# Lab 5: Eigen-decomposition of Images

---

Course Title: Image Processing I (Spring 2022)

Course Number: ECE 63700

Instructor: Prof. Charles A. Bouman

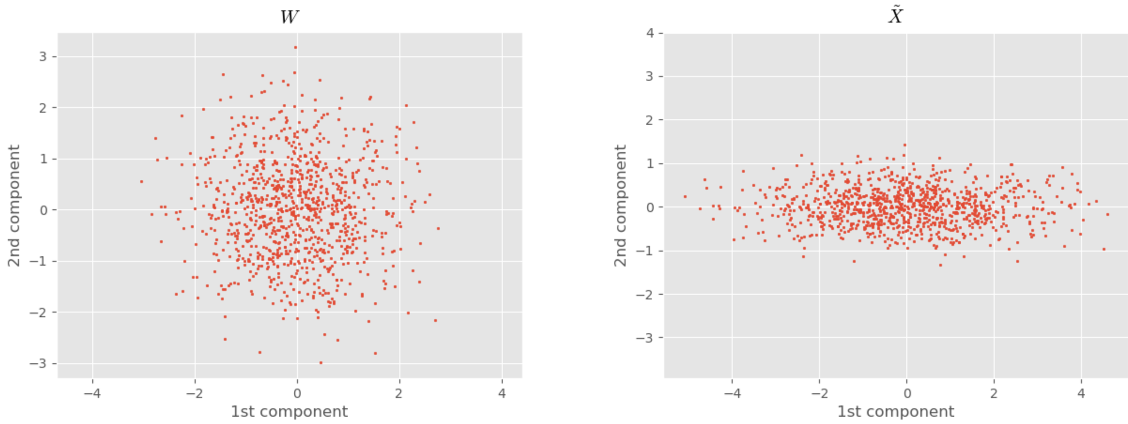
Author: **Zhankun Luo**

# 2. Multivariate Gaussian Distributions and Whitening

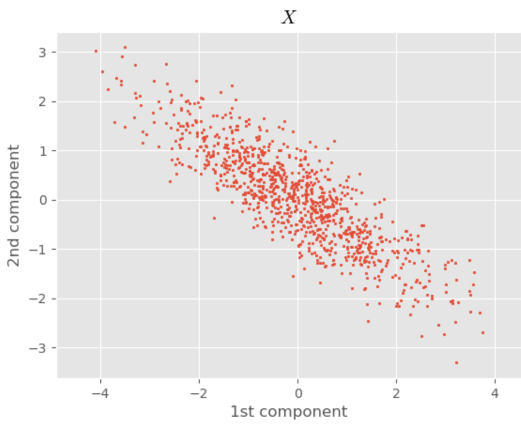
## 2.1. Exercise: Generating Gaussian random vectors

