Homework 1: PCA

Problem 1 - Principal Component Analysis

In this problem you'll be implementing Dimensionality reduction using Principal Component Analysis technique.

The gist of PCA Algorithm to compute principal components is follows:

- Calculate the covariance matrix X of data points.
- Calculate eigenvectors and corresponding eigenvalues.
- Sort the eigenvectors according to their eigenvalues in decreasing order.
- Choose first k eigenvectors which satisfies target explained variance.
- Transform the original data of shape m observations times n features into m observations times k selected features.

The skeleton for the *PCA* class is below. Scroll down to find more information about your tasks.

In [1]:
```
#!pip install pytest
#!python3 -m pip install --upgrade pip
```

In [2]:
```
import math
import pickle
import gzip
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
import pytest
from sklearn.preprocessing import StandardScaler
%matplotlib inline
```

In [3]:

```python
class PCA:
    def __init__(self, target_explained_variance=None):
        """
        explained_variance: float, the target level of explained variance
        """
        self.target_explained_variance = target_explained_variance
        self.feature_size = -1

    def standardize(self, X):
        """
        standardize features using standard scaler
        :param X: input data with shape m (# of observations) X n (# of f
eatures)
        :return: standardized features (Hint: use skleanr's StandardScale
r. Import any library as needed)
        """
        # your code here
        scaler = StandardScaler()
        X_std = scaler.fit_transform(X)
        return X_std

    def compute_mean_vector(self, X_std):
        """
        compute mean vector
        :param X_std: transformed data
        :return n X 1 matrix: mean vector
        """
        # your code here
        mean_vec = np.mean(X_std, axis=0)
        return mean_vec

    def compute_cov(self, X_std, mean_vec):
        """
        Covariance using mean, (don't use any numpy.cov)
        :param X_std:
        :param mean_vec:
        :return n X n matrix:: covariance matrix
        """
        # your code here
        m = X_std.shape[0]
        cov_mat = (X_std - mean_vec).T.dot(X_std - mean_vec) / (m - 1)
        return cov_mat

    def compute_eigen_vector(self, cov_mat):
        """
        Eigenvector and eigen values using numpy. Uses numpy's eigenvalue
function
        :param cov_mat:
        :return: (eigen_values, eigen_vector)
        """
        # your code here
        eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
        return eigen_vals, eigen_vecs

    def compute_explained_variance(self, eigen_vals):
        """
        sort eigen values and compute explained variance.
        explained variance informs the amount of information (variance)
        can be attributed to each of  the principal components.
```

```
        :param eigen_vals:
        :return: explained variance.
        """
        # your code here
        total = sum(eigen_vals)
        var_exp = [(i / total) for i in sorted(eigen_vals, reverse=True)]
        return np.array(var_exp)

    def cumulative_sum(self, var_exp):
        """
        return cumulative sum of explained variance.
        :param var_exp: explained variance
        :return: cumulative explained variance
        """
        return np.cumsum(var_exp)

    def compute_weight_matrix(self, eig_pairs, cum_var_exp):
        """
        compute weight matrix of top principal components conditioned on
target
        explained variance.
        (Hint : use cumilative explained variance and target_explained_va
riance to find
        top components)

        :param eig_pairs: list of tuples containing eigenvalues and eigen
vectors,
        sorted by eigenvalues in descending order (the biggest eigenvalue
and corresponding eigenvectors first).
        :param cum_var_exp: cumulative expalined variance by features
        :return: weight matrix (the shape of the weight matrix is n X k)
        """
        # your code here
        matrix_w = np.ones((self.feature_size, 1))
        for i in range(len(eig_pairs)):
            if cum_var_exp[i] < self.target_explained_variance:
                matrix_w = np.hstack((matrix_w,
                                      eig_pairs[i][1].reshape(self.featur
e_size,
                                                              1)))
        return np.delete(matrix_w, [0], axis=1).tolist()


    def transform_data(self, X_std, matrix_w):
        """
        transform data to subspace using weight matrix
        :param X_std: standardized data
        :param matrix_w: weight matrix
        :return: data in the subspace
        """
        return X_std.dot(matrix_w)

    def fit(self, X):
        """
        entry point to the transform data to k dimensions
        standardize and compute weight matrix to transform data.
        The fit functioin returns the transformed features. k is the numb
er of features which cumulative
        explained variance ratio meets the target_explained_variance.
        :param   m X n dimension: train samples
```

```
        :return  m X k dimension: subspace data.
        """

        self.feature_size = X.shape[1]

        # your code here

        # Standardize the data
        X_std = self.standardize(X)

        # Compute the mean vector
        mean_vec = self.compute_mean_vector(X_std)

        # Compute the covariance matrix
        cov_mat = self.compute_cov(X_std, mean_vec)

        # Compute the eigenvalues and eigenvectors
        eigen_vals, eigen_vecs = self.compute_eigen_vector(cov_mat)

        # Compute the explained variance
        var_exp = self.compute_explained_variance(eigen_vals)

        # Compute the cumulative explained variance
        cum_var_exp = self.cumulative_sum(var_exp)

        # Combine eigenvalues and eigenvectors into pairs
        eig_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i]) for i in r
ange(len(eigen_vals))]
        eig_pairs.sort(key=lambda x: x[0], reverse=True)

        # Compute the weight matrix
        matrix_w = self.compute_weight_matrix(eig_pairs, cum_var_exp)

        # Transform the data
        return self.transform_data(X_std, matrix_w)
```

