CS189

1. (a) By myself. consulting piazza.

(b) I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

Yao Cao

HW6 Yao Cao 25933599

2.(a) $A = (u_1\ u_2 \cdots u_n)\begin{pmatrix}\sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_d \\ & & 0 & \end{pmatrix}\begin{pmatrix} v_1^T \\ \vdots \\ v_d^T \end{pmatrix}$

$= (u_1\ u_2 \cdots u_n)\begin{pmatrix}\sigma_1 v_1^T \\ \sigma_2 v_2^T \\ \vdots \\ \sigma_d v_d^T \\ 0 \\ \vdots \\ 0\end{pmatrix} = \sum_{i=1}^{d} \sigma_i u_i v_i^T$

(b) $A^TA = V\Sigma U^T U \Sigma V^T = V\Sigma^2 V^T$

$A^TAV = V\Sigma^2 V^T V = V\Sigma^2 = \Sigma^2 V$

$A^TA(v_1 \cdots v_d) = \begin{pmatrix}\sigma_1^2 & & \\ & \ddots & \\ & & \sigma_d^2\end{pmatrix}(v_1 \cdots v_d)$

$\Rightarrow A^TAv_i = \sigma_i^2 v_i \qquad i = 1, \cdots, d.$

$AA^T = U\Sigma V^T V \Sigma U^T = U\Sigma^2 U^T$

$AA^TU = U\Sigma^2 U^T U = U\Sigma^2$

$AA^T(u_1 \cdots u_n) = (u_1 \cdots u_n)\begin{pmatrix}\sigma_1^2 & & & & \\ & \ddots & & & \\ & & \sigma_d^2 & & \\ & & & 0 & \\ & & & & \ddots \\ & & & & & 0\end{pmatrix}$

$AA^Tu_i = \sigma_i^2 u_i \qquad i = 1, \cdots n.$

$\sigma_i = 0$ when $i = d+1, \cdots n$

(c) $U^T A V$

$= U^T \sum_{i=1}^{d} U_i \sigma_i V_i^T \cdot V$

$= \left(\sum_{j=1}^{n} a_j U_j^T\right) \sum_{i=1}^{d} U_i \sigma_i V_i^T \left(\sum_{k=1}^{d} b_k V_k\right)$ only $i=j=k$ survives

$= \sum_{i=1}^{d} a_i U_i^T U_i \sigma_i V_i^T V_i b_i$

$= \sum_{i=1}^{d} a_i \sigma_i b_i$ achieves maximum when

$= \sigma_1$

$a_1 = 1$, $a_i = 0$ when $i > 1$

$b_1 = 1$, $b_i = 0$ when $i > 1$

and $U = \sum_{i=1}^{n} a_i U_i = U_1$

$V = \sum_{i=1}^{n} b_i V_i = V_1$

(d) $$\rho(a^TX, b^TY) = \frac{E[a^TXY^Tb]}{\sqrt{E[a^TXX^Ta]\,E[b^TYY^Tb]}}$$

$$= \frac{a^TE[XY^T]b}{(a^TE[XX^T]a)^{1/2}(b^TE[YY^T]b)^{1/2}}$$

$$= \frac{a^T\Sigma_{XY}b}{(a^T\Sigma_{XX}a)^{1/2}(b^T\Sigma_{YY}b)^{1/2}}$$

$$\rho(\alpha a^TX, \beta b^TY) = \frac{\alpha a^T\Sigma_{XY}\beta b}{(\alpha a^T\Sigma_{XX}\alpha a)^{1/2}(\beta b^T\Sigma_{YY}\beta b)^{1/2}}$$

$$= \frac{a^T\Sigma_{XY}b}{(a^T\Sigma_{XX}a)^{1/2}(b^T\Sigma_{YY}b)^{1/2}} \qquad (\alpha, \beta > 0)$$

$$= \rho(a^TX, b^TY)$$

so scaling doesn't affect correlation.

