Mathematical Foundations of Machine Learning. Homework 5

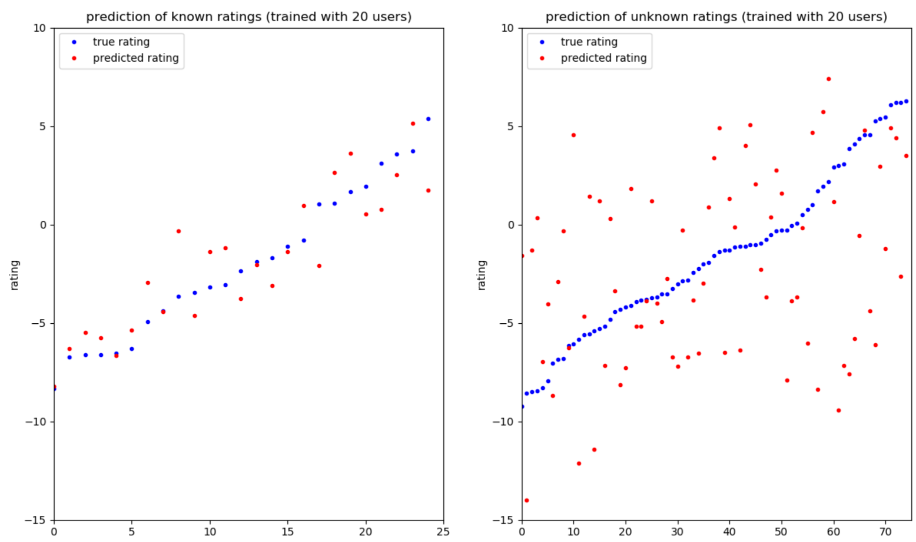
Teresa Morales

November 22nd, 2019

1. Problem 1

   (a) Least squares using 20 customers.
   The prediction seems to work well with training data but it works much worse when using the testing sample. This might suggest that the sample is too small for good predictions on new data to be made. Figure 1 shows results graphically:

Figure 1:



The average least squares error for the training data is 1.719.
The average least squares error for the test sample is 5.362
The code used is the following:

```python
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import norm
from mpl_toolkits.mplot3d import Axes3D


# Problem 1 a)

file_path = '/Users/teresamorales/Documents/Harris/MFML/Homework5/jesterdata.mat'
jester_data = sio.loadmat(file_path)
X = jester_data['X']


file_path = '/Users/teresamorales/Documents/Harris/MFML/Homework5/newuser.mat'
d_new = sio.loadmat(file_path)

y = d_new['y']
true_y = d_new['truey']

# total number of joke ratings should be m = 100, n = 7200
m, n = X.shape

# train on ratings we know for the new user
train_indices = np.squeeze(y != -99)
num_train = np.count_nonzero(train_indices)

# test on ratings we  d o n t  know
test_indices = np.logical_not(train_indices)
num_test = m - num_train

X_data = X[train_indices, 0:20]
y_data = y[train_indices]
y_test = true_y[test_indices]

# solve for weights


def calc_weights(X, y):
    Xt = np.transpose(X)
    XtX = np.matmul(Xt, X)
    Inverse_XtX = np.linalg.inv(XtX)
    w = np.matmul(np.matmul(Inverse_XtX, Xt), y)
    return w


w_hat = calc_weights(X_data, y_data)

# compute predictions
X_test = X[test_indices, 0:20]
y_hat_train = np.matmul(X_data, w_hat)  # Prediction for training data
y_hat_test = np.matmul(X_test, w_hat)  # Prediction for test data

```

```python
# measure performance on training jokes
error_train = np.subtract(y_data, y_hat_train)
error_train_squares = np.square(error_train)
avgerr_train = np.sqrt(np.mean(error_train_squares))

# display results

ax1 = plt.subplot(121)
sorted_indices = np.argsort(np.squeeze(y_data))
ax1.plot(range(num_train), y_data[sorted_indices], 'b.',
         range(num_train), y_hat_train[sorted_indices], 'r.')
ax1.set_title('prediction of known ratings (trained with 20 users)')
ax1.set_xlabel('jokes (sorted by true rating)')
ax1.set_ylabel('rating')
ax1.legend(['true rating', 'predicted rating'], loc='upper left')
ax1.axis([0, num_train, -15, 10])
print("Average l_2 error (train):", avgerr_train)

# measure performance on unrated jokes
error_test = np.subtract(y_test, y_hat_test)
error_test_squares = np.square(error_test)
avgerr_test = np.sqrt(np.mean(error_test_squares))

# display results
ax2 = plt.subplot(122)
sorted_indices = np.argsort(np.squeeze(y_test))
ax2.plot(range(num_test), y_test[sorted_indices], 'b.',
         range(num_test), y_hat_test[sorted_indices], 'r.')
ax2.set_title('prediction of unknown ratings (trained with 20 users)')
ax2.set_xlabel('jokes (sorted by true rating)')
ax2.set_ylabel('rating')
ax2.legend(['true rating', 'predicted rating'], loc='upper left')
ax2.axis([0, num_test, -15, 10])
print("Average l_2 (test):", avgerr_test)
plt.show()
```

(b) Using the entire features matrix.

With the new dimensions, there are infinitely many solutions. We can find the least squares solution, which will have the smallest norm.

In order to find the least squares solutions we need to redefine how we estimate the weights using the new dimensions. In particular, before we had:

$$\hat{w} = (X^T X)^{-1} X^T y$$

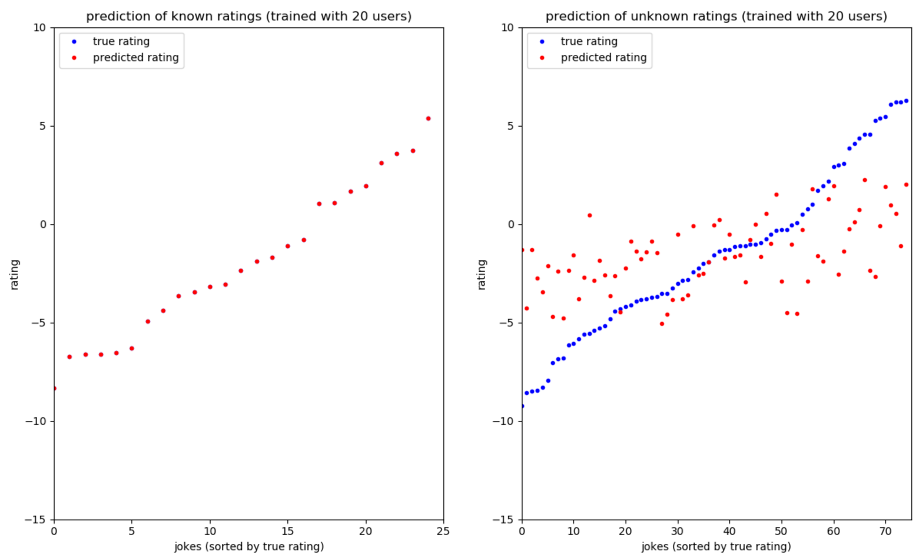
and now we have

$$\hat{w} = X^T (X X^T)^{-1} y$$

With this additional information, errors get smaller. In particular, the mean error for the training sample is 0 and the error for the testing sample is 3.494.

