
# CS 464 - Introduction to Machine Learning

## Homework Assignment 2

Yusuf Dalva

21602867
Section 2

December 11, 2019

Yusuf Dalva
21602867

# Homework 2

CS 464
December 11, 2019

## Question 1 - PCA & Cats

### Question 1.1:

For this question, as the data that the dimensionality is going to be reduced is 4096 dimensional(64 x 64), SVD approach used. In this manner the decomposition of the image data retrieved from the dataset provided for this question is decomposed with the method given below:

```
def get_svd_components(image_data):
    components = get_rgb_components(image_data)
    svd_red = linalg.svd(components[0])
    svd_red = [svd_red[0], np.diag(svd_red[1]), svd_red[2]]
    svd_green = linalg.svd(components[1])
    svd_green = [svd_green[0], np.diag(svd_green[1]), svd_green[2]]
    svd_blue = linalg.svd(components[2])
    svd_blue = [svd_blue[0], np.diag(svd_blue[1]), svd_blue[2]]
    return [svd_red, svd_green, svd_blue]
```

In the method given below, there are three parameters that are outputted by the svd method that is implemented by an external library(scipy.linalg.svd). The respective outputs are U (m x r matrix), S (vector with r dimensions) and $V^T$(r x n matrix). T explain the contents of these structures, brief explanations are given below:

- **Matrix U:** The coefficient matrix to reconstruct a matrix with the chosen eigenvalues and eigenvectors.
- **Vector S:** The vector containing the square roots of the eigenvalues. For computation, this vector is diagonalized into a matrix having dimensions r x r, where r is the number of eigenvalues found for the matrix.
- **Matrix $V^T$:** r x n matrix containing the eigenvectors of the matrix A, which the SVD method is applied. The eigenvectors subjected to this discussion is in the rows of matrix $V^T$.

In order to find the proportion of variance explained with a certain amount of eigenvectors the following formula is used:

$$\frac{\sum_{i=1}^{K} \lambda_i}{\sum_{i=1}^{N} \lambda_i} \tag{1}$$

Where K is the number of eigenvectors used and $\lambda$ is their respective value, N is the total number of eigenvectors found for the dataset with the SVD decomposition method. As mentioned in the explanation of the Vector S, the values outputted by this variable are the

square root values of the eigenvalues. Due to this fact when calculating the proportion of variance explained the squares of these values are taken. To calculate this variance preserved value, the following method is used, considering this discussion.

```
def get_k_principal_components(s, v_t, k):
    eigenvalues = np.square(np.diag(s))
    print('Proportion of variance preserved: ' +
    str(np.sum(eigenvalues[0:k]) / np.sum(eigenvalues)))
    for idx in range(k):
        print('Proportion of variance explained by eigenvalue '
        + str(idx) + ' : ' +
        str(eigenvalues[idx] / (np.sum(eigenvalues))))
    return v_t[0:k]
```

By using this method, three different PVE(Proportion of variance explained) value found, where there is a value for each color(red, green, blue). The results are given below:

```
PC analysis for red color:
Proportion of variance preserved: 0.9429543502314455
Proportion of variance explained by eigenvalue 0 : 0.8655538273524258
Proportion of variance explained by eigenvalue 1 : 0.02321516015811141
Proportion of variance explained by eigenvalue 2 : 0.018402944814549557
Proportion of variance explained by eigenvalue 3 : 0.012236007514589515
Proportion of variance explained by eigenvalue 4 : 0.0056464878202090295
Proportion of variance explained by eigenvalue 5 : 0.004512292983287661
Proportion of variance explained by eigenvalue 6 : 0.004386308641114517
Proportion of variance explained by eigenvalue 7 : 0.003455723474857545
Proportion of variance explained by eigenvalue 8 : 0.0030647092611860655
Proportion of variance explained by eigenvalue 9 : 0.0024808882111143774
PC analysis for green color:
Proportion of variance preserved: 0.9310301819610497
Proportion of variance explained by eigenvalue 0 : 0.8425230088999569
Proportion of variance explained by eigenvalue 1 : 0.025300943561787626
Proportion of variance explained by eigenvalue 2 : 0.021302134166077617
Proportion of variance explained by eigenvalue 3 : 0.013866453916350675
Proportion of variance explained by eigenvalue 4 : 0.007240810828342309
Proportion of variance explained by eigenvalue 5 : 0.0053161972176739795
Proportion of variance explained by eigenvalue 6 : 0.005053581669048906
Proportion of variance explained by eigenvalue 7 : 0.003871705447584532
Proportion of variance explained by eigenvalue 8 : 0.003601286951808314
Proportion of variance explained by eigenvalue 9 : 0.0029540593024188584
PC analysis for blue color:
Proportion of variance preserved: 0.9187456840267175
Proportion of variance explained by eigenvalue 0 : 0.816910336092947
Proportion of variance explained by eigenvalue 1 : 0.028499449260062534
Proportion of variance explained by eigenvalue 2 : 0.024424519814483188
Proportion of variance explained by eigenvalue 3 : 0.01645529713725322
Proportion of variance explained by eigenvalue 4 : 0.008788614747193923
Proportion of variance explained by eigenvalue 5 : 0.006128335993064822
Proportion of variance explained by eigenvalue 6 : 0.005818371346950794
Proportion of variance explained by eigenvalue 7 : 0.004669007358501018
Proportion of variance explained by eigenvalue 8 : 0.0037470597667111226
Proportion of variance explained by eigenvalue 9 : 0.0033046925095498066
```

Figure 1: PVE values found for PCA with 10 principal components

In summary by using 10 principal components in reconstruction of the images, 94% of the variance for red color, 93% of the variance for the green color and 92% of the variance for blue color can be explained. The values obtained for the distinct eigenvectors show that the proportion of variance that is explained drops drastically as more principal components are being used. This statement shows that, using few eigenvectors give a broad idea about the image and using more principle components only give small details about the image. The distinction between the first and second eigenvectors can be seen in the figure above: PVE for the first eigenvector is 87% and PVE for the second eigenvector is 2% (for color red as an example).

## Question 1.2:

Using the 10 eigenvectors found by SVD method the image reconstruction is performed. As the SVD method gives the eigenvectors for the matrix $A^T A$, in order to make the reconstruction the product of the vectors U, S and $V^T$ will be taken. As the first 10 eigenvectors will be used for reconstruction the first 10 columns of diagonalized form of S and the first 10 rows of the matrix $V^T$ will be used. To express this relation clearly, the following equation is given:

$$X_{reconstructed} = U * S_{FeaturesX10} * V^T_{10XFeatures} \tag{2}$$

The reconstruction explained for a predefined number of samples is done by the following method:

```
def apply_pca(image_data, k, sample_size):
    svd_comp = get_svd_components(image_data)
    pcs = rgb_pcs(svd_comp, k)
    rgb = []
    for color in range(3):
        rgb.append(np.dot(svd_comp[color][0][0:sample_size, 0: 4096],
        np.dot(svd_comp[color][1][:, 0:k],
        svd_comp[color][2][0:k])))
    return np.array(rgb)
```

This method is applied to all of the colors(red, green and blue) and then the first 10 images in the dataset is constructed. For shaping and reconstruction of the images the following method in being used:

```
def reconstruct_images(image_data, k, sample_size):
    reconst = apply_pca(image_data, k, sample_size)
    for idx in range(reconst.shape[1]):
        image = []
        for dim_no in range(4096):
```

```
            image.append(np.array([reconst[0][idx][dim_no],
            reconst[1][idx][dim_no],
            reconst[2][idx][dim_no]]))
    plt.figure()
    plt.imshow(np.array(image).reshape((64,64,3)).astype('uint8'))
```

In the reconstruction of the images, it is observed that the images are recognizable as cats and even they are distinguishable from each other. However, certain details about the which was the amount of variance that cannot be captured by the eigenvectors, where each eigenvector not used captures around 9-6% of variance, which explains why the details in these images are not visible here. Considering the image as a whole, the details that are not projected to the reconstruction are not essential in recognizing of the images but add the details that enables us to see them clearly. Overall, by using PCA method with 10 principal components the images are reconstructed preserving most of the variance that is in the dataset. The 10 images reconstructed is given below:

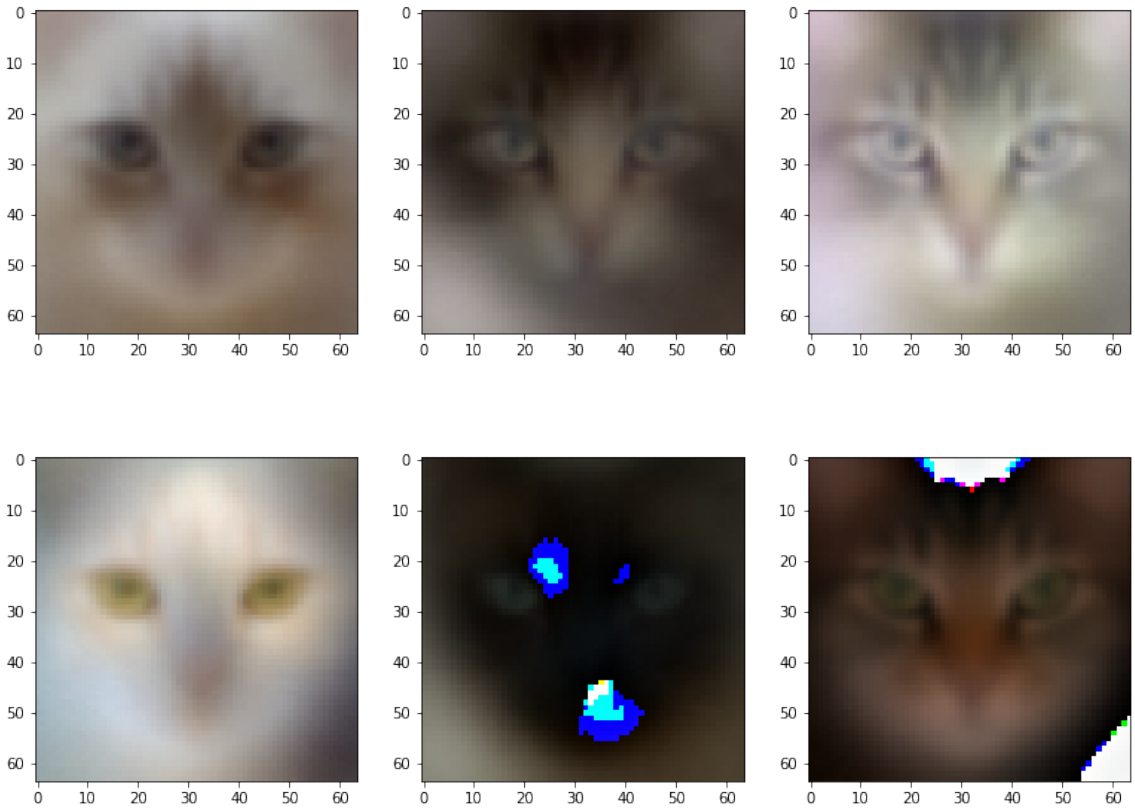

Figure 2: 10 images reconstructed by using PCA with 10 principal components

## Question 1.3:

For this part, 4 distinct values for number of principal components that are going to be used are tried to see their effect. From the conclusion reached at the end of the previous question, it is expected that the image will be recognizable even with one principal component but as more principal components added, the amount of details(proportion of variance preserved) that are represented by the reconstructed image will increase. For this part, the methods that are mentioned above are used with different k values. Before reconstructing the images directly, initially the proportion of variance preserved values calculated with the methodology mentioned in part *Question 1.1.* Applying the method *get_k_principal_components*, the PVE values for different k values are given below:

```
k = 1
Red:
Proportion of variance preserved: 0.8655538273524258
Green:
Proportion of variance preserved: 0.8425230088999569
Blue:
Proportion of variance preserved: 0.816910336092947
```

```
k = 50
Red:
Proportion of variance preserved: 0.9710930275548801
Green:
Proportion of variance preserved: 0.9650152664603874
Blue:
Proportion of variance preserved: 0.9591578748913245
k = 250
Red:
Proportion of variance preserved: 0.9876949416515256
Green:
Proportion of variance preserved: 0.9852397471939854
Blue:
Proportion of variance preserved: 0.9826079834650848
k = 500
Red:
Proportion of variance preserved: 0.9925574534646755
Green:
Proportion of variance preserved: 0.9910981884943049
Blue:
Proportion of variance preserved: 0.9894876479732455
```

Supporting the discussion provided above, the amount of variance explained by the eigenvectors has a decreasing pattern(in terms of distinct eigenvectors). The majority of the variance can be explained by using only the first principal component provided by the method SVD, as more eigenvectors added the PVE value increases but not in great amounts. as it can be seen from the data, after using the first principal component, the use of principal components 2-50 adds 11% PVE value, using principal components 51-250 adds 2% PVE value and so on. This shows that the images that are constructed with more principal components will contain much more detail but after a certain threshold, the image can be completely recognizable without using all of the eigenvectors. The images constructed were supporting this argument. As mentioned in the lecture slides, the principal components after 50 can be given up for efficiency purposes if the complete neatness of the images are not that crucial in the application. As it is observed in the images displayed, the principal components added after k = 50, does not have a crucial effect in image clarity. So giving up these principal components will not result in much loss. However, by giving up the principal components after k = 1, the loss is around 14% which can be considered as not tolerable. After inspecting the individual contributions of the eigenvectors to the PVE a suitable threshold would be k = 20, where the amount PVE by the eigenvectors after $20^th$ eigenvector is below 1%. The reconstructed images for k = 1, 50, 250, 500 are given below, for a suitable comparison the same sample is being reconstructed for all k values.

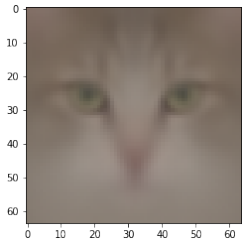

Figure 3: Reconstruction of sample 1 with k = 1

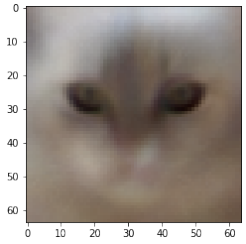

Figure 4: Reconstruction of sample 1 with k = 50

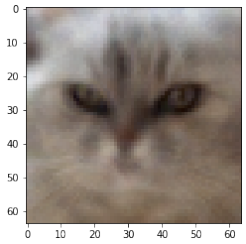

Figure 5: Reconstruction of sample 1 with k = 250

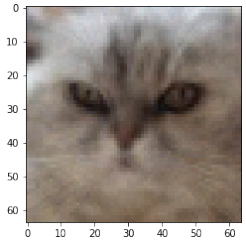

Figure 6: Reconstruction of sample 1 with k = 500

# Question 2 - Linear Regression

## Question 2.1:

Using the general form equation for the least squares loss function that is given by equation 2.1 in the assignment, the closed form solution for the weights($\beta$) is derived in the following

manner:

In order to find the closed form solution for the weights for the linear regression model, the least squares error function must be minimized such that:

$$\frac{\partial J_n}{\partial \beta} = 0 \tag{3}$$

Applying Matrix differentiation on the given equation, the following equality obtained:

$$\frac{\partial J_n}{\partial \beta} = \frac{\partial (y^T - \beta^T X)(y - X\beta)}{\partial \beta} \tag{4}$$

$$\frac{\partial J_n}{\partial \beta} = \frac{\partial}{\partial \beta}(y^T y - y^T X\beta + \beta^T X^T X\beta) \tag{5}$$

$$\frac{\partial J_n}{\partial \beta} = -2X^T y + 2X^T X\beta = 0 \tag{6}$$

Evaluating the expression given above, the following equality obtained for the closed form:

$$2X^T X\beta = 2X^T y \tag{7}$$

$$\hat{\beta} = (X^T X)^{-1} X^T y \tag{8}$$

The matrices and vectors used for the evaluation of the closed form of the weight vector are $\beta$ itself, y and X. The meanings of these variables are given below:

- **Vector y:** The numeric outcomes for the data samples that are given in the dataset.
- **Matrix x:** Denotes the data samples in a matrix form, the features in the first column is always 1 ($x_o = 1$) for every data point and the features that are given in the other columns in the matrix are the features obtained from the dataset about that sample.
- **Vector $\beta$:** The weight vector that represents the weights for each of the features that are given as the rows of matrix X.

## Question 2.2:

Using the equation derived from *Question 2.1*, the optimal weights which minimizes the value of the Mean Square Error(MSE) function is computed with the methods given below:

```
def get_x_matrix_model_a(data, y):
    x = np.zeros((y.shape[0],2))
    for idx in range(y.shape[0]):
        x[idx] = [1, data[idx]] # 1 for x_0
    return x
```

```
def compute_optimal_weights(x,y):
# x is the feature matrix, y is the outcome vector
    x_t_x = np.dot(np.transpose(x), x) # (x transpose) dot (x)
    inv = np.linalg.inv(x_t_x)
    beta_1 = np.dot(inv, np.transpose(x))
    return np.dot(beta_1, y)
```

The method *get_x_matrix_model_a* extracts the x matrix to be used for the calculations and the method *compute_optimal_weights* performs the calculation derived in Question 2.1.

As mentioned in the problem statement this model (Model A), only uses the 'Months Past Since 2014' data for a linear regression to predict the value of USD/TRY currency rate at a particular time. After using trivial methods to determine the linear regression model, the two weights ($w_0$ and $w_1$) are given below with the figure for the linear regression model A:

```
Weight 0: 2.002371537112798
Weight 1: 0.0634281614322188
MSE for model A: 0.1878714027639879
```

Figure 7: Weights found for Linear Regression Model A

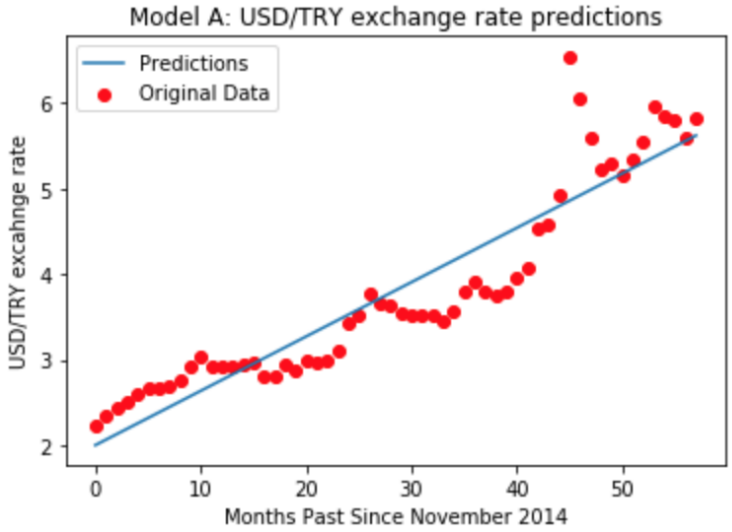


Figure 8: Months Since November 2014 vs. USD/TRY exchange rate plot for Linear Regression Model A

As it may be seen from the plot given in the figure below, there is a linear relationship between the 'Months Past Since November 2014' value and the USD/TRY exchange rate due to number of variables used for prediction(which is 1).

## Question 2.3:

As the attribute 'CPI' is also added to the features that is going to be used for prediction of USD/TRY exchange rate, the linear relationship between the feature 'Months Past Since November 2014' is expected to be nonlinear in that case.