### solution

the scatter plots for $W$, $\tilde{X}$, and $X$




*Scatter Plots for* **W***(left), and Scaled Random Vectors (right)*

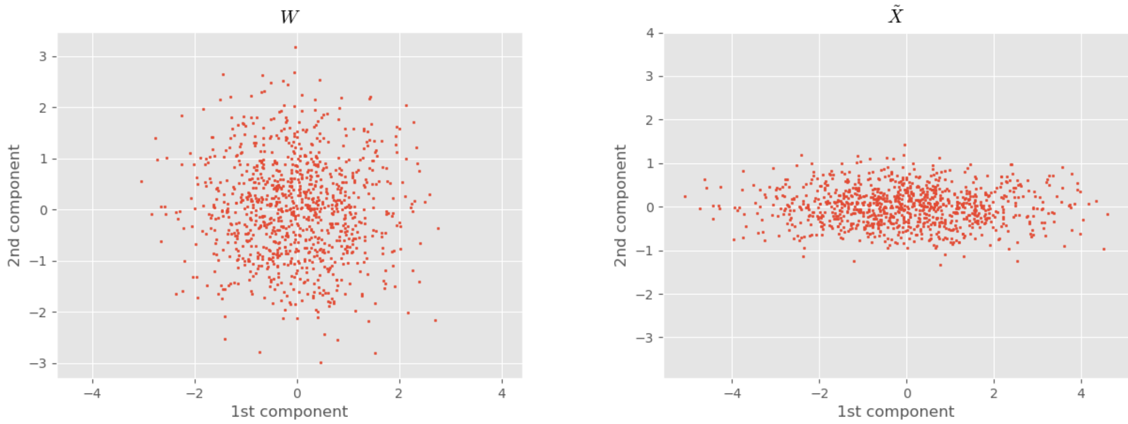



*Scatter Plot for Generated* **X**

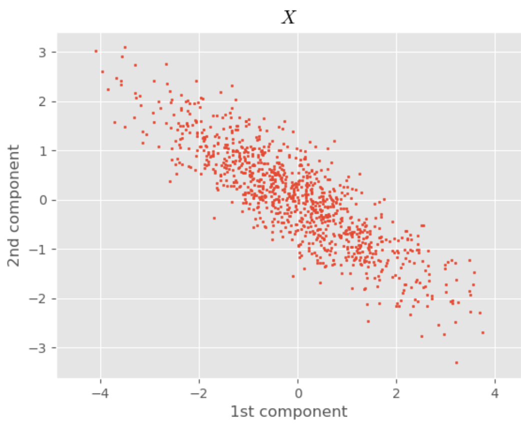
## 2.2. Exercise: Covariance Estimation and Whitening

**solution**

the scatter plots for $W$, $\tilde{X}$, and $X$




*Scatter Plots for Whitened Random Vector* **W***(left), and Uncorrelated Random Vectors (right)*

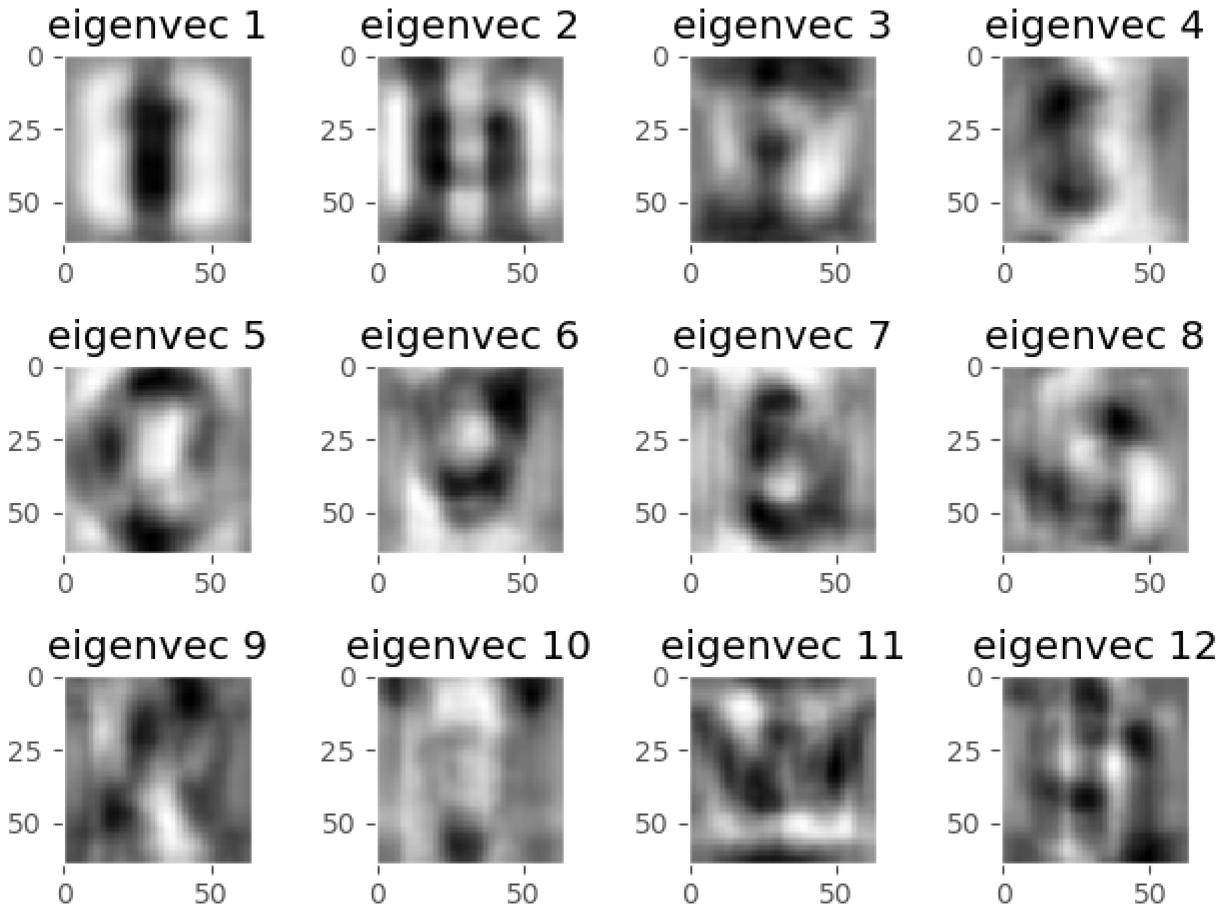



*Scatter Plot for Original Gaussian Random Vector* **X**

# 4. Eigenimages, PCA, and Data Reduction

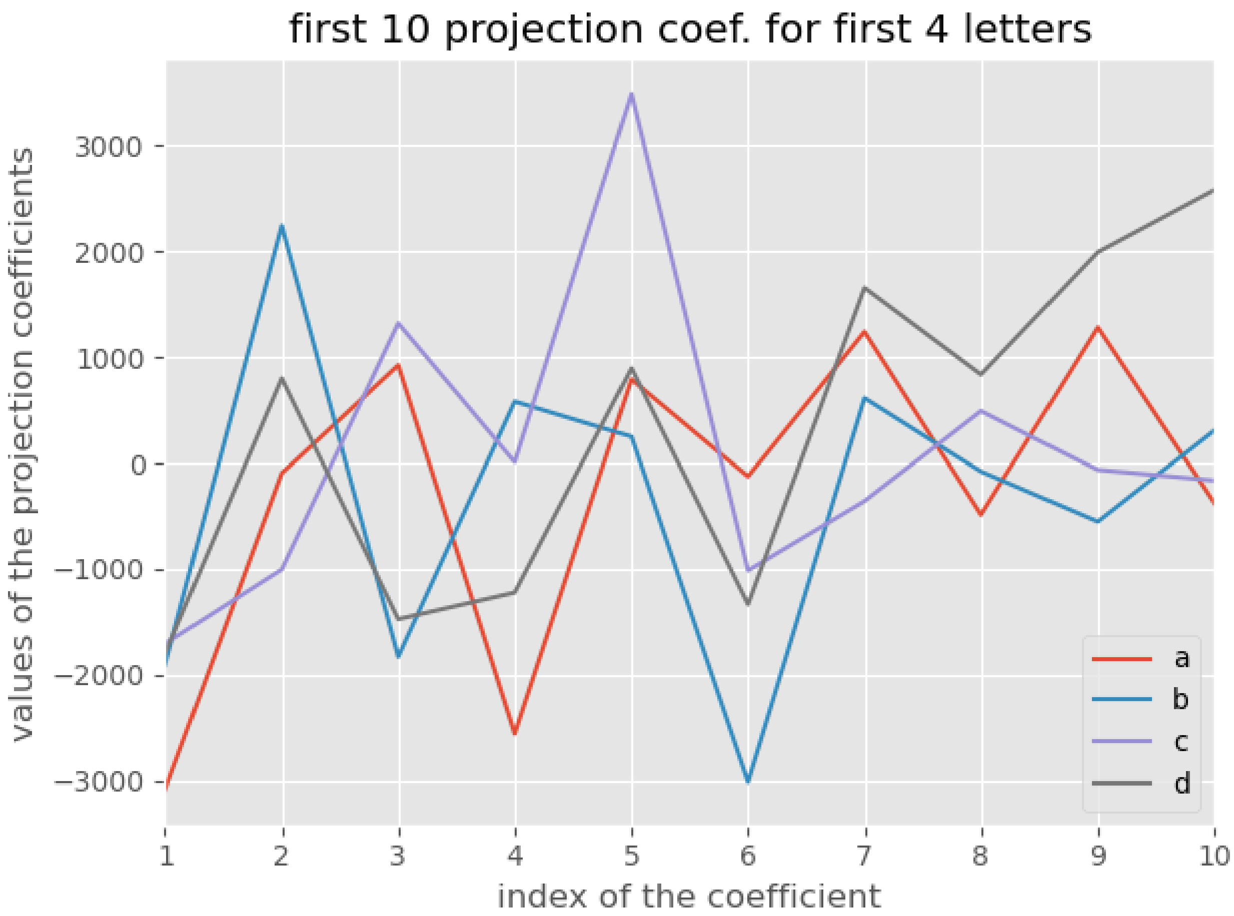
## 4.1. Exercise

### 4.1.1. the figure with the first $12$ eigen-images



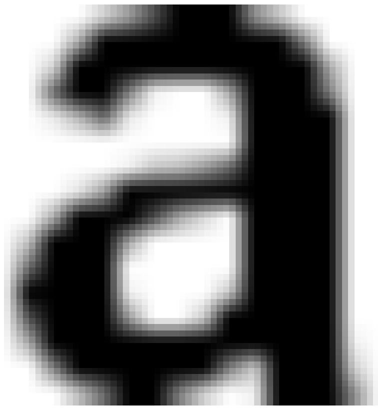

*Eigen-images for the 12 Largest Eigenvalues of Estimated* $\mathbf{R}$ *on the Training Dataset*

### 4.1.2. plots of projection coef. vs. eigenvector number for first 4 images

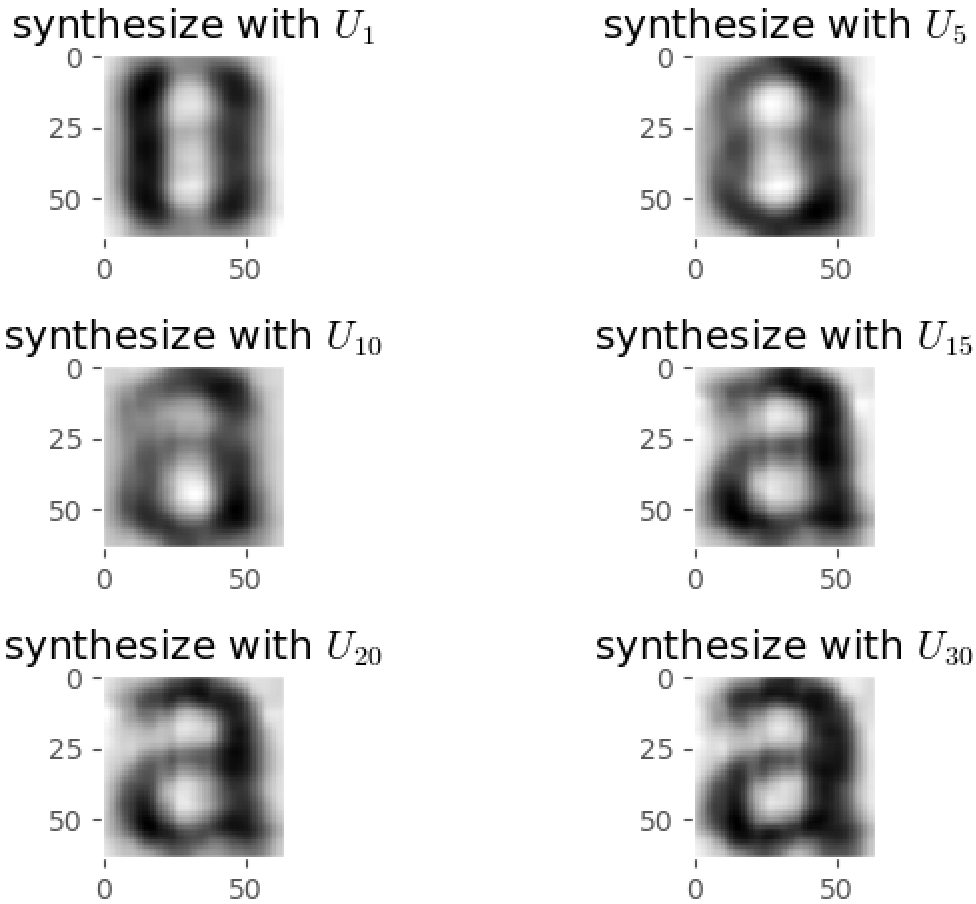



*First 10 Projection Coefficients for First 4 Images in the Training Dataset*

### 4.1.3. the 6 synthesized images for 1st image



*Original Image for 1st Image in the Training Dataset*




*Synthesized Images with first* **m=1, 5, 10, 15, 20, 30**
*Eigenvectors for 1st Image in the Training Dataset*

# 5. Image Classification

## 5.1. Exercise: Classification and PCA

### 5.1.1. table for each mis-classified input image when $B_k = R_k$

| INPUT CHARACTER | OUTPUT FROM THE CLASSIFIER |
|---|---|
| d | a |
| j | y |
| l | i |
| n | v |
| p | e |
| q | a |
| u | a |
| y | v |

### 5.1.2. table for each mis-classified input image when $B_k = \Lambda_k \equiv \mathrm{diag}(R_k)$

| INPUT CHARACTER | OUTPUT FROM THE CLASSIFIER |
|---|---|
| i | l |
| y | v |

### 5.1.3. table for each mis-classified input image when $B_k = R_{wc} \equiv \frac{1}{K} \sum_k R_k$

| INPUT CHARACTER | OUTPUT FROM THE CLASSIFIER |
|---|---|
| g | q |
| y | v |

### 5.1.4. table for each mis-classified input image when $B_k = \Lambda \equiv \mathrm{diag}(R_{wc})$

| INPUT CHARACTER | OUTPUT FROM THE CLASSIFIER |
|---|---|
| f | t |
| y | v |

### 5.1.5. table for each mis-classified input image when $B_k = I$

| INPUT CHARACTER | OUTPUT FROM THE CLASSIFIER |
|---|---|
| f | t |
| g | q |
| y | v |

### 5.1.6. which of the above classifiers worked the best

when $B_k = \Lambda_k \equiv \mathrm{diag}(R_k)$, ,$B_k = R_{wc} \equiv \frac{1}{K}\sum_k R_k$ or $B_k = \Lambda \equiv \mathrm{diag}(R_{wc})$

the error rate is lowest $\frac{2}{26} \times 100\%$, the accuracy reaches $\frac{24}{26} \times 100\% \approx 92.3\%$

### 5.1.7. trade-off between accuracies of the data model and the estimates

Even thought the classifier for $B_k = R_k$ has the best estimate for covariances, it has the worst accuracy for the classifier.

Even $R_k = I$ is better than it, but $B_k = \Lambda_k \equiv \mathrm{diag}(R_k)$, ,$B_k = R_{wc} \equiv \frac{1}{K}\sum_k R_k$ or $B_k = \Lambda \equiv \mathrm{diag}(R_{wc})$ have the best accuracy.

We should balance the data parameter estimation and the model complexity.

# Appendix

## Python codes for functions

### utils.py

```
from typing import Tuple, List
from string import ascii_lowercase as alphabet
import os
from os.path import join
from math import sqrt, floor
from numpy import ndarray, zeros, array
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['mathtext.fontset'] = 'cm'
plt.style.use('ggplot')

def plot_data2d(data: ndarray, str_title: str) -> None:
    plt.scatter(data[0], data[1], s=3)
    plt.grid(color='w')
    plt.xlabel("1st component")
    plt.ylabel("2nd component")
    plt.title(str_title)
    plt.axis("equal")

def read_data(dir_data: str, height: int=64, width: int=64) ->
Tuple[ndarray, List[str]]:
    _, folders, _ = list(os.walk(dir_data))[0]
    folders.sort()
    data = zeros((len(folders), len(alphabet), height, width))
    for i, folder in enumerate(folders):
        for j, char in enumerate(alphabet):
            data[i][j] = array(Image.open(join(dir_data, folder,
char+'.tif')))
    return data, folders

def get_divisor(x: int) -> Tuple[int, int]:
    t = floor(sqrt(x)+1e-6)
    while x%t != 0:
```

```
        t -= 1
    return t, x//t

def display_samples(data: ndarray, folders: List[str], char: str) ->
None:
    index = ord(char) - ord(alphabet[0])
    rows, cols = get_divisor(len(folders))
    fig, axs = plt.subplots(rows, cols)
    axs = [axs] if rows == 1 else axs
    axs = [[e] for e in axs] if cols == 1 else axs
    for i, folder in enumerate(folders):
        row, col = i//cols, i%cols
        axs[row][col].imshow(data[i][index],cmap=plt.cm.gray,
interpolation='none')
        axs[row][col].set_title(folder)
        axs[row][col].grid(False)
    plt.tight_layout()

def display_scaled(eigenvecs: ndarray, height: int=64, width: int=64)
-> None:
    assert(height*width == eigenvecs.shape[0])
    num_eigen = eigenvecs.shape[-1]
    rows, cols = get_divisor(num_eigen)
    fig, axs = plt.subplots(rows, cols)
    axs = [axs] if rows == 1 else axs
    axs = [[e] for e in axs] if cols == 1 else axs
    for i, vec in enumerate(eigenvecs.T):
        row, col = i//cols, i%cols
        v_max, v_min = vec.max(), vec.min()
        vec = (vec - v_min) / (v_max - v_min)
        axs[row][col].imshow(vec.reshape((height, width)),
cmap=plt.cm.gray, interpolation='none')
        axs[row][col].set_title("eigenvec {}".format(i+1))
        axs[row][col].grid(False)
    plt.tight_layout()

def plot_projection(Y: ndarray, range_coef: int=10) -> None:
    num_char = Y.shape[-1]
    for char, projection in list(zip(alphabet[:num_char], Y.T)):
        plt.plot(range(1, range_coef+1), projection[:range_coef],
label=char)
    plt.legend()
    plt.xlim([1, range_coef])
```

```
    plt.title("first {} projection coef. for first {}
letters".format(range_coef, num_char))
    plt.xlabel("index of the coefficient")
    plt.ylabel("values of the projection coefficients")
    plt.tight_layout()

def display_synthesized(vec_img: ndarray, mu_hat: ndarray,
    U: ndarray, list_num_eigen: list,
    height: int=64, width: int=64) -> None:
    assert(height*width == vec_img.shape[0])
    cols, rows = get_divisor(len(list_num_eigen))
    fig, axs = plt.subplots(rows, cols)
    axs = [axs] if rows == 1 else axs
    axs = [[e] for e in axs] if cols == 1 else axs
    for i, num_eigen in enumerate(list_num_eigen):
        U_top = U[:, :num_eigen]
        row, col = i//cols, i%cols
        Y = U_top.conj().T @ (vec_img - mu_hat)
        vec_img_syn = U_top @ Y + mu_hat
        axs[row][col].imshow(vec_img_syn.reshape((height, width)),
cmap=plt.cm.gray, interpolation='none')
        axs[row][col].set_title(r"synthesize with
$U_{{{}}}$".format(num_eigen))
        axs[row][col].grid(False)
    plt.tight_layout()

if __name__ == "__main__":
    print(len(alphabet))
    data_train, folders_train = read_data("resource/training_data")
    print(data_train.shape)
    data_test,  folders_test  = read_data("resource/test_data")
    print(data_test.shape)
    display_samples(data_train, folders_train, char='a')
    plt.show()
    display_samples(data_test, folders_test, char='a')
    plt.show()
```

