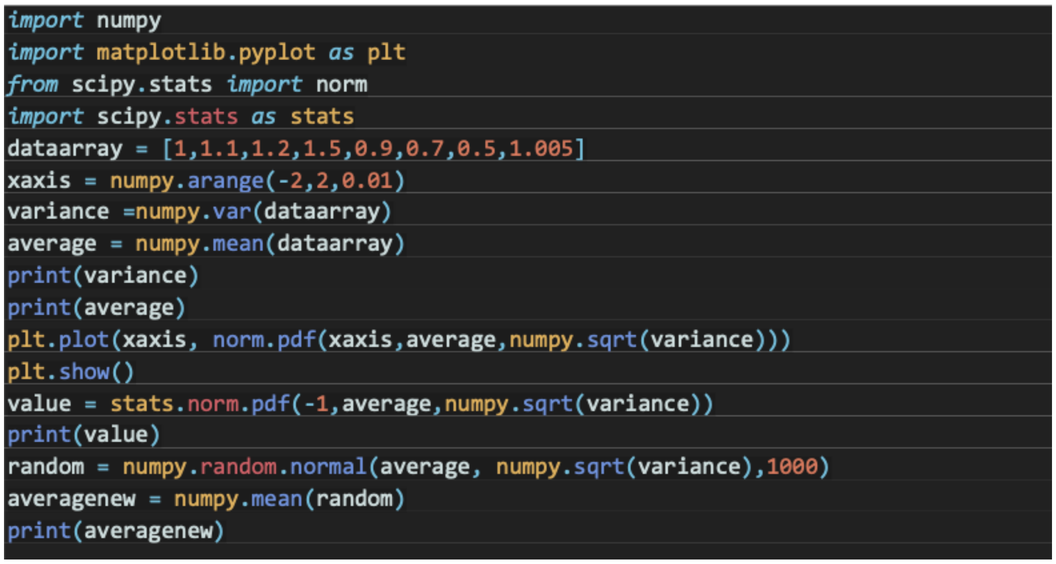
Machine Learning First Project - Report

Q1)

Then we moved on to the implementation section. In this section, we performed the calculations using the libraries NumPy and SciPy, and printed each part so the output is visible during execution. The code used is provided below.



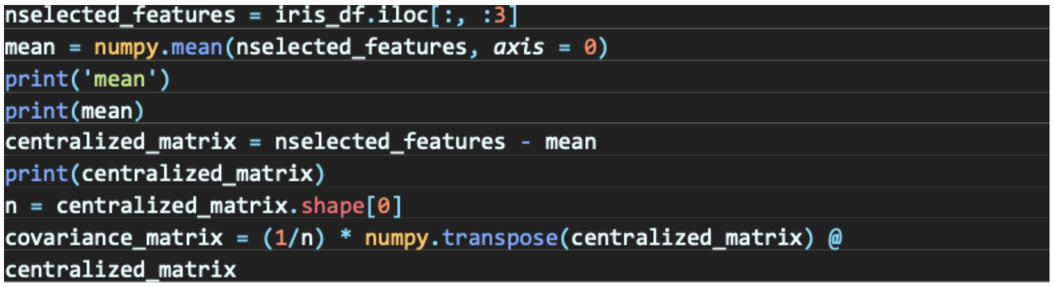
```
import numpy
import matplotlib.pyplot as plt
from scipy.stats import norm
import scipy.stats as stats
dataarray = [1,1.1,1.2,1.5,0.9,0.7,0.5,1.005]
xaxis = numpy.arange(-2,2,0.01)
variance =numpy.var(dataarray)
average = numpy.mean(dataarray)
print(variance)
print(average)
plt.plot(xaxis, norm.pdf(xaxis,average,numpy.sqrt(variance)))
plt.show()
value = stats.norm.pdf(-1,average,numpy.sqrt(variance))
print(value)
random = numpy.random.normal(average, numpy.sqrt(variance),1000)
averagenew = numpy.mean(random)
print(averagenew)
```

Q2)

First, we extracted the dataset according to the requirements of the question from the Scikit-Learn library. (a) Then, we considered the desired features. After that, using the Matplotlib library, we displayed the data in a two-dimensional space based on the features mentioned in section (b).

