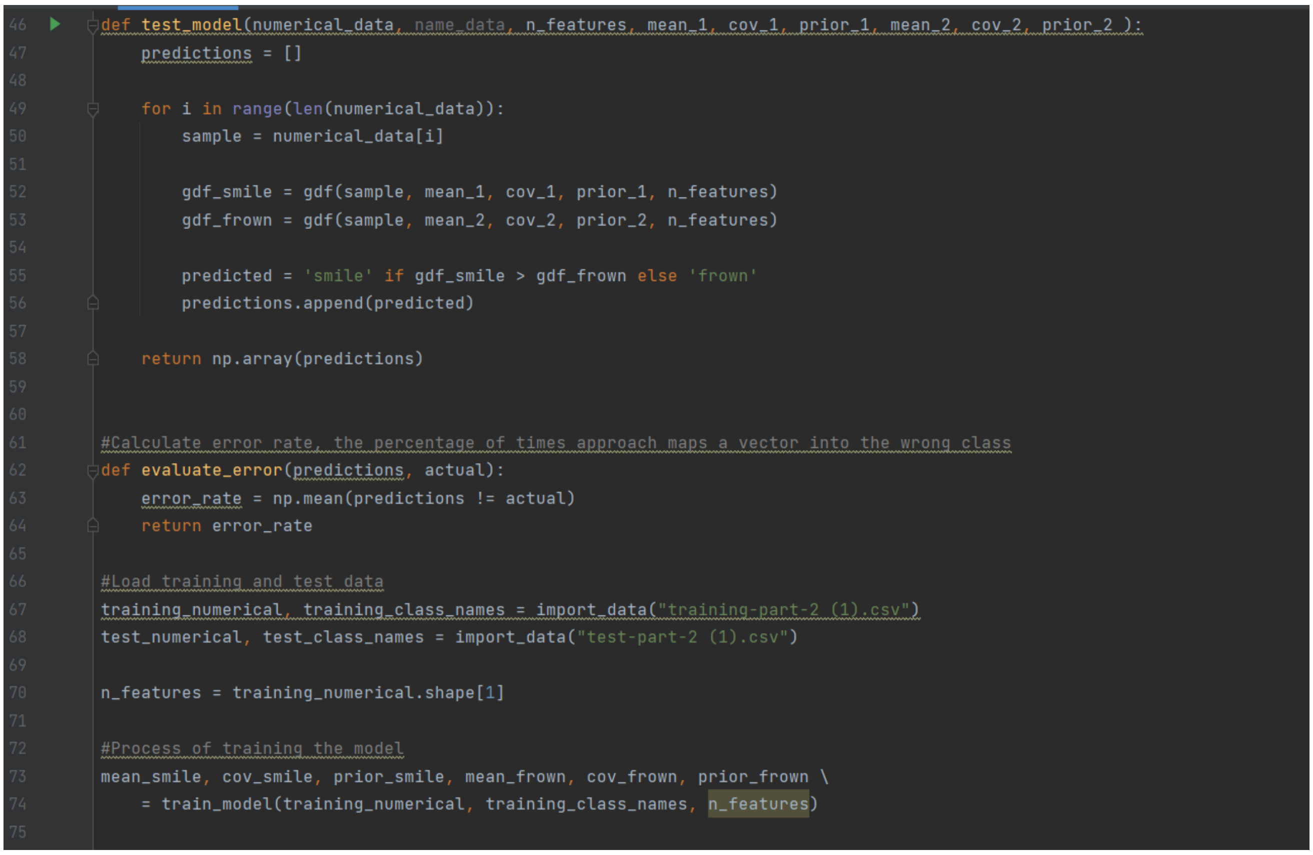


```python
import numpy as np
import csv

def import_data(file_path):
    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        header = next(reader)
        data = [row for row in reader]

    #Separate numerical data from class names
    numerical_data = np.array([list(map(float, row[:-1])) for row in data])
    names_data = np.array([row[-1] for row in data])
    return numerical_data, names_data

def calc_mean_cov(data):
    mean_vec = np.mean(data, axis=0)
    cov_matrix = np.cov(data, rowvar=False)
    return mean_vec, cov_matrix

#Calculate prior probability for class
def calc_p_ci(class_samples, total):
    return class_samples / total