(e) $X = \Sigma_{XX}^{1/2}\tilde{X}$ $\qquad Y = \Sigma_{YY}^{1/2}\tilde{Y}$

$$\rho(a^TX, b^TY) = \frac{E[a^T\Sigma_{XX}^{1/2}\tilde{X}\tilde{Y}^T\Sigma_{YY}^{1/2}b]}{(a^T\Sigma_{XX}^{1/2}\Sigma_{XX}^{1/2}a)^{1/2}(b^T\Sigma_{YY}^{1/2}\Sigma_{YY}^{1/2}b)^{1/2}}$$

$$= \frac{a^T\Sigma_{XX}^{1/2}E[\tilde{X}\tilde{Y}^T]\Sigma_{YY}^{1/2}b}{\|\Sigma_{XX}^{1/2}a\|_2\|\Sigma_{YY}^{1/2}b\|_2}$$

$$= \frac{c^TE[\tilde{X}\tilde{Y}^T]d}{\|c\|_2\|d\|_2}$$

$$= \frac{c^TE[\Sigma_{XX}^{-1/2}XY^T\Sigma_{YY}^{-1/2}]d}{\|c\|_2\|d\|_2}$$

$$= \frac{c^T\Sigma_{XX}^{-1/2}\Sigma_{XY}\Sigma_{YY}^{-1/2}d}{\|c\|_2\|d\|_2}$$

we know scaling doesn't matter so let $\|c\|_2 = \|d\|_2 = 1$

$$\rho = \max_{\|c\|_2=1,\ \|d\|_2=1} c^T\Sigma_{XX}^{-1/2}\Sigma_{XY}\Sigma_{YY}^{-1/2}d$$

(f) we know that $\sigma_1(A) = \max_{\substack{\|u\|_2=1 \\ \|v\|_2=1}} u^T A v$

and σ_1^2 is the largest eigenvalue of AA^T

Similarly, from (e) we know that

ρ is the largest singular value of $\Sigma_{XX}^{-1/2}\Sigma_{XY}\Sigma_{YY}^{-1/2}$ (A)

ρ^2 is the largest eigenvalue of $\Sigma_{XX}^{-1/2}\Sigma_{XY}\Sigma_{YY}^{-1}\Sigma_{XY}^{T}\Sigma_{XX}^{-1/2}$ (AA^T)

(g) from (c) and (b)

c^* should be the first eigenvector of

$AA^T = \Sigma_{XX}^{-1/2}\Sigma_{XY}\Sigma_{YY}^{-1}\Sigma_{XY}^{T}\Sigma_{XX}^{-1/2}$

(h) d^* should be the first eigenvector of

$A^TA = \Sigma_{YY}^{-1/2}\Sigma_{XY}^{T}\Sigma_{XX}^{-1}\Sigma_{XY}\Sigma_{YY}^{-1/2}$

(i) when $\mathrm{cov}(X_i, X_j) = 0 \Rightarrow \Sigma_{XY} = 0$

$\rho = 0$

there is no way to maximize it since it is always zero

If two random vectors are uncorrelated, we can't make them correlated no matter what we do.

(j) we should first square Y and the use CCA on X and Y^2

and find a and b that makes $\rho = 1$

(k) If Y is our target to learn, X is the feature, if we use CCA projection to make their linear projections $b^T Y$ and $a^T X$ more correlated, it will become easier to find their dependence and relationship

3.(a) $\Sigma_{XX} = E[x'x'^T] = \frac{1}{n}\sum_{i=1}^{n} x_i' x_i'^T = \frac{1}{n} X'^T X'$ $\quad X' \in \mathbb{R}^{n \times d}$

