SMAI Assignment 2

- Author: Vishal Reddy Mandadi
- Roll Number: 2019101119
- Date: 9th November 2021

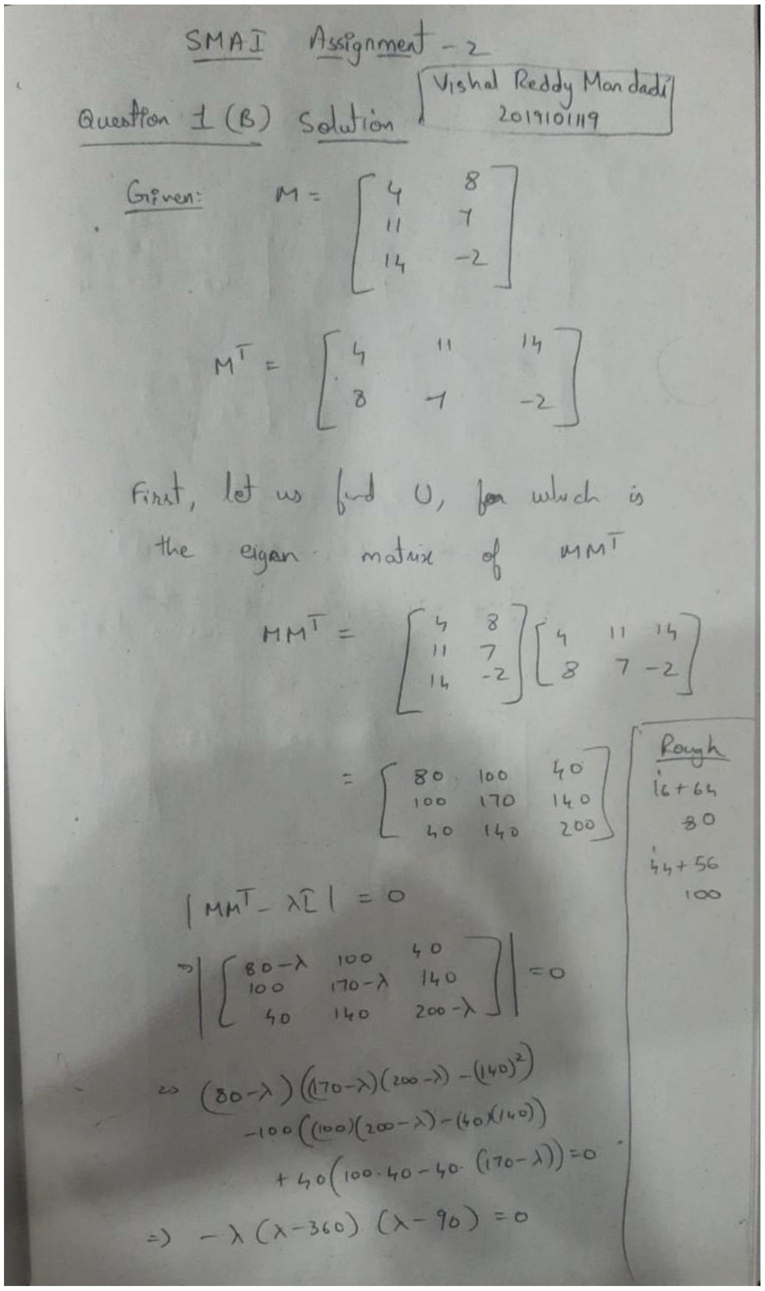
Question 1 Solutions:

A. Among Eigen Value Decomposition and Singular Value Decomposition, which one is more generalizable to matrices and why?

Answer: SVD is more generalizable to matrices because SVD exists for all matrices without any constraints, whereas eigenvalue decomposition exists only for square matrices and only when these square matrices have dimensions of eigenspace = the dimensions of the matrix.

B. Show the method and find the SVD of a given matrix

Answer:



SMAI Assignment - 2

Vishal Reddy Mandadi
2019101119

Question 1 (B) Solution

Given: $M = \begin{bmatrix} 4 & 8 \\ 11 & 7 \\ 14 & -2 \end{bmatrix}$

$$M^T = \begin{bmatrix} 4 & 11 & 14 \\ 8 & 7 & -2 \end{bmatrix}$$

First, let us find U, for which is the eigen matrix of MM^T

$$MM^T = \begin{bmatrix} 4 & 8 \\ 11 & 7 \\ 14 & -2 \end{bmatrix} \begin{bmatrix} 4 & 11 & 14 \\ 8 & 7 & -2 \end{bmatrix}$$

$$= \begin{bmatrix} 80 & 100 & 40 \\ 100 & 170 & 140 \\ 40 & 140 & 200 \end{bmatrix}$$