Figure 2 shows results graphically:

Figure 2:



The code used is the following:

```
def calc_weights_under(X, y):
    Xt = np.transpose(X)
    XXt = np.matmul(X, Xt)
    Inverse_XXt = np.linalg.inv(XXt)
    w = np.matmul(np.matmul(Xt, Inverse_XXt), y)
    return w


X_data = X[train_indices, :]
w_hat_under = calc_weights_under(X_data, y_data)

# compute predictions

X_test = X[test_indices, :]
y_hat_train = np.matmul(X_data, w_hat_under)  # Prediction
     for training data
y_hat_test = np.matmul(X_test, w_hat_under)  # Prediction
    for test data

# measure performance on training jokes
error_train = np.subtract(y_data, y_hat_train)
```

```
error_train_squares = np.square(error_train)
avgerr_train = np.sqrt(np.mean(error_train_squares))

# display results

ax1 = plt.subplot(121)
sorted_indices = np.argsort(np.squeeze(y_data))
ax1.plot(range(num_train), y_data[sorted_indices], 'b.',
         range(num_train), y_hat_train[sorted_indices], 'r.')
ax1.set_title('prediction of known ratings (trained with 20 users)')
ax1.set_xlabel('jokes (sorted by true rating)')
ax1.set_ylabel('rating')
ax1.legend(['true rating', 'predicted rating'], loc='upper left')
ax1.axis([0, num_train, -15, 10])
print("Average l_2 error (train):", avgerr_train)

# measure performance on unrated jokes
error_test = np.subtract(y_test, y_hat_test)
error_test_squares = np.square(error_test)
avgerr_test = np.sqrt(np.mean(error_test_squares))

# display results
ax2 = plt.subplot(122)
sorted_indices = np.argsort(np.squeeze(y_test))
ax2.plot(range(num_test), y_test[sorted_indices], 'b.',
         range(num_test), y_hat_test[sorted_indices], 'r.')
ax2.set_title('prediction of unknown ratings (trained with 20 users)')
ax2.set_xlabel('jokes (sorted by true rating)')
ax2.set_ylabel('rating')
ax2.legend(['true rating', 'predicted rating'], loc='upper left')
ax2.axis([0, num_test, -15, 10])
print("Average l_2 (test):", avgerr_test)
plt.show()
```

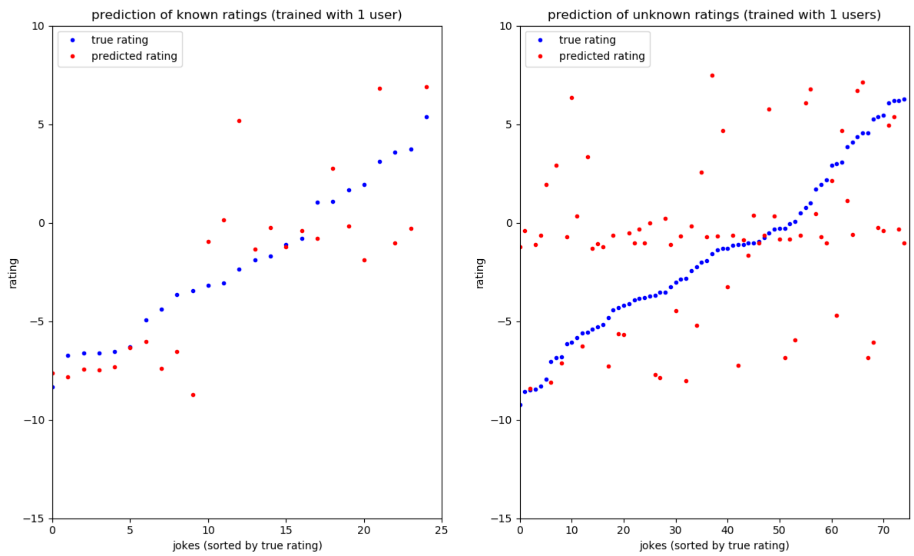
(c) One and two users

Suggested method: estimating the similarity between users using the cosine of the features vectors for the observed ratings.

For one user the mean error is 5.99 for the training sample and 5.82 for the testing samples.

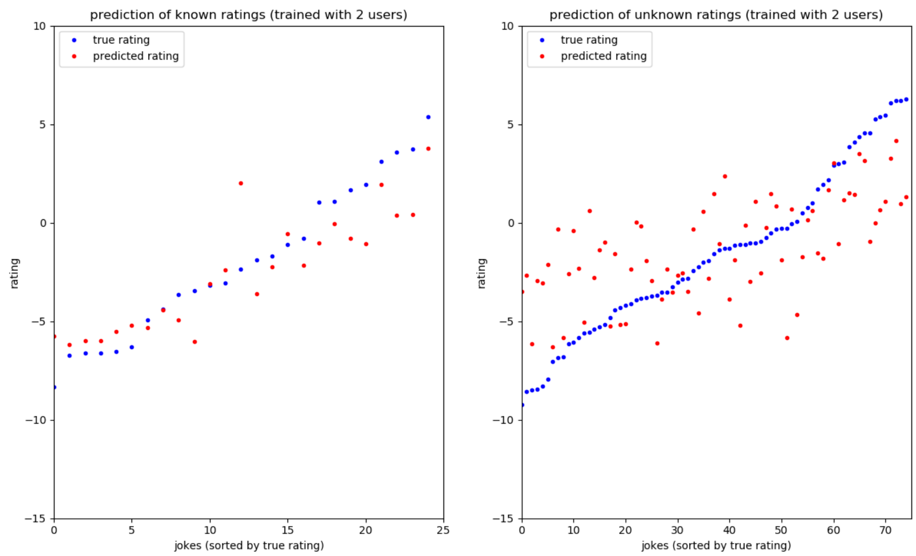
Results are shown in the following graph (figure 3):

Figure 3:



Adding a second user has a huge impact in terms of error. The mean error for the training sample is 1.892 and for the testing sample 3.195. Results are showin in figure 4. For one user the code is the following:

Figure 4:




```
def similarity(v1, v2):
    return np.dot(v1, v2)/(np.linalg.norm(v1)*np.linalg.
    norm(v2))


sim_list = [similarity(X_data[:, i], y_data) for i in
    range(n)]

cust_1 = sim_list.index(max(sim_list))

# Customer with index 588 would be the one closest to the
    new costumer in ratings of training data
y_hat_train = X_data[:, cust_1]
y_hat_test = X_test[:, cust_1]

# measure performance on training jokes
error_train = np.subtract(y_data, y_hat_train)
error_train_squares = np.square(error_train)
avgerr_train = np.sqrt(np.mean(error_train_squares))

# display results
ax1 = plt.subplot(121)
sorted_indices = np.argsort(np.squeeze(y_data))
ax1.plot(range(num_train), y_data[sorted_indices], 'b.',
         range(num_train), y_hat_train[sorted_indices], 'r
    .')
ax1.set_title('prediction of known ratings (trained with 1
     user)')
```

```
ax1.set_xlabel('jokes (sorted by true rating)')
ax1.set_ylabel('rating')
ax1.legend(['true rating', 'predicted rating'], loc='upper
    left')
ax1.axis([0, num_train, -15, 10])
print("Average l_2 error (train):", avgerr_train)

# measure performance on unrated jokes
error_test = np.subtract(y_test, y_hat_test)
error_test_squares = np.square(error_test)
avgerr_test = np.sqrt(np.mean(error_test_squares))

# display results
ax2 = plt.subplot(122)
sorted_indices = np.argsort(np.squeeze(y_test))
ax2.plot(range(num_test), y_test[sorted_indices], 'b.',
         range(num_test), y_hat_test[sorted_indices], 'r.'
    )
ax2.set_title('prediction of unknown ratings (trained with
    1 users)')
ax2.set_xlabel('jokes (sorted by true rating)')
ax2.set_ylabel('rating')
ax2.legend(['true rating', 'predicted rating'], loc='upper
    left')
ax2.axis([0, num_test, -15, 10])
print("Average l_2 (test):", avgerr_test)
plt.show()
```

For two users the code is the following:

```
sim_list.remove(max(sim_list))

cust_2 = sim_list.index(max(sim_list))+1

X_train_2 = X_data[:, [cust_1, cust_2]]
X_test_2 = X_test[:, [cust_1, cust_2]]

w_hat = calc_weights(X_train_2, y_data)

# compute predictions

y_hat_train = np.matmul(X_train_2, w_hat)  # Prediction
    for training data
y_hat_test = np.matmul(X_test_2, w_hat)  # Prediction for
    test data

# measure performance on training jokes
error_train = np.subtract(y_data, y_hat_train)
error_train_squares = np.square(error_train)
avgerr_train = np.sqrt(np.mean(error_train_squares))

# display results
ax1 = plt.subplot(121)
sorted_indices = np.argsort(np.squeeze(y_data))
ax1.plot(range(num_train), y_data[sorted_indices], 'b.',
         range(num_train), y_hat_train[sorted_indices], 'r
    .')
ax1.set_title('prediction of known ratings (trained with 2
```

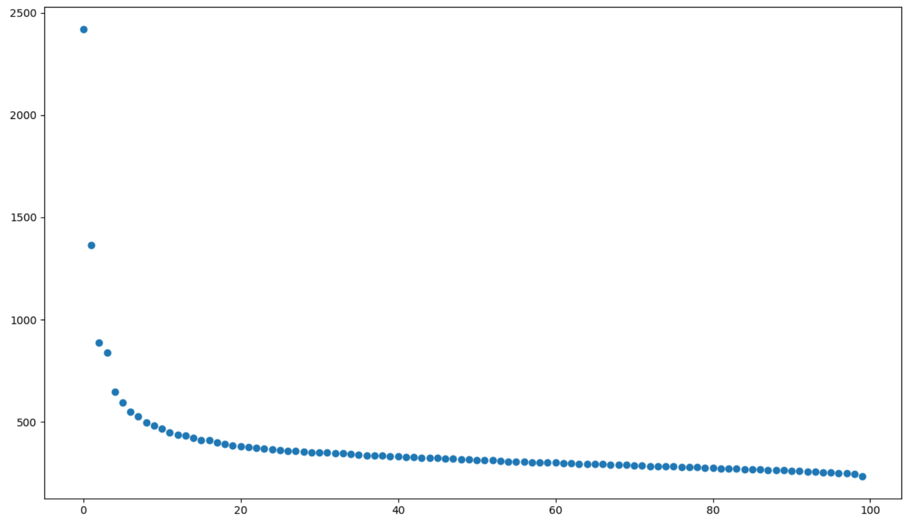
```
    users)')
ax1.set_xlabel('jokes (sorted by true rating)')
ax1.set_ylabel('rating')
ax1.legend(['true rating', 'predicted rating'], loc='upper
    left')
ax1.axis([0, num_train, -15, 10])

print("Average l_2 error (train):", avgerr_train)

# measure performance on unrated jokes
error_test = np.subtract(y_test, y_hat_test)
error_test_squares = np.square(error_test)
avgerr_test = np.sqrt(np.mean(error_test_squares))

# display results
ax2 = plt.subplot(122)
sorted_indices = np.argsort(np.squeeze(y_test))
ax2.plot(range(num_test), y_test[sorted_indices], 'b.',
         range(num_test), y_hat_test[sorted_indices], 'r.'
    )
ax2.set_title('prediction of unknown ratings (trained with
    2 users)')
ax2.set_xlabel('jokes (sorted by true rating)')
ax2.set_ylabel('rating')
ax2.legend(['true rating', 'predicted rating'], loc='upper
    left')
ax2.axis([0, num_test, -15, 10])
print("Average l_2 (test):", avgerr_test)
plt.show()
```

(d) The rank of X is 100.

Looking at the plot it seems that only 4 or 5 dimensions seem important. This means that having that number of users, if selected to be as similar to the costumer we are predicting as possible, should give enough information to perform a good prediction. In mathematical terms, this means that we can use a 4 dimension subspace that approximates this data.

```
u, s, vh = np.linalg.svd(X, full_matrices=False)
s.shape
# The rank of X is 100.
plt.scatter(range(len(s)), s)
plt.show()
```

Figure 5:



(e) Section e)

The first three principal components X (100x7200) are the first three columns of U. The first three principal components of X transpose (7200x100) are the first three columns of U.

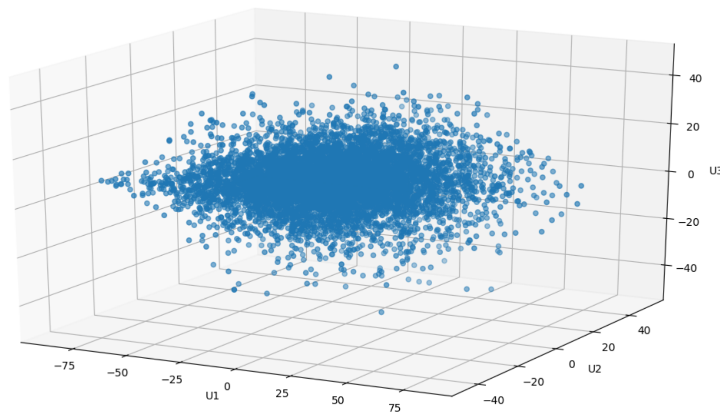
In order to find the projection of each point (each costumer) onto the principal components we would do the following:

```
X_p1 = np.matmul(np.diag(s[0:3]), vh[0:3, :])


fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_p1[0, :], X_p1[1, :], X_p1[2, :])
ax.set_xlabel('U1', fontsize=10)
ax.set_ylabel('U2', fontsize=10)
ax.set_zlabel('U3', fontsize=10)
ax.legend()
plt.show()
```

The resulting graph can be seen in figure 6:

As we can see, there is little structure in the data, which means that the three principal components give a good amount of information about costumers' tastes. However, we can still observe some higher variance on one direction which might indicate that a fourth principal

Figure 6:



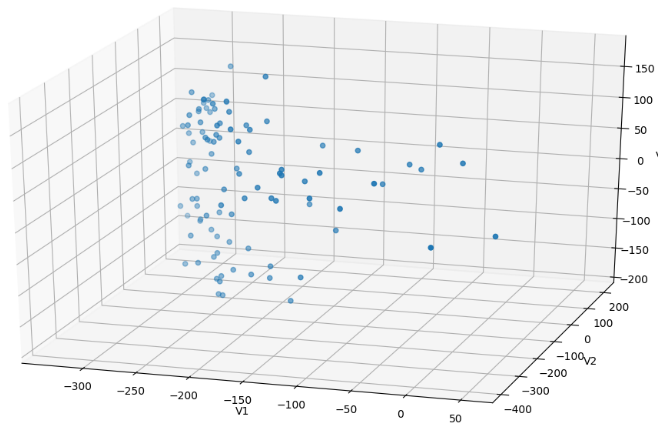
component could help better predict ratings.

Similarly, we can project each joke onto the corresponding three principal components as follows:

```
U_t = np.transpose(u)
X_p2 = np.matmul(np.diag(s[0:3]), U_t[0:3, :])

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_p2[0, :], X_p2[1, :], X_p2[2, :])
ax.set_xlabel('V1', fontsize=10)
ax.set_ylabel('V2', fontsize=10)
ax.set_zlabel('V3', fontsize=10)
ax.legend()
plt.show()
```

The corresponding graph is in figure 7:

Figure 7:



As before, it seems that a fourth principal component could be helpful although probably enough information is captured by the first three principal components to make good predictions.

(f) Power iterations

As we can see the method gives the same result as the svd function (except for the sign which is interchangeable across U and V).

```
def power_iteration(A, epsilon):
    u_0 = np.random.rand(A.shape[1])
    u_i = np.dot(A, u_0)/np.linalg.norm(np.dot(A, u_0))
    while(np.linalg.norm(u_0-u_i) > epsilon):
        u_0 = u_i
        u_i = np.matmul(A, u_0)/np.linalg.norm(np.matmul(A
    , u_0))
    return u_i


XtX = np.matmul(np.transpose(X), X)

XXt = np.matmul(X, np.transpose(X))

u_1 = power_iteration(XtX, 0.0000000001)

similarity(u_1, vh[0, :])

```

```
# -0.9999999999999998

u_2 = power_iteration(XXt, 0.0000000001)
similarity(u_2, u[:, 0])

# -1
```

(g) Our initial guess can't be completely orthogonal to the final vector. This means, that an initial guess that would not work is the second eigenvector (second column of U to find U1 and second column of V when you are trying to find V1).

2. Problem 2

   (a) Truncated SVD solution Results: The average error is 0.0692
   Code used:

```
import random
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt

file_path = '/Users/teresamorales/Documents/Harris/MFML/Homework2/face_emotion_data.mat'
face_data = sio.loadmat(file_path)
X = face_data['X']
y = face_data['y']


# a)

# Inverse

def calc_inverse(X, k):
    n, p = X.shape
    u, s, vh = np.linalg.svd(X, full_matrices=True)
    v = np.transpose(vh)
    u_t = np.transpose(u)
    sigma_inv_0 = np.array([[0.0 for i in range(n)] for j in range(p)])
    for i in range(k):
        sigma_inv_0[i, i] = 1/s[i]
    return np.matmul(np.matmul(v, sigma_inv_0), u_t)


# Randomize sample partition

def random_partition(sample_size, n_chunks, chunk_size):
    sample_size_index = list(range(sample_size))
    random.Random(222).shuffle(sample_size_index)  # randomizes selection setting seed 222
    return [sample_size_index[round(chunk_size * i):round(chunk_size * (i + 1))] for i in range(n_chunks)]


def sign(num):
```

```
    return -1 if num < 0 else 1


# Predict error for sample reserved (both for choosing k and for hold-out sample)

def calc_error(reserved_calc, reserved_exclude, features_matrix, labels_vector, n_chunks, chunk_size, k):
    X_reserved_calc = features_matrix[sliced_index[reserved_calc]]
    y_reserved_calc = labels_vector[sliced_index[reserved_calc]]
    if reserved_calc < reserved_exclude:
        X_training = features_matrix[sum(
            sliced_index[0:reserved_calc] + sliced_index[reserved_calc+1:reserved_exclude] + sliced_index[reserved_exclude+1:n_chunks], [])]
        y_training = labels_vector[sum(sliced_index[0:reserved_calc] + sliced_index[reserved_calc +
                                 1:reserved_exclude] + sliced_index[reserved_exclude+1:n_chunks], [])]
    elif reserved_calc > reserved_exclude:
        X_training = features_matrix[sum(
            sliced_index[0:reserved_exclude] + sliced_index[reserved_exclude+1:reserved_calc] + sliced_index[reserved_calc+1:n_chunks], [])]
        y_training = labels_vector[sum(
            sliced_index[0:reserved_exclude] + sliced_index[reserved_exclude+1:reserved_calc] + sliced_index[reserved_calc+1:n_chunks], [])]
    svd_inv = calc_inverse(X_training, k)
    w_training = np.matmul(svd_inv, y_training)
    y_hat = np.matmul(X_reserved_calc, w_training)
    y_tilda = [sign(i) for i in y_hat]
    return np.sum([y_tilda[i] != y_reserved_calc[i] for i in range(int(chunk_size))])/chunk_size


# Choose best k
def choose_k(reserved_k, reserved_test, features_matrix, labels_vector, n_chunks, chunk_size):
    error_rates_perk = [calc_error(reserved_k, reserved_test, features_matrix,
                                   labels_vector, n_chunks, chunk_size, i) for i in range(9)]
    min_error_k = error_rates_perk.index(min(error_rates_perk))
    return min_error_k


# Run program repeating the process 56 times
sample_size = 128
number_of_chunks = 8
chunk_size = sample_size/number_of_chunks

```

```
sliced_index = random_partition(sample_size, number_of_chunks, chunk_size)

error_test = []

for i in range(number_of_chunks):
    for j in range(number_of_chunks):
        if j != i:
            min_k = choose_k(i, j, X, y, number_of_chunks, chunk_size)
            error_test.append(calc_error(j, i, X, y, number_of_chunks, chunk_size, min_k))

average_error = np.mean(error_test)
```

(b) Ridge regression

Results: The average error is 0.0335

Code used:

```
# Problem 2b)

# We redefine the inverse:

def calc_inverse(X, k):
    n, p = X.shape
    u, s, vh = np.linalg.svd(X, full_matrices=True)
    v = np.transpose(vh)
    u_t = np.transpose(u)
    sigma_inv_0 = np.array([[0.0 for i in range(n)] for j in range(p)])
    for i in range(9):
        sigma_inv_0[i, i] = s[i]/(s[i]**2+k)
    return np.matmul(np.matmul(v, sigma_inv_0), u_t)

# We redefine the calculation of optimal lambda (k)


def choose_k(reserved_k, reserved_test, features_matrix, labels_vector, n_chunks, chunk_size, lambda_vals):
    error_rates_perk = [calc_error(reserved_k, reserved_test, features_matrix,
                                   labels_vector, n_chunks, chunk_size, i) for i in lambda_vals]
    min_error_k = lambda_vals[error_rates_perk.index(min(error_rates_perk))]
    return min_error_k


lambda_vals = np.array([0, 0.5, 1, 2, 4, 8, 16])

error_test_ridge = []

for i in range(number_of_chunks):
    for j in range(number_of_chunks):
        if j != i:
            min_k = choose_k(i, j, X, y, number_of_chunks, chunk_size, lambda_vals)
```

```
            error_test_ridge.append(calc_error(j, i, X, y, number_of_chunks, chunk_size, min_k))

average_error = np.mean(error_test_ridge)
```

(c) Theoretically, results should not change since the additional singular values will be 0. However when we perform the operations error rates might change due to rounding errors.Actually when calculating inverses, small rounding errors for zero values can have a huge impact although we are trying to reduce this impact using truncates and ridge regression.

When we repeat parts a and b using these new features we obtain an average error rate of 0.09154 for truncated SVD and 0.04576 for ridge regression.

The increase in error rate due to this noise seems to be smaller for ridge regression than for truncated SVD.

The code used is the following:

```
# Problem 2 c)

New_X = np.hstack((X, np.matmul(X, np.random.randn(9, 3))))



# Part a'


def calc_inverse(X, k):
    n, p = X.shape
    u, s, vh = np.linalg.svd(X, full_matrices=True)
    v = np.transpose(vh)
    u_t = np.transpose(u)
    sigma_inv_0 = np.array([[0.0 for i in range(n)] for j in range(p)])
    for i in range(k):
        sigma_inv_0[i, i] = 1/s[i]
    return np.matmul(np.matmul(v, sigma_inv_0), u_t)

# Choose best k


def choose_k(reserved_k, reserved_test, features_matrix, labels_vector, n_chunks, chunk_size):
    error_rates_perk = [calc_error(reserved_k, reserved_test, features_matrix,
                                   labels_vector, n_chunks, chunk_size, i) for i in range(12)]  # Should we change this to 12 for the experiment or leave 9 since there is rounding error?
    min_error_k = error_rates_perk.index(min(error_rates_perk))
    return min_error_k

```

```python

error_test = []

for i in range(number_of_chunks):
    for j in range(number_of_chunks):
        if j != i:
            min_k = choose_k(i, j, New_X, y, number_of_chunks, chunk_size)
            error_test.append(calc_error(j, i, New_X, y, number_of_chunks, chunk_size, min_k))

average_error = np.mean(error_test)


# Part b'

def calc_inverse(X, k):
    n, p = X.shape
    u, s, vh = np.linalg.svd(X, full_matrices=True)
    v = np.transpose(vh)
    u_t = np.transpose(u)
    sigma_inv_0 = np.array([[0.0 for i in range(n)] for j in range(p)])
    for i in range(12):
        sigma_inv_0[i, i] = s[i]/(s[i]**2+k)
    return np.matmul(np.matmul(v, sigma_inv_0), u_t)

# We redefine the calculation of optimal lambda (k)


def choose_k(reserved_k, reserved_test, features_matrix, labels_vector, n_chunks, chunk_size, lambda_vals):
    error_rates_perk = [calc_error(reserved_k, reserved_test, features_matrix,
                                   labels_vector, n_chunks, chunk_size, i) for i in lambda_vals]
    min_error_k = lambda_vals[error_rates_perk.index(min(error_rates_perk))]
    return min_error_k


error_test_ridge = []

for i in range(number_of_chunks):
    for j in range(number_of_chunks):
        if j != i:
            min_k = choose_k(i, j, New_X, y, number_of_chunks, chunk_size, lambda_vals)
            error_test_ridge.append(calc_error(j, i, New_X, y, number_of_chunks, chunk_size, min_k))

average_error = np.mean(error_test_ridge)
```

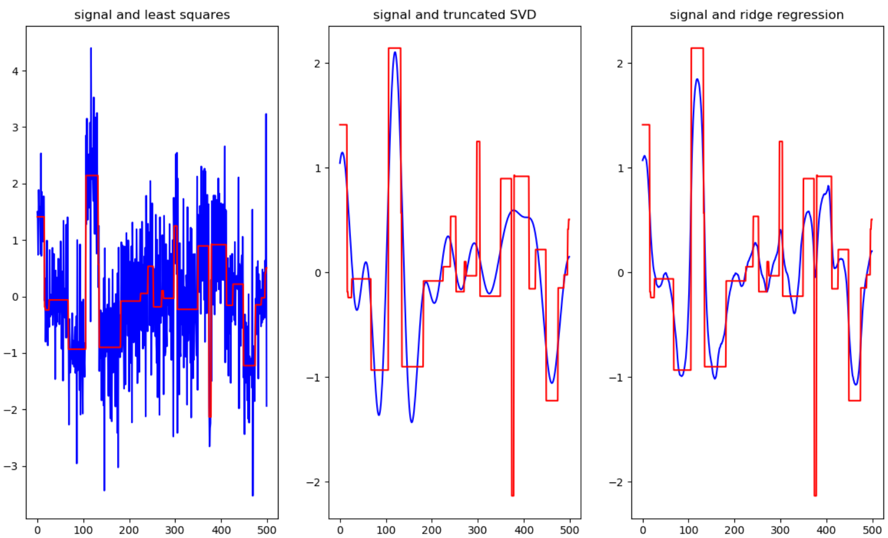
3. Problem 3

   (a) The code is the following. Results are shown in the graph.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# deblurring

n = 500
k = 30
sigma = 0.01

# generate random piecewise constant signal
w = np.zeros((n, 1))
w[0] = np.random.standard_normal()
for i in range(1, n):
    if np.random.rand(1) < 0.95:
        w[i] = w[i-1]
    else:
        w[i] = np.random.standard_normal()


# generate k-point averaging function
h = np.ones(k) / k

# make a matrix for blurring
m = n + k - 1
X = np.zeros((m, m))
for i in range(m):
    if i < k:
        X[i, :i+1] = h[:i+1]
    else:
        X[i, i - k: i] = h

X = X[:, 0:n]

# blurred signal + noise
y = np.dot(X, w) + sigma*np.random.standard_normal(size=(m
    , 1))

# plot
f, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_title('signal')
ax1.plot(w, 'b')
ax2.set_title('blurred and noisy version')
ax2.plot(y[0:n])

plt.show()

# Problem 3 a)

# (1  )Least squares


def calc_weights(X, y):
    Xt = np.transpose(X)
    XtX = np.matmul(Xt, X)
    Inverse_XtX = np.linalg.inv(XtX)
    w = np.matmul(np.matmul(Inverse_XtX, Xt), y)
```