X' is the standardized, zero-mean matrix of X

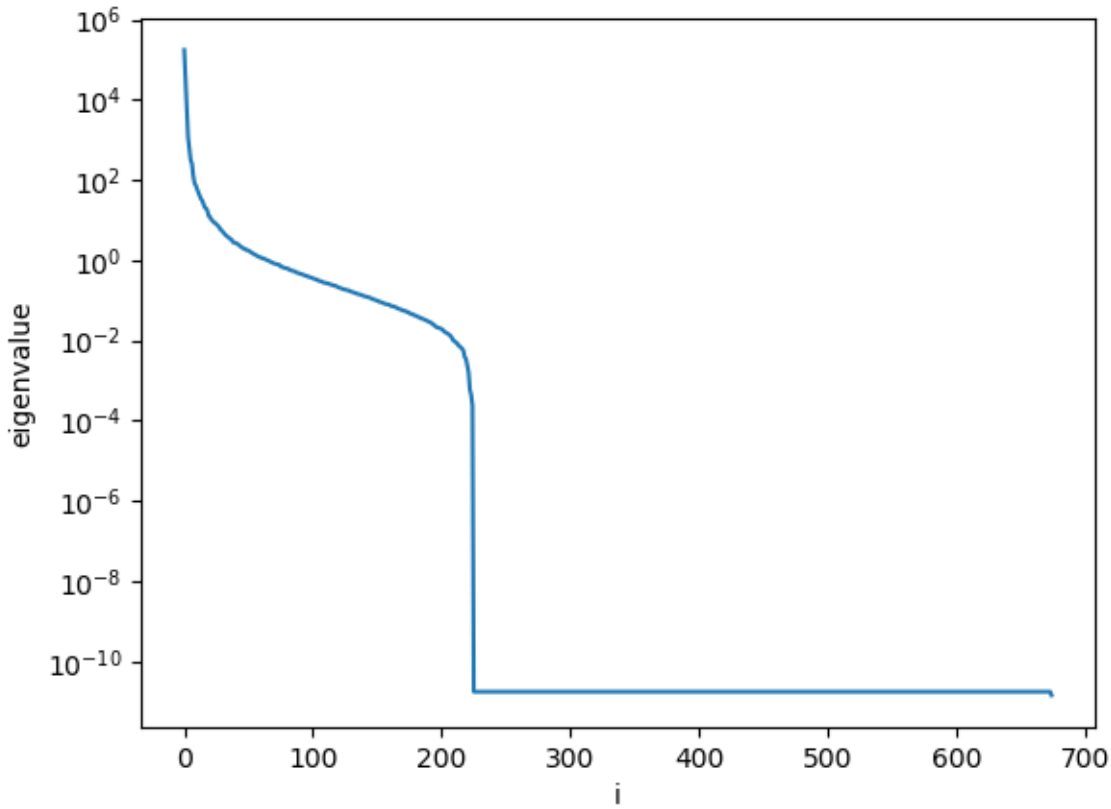
similar for Σ_{YY}

$\Sigma_{XY} = E[x'y'^T] = \frac{1}{n}\sum_{i=1}^{n} x_i' y_i'^T = \frac{1}{n} X'^T Y'$ $\quad X' \in \mathbb{R}^{n \times d}$

$Y' \in \mathbb{R}^{n \times d}$.

Y' is the standardized, zero-mean matrix of Y.

