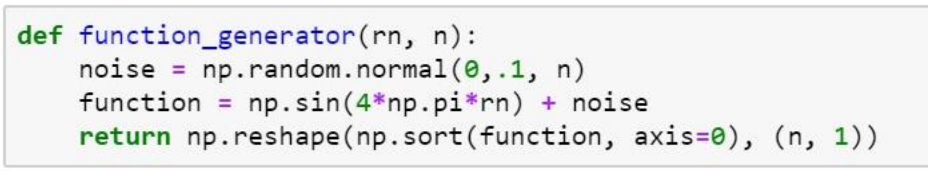
Foundations of Machine Learning Report

Suyash Datt Dubey   Student ID: sdd1n17

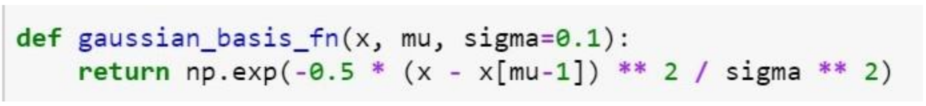
Linear Regression with non-linear functions



```
def function_generator(rn, n):
    noise = np.random.normal(0,.1, n)
    function = np.sin(4*np.pi*rn) + noise
    return np.reshape(np.sort(function, axis=0), (n, 1))
```

The first step in linear regression involves fitting the function. In this case, the function is a polynomial function sin(4πx) + ϵ where ϵ is the noise term. The code for generating the N input points was already provided in the specifications. For creating the set of target points y, I created the function above by the name of function_generator. It takes an array of points and the number of points as input and returns a target vector that has the same shape as the input vector. The noise added is randomly selected between 0 and 0.1 for each point, ensuring that a unique noise is added to each point.

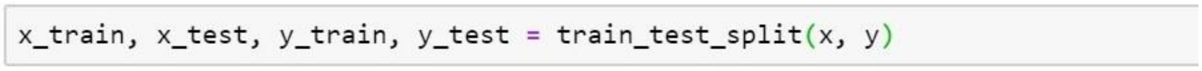
Performing linear regression with polynomials and radial basis functions



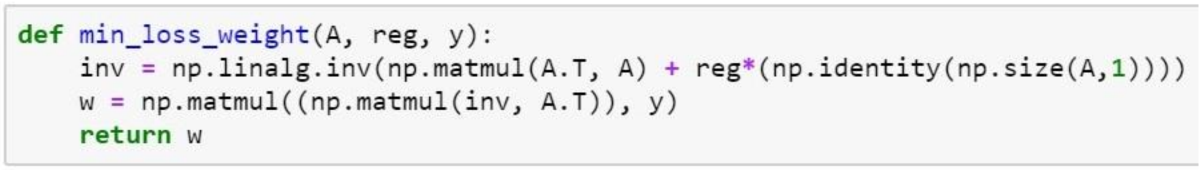
```
def gaussian_basis_fn(x, mu, sigma=0.1):
    return np.exp(-0.5 * (x - x[mu-1]) ** 2 / sigma ** 2)
```

The function for the polynomial and Gaussian basis functions, and for creating the design matrix was provided to us. However, I modified the gaussian_basis_fn a bit to ensure that it takes the appropriate value of mu for each x_n. The value of mu for each input x_n is selected using the formula in the specifications, i.e., by taking the j^{th} element from the set of points x, where j = 1, …, p.

How does regression generalise? Computing bias and variance

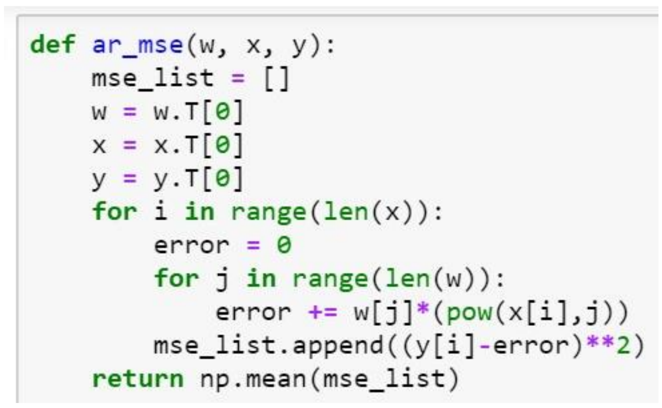


```
x_train, x_test, y_train, y_test = train_test_split(x, y)
```



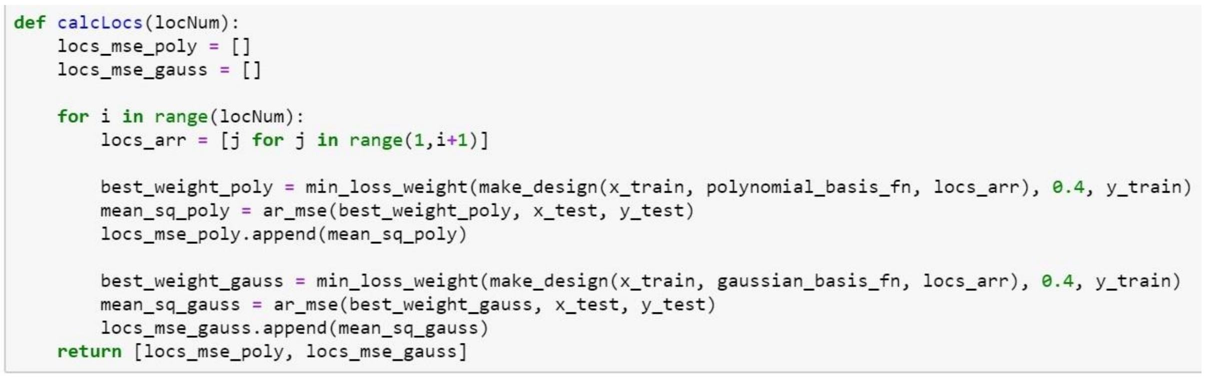
```
def min_loss_weight(A, reg, y):
    inv = np.linalg.inv(np.matmul(A.T, A) + reg*(np.identity(np.size(A,1))))
    w = np.matmul((np.matmul(inv, A.T)), y)
    return w
```

I split the input x and target y into the training and test sets using train_test_split from sklearn.model_selection. I have created a function, min_loss_weight which can be seen above. It takes as parameters a design matrix, a regularisation coefficient (λ), and a target vector, and calculates and returns the weight that minimises the loss function, i.e., best weight w according to the formula in the specifications.



```
def ar_mse(w, x, y):
    mse_list = []
    w = w.T[0]
    x = x.T[0]
    y = y.T[0]
    for i in range(len(x)):
        error = 0
        for j in range(len(w)):
            error += w[j]*(pow(x[i],j))
        mse_list.append((y[i]-error)**2)
    return np.mean(mse_list)
```