**eigen_decompose.py**

```
from typing import Union, Tuple
from string import ascii_lowercase as alphabet
from numpy import sqrt, diag, ndarray, reshape, empty, zeros, log
from numpy.linalg import eig, svd, det, inv
from numpy.random import seed, randn

def generate_data(R: ndarray, mu: Union[None, ndarray]=None,
    N: int=1000, se=0) -> ndarray:
    """Generate N data samples for a Guassian distribution cluster
    V[:,i] is the eigenvector corresponding to the eigenvalue D[i]
    if W ~ N(0, I), X = sqrt(D) W
        => X ~ N(0, D),
    if X ~ N(0, D), Y = V X
        => Y ~ N(0, V D V^T)
    thus Y = V sqrt(D) W
        => Y ~ N(0, R)
        with decomposition: R = V D V^T
    """
    seed(se)
    dimension = R.shape[0]
    D, V = eig(R)
    if mu is None:
        return V @ diag(sqrt(D)) @ randn(dimension, N)
    return V @ diag(sqrt(D)) @ randn(dimension, N) + mu

def estimate_param(X: ndarray) -> Tuple[ndarray, ndarray]:
    N, mu = X.shape[-1], X.mean(axis=-1, keepdims=True)
    R = ((X - mu) @ (X - mu).T) / (N-1)
    return R, mu

def decorrelate_data(X: ndarray, R: ndarray, mu: Union[None,
ndarray]=None) -> ndarray:
    if mu is not None:
        X -= mu
    D, V = eig(R)
    return V.T @ X

def whiten_data(X: ndarray, R: ndarray, mu: Union[None,
ndarray]=None) -> ndarray:
    if mu is not None:
```

```
        X -= mu
    D, V = eig(R)
    return diag(1./sqrt(D)) @ (V.T @ X)

def svd_data(X: ndarray) -> Tuple[ndarray, list, ndarray]:
    """X = U diag(D) V^H = U @ np.diag(D) @ Vh = (U * D) @ Vh
    note: The rows of Vh are the eigenvectors of A^H A
        the columns of U are the eigenvectors of A A^H
        Vector(s) with the singular values sorted in descending order
    If full_matrices=True (default),
        U and Vh have the shapes (..., M, M) and (..., N, N)
    Otherwise, full_matrices=False,
        the shapes are (..., M, K) and (..., K, N)
        where K = min(M, N)
    """
    U, D, Vh = svd(X, full_matrices=False)
    return U, D, Vh.conj().T


class Classifier_PCA:
    def __init__(self, num_eigen: int=10):
        self.num_folder = None
        self.alphabet = alphabet
        self.num_char = None
        self.num_eigen = num_eigen
        self.height = None
        self.width = None
        self.mean_global = None
        self.A = None
        self.params = None
        self.R_wc = None

    def fit(self, data_train: ndarray):
        self.num_folder, self.num_char, \
            self.height, self.width = data_train.shape
        X = reshape(data_train, (self.num_folder*self.num_char,
self.height*self.width)).T
        self.mean_global = X.mean(axis=-1, keepdims=True)
        U, D, V = svd_data((X - self.mean_global) /
sqrt(self.num_folder*self.num_char-1))
        self.A = U[:, :self.num_eigen].conj().T
        self.params = [None] * self.num_char
        self.R_wc = zeros((self.num_eigen, self.num_eigen))
```

```python
        for ind, char in enumerate(self.alphabet[:self.num_char]):
            x = data_train[:, ind].reshape(self.num_folder, -1).T
            y = self.A @ (x - self.mean_global)
            R, mu = estimate_param(y)
            self.params[ind] = {"mean": mu, "cov": R,
                                "invcov": inv(R),
                                "logdetcov": log(det(R))}
            self.R_wc += R
        self.R_wc /= self.num_char
        self.invR_wc = inv(self.R_wc)

    def predict(self, data_test: ndarray,
                option: Union[str, None]=None):
        shape_data = list(data_test.shape)
        X = reshape(data_test, (-1, self.height*self.width)).T
        Y = self.A @ (X - self.mean_global)
        pred = empty((Y.shape[-1],), dtype='str')
        pred[:] = ' '
        if option is None or option == "default":
            f = lambda y, i: \
                (y-self.params[i]["mean"]).T \
                @ self.params[i]["invcov"] \
                @ (y-self.params[i]["mean"]) \
                + self.params[i]["logdetcov"]
        elif option == "diag":
            f = lambda y, i: \
                (y-self.params[i]["mean"]).T \
                @ inv(diag(diag(self.params[i]["cov"]))) \
                @ (y-self.params[i]["mean"]) \
                + log(det(diag(diag(self.params[i]["cov"]))))
        elif option == "global":
            f = lambda y, i: \
                (y-self.params[i]["mean"]).T \
                @ self.invR_wc \
                @ (y-self.params[i]["mean"])
        elif option == "global diag":
            f = lambda y, i: \
                (y-self.params[i]["mean"]).T \
                @ inv(diag(diag(self.R_wc))) \
                @ (y-self.params[i]["mean"])
        elif option == "identity":
            f = lambda y, i: \
                (y-self.params[i]["mean"]).T \
```

```
                @ (y-self.params[i]["mean"])
        else:
            print("error")
        for ind, y in enumerate(Y.T):
            y = y.reshape((-1, 1))
            id_pred = min(range(self.num_char), key=lambda i: f(y,
i))
            pred[ind] = self.alphabet[id_pred]
        return pred.reshape(shape_data[:-2])
```

# Python codes for solutions

## solution to section 2.1: `soln_2_1.py`

```python
import sys
from os.path import dirname
sys.path.insert(0, dirname(dirname(__file__)))
from numpy import array, diag, sqrt
from numpy.linalg import eig
from numpy.random import seed, randn
import matplotlib.pyplot as plt
from src.utils import plot_data2d

if __name__ == "__main__":
    N = 1000
    R = array([ [2, -1.2],
                [-1.2, 1]])
    dimension = R.shape[0]
    seed(0)
    D, V = eig(R) # eigenvalues, eigenvectors
    W = randn(dimension, N)
    X_tilde = diag(sqrt(D)) @ W
    X = V @ X_tilde
    for data, str_title, char \
        in list(zip([W, X_tilde, X],
                [r'$W$', r'$\tilde{X}$', r'$X$'],
                ['a', 'b', 'c'])):
        plot_data2d(data, str_title)
        plt.savefig('result/fig_2_1' + char + '.png', Bbox='tight')
        plt.show()
```

## solution to section 2.2: `soln_2_2.py`

```
import sys
from os.path import dirname
sys.path.insert(0, dirname(dirname(__file__)))
from numpy import array
from src.eigen_decompose import generate_data, estimate_param, \
    decorrelate_data, whiten_data
from src.utils import plot_data2d
import matplotlib.pyplot as plt

if __name__ == "__main__":
    N = 1000
    R = array([ [2, -1.2],
                [-1.2, 1]])
    X = generate_data(R, N=N, se=0)
    R_X_hat, mu_X_hat = estimate_param(X)
    print("R_X\n", R_X_hat)
    print("mu_X\n", mu_X_hat)
    X_tilde = decorrelate_data(X, R)
    W = whiten_data(X, R)
    R_W_hat, mu_W_hat = estimate_param(W)
    print("R_W\n", R_W_hat)
    print("mu_W\n", mu_W_hat)
    for data, str_title, char \
        in list(zip([W, X_tilde, X],
                [r'$W$', r'$\tilde{X}$', r'$X$'],
                ['a', 'b', 'c'])):
        plot_data2d(data, str_title)
        plt.savefig('result/fig_2_2' + char + '.png', Bbox='tight')
        plt.show()
```

### solution to section 4: `soln_4.py`

```
import sys
from os.path import dirname
sys.path.insert(0, dirname(dirname(__file__)))
from src.eigen_decompose import svd_data
from src.utils import read_data, display_scaled, plot_projection,
display_synthesized
from numpy import reshape, sqrt
import matplotlib.pyplot as plt
if __name__ == "__main__":
    data_train, folders_train = read_data("resource/training_data")
    num_folder, num_alphabet, height, width = data_train.shape
    X = reshape(data_train, (num_folder*num_alphabet,
height*width)).T
    mu_hat = X.mean(axis=-1, keepdims=True)
    U, D, V = svd_data((X - mu_hat) / sqrt(num_folder*num_alphabet-
1))
    num_top = 12
    U_top = U[:, :num_top]
    display_scaled(U_top)
    plt.savefig('result/fig_4_1a.png', Bbox='tight')
    plt.show()
    num_char = 4
    Y_concat = U.conj().T @ (X[:, :num_char] - mu_hat)
    plot_projection(Y_concat)
    plt.savefig('result/fig_4_1b.png', Bbox='tight')
    plt.show()
    vec_img = X[:, 0].reshape((-1, 1))
    list_num_eigen = [1, 5, 10, 15, 20, 30]
    display_synthesized(vec_img, mu_hat, U, list_num_eigen)
    plt.savefig('result/fig_4_1c.png', Bbox='tight')
    plt.show()
```

## solution to section 5: `soln_5.py`

```
import sys
from os.path import dirname
sys.path.insert(0, dirname(dirname(__file__)))
from src.eigen_decompose import Classifier_PCA, alphabet
from src.utils import read_data

if __name__ == "__main__":
    data_train, folders_train = read_data("resource/training_data")
    data_test,  folders_test  = read_data("resource/test_data")
    classifier = Classifier_PCA(num_eigen=10)
    classifier.fit(data_train)
    for option in ["default", "diag", "global", "global diag",
"identity"]:
        pred = classifier.predict(data_test, option=option)
        print([(s, t) for s, t in list(zip(alphabet, pred[0])) \
            if s != t])
```