Results And Analysis IML lab 1

Question 1:

The code contains 2 functions, one to generate a matrix with 0 mean and unit variance and the other to calculate the frobenius norm of the matrix.

When we pass the matrix in the frobenius norm function, it calculates and return the frobenius norm.

```python
import numpy as np
from math import sqrt

def generate_matrix(a,b):
  return np.random.normal(loc=0,scale=1,size=(a,b))

def frobenius_norm(a):
  norm = sqrt(np.sum(np.square(a)))
  return norm

frob_norm = frobenius_norm(generate_matrix(6,8))
print(frob_norm)
```

```
C:\Users\jivit\Desktop\IML\self>C:/Python313/python.exe c:/Users/jivit/Desktop/IML/self/q1.py
Frobenius norm: 7.081158477062735
```

Question 2:

(a) Here we have to find the eigen values and eigen vectors of a symmetric matrix B = A + A.T. We also ensure that the order of eigen values is such that $\lambda 1 \geq \lambda 2 \geq \cdots \geq \lambda n$.

```python
def generate_matrix(a,b):
    return np.random.normal(loc=0,scale=1,size=(a,b))

a = generate_matrix(100,100)
b = a + a.T

val, vec = np.linalg.eigh(b)
final_val = val[::-1]
final_val = np.diag(val)
print(final_val,"\n")

final_vec = vec[:,::-1]
print(final_vec)
```

Result:

```
C:\Users\jivit\Desktop\IML\self>C:/Python313/python.exe c:/Users/jivit/Desktop/IML/self/q2.py
[[-27.8714569    0.           0.         ...   0.           0.
    0.        ]
 [  0.         -26.40971608   0.         ...   0.           0.
    0.        ]
 [  0.           0.         -24.57325857 ...   0.           0.
    0.        ]
 ...
 [  0.           0.           0.         ...  25.5631128    0.
    0.        ]
 [  0.           0.           0.         ...   0.          26.46011453
    0.        ]
 [  0.           0.           0.         ...   0.           0.
   27.68052813]]

[[-0.02161619  0.0208874  -0.10290463 ...  0.14154861 -0.10246963
   0.06164548]
 [ 0.06938928 -0.05701891 -0.07744794 ...  0.10661496 -0.11530585
   0.14984763]
 [-0.01243752  0.00965891  0.08484245 ... -0.14448885 -0.11660967
   0.07010528]
 ...
 [-0.04291186 -0.02217966  0.01599516 ... -0.04473111  0.01780051
   0.08575787]
 [-0.1145134   0.01976476  0.06829126 ... -0.05265859 -0.00755685
   0.08702755]
 [ 0.1204628   0.03754411  0.12726968 ...  0.01177607  0.00770957
   0.11000997]]
```

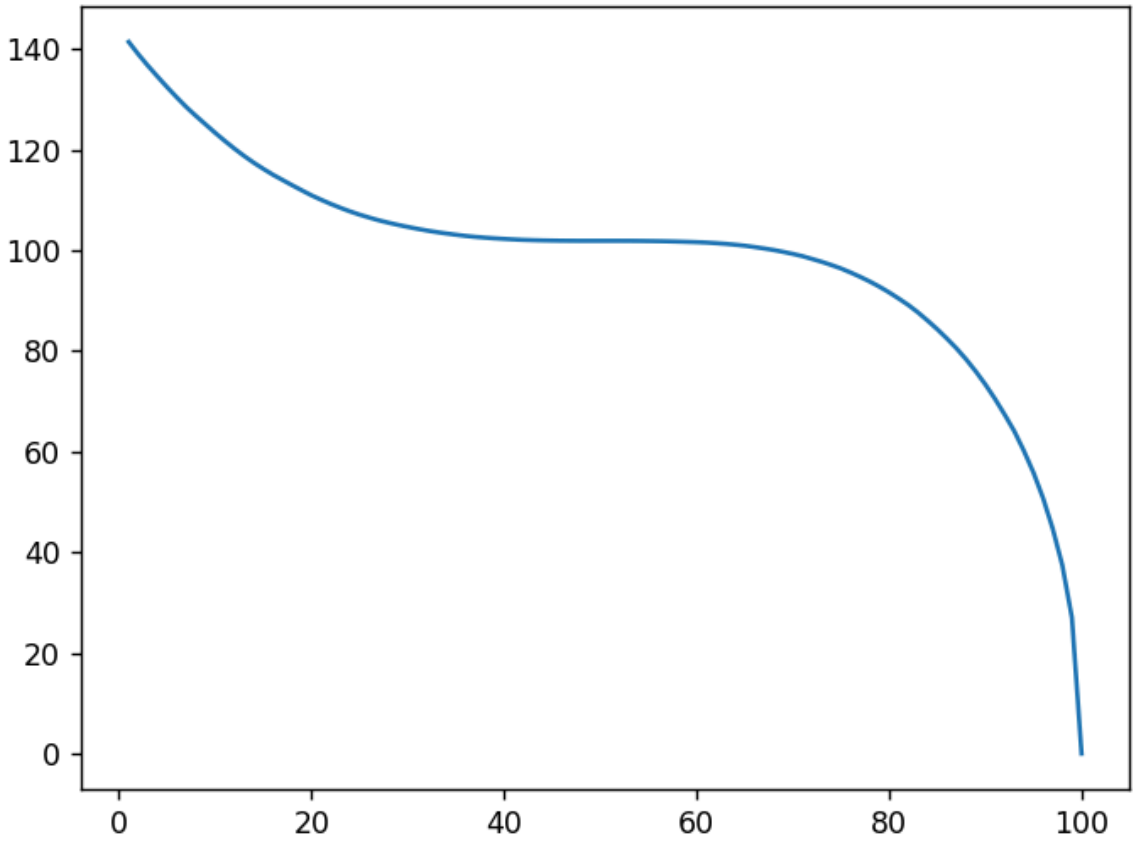
b) Here we are required to find the matrix Bk and find the frobenius norm of B - Bk.

```
def frobenius_norm(a):
  norm = sqrt(np.sum(np.square(a)))
  return norm

def beekay(b,k):
  BK = np.zeros(b.shape)
  val, vec = np.linalg.eigh(b)
  val = val[::-1]
  vec = vec[:, ::-1]
  for i in range(k):
    vi=np.expand_dims(vec[:,i],axis=1)
    BK = BK + val[i]*vi@vi.T

  # print(BK, "\n")
  return BK

BK = beekay(b,2)  #taking k = 2
frob = frobenius_norm(b - BK)
print("frobenius norm:",frob)
```

Result:

```
frobenius norm: 137.297530910679
```

c) Plotting graph (B - Bk) vs k, we get



From this graph we can observe that the error is very high when the value of k is low and decreases as we increase the k value, so for Bk to be approximately B we need to take 90 - 100 eigenvalues that is, the value of k should be high.