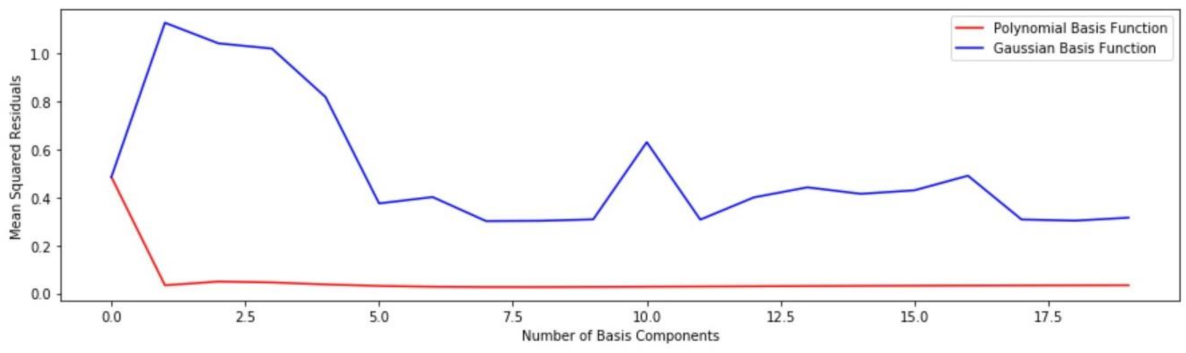
To calculate the mean squared error, I created a function called ar_mse which takes a weight, an input vector and a target vector and returns the mean squared error. The function loops over all the input elements, calculates the error for each and adds the error to a list. Once the error for each input element has been calculated the mean of the list of errors is returned, which is the mean squared error.



```
def calcLocs(locNum):
    locs_mse_poly = []
    locs_mse_gauss = []

    for i in range(locNum):
        locs_arr = [j for j in range(1,i+1)]

        best_weight_poly = min_loss_weight(make_design(x_train, polynomial_basis_fn, locs_arr), 0.4, y_train)
        mean_sq_poly = ar_mse(best_weight_poly, x_test, y_test)
        locs_mse_poly.append(mean_sq_poly)

        best_weight_gauss = min_loss_weight(make_design(x_train, gaussian_basis_fn, locs_arr), 0.4, y_train)
        mean_sq_gauss = ar_mse(best_weight_gauss, x_test, y_test)
        locs_mse_gauss.append(mean_sq_gauss)
    return [locs_mse_poly, locs_mse_gauss]
```

For plotting the graph of mean squared errors and the number of basis components, I created a function calcLocs to obtain the mean squared errors for a range of p values. The function takes the number of locs, i.e., p as a parameter locNum and returns two lists of mean squared errors, the first in which the design matrix is created using the polynomial basis function and the second in which the design matrix is created using the Gaussian basis function. The function loops p (locNum) times in order to calculate the mean squared error for each different value of p. The list of locs used in the design matrix is generated in each iteration and stored in a variable locs_arr. The best weight is calculated using the min_loss_weight. The design matrix is created with the training dataset and using the polynomial basis function. Then, the mean square error of the test dataset is calculated for the best weight obtained from the training dataset and stored in a list. The regularisation coefficient used is the same in both cases. The same steps are then repeated with the design matrix created using the Gaussian basis function.

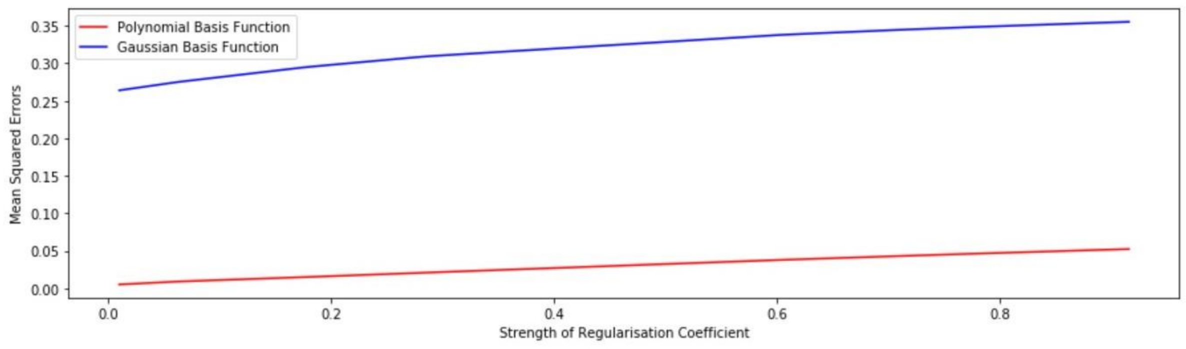
Once I obtained the two lists of mean squared errors, I plotted both with respect to the number of basis components. I obtained the graph below.



As can be seen from the graph, at p = 0, the mean squared error for both cases where the design matrix is created using the polynomial basis function and the Gaussian basis function is the same. p = 0 is the peak in case of the polynomial basis function. As the value of p increases, the mean squared error in case of the polynomial basis function decreases and becomes almost constant. This implies that the function is a reasonably good fitting model.

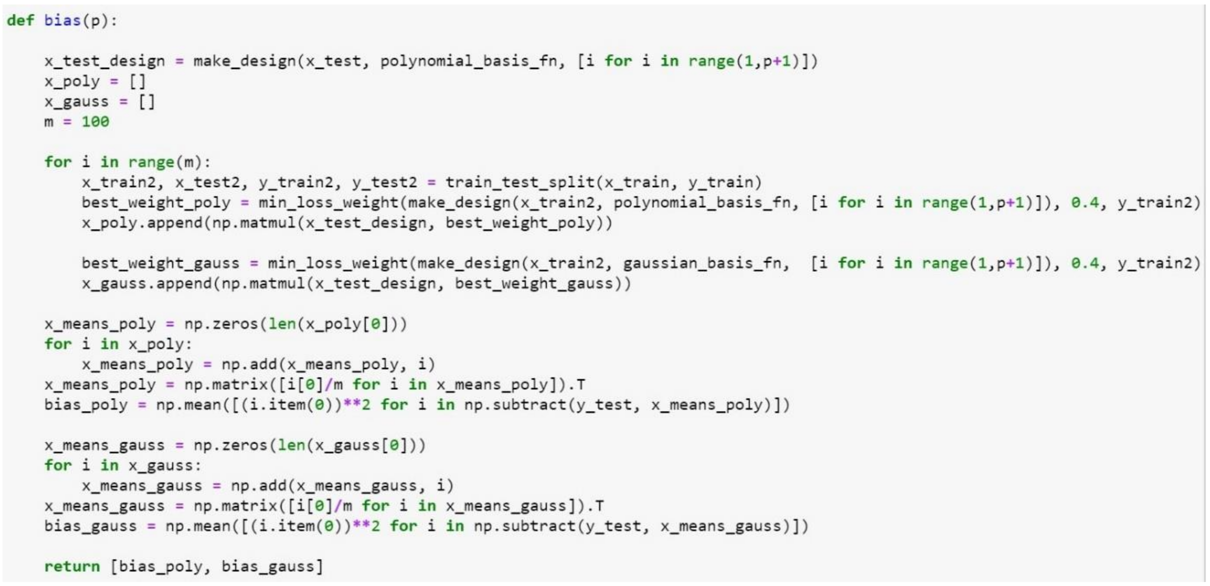
In the case of the Gaussian basis function, the graph is more irregular. It first increases and peaks at p = 1, it then decreases till p = 5 after which there is again a slight increase. There is another sharp increase at p = 10 after which there is a little increase around p = 13 and p = 16 after which it again decreases and seems to become a little steadier. It can be seen from the graph that the mean squared errors in case of the Gaussian basis function are quite high compared to the polynomial basis function. This implies that the Gaussian basis function leads to an underfitting model since it has poor performance on the test data.

I plotted the graph of the mean squared errors and the strength of the regularisation coefficient by calculating the mean squared error for a range of different regularisation coefficients. I used a list of 15 randomly generated regularisation coefficients in the range of 0 to 1 to plot the graph.



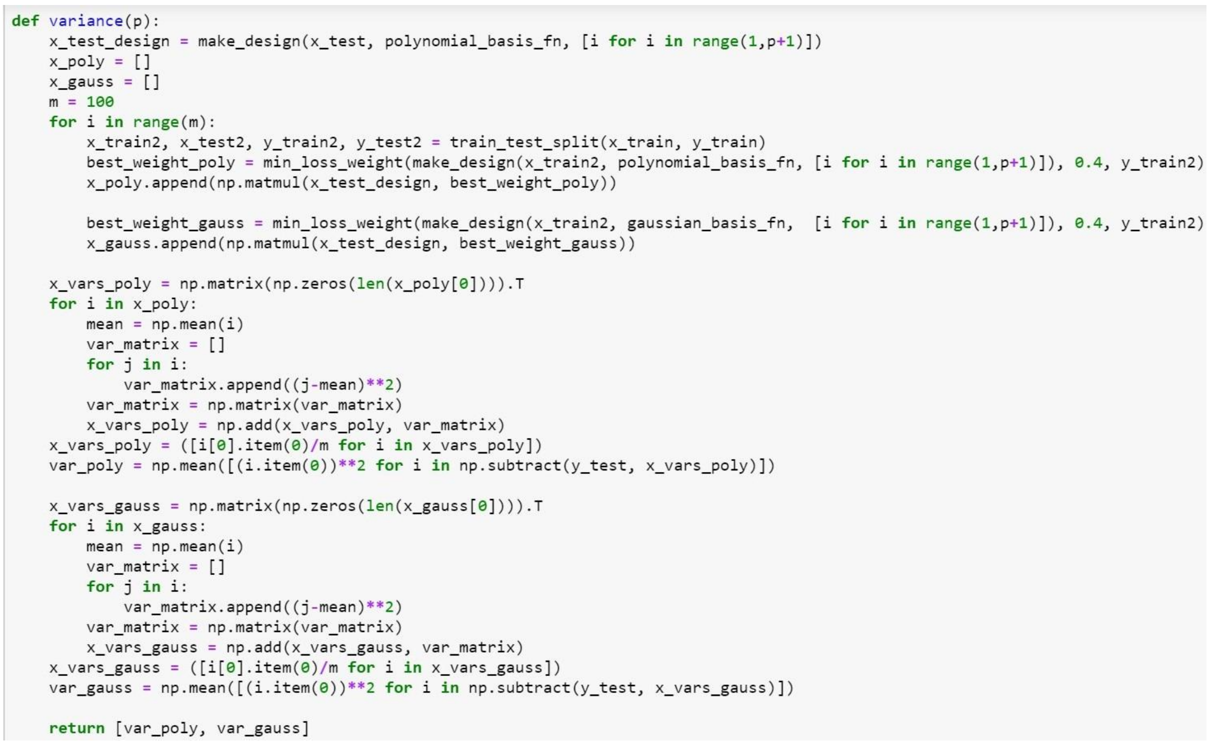
As can be seen in the graph, in the case where the design matrix is generated using the polynomial basis function, the errors, in general, are low, and they increase linearly with increasing regularisation coefficient. However, in case of the Gaussian basis function, the errors are much higher and grow in a much more non-linear manner. Although, as the regularisation coefficients increase, the errors steadily become more linear. Similar to the previous graph, this graph also implies that the polynomial basis function is a reasonably good fitting model and that the Gaussian basis function is underfitting.

I wrote two functions that evaluate the bias and variance for a range of p values and return the bias and variance for design matrices created using both the polynomial and Gaussian basis functions.



```python
def bias(p):

    x_test_design = make_design(x_test, polynomial_basis_fn, [i for i in range(1,p+1)])
    x_poly = []
    x_gauss = []
    m = 100

    for i in range(m):
        x_train2, x_test2, y_train2, y_test2 = train_test_split(x_train, y_train)
        best_weight_poly = min_loss_weight(make_design(x_train2, polynomial_basis_fn, [i for i in range(1,p+1)]), 0.4, y_train2)
        x_poly.append(np.matmul(x_test_design, best_weight_poly))

        best_weight_gauss = min_loss_weight(make_design(x_train2, gaussian_basis_fn,  [i for i in range(1,p+1)]), 0.4, y_train2)
        x_gauss.append(np.matmul(x_test_design, best_weight_gauss))

    x_means_poly = np.zeros(len(x_poly[0]))
    for i in x_poly:
        x_means_poly = np.add(x_means_poly, i)
    x_means_poly = np.matrix([i[0]/m for i in x_means_poly]).T
    bias_poly = np.mean([(i.item(0))**2 for i in np.subtract(y_test, x_means_poly)])

    x_means_gauss = np.zeros(len(x_gauss[0]))
    for i in x_gauss:
        x_means_gauss = np.add(x_means_gauss, i)
    x_means_gauss = np.matrix([i[0]/m for i in x_means_gauss]).T
    bias_gauss = np.mean([(i.item(0))**2 for i in np.subtract(y_test, x_means_gauss)])

    return [bias_poly, bias_gauss]
```

The bias function takes a value of p as a parameter and returns two bias values, one for which the design matrix is created using the polynomial basis function, and for the other the design matrix is created using the Gaussian basis function.



```python
def variance(p):
    x_test_design = make_design(x_test, polynomial_basis_fn, [i for i in range(1,p+1)])
    x_poly = []
    x_gauss = []
    m = 100
    for i in range(m):
        x_train2, x_test2, y_train2, y_test2 = train_test_split(x_train, y_train)
        best_weight_poly = min_loss_weight(make_design(x_train2, polynomial_basis_fn, [i for i in range(1,p+1)]), 0.4, y_train2)
        x_poly.append(np.matmul(x_test_design, best_weight_poly))

        best_weight_gauss = min_loss_weight(make_design(x_train2, gaussian_basis_fn,  [i for i in range(1,p+1)]), 0.4, y_train2)
        x_gauss.append(np.matmul(x_test_design, best_weight_gauss))

    x_vars_poly = np.matrix(np.zeros(len(x_poly[0]))).T
    for i in x_poly:
        mean = np.mean(i)
        var_matrix = []
        for j in i:
            var_matrix.append((j-mean)**2)
        var_matrix = np.matrix(var_matrix)
        x_vars_poly = np.add(x_vars_poly, var_matrix)
    x_vars_poly = ([i[0].item(0)/m for i in x_vars_poly])
    var_poly = np.mean([(i.item(0))**2 for i in np.subtract(y_test, x_vars_poly)])

    x_vars_gauss = np.matrix(np.zeros(len(x_gauss[0]))).T
    for i in x_gauss:
        mean = np.mean(i)
        var_matrix = []
        for j in i:
            var_matrix.append((j-mean)**2)
        var_matrix = np.matrix(var_matrix)
        x_vars_gauss = np.add(x_vars_gauss, var_matrix)
    x_vars_gauss = ([i[0].item(0)/m for i in x_vars_gauss])
    var_gauss = np.mean([(i.item(0))**2 for i in np.subtract(y_test, x_vars_gauss)])

    return [var_poly, var_gauss]
```

The variance function works similarly to the bias function but returns two variance values.

The values I obtained for the bias and variance for different values of p are summarised in the table below.

p	Polynomial Bias	Gaussian Bias	Polynomial Variance	Gaussian Variance
1	0.02602428316217	0.60080332573728	0.79950444571204	0.48860055082765
3	0.03316111718689	0.79233274945372	1.59554109341448	0.65527730572205
6	0.03509561441980	0.62580910344198	1.83307040058548	0.56730643495969
7	0.03364050162022	0.83321035397298	1.82883811872603	0.51607194290435
8	0.03379549757079	0.79013852999455	1.87262905747467	0.50760277502692
10	0.03427738653864	0.73021827768320	1.80785291516591	0.46966654310610

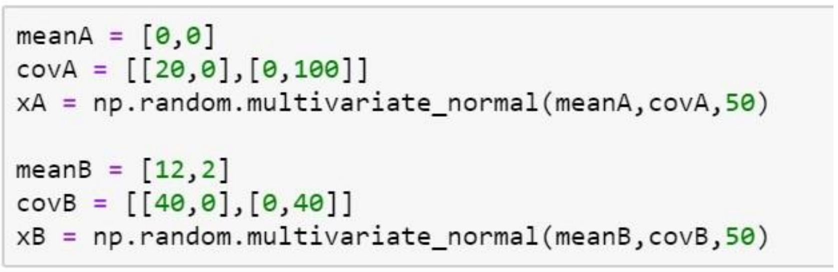
In the case of the polynomial basis function, the bias is small, but the variance is large for different values of p. This implies that the polynomial basis function does not have a very balanced trade-off between the bias and variance. It also implies that the function does not oversimplify the model, pays attention to training data and does not generalise on data it hasn't seen before. This suggests that the function may be overfitting, i.e., it might lead to good performance on the training data but poor result and high error rates on test data.

The Gaussian basis function seems to have a reasonably balanced trade-off between the bias and variance. However, the bias is slightly higher than the variance for all values of p. This implies that it often oversimplifies the model and may often generalise on data it hasn't seen before. It also suggests that it may be underfitting, i.e., it may lead to poor performance on the training data, poor result on test data and high error on both training and test data.

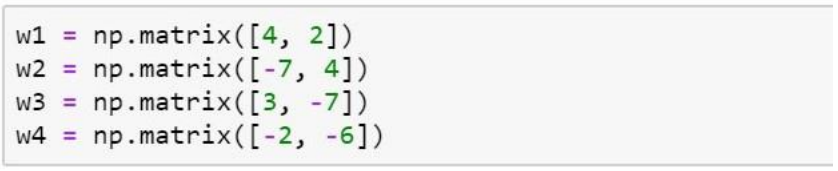
Classification

Data 1: separate 2 Gaussians

Visual Exploration of Consequences of Projecting Data onto a Lower Dimension



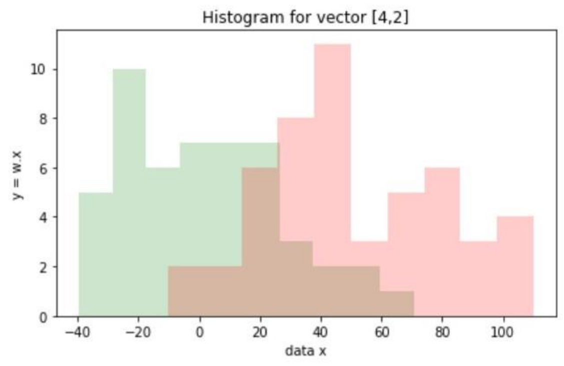
```
meanA = [0,0]
covA = [[20,0],[0,100]]
xA = np.random.multivariate_normal(meanA,covA,50)

meanB = [12,2]
covB = [[40,0],[0,40]]
xB = np.random.multivariate_normal(meanB,covB,50)
```

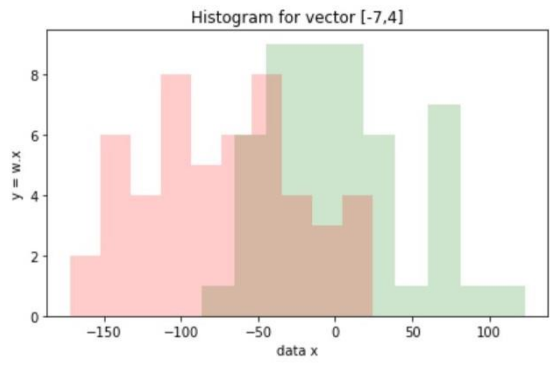


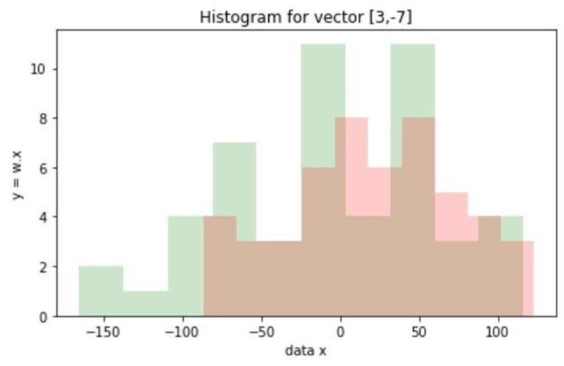
```
w1 = np.matrix([4, 2])
w2 = np.matrix([-7, 4])
w3 = np.matrix([3, -7])
w4 = np.matrix([-2, -6])
```

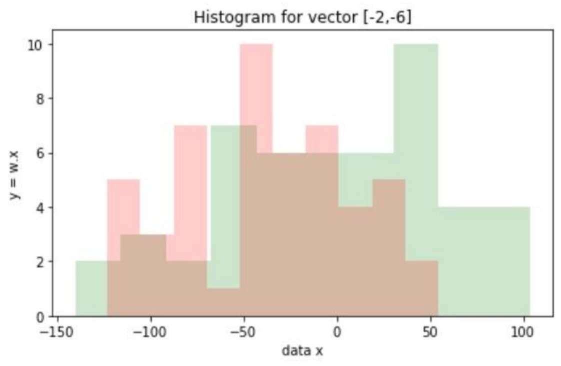
I obtained two datasets containing 50 data points each from a normal distribution with mean vectors and covariance matrices of dimensions (2x1) and (2x2) respectively. I did this using the multivariate_normal function in numpy.random. For the weight vectors, I chose 4 weight vectors that pointed in different directions to get uniquely differing histograms.

The histograms I obtained are present below.



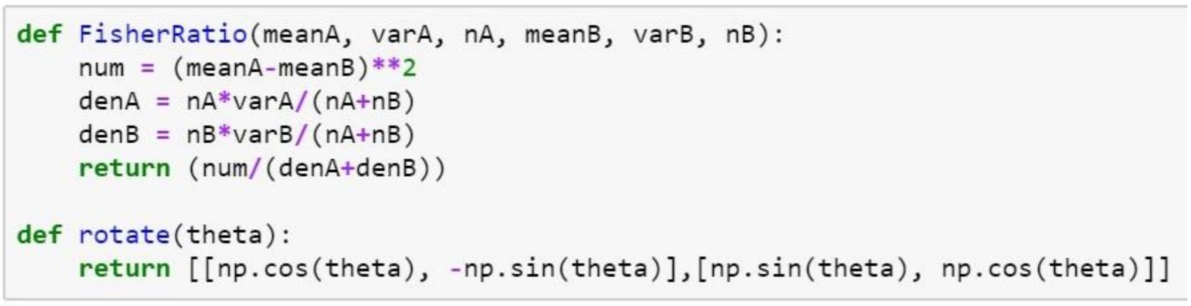




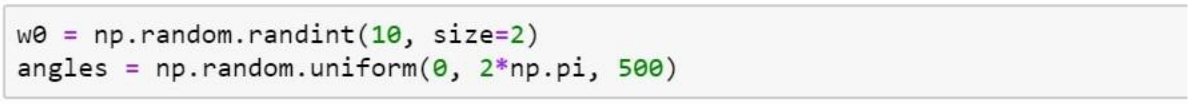


As can be seen from the histograms, the extent of overlap between the two distributions in case of vectors [3, -7] and [-2, -6] is much greater as compared to the overlap in case of vectors [4, 2] and [-7, 4].

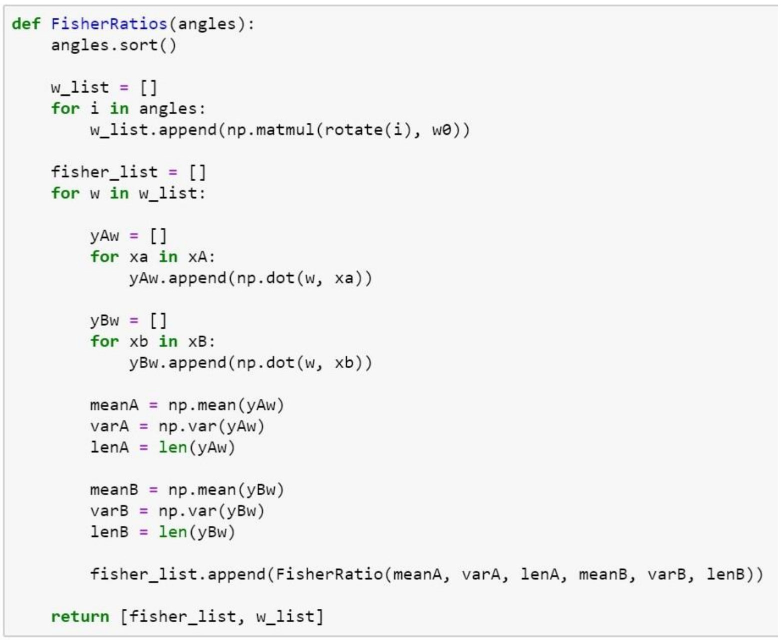
For the second part, I plotted the dependence of the Fisher ratio on the direction of an arbitrarily chosen weight vector w by rotating it by different angles. I created a function FisherRatio which takes two means, variances and number of elements of two normal distributions and returns the Fisher Ratio for these values. Another function rotate, computes and returns R(θ). Both these functions have been created from the formulae specified in the specifications and are present below. I randomly generate w0 and a list of 500 angles for which to calculate the Fisher ratios.



```
def FisherRatio(meanA, varA, nA, meanB, varB, nB):
    num = (meanA-meanB)**2
    denA = nA*varA/(nA+nB)
    denB = nB*varB/(nA+nB)
    return (num/(denA+denB))

def rotate(theta):
    return [[np.cos(theta), -np.sin(theta)],[np.sin(theta), np.cos(theta)]]
```

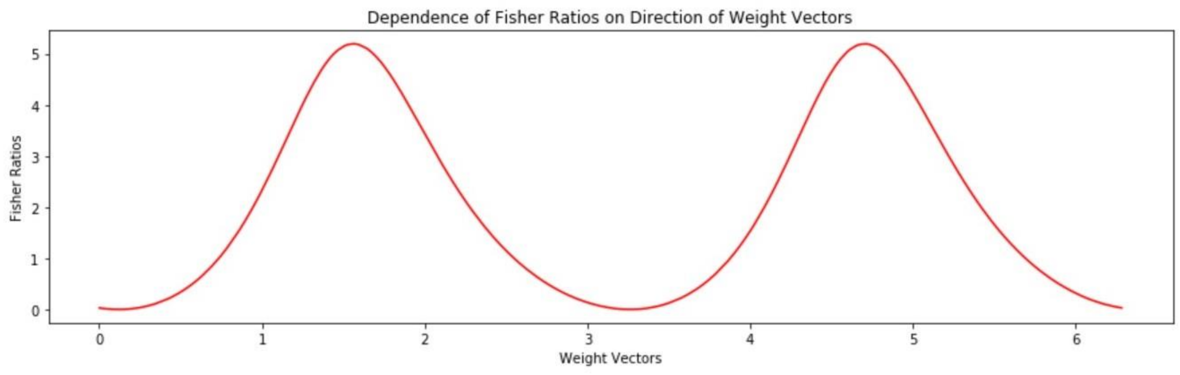


```
w0 = np.random.randint(10, size=2)
angles = np.random.uniform(0, 2*np.pi, 500)
```

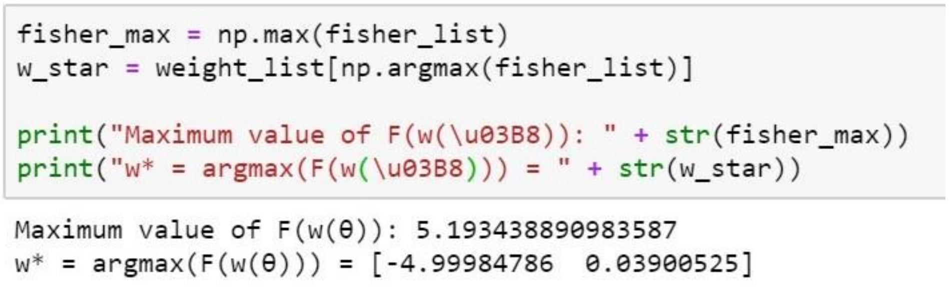


```python
def FisherRatios(angles):
    angles.sort()

    w_list = []
    for i in angles:
        w_list.append(np.matmul(rotate(i), w0))

    fisher_list = []
    for w in w_list:

        yAw = []
        for xa in xA:
            yAw.append(np.dot(w, xa))

        yBw = []
        for xb in xB:
            yBw.append(np.dot(w, xb))

        meanA = np.mean(yAw)
        varA = np.var(yAw)
        lenA = len(yAw)

        meanB = np.mean(yBw)
        varB = np.var(yBw)
        lenB = len(yBw)

        fisher_list.append(FisherRatio(meanA, varA, lenA, meanB, varB, lenB))

    return [fisher_list, w_list]
```

For calculating the Fisher ratio for a list of angles, I created the function above called FisherRatios. It takes a list of angles and returns a list of Fisher ratios and w(θ) values associated with each angle. The two lists returned have been plotted on the graph below. The graph obtained is a continuous, bell-shaped curve.



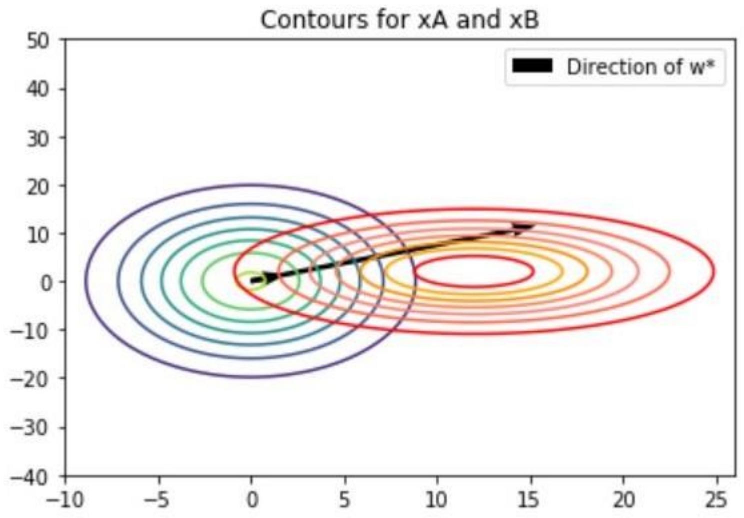
Fisher ratio is used to assess how good a particular classifier is. A high Fisher ratio implies that the corresponding vector is suitable for classifying the data. The maximum value from the list of all the Fisher ratios is obtained and used to calculate the corresponding direction defined by vector w*. It is calculated using the function described in the specifications and is highlighted in the code snippet below.



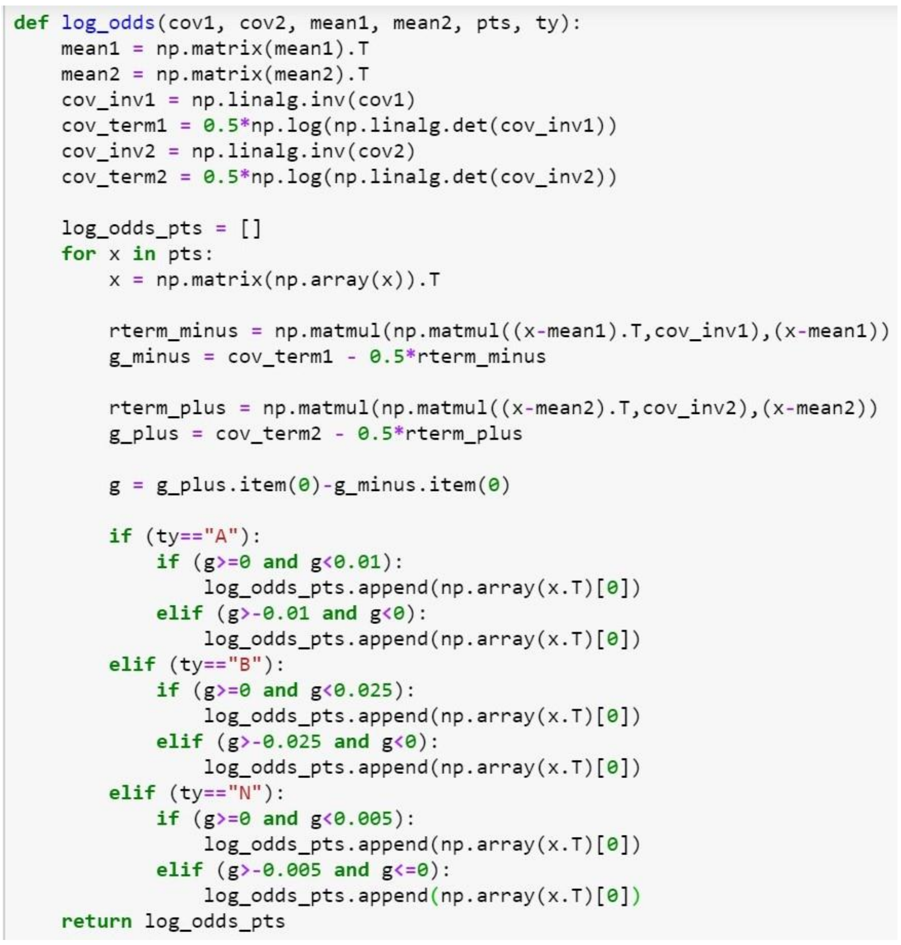
```python
fisher_max = np.max(fisher_list)
w_star = weight_list[np.argmax(fisher_list)]

print("Maximum value of F(w(\u03B8)): " + str(fisher_max))
print("w* = argmax(F(w(\u03B8))) = " + str(w_star))
```

```
Maximum value of F(w(θ)): 5.193438890983587
w* = argmax(F(w(θ))) = [-4.99984786  0.03900525]
```

Visualisation of Probability Distributions



The purple-green contour represents the distribution xA, and the red-orange contour represents the distribution xB, both of which I have defined previously. The black arrow represents the direction of the optimal choice vector w* obtained prior. There is a certain degree of overlap between the two contours implying that the distributions have certain points in common, but do not overlap completely and hence also consist of unique points. The tilt and shape of the contours accurately depicts the choice of means and covariance matrices of the two distribution.



```python
def log_odds(cov1, cov2, mean1, mean2, pts, ty):
    mean1 = np.matrix(mean1).T
    mean2 = np.matrix(mean2).T
    cov_inv1 = np.linalg.inv(cov1)
    cov_term1 = 0.5*np.log(np.linalg.det(cov_inv1))
    cov_inv2 = np.linalg.inv(cov2)
    cov_term2 = 0.5*np.log(np.linalg.det(cov_inv2))

    log_odds_pts = []
    for x in pts:
        x = np.matrix(np.array(x)).T

        rterm_minus = np.matmul(np.matmul((x-mean1).T,cov_inv1),(x-mean1))
        g_minus = cov_term1 - 0.5*rterm_minus

        rterm_plus = np.matmul(np.matmul((x-mean2).T,cov_inv2),(x-mean2))
        g_plus = cov_term2 - 0.5*rterm_plus

        g = g_plus.item(0)-g_minus.item(0)

        if (ty=="A"):
            if (g>=0 and g<0.01):
                log_odds_pts.append(np.array(x.T)[0])
            elif (g>-0.01 and g<0):
                log_odds_pts.append(np.array(x.T)[0])
        elif (ty=="B"):
            if (g>=0 and g<0.025):
                log_odds_pts.append(np.array(x.T)[0])
            elif (g>-0.025 and g<0):
                log_odds_pts.append(np.array(x.T)[0])
        elif (ty=="N"):
            if (g>=0 and g<0.005):
                log_odds_pts.append(np.array(x.T)[0])
            elif (g>-0.005 and g<=0):
                log_odds_pts.append(np.array(x.T)[0])
    return log_odds_pts
```

To acquire the points for plotting the decision boundary, I created the function above called log_odds. This function takes two mean vectors, covariance matrices, a list of points and a string type. The function goes over each point in the list, and using the formula specified below, calculates the log odds.

$$log - odds = g_{+}(x) - g_{-}(x)$$

where

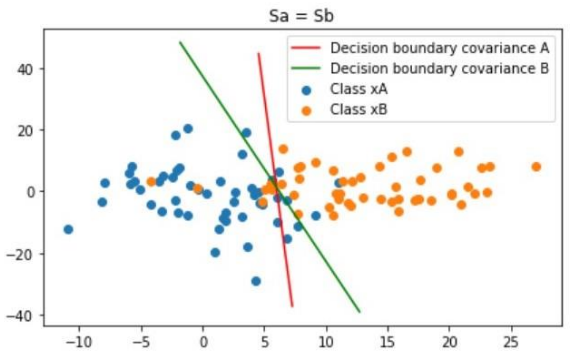
$$g_{-}(x) = \ln P(C_{-}) + \frac{1}{2}\ln|\Lambda_{-}| - \frac{1}{2}(x - \mu_{-})^{T}\Lambda_{-}(x - \mu_{-})$$

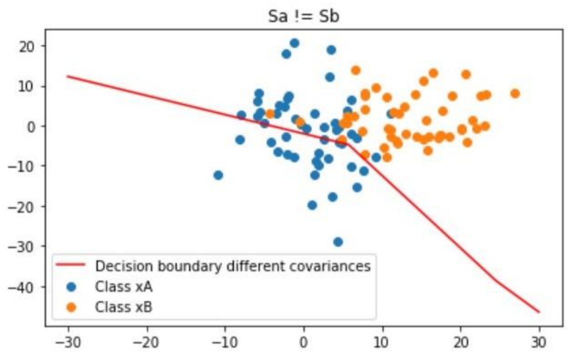
$$g_{+}(x) = \ln P(C_{+}) + \frac{1}{2}\ln|\Lambda_{+}| - \frac{1}{2}(x - \mu_{+})^{T}\Lambda_{+}(x - \mu_{+})$$

and Λ is the inverse of the covariance matrix. In the log_odds function, I make the assumption that ln $P(C_{-})$ and ln $P(C_{+})$ in $g_{-}(x)$ and $g_{+}(x)$ respectively, cancel out each other.

The set of points $\in \mathbb{R}^2$ to iterate over is obtained by getting the Cartesian product of two lists of arrays generated using numpy.linspace.

Since it is difficult to get points where log-odds are exactly 0, I consider points where the log-odds are small and close to 0. Therefore, if the log-odds are close to 0, the point is added to a list within the function. Once iteration over all the points is completed, the list of points at which log-odds are 0 is returned. These points are plotted as the decision boundary shown in the graph, and the scattered points are the points obtained from the distributions xA and xB.



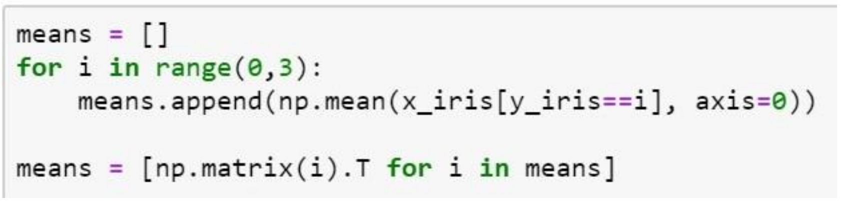


For $S_a = S_b$ I plotted two decision boundaries. The red decision boundary which is obtained using the covariance matrix of distribution xA for both g_{+} and g_{-}. The green decision boundary is obtained using the covariance matrix of distribution xB for both g_{+} and g_{-}. Since the covariance matrices are the same in both these cases, the decision boundaries obtained are linear. This is because the quadratic term is common to both classes, and hence gets cancelled out.

For $S_a \neq S_b$ I plotted a decision boundary which is obtained using the covariance matrices of both distributions xA and xB, which are different. Since the covariance matrices are different, the decision boundary obtained should be quadratic, i.e., it should have a curve. However, due to the range of points that I selected to iterate over; I was not able to obtain an appropriate decision boundary. A better choice of points could have led to a more precise decision boundary.

Data 2: Iris data

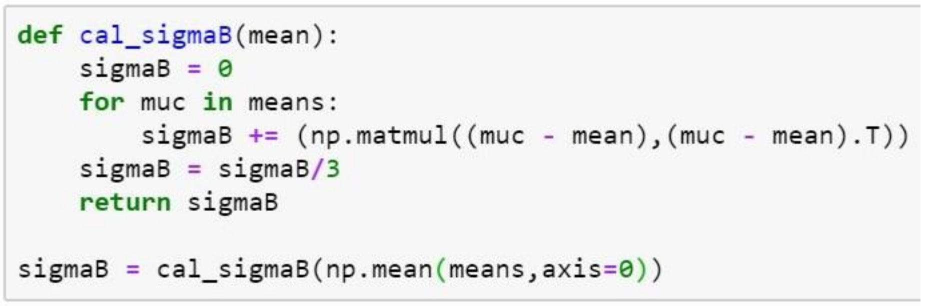
I used sklearn.datasets.load_iris() to obtain the iris dataset which I then split into two lists, one containing the data and the other the target.



```
means = []
for i in range(0,3):
    means.append(np.mean(x_iris[y_iris==i], axis=0))

means = [np.matrix(i).T for i in means]
```

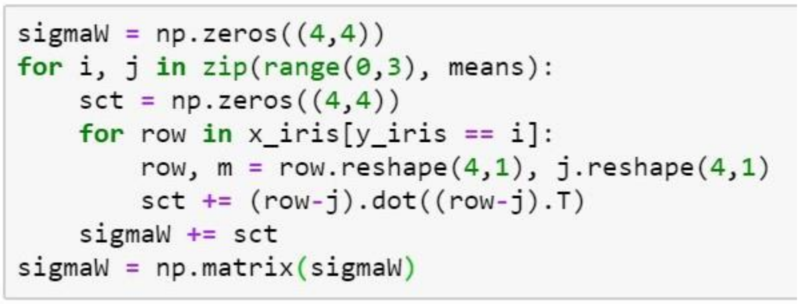
The mean vectors for the three classes are obtained from the data as described in the code snippet above.

The variance-covariance between-class matrix Σ_B is a (4 x 4) matrix and is calculated using the function described below. The function implements the formula given in the specifications.



```
def cal_sigmaB(mean):
    sigmaB = 0
    for muc in means:
        sigmaB += (np.matmul((muc - mean),(muc - mean).T))
    sigmaB = sigmaB/3
    return sigmaB

sigmaB = cal_sigmaB(np.mean(means,axis=0))
```

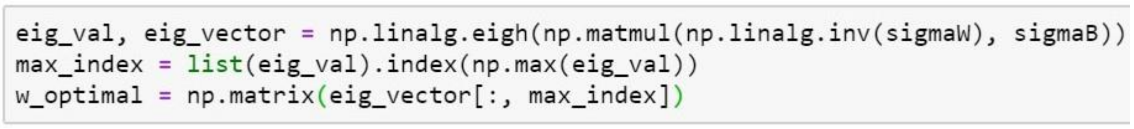
The variance-covariance within-class matrix Σ_W is the sum of the covariance matrices for each class. It is also a (4 x 4) matrix and is calculated using the function below.



```
sigmaW = np.zeros((4,4))
for i, j in zip(range(0,3), means):
    sct = np.zeros((4,4))
    for row in x_iris[y_iris == i]:
        row, m = row.reshape(4,1), j.reshape(4,1)
        sct += (row-j).dot((row-j).T)
    sigmaW += sct
sigmaW = np.matrix(sigmaW)
```

The sigmaW matrix is initially set to be a (4 x 4) matrix of 0s. Next, by looping over a zipped list of [0,1,2] (the target values) and the mean vectors, the covariance matrix for each class is obtained. The nested loop is used to create the column vectors of the covariance matrix of each class. At the end of the nested loop, the covariance matrix of the current class is added to sigmaW, which at the end of the outer loop gives the sum of the covariance matrices of all the classes.

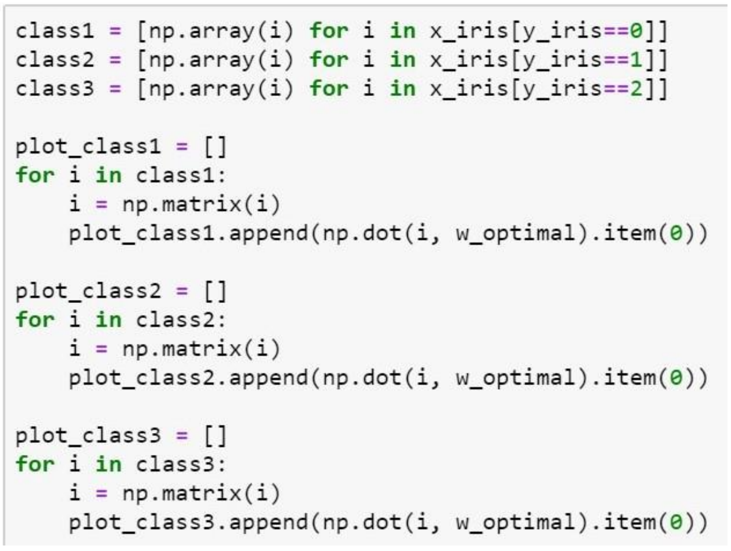
The optimal direction w* for projecting the data is obtained by solving the generalised eigenvalue problem $\Sigma_B w = \lambda \Sigma_W W$, for the matrix $\Sigma_W^{-1}\Sigma_B$. I used the eigh function in numpy to solve this as can be seen in the code below.



```
eig_val, eig_vector = np.linalg.eigh(np.matmul(np.linalg.inv(sigmaW), sigmaB))
max_index = list(eig_val).index(np.max(eig_val))
w_optimal = np.matrix(eig_vector[:, max_index])
```

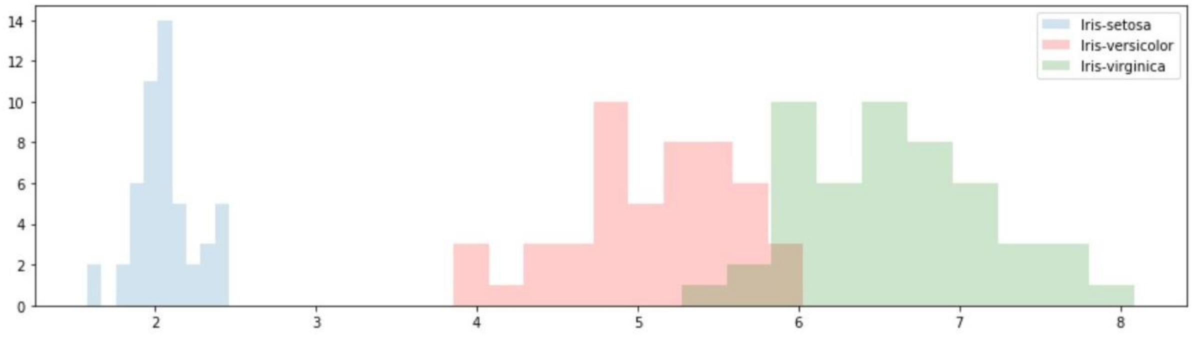
Once all the eigenvalues and eigenvectors are obtained, the optimal direction w* is selected as the eigenvector corresponding to the largest eigenvalue, which is what I have done in the code above.

To display the histograms of the three classes in the reduced dimensional space defined by w*, I first separated the three classes into lists. After separating the classes, I obtained the data to be plotted for each class by multiplying each row of each class with the optimal direction w*. This can be seen in the code snippet below.



```
class1 = [np.array(i) for i in x_iris[y_iris==0]]
class2 = [np.array(i) for i in x_iris[y_iris==1]]
class3 = [np.array(i) for i in x_iris[y_iris==2]]

plot_class1 = []
for i in class1:
    i = np.matrix(i)
    plot_class1.append(np.dot(i, w_optimal).item(0))

plot_class2 = []
for i in class2:
    i = np.matrix(i)
    plot_class2.append(np.dot(i, w_optimal).item(0))

plot_class3 = []
for i in class3:
    i = np.matrix(i)
    plot_class3.append(np.dot(i, w_optimal).item(0))
```

The histogram plotted below suggests that there is a high separation between the data of the Iris-setosa class and the other two classes. It also indicates that there is a small degree of overlap between the data of Iris-versicolor and of Iris-virginica classes.



The Fisher ratio can also be represented as in the formula below.

$$F(w) = \frac{w^T A w}{w^T B w}$$

Since the Fisher ratio can also be represented as a generalised eigenvalue problem, the relation between the class separation task and the generalised eigenvector formulation is that in both cases the method of computation performed is essentially the same. For both cases, the eigenvalue problem can be solved in order to obtain the optimal direction w*.