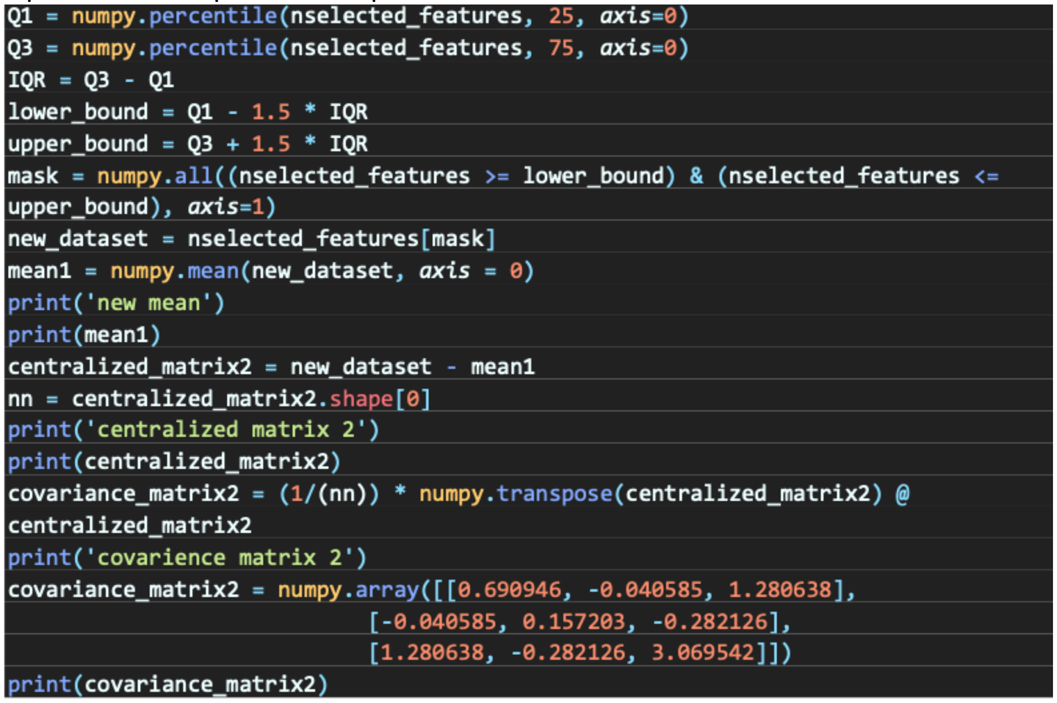
Then, for section (c), according to the following code, we first calculated the mean of each column and then the centralized matrix.

For section (d), using the centralized matrix from the previous section, we calculated the covariance matrix, which is shown below.



```
nselected_features = iris_df.iloc[:, :3]
mean = numpy.mean(nselected_features, axis = 0)
print('mean')
print(mean)
centralized_matrix = nselected_features - mean
print(centralized_matrix)
n = centralized_matrix.shape[0]
covariance_matrix = (1/n) * numpy.transpose(centralized_matrix) @
centralized_matrix
```