Rough

$16+64$
80

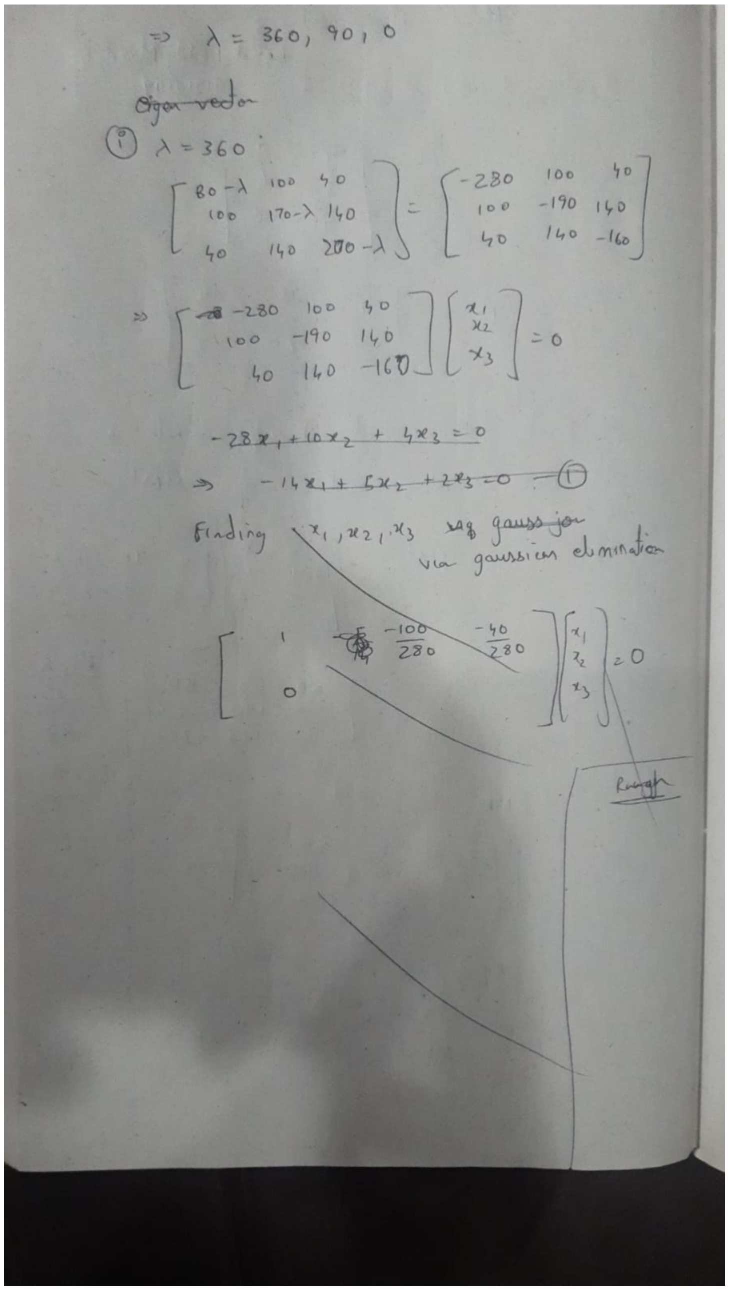
$44+56$
100

$$|MM^T - \lambda I| = 0$$

$$\Rightarrow \left| \begin{bmatrix} 80-\lambda & 100 & 40 \\ 100 & 170-\lambda & 140 \\ 40 & 140 & 200-\lambda \end{bmatrix} \right| = 0$$

$$\Rightarrow (80-\lambda)\left((170-\lambda)(200-\lambda) - (140)^2\right)$$
$$-100\left((100)(200-\lambda) - (40)(140)\right)$$
$$+40\left(100 \cdot 40 - 40 \cdot (170-\lambda)\right) = 0$$

$$\Rightarrow -\lambda(\lambda-360)(\lambda-90) = 0$$



$\Rightarrow \lambda = 360, 90, 0$

~~Eigen vector~~

(i) $\lambda = 360$

$$\begin{bmatrix} 80-\lambda & 100 & 40 \\ 100 & 170-\lambda & 140 \\ 40 & 140 & 200-\lambda \end{bmatrix} = \begin{bmatrix} -280 & 100 & 40 \\ 100 & -190 & 140 \\ 40 & 140 & -160 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} -280 & 100 & 40 \\ 100 & -190 & 140 \\ 40 & 140 & -160 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

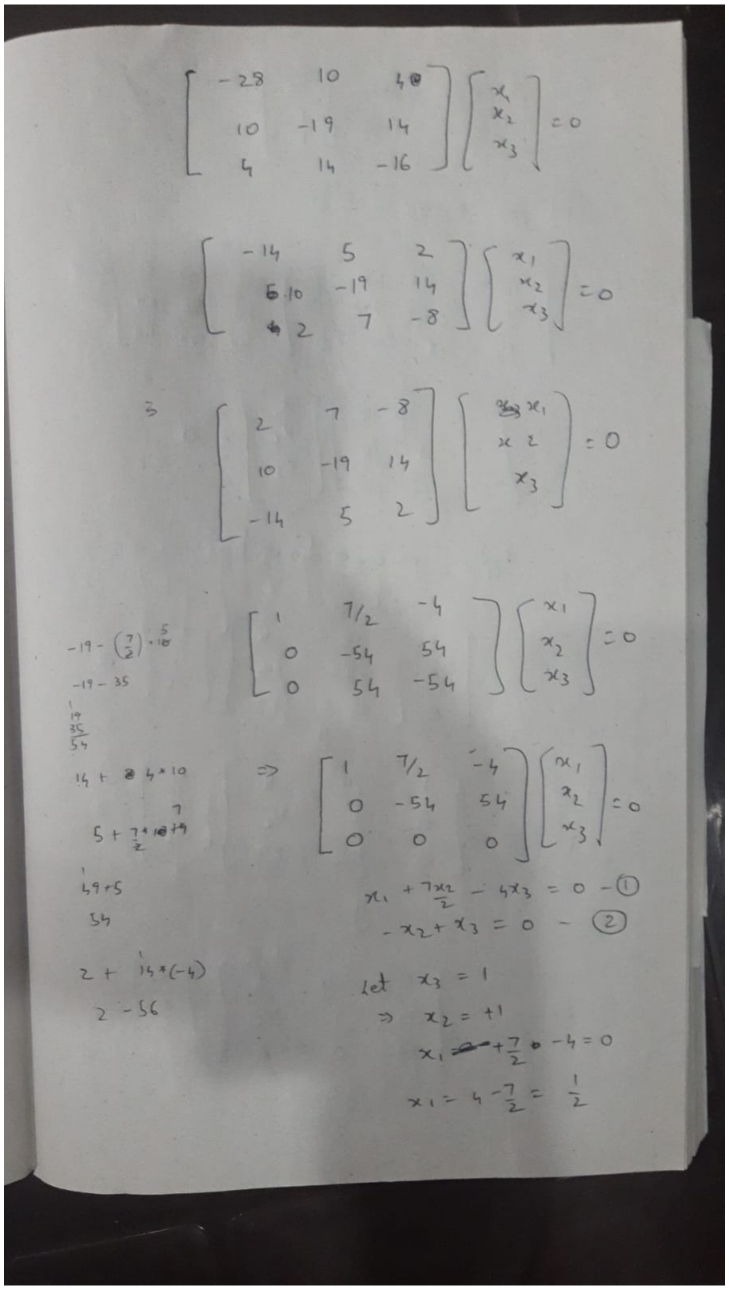
~~$-28x_1 + 10x_2 + 4x_3 = 0$~~

~~$\Rightarrow -14x_1 + 5x_2 + 2x_3 = 0$~~ — (1)

Finding x_1, x_2, x_3 ~~using gauss jor~~
via gaussian elimination

$$\begin{bmatrix} 1 & \frac{-100}{280} & \frac{-40}{280} \\ 0 & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