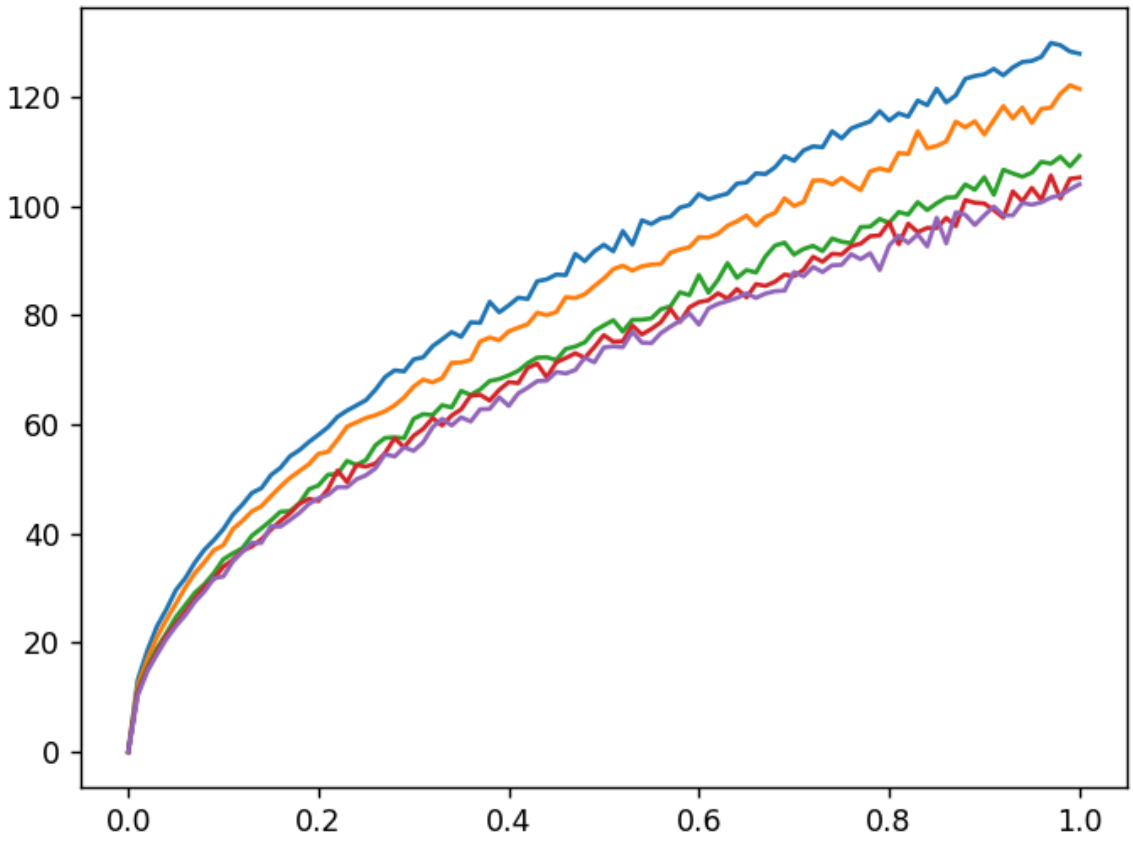
d) finding frobenius norm of (Vt.V - I)

```
I = np.eye(100)
print(frobenius_norm(vec@vec.T - I))
```

Output:

```
1.937519395335164e-14
```

e) Result



From the plots we see that frobenius norm increases as sigma square increases. For larger k, the error is always smaller, since more eigenvalues/eigenvectors are included in Bk. All curves start near zero and grow smoothly with sigma square. Thus, higher k gives a better approximation, and the error increases with the variance.

Question 3)

a) result:

```
[7.1 3.  5.9 2.1]
[6.5 3.  5.8 2.2]
[4.9 2.5 4.5 1.7]
[6.7 2.5 5.8 1.8]
[6.5 3.2 5.1 2. ]
[6.8 3.  5.5 2.1]
[5.8 2.8 5.1 2.4]
[6.5 3.  5.5 1.8]
[7.7 2.6 6.9 2.3]
[6.9 3.2 5.7 2.3]
[7.7 2.8 6.7 2. ]
[6.7 3.3 5.7 2.1]
[6.2 2.8 4.8 1.8]
[6.4 2.8 5.6 2.1]
[7.4 2.8 6.1 1.9]
[6.4 2.8 5.6 2.2]
[6.1 2.6 5.6 1.4]
[6.3 3.4 5.6 2.4]
[6.  3.  4.8 1.8]
[6.7 3.1 5.6 2.4]
[5.8 2.7 5.1 1.9]
[6.7 3.3 5.7 2.5]
[6.3 2.5 5.  1.9]
[6.2 3.4 5.4 2.3]]

THe covariance matrix is:
[[ 0.64945946 -0.09489189  1.22948649  0.50537838]
[-0.09489189  0.18963243 -0.42141622 -0.15666486]
[ 1.22948649 -0.42141622  3.17887568  1.36285405]
[ 0.50537838 -0.15666486  1.36285405  0.62820901]]
```

b) result

```
most (array([2]), array([2]))
least (array([1, 2]), array([2, 1]))
```

c) Result

```
1: 5.84 2: 3.0640000000000005 3: 3.776 4: 1.2173333333333334
1: 0.6408 2: 0.18710399999999996 3: 3.1364906666666665 4: 0.6198328888888888
```

d) result:

```
val [0.0197967  0.10053045 0.21638968 4.30945975]
```

The variance of the i-th feature corresponds to the diagonal element of the covariance matrix, while the i-th eigenvalue of the covariance matrix represents the variance captured along a principal direction. Individual feature variances may not match eigenvalues one-to-one, but the total variance across all features equals the sum of all eigenvalues. Thus, eigenvalues can be viewed as a redistribution of the feature variances along rotated axes.