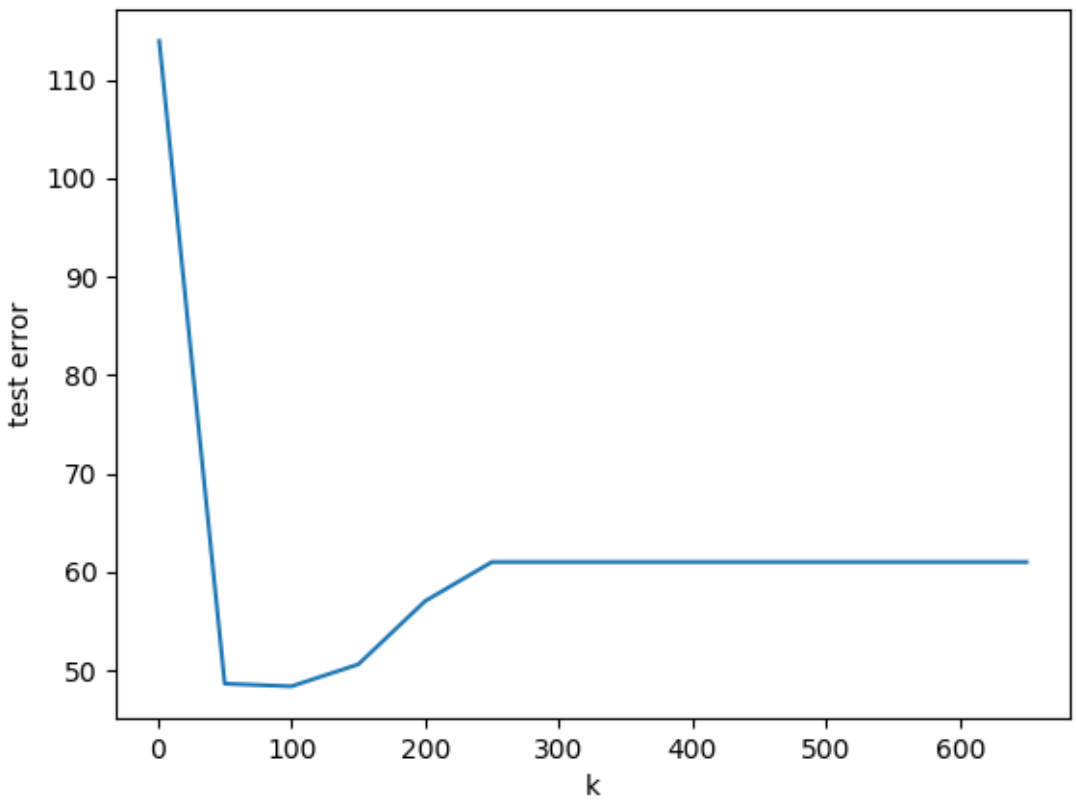
3b


10^{6}
10^{4}
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
eigenvalue
0
100
200
300
400
500
600
700
i

3c

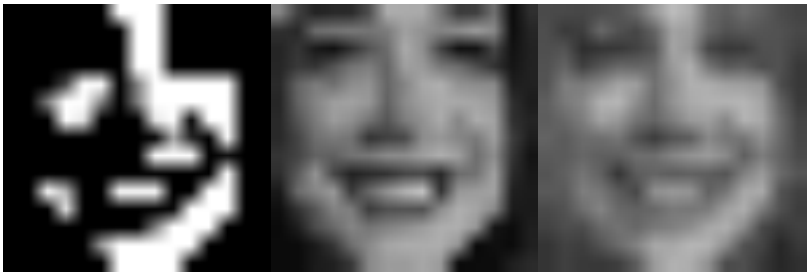
Looks like a face if you look carefully. The features(eyes, nose, etc) on the face are the correlated parts between the mooney image and gray scale image.

3d



3e

We use k=100 since the test error is the smallest



```
import os
import numpy as np
import cv2
import copy
import glob

import sys

from numpy.random import uniform

import pickle
from scipy.linalg import eig
from scipy.linalg import sqrtm
from numpy.linalg import inv
from numpy.linalg import svd
import numpy.linalg as LA
import matplotlib.pyplot as plt
import IPython
from sklearn.preprocessing import StandardScaler

def standardized(v):
    return (v/255.0) * 2.0 - 1.0

def flatten_and_standardize(data):
    result = []
    for d in data:
        d = d.flatten()
        d = standardized(d)
        result.append(d)
    return result

class Mooney(object):

        def __init__(self):
                self.lmbda = 1e-5

        def load_data(self):
                self.x_train = pickle.load(open('x_train.p','rb'))
                self.y_train = pickle.load(open('y_train.p','rb'))
                self.x_test = pickle.load(open('x_test.p','rb'))
```

```
        self.y_test = pickle.load(open('y_test.p','rb'))

    def compute_covariance_matrices(self):
        # USE STANDARD SCALAR TO DO MEAN SUBTRACTION
        ss_x = StandardScaler(with_std = False)
        ss_y = StandardScaler(with_std = False)

        num_data = len(self.x_train)

        x = self.x_train[0]
        y = self.y_train[0]

        x_f = x.flatten()
        y_f = y.flatten()

        x_f_dim = x_f.shape[0]
        y_f_dim = y_f.shape[0]

        self.x_dim = x_f_dim
        self.y_dim = y_f_dim

        self.C_xx = np.zeros([x_f_dim,x_f_dim])
        self.C_yy = np.zeros([y_f_dim,y_f_dim])
        self.C_xy = np.zeros([x_f_dim,y_f_dim])

        x_data = []
        y_data = []

        for i in range(num_data):
            x_image = self.x_train[i]
            y_image = self.y_train[i]

            # FLATTEN DATA
            x_f = x_image.flatten()
            y_f = y_image.flatten()

            # STANDARDIZE DATA
            x_f = standardized(x_f)
            y_f = standardized(y_f)

            x_data.append(x_f)
            y_data.append(y_f)

        # SUBTRACT MEAN
```

```
        ss_x.fit(x_data)
        x_data = ss_x.transform(x_data)

        ss_y.fit(y_data)
        y_data = ss_y.transform(y_data)

        for i in range(num_data):
                x_f = np.array([x_data[i]])
                y_f = np.array([y_data[i]])
                #####TO DO:
                self.C_xx += x_f.T @ x_f
                self.C_yy += y_f.T @ y_f
                self.C_xy += x_f.T @ y_f

        # DIVIDE BY THE NUMBER OF DATA POINTS
        self.C_xx = 1.0/float(num_data)*self.C_xx
        self.C_yy = 1.0/float(num_data)*self.C_yy
        self.C_xy = 1.0/float(num_data)*self.C_xy

def compute_projected_data_matrix(self,X_proj):

        Y = []
        X = []

        Y_test = []
        X_test = []

        # LOAD TRAINING DATA
        for x in self.x_train:
                x_f = np.array([x.flatten()])
                # STANDARDIZE DATA
                x_f = standardized(x_f)
                # TODO: PROJECT DATA
                X.append(x_f @ X_proj.T)



        Y = flatten_and_standardize(self.y_train)

        for x in self.x_test:
                x_f = np.array([x.flatten()])
                # STANDARDIZE DATA
                x_f = standardized(x_f)
                # TODO: PROJECT DATA
```

```
                X_test.append(x_f @ X_proj.T)

        Y_test = flatten_and_standardize(self.y_test)

        # CONVERT TO MATRIX
        self.X_ridge = np.vstack(X)
        self.Y_ridge = np.vstack(Y)

        self.X_test_ridge = np.vstack(X_test)
        self.Y_test_ridge = np.vstack(Y_test)

def compute_data_matrix(self):
        X = flatten_and_standardize(self.x_train)
        Y = flatten_and_standardize(self.y_train)
        X_test = flatten_and_standardize(self.x_test)
        Y_test = flatten_and_standardize(self.y_test)

        # CONVERT TO MATRIX
        self.X_ridge = np.vstack(X)
        self.Y_ridge = np.vstack(Y)

        self.X_test_ridge = np.vstack(X_test)