[PART A] Your task involves implementing helper functions to compute *mean, covariance, eigenvector and weights*.

complete `fit()` to using all helper functions to find reduced dimension data.

Run PCA on *fashion mnist dataset* to reduce the dimension of the data.

fashion mnist data consists of samples with *784 dimensions*.

Report the reduced dimension k for target explained variance of **0.99**

In [4]:
```
X_train = pickle.load(open('./data/fashionmnist/train_images.pkl','rb'))
y_train = pickle.load(open('./data/fashionmnist/train_image_labels.pk
l','rb'))

X_train = X_train[:1500]
y_train = y_train[:1500]
```

In [5]:
```
pca_handler = PCA(target_explained_variance=0.99)
X_train_updated = pca_handler.fit(X_train)
```

In [6]:
```
X_train_updated
```

Out[6]:
```
array([[-1.06925912e+01, -1.54564095e+01, -4.82051696e+00, ...,
        -1.77897033e-01,  2.64530516e-01, -1.12071758e-01],
       [ 1.21514911e+01, -1.22518345e+01, -6.11859830e+00, ...,
        -1.82124001e-01,  2.21649909e-02, -7.39387984e-02],
       [-2.06815612e+01, -1.70920686e+00,  4.78189888e+00, ...,
         1.69850158e-03, -2.21422945e-01,  1.28143482e-01],
       ...,
       [ 8.94357946e+00, -1.37147893e+01, -7.20371277e+00, ...,
        -1.41701336e-01,  8.80752645e-02, -2.35743279e-01],
       [ 6.57779547e-01, -1.09376244e+01, -5.77985498e+00, ...,
         1.95382327e-01, -1.03141430e+00, -2.71609584e-01],
       [ 1.68946454e+01, -6.28213604e+00, -2.53667838e+00, ...,
        -1.73813696e-01,  1.58667787e-01, -4.72641358e-01]])
```

Sample Testing of implemented functions

Use the below cells one by one to test each of your functions implemented above.
This should serve handy for debugging sections of your code

Please Note - The hidden tests on which the actual points are awarded are different from these and usually run on a different, more complex dataset

In [7]:
```
np.random.seed(42)
X = np.array([[0.39, 1.07, 0.06, 0.79], [-1.15, -0.51, -0.21, -0.7],
[-1.36, 0.57, 0.37, 0.09], [0.06, 1.04, 0.99, -1.78]])
pca_handler = PCA(target_explained_variance=0.99)
```

In [8]:
```
X_std_act = pca_handler.standardize(X)

X_std_exp = [[ 1.20216033, 0.82525828, -0.54269609, 1.24564656],
             [-0.84350476, -1.64660539, -1.14693504, -0.31402854],
             [-1.1224591, 0.04302294, 0.15105974, 0.51291329],
             [ 0.76380353, 0.77832416, 1.53857139, -1.4445313]]

for act, exp in zip(X_std_act, X_std_exp):
    assert pytest.approx(act, 0.01) == exp, "Check Standardize function"
```

In [9]:
```
mean_vec_act = pca_handler.compute_mean_vector(X_std_act)

mean_vec_exp = [5.55111512, 2.77555756, 5.55111512, -5.55111512]

mean_vec_act_tmp = mean_vec_act * 1e17

assert pytest.approx(mean_vec_act_tmp, 0.1) == mean_vec_exp, "Check compu
te_mean_vector function"
```

In [10]:
```
cov_mat_act = pca_handler.compute_cov(X_std_act, mean_vec_act)

cov_mat_exp = [[ 1.33333333, 0.97573583, 0.44021511, 0.02776305],
 [ 0.97573583, 1.33333333, 0.88156376, 0.14760488],
 [ 0.44021511, 0.88156376, 1.33333333, -0.82029039],
 [ 0.02776305, 0.14760488, -0.82029039, 1.33333333]]

assert pytest.approx(cov_mat_act, 0.01) == cov_mat_exp, "Check compute_co
v function"
```