For this part, the method to obtain the feature matrix is updated in the following way:

```
def get_x_matrix_model_b(col1, col2, y):
    x = np.zeros((y.shape[0], 3))
    for idx in range(y.shape[0]):
        x[idx] = [1, col1[idx], col2[idx]] # 1 for x_0
    return x
```

In the same way for making predictions in Question 2, the optimal weights for prediction is found by the help of the method *compute_optimal_weights*. Using these methods, the optimal weights and the Mean Square Error(MSE) computed for Model B. The results are given below:

```
Weight 0: -3.2841654813776393
Weight 1: -0.010325747619400985
Weight 2: 0.02338648171824541
MSE for Model B: 0.0891442586694583
```

Figure 9: Optimal Weights and MSE found for Model B

As shown in the figure below, the linear regression now consists of 2 features. Because of that, it is observed that the linear relationship between the feature 'Months Past since November 2014' and 'USD/TRY exchange rate' is not maintained anymore. Observing the MSE value for Model B, it can be seen that the error value decreased significantly, that explains these two variables explain the data better than one variable given in Question 2.2. The plot for Linear Regression model B is given below:

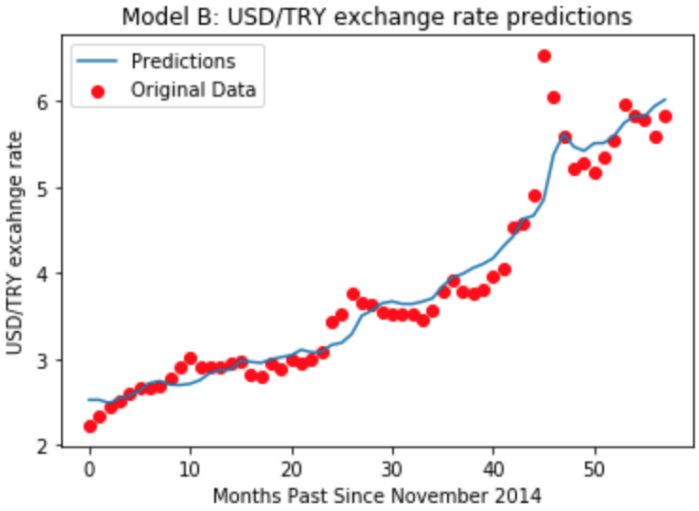


Figure 10: Months Past Since November 2014 vs USD/TRY exchange rate plot for Model B

## Question 2.4:

Using the same methodology with the previous two parts, now to construct Model C, the feature 'Unemployment Rate' is also used in addition to the two parameters used for Model B. By using another parameter it is expected naturally that the MSE value for Model C will be lower than the value found for Model B. However when the difference is inspected, the difference is very low which is due to closely correlated behavior of the features CPI and Unemployment Rate. The Weights found for this model is given below: As it can be observed from the comparison of the MSE values for Model B and Model C, the addition of the Unemployment Rate attribute has nearly no effect on the prediction accuracy and even

```
Weight 0: -3.2900591745731824
Weight 1: -0.01018874946956318
Weight 2: 0.023293011571620 92
Weight 3: 0.0015308366268742962
MSE for Model C: 0.08913535296420344
```

Figure 11: Optimal Weights and MSE value found for Model C

has a negative effect (really small). By using the values found for the weights, it can be seen that there are closely correlated values in the features, which are CPI and Unemployment Rate here. To support this argument the plot for Model C is given below, which shows great similarity with the plot for Model B.

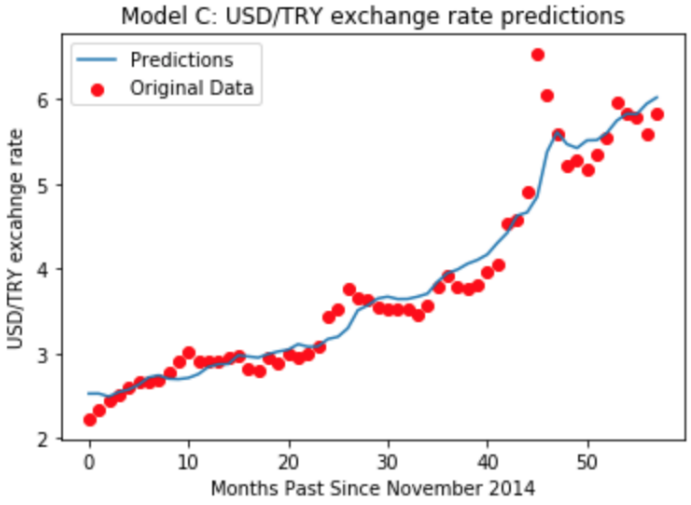


Figure 12: Months Past Since November 2014 vs USD/TRY exchange rate plot for Model C

## Question 2.5:

The division of the data into training set and the test set is done with the procedure given below:

```
test_set = dataset.iloc[-3:]
training_set = dataset.iloc[0:-3]
def evaluate_models(train, test):
    # Training data
    col1_train = training_set['Months Past Since November 2014'].to_numpy()
```

```
    col2_train = training_set['CPI'].to_numpy()
    col3_train = training_set['Unemployment Rate'].to_numpy()
    outcomes_train = training_set['Price'].to_numpy()
    # Test Data
    col1_test = test_set['Months Past Since November 2014'].to_numpy()
    col2_test = test_set['CPI'].to_numpy()
    col3_test = test_set['Unemployment Rate'].to_numpy()
    outcomes_test = test_set['Price'].to_numpy()
    # Model A
    x_a = get_x_matrix_model_a(col1_train, outcomes_train)
    w_a = compute_optimal_weights(x_a, outcomes_train)
    print(w_a)
    pred = evaluate_model_A(col1_test, w_a)
    mse_a = get_MSE(pred, outcomes_test)
    # Model B
    x_b = get_x_matrix_model_b(col1_train, col2_train, outcomes_train)
    w_b = compute_optimal_weights(x_b, outcomes_train)
    print(w_b)
    pred = evaluate_model_B(col1_test, col2_test, w_b)
    mse_b = get_MSE(pred, outcomes_test)
    # Model C
    x_c = get_x_matrix_model_c
        (col1_train, col2_train, col3_train, outcomes_train)
    w_c = compute_optimal_weights(x_c, outcomes_train)
    print(w_c)
    pred = evaluate_model_C(col1_test, col2_test, col3_test, w_c)
    mse_c = get_MSE(pred, outcomes_test)
    return (mse_a, mse_b, mse_c)
```

By using method *evaluate_models*, the MSE values for all three models are evaluated and then plotted with a helper method that is not given here for not to be irrelevant. The weights and the MSE values obtained for these models are given below:

```
Weights:
Model A: [2.02311968 0.062292]
Model B: [-3.58466521 -0.01285093  0.02460791]
Model C: [-3.66437537 -0.01214502  0.02406634  0.01128299]
MSE values
A: 0.0630578697927954 , B: 0.08556456925682403  , C: 0.10555750471011359
```

The plot showing the comparison of the models in terms of the MSE values is given below:

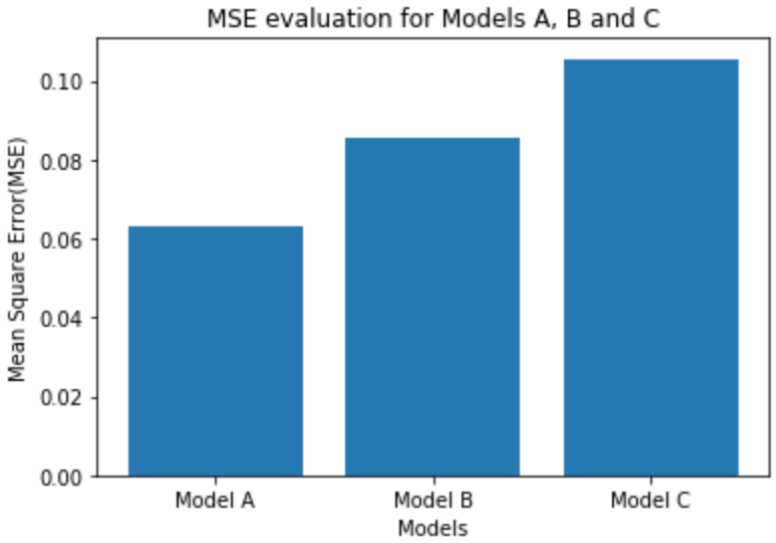


Figure 13: Evaluation of Models in terms of MSE values

As it can be seen from the figure, the models behaved just the opposite comparing the previous evaluations of the models. In order find a proper reason, the contents of the training set and the test set is observed accordingly. Comparing the behavior of data in training set and the test set, in the training set a proper relation with the parameter 'Months Past Since November 2014' feature and the USD/TRY exchange rate. Looking at the other models, the features 'CPI' and 'Unemployment Rate' are fluctuating values and the behavior of these attributes are different in the training set and the test set. To avoid these issues, while selecting the test set, the time dependency should be eliminated due to provide a better generalization and a better evaluation of the test data.

Using the MSE values found, it can be said that Model A is better at predicting the USD/TRY exchange rate values. Mentioning again, the time bias is an important issue here. As a comment to the models, these models are not good for predicting the USD/TRY exchange rates. Even though they perform well in terms of MSE values, the training data given only investigates an increasing behavior and does not have any information on fluctuating values. As the exchange rate has many dependencies, a model with more parameters about the subject and a dataset with diverse values that does not depend on time would enable a model that gives better estimates about the USD/TRY exchange rate.

## Question 2.6:

For this subject matter, I would propose a completely new model that does not depend on time constraints due to possible fluctuations in the exchange rates in the stock market. The choice would be another linear regression model, with a more extended dataset that contains

more features about the subject. In case of the constraint of not being able to change the dataset, I would follow a model that does not depend on time but using 'CPI' and 'Unemployment Rate' features. By using this kind of a model, the main aim is to eliminate bias from time. Due to increasing behavior of the dataset in time, the approach containing the feature 'Months Past Since November 2014' is just like sorting the values in increasing order and creating dependency on the index of the data. This approach has nothing to do with the behavior of the exchange rate and most likely to give false predictions in a trend that the USD/TRY exchange rate drops.