        self.Y_test_ridge = np.vstack(Y_test)


def solve_for_variance(self):
        eigen_values = np.zeros((675,))
        eigen_vectors = np.zeros((675, 675))
        # TODO:
        D1 = self.C_xx+self.lmbda*np.eye(675)
        D1[np.diag_indices(675)] = 1/(D1.diagonal()**0.5)

        D2 = self.C_yy+self.lmbda*np.eye(675)
        D2[np.diag_indices(675)] = 1/(D2.diagonal()**0.5)

        matrix = D1 @ self.C_xy @ D2

        eigen_vectors, eigen_values, dummy = svd(matrix)

        return eigen_values, eigen_vectors

def project_data(self, eig_val,eig_vec,proj=150):

        # TODO: COMPUTE PROJECTION SINGULAR VECTORS
```

```
        return eig_vec[:, 0:proj].T

    def ridge_regression(self):
        w_ridge = []
        for i in range(self.y_dim):
            # TODO: IMPLEMENT RIDGE REGRESSION
            y = self.Y_ridge[:, i]
            w = inv(self.X_ridge.T @ self.X_ridge +
self.lmbda*np.eye(self.X_ridge.shape[1])) @ self.X_ridge.T @ y
            w_ridge.append(w)

        self.w_ridge = np.vstack(w_ridge)


    def plot_image(self, vector):
        vector = ((vector+1.0)/2.0)*255.0
        vector = np.reshape(vector,(15,15,3))
        p = vector.astype("uint8")
        p = cv2.resize(p,(100,100))
        count = 0

        cv2.imwrite('a_face_'+str(count)+'.png',p)

    def measure_error(self, X_ridge, Y_ridge):
        prediction = np.matmul(self.w_ridge,X_ridge.T)
        evaluation = Y_ridge.T - prediction

        #print(evaluation)

        dim,num_data = evaluation.shape

        error = []

        for i in range(num_data):
            # COMPUTE L2 NORM for each vector then square
            error.append(LA.norm(evaluation[:,i])**2)

        # Return average error
        return np.mean(error)


    def draw_images(self):
        for count, x in enumerate(self.X_test_ridge):
            prediction = np.matmul(self.w_ridge,x)
```

```
                        prediction = ((prediction+1.0)/2.0)*255.0
                        prediction = np.reshape(prediction,(15,15,3))
                        p = prediction.astype("uint8")
                        p = cv2.resize(p,(100,100))
                        cv2.imwrite('face_'+str(count)+'.png',p)

                for count, x in enumerate(self.x_test):
                        x = x.astype("uint8")
                        x = cv2.resize(x,(100,100))
                        cv2.imwrite('og_face_'+str(count)+'.png',x)

                for count, x in enumerate(self.y_test):
                        x = x.astype("uint8")
                        x = cv2.resize(x,(100,100))
                        cv2.imwrite('gt_face_'+str(count)+'.png',x)

if __name__ == '__main__':

        mooney = Mooney()

        mooney.load_data()
        mooney.compute_covariance_matrices()
        eig_val, eig_vec = mooney.solve_for_variance()
        plt.semilogy(np.arange(675), eig_val)
        plt.xlabel('i')
        plt.ylabel('eigenvalue')
        plt.savefig('spectrum.png')
        plt.show()

        mooney.plot_image(eig_vec[:, 0])

        proj = [1,50,100,150,200,250,300,350,400,450,500,650]
        error_test = []
        #for p in proj:

        #       X_proj = mooney.project_data(eig_val,eig_vec, proj=p)

                # COMPUTE REGRESSION
        #       mooney.compute_projected_data_matrix(X_proj)
        #       mooney.ridge_regression()
        #       training_error = mooney.measure_error(mooney.X_ridge, mooney.Y_ridge)
        #       test_error = mooney.measure_error(mooney.X_test_ridge,
mooney.Y_test_ridge)
                ## mooney.draw_images()
```

```
#	error_test.append(test_error)

#plt.plot(proj,error_test)
#plt.xlabel('k')
#plt.ylabel('test error')
#plt.savefig('error.png')
#plt.show()

# COMPUTE REGRESSION NO PROJECT
X_proj = mooney.project_data(eig_val,eig_vec, proj=100)
mooney.compute_projected_data_matrix(X_proj)
mooney.ridge_regression()
mooney.draw_images()
#training_error = mooney.measure_error(mooney.X_ridge, mooney.Y_ridge)
#test_error = mooney.measure_error(mooney.X_test_ridge, mooney.Y_test_ridge)
```