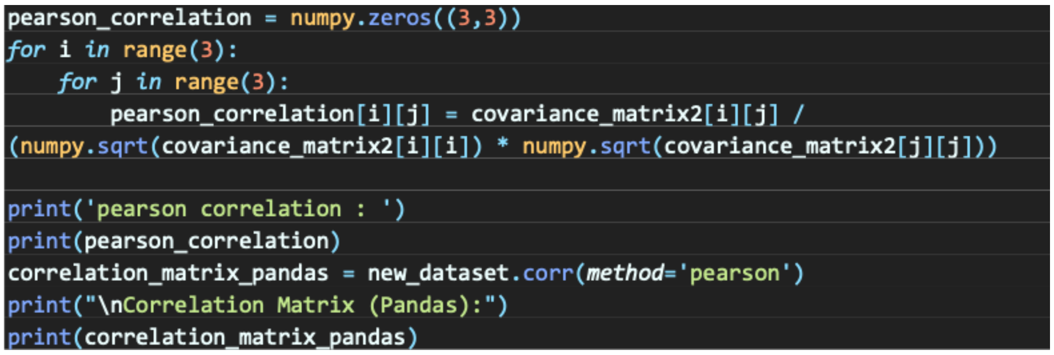
After that, for section (e), we removed the outliers according to the specified limits and repeated the steps from the previous section.



```
Q1 = numpy.percentile(nselected_features, 25, axis=0)
Q3 = numpy.percentile(nselected_features, 75, axis=0)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
mask = numpy.all((nselected_features >= lower_bound) & (nselected_features <=
upper_bound), axis=1)
new_dataset = nselected_features[mask]
mean1 = numpy.mean(new_dataset, axis = 0)
print('new mean')
print(mean1)
centralized_matrix2 = new_dataset - mean1
nn = centralized_matrix2.shape[0]
print('centralized matrix 2')
print(centralized_matrix2)
covariance_matrix2 = (1/(nn)) * numpy.transpose(centralized_matrix2) @
centralized_matrix2
print('covarience matrix 2')
covariance_matrix2 = numpy.array([[0.690946, -0.040585, 1.280638],
                              [-0.040585, 0.157203, -0.282126],
                              [1.280638, -0.282126, 3.069542]])
print(covariance_matrix2)
```

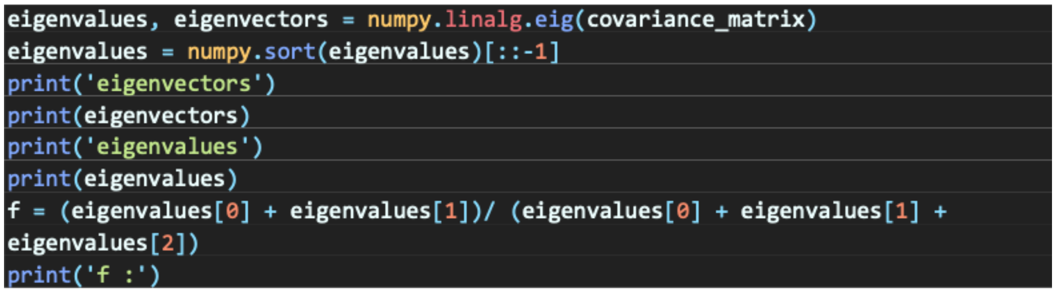
This change can be interpreted in such a way that the outliers have been removed. The variance of the first feature has slightly increased, while the variance of the second feature has decreased. Meanwhile, the variance of the third feature has also decreased slightly. Additionally, the linear correlation between the second and third features has increased.

For section (f), we calculated the Pearson correlation for the two data sets and their responses. It was observed that the results were the same and did not differ.

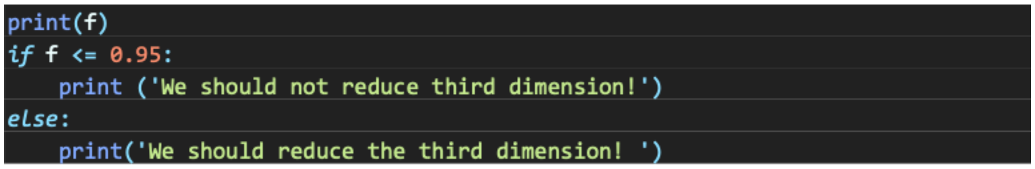


```
pearson_correlation = numpy.zeros((3,3))
for i in range(3):
    for j in range(3):
        pearson_correlation[i][j] = covariance_matrix2[i][j] /
(numpy.sqrt(covariance_matrix2[i][i]) * numpy.sqrt(covariance_matrix2[j][j]))

print('pearson correlation : ')
print(pearson_correlation)
correlation_matrix_pandas = new_dataset.corr(method='pearson')
print("\nCorrelation Matrix (Pandas):")
print(correlation_matrix_pandas)
```

For the last part of the second question, section (g), we followed the steps in the book. Using the Principal Component Analysis (PCA) method, we reduced the dimensions and managed to meet the requirements of the question. The results were consistent with the example in the book and were the same.

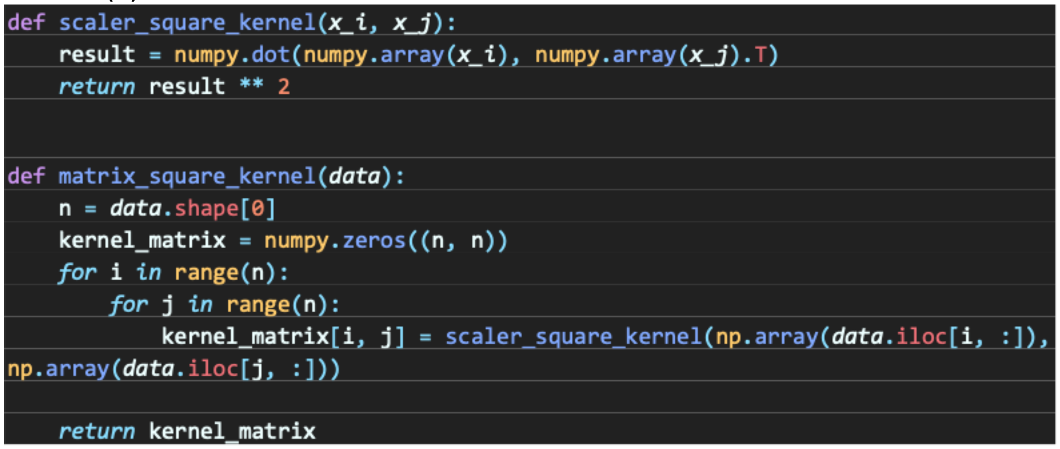


```
eigenvalues, eigenvectors = numpy.linalg.eig(covariance_matrix)
eigenvalues = numpy.sort(eigenvalues)[::-1]
print('eigenvectors')
print(eigenvectors)
print('eigenvalues')
print(eigenvalues)
f = (eigenvalues[0] + eigenvalues[1])/ (eigenvalues[0] + eigenvalues[1] +
eigenvalues[2])
print('f :')
```



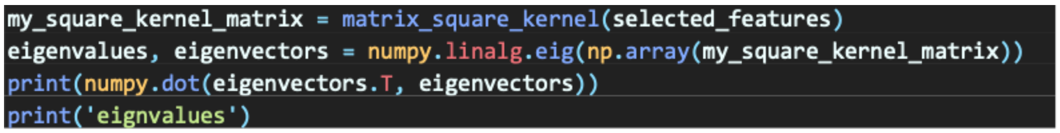
```
print(f)
if f <= 0.95:
    print ('We should not reduce third dimension!')
else:
    print('We should reduce the third dimension! ')
```

Q3)
First, according to the question's requirement, we removed the fifth feature. Then we went to section (a) and calculated it.

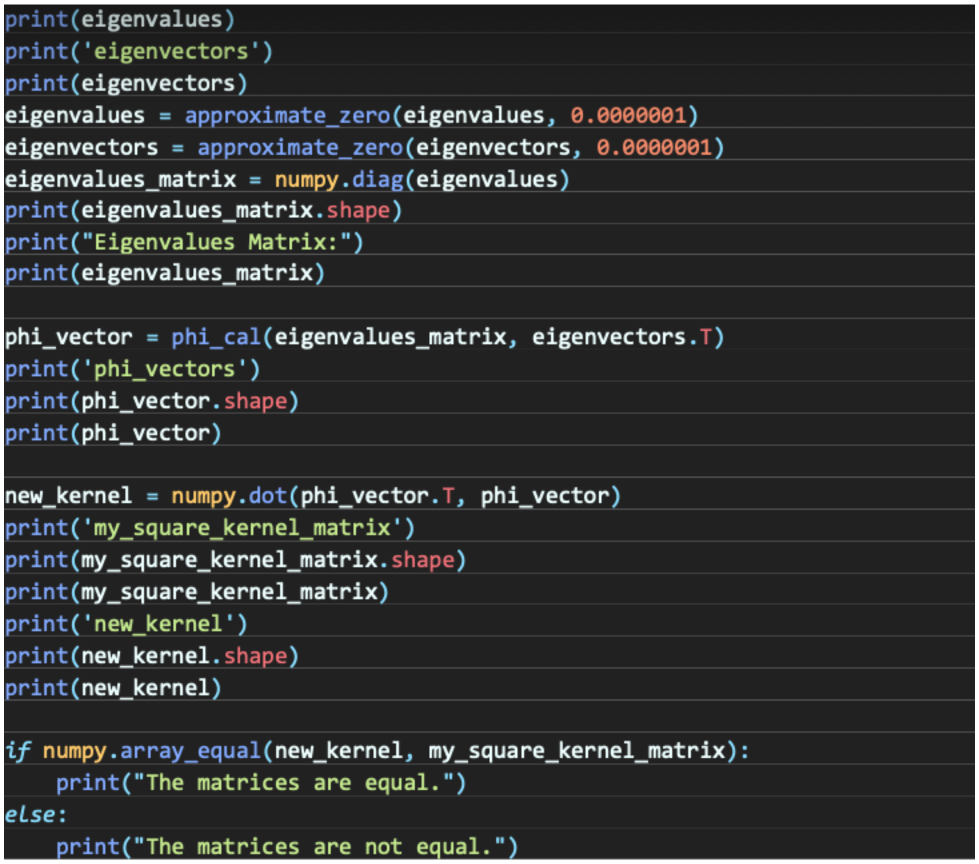


```
def scaler_square_kernel(x_i, x_j):
    result = numpy.dot(numpy.array(x_i), numpy.array(x_j).T)
    return result ** 2


def matrix_square_kernel(data):
    n = data.shape[0]
    kernel_matrix = numpy.zeros((n, n))
    for i in range(n):
        for j in range(n):
            kernel_matrix[i, j] = scaler_square_kernel(np.array(data.iloc[i, :]),
np.array(data.iloc[j, :]))

    return kernel_matrix
```

Then for section (b), we calculated the $\phi(x_i)$ value for each i, which is observable in the provided code. For section (c), using the obtained results and multiplying them, we reconstructed the kernel matrix and compared the results with section (a), which were identical.

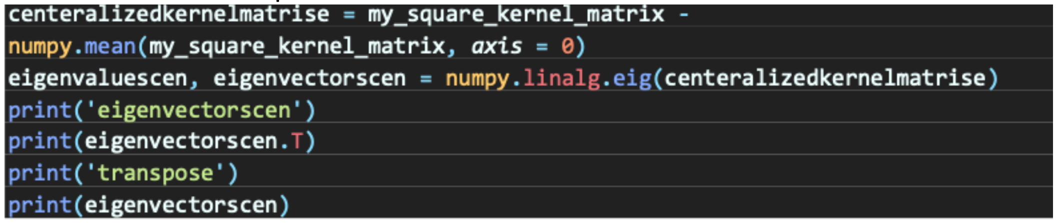


```
my_square_kernel_matrix = matrix_square_kernel(selected_features)
eigenvalues, eigenvectors = numpy.linalg.eig(np.array(my_square_kernel_matrix))
print(numpy.dot(eigenvectors.T, eigenvectors))
print('eignvalues')
```