# Question 3 - Support Vector Machines (SVMs)

## Question 3.1:

For the implementation of the SVM classifier the software package sklearn.svm is used. For the linear classifier, which is the subject for this question, the class LinearSVM is being used specifically. For implementation of the specified classifier and validating it by using 5-fold cross validation, initially a method has been created to construct a model with a given C parameter, which is the independent variable for this question. The implementation of that method is given below:

```
def train_linear_binary_SVM(train_feature_set, train_labels, c_value):
    model = svm.LinearSVC(C = c_value, max_iter = 2000000,
    random_state = 0)
    model.fit(train_feature_set, train_labels)
    return model
```

At the process of running the tests on the training data, the issue of the objective function not converging occurred. In order to overcome the issue of convergence, the maximum number of iterations that can be done for minimizing the objective function that is given below:

$$min||\mathbf{w}||^2 + C\sum_{1}^{N}\varepsilon_i \tag{9}$$

Where w is the weight vector for the decision boundary and the support vectors in the SVM model. By intuition, it is expected that larger the C value gets, minimization will enforce the classifier to apply a narrow margin, meaning that ignoring the classification error will be harder($\varepsilon_i$). Following this training process, the next task that is being performed in the training process was trying different C values in the training set and deciding on the optimal C value(hyper-parameter). In order to achieve this 5-fold cross validation is used on training set and the performance metrics on the folds are being calculated. In the implementation, initially the training data is divided into 5 folds and each fold will be used as a validation set

respective to the training set. Here there are 5 folds for validation and 5 different slices that are going to be used as training set for the folds. The method that is being used to construct the folds and perform the cross validation is given below:

```
def evaluate_linear_binary_SVM(train_feature_set, train_label_set, c):
    folds = []
    label_folds = []
    run_features = []
    run_labels = []
    models = []
    idx = 0
    metrics = []
    for fold_no in range(5):
        folds.append(np.array(train_feature_set[idx:(idx +
        (train_feature_set.shape[0] // 5))]))
        label_folds.append(np.array(train_label_set[idx:(idx +
        (train_label_set.shape[0] // 5))]))
        run_features.append(np.concatenate((train_feature_set[0:idx],
        train_feature_set[idx + (train_feature_set.shape[0] // 5):])))
        run_labels.append(np.concatenate((train_label_set[0:idx],
        train_label_set[idx + (train_label_set.shape[0] // 5):])))
        models.append(train_linear_binary_SVM(run_features[fold_no],
        run_labels[fold_no], c))
        metrics.append(evaluate_metrics(folds[fold_no],
        label_folds[fold_no], models[fold_no]))
        idx += (train_feature_set.shape[0] // 5)
    return metrics
```

This method returns on object named metrics which contains the decision matrix and the metrics specified in the question for the calculation of the macro averages. In this method a helper method named *evaluate_metrics* is being used to evaluate these metrics. The respective method is given below:

```
def evaluate_metrics(valid_set, valid_labels, model):
    predictions = model.predict(valid_set)
    tp, fp, fn, tn = 0, 0, 0, 0
    for idx in range(predictions.size):
        if valid_labels[idx] == 1:
            if valid_labels[idx] == predictions[idx]:
                tp += 1
            else:
                fn += 1
        else:
```

```
            if valid_labels[idx] == predictions[idx]:
                tn += 1
            else:
                fp += 1
    if tp + fp + tn + fn == 0:
        accuracy = 0
    else:
        accuracy = (tp + tn) / (tp + fp + tn + fn)
    if tp + fp == 0:
        precision = 0
        fdr = 0
    else:
        precision = tp / (tp + fp)
        fdr = fp / (tp + fp)
    if tp + fn == 0:
        recall = 0
    else:
        recall = tp / (tp + fn)
    if tn + fn == 0:
        npv = 0
    else:
        npv = tn / (tn + fn)
    if fp + tn == 0:
        fpr = 0
    else:
        fpr = fp / (fp + tn)
    if precision +  recall == 0:
        f1 = 0
        f2 = 0
    else:
        f1 = (2 * precision * recall) / (precision + recall)
        f2 = (5 * precision * recall) / (4 * precision + recall)
    return [[tp, fp, fn, tn], accuracy, precision, recall, npv, fpr, fdr, f1, f2]
```

By using the given two methods ($evaluate_linear_binary_SVM$ and $evaluate_metrics$) the analysis for a certain C value in terms of performance metrics is done with the function given below:

```
def tune_linear_SVM(train_feature_set, train_label_set):
    c_set = (10**(-3), 10**(-2), 10**(-1), 1, 10**(1), 10**(2))
    acc_values = []
    recall_values = []
```

```
    fpr_values = []
    for c in c_set:
        print('For C = ' + str(c))
        metrics = evaluate_linear_binary_SVM(train_feature_set, train_label_set, c)
        acc_roc = analyze_metric_values(metrics)
        acc_values.append(acc_roc[0])
        recall_values.append(acc_roc[1])
        fpr_values.append(acc_roc[2])
    return [acc_values, recall_values, fpr_values]
```

Here, the method analyze_metric_values is used to compute the micro averages and the macro averages for the metrics desired. By using the method given above, different C values are applied to the linear SVM model and the performance can be observed with these metrics.

Using this method, the performance metrics value for each C value is provided below:

```
For C = 0.001
Accuracy: 0.825
Macro Precision: 0.0
Micro Precision: 0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.825
Micro NPV: 0.825
Macro FPR: 0.0
Micro FPR: 0.0
Macro FDR: 0.0
Micro FDR: 0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 14: Performance metrics for C = 0.001

```
For C = 0.01
Accuracy: 0.825
Macro Precision: 0.0
Micro Precision: 0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.825
Micro NPV: 0.825
Macro FPR: 0.0
Micro FPR: 0.0
Macro FDR: 0.0
Micro FDR: 0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 15: Performance metrics for C = 0.01

```
For C = 0.1
Accuracy: 0.8142857142857143
Macro Precision: 0.0
Micro Precision: 0.0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.8232900432900433
Micro NPV: 0.8231046931407943
Macro FPR: 0.012677304964539008
Micro FPR: 0.012987012987012988
Macro FDR: 0.4
Micro FDR: 1.0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 16: Performance metrics for C = 0.1

```
For C = 1
Accuracy: 0.7928571428571429
Macro Precision: 0.0
Micro Precision: 0.0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.8193697043697042
Micro NPV: 0.8191881918819188
Macro FPR: 0.03862715298885512
Micro FPR: 0.03896103896103896
Macro FDR: 0.8
Micro FDR: 1.0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 17: Performance metrics for C = 1

As it can be observed from the performance metrics calculated, the linear model cannot predict the true value and since the number of false labels are significantly more than the true labels, the selection of the optimal parameter is depending on the ROC curve produced by the

```
For C = 10
Accuracy: 0.7892857142857144
Macro Precision: 0.0
Micro Precision: 0.0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.818810654754051
Micro NPV: 0.8185185185185185
Macro FPR: 0.04279381965552178
Micro FPR: 0.04329004329004329
Macro FDR: 0.8
Micro FDR: 1.0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 18: Performance metrics for C = 10

```
Accuracy: 0.7892857142857144
Macro Precision: 0.0
Micro Precision: 0.0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.818810654754051
Micro NPV: 0.8185185185185185
Macro FPR: 0.04279381965552178
Micro FPR: 0.04329004329004329
Macro FDR: 0.8
Micro FDR: 1.0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 19: Performance metrics for C = 100

recall and FPR values. As the AUC(Area Under Curve) metric gives significant information about how distinguishing the model between the 1 and 0 labels, the AUC approach is followed. However, since the linear model does not output any TP values, looking at the sum TPR + FPR would be more suitable to be sure that the model selects a positive label at least. To investigate this issue for this part, the following method is being used and a decision is being made accordingly.

```
def test_linear_binary_model(train_feature_set, train_label_set, test_feature_set, test_
    results = tune_linear_SVM(train_feature_set, train_label_set)
    print(results)
    plot_results(results[0], 'C', ('0.001', '0.01', '0.1', '1', '10', '100'))
    c_set = (10**(-3), 10**(-2), 10**(-1), 1, 10, 100)
    acc_auc_c = []
    for idx in range(len(c_set)):
        acc_auc_c.append((c_set[idx], results[1][idx] + results[2][idx],
        results[0][idx]))
    acc_auc_c.sort(key=lambda tup: tup[1])
    print(acc_auc_c)
    print('Optimum C value: ' + str(acc_auc_c[len(acc_auc_c) - 1][0]),
    ', respective accuracy: ' + str(acc_auc_c[len(acc_auc_c) - 1][2]))
```

```
print('Slope of ROC curve for optimum c value: ' +
str(acc_auc_c[len(acc_auc_c) - 1][1]))
model = train_linear_binary_SVM(train_feature_set,
train_label_set, acc_auc_c[len(acc_auc_c) - 1][0])
metrics = evaluate_metrics(test_feature_set, test_label_set, model)
print('For the final model: ')
print('Accuracy: ' + str(metrics[1]))
print('Precision: ' + str(metrics[2]))
print('Recall: ' + str(metrics[3]))
print('NPV: ' + str(metrics[4]))
print('FPR: ' + str(metrics[5]))
print('FDR: ' + str(metrics[6]))
print('F1-Measure: ' + str(metrics[7]))
print('F2-Measure: ' + str(metrics[8]))
confusion_matrix = pd.DataFrame([[metrics[0][0],
metrics[0][1]], [metrics[0][2], metrics[0][3]]],
        columns = ['1', '0'], index = ['1', '0'])
sb.heatmap(confusion_matrix, annot = True)
plt.xlabel('Actual Value')
plt.ylabel('Predicted Value')
plt.title('Confusion Matrix for C value ' +
str(acc_auc_c[len(acc_auc_c) - 1][0]))
plt.show()
```

As the result of running this method, the performace metrics of the SVM model with no kernel is given below:

```
Optimum C value: 100 , respective accuracy: 0.7892857142857144
TPR + FPR: 0.04279381965552178
For the final model:
Accuracy: 0.8260869565217391
Precision: 0.5
Recall: 0.1
NPV: 0.8378378378378378
FPR: 0.021052631578947368
FDR: 0.5
F1-Measure: 0.16666666666666669
F2-Measure: 0.11904761904761904
```

Following the testing process, it is observed that the trained model has the ability of recognizing the true values even if it is with low recall. This reasoning can be seen clearly from the confusion matrix. The confusion matrix is given below:

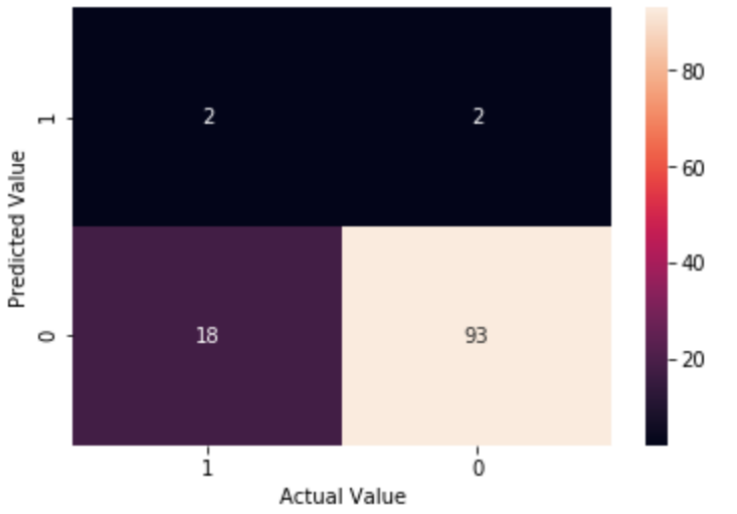


Figure 20: Confusion matrix for the final model with C = 100

As a concluding remark, it can be seen that the linear kernel is not suitable for the given dataset, in terms of identifying the classes and classifying the data accordingly. As the distribution of the training data to the dataset is not close to even in any way, evaluating the dataset in terms of accuracy would be misleading. Due to that, such approach is not followed. For example, the maximum accuracy value found for C values is 82.5%, which is obtained by labeling all instances as 0, which is the same with not applying any classification at all. The dominance of the 0 classes can be seen from the value of TPR + FPR, which is around 0.4%. To take into account the power of classifying the class 1, the approach of taking the sum of TPR and FPR is followed.