4. (a) $E[\hat{w}_\lambda] = E\left[\frac{S_{xy}}{S_x^2+\lambda}\right] = \frac{\sum_{i=1}^{n} x_i E[Y_i]}{S_x^2+\lambda} = \frac{\sum_{i=1}^{n} x_i \cdot x_i w^*}{S_x^2+\lambda} = \frac{S_x^2 w^*}{S_x^2+\lambda}$

$$E[\hat{w}_\lambda] - w^* = \frac{S_x^2 w^*}{S_x^2+\lambda} - w^* = \frac{-\lambda}{S_x^2+\lambda} w^*$$

$$\text{Bias}^2(\hat{w}_\lambda) = \frac{\lambda^2}{(S_x^2+\lambda)^2} w^{*2}$$

(b) $\text{Var}(\hat{w}_\lambda) = \text{Var}\left(\frac{S_{xy}}{S_x^2+\lambda}\right)$

$$= \text{Var}\left(\frac{\sum_{i=1}^{n} x_i Y_i}{S_x^2+\lambda}\right)$$

$$= \frac{1}{(S_x^2+\lambda)^2} \sum_{i=1}^{n} x_i^2 \text{Var}(Y_i)$$

$$= \frac{1}{(S_x^2+\lambda)^2} \sum_{i=1}^{n} x_i^2 \text{Var}(Z_i)$$

$$= \frac{1}{(S_x^2+\lambda)^2} \sum_{i=1}^{n} x_i^2$$

$$= \frac{S_x^2}{(S_x^2+\lambda)^2}$$

(c) as λ increases, bias^2 increases, variance decreases

as $\lambda \to 0$, bias$^2 \to 0$, variance $\to \frac{1}{S_x^2}$

as $\lambda \to \infty$, bias$^2 \to w^{*2}$, variance $\to 0$

We should choose λ such that bias^2 + Variance is the smallest.

5. (a) in ridge we have $w' \sim N(0, I)$

ridge regression to solve $Xw' = y$ is

$$\hat{w}' = (X^T X + I)^{-1} X^T y$$

$w \sim N(\hat{w}_B, \Psi)$, we can write w as

$$w = \Psi^{1/2} w' + \hat{w}_B$$

$$y = Xw = X(\Psi^{1/2} w' + \hat{w}_B) = X\Psi^{1/2} w' + X\hat{w}_B$$

$$X\Psi^{1/2} w' = y - X\hat{w}_B$$

$$\hat{w}' = (\Psi^{1/2} X^T X \Psi^{1/2} + I)^{-1} \Psi^{1/2} X^T (y - X\hat{w}_B)$$

$$\hat{w} = \Psi^{1/2} w' + \hat{w}_B$$

$$= \Psi^{1/2} (\Psi^{1/2} X^T X \Psi^{1/2} + I)^{-1} \Psi^{1/2} X^T (y - X\hat{w}_B) + \hat{w}_B$$

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

(b) we could assume that $w \sim N(\hat{w}_B, \sigma^2)$ and treat σ^2 as a hyperparameter.

use 5-fold cross validation, at different σ^2, use MAP to estimate $\hat{w}$ on the training data, and calculate error on validation data, average the 5-fold error.

The σ^2 that produces the smallest validation error is the value we want, and the model associated with it is the desired model.

6. (a) $\hat{y}_{test} = x^T_{test} V \hat{w}_{ridge}$

$$= \hat{x}_{test} V \begin{pmatrix} \frac{\sigma_1}{\lambda+\sigma_1^2} & & 0 \\ & \ddots & \\ & & \frac{\sigma_d}{\lambda+\sigma_d^2} \end{pmatrix} \begin{pmatrix} u_1^T \\ \vdots \\ u_n^T \end{pmatrix} y$$

$$= \hat{x}_{test} (v_1 \cdots v_d) \begin{pmatrix} \frac{\sigma_1}{\lambda+\sigma_1^2} u_1^T \\ \frac{\sigma_2}{\lambda+\sigma_2^2} u_2^T \\ \vdots \\ \frac{\sigma_d}{\lambda+\sigma_d^2} u_d^T \end{pmatrix} y$$

$$= \hat{x}_{test} \sum_{i=1}^{d} v_i \frac{\sigma_i}{\lambda+\sigma_i^2} u_i^T y$$

$$\beta_i = \frac{\sigma_i}{\lambda+\sigma_i^2}$$

(b) $X V_k w = y$ $\qquad V = (v_1 \cdots v_k)$

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

$$\hat{y}_{test} = x_{test}^T V_k (V_k^T X^T X V_k)^{-1} V_k^T X^T y$$
$$= x_{test}^T V_k (V_k^T V \Sigma U^T U \Sigma V^T V_k)^{-1} V_k^T V \Sigma U^T y$$
$$= x_{test}^T V_k (V_k^T V \Sigma \Sigma V^T V_k)^{-1} V_k^T V \Sigma U^T y$$