Rough



$$\begin{bmatrix} -28 & 10 & 4 \\ 10 & -19 & 14 \\ 4 & 14 & -16 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

$$\begin{bmatrix} -14 & 5 & 2 \\ 10 & -19 & 14 \\ 2 & 7 & -8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

3

$$\begin{bmatrix} 2 & 7 & -8 \\ 10 & -19 & 14 \\ -14 & 5 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

$-19 - \left(\frac{7}{2}\right) \cdot \overset{5}{\cancel{10}}$

$-19 - 35$

19
35
54

$$\begin{bmatrix} 1 & 7/2 & -4 \\ 0 & -54 & 54 \\ 0 & 54 & -54 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

$14 + \cancel{8}\ 4 * 10$

$5 + \frac{7}{2} * \overset{7}{\cancel{10}} + 14$

$49 + 5$

54

$2 + 14 * (-4)$

$2 - 56$

$$\Rightarrow \begin{bmatrix} 1 & 7/2 & -4 \\ 0 & -54 & 54 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

$x_1 + \frac{7x_2}{2} - 4x_3 = 0$ — ①

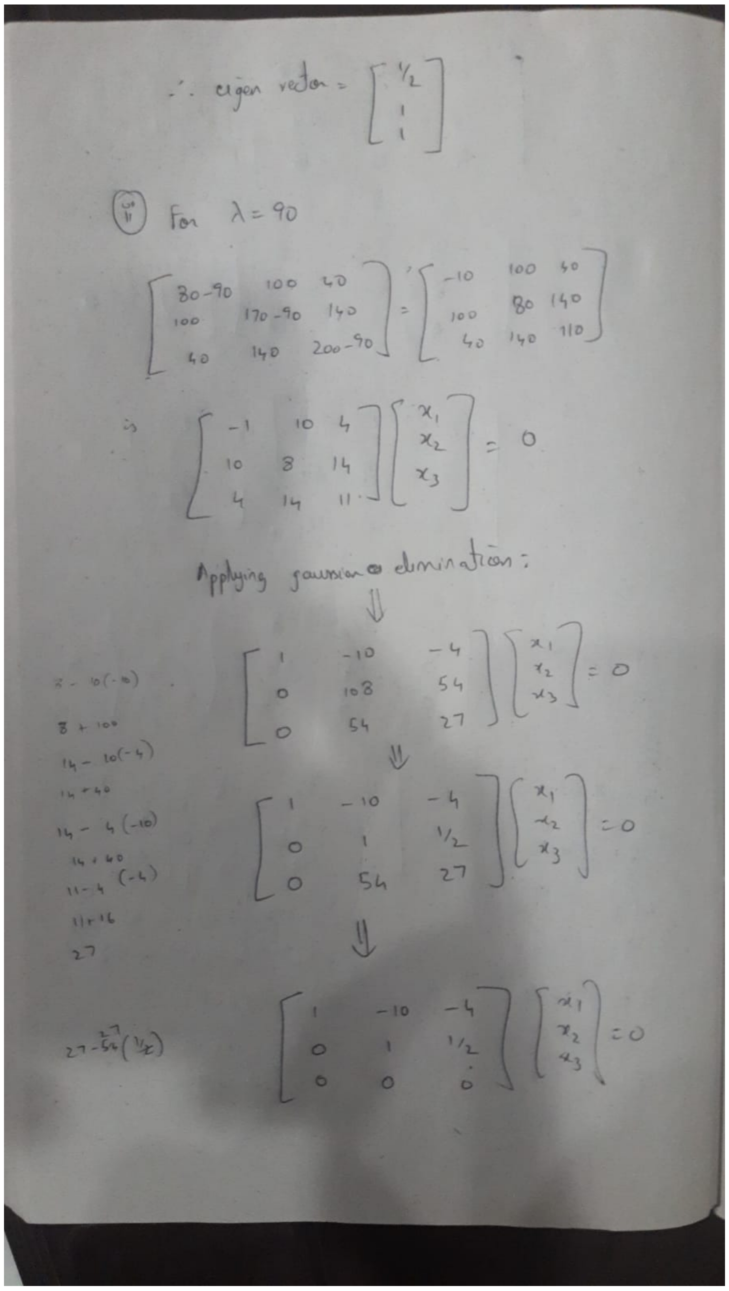
$-x_2 + x_3 = 0$ — ②

let $x_3 = 1$

$\Rightarrow x_2 = +1$

$x_1 + \frac{7}{2} - 4 = 0$

$x_1 = 4 - \frac{7}{2} = \frac{1}{2}$



$\therefore$ eigen vector = $\begin{bmatrix} 1/2 \\ 1 \\ 1 \end{bmatrix}$

(iii) For $\lambda = 90$

$$\begin{bmatrix} 80-90 & 100 & 40 \\ 100 & 170-90 & 140 \\ 40 & 140 & 200-90 \end{bmatrix} = \begin{bmatrix} -10 & 100 & 40 \\ 100 & 80 & 140 \\ 40 & 140 & 110 \end{bmatrix}$$

is $\begin{bmatrix} -1 & 10 & 4 \\ 10 & 8 & 14 \\ 4 & 14 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$

Applying gaussian elimination:

$$\Downarrow$$

$$\begin{bmatrix} 1 & -10 & -4 \\ 0 & 108 & 54 \\ 0 & 54 & 27 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