## Question 3.2:

Different than the first part of this question, since the kernel that is going to be used for SVM model is not linear, the SVC class from the sklearn.svm package will be used instead of the package LinearSVC. The updated method for constructing the SVM models is given below:

```
def train_RBF_binary_SVM(train_feature_set, train_label_set, gamma):
    model = svm.SVC(kernel = 'rbf', gamma = gamma, max_iter = 2000000,
    C = 100, random_state = 0)
    model.fit(train_feature_set, train_label_set)
    return model
```

The C value is kept constant for all of the values of gamma for this question. To give details about the RBF kernel, it can be identified with the following equation:

$$K(x, x^{'}) = exp(-\frac{||x - x^{'}||^2}{2\sigma^2}) \tag{10}$$

With the same method for cross-validation, which is 5-fold cross validation, the gamma values given in the problem statement is tested. The methods used only differs from the ones used in the second part only differs in terms of using train_RBF_binary_SVM instead of train_linear_binary_SVM, beacuse of that helper methods that are used for cross validation and the performance metrics are not provided for a second time. In order to overcome the issue caused by the imbalance of the dataset, the approach of comparing macro TPR + macro FPR for different gamma values followed instead of accuracy. As expected, the number of zero labels outputted was very high, due to that the distinguishing power of the model between 0 label values and the 1 label values was crucially important when deciding on the optimal gamma value. The decision for the gamma value and the training of the final model is done with the method given below:

```
def test_RBF_binary_model(train_feature_set, train_label_set,
test_feature_set, test_label_set):
    results = tune_RBF_SVM(train_feature_set, train_label_set)
    print(results)
    plot_results(results[0], 'Gamma', ('0.0625', '0.125', '0.25', '0.5', '1', '2'))
    gamma_set = (2**(-4), 2**(-3), 2**(-2), 2**(-1), 1, 2)
    acc_auc_gamma = []
    for idx in range(len(gamma_set)):
        acc_auc_gamma.append((gamma_set[idx], results[1][idx] +
        results[2][idx], results[0][idx]))
    acc_auc_gamma.sort(key=lambda tup: tup[1])
    print(acc_auc_gamma)
    print('Optimum Gamma value: ' + str(acc_auc_gamma[len(acc_auc_gamma) - 1][0]),
    ', respective accuracy: ' + str(acc_auc_gamma[len(acc_auc_gamma) - 1][2]))
    print('F1 Measure for optimal Gamma value: ' +
    str(acc_auc_gamma[len(acc_auc_gamma) - 1][1]))
    model = train_RBF_binary_SVM(train_feature_set,
    train_label_set, acc_auc_gamma[len(acc_auc_gamma) - 1][0])
    metrics = evaluate_metrics(test_feature_set, test_label_set, model)
    print('For the final model: ')
    print('Accuracy: ' + str(metrics[1]))
    print('Precision: ' + str(metrics[2]))
    print('Recall: ' + str(metrics[3]))
    print('NPV: ' + str(metrics[4]))
    print('FPR: ' + str(metrics[5]))
```

```
    print('FDR: ' + str(metrics[6]))
    print('F1-Measure: ' + str(metrics[7]))
    print('F2-Measure: ' + str(metrics[8]))
    confusion_matrix = pd.DataFrame([[metrics[0][0], metrics[0][1]],
    [metrics[0][2], metrics[0][3]]],
        columns = ['1', '0'], index = ['1', '0'])
    sb.heatmap(confusion_matrix, annot = True)
    plt.xlabel('Actual Value')
    plt.ylabel('Predicted Value')
    plt.title('Confusion Matrix for Gamma value ' +
    str(acc_auc_gamma[len(acc_auc_gamma) - 1][0]))
    plt.show()
    return acc_auc_gamma[len(acc_auc_gamma) - 1][0]
```

Using this method for model evaluation the gamma value 0.0625 was found out to be the optimal gamma value for the given dataset. The micro and macro performance metrics calculated for this question is given below:

```
For Gamma = 0.0625
Accuracy: 0.7821428571428573
Macro Precision: 0.04
Micro Precision: 0.07142857142857142
Macro Recall: 0.025
Micro Recall: 0.02040816326530612
Macro NPV: 0.8205909776498013
Micro NPV: 0.8195488721804511
Macro FPR: 0.054964539007092195
Micro FPR: 0.05627705627705628
Macro FDR: 0.76
Micro FDR: 0.9285714285714286
F1 Measure Value: 0.03076923076923077
F2 Measure Value: 0.02702702702702702
```

Figure 21: Performance metrics for $\gamma = 0.0625$

```
For Gamma = 0.125
Accuracy: 0.8214285714285715
Macro Precision: 0.1
Micro Precision: 0.3333333333333333
Macro Recall: 0.025
Micro Recall: 0.02040816326530612
Macro NPV: 0.827061087061087
Micro NPV: 0.8267148014440433
Macro FPR: 0.008421985815602837
Micro FPR: 0.008658008658008658
Macro FDR: 0.3
Micro FDR: 0.6666666666666666
F1 Measure Value: 0.04
F2 Measure Value: 0.029411764705882353
```

Figure 22: Performance metrics for $\gamma = 0.125$

```
For Gamma = 0.25
Accuracy: 0.8214285714285715
Macro Precision: 0.0
Micro Precision: 0.0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.8244155844155845
Micro NPV: 0.8243727598566308
Macro FPR: 0.00425531914893617
Micro FPR: 0.004329004329004329
Macro FDR: 0.2
Micro FDR: 1.0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 23: Performance metrics for $\gamma = 0.25$

```
For Gamma = 0.5
Accuracy: 0.8214285714285715
Macro Precision: 0.0
Micro Precision: 0.0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.8244155844155845
Micro NPV: 0.8243727598566308
Macro FPR: 0.00425531914893617
Micro FPR: 0.004329004329004329
Macro FDR: 0.2
Micro FDR: 1.0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 24: Performance metrics for $\gamma = 0.5$

```
For Gamma = 1
Accuracy: 0.825
Macro Precision: 0.0
Micro Precision: 0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.825
Micro NPV: 0.825
Macro FPR: 0.0
Micro FPR: 0.0
Macro FDR: 0.0
Micro FDR: 0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 25: Performance metrics for $\gamma = 1$

```
For Gamma = 2
Accuracy: 0.825
Macro Precision: 0.0
Micro Precision: 0
Macro Recall: 0.0
Micro Recall: 0.0
Macro NPV: 0.825
Micro NPV: 0.825
Macro FPR: 0.0
Micro FPR: 0.0
Macro FDR: 0.0
Micro FDR: 0
F1 Measure Value: 0.0
F2 Measure Value: 0.0
```

Figure 26: Performance metrics for $\gamma = 2$

The reported accuracy values for the different gamma values is given in the plot below:

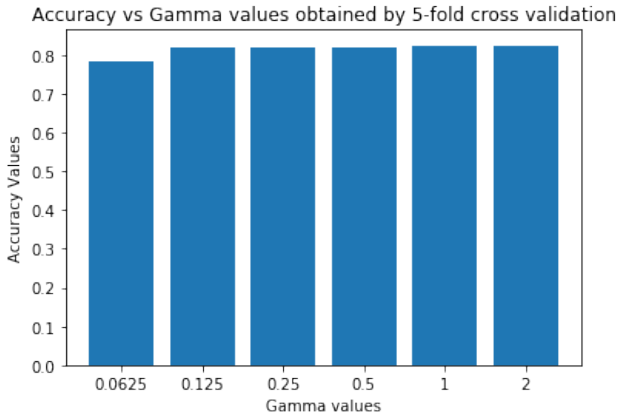


Figure 27: Accuracy values obtained for different gamma values

Following this evaluation the final model is trained with $\gamma = 0.0625$ and the following confusion matrix and the performance metrics obtained as a result of testing:

```
Optimum Gamma value: 0.0625 , respective accuracy: 0.7821428571428573
TPR + FPR : 0.07996453900709219
For the final model:
Accuracy: 0.8347826086956521
Precision: 1.0
Recall: 0.05
NPV: 0.8333333333333334
FPR: 0.0
FDR: 0.0
F1-Measure: 0.09523809523809523
F2-Measure: 0.0617283950617284
```

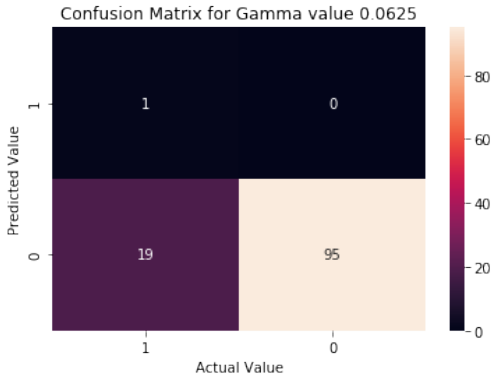


Figure 28: Accuracy values obtained for different gamma values

## Question 3.3:

For training a multi-class classifier for this part of the question, the capabilities of the SVC class is used with the option given as "one-vs-rest"(ovr) to train a classifier with a "one-vs-all" methodology. The method used for training the model separated as it is separated in the first two parts of the question. The training method is given below:

```
def train_RBF_multiclass_SVM(train_feature_set, train_label_set, gamma_val):
    model = svm.SVC(kernel = 'rbf', gamma = gamma_val, C = 1000,
        decision_function_shape = 'ovr')
    model.fit(train_feature_set, train_label_set)
    return model
```

As the model trained is a multiclass classifier now, the confusion matrix after a training process is constructed differently. The method used for constructing the decision matrix for this part is given below:

```
def evaluate_multiclass_confusion(valid_features, valid_set, model):
    predictions = model.predict(valid_features)
    class_0 = [0,0,0]
    class_1 = [0,0,0]
    class_2 = [0,0,0]
    for idx in range(predictions.size):
        if predictions[idx] == 0:
            if predictions[idx] == valid_set[idx]:
                class_0[0] += 1
            elif valid_set[idx] == 1:
                class_0[1] += 1
            else:
                class_0[2] += 1
        if predictions[idx] == 1:
            if predictions[idx] == valid_set[idx]:
                class_1[1] += 1
            elif valid_set[idx] == 1:
                class_1[0] += 1
            else:
                class_1[2] += 1
        if predictions[idx] == 2:
            if predictions[idx] == valid_set[idx]:
                class_2[2] += 1
            elif valid_set[idx] == 1:
                class_2[1] += 1
            else:
```

```
                class_2[0] += 1
    return [class_0, class_1, class_2]
```

After the construction of the confusion matrix the variables TP, FP, FN, TN are evaluated for ease of computation. After that, using the same evaluation methods used before, the desired components are found both in terms of micro and macro average. For this question, the optimal parameters found in the previous parts are being used (C = 1000 and $\gamma = 0.0625$). The results according to the evaluation of the cross validation is given below:

```
Accuracy: 0.4868600170161653
Macro Precision: 0.5665089775540586
Micro Precision: 0.5677749360613811
Macro Recall: 0.35015699223728414
Micro Recall: 0.3490566037735849
Macro NPV: 0.0
Micro NPV: 0.0
Macro FPR: 1.0
Micro FPR: 1.0
Macro FDR: 0.43349102244594134
Micro FDR: 0.4322250639386189
F1 Measure Value: 0.4324819347782163
F2 Measure Value: 0.3789794613540224
```

For the construction of the 2x2 confusion matrix the following method is used:

```
def evaluate_multiclass_metrics(valid_features, valid_set, model):
    confusion_matrix = evaluate_multiclass_confusion
    (valid_features, valid_set, model)
    accuracy = (confusion_matrix[0][0] + confusion_matrix[1][1] +
    confusion_matrix[2][2]) / valid_set.size
    # tp, fp, fn, tn
    class_0_matrix = [0,0,0,0]
    class_1_matrix = [0,0,0,0]
    class_2_matrix = [0,0,0,0]
    class_0_matrix[0] = confusion_matrix[0][0]
    class_1_matrix[0] = confusion_matrix[1][1]
    class_2_matrix[0] = confusion_matrix[2][2]
    class_0_matrix[1] = confusion_matrix[0][1] +
    confusion_matrix[0][2]
    class_1_matrix[1] = confusion_matrix[1][0] +
    confusion_matrix[1][2]
    class_2_matrix[1] = confusion_matrix[2][0] +
    confusion_matrix[2][1]
```

```
    class_0_matrix[2] = confusion_matrix[0][1] +
    confusion_matrix[0][2] + confusion_matrix[1][2] +
    confusion_matrix[2][1]
    class_1_matrix[2] = confusion_matrix[0][1] +
    confusion_matrix[0][2] + confusion_matrix[2][0] +
    confusion_matrix[2][1]
    class_2_matrix[2] = confusion_matrix[0][1] +
    confusion_matrix[0][2] + confusion_matrix[1][0] +
    confusion_matrix[1][2]
    class_0_matrix[0] = confusion_matrix[1][1] +
    confusion_matrix[2][2]
    class_1_matrix[0] = confusion_matrix[0][0] +
    confusion_matrix[2][2]
    class_2_matrix[0] = confusion_matrix[0][0] +
    confusion_matrix[1][1]
    return [class_0_matrix, class_1_matrix, class_2_matrix]
```

Where each matrix here has the data structure of [TP, FP, FN, TN]. According to the analysis during the cross validation the confusion matrix is constructed in the following way: Commenting on the confusion matrix and the performance metric values obtained it can

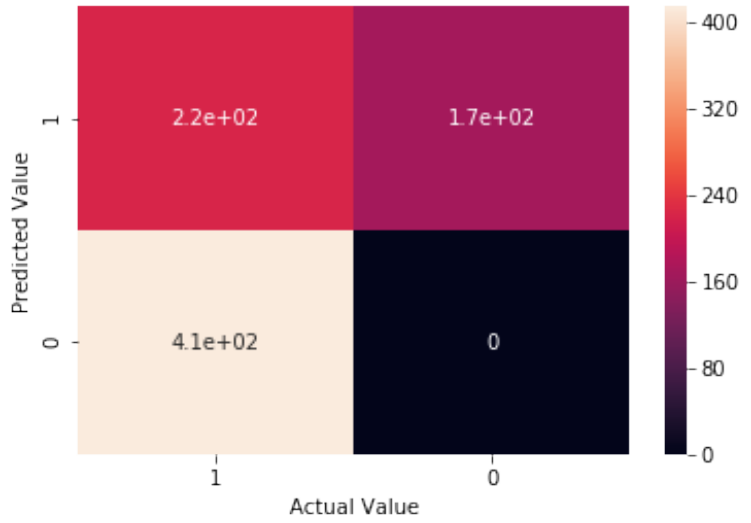


Figure 29: Generalized version of the predictions that is used for analysis, for easiness only values of TP, FP, FN, TN are shown

be said that this classifier is better than a random classifier with respect to its accuracy value. Considering all of the values that are predicted during the cross validation, it can be concluded the model is poor at predicting true negatives. It tells that the patterns that the

model that can't classify are prediction errors all the time. Considering the true prediction rate, the rate is not promising for the the classes in a cumulative way. If the classification of a class is crucial, it can be concluded that the model is not that dependable for this approach. The final confusion matrix is provided below: The evaluation of the test data also supports

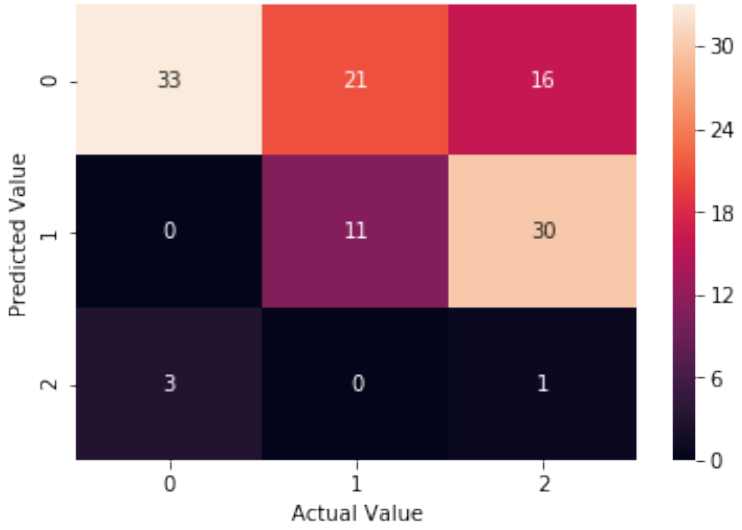


Figure 30: Confusion matrix for Multiclass classification

the argument that is provided above. As like in the binary models, the model tends to predict 0 label most of the time(significantly). The label value 2 is predicted only 4 times, meaning a big difference in the frequencies of label prediction. Considering the accuracy in the prediction the dataset can still be considered as imbalanced in terms of 0 values, which is an obstacle against good results with SVM approach.

As one versus all approach is shown to produce result 0 compared to result 1 in the first two parts, by using all-versus-all approach the majority of samples will be predicted as 0 which will produce imbalanced predictions. At the second step the classes will be distinguished between 1 and 2 respectively. As the binary classifier is proved to produce imbalanced results one-versus-all was not expected to give any good results in terms of class imbalance, which is the case observed after the testing process. By using all-vs-all approach, the class distinction each of the classes fill be compared with their original values. However, due to poor performance of the binary SVM classifiers in the given dataset it is not expected to give good results. This result can be concluded that the instance numbers between all of the classes is imbalanced between each other. Computationally there will be 3 * (3 - 1) = 6 different classifiers to distinguish between the three classes. So the all-vs-all approach seems as computationally expensive in this case. But since the classification between imbalanced classes results with not being able to classifying the minority class in our case, while comparing the classes it is a great possibility that class 2 will be classified as class 0 due to great amount of imbalance (70 samples of label 0 is present in test set, with 4 samples of label 2).

# Question 4 - Logistic Regression

## Question 4.1:

After investigating the dataset provided, from the decimal places that the dataset contains in its principal components, a normalization process is required to process the data for the gradient ascent approach. As in the problem statement it is not stated whether the class 1 or class 0 is above the decision boundary, as a starting point the class that has more numbers of instances that is above the decision boundary is going to be treated accordingly and the classification is going to be done accordingly after applying gradient ascent approach. The following method is used for determining the initial decision for the decision boundary and the initial weights as random values from the normal distribution with mean 0 and standard deviation 0.01. The implementation is given below:

```
def decide_decision_boundary(train_matrix, mean, stdev, labels):
    w = np.random.normal(mean, stdev, train_matrix.shape[1])
    positive_above_bound = 0
    negative_above_bound = 0
    for data_idx in range(train_matrix.shape[0]):
        boundary = np.sum(np.dot(train_matrix[data_idx], w))
        if boundary > 0:
            if labels[data_idx] == 1:
                positive_above_bound += 1
            else:
                negative_above_bound += 1
    if negative_above_bound > positive_above_bound:
        return (w, 0)
    return (w, 1)
```

In addition to this step, for applying the logistic regression a normalization process is needed to be applied to the dataset and an initial feature $x_0$ will be added, which is always equal to 1. This task is done with the following two methods:

```
def normalize_features(data):
    np_data = data.to_numpy()
    minimum = np.array(data.min())
    maximum = np.array(data.max())
    train_data = []
    for sample in np_data:
        train_data.append(np.divide(np.subtract(sample[0:-1],
        minimum[0:-1]), np.subtract(maximum[0:-1], minimum[0:-1])))
    return np.array(train_data)
```

```
def get_features(train):
    feature_matrix = np.zeros((train.shape[0], train.shape[1] + 1))
    for sample_idx in range(train.shape[0]):
        sample = [1]
        for feature_idx in range(train.shape[1]):
            sample.append(train[sample_idx][feature_idx])
        feature_matrix[sample_idx] = sample
    return feature_matrix
```

As it can be observed from the method normalize_features, the normalization process is done as the follows:

$$Z = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{11}$$

The value Z here is the new value of the feature constructed. For the discussion of the most important features, after a research about the topic, it is observed that in a logistic regression model, the weights of the features is a good indicator on the feature importance. In other words, more the weight is, more important the feature is. For calculating a step in the full-batch gradient ascent approach, the following method used:

```
def calc_full_batch_step(feature_matrix, weights, labels, boundary_type):
    if boundary_type == 0:
        y = 1 - labels
    else:
        y = labels
    reg_sum = np.dot(feature_matrix, weights)
    reg_sum = np.exp(reg_sum)
    reg_const = reg_sum / (1 + reg_sum)
    return np.dot(np.transpose(feature_matrix), (y - reg_const))
```

This approach for calculating the gradient ascent step involves a matrix based approach. In summary, the weight update amounts can be summarized with the following equation: If the class 0 is above the decision boundary:

$$\frac{\partial l(w)}{\partial w_i} = (1 - y_i)x_i - \frac{\exp(w_o + \sum_1^n w_i x_i)x_i}{1 + \exp(w_o + \sum_1^n w_i x_i)} \tag{12}$$

And if the class 1 is above the decision boundary:

$$\frac{\partial l(w)}{\partial w_i} = (y_i)x_i - \frac{\exp(w_o + \sum_1^n w_i x_i)x_i}{1 + \exp(w_o + \sum_1^n w_i x_i)} \tag{13}$$

To get the weight update vector, the following matrix equality is used:

$$\nabla w = X^T.[...\frac{\partial l(w)}{\partial w_i}...] \tag{14}$$

For full-batch gradient ascent approach, the matrix X will contain all of the features for the training set. As a final step, the gradient ascent step is applied in the following way:

```
def train_full_batch_model(feature_matrix, labels, step_size):
    initial_decision = decide_decision_boundary(feature_matrix, 0, 0.01, labels)
    weights = initial_decision[0]
    boundary_selection = initial_decision[1]
    for iteration in range(1000):
        weights = np.add(weights, step_size *
        calc_full_batch_step(feature_matrix, weights,
        train_labels, boundary_selection))
        if iteration \% 100 == 99:
            print('Weights at iteration ' +
            str(iteration + 1) + ': ' + str(weights))
            print('Most important weights: ' +
            str(report_most_important_weights(weights, 10)))
    return [boundary_selection, weights]
```

For the question, the feature importance is determined on the amount of weights which is a common method used for determining feature importance. As the algorithm tries to find the optimal weights for determining the classification, it is expected that the more important features will have higher weights in the regression. Here the crucial feature of the regression is, it determines a descriptive bound for the samples which makes it logical that the features with more weights has greater importance. The evaluation of the trained model is done as follows:

```
def predict_labels(test_set, weights, boundary_label):
    predictions = []
    for sample in test_set:
        if boundary_label == 0:
            if np.dot(sample, weights) > 0:
                predictions.append(0)
            else:
                predictions.append(1)
        else:
            if np.dot(sample,weights) > 0:
                predictions.append(1)
            else:
                predictions.append(0)
    return predictions

def evaluate_full_batch_model(train_set, train_labels, test_set, test_labels):
    train_features = normalize_features(train_set)
```

```
test_features = normalize_features(test_set)
train_label = train_labels.to_numpy()
test_label = test_labels.to_numpy()
train_features = get_features(train_features)
test_features = get_features(test_features)
decision = train_full_batch_model(train_features, train_label, 0.05)
boundary_label = decision[0]
weights = decision[1]
predictions = predict_labels(test_features, weights, boundary_label)
tp, fp, fn, tn = 0, 0, 0, 0
for idx in range(test_label.size):
    if test_label[idx] == 1:
        if test_label[idx] == predictions[idx]:
            tp += 1
        else:
            fn += 1
    else:
        if test_label[idx] == predictions[idx]:
            tn += 1
        else:
            fp += 1
print('Performance Metrics for Full-Batch Model:')
accuracy = (tp + tn) / (tp + tn + fp + fn)
print('Accuracy:' + str(accuracy))
precision = tp / (tp + fp)
print('Precision:' + str(precision))
recall = tp / (tp + fn)
print('Recall:' + str(recall))
npv = tn / (tn + fn)
print('Negative Predictive Value(NPV): ' + str(npv))
fpr = fp / (fp + tn)
print('False Positive Rate(FPR): ' + str(fpr))
fdr = fp / (tp + fp)
print('False Discovery Rate(FDR): ' + str(fdr))
f1 = (2 * precision * recall) / (precision + recall)
print('F1 Measure: ' + str(f1))
f2 = (5 * precision * recall) / (4 * precision + recall)
print('F2 Measure: ' + str(f2))
confusion_matrix = pd.DataFrame([[tp, fp], [fn, tn]],
columns = ['1', '0'], index = ['1', '0'])
sb.heatmap(confusion_matrix, annot = True)
plt.xlabel('Actual')
```

```
    plt.ylabel('Predicted')
    plt.title('Confusion Matrix for Full-Batch Gradient Ascent')
    plt.show()
    plt.clf()
    return [accuracy, precision, recall, npv, fpr, fdr, f1, f2]
```

Using this method the following performance metrics and the confusion matrix obtained:

```
Performance Metrics -- Full-Batch Model:
Accuracy:0.9047619047619048
Precision:0.8695652173913043
Recall:0.9523809523809523
NPV: 0.9473684210526315
FPR: 0.14285714285714285
FDR: 0.13043478260869565
F1 Measure: 0.909090909090909
F2 Measure: 0.9345794392523363
```

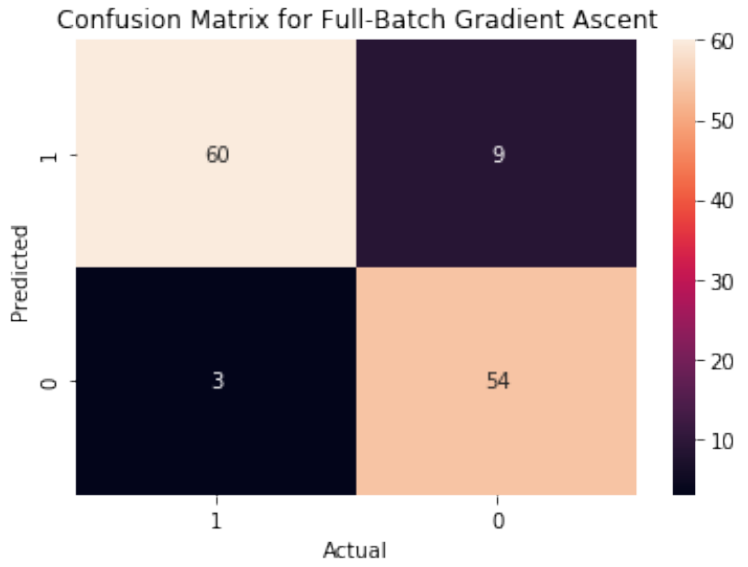


Figure 31: Confusion Matrix for Full-Batch Gradient Ascent Logistic Regression

According to these metrics that are outputted by the final model after 1000 iterations of full-batch gradient ascents(1000 epochs), it can be said that the model performs well on both of the classes (class 1 and class 0). For the class that is labeled as "fraud detected", recall is high which shows that if a fraud is present it will be classified "fraud detected" most probably(with 95% probability in the test set). For the predictions that are done as 1, approximately 87% of these accusations are accurate which the precision metric tell us. This may lead to the search of a more efficient classification algorithm as 13% (FDR) of the customers are accused wrongly which may be a problematic situation here. For the values that fraud is not detected, the results seems accurate as NPV is approximately 95% and the values not detected does not involve any fraud in this percentage. Regarding the discussion about precision and recall, as expected when the recall is favored the F measure gives a higher result, which is important if detection of fraud is more important than how accurate the fraud detection is.
For the learning rate, the value $\eta = 0.05$ seems accurate as it provides slow convergence and avoids missing the optimum point in a reasonable value(value is not extremely small)

## Question 4.2:

For this part of the question most of the modules are being used as for training method, only added feature is the batch size. Batch size for mini-batch gradient ascent algorithm is 32 and for stochastic gradient ascent approach, step size is 1. Even though they are implemented separately, to show how the algorithm proceeds only the method for mini-batch gradient ascent algorithm is given:

```
def train_mini_batch_model(feature_matrix, labels, step_size, batch_size):
    initial_decision = decide_decision_boundary(feature_matrix, 0, 0.01, labels)
    weights = initial_decision[0]
    boundary_selection = initial_decision[1]
    for idx in range(1000):
        batch_start = 0
        for i in range(feature_matrix.shape[0] // 32):
            weights = np.add(weights, step_size *
            calc_full_batch_step(feature_matrix[batch_start: batch_start + 32],
            weights, train_labels[batch_start: batch_start + 32],
            boundary_selection))
            batch_start = batch_start + 32
    return [boundary_selection, weights]
```

Evaluating the mini-batch model and the stochastic model, the following results obtained for mini-batch and stochastic approaches respectively:

```
Performance Metrics -- Mini-Batch Model:
Accuracy:0.8809523809523809
```

```
Precision:0.98
Recall:0.7777777777777778
NPV: 0.8157894736842105
FPR: 0.015873015873015872
FDR: 0.02
F1 Measure: 0.8672566371681417
F2 Measure: 0.8112582781456954

Performance Metrics -- Stochastic Model:
Accuracy:0.9047619047619048
Precision:0.9811320754716981
Recall:0.8253968253968254
NPV: 0.8493150684931506
FPR: 0.015873015873015872
FDR: 0.018867924528301886
F1 Measure: 0.8965517241379309
F2 Measure: 0.8524590163934426
```

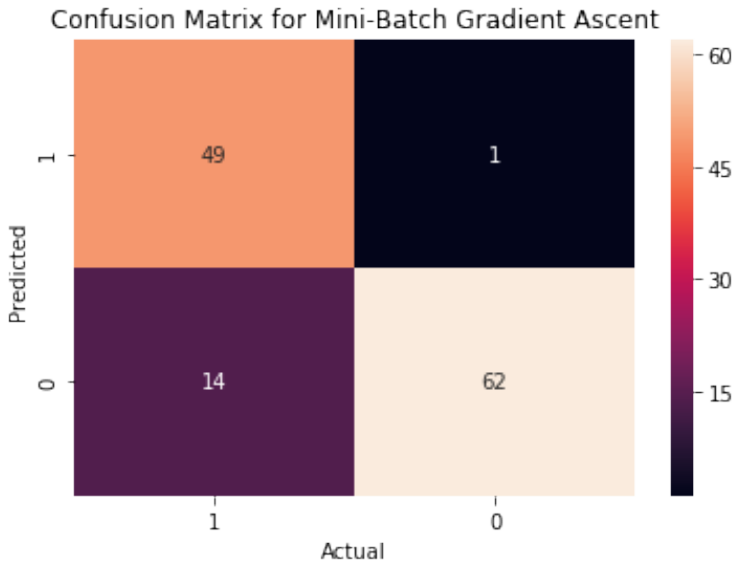


Figure 32: Confusion Matrix for Mini-Batch Gradient Ascent Logistic Regression

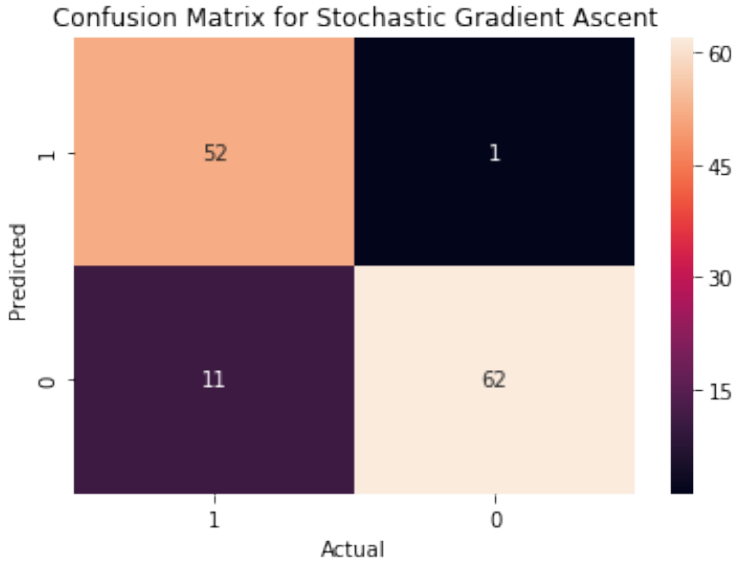


Figure 33: Confusion Matrix for Stochastic Gradient Ascent Logistic Regression

For clarification purposes, the comparison of the three gradient descent approaches are shown as bar charts, in terms of different metrics. As it can be seen from the plots representing the comparison of three gradient ascent approaches and the confusion matrices, it can be seen that the accuracy values seems really close that it is not descriptive in any way. The main differences in these metrics are in precision and recall(also in F1 and F2 measures respectively). In terms of the abilities of the models of predicting the positive values correctly, which the recall values tells us, the stochastic approach performs slightly better than the mini-batch gradient ascent algorithm. Comparing with the full-batch gradient ascent algorithm the full-batch approach performs better in terms of recall. Looking at the Recall values produced from three different approaches, the opposite scenario is valid. Looking at the overall results, by using the F1 measure it can be said that the Stochastic approach is the best approach in terms of predicting positives but full-batch gradient ascent approach is a close approximation in a situation that predicting all the true values is not crucial but predicting true values accurately is crucial (as it can be seen from the F2 measure value). Looking at the performance of not classifying(classifying as 0), the stochastic approach and the mini-batch gradient ascent performs better than the full-batch gradient ascent approach. Comparing the runtimes of the algorithms, due to efficiency of the vector operations, the full-batch approach completed the fastest where mini-batch approach follows it. However the stochastic approach performs better in almost all of the cases, the computational efficiency is a factor that needs to be considered. Thinking of the context given in the question the prediction predicting a value as positive has crucial importance for the banks and the customers so, despite the extra computational burden it brings, the stochastic approach seems the best approach for this case.

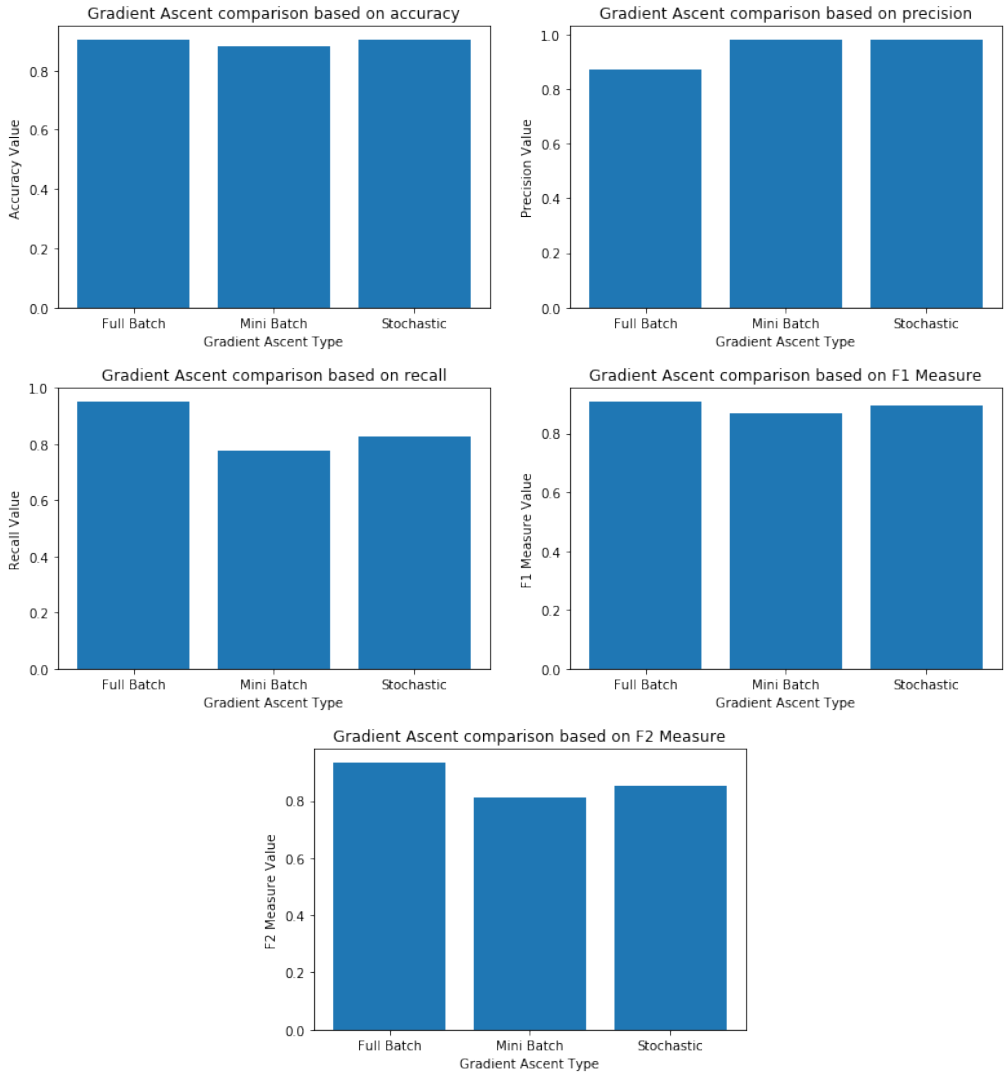


Figure 34: Comparison of performance metrics for three gradient ascent approaches

## Question 4.3:

As much as classification accuracy is an important factor in the context of deciding on the best model, related with the case some other metrics may be more informative.

In order to explain when the other metrics be more informative firstly the definitions of the metrics precision, recall and accuracy is given:

- **Precision:** Fraction of true positives among all positive predictions
- **Recall:** Fraction of true positive predictions among all positive samples
- **Accuracy:** Number of test samples that are labeled correctly over all of the predictions

made

Due to imbalanced data the metric accuracy may be misleading (which is the case for Question 3). For the precision and recall metrics, the use of them alone shows completely different results about the model. The precision metric shows that among all of the positive predictions, how much of the predictions is true. Recall states that among all of the positive samples how much of them can be labeled. For evaluation of the model in terms of predicting the true values, these metrics are complementary and needs to be used together in most of the cases. As a result the F-Measure is a good better metric as it considers them in a composite way where one of them may have more importance.

Even though the precision, recall and F-Measure (F1 and F2) metrics shows performance about the prediction of the labels that have positive value (1), the performance for predicting negative features (label is 0) is also important. For this purpose, a small FDR value, small FPR value and a good NPV value is desired in a model. As some issues arise when one of them is used individually in precision and recall, these metrics shows different aspects of the model and they should be used by combining the metrics introduced for positive value prediction. At this point, constructing ROC curves is a good example of that.

Overall, for evaluation of the model the measures for positive value prediction and negative value prediction depending on the aim of the classification there are cases where these metrics are needed to be used altogether.