$$\Sigma V^T V_k = \Sigma \begin{pmatrix} v_1^T \\ \vdots \\ v_k^T \\ v_{k+1}^T \\ \vdots \\ v_d^T \end{pmatrix} (v_1 \cdots v_k) = \Sigma \begin{pmatrix} I_k \\ 0 \end{pmatrix} = \begin{pmatrix} \Sigma_k \\ 0 \end{pmatrix}$$

$$\hat{y}_{test} = x_{test}^T V_k (\Sigma_k \Sigma_k)^{-1} (\Sigma_k \ 0) \begin{pmatrix} u_1^T \\ \vdots \\ u_d^T \end{pmatrix} y$$

$$= x_{test}^T V_k \Sigma_k^{-1} U_k^T y = x_{test}^T \sum_{i=1}^{k} v_i \frac{1}{\sigma_i} u_i^T y$$

$$\beta_i = \frac{1}{\sigma_i} \qquad (1 \le i \le k)$$

(c) (i) ridge

(ii) K-PCA-OLS

(iii) OLS

7. (a) $XX^T = U\Sigma V^T V \Sigma^T U^T = U\Sigma\Sigma^T U^T$

$$XX^T U = U\Sigma\Sigma^T U^T U = U\Sigma\Sigma^T$$

$$XX^T (U_1 \cdots U_n) = (U_1 \cdots U_n) \begin{pmatrix} \sigma_1^2 & & & & \\ & \sigma_2^2 & & & \\ & & \ddots & & \\ & & & \sigma_i^2 & \\ & & & & \ddots \\ & & & & & 0 \end{pmatrix}$$

$$XX^T U_i = \sigma_i^2 U_i$$

the columns of U are the eigenvectors of XX^T

(b) $X = U\Sigma V^T$ $\quad U \in \mathbb{R}^{n\times l}$ $\quad \Sigma \in \mathbb{R}^{l\times l}$ $\quad V \in \mathbb{R}^{l\times l}$

$$U^T X = U^T U \Sigma V^T = \Sigma V^T$$

$$V^T = \Sigma^{-1} U^T X$$

$$= \begin{pmatrix} \frac{1}{\sigma_1} & & \\ & \ddots & \\ & & \frac{1}{\sigma_l} \end{pmatrix} \begin{pmatrix} U_1^T \\ \vdots \\ U_l^T \end{pmatrix} X$$

$$\begin{pmatrix} V_1^T \\ \vdots \\ V_l^T \end{pmatrix} = \begin{pmatrix} \frac{1}{\sigma_1} U_1^T \\ \vdots \\ \frac{1}{\sigma_l} U_l^T \end{pmatrix} X$$

$$V_j^T = \frac{1}{\sigma_j} U_j^T X$$

$$z_j = V_j^T x_{test} = \frac{1}{\sigma_j} U_j^T X x_{test}$$

(c) the analog of $X x_{test}$ in the general kernel setting is

$$K_{test} = \begin{pmatrix} k(x_1, x_{test}) \\ \vdots \\ k(x_n, x_{test}) \end{pmatrix}$$

σ_j and U_j are the jth singular value and left singular vector of X, respectively.

then $z_j = \frac{1}{\sigma_j} u_j^T K_{test}$

8. (a) $\sum_{i=1}^{k} \sigma_i u_i v_i^T$

(b) $1, x_i, y_i, x_i^2, x_i y_i, y_i^2$

(c) $k(x,y) = x^T y$

$k(x,y) = e^{-\frac{1}{2}\|x-y\|^2}$

$k(x,y) = (1+x^T y)^p$ for some degree p.

(d) TLS

(e) Preprocess X using PCA with n components

Use a kernel approach

Add a ridge penalty to OLS

(f) Use of a kernel

Adding poly features

(g) Training error

Bias

(h) Validation error

Variance

(i) Validation error