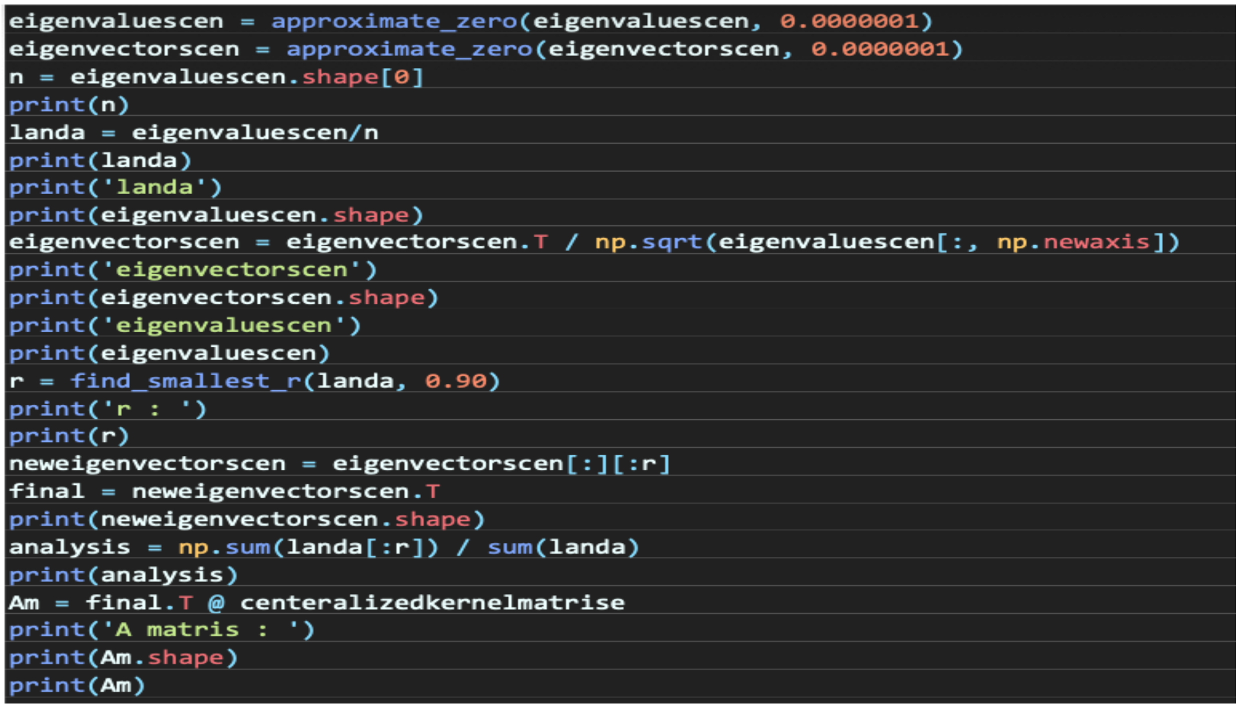


```python
print(eigenvalues)
print('eigenvectors')
print(eigenvectors)
eigenvalues = approximate_zero(eigenvalues, 0.0000001)
eigenvectors = approximate_zero(eigenvectors, 0.0000001)
eigenvalues_matrix = numpy.diag(eigenvalues)
print(eigenvalues_matrix.shape)
print("Eigenvalues Matrix:")
print(eigenvalues_matrix)

phi_vector = phi_cal(eigenvalues_matrix, eigenvectors.T)
print('phi_vectors')
print(phi_vector.shape)
print(phi_vector)

new_kernel = numpy.dot(phi_vector.T, phi_vector)
print('my_square_kernel_matrix')
print(my_square_kernel_matrix.shape)
print(my_square_kernel_matrix)
print('new_kernel')
print(new_kernel.shape)
print(new_kernel)

if numpy.array_equal(new_kernel, my_square_kernel_matrix):
    print("The matrices are equal.")
else:
    print("The matrices are not equal.")
```

Then for section (d), we implemented the Kernel Principal Component Analysis (Kernel PCA) algorithm on the centered kernel matrix and observed that with retaining only one feature, 93% of the information is preserved.



```python
centeralizedkernelmatrise = my_square_kernel_matrix -
numpy.mean(my_square_kernel_matrix, axis = 0)
eigenvaluescen, eigenvectorscen = numpy.linalg.eig(centeralizedkernelmatrise)
print('eigenvectorscen')
print(eigenvectorscen.T)
print('transpose')
print(eigenvectorscen)
```



```
eigenvaluescen = approximate_zero(eigenvaluescen, 0.0000001)
eigenvectorscen = approximate_zero(eigenvectorscen, 0.0000001)
n = eigenvaluescen.shape[0]
print(n)
landa = eigenvaluescen/n
print(landa)
print('landa')
print(eigenvaluescen.shape)
eigenvectorscen = eigenvectorscen.T / np.sqrt(eigenvaluescen[:, np.newaxis])
print('eigenvectorscen')
print(eigenvectorscen.shape)
print('eigenvaluescen')
print(eigenvaluescen)
r = find_smallest_r(landa, 0.90)
print('r : ')
print(r)
neweigenvectorscen = eigenvectorscen[:][:r]
final = neweigenvectorscen.T
print(neweigenvectorscen.shape)
analysis = np.sum(landa[:r]) / sum(landa)
print(analysis)
Am = final.T @ centeralizedkernelmatrise
print('A matris : ')
print(Am.shape)
print(Am)
```