```
    return w


w_hat = calc_weights(X, y)


# (2  )Truncated SVD setting k to 20

def calc_inverse_trunc(X, k):
    n, p = X.shape
    u, s, vh = np.linalg.svd(X, full_matrices=True)
    v = np.transpose(vh)
    u_t = np.transpose(u)
    sigma_inv_0 = np.array([[0.0 for i in range(n)] for j in range(p)])
    for i in range(k):
        sigma_inv_0[i, i] = 1/s[i]
    return np.matmul(np.matmul(v, sigma_inv_0), u_t)


SVD_trunc = calc_inverse_trunc(X, 20)

w_trunc = np.matmul(SVD_trunc, y)


# (3  ) Ridge regression setting lambda to 0.1

def calc_inverse_ridge(X, k):
    n, p = X.shape
    u, s, vh = np.linalg.svd(X, full_matrices=True)
    v = np.transpose(vh)
    u_t = np.transpose(u)
    sigma_inv_0 = np.array([[0.0 for i in range(n)] for j in range(p)])
    for i in range(500):
        sigma_inv_0[i, i] = s[i]/(s[i]**2+k)
    return np.matmul(np.matmul(v, sigma_inv_0), u_t)


SVD_ridge = calc_inverse_ridge(X, 0.1)

w_ridge = np.matmul(SVD_ridge, y)

f, (ax1, ax2, ax3) = plt.subplots(1, 3)
ax1.set_title('signal and least squares')
ax1.plot(w_hat, 'b')
ax1.plot(w, 'r')
ax2.set_title('signal and truncated SVD')
ax2.plot(w_trunc, 'b')
ax2.plot(w, 'r')
ax3.set_title('signal and ridge regression')
ax3.plot(w_ridge, 'b')
ax3.plot(w, 'r')

plt.show()
```

Figure 8:



(b) Using the suggested values for lambda and k, the results we obtain can be observed in figure 9.

As we can see in the left graph for a constant sigma, the optimal truncating k gets smaller as the blurring k increases. However for extreme values of sigma, the relation is constant (the optimal k does not change when sigma is as small as 0.01 or as big as 10).

Similarly, in that same graph we observe that, for a constant blurring k, the optimal truncating k gets smaller as the value of sigma increases (see shifts in the lines as sigma changes).

Looking at the right graph we observe that:

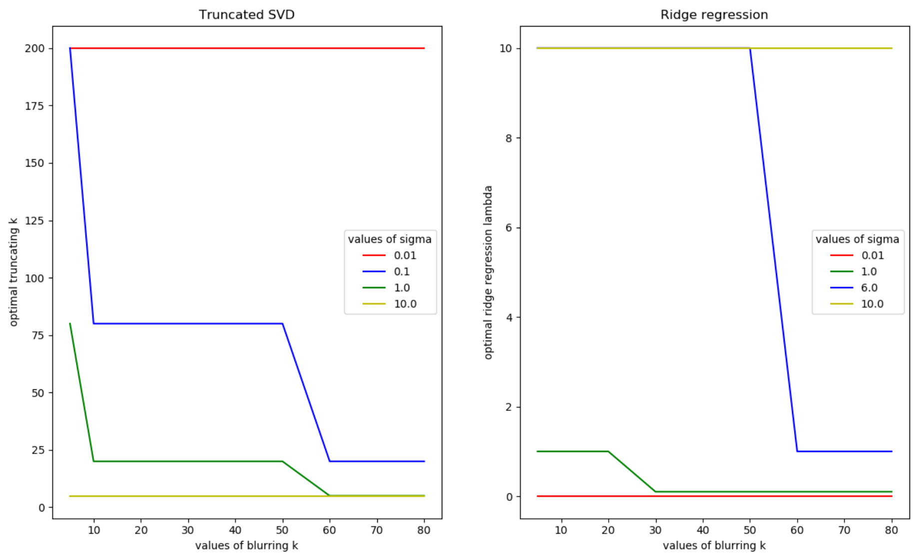
- Similarly to what we saw for truncated SVD, the optimal lambda gets smaller as the blurring k increases (for a constant sigma). As before, for extreme values of sigma the relation is constant.

- However,contrary to what we saw before, for a constant blurring k, the optimal lambda increases as sigma increases.

The code used to produce that graph is the following.

```
def calc_min_error_trunc(X, y):
    list_trunc = [5, 20, 80, 200]
    error_sq = []
    for k in list_trunc:
        SVD_trunc = calc_inverse_trunc(X, k)
        w_trunc = np.matmul(SVD_trunc, y)
        error_sq.append(sum((w_trunc-w)**2))
```

Figure 9:




```
    k_min_error = list_trunc[error_sq.index(min(error_sq))]
    return k_min_error


def calc_min_error_ridge(X, y):
    list_ridge = [0.01, 0.1, 1, 10]
    error_sq = []
    for k in list_ridge:
        SVD_ridge = calc_inverse_ridge(X, k)
        w_ridge = np.matmul(SVD_ridge, y)
        error_sq.append(sum((w_ridge-w)**2))
    k_min_error = list_ridge[error_sq.index(min(error_sq))]
    return k_min_error


k_list = [5, 10, 20, 30, 40, 50, 60, 80]
sigma = [0.000001, 0.0001, 0.01, 0.1, 1, 2, 6, 10]

opt_trunc = {s: {k: 0 for k in k_list} for s in sigma}
opt_ridge = {s: {k: 0 for k in k_list} for s in sigma}

for s in sigma:
    for k in k_list:
        h = np.ones(k) / k
        m = n + k - 1
        X = np.zeros((m, m))
```

```
        for i in range(m):
            if i < k:
                X[i, :i+1] = h[:i+1]
            else:
                X[i, i - k: i] = h
        X = X[:, 0:n]
        y = np.dot(X, w) + s*np.random.standard_normal(
    size=(m, 1))
        opt_trunc[s][k] = calc_min_error_trunc(X, y)
        opt_ridge[s][k] = calc_min_error_ridge(X, y)

trunc_data = pd.DataFrame(data=opt_trunc)
ridge_data = pd.DataFrame(data=opt_ridge)

sigma = [0.000001, 0.0001, 0.01, 0.1, 1, 2, 6, 10]

f, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_title('Truncated SVD')
ax1.plot(trunc_data[0.01], 'r')
ax1.plot(trunc_data[0.1], 'b')
ax1.plot(trunc_data[1], 'g')
ax1.plot(trunc_data[10], 'y')
ax1.legend(title='values of sigma')
ax1.set_xlabel('values of blurring k')
ax1.set_ylabel('optimal truncating k')
ax2.set_title('Ridge regression')
ax2.plot(ridge_data[0.01], 'r')
ax2.plot(ridge_data[1], 'g')
ax2.plot(ridge_data[6], 'b')
ax2.plot(ridge_data[10], 'y')
ax2.legend(title='values of sigma')
ax2.set_xlabel('values of blurring k')
ax2.set_ylabel('optimal ridge regression lambda')

plt.show()
```

4. Problem 4

   (a) Let

$$x = \begin{bmatrix} 1 \\ 1 \\ ... \\ 1 \end{bmatrix}$$

$$A^T x = \begin{bmatrix} a_1^T * 1 \\ a_2^T * 1 \\ ... \\ a_n^T * 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ ... \\ 1 \end{bmatrix}$$

   because each column of A sums to one. This is to say, $A^T x = x$
   So x is an eigenvector and 1 is an eigenvalue of A^T and A.

   (b) By inspection we observe that to keep the columns of G summing to one, the elements of vector u need to also sum to 1.

(c)

$$M = \begin{bmatrix} 1 & 1 & 1 & . & . & 1 \\ 0 & 0 & 0 & . & . & 0 \\ 0 & 0 & 0 & . & . & 0 \\ 0 & 0 & 0 & . & . & 0 \\ 0 & 0 & 0 & . & . & 0 \\ 0 & 0 & 0 & . & . & 0 \end{bmatrix}$$

$$G = \begin{bmatrix} \frac{(n-1)\alpha+1}{n} & \frac{(n-1)\alpha+1}{n} & . & . & . & \frac{(n-1)\alpha+1}{n} \\ \frac{(1-\alpha)}{n} & \frac{(1-\alpha)}{n} & . & . & . & \frac{(1-\alpha)}{n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ \frac{(1-\alpha)}{n} & \frac{(1-\alpha)}{n} & . & . & . & \frac{(1-\alpha)}{n} \end{bmatrix}$$

We calculate the eigenvector π as follows:

$$G\pi = \pi$$

which implies,

$$(\pi_1 + \pi_2 + ... + \pi_n)\frac{(n-1)\alpha + 1}{n} = \pi_1$$

and

$$(\pi_1 + \pi_2 + ... + \pi_n)\frac{(1-\alpha)}{n} = \pi_i$$

for all $i \neq 1$

Since $(\pi_1 + \pi_2 + ... + \pi_n) = 1$ the Pagerank of Facebook is:

$$\frac{(n-1)\alpha + 1}{n} = \pi_1$$

And the PageRank of the rest n-1 Web pages is:

$$\frac{(1-\alpha)}{n} = \pi_i$$

for all $i \neq 1$

We can check that the resulting probability vector sums to 1 as follows:

$$\frac{(n-1)\alpha + 1}{n} + (n-1)\frac{(1-\alpha)}{n} = \frac{(1-\alpha+\alpha)(n-1)+1}{n} = \frac{n}{n} = 1$$

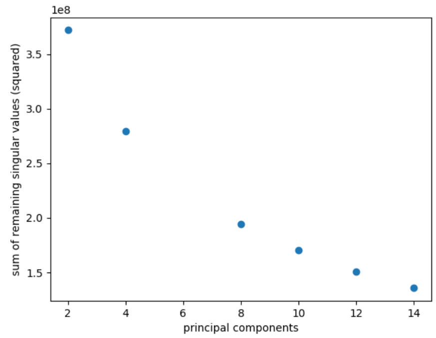
5. Problem 5

(a) Reconstruction accuracy The code used is the following:

```
import random
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


file_path = '/Users/teresamorales/Documents/Harris/MFML/Homework5/mnist.mat'
number_data = sio.loadmat(file_path)
train_data = number_data['train_data']
test_data = number_data['test_data']
y_train = number_data['train_target']
y_test = number_data['test_target']

u, s, vh = np.linalg.svd(train_data, full_matrices=False)

list_pc = [2, 4, 8, 10, 12, 14]

s_squared = [s[i]**2 for i in range(200)]

acc = {j: sum(s_squared) - sum([s_squared[i] for i in range(j)]) for j in list_pc}

accuracy = {'principal_components': list(acc.keys()),
            'reconstruction_accuracy': list(acc.values())}

accuracy_df = pd.DataFrame(data=accuracy)

fig, ax = plt.subplots()
ax.scatter(accuracy_df['principal_components'], accuracy_df['reconstruction_accuracy'])
ax.set_xlabel("principal components")
ax.set_ylabel("sum of remaining singular values (squared)")
plt.show()
```

The resulting plot is in figure 10:

Figure 10:



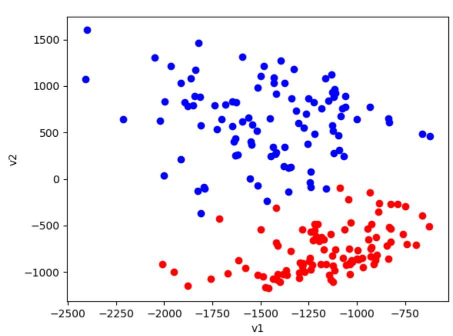
(b) Projection into 2d space

The code used is the following:

```
u_t = np.transpose(u)

X_p = np.matmul(np.diag(s[0:2]), u_t[0:2, :])

fig, ax = plt.subplots()
ax.scatter(X_p[0, 0:100, ], X_p[1, 0:100], color='r')
ax.scatter(X_p[0, 100:200], X_p[1, 100:200], color='b')
ax.set_xlabel("v1")
ax.set_ylabel("v2")
plt.show()
```

The resulting plot is the following

Figure 11:



(c) Linear regression

The code used is the following:

```python

def calc_weights(X, y):
    Xt = np.transpose(X)
    XtX = np.matmul(Xt, X)
    Inverse_XtX = np.linalg.inv(XtX)
    w = np.matmul(np.matmul(Inverse_XtX, Xt), y)
    return w


col1 = np.ones((200, 1))
X_p_constant = np.hstack((np.transpose(X_p), col1))

w_hat = calc_weights(X_p_constant, np.transpose(y_train))


# measure performance on training sample
y_hat_train = np.matmul(X_p_constant, w_hat)
error_train = np.subtract(y_train, y_hat_train)
error_train_squares = np.square(error_train)
avgerr_train = np.sqrt(np.mean(error_train_squares))

print('Mean squared error for training sample is', avgerr_train)

# measure performance on test sample

# project test data into the same subspace as the train data (using vh from train data )
X_test_p = np.matmul(vh[0:2, :], np.transpose(test_data))