In [11]:
```
eig_vals_act, eig_vecs_act = pca_handler.compute_eigen_vector(cov_mat_ac
t)

eig_vals_exp = [2.96080083e+00, 1.80561744e+00, 5.66915059e-01, 7.8690727
6e-17]

eig_vecs_exp = [[ 0.50989282,  0.38162981,  0.72815056,  0.25330765],
 [ 0.59707545,  0.33170546, -0.37363029, -0.62759286],
 [ 0.57599397, -0.37480162, -0.41446394,  0.59663585],
 [-0.22746684,  0.77708038, -0.3980161,   0.43126337]]

assert pytest.approx(eig_vals_act, 0.01) == eig_vals_exp, "Check compute_
eigen_vector function"

for act, exp in zip(eig_vecs_act, eig_vecs_exp):
    assert pytest.approx(act, 0.01) == exp, "Check compute_eigen_vector f
unction"
```

In [12]:
```
pca_handler.feature_size = X.shape[1]
var_exp_act = pca_handler.compute_explained_variance(eig_vals_act)

var_exp_exp = [0.5551501556710813, 0.33855327084133857, 0.106296573487580
19, 1.475451142706682e-17]

assert pytest.approx(var_exp_act, 0.01) == var_exp_exp, "Check compute_ex
plained_variance function"
```

In [13]:
```
eig_pairs = np.array([(2.9608008302457662, np.array([ 0.50989282,  0.5970
7545,  0.57599397, -0.22746684])),
(1.8056174444871387, np.array([ 0.38162981,  0.33170546, -0.37480162,
0.77708038])),
(0.5669150586004276, np.array([ 0.72815056, -0.37363029, -0.41446394,
-0.3980161 ])),
(7.869072761102302e-17, np.array([ 0.25330765, -0.62759286,  0.59663585,
0.43126337]))])

cum_var_exp = [0.55515016, 0.89370343, 1, 1]

matrix_w_exp = [[0.50989282, 0.38162981],
                [0.59707545, 0.33170546],
                [0.57599397, -0.37480162],
                [-0.22746684, 0.77708038]]

matrix_w_act = pca_handler.compute_weight_matrix(eig_pairs=eig_pairs, cum
_var_exp=cum_var_exp)

for act, exp in zip(matrix_w_act, matrix_w_exp):
    assert pytest.approx(act, 0.001) == exp, "Check compute_weight_matrix
function"
```

Result Comparison with Sklearn

The below cells should help you compare the output from your implementation against the sklearn implementation with a similar configuration. This is solely to help you validate your work.

In [14]:
```
# Use this cell to verify against the sklearn implementation given in the
next cell
pca_handler.transform_data(X_std=X_std_act, matrix_w=matrix_w)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call la
st)
<ipython-input-14-72133c3c3d82> in <module>
      1 # Use this cell to verify against the sklearn implementation giv
en in the next cell
----> 2 pca_handler.transform_data(X_std=X_std_act, matrix_w=matrix_w)

NameError: name 'matrix_w' is not defined
```

In [15]:
```
# Sklearn implementation to compare your results

# import all libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.decomposition import PCA as pca1
from sklearn.preprocessing import StandardScaler

# Scale data before applying PCA
scaling=StandardScaler()

# Use fit and transform method
# You may change the variable X if needed to verify against a different d
ataset
print("Sample data:", X)
scaling.fit(X)
Scaled_data=scaling.transform(X)
print("\nScaled data:", Scaled_data)

# Set the n_components=3
principal=pca1(n_components=2)
principal.fit(Scaled_data)
x=principal.transform(Scaled_data)

# Check the dimensions of data after PCA
print("\nTransformed Data",x)
```

```
Sample data: [[ 0.39  1.07  0.06  0.79]
 [-1.15 -0.51 -0.21 -0.7 ]
 [-1.36  0.57  0.37  0.09]
 [ 0.06  1.04  0.99 -1.78]]

Scaled data: [[ 1.20216033  0.82525828 -0.54269609  1.24564656]
 [-0.84350476 -1.64660539 -1.14693504 -0.31402854]
 [-1.1224591   0.04302294  0.15105974  0.51291329]
 [ 0.76380353  0.77832416  1.53857139 -1.4445313 ]]

Transformed Data [[ 0.50978142  1.90389378]
 [-2.00244127 -0.68224688]
 [-0.57630715 -0.07213548]
 [ 2.068967   -1.14951141]]
```

In [ ]: