CSE 446 HW3

Zachary McNulty

May 2019

Collaborators: Pei Lee Yap, Conrad Spiekerman, Valerie Liao

Classification

Problem 1 - Perceptron Algorithm

Suppose our weights are initialized to some arbitrary $w = w_0$ and we continuously feed in the vector x whose true label is 1. Note that since the weights are not updated when the label is correctly predicted, as soon the weights correctly predict x once they will always correctly predict it because we always pass in the same x every time. Thus, in the worst case scenario it must be true that x is initially misclassified. Otherwise the training algorithm will never update its weights and hence never make a mistake. Thus as the true label of x is 1 our update rule is:

$$w_{k+1} = w_k + x$$

and we will continue making these updates until x is finally correctly classified (until $x^T w_k > 0$). Thus, we will stop when:

$$x^T w_k = x^T(w_0 + kx) = x^T w_0 + k||x||_2^2 > 0$$

$$\rightarrow k > -\frac{x^T w_0}{||x||_2^2}$$

$$-\frac{x^T w_0}{||x||_2^2} = -\frac{||x||_2||w_0||_2 \cos(\theta)}{||x||_2^2} \leq \frac{||x||_2||w_0||_2}{||x||_2^2} = \frac{1}{||x||_2}$$

As k represents the number of updates we have to make (and hence to number of mistakes made) this implies that on this set up the training algorithm will make at most $\left\lceil \frac{1}{||x||_2} \right\rceil$ mistakes.

Problem 2

(a)

$$\ell((x,y) \mid w) = max(0, -y(w^T x))$$

$$\delta_w\left(\ell((x,y) \mid w)\right) = \begin{cases} 0 & -yw^T x < 0 \\ [-yx, 0] & -yw^x = 0 \\ -yx & -yw^T x > 0 \end{cases} \tag{1}$$

Since we choose any suitable subgradient, we can simplify this to:

$$\delta_w\left(\ell((x,y) \mid w)\right) = (-yx)\mathbb{I}\{-yw^T x \geq 0\}$$

Let $w_k = \tilde{w}$ be the current iterate. Then the next iterate would be:

$$\boxed{w_{k+1} = w_k - \eta\left(\frac{1}{n}\sum_{i=1}^{n}(-y_i x_i)\mathbb{I}\{-y_i w_k^T x_i \geq 0\}\right)}$$

(b) If we choose $\eta = 1$ and run stochastic gradient descent on the following loss function (with a batch size of one) then the above update rule becomes:

$$w_{k+1} = w_k + y_i x_i \mathbb{I}\{-y_i w_k^T x_i \geq 0\}$$

We drop the $\frac{1}{n}$ in front of the sum to scale our gradient appropriately to the batch size of one. The indicator random variable is one when $sign(y_i) \neq sign(w_k^T x_i)$, the event that our linear decision boundary incorrectly predicts the label of the variable. In this case, we will be adding the value $y_i x_i$ to our weights, just as we do in the perceptron training algorithm. If the indicator random variable is zero, $w_{k+1} = w_k$ so we do not update the weights at all just as in the perceptron training algorithm. Thus, we can see that the perceptron algorithm is essentially just stochastic gradient descent with a batch size of one and a learning rate $\eta = 1$ on the above loss function.

(c) For the loss function described in this problem there would be zero loss associated any correctly classified points, even with points just barely on the correct side of the decision boundary (for which $yw^T x$ is slighlty positive). As we discussed with SVM in class, there can be a benefit to generating decision boundaries that have some significant margin. With the standard hidge loss function, we can see there can still be loss associated to correctly classified points. Even if $yw^T x > 0$ and the point is correctly classified, there is still a loss associated to it as long as $yw^T x < 1$. Intuitively, we can expect the standard hidge function to generate a decision boundary with a larger margin.

Problem 3

(a)

$$\mathcal{L}(W) = -\sum_{i=1}^{n}\sum_{\ell=1}^{k} \mathbb{I}\{label_i = \ell\} log\left(\frac{exp(w^{(\ell)} \cdot x_i)}{\sum_{j=1}^{k} exp(w^{(j)} \cdot x_i)}\right) \tag{1}$$

Note that due to the indicator random variable, all terms in the inner summand are zero except the ones associated to the correct label. Let y_i be the one-hot encoding of each label $label_i$. Since only the ℓ^{th} entry of y_i will be one and the rest are zero, $Wy_i = w^{(\ell)}$ and $We_j = w^{(j)}$. This allows us to condense $\mathcal{L}(w)$ to:

$$\mathcal{L}(W) = -\sum_{i=1}^{n} log\left(\frac{exp(Wy_i \cdot x_i)}{\sum_{j=1}^{k} exp(We_j \cdot x_i)}\right) = -\sum_{i=1}^{n}\left(Wy_i \cdot x_i - log\left(\sum_{j=1}^{k} exp((W^T x_i)[j])\right)\right) \tag{2}$$

Where $(W_T x_i)[j]$ denotes the j^{th} entry in the vector $W^T x_i$. Note that $Wy_i \cdot x_i = y_i^T W^T x_i = x_i^T W y_i = \sum_{r=1}^{d}\sum_{c=1}^{k} x_i[r]W_{rc}y_i[c]$. Thus, when we take the partial derivative with respect to W_{rc} all we get is $x_i[r]y_i[c]$. We use bracket notation below to denote an element in a vector (i.e. $x[i]$ is the i^{th} element in the vector x). Thus:

$$\nabla_{W_{rc}}(Wy_i \cdot x_i) = x_i[r]y_i[c] \rightarrow \nabla_W(Wy_i \cdot x_i) = x_i y_i^T$$

In the below sum, we can see only the term when $j = c$ will contain W_{rc}. The derivative of $exp(w^{(c)}x_i)$ with respect to W_{rc} is clearly $x_i[r]exp(w^{(c)}x_i) = x_i[r]exp(W^T x_i[c])$

$$\nabla_{W_{rc}} log\left(\sum_{j=1}^{k} exp((W^T x_i)[j])\right) = \frac{x_i[r]exp((W^T x_i)[c]))}{\sum_{j=1}^{k} exp((W^T x_i)[j]}$$

$$\rightarrow \nabla_W log\left(\sum_{j=1}^{k} exp((W^T x_i)[j])\right) = x_i\left(\frac{exp(W^T x_i)}{\sum_{j=1}^{k} exp((W^T x_i)[j]}\right)^T = x_i(\hat{y}_i^{(W)})^T$$

$$\nabla_W \mathcal{L}(w) = -\sum_{i=1}^{n}\left(x_i y_i^T - x_i(\hat{y}_i^{(W)})^T\right) = \boxed{-\sum_{i=1}^{n} x_i(y_i - \hat{y}_i^{(W)})^T} \tag{3}$$

(b)

$$J(W) = \frac{1}{2}\sum_{i=1}^{n} ||y_i - W^T x_i||_2^2 = \frac{1}{2}\sum_{i=1}^{n}(y_i - W^T x_i)^T(y_i - W^T x_i)$$

$$= \frac{1}{2}\sum_{i=1}^{n}(y_i - x_i[1]W_{1\bullet} - x_i[2]W_{2\bullet} - ... - x_i[d]W_{d\bullet})^T(y_i - x_i[1]W_{1\bullet} - x_i[2]W_{2\bullet} - ... - x_i[d]W_{d\bullet})$$

where $W_{i\bullet}$ denotes the i^{th} row of W. From here, we can apply product rule to calculate the partial derivative with respect to W_{rc}. Since only the $x_i[r]W_{r\bullet}$ term contains W_{rc} and it contains it in the c^{th} row only this yields:

$$\nabla_{W_{rc}} J(W) = \frac{1}{2}\sum_{i=1}^{n} -2x_i[r]e_c^T(y_i - W^T x_i) = \sum_{i=1}^{n} -x_i[r](y_i - \tilde{y}_i^{(W)})[c]$$

where e_c is the c^{th} column of the identity matrix and $(y_i - \tilde{y}_i^{(W)})[c]$ denotes the c^{th} entry of $(y_i - \tilde{y}_i^{(W)})$. Thus, combining these individual partial derivatives we can see:

$$\boxed{\nabla_W J(W) = -\sum_{i=1} x_i(y_i - \tilde{y}_i^{(W)})^T}$$

(c) First we will show some efficient ways for calculating $\nabla J(W)$ and $\nabla L(W)$. Note that for any matrix multiply AB with $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times k}$ we can write AB as the sum of outerproducts between columns of A and rows of B. Specifically:

$$AB = \sum_{i=1}^{n} A_{\bullet i} B_{i\bullet}$$

Thus we can see:

$$y = \begin{bmatrix} y_1^T \\ y_2^T \\ \dots \\ y_n^T \end{bmatrix} \qquad y - \tilde{y}^{(W)} = \begin{bmatrix} (y_1 - \tilde{y}_1^{(W)})^T \\ (y_2 - \tilde{y}_2^{(W)})^T \\ \dots \\ (y_n - \tilde{y}_n^{(W)})^T \end{bmatrix} \qquad X = \begin{bmatrix} - - x_1^T - - \\ - - x_2^T - - \\ \dots \\ - - x_n^T - - \end{bmatrix}$$

$$-X^T(y - \tilde{y}^{(W)}) = -\sum_{i=1}^{n} x_i (y_i - \tilde{y}_i^{(W)})^T = \nabla_W J(W)$$

Where $\tilde{y}_i^{(W)} = W^T x_i$. Clearly the above is also true for $\nabla_W L(W)$ if we just redefine $\tilde{y}_i^{(W)}$ to include the softmax elementwise operation.

Running gradient descent on $\nabla_W L(W)$ and $\nabla_W J(W)$ until convergence (which we defined to be the point when the maximum entry in magnitude in the gradient is less than some pre-specified δ) we find that the final W matrices achieve the below accuracies. Since the objective functions $J(W), L(W)$ do not normalize the error with respect to the sample size n (e.g. there is no $\frac{1}{n}$ scaling factor), the entries in the gradient are quite large and scale with n. As a result, we define our δ with respect to n. For $J(W)$ we chose $\delta = n * 10^{-4} = 60$, $\eta = 7 * 10^{-7}$. For $L(W)$ we chose $\delta = 5n * 10^{-4} = 30$, $\eta = 10^{-5}$.

J(W)

- training error rate: 14.95%
- testing error rate: 14.31%

L(W)

- training error rate: 7.5 %
- testing error rate: 7.67 %

Problem 3 Code

```
'''
hw3_q3c.py

Given a dataset whose response variable is a class y = 1,2,3,..., k (i.e. a classification
                                                        problem with k classes)
determine the optimal classifier according to the loss functions specified in the homework
                                                        specification

'''

import numpy as np
import random
import matplotlib.pyplot as plt
from mnist.loader import MNIST


def grad_J(X, y, W, lam=0):
    '''

    X = data matrix with rows as measurements and columns are features (n x d)
    y = response variable (n x k) in one-hot encoding; each row is a label
    W = weights vector (d x k)
    lam = L2 regularizer (more specifically frobenius norm)
    '''

    yhat = np.dot(X, W)
    return -1 * np.dot(X.T, y - yhat) + 2 * lam * W



def J_function(X, y, W):
    '''

    X = data matrix with rows as measurements and columns are features (n x d)
    y = response variable (n x k) in one-hot encoding; each row is a label
    W = weights matrix (d x k)
    '''

    # NOTE: since we are summing over the two norms, we could equivalently just square all
    # elements and sum over entire matrix
    return 0.5 * np.sum(np.square(y - np.dot(X, W)))

def grad_L(X,y, W):
    '''

    X = data matrix with rows as measurements and columns are features (n x d)
    y = response variable (n x k) in one-hot encoding; each row is one label
    W = weights vector (d x k)
    '''
    # perform the softmax operation on each row
    yhat = np.exp(np.dot(X, W))
    summand = np.expand_dims(np.sum(yhat, axis=1), axis=1)
    yhat = yhat / summand

    return -1 * np.dot(X.T, y - yhat)

def L_function(X,y,W):
    '''

    X = data matrix with rows as measurements and columns are features (n x d)
    y = response variable (n x k) in one-hot encoding; each row is one label
    W = weights vector (d x k)
    '''
    Wyx = np.dot(X, np.dot(W, y.T))
    summand = np.sum(np.exp(np.dot(W.T, X.T)) , axis=0)
    return -1 * np.sum(Wyx - np.log(summand))


def error_rate(X, labels, W):
    '''

    X = data matrix with rows as measurements and columns are features (n x d)
    labels = class number in {1,2,...,1}
    W = weights vector (d x 1)
```

```
    '''
    predictions = np.argmax(np.dot(W.T, X.T), axis=0)

    return np.sum(np.where(predictions != labels, 1, 0)) / len(labels)


def gradient_descent(x_init, gradient_function, eta=0.1, delta=1e-4, X=None, y=None):
    '''
    Runs gradient descent to calculate minimizer x of the function whose gradient
    is defined by the given gradient_function.

    x_init = [w,b] is the initial values to set to the vector being descended on (in this
                                                    problem w and b)
    gradient_function = a function that takes in a vector x and outputs gradient evaluated at
                                                    that point
    eta  = the learning rate for gradient descent
    delta = stopping condition; stop if all entries in gradient change by less than delta in
                                                    an iteration.

    '''
    x = x_init
    grad = gradient_function(x)
    it = 0
    while np.amax(np.abs(grad)) > delta:
        # perform a step in gradient descent

        it += 1
        print(it)
        x = x - eta * grad

        print(J_function(X, y, x))
        #print(L_function(X, y, x))
        grad = gradient_function(x)

    # x is the best variable values
    return x



#
                                                ============================================================


# Load MNIST Data
mndata = MNIST('./data')
X_train, labels_train = map(np.array, mndata.load_training())
X_test, labels_test = map(np.array, mndata.load_testing())
X_train = X_train / 255.0
X_test = X_test / 255.0


#convert training labels to one hot
# i.e. encode an i as [0, 0, ..., 0, 1, 0, ... 0] where only the ith entry is nonzero
Y_train = np.zeros((X_train.shape[0], 10))
for i, digit in enumerate(labels_train):
        Y_train[i, digit] = 1
#
                                                ============================================================


# Problem 3c)
n = X_train.shape[0]
d = X_train.shape[1] # 28 x 28 = 784
k = 10 # 10 classes, one for each digit


# define initial vector and the gradient function
W_init_L = np.zeros((d, k))
W_init_J = np.zeros((d, k))
gradient_function_J = lambda W: grad_J(X=X_train, y=Y_train, W=W, lam=lam)
gradient_function_L = lambda W: grad_L(X=X_train, y=Y_train, W=W)


# J function
delta = 1e-4 * n
print(delta)
```

```
eta = 7e-7 # learning rate
lam = 0

J_best_train = gradient_descent(W_init_J, gradient_function_J, eta, delta, X=X_train, y=
                                                    Y_train)

J_training_error_rate = error_rate(X_train, labels_train, J_best_train)
J_testing_error_rate = error_rate(X_test, labels_test, J_best_train)

print("J train: ", J_training_error_rate)
print("J test: ", J_testing_error_rate)

# L function
delta = 5e-4 * n
eta = 1e-5 # learning rate
lam = 0
L_best_train = gradient_descent(W_init_L, gradient_function_L, eta, delta, X=X_train, y=
                                                    Y_train)

L_training_error_rate = error_rate(X_train, labels_train, L_best_train)
L_testing_error_rate = error_rate(X_test, labels_test, L_best_train)

print("L train: ", L_training_error_rate)
print("L test: ", L_testing_error_rate)
```

Kernels

Problem 4

Define $\phi(x)$ to be the infinite dimensional feature map from $\mathbb{R}$ defined by:

$$\phi(x)_i = \frac{1}{\sqrt{i!}} e^{-\frac{x^2}{2}} x^i \qquad \forall i \in \mathbb{N} \cup \{0\}$$

Then it holds that:

$$\begin{aligned}
\phi(x) \cdot \phi(x') &= \sum_{i=0}^{\infty} \left(\frac{1}{\sqrt{i!}} e^{-\frac{x^2}{2}} x^i \right) \left(\frac{1}{\sqrt{i!}} e^{-\frac{(x')^2}{2}} (x')^i \right) \\
&= e^{-\frac{x^2+(x')^2}{2}} \sum_{i=0}^{\infty} \left(\frac{1}{i!} (xx')^i \right)
\end{aligned}$$

The right sum is just the Taylor Expansion for $e^{\tilde{x}}$ with $\tilde{x} = xx'$. Thus:

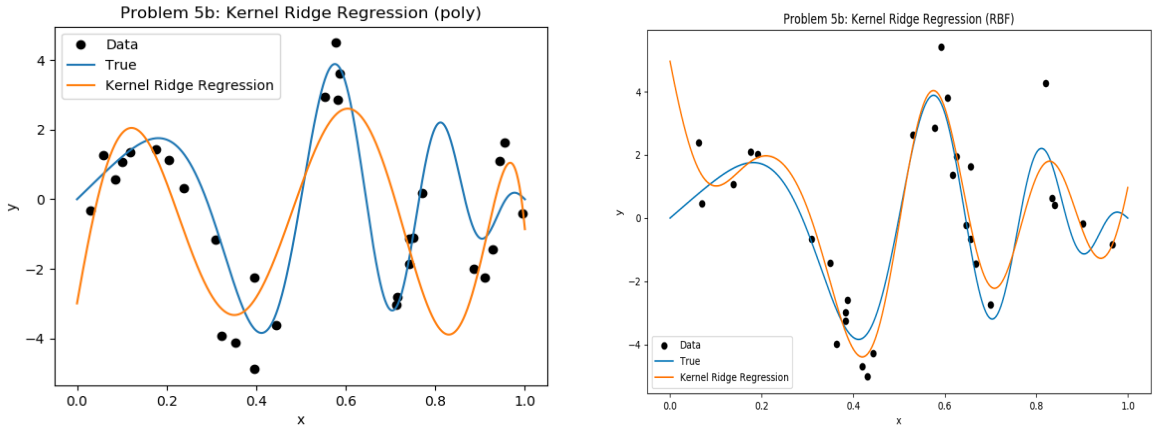
$$\begin{aligned}
&= e^{-\frac{x^2+(x')^2}{2}} e^{xx\prime} \\
&= e^{-\frac{x^2-2xx'+(x')^2}{2}} \\
&\boxed{= e^{-\frac{(x-x')^2}{2}}}
\end{aligned}$$

Problem 5

(a) For the polynomial kernel, we ran LOO cross-validation on λ values in the range $[10^{-6}, 10]$ and d values (integrs) in the range $[1, 20]$. This generated the parameters $\lambda = 10^{-6}$ and $d = 10$

For the RBF kernel, we ran LOO cross-validation on λ in the range $[10^{-6}, 10]$ and γ values in the range $[1, 25]$. This generated the parameters $\lambda = 0.09$ and $\gamma = 25$

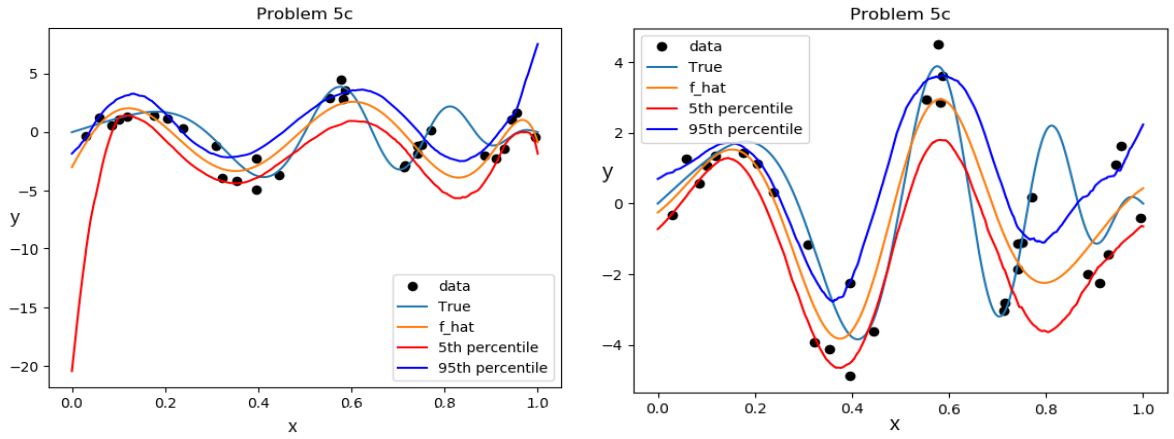
(b) Below we can see the functions generated using kernelized ridge regression with the hyperparameters determined in part a). Here we plot $\hat{f}_{poly}$ and $\hat{f}_{rbf}$ on a fine grid of 1000 points linearly spaced between 0 and 1.



(a) Polynomial Kernel (b) RBF Kernel

Figure 1: Kernelized Ridge Regression on $n = 30$ data points drawn from $4\sin(\pi x)\cos(6\pi x^2)$

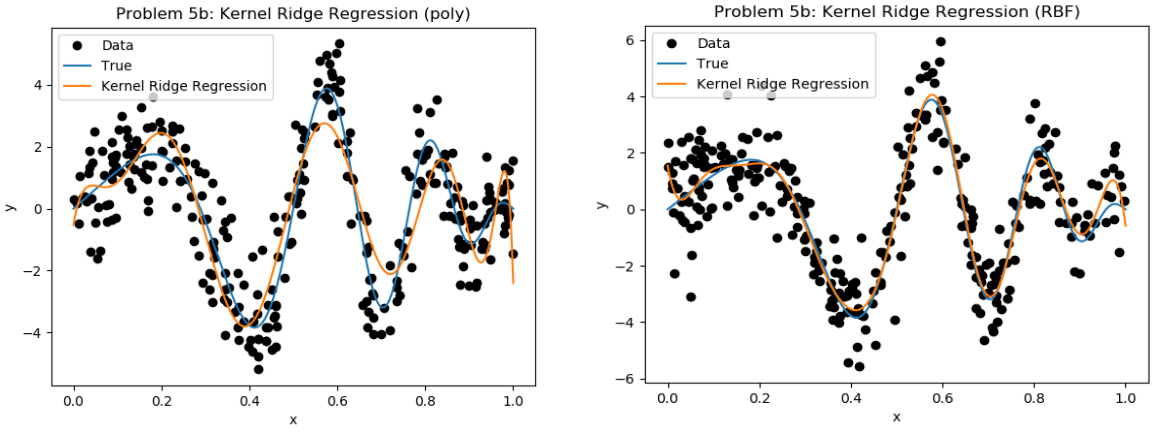
(c) Using 300 bootstrap samples, we generate a confidence interval of a sorts for $\hat{f}$ under both kernels.



(a) Polynomial Kernel (b) RBF Kernel

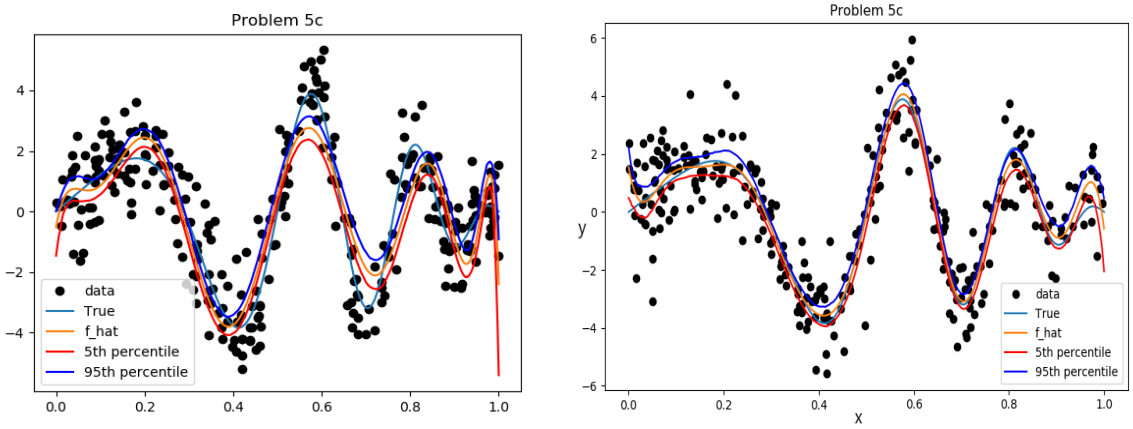
Figure 2: Kernelized Ridge Regression on $n = 30$ data points drawn from $4\sin(\pi x)\cos(6\pi x^2)$. Here, we plot the 95^{th} and 5^{th} percentile for $\hat{f}$ on our fine grid for both kernels.

(d) We repeat this procedure now with $n = 300$ data points. Rather than use LOO cross-validation, we use 10-fold cross-validation. This generates the parameters $\lambda = 10^{-6}, d = 18$ for the polynomial kernel and $\lambda = 1.27 * 10^{-5}, \gamma = 21.21$ for the RBF kernel.



(a) Polynomial Kernel (b) RBF Kernel

Figure 3: Kernelized Ridge Regression on $n = 300$ data points drawn from $4\sin(\pi x)\cos(6\pi x^2)$



(a) Polynomial Kernel (b) RBF Kernel

Figure 4: Kernelized Ridge Regression on $n = 300$ data points drawn from $4\sin(\pi x)\cos(6\pi x^2)$. Here, we plot the 95^{th} and 5^{th} percentile for $\hat{f}$ on our fine grid for both kernels.

(e) Using the results from part d we constructed a confidence interval for $E[(Y - \hat{f}_{poly}(X))^2 - (Y - \hat{f}_{rbf}(X))^2]$.

$$(5^{th} \text{ percentile}, 95^{th} \text{ percentile}) = (0.35, 0.48)$$

Using this confidence interval we can see that the RBF kernel performs significantly better than the Polynomial kernel for kernelized ridge regression as the confidence interval for $E[(Y - \hat{f}_{poly}(X))^2 - (Y - \hat{f}_{rbf}(X))^2]$ is completely above zero. This provides evidence that $E[(Y - \hat{f}_{poly}(X))^2] > E[(Y - \hat{f}_{RBF}(X))^2]$, meaning that the expected squared-error in prediction for the model with the Polynomial Kernel is greater than that for the RBF kernel.

Problem 5 Code

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def generate_data(n):
    '''
    Generates n random datapoints with the x values distributed uniformly on [0,1].
    y = f(x) + N(0,1)
    '''

    #np.random.seed(1234) # for reproducibility

    f = lambda x: 4*np.sin(np.pi*x)*np.cos(6*np.pi* x ** 2)
    x = np.random.uniform(low=0, high=1, size=(n, ))
    error = np.random.normal(size=(n, ))
    y = f(x) + error
    return (x,y)

# ========================================================================


def find_K(kernel_function, x, n):
    '''
    Finds the kernel matrix K for the specified data (x) and kernel function
    '''
    return np.fromfunction(lambda i, j: kernel_function(x[i], x[j]), shape=(n,n), dtype=int)


def kernel_ridge_regress(x, y, kf, lam, n):
    '''

    Returns the function f obtained via kernelized ridge regression

    '''
    K = find_K(kernel_function=kf, x=x, n=n)

    alpha_hat = np.linalg.solve(K + lam * np.eye(n), y)

    f = lambda z: np.sum(np.dot(alpha_hat, kf(z, x)))
    return f

# Part a)
def error_loo_tunning(x, y, kernel_function, lam_vals, hyperparam_vals, n):
    '''
    PART A)

    Runs Leave One Out cross-validation to calculate the optimal hyperparameters for
    kernel ridge regression using the specified kernel function.

    :param x: input data
    :param y: input labels
    :param kf: kernel function
    :param lam_vals: lambda values to cross validate at
    :param hyperparam_vals: hyperparameter (d or gamma) values to cross validate at
    :param n: number data points
    :return: best parameter choice
    '''

    errors = np.empty((len(lam_vals), len(hyperparam_vals)))
    for row, lam in enumerate(lam_vals):
        for col, hyperparam in enumerate(hyperparam_vals):
            error_loo = 0
            for j in range(n):
                x_val = x[j]
                y_val = y[j]
                x_train = np.concatenate((x[:j], x[j+1:]))
                y_train = np.concatenate((y[:j], y[j+1:]))

                kf = lambda x, xprime: kernel_function(x, xprime, hyperparam)
                f = kernel_ridge_regress(x=x_train, y=y_train, kf=kf, lam=lam, n=n-1)

                error_loo += (y_val - f(x_val)) ** 2
```

```
            error_loo = error_loo / n
            errors[row, col] = error_loo


    #fig = plt.figure(1)
    #ax = fig.gca(projection='3d')

    #X, Y = np.meshgrid(lam_vals, hyperparam_vals)
    #ax.plot_surface(X, Y, errors.T)
    #plt.xlabel('Lambda Value')
    #plt.ylabel('hyperparameter value')
    #ax.set_zlabel('Error (leave one out)')
    #plt.show()

    lam_ind, h_ind = np.unravel_index(np.argmin(errors, axis=None), errors.shape)
    return (lam_vals[lam_ind], hyperparam_vals[h_ind])



# Part b)
def part_b(x,y, kf, lam, hyperparameter, n, kernel_name):
    '''
    PART B)

    Plots the original data (x,y) and the function f_hat (determined through kernel ridge
                                                          regression) evaluated
    at each point in a fine grid. Also plots the true distribution.
    '''
    plt.figure()
    f_true = lambda x: 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x ** 2)

    x_fine, y_kernel = find_fhat(x=x,y=y,kf=kf,hyperparameter=hyperparameter, lam=lam, n=n)
    plt.plot(x, y, 'ko')
    plt.plot(x_fine, f_true(x_fine))

    plt.plot(x_fine, y_kernel)
    plt.legend(['Data', 'True', 'Kernel Ridge Regression'])
    plt.ylabel('y')
    plt.xlabel('x')
    plt.title('Problem 5b: Kernel Ridge Regression (' + kernel_name + ')')
    plt.show()

def find_fhat(x,y,kf, hyperparameter, lam, n, num_points=1000):
    '''
    Returns the values of f_hat evaluated at each point in the fine grid
    '''


    f_hat = kernel_ridge_regress(x=x,y=y, lam=lam, kf=lambda x, xprime: kf(x, xprime,
                                                          hyperparameter), n=n)

    x_fine = np.linspace(0,1, num_points)
    y_kernel = [f_hat(xi) for xi in x_fine]
    return x_fine, y_kernel

def part_c(x,y, kf, hyperparameter, lam, n, num_points=1000, B=300):
    '''
    PART C)

    Run bootstrap to generate an approximate range (5th percentile and 95th percentile) for
                                                          the predictions of
    f_hat at each point in the fine grid (between [0,1] with num_points linearly spaced points
                                                          )

    :param x: input data
    :param y: input labels
    :param kf: kernel function
    :param lam: lambda value to use for kernel ridge regression
    :param hyperparam: hyperparameter (d or gamma) to use for kernel ridge regression
    :param n: number data points
    :param num_points: number of points to use in fine grid to plot f_hat
    :param B: number of bootstrap samples to use
    '''
    fhat = np.zeros((B, num_points))
```

```
    for b in range(B):

        print(b)

        # sample from the data with replacement
        indices = np.random.choice(list(range(n)), n, replace=True)
        x_b = x[indices]
        y_b = y[indices]

        _ , fhat[b, :] = find_fhat(x=x_b, y=y_b, kf=kf, hyperparameter=hyperparameter, lam=lam
                                                        , n=n, num_points=num_points)

    percentile_5 = np.percentile(fhat, q=5, axis=0)
    percentile_95 = np.percentile(fhat, q=95, axis=0)

    f_true = lambda x: 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x ** 2)

    x_fine, y_kernel = find_fhat(x,y,kf,hyperparameter, lam, n)

    plt.figure()
    plt.plot(x, y, 'ko')
    plt.plot(x_fine, f_true(x_fine))
    plt.plot(x_fine, y_kernel)
    plt.plot(x_fine, percentile_5, 'r-')
    plt.plot(x_fine, percentile_95, 'b-')
    plt.legend(['data', 'True', 'f_hat', '5th percentile', '95th percentile'])
    plt.title('Problem 5c')
    plt.show()


def ten_fold_CV(x, y, kernel_function, lam_vals, hyperparam_vals, n):
    '''

    Runs 10-fold cross-validation to estimate the parameters of the given kernel function that
    minimize the validation error

    :param x: input data
    :param y: input labels
    :param kf: kernel function
    :param lam_vals: lambda values to cross validate at
    :param hyperparam_vals: hyperparameter (d or gamma) values to cross validate at
    :param n: number data points
    :return: best parameter choice
    '''

    errors = np.empty((len(lam_vals), len(hyperparam_vals)))
    for row, lam in enumerate(lam_vals):
        for col, hyperparam in enumerate(hyperparam_vals):
            error_CV = 0
            for j in range(10):
                start_val = j* (n//10)
                end_val = (j+1) * (n //10)
                x_val = x[start_val:end_val]
                y_val = y[start_val:end_val]
                x_train = np.concatenate((x[:start_val], x[end_val:]))
                y_train = np.concatenate((y[:start_val], y[end_val:]))

                kf = lambda x, xprime: kernel_function(x, xprime, hyperparam)
                f = kernel_ridge_regress(x=x_train, y=y_train, kf=kf, lam=lam, n=(n - n//10) )

                error_CV += 1/x_val.size * np.sum(np.square((y_val - np.asarray([f(x) for x in
                                                                x_val]))))

            errors[row, col] = error_CV


    #fig = plt.figure(1)
    #ax = fig.gca(projection='3d')

    #X, Y = np.meshgrid(lam_vals, hyperparam_vals)
    #ax.plot_surface(X, Y, errors.T)
    #plt.xlabel('Lambda Value')
    #plt.ylabel('hyperparameter value')
    #ax.set_zlabel('Error (leave one out)')
    #plt.show()
```

```
    lam_ind, h_ind = np.unravel_index(np.argmin(errors, axis=None), errors.shape)
    return (lam_vals[lam_ind], hyperparam_vals[h_ind])

#                                                   ============================================================


# parts a,b,c)
# part a is mostly accomplished by the function error_loo_tuning
# part b,c are accomplished by the part_b(), part_c() functions repspectively.

# part i) Polynomial Kernel
n = 30
x, y = generate_data(n)

lam_vals = [10**k for k in np.linspace(-6, 1, 25)]  # all lambda values of interest
d_vals = list(range(1, 11, 1))
kf_p = lambda x, xprime, d: np.power((1 + x * xprime), d)
best_lam, best_d = error_loo_tunning(x=x, y=y, kernel_function=kf_p, lam_vals=lam_vals,
                                                    hyperparam_vals=d_vals, n=n)

print("Polynomial Kernel Function \n")
print("best lambda: ", best_lam, " best d: ", best_d)

part_b(x=x, y=y, kf=kf_p, lam=best_lam, hyperparameter=best_d, n=n, kernel_name='poly')
part_c(x=x, y=y, kf=kf_p, lam=best_lam, hyperparameter=best_d, n=n, num_points=1000, B=300)


# part ii) RBF Kernel

gamma_ballpark = 1 / np.median([(x[i] - x[j]) ** 2 for i in range(n) for j in range(n)])
print('gamma ballpark: ', gamma_ballpark) # find a ballpark estimate for gamma.

# use lambda values from above
gamma_vals = np.linspace(1, 25, 20)
kf_rbf = lambda x, xprime, gamma: np.exp(-gamma * np.power(x - xprime, 2))
best_lam, best_gamma = error_loo_tunning(x=x, y=y, kernel_function=kf_rbf, lam_vals=lam_vals,
                                                    hyperparam_vals=gamma_vals, n=n)

print("\n\nRBF Kernel Function \n")
print("best lambda: ", best_lam, " best gamma: ", best_gamma)

part_b(x=x, y=y, kf=kf_rbf, lam=best_lam, hyperparameter=best_gamma, n=n, kernel_name='RBF')
part_c(x=x, y=y, kf=kf_rbf, hyperparameter=best_gamma, lam=best_lam, n=n, num_points=1000, B=
                                                    300)



# ============================================================================================
# part d)


# Polynomial Kernel
n = 300
x, y = generate_data(n)

lam_vals = [10**k for k in np.linspace(-6, 1, 20)]  # all lambda values of interest
d_vals = list(range(1, 21, 1))
kf_p = lambda x, xprime, d: np.power((1 + x * xprime), d)

best_lam_poly, best_d = ten_fold_CV(x=x, y=y, kernel_function=kf_p, lam_vals=lam_vals,
                                                    hyperparam_vals=d_vals, n=n)
print("Polynomial Kernel Function \n")
print("best lambda: ", best_lam_poly, " best d: ", best_d)

part_b(x=x, y=y, kf=kf_p, lam=best_lam_poly, hyperparameter=best_d, n=n, kernel_name='poly')
part_c(x=x, y=y, kf=kf_p, lam=best_lam_poly, hyperparameter=best_d, n=n, num_points=1000, B=
                                                    300)



# RBF Kernel
# use lambda values from above
gamma_vals = np.linspace(1, 25, 20)
kf_rbf = lambda x, xprime, gamma: np.exp(-gamma * np.power(x - xprime, 2))
```

```
best_lam_rbf, best_gamma = ten_fold_CV(x=x, y=y, kernel_function=kf_rbf, lam_vals=lam_vals,
                                       hyperparam_vals=gamma_vals, n=n)
print("\n\nRBF Kernel Function \n")
print("best lambda: ", best_lam_rbf, " best gamma: ", best_gamma)
part_b(x=x, y=y, kf=kf_rbf, lam=best_lam_rbf, hyperparameter=best_gamma, n=n, kernel_name='RBF
                                       ')
part_c(x=x, y=y, kf=kf_rbf, hyperparameter=best_gamma, lam=best_lam_rbf, n=n, num_points=1000,
                                        B=300)


# ==========================================
# part e)
m = 1000
xm, ym = generate_data(m)

# reuse old data to generate f_hats
f_hat_poly = kernel_ridge_regress(x=x,y=y, lam=best_lam_poly, kf=lambda x, xprime: kf_p(x,
                                       xprime, best_d), n=n)
f_hat_rbf = kernel_ridge_regress(x=x,y=y, lam=best_lam_rbf, kf=lambda x, xprime: kf_rbf(x,
                                       xprime, best_gamma), n=n)

B = 300

bootstrap_values = np.zeros((B, ))

for b in range(B):
    indices = np.random.choice(list(range(m)), m, replace=True)
    x_b = xm[indices]
    y_b = ym[indices]

    poly_error = np.square(y_b - np.asarray([f_hat_poly(x) for x in x_b]))
    rbf_error = np.square(y_b - np.asarray([f_hat_rbf(x) for x in x_b]))
    bootstrap_values[b] = 1/m * np.sum(poly_error - rbf_error)


percentile_95 = np.percentile(bootstrap_values, q=95)
percentile_5 = np.percentile(bootstrap_values, q=5)
print('95th percentile: ', str(percentile_95))
print('5th Percentile: ', str(percentile_5))
```

K-Means Clustering

Problem 6

(a) Below we can see the objective value at each iteration and a visualization of the 10 centroids (reshaped into 28 x 28 images to match the natural format of the data). Clearly, the modes are capturing some of the digits.

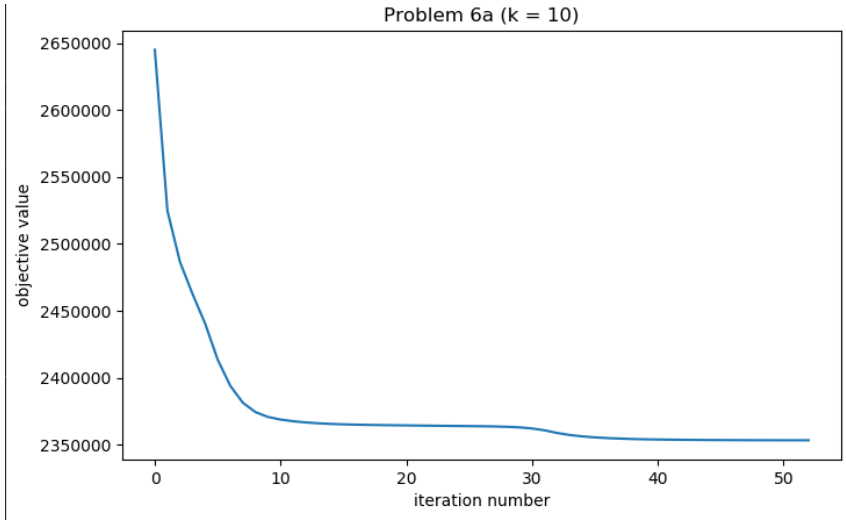


Figure 5: Objective value of K-means algorithm on MNIST as a function of iteration

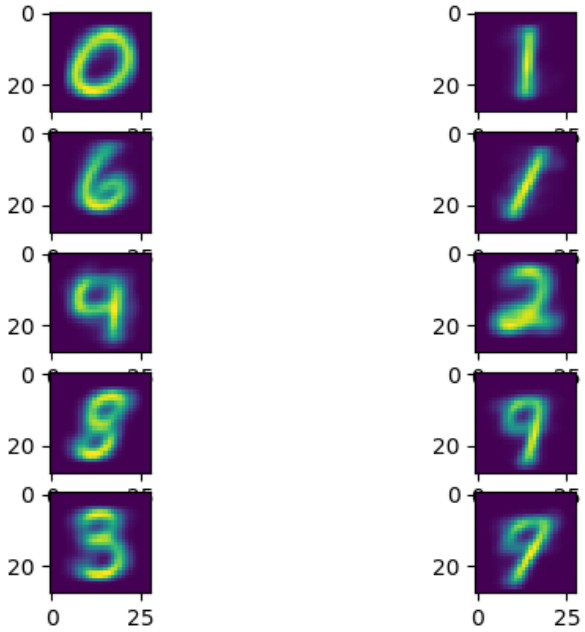


Figure 6: Centroids of K-means algorithm on MNIST

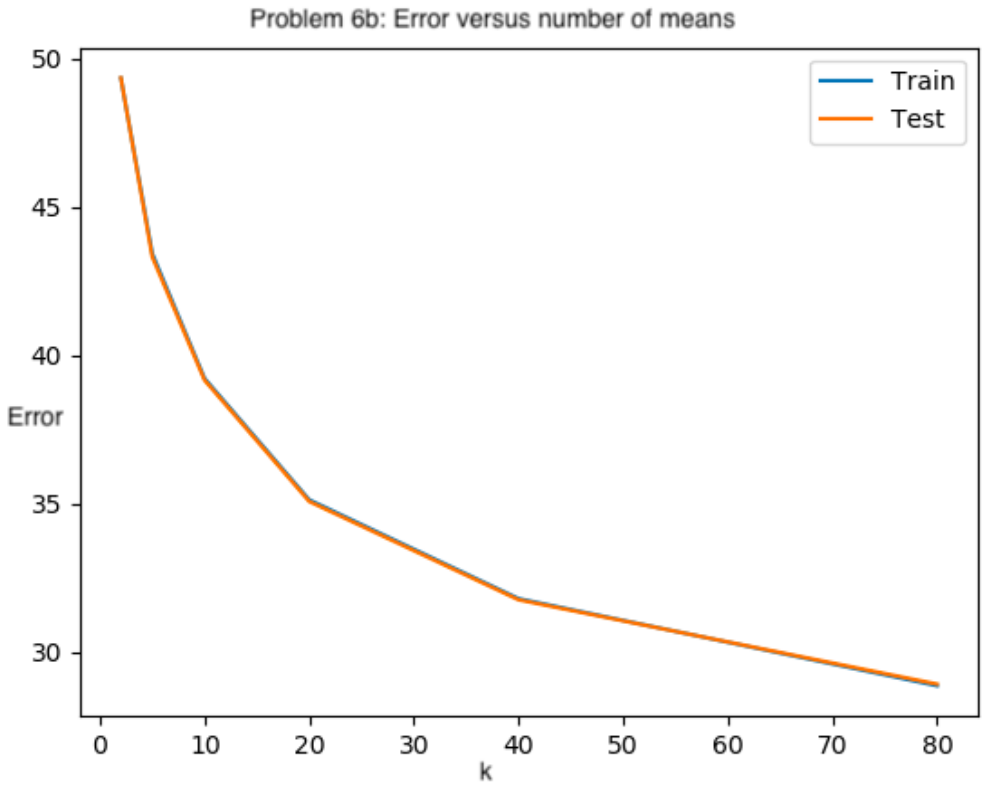


Figure 7: Training/Testing Error versus number of means used in K means. As expected, increasing the number of means naturally decreases the error (measured by distance of each point from its mean)

(b)

Problem 6 Code

```
'''
hw3_q3c.py

Given a dataset whose response variable is a class y = 1,2,3,..., k (i.e. a classification
                                                        problem with k classes)
determine the optimal classifier according to the loss functions specified in the homework
                                                        specification

'''

import numpy as np
import random
import matplotlib.pyplot as plt
from mnist.loader import MNIST


# Load MNIST Data
mndata = MNIST('./data')
X_train, labels_train = map(np.array, mndata.load_training())
X_test, labels_test = map(np.array, mndata.load_testing())
X_train = X_train / 255.0
X_test = X_test / 255.0

n = X_train.shape[0]
d = X_train.shape[1] # 28 x 28 = 784

#
                                                        ============================================================


def objective_function(X, partitions, MU):
    '''
        partitions is a dictionary mapping 1 through k to a set of indices (i.e. refering to
                                                                datapoints )
        closest to MU_i (where i matches the key of the dictionary)
        MU is a matrix where each row is one of the k means
    '''
    cost = 0
    for part in partitions:
        mu = np.expand_dims(MU[part, :], axis=0)
        for j in partitions[part]:
            temp1 = mu - X[j, :]
            temp2 = np.linalg.norm(temp1) ** 2
            cost += temp2
        #cost += np.sum(np.square(np.linalg.norm(mu - X[partitions[part]], axis=1)))
    return cost


def find_partitions(X, MU):
    n = X.shape[0]
    partitions = {}
    for i in range(n):
        x = np.expand_dims(X[i, :], axis=0)
        temp1 = MU - x
        temp2 = np.linalg.norm(temp1, axis=1)
        part = np.argmin(temp2)
        if part in partitions:
            partitions[part].append(i)
        else:
            partitions[part] = [i]

    return partitions



def k_means(X, k):
    # randomly choose the starting points in [0,1] (the range of the data); each mu_i is a row
                                                            in MU
    #MU = np.random.uniform(low=0, high=1, size=(k, d))

    # randomly choose the starting point to be a random data entry
    indices = np.random.choice(list(range(X.shape[0])), k, replace=False)
    MU = X[indices, :]
```

```
    prev_MU = np.zeros((k,d))
    delta = 1e-3
    obj_delta = 10

    objective_values = []
    itr = 0

    # stop when the MUs stop moving or the objective value stops changing
    while np.amax(np.abs(MU - prev_MU)) > delta and (len(objective_values) < 2 or abs(
                                                objective_values[-1] - objective_values[-2]
                                                ) > obj_delta):
        itr += 1
        prev_MU = np.copy(MU)
        #print("iteration number: ", itr)

        # find partitions
        partitions = find_partitions(X=X, MU=MU) # a dictionary that maps the part # (the i in
                                                        MU_i) to the index (the i in x_i)

        # Choose the new centroids
        for i in range(k):
            if i in partitions:
                MU[i, :] = 1/len(partitions[i]) * np.sum(X[partitions[i], :], axis=0)

            # else just leave the current MU as is

        objective_values.append(objective_function(X=X, partitions=partitions, MU=MU))

    #    print(objective_values[-1])

    plt.figure(1)
    plt.plot(objective_values)
    plt.xlabel("iteration number")
    plt.ylabel("objective value")
    plt.title('Problem 6a (k = 10)')
    plt.show()


    plt.figure(2)
    for i in range(k):
        plt.subplot(5,2,i+1)
        plt.imshow(np.reshape(MU[i, :], (28, 28)))

    plt.show()

    return MU, partitions

# Problem 6a)

k_means(X_train, 10)

# Problem 6b)

all_train_errors = []
all_test_errors = []
k_vals = [2,5
    ,10,20,40,80,160] #,640,1280]
for k in k_vals:
    print("Running at k = ", k)
    MU_k, partitions_k = k_means(X_train, k)
    train_error = 1/n * objective_function(X_train, partitions=partitions_k, MU=MU_k)

    # NOTE: partitions is made for X_train; have to many refind partition for X_test
    partitions_test_k = find_partitions(X=X_test, MU=MU_k)
    test_error = 1/X_test.shape[0] * objective_function(X_test, partitions=partitions_test_k,
                                                MU=MU_k)
    all_train_errors.append(train_error)
    all_test_errors.append(test_error)


plt.figure(3)
plt.plot(k_vals, all_train_errors)
plt.plot(k_vals, all_test_errors)
plt.legend(['Train', 'Test'])
plt.show()
```

Multivariate Gaussians

Problem 7

(a) Our goal is to find A such that $AA^T = \Sigma$. Note that since Σ is symmetric, its eigenvalue decomposition is:

$$\Sigma = Q\Lambda Q^T$$

For the orthogonal matrix Q whose columns are eigenvectors of Σ corresponding to the eigenvalues in the diagonal of Λ. The covariance matrix Σ is always positive semi-definite and hence its eigenvalues are all non-negative. Thus, the square root of the diagonal Λ matrix is well-defined. Hence:

$$\Sigma = Q\Lambda Q^T = Q(\sqrt{\Lambda})^2 Q^T = Q\sqrt{\Lambda}Q^T Q\sqrt{\Lambda}Q^T = Q\sqrt{\Lambda}Q^T(Q\sqrt{\Lambda}Q^T)^T$$

Above we use the fact that Q is orthogonal and thus $Q^T Q = I$ and that the transpose of a diagonal matrix is the matrix itself. Thus we can see the desired A matrix is:

$$A = Q\sqrt{\Lambda}Q^T$$

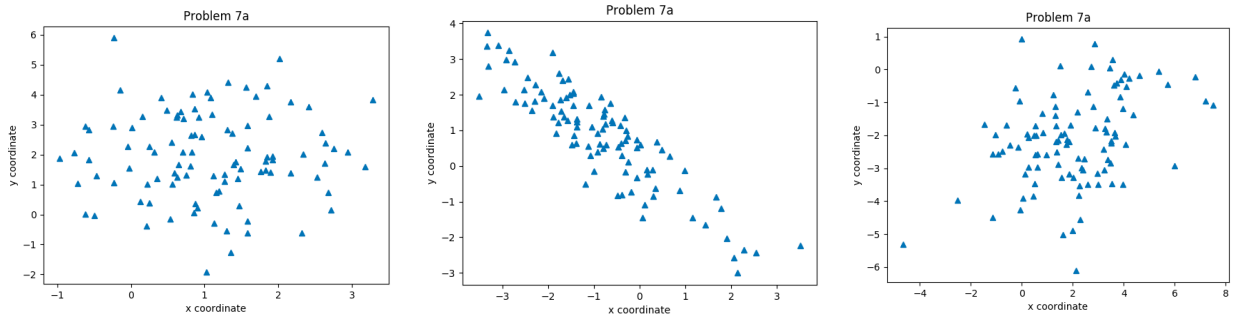


Figure 8: Problem 7a: 100 datapoints drawn from various multivariate gaussians. (i=1,2,3 from left to right)

(b) Below we see the relevant eigenvectors of the empirical covariance matrices.

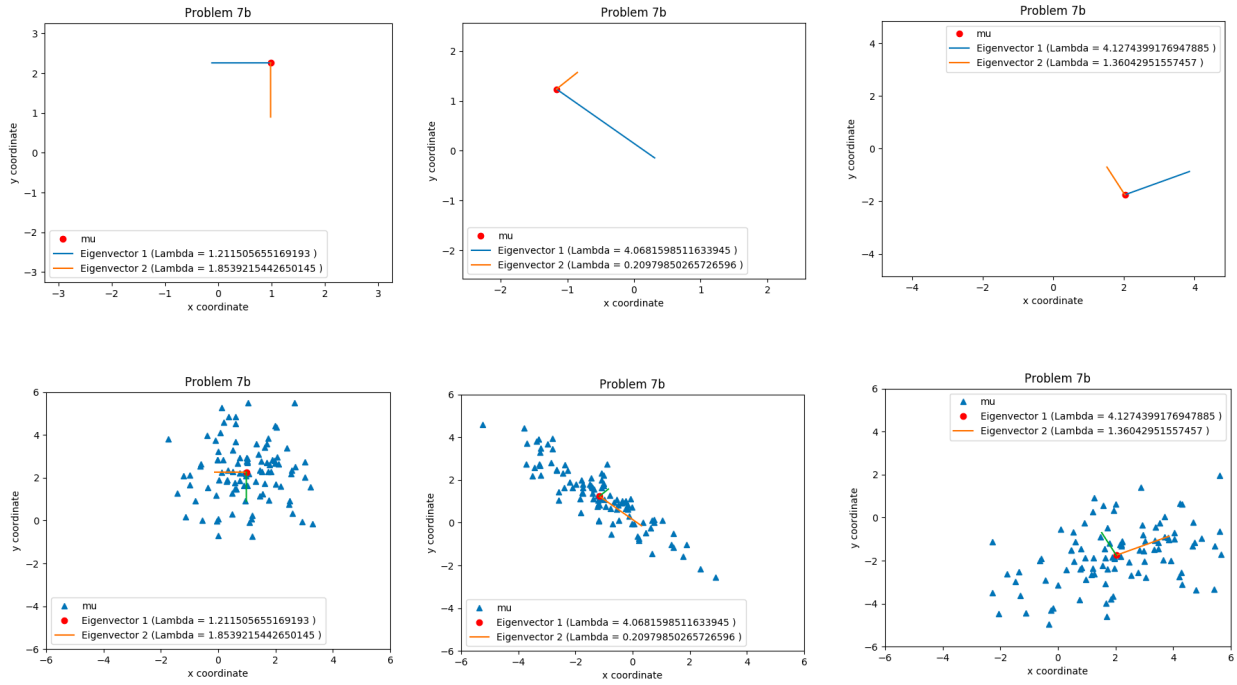


Figure 10: Problem 7b: 100 datapoints drawn from various multivariate gaussians. (i=1,2,3 from left to right). Here we plot the eigenvectors of the empirical covariance matrices for each of the datasets. Each eigenvector has norm equal to the square root of the eigenvalue (i.e. the singular value). The top row shows just the eigenvectors while the bottom row shows the eigenvectors plotted on top of the corresponding data.

(c) Finally, we can plot the transformed datapoints.

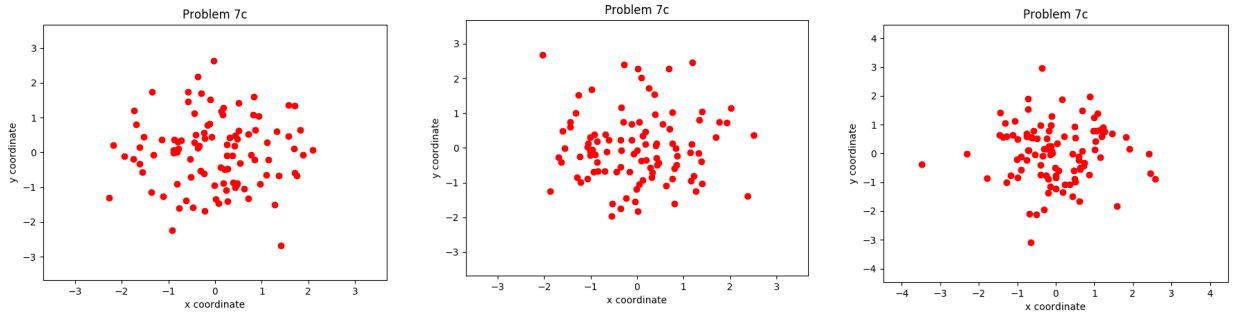


Figure 11: Problem 7a: 100 datapoints drawn from various multivariate gaussians. (i=1,2,3 from left to right). Here, we transform the datapoints in a way that generates a new uncorrelated dataset.

Problem 7 Code

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

# Problem 7


def plot_mvg(mu, Sigma, n):
    '''
    PART A)

    plots n datapoints drawn from the multivariate gaussian
    specified by the parameters mu and Sigma
    '''
    L, Q = np.linalg.eig(Sigma)

    # L is returned as an array and NOT a diagonal matrix, so
    # just convert it to a matrix.
    L = np.diag(L)
    A = np.matmul(np.matmul(Q, np.sqrt(L)), Q.T)

    np.random.seed(452254769) # for reproducibility

    Z = np.random.randn(2, n)
    X =  np.matmul(A, Z) + mu

    plt.figure(1)
    plt.plot(X[0, :], X[1, :], '^')
    plt.xlabel('x coordinate')
    plt.ylabel('y coordinate')
    plt.title('Problem 7a')
    plt.show()

    return X


def calc_mean_and_cov(X, n):
    '''
    PART B)

    Calculates an estimate of mu and the covariance matrix for the given data
    '''

    mu_hat = 1/n * np.sum(X, axis=1)
    mu_hat = np.expand_dims(mu_hat, axis=1)


    temp = X - mu_hat
    Sigma_hat = 1/(n-1) * sum([np.outer(temp[:, i], temp[:, i]) for i in range(n)])

    L, Q = np.linalg.eig(Sigma_hat)

    start = mu_hat
    end1 = mu_hat + np.expand_dims(np.sqrt(L[0]) * Q[:, 0], axis=1)
    end2 = mu_hat + np.expand_dims(np.sqrt(L[1]) * Q[:, 1], axis=1)
    points1 = np.hstack((start, end1))
    points2 = np.hstack((start, end2))

    plt.figure(1)
    plt.plot(mu_hat[0], mu_hat[1], 'ro')
    plt.plot(points1[0, :], points1[1, :])
    plt.plot(points2[0, :], points2[1, :])

    r = max(np.amax(np.abs(points1)), np.amax(np.abs(points2))) + 1
    limits = [-r, r]
    plt.xlim(limits)
    plt.ylim(limits)
    plt.legend(['mu', 'Eigenvector 1 (Lambda = ' + str(L[0]) + ' )', 'Eigenvector 2 (Lambda =
                                                ' + str(L[1]) + ' )',])
    plt.xlabel('x coordinate')
    plt.ylabel('y coordinate')
    plt.title('Problem 7b')

    plt.show()
```

```
    return mu_hat, Sigma_hat, L, Q


def part_c(X, mu_hat, L, Q):
    '''
    PART C)

    Transforms the data points based on the transformation defined in the problem
    and plots these transformed points.
    '''


    row1 = 1/np.sqrt(L[0]) * np.dot(Q[:, 0], X - mu_hat)
    row2 = 1/np.sqrt(L[1]) * np.dot(Q[:, 1], X - mu_hat)

    r = 1 + max(np.amax(np.abs(row1)), np.amax(np.abs(row2)))

    plt.figure(3)
    plt.plot(row1, row2, 'ro')
    plt.xlim([-r, r])
    plt.ylim([-r, r])
    plt.xlabel('x coordinate')
    plt.ylabel('y coordinate')
    plt.title('Problem 7c')
    plt.show()


# Part a) and b) and c)
# Here we just call the methods for part a,b,c above for each of the three
# different Gaussians (MU, Sigma pairs)
n = 100
mu1 = np.asarray([[1], [2]])
Sigma1 = np.asarray([[1, 0], [0, 2]])
X1 = plot_mvg(mu1, Sigma1, n)
mu1_hat, Sigma1_hat, L1, Q1 = calc_mean_and_cov(X1, n)
part_c(X1, mu1_hat, L1, Q1)


mu2 = np.asarray([[-1], [1]])
Sigma2 = np.asarray([[2, -1.8], [-1.8, 2]])
X2 = plot_mvg(mu2, Sigma2, n)
mu2_hat, Sigma2_hat, L2, Q2 = calc_mean_and_cov(X2, n)
part_c(X2, mu2_hat, L2, Q2)

mu3 = np.asarray([[2], [-2]])
Sigma3 = np.asarray([[3, 1], [1, 2]])
X3 = plot_mvg(mu3, Sigma3, n)
mu3_hat, Sigma3_hat, L3, Q3 = calc_mean_and_cov(X3, n)
part_c(X3, mu3_hat, L3, Q3)
```

1 Extra Credit

1.1 Problem 8

(a) Under H_0 it holds that $X \sim \mathcal{N}(0,1)$. We reject the null hypothesis iff $|X| \geq z_{\alpha/2}$. This implies that either $X \geq z_{\alpha/2}$ or $X \leq -z_{\alpha/2}$

$$\begin{aligned} P(|X| \geq z_{\alpha/2}) &= P(X \leq -z_{\alpha/2}) + P(X \geq z_{\alpha/2}) \\ &= \int_{-\infty}^{z_{\alpha/2}} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx + \int_{z_{\alpha/2}}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx \end{aligned}$$

Due to the symmetry of the normal distribution about its center μ, zero in this case, we know these two must be equal. Thus:

$$= 2 \int_{\frac{z_\alpha}{2}}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$$

By the definition of $z_{\alpha/2}$ we know this integral is $\frac{\alpha}{2}$. Thus:

$$P(|X| \geq z_{\alpha/2}) = \alpha$$

Thus the probability of rejecting the null given H_0 is α.

(b) No failing to reject the null hypothesis is not evidence to support H_0 is true as all our calculations are based on the assumption that H_0 is true. We cannot show evidence for something by assuming its truth. Intuitively, we can think of a hypothesis test as trying to determine whether a random variable X, our data, comes from the random distribution defined by H_0 or some other random distribution (with $\mu \neq 0$). This other distribution is unknown, but if we had to guess what it was, what we have learned in this class about Maximum Likelihood Estimation would lead us to guess $\mu = X$ for this new distribution. If we fail to reject the null, it is because these distributions are too close together. Thus, we cannot tell which one it really should be. This intuition is not completely rigourous, but its how I try to visualize this problem.

1.2 Problem 9

(a) Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ and $\mathcal{D}' = \{(x_i', y_i')\}_{i=1}^m\}$

$$E[\hat{p}_j] = \frac{1}{n}\sum_{\mathcal{D}} P(\mathcal{D}) \sum_{(x_i,y_i)\in\mathcal{D}} 1\{f_j(x_i) \neq y_i\}$$

$$= \frac{1}{n}\sum_{(x_1,y_1)}\sum_{(x_2,y_2)} ... \sum_{(x_n,y_n)} 1\{f_j(x_i) \neq y_i\} P((x_1,y_1),...,(x_n,y_n))$$

As the points are sampled independently from $\mathcal{P}$ this equals:

$$= \frac{1}{n}\sum_{i=1}^n \sum_{(x_1,y_1)}\sum_{(x_2,y_2)} ... \sum_{(x_n,y_n)} 1\{f_j(x_i) \neq y_i\} P((x_1,y_1))...P((x_n,y_n))$$

$$= \frac{1}{n}\sum_{i=1}^n \sum_{(x_1,y_1)} P((x_1,y_1)) \sum_{(x_2,y_2)} P((x_2,y_2))... \sum_{(x_n,y_n)} 1\{f_j(x_i) \neq y_i\} P((x_n,y_n))$$

As we did in homework 1, since $1\{f_j(x_i) \neq y_i\}$ only depends on x_i, y_i we can pull it out to the respective sum. Thus:

$$= \frac{1}{n}\sum_{i=1}^n \sum_{(x_1,y_1)} P((x_1,y_1))... \sum_{(x_i,y_i)} P((x_i,y_i)) 1\{f_j(x_i) \neq y_i\}... \sum_{(x_n,y_n)} P((x_n,y_n))$$

All the sums to the right of the i^{th} sum are just summing over a probability distribution and are thus equal to one.

$$= \frac{1}{n}\sum_{i=1}^n \sum_{(x_1,y_1)} P((x_1,y_1))... \sum_{(x_i,y_i)} P((x_i,y_i)) 1\{f_j(x_i) \neq y_i\}$$

This inner sum is just $E[1\{f_j(x_i) \neq y_i\}] = p_j$. Pulling this to the front leaves:

$$= \frac{p_j}{n}\sum_{i=1}^n \sum_{(x_1,y_1)} P((x_1,y_1))... \sum_{(x_{i-1},y_{i-1})} P((x_{i-1},y_{i-1}))$$

Again, now all the sums of (x, y) are just summing over probability distributions and thus sum to one. Thus:

$$= \frac{p_j}{n}\sum_{i=1}^n 1 = p_j$$

Identical logic can be used to show $E[\hat{p}_j'] = p_j'$

(b)

$$E[(\hat{p}_j - p_j)^2] = \sum_{\mathcal{D}} P(\mathcal{D}) \left(\frac{1}{n}\sum_{(x,y)\in\mathcal{D}} 1\{f_j(x_i) \neq y_i\} - p_j\right)^2$$

$$= \frac{1}{n^2}\sum_{\mathcal{D}} P(\mathcal{D}) \left(\sum_{(x,y)\in\mathcal{D}} (1\{f_j(x_i) \neq y_i\} - p_j)\right)^2$$

Using the same trick we used above and using the fact that the data points (x_i, y_i) are sampled independently, we can separate this sum into a series of sums over each of the datapoints.

$$= \frac{1}{n^2}\sum_{i=1}^n \sum_{(x_1,y_1)} P((x_1,y_1))... \sum_{(x_i,y_i)} P((x_i,y_i)) \left(1\{f_j(x_i) \neq y_i\} - p_j\right)^2 ... \sum_{(x_n,y_n)} P((x_n,y_n))$$

Again, the rightmost sums (past the i^{th} one) are just summing over probability distributions and thus they sum to one.

$$= \frac{1}{n^2} \sum_{i=1}^{n} \sum_{(x_1,y_1)} P((x_1,y_1)) ... \sum_{(x_i,y_i)} P((x_i,y_i)) \left((1\{f_j(x_i) \neq y_i\} - p_j)\right)^2$$

$$= \frac{1}{n^2} \sum_{i=1}^{n} \sum_{(x_1,y_1)} P((x_1,y_1)) ... \sum_{(x_{i-1},y_{i-1})} P((x_{i-1},y_{i-1})) E[(1\{f_j(x_i) \neq y_i\} - p_j)^2]$$

We know from the problem statement that $E[(1\{f_j(x_i) \neq y_i\} - p_j)^2] = p_j(1-p_j)$

$$= \frac{p_j(1-p_j)}{n^2} \sum_{i=1}^{n} \sum_{(x_1,y_1)} P((x_1,y_1)) ... \sum_{(x_{i-1},y_{i-1})} P((x_{i-1},y_{i-1}))$$

Again, all the remaining sums are just summing over probability distributions and thus they sum to one. Thus:

$$= \frac{p_j(1-p_j)}{n^2} \sum_{i=1}^{n} 1$$

$$\boxed{E[(\hat{p}_j - p_j)^2] = \frac{p_j(1-p_j)}{n}}$$

Identical logic can be used to show $E[(\hat{p}'_j - p_j)^2] = \frac{p_j(1-p_j)}{m}$

(c) $\hat{p}_j$ is the average of n independent trials of the random variable $1\{f_j(x) \neq y\}$ whose expectation is $E[1\{f_j(x) \neq y\}] = p_j$. By the law of larger numbers this average gets arbitrarily close to its expectation as $n \to \infty$. Thus this implies $\hat{p}_j \to p_j$ which also naturally implies $\hat{p}_j(1-\hat{p_j}) \to p_j(1-p_j)$.

(d) Under the null hypothesis, the expected value of $\hat{p}_1 - \hat{p}_2 = 0$. Furthemore, since $\hat{p}'_j$ and $\hat{p}_j$ are generated from separate independent samples from the distribution, the two statistics are independent. Therefore $Var[\hat{p}_j - \hat{p}'_j] = Var[\hat{p}_j] + Var[\hat{p}'_j]$. Using our approximations from part c, we can see that as θ is defined it holds.

$$\theta = \frac{(\hat{p}_1 - \hat{p}'_2) - E[\hat{p}_1 - \hat{p}'_2]}{\sqrt{Var[\hat{p}_1 - \hat{p}'_2]}}$$

From CLT we would expect this random variable θ to approach $\mathcal{N}(0,1)$ as the sample size gets sufficiently large ($m, n \to \infty$).

(e) The probability that $|\phi| \geq z_{\alpha/2}$ is no longer bounded by α as the calculations we performed in question 8 assumed the data was coming from $\mathcal{N}(0,1)$. With ϕ, the variance is no longer one: its $1-\rho$. As ρ can take on negative values ($\rho \in [-1,1]$) it is possible that these statistics have higher than one variance. Thus, a larger interval would be needed to bound $|\phi|$ with a rate of α.

(f) We see in the worst-case scenario $\phi \sim \mathcal{N}(0,2)$ when $\rho = -1$, as low as it can go. This is the highest variance possible, generating the Gaussian distribution with the largest tails. However, this implies that $\phi/\sqrt{2}$ is at most as wide as $\mathcal{N}(0,1)$ as $Var[\phi/\sqrt{2}] = \frac{1}{2}Var[\phi]$. Thus, $\phi \sim N(0, 1-\rho)$ under the null hypothesis and $\phi/\sqrt{2}$ is at most as wide as $\mathcal{N}(0,1)$, implying $|\phi|/\sqrt{2} \geq z_{\alpha/2}$ with rate at most α.

(g) I would argue the test using ψ is more powerful. The test using ϕ loses a lot of its power as we have to restrict its variance (by dividing by $\sqrt{2}$) to handle the worst case scenario when $\rho = -1$. In cases where $\rho > -1$, we have that the probability of rejecting the null hypothesis is actually less than α, and by lowering α we naturally decrease the power of the test: we need more evidence in order to reject the null, so we will certainly fail to reject the null incorrectly more often. However, $\rho = -1$ will not always be the case, and thus we are losing some power unnecessarily. ψ does not seem to suffer from the same pitfall.