$8 - 10(-10)$

$8 + 100$

$14 - 10(-4)$

$14 + 40$

$14 - 4(-10)$

$14 + 40$

$11 - 4(-4)$

$11 + 16$

27

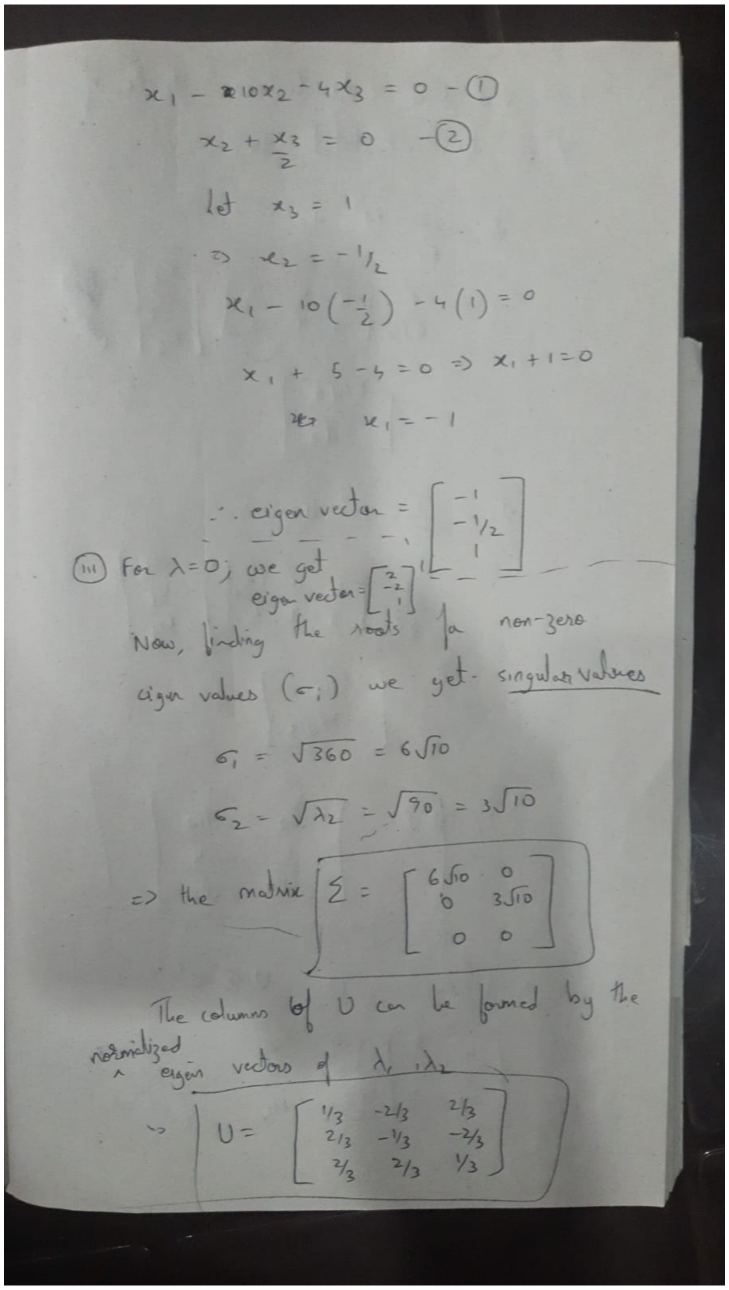
$$\Downarrow$$

$$\begin{bmatrix} 1 & -10 & -4 \\ 0 & 1 & 1/2 \\ 0 & 54 & 27 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

$$\Downarrow$$

$27 - \overset{27}{\cancel{54}}(1/2)$

$$\begin{bmatrix} 1 & -10 & -4 \\ 0 & 1 & 1/2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$



$x_1 - 10x_2 - 4x_3 = 0$ — ①

$x_2 + \frac{x_3}{2} = 0$ — ②

let $x_3 = 1$

$\Rightarrow x_2 = -1/2$

$x_1 - 10\left(-\frac{1}{2}\right) - 4(1) = 0$

$x_1 + 5 - 4 = 0 \Rightarrow x_1 + 1 = 0$

$\Rightarrow x_1 = -1$

$\therefore$ eigen vector $= \begin{bmatrix} -1 \\ -1/2 \\ 1 \end{bmatrix}$

(iii) For $\lambda = 0$; we get eigen vector $= \begin{bmatrix} 2 \\ -2 \\ 1 \end{bmatrix}$

Now, finding the roots for non-zero eigen values (σ_i) we get singular values

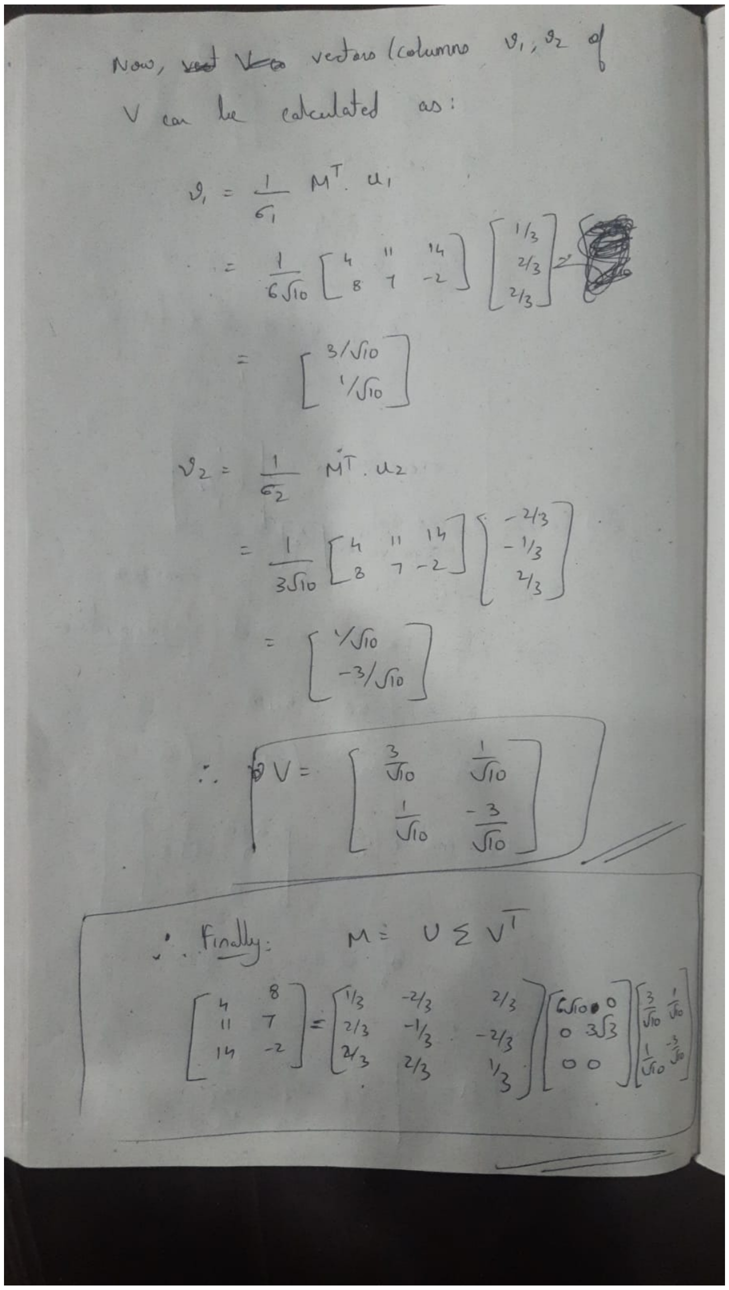
$\sigma_1 = \sqrt{360} = 6\sqrt{10}$

$\sigma_2 = \sqrt{\lambda_2} = \sqrt{90} = 3\sqrt{10}$

$\Rightarrow$ the matrix $\boxed{\Sigma = \begin{bmatrix} 6\sqrt{10} & 0 \\ 0 & 3\sqrt{10} \\ 0 & 0 \end{bmatrix}}$

The columns of U can be formed by the normalized eigen vectors of λ_1, λ_2

$\Rightarrow \boxed{U = \begin{bmatrix} 1/3 & -2/3 & 2/3 \\ 2/3 & -1/3 & -2/3 \\ 2/3 & 2/3 & 1/3 \end{bmatrix}}$



Now, ~~vect~~ ~~V~~ vectors (columns v_1, v_2 of V can be calculated as:

$$v_1 = \frac{1}{\sigma_1} M^T . u_1$$

$$= \frac{1}{6\sqrt{10}} \begin{bmatrix} 4 & 11 & 14 \\ 8 & 7 & -2 \end{bmatrix} \begin{bmatrix} 1/3 \\ 2/3 \\ 2/3 \end{bmatrix}$$

$$= \begin{bmatrix} 3/\sqrt{10} \\ 1/\sqrt{10} \end{bmatrix}$$

$$v_2 = \frac{1}{\sigma_2} M^T . u_2$$

$$= \frac{1}{3\sqrt{10}} \begin{bmatrix} 4 & 11 & 14 \\ 8 & 7 & -2 \end{bmatrix} \begin{bmatrix} -2/3 \\ -1/3 \\ 2/3 \end{bmatrix}$$

$$= \begin{bmatrix} 1/\sqrt{10} \\ -3/\sqrt{10} \end{bmatrix}$$

$$\therefore V = \begin{bmatrix} \frac{3}{\sqrt{10}} & \frac{1}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} & -\frac{3}{\sqrt{10}} \end{bmatrix}$$

$\therefore$ Finally: $M = U \Sigma V^T$

$$\begin{bmatrix} 4 & 8 \\ 11 & 7 \\ 14 & -2 \end{bmatrix} = \begin{bmatrix} 1/3 & -2/3 & 2/3 \\ 2/3 & -1/3 & -2/3 \\ 2/3 & 2/3 & 1/3 \end{bmatrix} \begin{bmatrix} 6\sqrt{10} & 0 \\ 0 & 3\sqrt{3} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{3}{\sqrt{10}} & \frac{1}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} & \frac{-3}{\sqrt{10}} \end{bmatrix}$$

Question 2 solutions

A. Solution:

Assumption: "All elements" implies all diagonal entries (because, if I take the literal meaning of "all elements", the answers will be fairly obvious - (b) and (c) which is not fun and may not be the real intent of the question maker. Hence, I take "all elements" as "all diagonal elements" for this question. Not that, either way, the answer will be the same)

Options (b), (c) are correct.

1. Option (a) is incorrect because when all elements are equal, implies all diagonal elements are equal, implies all eigenvalues are equal, implies the variance along all directions are equal. This implies that we will definitely lose crucial information while choosing which component to remove because all components have equal contributions to the spread of the data (equally important)
2. Option (b) is correct because PCA is useful only when elements of D are not equal, i.e. the eigenvalues are not equal. The eigenvalues capture the variance of the data along perpendicular directions. And PCA is used to reduce the dimensionality of data in such a way that maximum variance is retained during the reduction. If the variances are all equal, we won't be able to choose which principle component to eliminate, as all the components are equally important. In such cases, on random reduction, you might end up losing a lot of information. Hence, PCA works well only when eigenvalues are unequal.
3. Option (c) is correct because when all the points of X lie on a straight line (assuming n-dimensional data with n≥2), the first eigenvector will be along the line and hence its corresponding eigenvalue will be non-zero. The remaining eigenvalues will all be zero because the spread of data along any other direction perpendicular to the first eigenvector is 0. Implies, if the data is n-dimensional, D will have (n-1) rows will all zero values. Therefore, the rank(D) is less than n, i.e. not a full ranked matrix.
4. Option (d) is incorrect because, irrespective of the data, eigenvectors are always perpendicular to each other, this implies that the rows and columns of the matrix are linearly independent, which means that the matrix is full-rank.
5. Option (e) is incorrect because, when the data points lie in a circle, the spread of the data along the two eigenvectors will be equal and non-zero (assuming that the radius of the circle > 0), this implies that the eigenvalues will also be non-zero (as they represent the variance of the data). This implies that D will be a diagonal matrix with non-zero diagonal entries, therefore, D is full rank (as all the columns will be linearly independent)

B. Solution

Answer: False, PCA does not project onto the line that is most helpful for data classification. It just projects into a subspace so that the maximum amount of variance is still retained, which need not always be the favorable direction of projection for easier classification.

Question 3 Solutions

A. solution

Prior probability denoted simply as $P(x)$ defines the probability of an event x when no additional information is known about that event. It is the best
rational estimate of the probability of an outcome based on the current knowledge before an experiment is conducted. For example, what is the probability of two heads? The answer is 1/4 (as HH occurs only once in the set {HH, HT, TH, TT})

Posterior probability denoted as $P(x/w)$ denotes the (updated) probability of event x, when some additional information w is known that relates the event x. With the new information w, the odds and probability will change from the initial estimate $P(x)$ to a new estimate $P(x/w)$ which is revised based on the new information w. For example, what is the probability of two heads, given that the first coin toss resulted in Heads? The answer is 1/2, because, we already know that the first toss is head, this implies the resultant sample space will be {HH, HT}, so probability $P(HH/1\text{st toss is }H) = 1/2$

B. solution

Notion:

$$S = \text{sore throat}$$
$$H = \text{headache}$$
$$F = \text{flu}$$

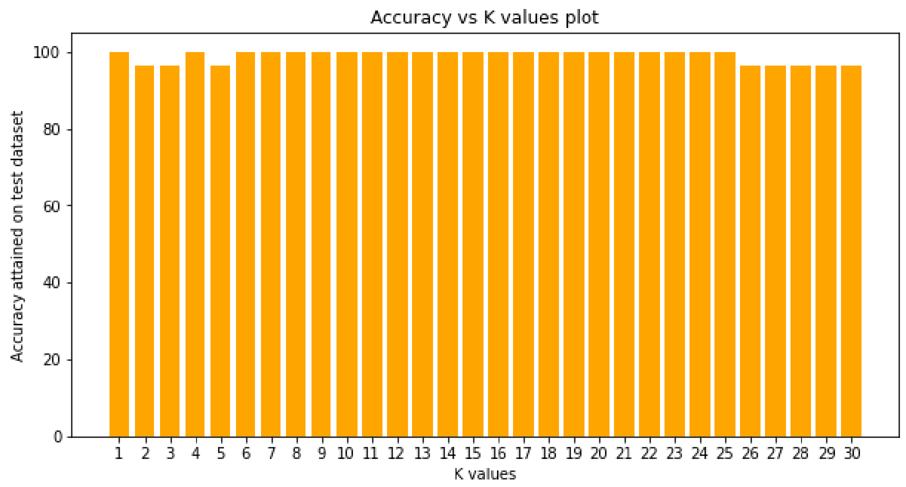
Solution:

$$\text{Given:}$$
$$P(SH/F) = 0.9$$
$$\text{Probability of having flu (in a year)} = P(F) = 0.05$$
$$\text{Probability of having H and S} = P(HS) = 0.2$$
$$\text{To find: Probability of having a flu given H and S} = P(F/HS)$$
$$P(F/HS) = \frac{P(FHS)}{P(HS)}$$
$$= \frac{P(HS/F) * P(F)}{P(HS)}$$
$$= \frac{0.9 * 0.05}{0.2} = 0.225$$
$$\therefore P(F/HS) = 0.225$$

Final answer: $P(F/HS) = 0.225$.

Question 4 solution (Code can be found in the Jupyter notebook as well)

1. For most of the values of K, the algorithm returned 100% accuracy. Following is the bar plot on **accuracy vs k-value** plot

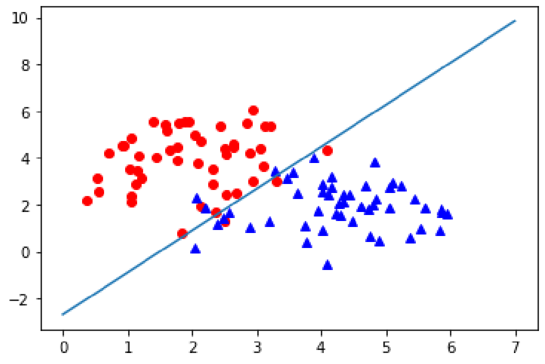


Code:

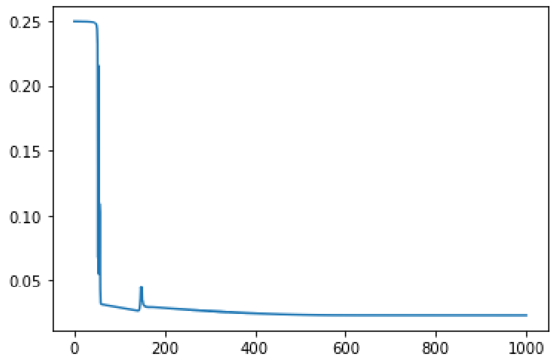
```
class KNN:
    def __init__(self, X, y, num_of_neighbours=10, cost_metric='Euclidean', choice_metric='max'):
        '''
        Input: X: (no_of_samples, no_of_features) (np.ndarray of float) [standard] (training data features)
               y: (no_of_samples, 1) (np.ndarray of Strings) [standard] (training data ground truth labels (strings))
        '''
        self.k = num_of_neighbours
        self.cost_metric = cost_metric
        self.choice_metric = choice_metric
        self.X_train = X
        self.y_train = y
        self.y_encoded_train, self.encoding_dict = self.encode(y)

    def encode(self, y):
        '''
        Input: y: (no_of_samples, 1) (np.ndarray of strings)
        Encodes each string to a particular number and returns the encoded y and the encoding dictionary
        Return: y_encoded, encodings_dictionary
            1. y_encoded (no_of_samples, 1) (np.ndarray of integers)
            2. encodings_dictionary structure:
                {
                    string_encoding: int_key
                }
                That is, let us say we have the following car classes as input ['ford', 'bugati', 'merc']
                then encoding_dict will be:
                encoding_dict: {
                    'ford': 1,
                    'bugati': 2,
                    'merc': 3
                }
                This method of helps in faster encoding. This would be slightly slower at output prediction
        '''
        labels = np.unique(y)
        encoding_dict = {}
        for i, label in enumerate(labels):
            encoding_dict[label] = int(i+1)
        y_encoded = []
        for label in y:
            y_encoded.append(int(encoding_dict[label]))
        print("Encodings dictionary: \n{}".format(encoding_dict))
        return np.array(y_encoded), encoding_dict

    def euclidean_dist(self, x1, x2):
        '''
        Calculates the euclidean distance between x1 and x2 (np.sum((X-Y)**2))
        Input: x1 (3, 1) or (1, 3) vector
               x2 (3, 1) or (1, 3) vector
        Output: float (distance)
        '''
        x1_f = x1.flatten()
        x2_f = x2.flatten()
        if x1.shape != x2.shape:
            print("Error: Mismatched shapes in euclidean distance function: {}, {}".format(x1.shape, x2.shape))
            exit()
        dist = (x1_f-x2_f)**2
        dist = dist.flatten()
        dist = np.sqrt(np.sum(dist))
        return dist

    def get_nearest_neighbours(self, X):
        '''
        Input: X: (1, no_of_features) (np.ndarray) [standard]
        Return: K nearest neighbours based on cost_metric (np.ndarray) (distances, labels)
            1. If cost_metric=='Euclidean': then returns asending order based on distance values
        '''
        x = X.flatten()
        nearest_neighbours = []
        if self.cost_metric=='Euclidean':
            neighbours = []
            for i, point in enumerate(self.X_train):
                dist = self.euclidean_dist(X, point)
                label = self.y_encoded_train[i]
                label_str = self.y_train[i]
                neighbours.append([dist, label_str])
            neighbours.sort(key = lambda tup: tup[0])
            # print("Sorted first neighbours: {}".format(neighbours[0:10])) # Debugging statement
            nearest_neighbours = neighbours[0:self.k]

        return np.array(nearest_neighbours)
    def predict(self, X):
        '''
        Input: X: (no_of_samples, no_of_features)
        Output: Predicted labels (strings) for each sample - (no_of_samples, ) (1D np array)
        '''
        output = []
        for row in X:
            nearest_neighbours = self.get_nearest_neighbours(row)[:, 1] # Only labels
            # print(nearest_neighbours)
            label = max(set(nearest_neighbours), key=list(nearest_neighbours).count)
            output.append(label)
        return np.array(output)

    def test(self, X, y):
        '''
        Input: X: (no_of_samples, no_of_features) - data, y: (no_of_samples) - labels (np arrays)
        Output: dictionary
            result = {
                'counts': {
                    'True': no of correct predictions,
                    'False': no of incorrect predictions
                },
                'accuracy': accuracy in %
            }
        '''
        y_pred = self.predict(X)

        results = []
        for i in range(len(y_pred)):
            if y_pred[i]==y[i]:
                results.append(True)
            else:
                results.append(False)

        # print("y_pred.shape: {}".format(y_pred.shape))
        # results = [y_pred==y]
        print("results shape: {}".format(len(results)))
        unique, counts = np.unique(results, return_counts=True)
        # for i, y_prediction in enumerate(y_pred):
        #     print(y_prediction, y[i])
        print(unique, counts)
        result = {}
        result['counts'] = {}
        for i, label in enumerate(unique):
            result['counts'][str(label)] = counts[i]
        print("Results: {}".format(results))
        # print("No of true(correct) and false(wrong) predictions: {}".format(result['counts']))
        # print(len(y_pred))
        # print(len(y))
        # print(y_pred.shape)
        # print(y.shape)
        # print([y_pred==y])
        accuracy = (result['counts']['True']*100.0)/(len(y_pred))
        result['accuracy'] = accuracy
        return result


# Train KNNs with various values of K and plot of bar graph of k vs accuracy
results_dict = {}
for i in range(1, 31):
    k = i
    knn_model = KNN(X_train, y_train, num_of_neighbours=k)
    result = knn_model.test(X_test, y_test)
    print("Result: -------------------------\n{}".format(result))
    results_dict['{}'.format(k)] = result['accuracy']

k_values = list(results_dict.keys())
accuracy = list(results_dict.values())

fig = plt.figure(figsize = (10, 5))
# creating the bar plot
plt.bar(k_values, accuracy, color ='orange',
        width = 0.8)

plt.xlabel("K values")
plt.ylabel("Accuracy attained on test dataset")
plt.title("Accuracy vs K values plot")
plt.show()
# print("Test value: {}, {}".format(X_test[0], y_test[0]))
# print("Nearest neighbours in sorted order: {}".format(knn_model.get_nearest_neighbours(X_test[0])))
```

Question 5 solution (Code can be found in the Jupyter notebook as well)

1. Obtained Decision boundary graph:



2. Iterations vs training loss plot



3. Final mean training loss after 1000th iteration: 0.022882108300966087

Code for Logistic regression

```
'''Logistic regression:
Input Data: X - 2D Data - [x1, x2]
    Feature vector dimensions: X_features - 3D - [1, x1, x2]
Weight vector/matrix: w - 3D
    w = [w0, w1, w2]
Hypothesis function: Logistic/Sigmoid function
    h_w(x) = 1/(1+exp(-w.T @ x))
        h_w(x) >= 0.5, then y_pred = 1
        h_w(x) < 0.5,  then y_pred = 0
Loss function:
    1. Negitive log likelihood function
        L(w) = -y*log( h_w(x) ) - (1-y)*log( 1 - h_w(x) )
    2. Mean Squared error:
        L(w) = (1/2) * (h_w(x) - y)**2
        To be minimized
Optimizer: Gradient Descent Algorithm
    w(i+1) = w(i) - (learning_rate)*gradient_w(h_w(x))
Partial derivatives:
    w.r.t dw0: (h_w(x) - y) * h_w(x)*(1-h_w(x))
    w.r.t dw1: (h_w(x) - y) * x1 * h_w(x)*(1-h_w(x))
    w.r.t dw0: (h_w(x) - y) * x2 * h_w(x)*(1-h_w(x))
    i.e. dw = (h_w(x) - y) * x * h_w(x)*(1-h_w(x))
Decision Boundary:
    w.T @ x = 0
'''
class LogisticRegression:
    def __init__(self, no_of_features_without_1, learning_rate):
        '''
        No of features = 2 for 2D data. Do not count 1 as a feature while sending it as
        an arguement
        '''
        self.no_of_features_without_1 = no_of_features_without_1
        self.no_of_features = no_of_features_without_1 + 1
        # Random non-zero initialization of weight matrix
        self.w = np.random.uniform(low=1.0, high=2.0, size=no_of_features_without_1+1)
        self.w = self.w.T[:, np.newaxis] # w must be a column vector
        self.learning_rate = learning_rate # learning rate

    def convert_data_to_feature_vector(self, x):
        '''
        Given x (no_of_features * no_of_samples), convert it into - (no_of_features+1)*(no_of_samples)
        i.e. x_old = [[x1],
                      [x2]]
        then, x_new = [[1],
                       [x1],
                       [x2]]
        '''
        if len(x) == self.no_of_features:
            return x
        elif len(x) == self.no_of_features - 1:
            return np.vstack((np.ones(shape=(1, x.shape[1])), x))
        else:
            print("Dimensions of X mismatched, please check!!! Each column is a sample, hence rows should contain features")
            exit()

    def h(self, x):
        '''
        Hypothesis function: h_w(x) = 1/(1+exp(-w.T@x))
        x = no_of_features * 1 vector (or) no_of_features * n matrix (where n = no of samples)
        x - np.ndarray
        Output:
        '''
        if len(x) < self.no_of_features:
            x = np.vstack((np.ones(shape=(1, x.shape[1])), x))
        # print("Shapes of W and x: {} and {}".format(self.w.shape, x.shape))
        y_pred = 1/(1+np.exp(-1 * self.w.T @ x))
        y_pred = (y_pred.flatten())[np.newaxis, :]
        # print("Shape of h(x) is {}".format(y_pred.shape))
        return y_pred

    def entropy_cost_from_preds(self, y_predicted, y):
        '''
        Metadata
        y_predicted: Predicted values
        y: Ground truth
        Input: y_predicted - (int) or (1D np.array), y - (int) or (1D np.array)
        Output: Mean total cost - (int)
        '''
        cost = -1  *  (y*(np.log(y_predicted)) + (1-y)*np.log(1-y_predicted))
        cost = cost.flatten() # np.flatten(cost)
        return (1/self.no_of_features) * np.sum(cost)

    def entropy_cost_from_x(self, x, y):
        if len(x) == self.no_of_features_without_1:
            x = np.vstack((np.ones(shape=(1, x.shape[1])), x))
        y_predicted = self.h(x)
        return self.entropy_cost_from_preds(y_predicted, y)

    def l2_cost_from_preds(self, y_predicted, y):
        '''
        Returns half of mean squared error
        error = (1/2)* (norm(y - h_w(x))**2)
        '''
        y1 = y.flatten() # np.flatten(y)
        y2 = y_predicted.flatten() # np.flatten(y_predicted)
        cost = np.mean(np.sum((y2 - y1)**2)) / 2
        return cost

    def l2_cost_from_x(self, x, y):
        if len(x) == self.no_of_features_without_1:
            x = np.vstack((np.ones(shape=(1, x.shape[1])), x))
        y_predicted = self.h(x)
        return self.l2_cost_from_preds(y_predicted, y)


    def l2_gradient(self, x, y):
        '''
        Dimensions of x: no_of_features * no_of_samples
        Calculates the gradient for l2_cost
        '''
        y_pred = self.h(x)
        no_of_samples = len(y_pred)
        print("No of samples: {}".format(y_pred.shape))
        dw = (np.sum(2 * (y_pred - y) * self.h(x) * (1-y_pred) * x, axis=1, keepdims=True))/no_of_samples
        # print("Gradient: {}".format(dw))
        return dw

    def gradient_descent(self, x, y):
        '''
        Gradient descent algorithm on l2 loss using l2_gradient
        w = w - learning_rate*(l2_gradient(y_pred, y))
        Vectorized! Zoom Zoom!
        '''
        # Update weights
        self.w = self.w - self.learning_rate*self.l2_gradient(x, y)

    def train(self, x, y, iterations):
        '''
        Numpy arrays - x and y
        Input: x: (no_of_features * no_of_samples)
               y: (1*no_of_samples)
               iterations: (int)
        '''
        if len(x) == self.no_of_features_without_1:
            x_modified = self.convert_data_to_feature_vector(x)
        # Check shapes of x and y
        if len(x) != self.no_of_features:
            print("Error, shape of x ({}) is invalid!!".format(x.shape))
            exit()
        if y.shape[0] != 1:
            print("Error, shape of y ({}) is invalid".format(y.shape))
        if x.shape[1] != y.shape[1]:
            print("Error, x ({}) and y ({}) don't have same no of columns (no of samples)".format(x.shape, y.shape))
        collected_costs = []
        # Start the training loop
        for i in range(iterations):
            # 1. Predict y
            y_pred = self.h(x_modified)
            # 2. Calculate mean squared error loss
            cost = self.l2_cost_from_preds(y_pred, y)
            collected_costs.append(cost)
            print("\rIteration: {}, Cost: {}".format(i, cost))
            # 3. Perform gradient descent i.e. Update weights
            self.gradient_descent(x_modified, y)

        # Final cost
        collected_costs.append(self.l2_cost_from_x(x_modified, y))
        # Finally plot the iteration vs cost graph
        fig = plt.figure()
        plt.plot(np.arange(iterations+1), collected_costs)
        plt.show()

    def test(self, x, y):
        '''
        Input: x: (no_of_features, no_of_samples) (or) (no_of_features_without_1, no_of_samples)
               y: (1, no_of_samples)
        '''
        x_modified = 1 * x
        if len(x) == self.no_of_features_without_1:
            x_modified = self.convert_data_to_feature_vector(x)
        # Check shapes of x and y
        if len(x_modified) != self.no_of_features:
            print("Error, shape of x ({}) is invalid!!".format(x_modified.shape))
            exit()
        if y.shape[0] != 1:
            print("Error, shape of y ({}) is invalid".format(y.shape))
        if x_modified.shape[1] != y.shape[1]:
            print("Error, x ({}) and y ({}) don't have same no of columns (no of samples)".format(x_modified.shape, y.shape))

        y_pred = self.h(x)
        # Average mean squared loss
        cost = self.l2_cost_from_preds(y_pred, y)
        print("Average_cost: {}".format(cost))
        # Accuracy and F1 score: TBD
        return cost

    def plot_decision_boundary(self, X, Y):
        # Creating vectors X and Y representing the line
        if self.w[2] != 0:
            x = np.linspace(0, 7, 100)
            y = -1 * (self.w[0] + self.w[1]*x) / (self.w[2])
        elif self.w[1] != 0:
            y = np.linspace(0, 7, 100)
            x = -1 * (self.w[0] + self.w[2]*y) / (self.w[1])

        fig = plt.figure()
        # Create the plot
        plt.plot(X[:,0][Y==0],X[:,1][Y==0],'o',color='red') # data from class 0
        plt.plot(X[:,0][Y==1],X[:,1][Y==1],'^',color='blue') # data from class 1
        plt.plot(x, y) # decision boundary

        # Show the plot
        plt.show()

# Data has been already generated in the previous cell - please run all the cells
# X - input data - shape - (no_of_samples, no_of_features_without_1)
# y - labels (ground truth) - (no_of_samples, 1)

# Reshape X and y to
# X: (no_of_features_without_1, no_of_samples)
# y: (1, no_of_samples)
X_modified = X.T
y_modified = y.T
y_modified = y[np.newaxis, :]

no_of_features_without_1 = X_modified.shape[0]
learning_rate = 0.06
log_reg = LogisticRegression(no_of_features_without_1, learning_rate)

iterations = 1000
log_reg.train(x=X_modified, y=y_modified, iterations=iterations)

log_reg.test(X_modified, y_modified)

log_reg.plot_decision_boundary(X, y)
```

END OF THE REPORT (THANK YOU!)