```

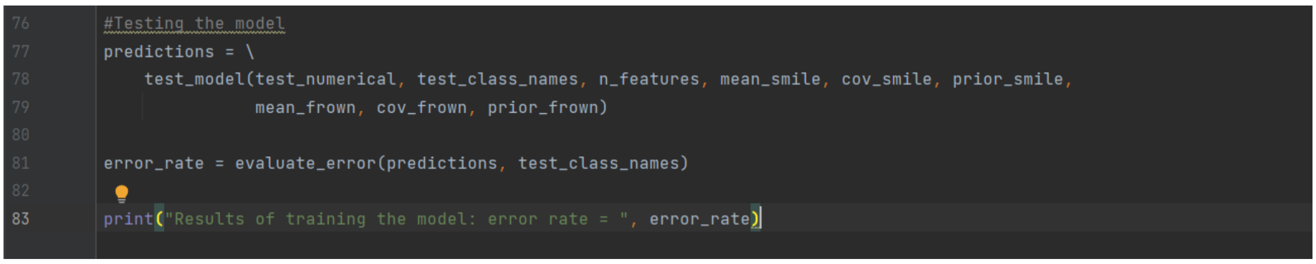
***Appendix with software***



```python
def gdf(x, mean_vec, cov, prior_prob, n_features):
    x_mean = x - mean_vec
    cov_inv = np.linalg.inv(cov)
    normalization_res = -0.5 * n_features * np.log(2 * np.pi)
    log_det_cov = -0.5 * np.log(np.linalg.det(cov))
    exponent_res = -0.5 * np.dot(x_mean.T, np.dot(cov_inv, x_mean))
    gdf = normalization_res + log_det_cov + exponent_res + np.log(prior_prob)

    return gdf

def train_model(numerical_data, name_data):
    smile_data = numerical_data[name_data == 'smile']
    frown_data = numerical_data[name_data == 'frown']

    mean_smile, cov_smile = calc_mean_cov(smile_data)
    mean_frown, cov_frown = calc_mean_cov(frown_data)

    prior_smile = calc_p_ci(len(smile_data), len(numerical_data))
    prior_frown = calc_p_ci(len(frown_data), len(numerical_data))

    return mean_smile, cov_smile, prior_smile, mean_frown, cov_frown, prior_frown

```

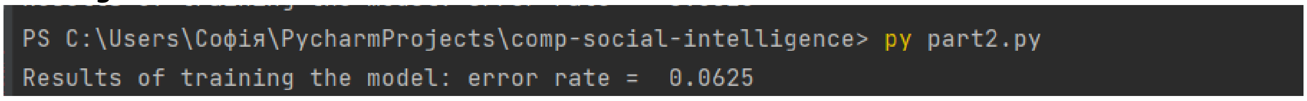


```python
def test_model(numerical_data, name_data, n_features, mean_1, cov_1, prior_1, mean_2, cov_2, prior_2 ):
    predictions = []

    for i in range(len(numerical_data)):
        sample = numerical_data[i]

        gdf_smile = gdf(sample, mean_1, cov_1, prior_1, n_features)
        gdf_frown = gdf(sample, mean_2, cov_2, prior_2, n_features)

        predicted = 'smile' if gdf_smile > gdf_frown else 'frown'
        predictions.append(predicted)

    return np.array(predictions)


#Calculate error rate, the percentage of times approach maps a vector into the wrong class
def evaluate_error(predictions, actual):
    error_rate = np.mean(predictions != actual)
    return error_rate

#Load training and test data
training_numerical, training_class_names = import_data("training-part-2 (1).csv")
test_numerical, test_class_names = import_data("test-part-2 (1).csv")

n_features = training_numerical.shape[1]

#Process of training the model
mean_smile, cov_smile, prior_smile, mean_frown, cov_frown, prior_frown \
    = train_model(training_numerical, training_class_names, n_features)

```



```python
#Testing the model
predictions = \
    test_model(test_numerical, test_class_names, n_features, mean_smile, cov_smile, prior_smile,
               mean_frown, cov_frown, prior_frown)

error_rate = evaluate_error(predictions, test_class_names)

print("Results of training the model: error rate = ", error_rate)
```

***Running the code:***



```
PS C:\Users\Софія\PycharmProjects\comp-social-intelligence> py part2.py
Results of training the model: error rate =  0.0625
```