X_test_constant = np.hstack((np.transpose(X_test_p), col1))
y_hat_test = np.matmul(X_test_constant, w_hat)  # Prediction for test data

error_test = np.subtract(y_test, y_hat_test)
error_test_squares = np.square(error_test)
avgerr_test = np.sqrt(np.mean(error_test_squares))

print('Mean squared error for test sample is', avgerr_test)
```

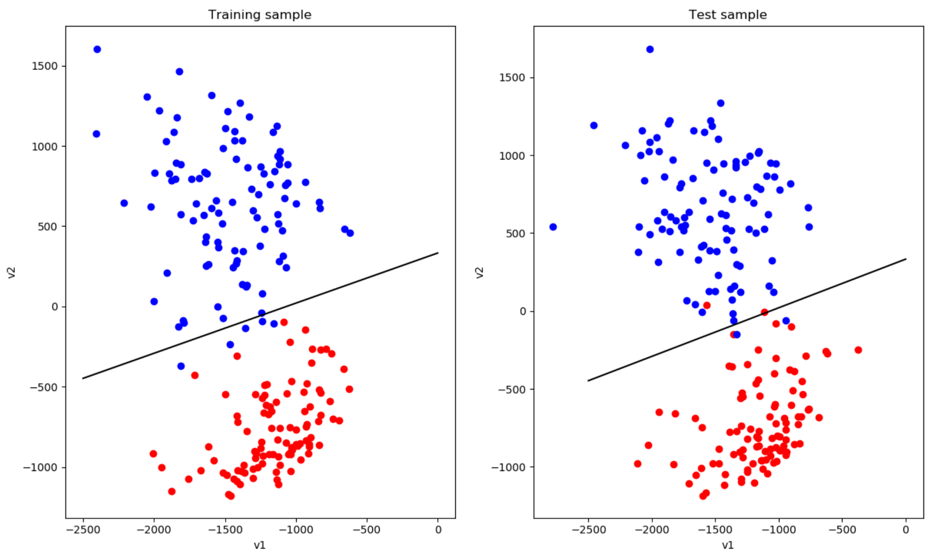
Results: Mean squared error for training sample is 1.3482749026367424

Mean squared error for test sample is 1.3528216855250104

(d) The code and plot would be the following:

```python
f, (ax1, ax2) = plt.subplots(1, 2)
ax1.scatter(X_p[0, 0:100, ], X_p[1, 0:100], color='r')
ax1.scatter(X_p[0, 100:200], X_p[1, 100:200], color='b')
ax1.plot([-2500, 0], [(-w_hat[0][0]*-2500/w_hat[1][0]) - w_hat[2]
                    [0]/w_hat[1][0], -w_hat[2][0]/w_hat[1][0]], color='black')
ax1.set_xlabel("v1")
ax1.set_ylabel("v2")
```

Figure 12:




```python
ax1.set_title('Training sample')
ax2.scatter(X_test_p[0, 0:100, ], X_test_p[1, 0:100], color='r')
ax2.scatter(X_test_p[0, 100:200], X_test_p[1, 100:200], color='b')
ax2.plot([-2500, 0], [(-w_hat[0][0]*-2500/w_hat[1][0]) - w_hat[2]
                    [0]/w_hat[1][0], -w_hat[2][0]/w_hat[1][0]], color='black')
ax2.set_xlabel("v1")
ax2.set_ylabel("v2")
ax2.set_title('Test sample')

plt.show()
```

6. Problem 6

   (a) Only two dimensions are necessary since the third singular value is 0 (very close to zero given rounding error)

```python
import random
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


```

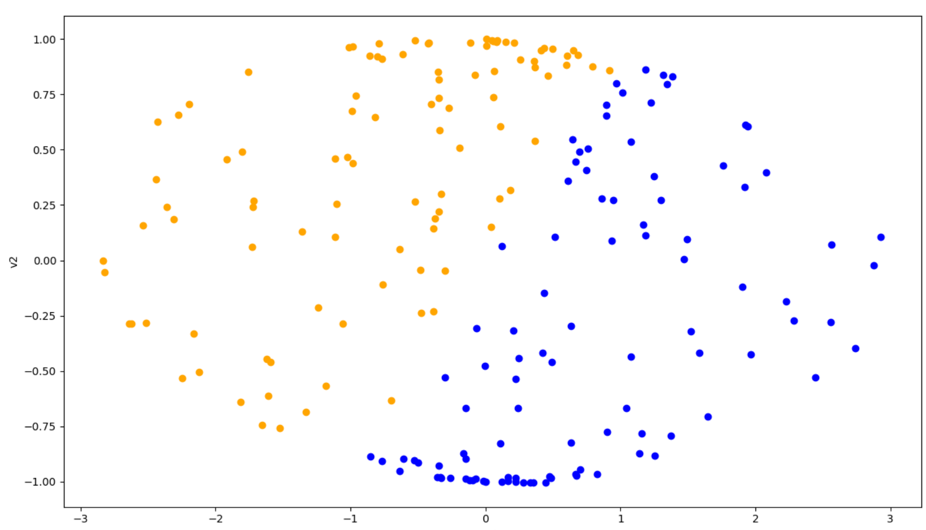
```
file_path = '/Users/teresamorales/Documents/Harris/MFML/Homework5/pca_3d.mat'
p6_data = sio.loadmat(file_path)
X = p6_data['point']
y = p6_data['target']

u, s, vh = np.linalg.svd(X, full_matrices=True)
```

(b) Projection into 2 and 1 dimensional spaces For the projection onto a 2 dimensional space we would use the following code:

```
u_t = np.transpose(u)
X_p = np.matmul(np.diag(s[0:2]), u_t[0:2, :])

#graph
value_1 = np.squeeze(y == 1)
value_0 = np.logical_not(value_1)

fig, ax = plt.subplots()
ax.scatter(X_p[0, value_1], X_p[1, value_1], color='orange')
ax.scatter(X_p[0, value_0], X_p[1, value_0], color='b')
ax.set_xlabel("v1")
ax.set_ylabel("v2")
plt.show()
```

The resulting plot is the following:

Figure 13:

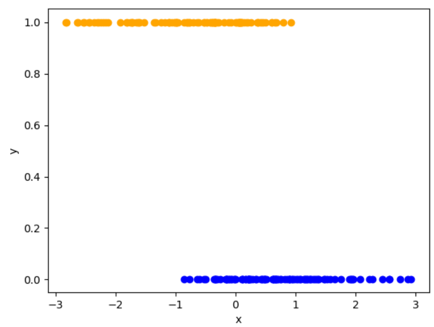


For one dimensions the code would be the following:

```
X_p = np.matmul(np.diag(s[0:1]), u_t[0:1, :])
```

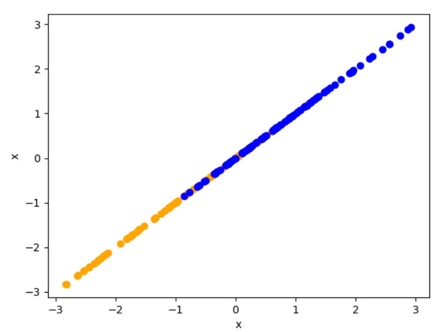
In order to graph this new projection we represent the features in the x axis and the labels in the y axis.

Figure 14:



Another possible way to represent the subspace is plotting the features against their own.

Figure 15:



(c) After whitening the features matrix, the resulting plot is the same as in part b for two dimensions, except for the scale in the axes.

```python
W_pca = np.matmul(np.diag(1/s[0:2]), vh[0:2, :])

New_X = np.matmul(X, np.transpose(W_pca))


def check_diag(X_tilda):
    return np.matmul(np.transpose(X_tilda), X_tilda)


fig, ax = plt.subplots()
ax.scatter(New_X[value_1, 0], New_X[value_1, 1], color='orange')
ax.scatter(New_X[value_0, 0], New_X[value_0, 1], color='b')
ax.set_xlabel("v1")
ax.set_ylabel("v2")
plt.show()
```

Figure 16: