Homework #3 - A

Spring 2020, CSE 446/546: Machine Learning
Richy Yun
Due: Thursday 5/28/2020 11:59 PM

Conceptual questions

A.1

a. *[2 points]* True or False: Given a data matrix $X \in \mathbb{R}^{n \times d}$ where d is much smaller than n, if we project our data onto a k dimensional subspace using PCA where $k = \text{rank}(X)$, our projection will have 0 reconstruction error (we find a perfect representation of our data, with no information loss).

True. Because $k = \text{rank}(X)$ we're not actually doing any dimensionality reduction, just a linear transformation.

b. *[2 points]* True or False: The maximum margin decision boundaries that support vector machines construct have the lowest generalization error among all linear classifiers.

False. SVM is not the most optimal linear method in every situation.

c. *[2 points]* True or False: An individual observation x_i can occur multiple times in a single bootstrap sample from a dataset X, even if x_i only occurs once in X.

True. Bootstrapping is sampling with replacement, so there is a chance to have the same observation multiple times.

d. *[2 points]* True or False: Suppose that the SVD of a square $n \times n$ matrix X is $USV^\top$, where S is a diagonal $n \times n$ matrix. Then the rows of V are equal to the eigenvectors of $X^\top X$.

False. As shown in section: $X^\top X = VS^\top U^\top USV^\top = VS^2V^\top$. Thus the columns of V is the eigenvectors of $X^\top X$, not the rows. X being a square matrix does not change this.

e. *[2 points]* True or False: Performing PCA to reduce the feature dimensionality and then applying the Lasso results in an interpretable linear model.

False. The new feature dimension is often not easily interpretable as each feature is now a weighted combination of all the original features. Although there may be some cases it's easy to interpret, it usually won't be.

f. *[2 points]* True or False: choosing k to minimize the k-means objective (see Equation (1) below) is a good way to find meaningful clusters.

False. The higher the k the lower the objective becomes due to the points being closer to centroids if there are simply more clusters. The k that absolutely minimizes the objective is to have as many clusters as data points so that each point is a cluster, which would give an objective of 0, but is clearly not a reasonable solution.

g. *[2 points]* Say you trained an SVM classifier with an RBF kernel ($K(u, v) = \exp(-\frac{||u,v||_2^2}{2\sigma^2})$). It seems to underfit the training set: should you increase or decrease σ?

You should decrease σ. If the classifier was underfit it means the classification was not as specific to the train data. A larger σ amplifies the kernel values (distances between points) resulting in a more specific (more overfit) classification.

Kernels and Bootstrap

A.2 *[5 points]* We can write the dot product as:

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

The step that removes the sum is due to the Taylor series expansion of e^x. Thus, $K(x, x')$ is a kernal function of this feature map.

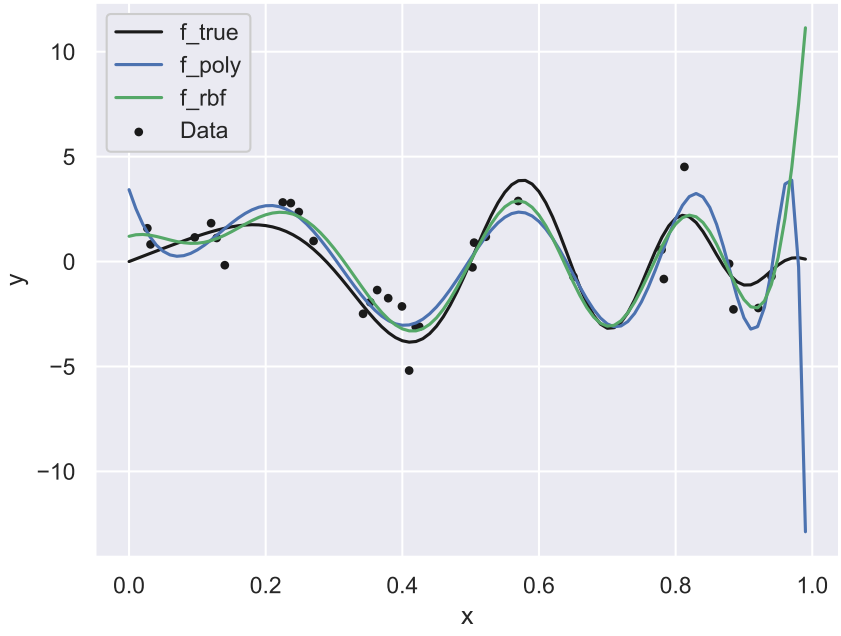
A.3

a. *[10 points]* To implement we first find the gradient of the objective with respect to α:

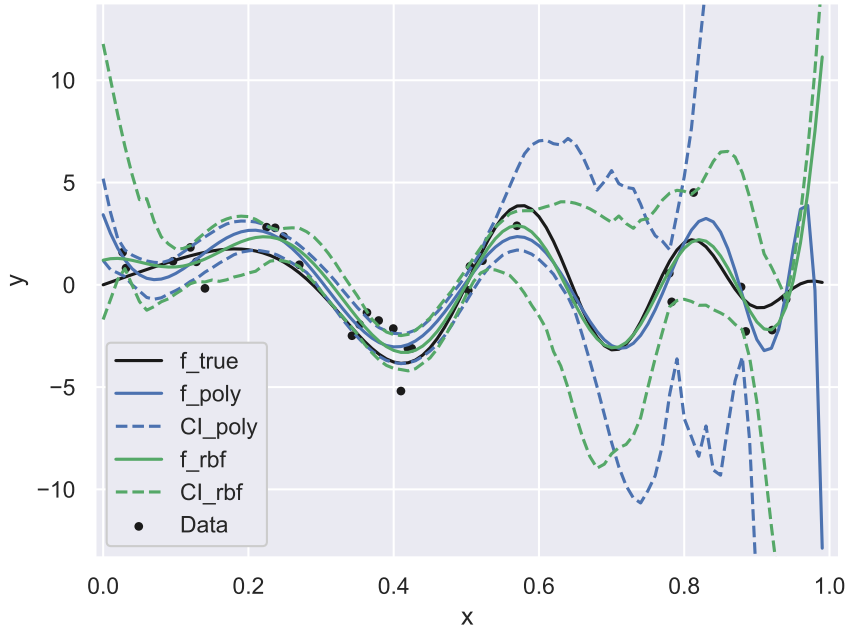
$$\begin{aligned}
\nabla_\alpha(||K\alpha - y||^2 + \lambda \alpha^\top K\alpha) &= 2K(K\alpha - y) + 2\lambda K\alpha \\
0 &= 2K(K + \lambda I)\alpha - 2Ky \\
\alpha &= (K + \lambda I)^{-1} y
\end{aligned}$$

Due to the small sample size of $n = 30$ the optimal values vary a lot trial by trial. Various trials of cross validation was performed with grid search over 50 values each of $\lambda = [1e-8, 1]$, $\doteq [1, 50]$, and $\gamma = [1, 25]$ For k_{poly}. Although it is difficult to get "the" optimal parameters, for the purposes of the problem I used what seemed to be the approximate median values that avoided overfitting: $\lambda = 1\text{e}{-4}$, $d = 25$, and $\gamma = 20$.

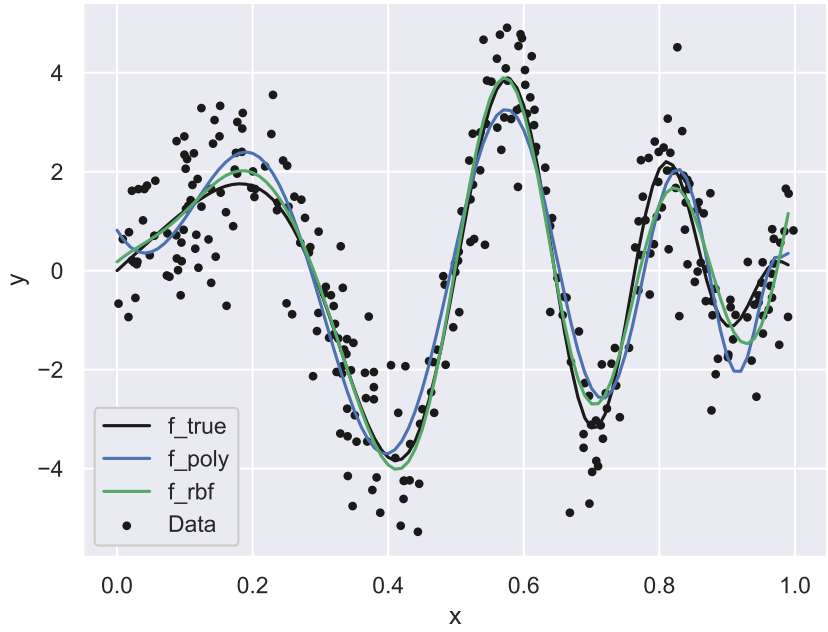
b. *[10 points]* The curve predicts the true function fairly well. As expected the extremes of the function vary a lot due to both lack of data at the extremes and the slight overfitting that occurs.

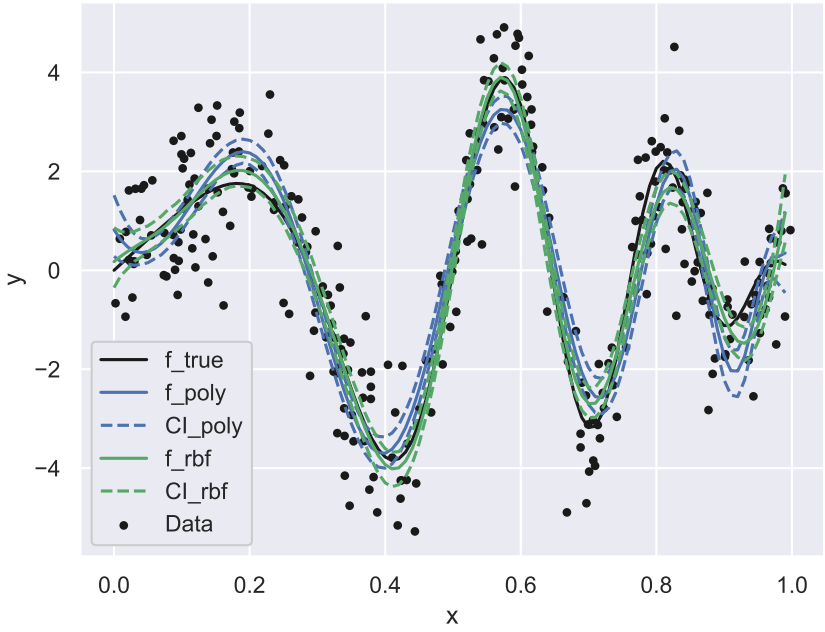


c. *[5 points]* Although the curves seem to fit fairly well, the 5 and 95% confidence intervals tell a different story. The interval is very tight where there is a lot of data points, suggesting the curve was fit very well at those spots, but begin diverging very quickly. The confidence intervals at $x = 1$ (cut off because of the zoom) go to ± 100.



d. *[5 points]* For $n = 300$ the optimal hyperparameters were: $\lambda = 5\text{e}{-3}$, $d = 30$, and $\gamma = 25$. Clearly the predictor does a much better job as it was trained with 10 times the amount of data. The confidence intervals are very tight across all x as well.





e. *[5 points]* The 5 and 95% confidence interval obtained were -0.4372 and 0.2371 respectively for this specific trial. Since the confidence interval contains 0 (which would be when the squared error is the same for both techniques on average), there is insignificant statistical evidence to suggest one of $\hat{f}_{poly}$ and $\hat{f}_{rbf}$ is the better predictor.

Source Code A.3a

```
import numpy as np
import scipy.spatial.distance as sp
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

def fstar(x):
    return 4*np.sin(np.pi*x)*np.cos(6*np.pi*np.square(x))

def kpoly(x1, x2, d):
    temp = np.matmul(x1, np.transpose(x2))
    return np.power(1 + temp, d)

def krbf(x1, x2, g):
    dist = sp.cdist(x1, x2, 'euclidean')
    return np.exp(-g*np.square(dist))

def err(f, y):
    return np.mean(np.square(f - y))

def alpha(k, l, y):
    return np.matmul(np.linalg.inv(k+l*np.eye(len(k))), y)

n = 300
```

```
cv = 10  # cross validation width
x = np.random.uniform(0, 1, (n, 1))
e = np.random.normal(0, 1, (n, 1))
y = fstar(x) + e

# Set range of hyperparameters
hp = np.linspace(1, 50, 50)  # for rbf
#hp = np.linspace(10, 40, 50)  # for poly

# Set range of lambda
lambdas = np.logspace(-8, -1, 50)
errors = np.zeros((len(lambdas), len(hp)))

# Loop through all parameters
for l in range(len(lambdas)):
    lam = lambdas[l]
    for h in range(len(hp)):
        print(l, h)
        errs = np.zeros(n)
        allK = krbf(x, x, hp[h])

        # Cross validation
        for i in range(int(n/cv)):
            inds = np.arange((i-1)*cv, i*cv, 1)
            tempx = np.delete(x, inds, axis=0)
            tempy = np.delete(y, inds, axis=0)
            K = krbf(tempx, tempx, hp[h])
            a = alpha(K, lam, tempy)

            K = krbf(np.expand_dims(x[inds, 0], axis=1), tempx, i)
            f = np.dot(K, a)

            errs[i] = np.sum(np.square(f-y[inds]))

        # Calculate total error
        errors[l, h] = np.sum(errs)/n

# Find optimal hyperparameters
temp = np.where(errors == np.min(errors))
optlam = lambdas[temp[0][0]]
opthp = hp[temp[1][0]]

# Find optimal alpha
K = krbf(x, x, opthp)
a = alpha(K, optlam, y)

# Get f and fhat
plotx = np.arange(0, 1, 0.01)
plotx = np.expand_dims(plotx, axis=1)
plotk = krbf(plotx, x, opthp)
fhat = np.matmul(plotk, a)
f = fstar(plotx)

# Plot
2plt.scatter(x, y, 10, 'k')
```

```
plt.plot(plotx, f)
plt.plot(plotx, fhat)
plt.xlabel('x')
plt.ylabel('y')
plt.legend(('f_true', 'f_poly', 'f_rbf', 'Data'))
plt.show()

plt.imshow(errors, origin='lower')
plt.colorbar()
plt.xlabel('hyperparameter')
plt.ylabel('lambda')
plt.show()
```

Source Code A.3bc

```
import numpy as np
import scipy.spatial.distance as sp
import matplotlib.pyplot as plt
import seaborn as sns

sns.set()

def fstar(x):
    return 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * np.square(x))

def kpoly(x1, x2, d):
    temp = np.matmul(x1, np.transpose(x2))
    return np.power(1 + temp, d)

def krbf(x1, x2, g):
    dist = sp.cdist(x1, x2, 'euclidean')
    return np.exp(-g * np.square(dist))

def err(f, y):
    return np.mean(np.square(f - y))

def alpha(k, l, y):
    return np.matmul(np.linalg.inv(k + l * np.eye(len(k))), y)

# initialize variables
n = 300
lam = 5e-3
beta = 30
gamma = 25
boot = 300
x = np.random.uniform(0, 1, (n, 1))
e = np.random.normal(0, 1, (n, 1))
y = fstar(x) + e
finex = np.arange(0, 1, 0.01)
finex = np.expand_dims(finex, axis=1)
f = fstar(finex)

# bootstrap
CI_poly = np.zeros((len(finex), boot))
CI_rbf = np.zeros((len(finex), boot))
```

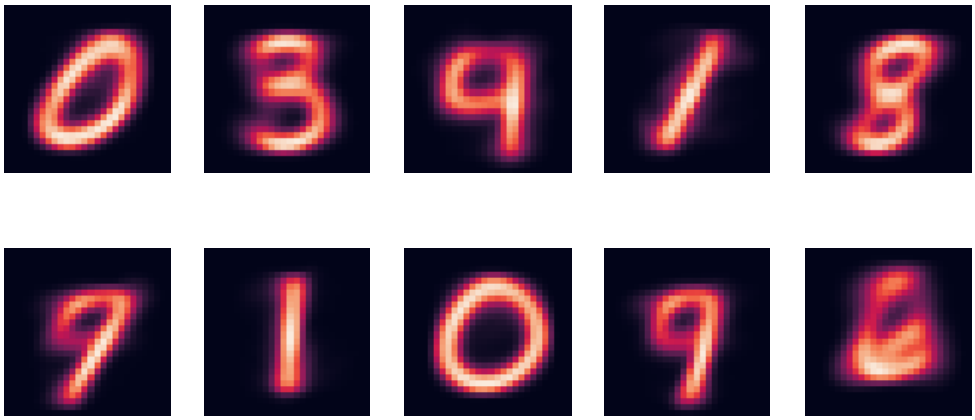
```
errors = np.zeros(boot)
for b in range(boot):
    ind = np.random.choice(range(n), n)
    newx = x[ind]
    newy = y[ind]
    K = kpoly(newx, newx, beta)
    apoly = alpha(K, lam, newy)
    K = krbf(newx, newx, gamma)
    arbf = alpha(K, lam, newy)
    plotk = kpoly(finex, newx, beta)
    CI_poly[:, b] = np.squeeze(np.matmul(plotk, apoly))
    plotk = krbf(finex, newx, gamma)
    CI_rbf[:, b] = np.squeeze(np.matmul(plotk, arbf))
    errors[b] = np.mean(np.square(f-CI_poly[:, b])-np.square(f-CI_rbf[:, b]))

# optimal curves
K = kpoly(x, x, beta)
apoly = alpha(K, lam, y)
K = krbf(x, x, gamma)
arbf = alpha(K, lam, y)

# plot variables
plotk = kpoly(finex, x, beta)
fhatpoly = np.matmul(plotk, apoly)
plotk = krbf(finex, x, gamma)
fhatrbf = np.matmul(plotk, arbf)

# plot
fig1 = plt.figure()
plt.scatter(x, y, 10, 'k')
plt.plot(finex, f, 'k')
plt.plot(finex, fhatpoly, 'b')
plt.plot(finex, fhatrbf, 'g')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(('f_true', 'f_poly', 'f_rbf', 'Data'))

# plot with confidence intervals
fig2 = plt.figure()
scatter1 = plt.scatter(x, y, 10, 'k', label='Data')
plt.plot(finex, f, 'k', label='f_true')
plt.plot(finex, fhatpoly, 'b', label='f_poly')
perc_poly = np.percentile(CI_poly, [5, 95], axis=1)
plt.plot(finex, perc_poly[0, :], 'b--', label='CI_poly')
plt.plot(finex, perc_poly[1, :], 'b--')
plt.plot(finex, fhatrbf, 'g', label='f_rbf')
perc_rbf = np.percentile(CI_rbf, [5, 95], axis=1)
plt.plot(finex, perc_rbf[0, :], 'g--', label='CI_rbf')
plt.plot(finex, perc_rbf[1, :], 'g--')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()

perc_err = np.percentile(errors, [5, 95])
```

k-means clustering

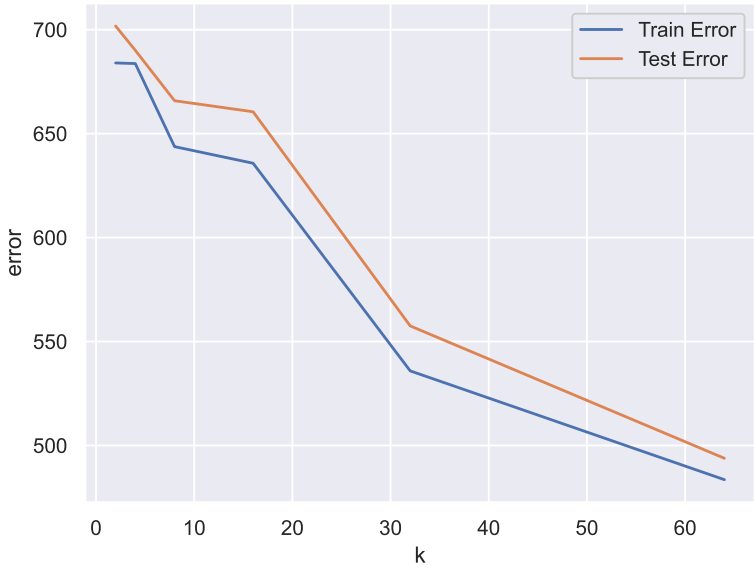
A.4

a. *[5 points]* See code at the end of problem for implementation.

b. *[5 points]* The 10 centroids look like:



c. *[5 points]* The errors with respect to k:



Source Code A.4

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

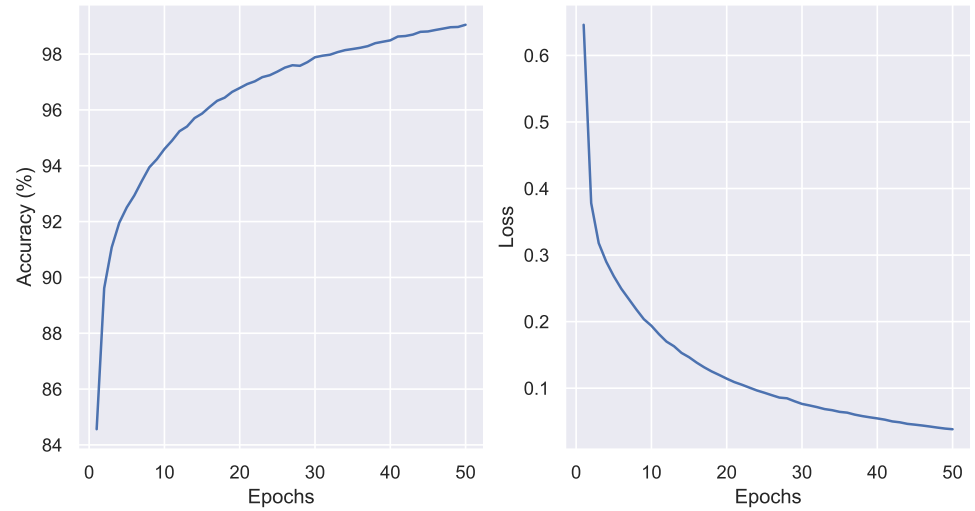
```
import scipy.spatial.distance as sp
import scipy.stats as st
from mnist import MNIST
import seaborn as sns
sns.set()

# kmeans++ initialization
def kmeansplusplus(data, k):
    n = np.size(data, axis=0)
    ind = np.random.choice(range(n))
    centroid = np.expand_dims(data[ind, :], axis=0)
    for c in range(k - 1):
        d = sp.cdist(data, centroid, 'euclidean') ** 2
        d = np.min(d, axis=1)
        d = d / np.sum(d)
        ind = np.random.choice(range(n), 1, p=d)
        centroid = np.append(centroid, data[ind, :], axis=0)
    return centroid

def kmeans(data, k):
    centroid = kmeansplusplus(data, k)
    maxchange = 100
    while maxchange > 0.001:
        d = sp.cdist(data, centroid, 'euclidean')
        clusters = np.argmin(d, 1)
        newcentroid = np.copy(centroid)
        for c in range(k):
            newcentroid[c, :] = np.mean(data[clusters == c, :], axis=0)
        maxchange = np.max(np.mean(np.abs(newcentroid - centroid), axis=1))
        centroid = np.copy(newcentroid)
        print(maxchange)

    d = sp.cdist(data, centroid, 'euclidean')
    clusters = np.argmin(d, 1)

    return centroid, clusters

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

# Rearrange data
X_train = X_train/255.0
X_test = X_test/255.0

# Plot centroids
k = 10
cent, clust = kmeans(X_train, k)
for i in range(k):
    plt.subplot(2, k/2, i+1)
    plt.imshow(np.reshape(cent[i, :], (28, 28)))
    plt.axis('off')

# Train and test error as a function of k
```

```
k = [2, 4, 8, 16, 32, 64]
trainerr = np.zeros(len(k))
testerr = np.zeros(len(k))
for i in range(len(k)):
    cent, clust = kmeans(X_train, k[i])
    d = sp.cdist(X_train, cent, 'euclidean')
    clusters = np.argmin(d, 1)
    trainerr[i] = np.mean(np.square(np.sum(X_train - cent[clusters, :], axis=1)))
    d = sp.cdist(X_test, cent, 'euclidean')
    clusters = np.argmin(d, 1)
    testerr[i] = np.mean(np.square(np.sum(X_test - cent[clusters, :], axis=1)))

plt.plot(k, trainerr)
plt.plot(k, testerr)
plt.xlabel('k')
plt.ylabel('error')
plt.legend(('Train Error', 'Test Error'))
```

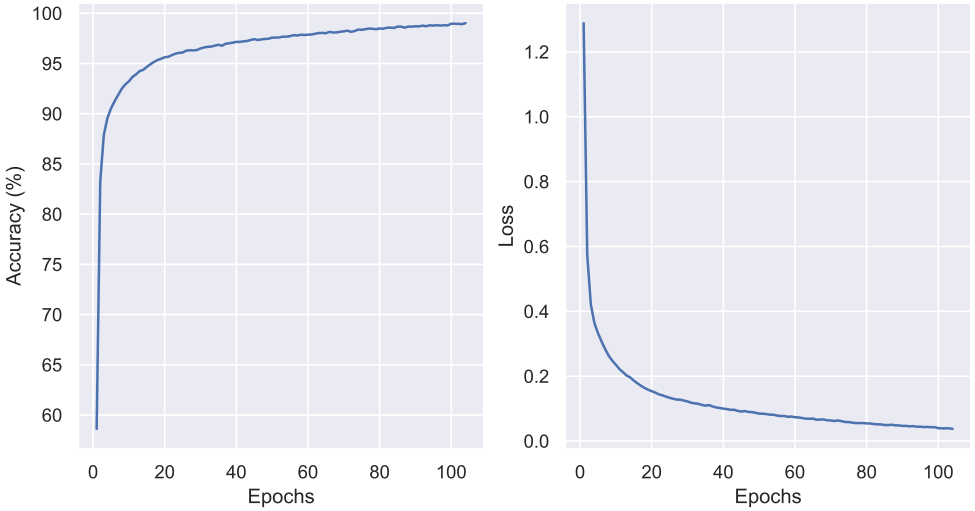
Neural Networks for MNIST

A.5

a. *[10 points]* For both a and b I used a learning rate of 1e-3 and mini-batch of 1000 samples.



b. *[10 points]*



c. *[5 points]* For (a) there are: $64 \times 784 + 64 + 10 \times 64 + 10 = 50890$ parameters. For (b) there are: $32 \times 784 + 32 + 32 \times 32 + 32 + 10 \times 32 + 10 = 26506$ parameters, around half of (a). It's difficult to determine which method is explicitly "better" than the other. Although the wide shallow (a) method reaches 99% accuracy about twice as fast as the narrow deep (b) method, they do both ultimately reach it. On the other hand, the narrow deep method uses about half of the parameters the wide shallow method does meaning it doesn't require as much memory. Fewer parameters also often means less overfitting. Thus the "better" method doesn't depend on accuracy, but rather on your prioritization between speed and memory/generalization.

Source Code A.5

```
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST
import torch
import torch.nn.functional as F
import seaborn as sns

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

# Rearrange data
X_train = X_train/255.0
X_train = torch.from_numpy(X_train)
X_train = X_train.type(torch.FloatTensor)
X_test = X_test/255.0
X_test = torch.from_numpy(X_test)
X_test = X_test.type(torch.FloatTensor)

Y_train = torch.from_numpy(labels_train)
Y_train = Y_train.type(torch.long)
Y_test = torch.from_numpy(labels_test)
Y_test = Y_test.type(torch.long)

# Initialize variables
d = X_train.size()[1]
train_samples = len(Y_train)
k = 10
ha = 64
hb = 32
alpha = 1/np.sqrt(d)

# for a
W0a = torch.rand(d, ha)*2*alpha-alpha
W0a = W0a.requires_grad_()
B0a = torch.rand(1, ha)*2*alpha-alpha
B0a = B0a.requires_grad_()
W1a = torch.rand(ha, k)*2*alpha-alpha
W1a = W1a.requires_grad_()
B1a = torch.rand(1, k)*2*alpha-alpha
B1a = B1a.requires_grad_()
# for b
W0b = torch.rand(d, hb)*2*alpha-alpha
W0b = W0b.requires_grad_()
B0b = torch.rand(1, hb)*2*alpha-alpha
B0b = B0b.requires_grad_()
W1b = torch.rand(hb, hb)*2*alpha-alpha
W1b = W1b.requires_grad_()
B1b = torch.rand(1, hb)*2*alpha-alpha
B1b = B1b.requires_grad_()
W2b = torch.rand(hb, k)*2*alpha-alpha
W2b = W2b.requires_grad_()
```

```
B2b = torch.rand(1, k)*2*alpha-alpha
B2b = B2b.requires_grad_()


# a. Wide shallow network
class F1(torch.nn.Module):
    def __init__(self, w0, b0, w1, b1):
        super().__init__()
        self.w0 = torch.nn.Parameter(w0)
        self.b0 = torch.nn.Parameter(b0)
        self.w1 = torch.nn.Parameter(w1)
        self.b1 = torch.nn.Parameter(b1)

    def forward(self, x):
        temp = F.relu(torch.matmul(x, self.w0) + self.b0)
        return torch.matmul(temp, self.w1) + self.b1


# b. Narrow deep network
class F2(torch.nn.Module):
    def __init__(self, w0, b0, w1, b1, w2, b2):
        super().__init__()
        self.w0 = torch.nn.Parameter(w0)
        self.b0 = torch.nn.Parameter(b0)
        self.w1 = torch.nn.Parameter(w1)
        self.b1 = torch.nn.Parameter(b1)
        self.w2 = torch.nn.Parameter(w2)
        self.b2 = torch.nn.Parameter(b2)

    def forward(self, x):
        temp = F.relu(torch.matmul(x, self.w0) + self.b0)
        temp = F.relu(torch.matmul(temp, self.w1) + self.b1)
        return torch.matmul(temp, self.w2) + self.b2


# Set training parameters
lr = 1e-3
n_epochs = 2000
batch = 1000
loops = int(train_samples/batch)
#model = F1(W0a, B0a, W1a, B1a)  # choose between F1 and F2 here
model = F2(W0b, B0b, W1b, B1b, W2b, B2b)
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# Train
train_acc = []
losses = []
acc = 0
count = 0
while acc < 0.99:
    # for debugging
    print(count)
    count = count+1

    # mini-batch steps
```

```
    perm = torch.randperm(train_samples)
    for b in range(loops):
        ind = perm[b*batch:((b+1)*batch-1)]
        y_hat = model(X_train[ind, :])

        # cross entropy combines softmax calculation with NLLLoss
        loss = F.cross_entropy(y_hat, Y_train[ind])

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    # Calculate accuracy and loss per epoch
    y_hat = model(X_train)
    y = y_hat.argmax(axis=1)
    temp = sum(y == Y_train)
    acc = temp.item()/len(Y_train)
    train_acc.append(acc)
    loss = F.cross_entropy(y_hat, Y_train)
    losses.append(loss)
    print(acc)  # for debugging

# Plot accuracy and loss
plt.figure()
x = range(1, len(train_acc)+1)
plt.subplot(1, 2, 1)
plt.plot(x, [i * 100 for i in train_acc])
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.subplot(1, 2, 2)
plt.plot(x, losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

PCA

A.6

a. *[2 points]* Note: the images were all normalized by dividing by 255 for all parts of this problem.

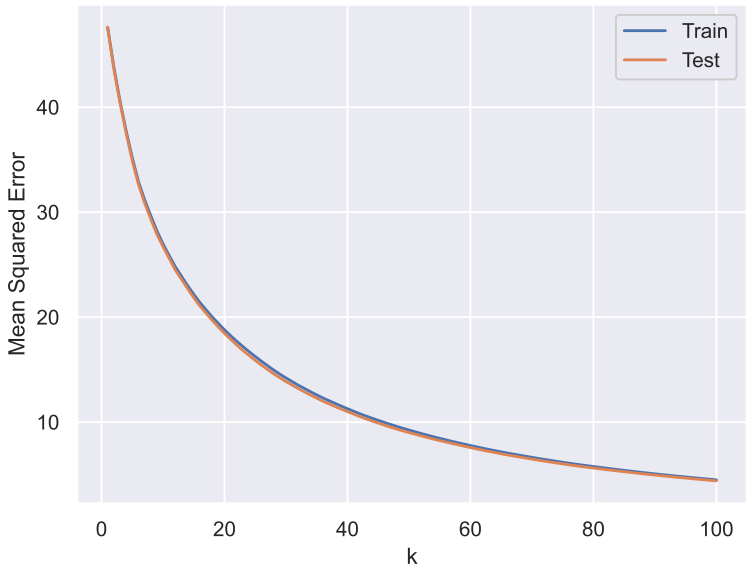
$$\lambda_1 = 5.1168$$
$$\lambda_2 = 3.7413$$
$$\lambda_{10} = 1.2427$$
$$\lambda_{30} = 0.3643$$
$$\lambda_{50} = 0.1697$$
$$\sum_{i=1}^{d} = 52.7250$$

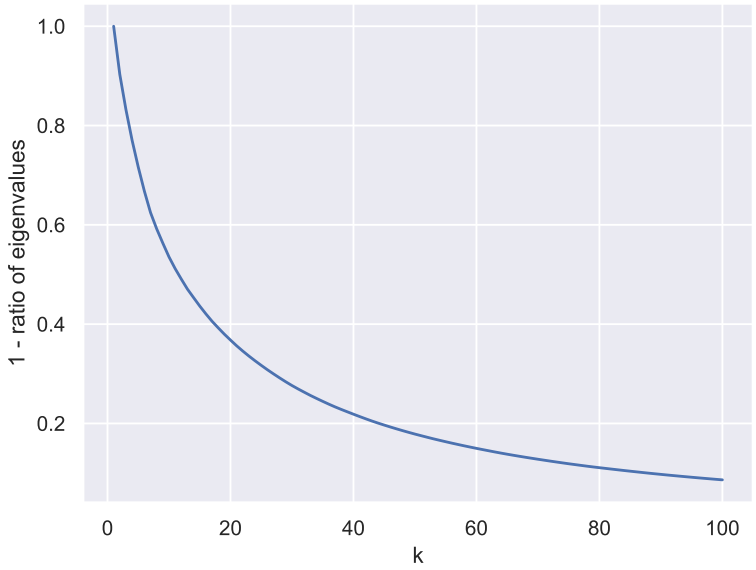
b. *[5 points]* $x \approx \mu + U_k D_k$ where U_k is the first k eigenvectors and D_k is a diagonal matrix with the first k eigenvalues. Since we know $D_k = U_K^\top (x - \mu)$ we have:

$$x \approx \mu + U_k U_k^\top (x - \mu)$$

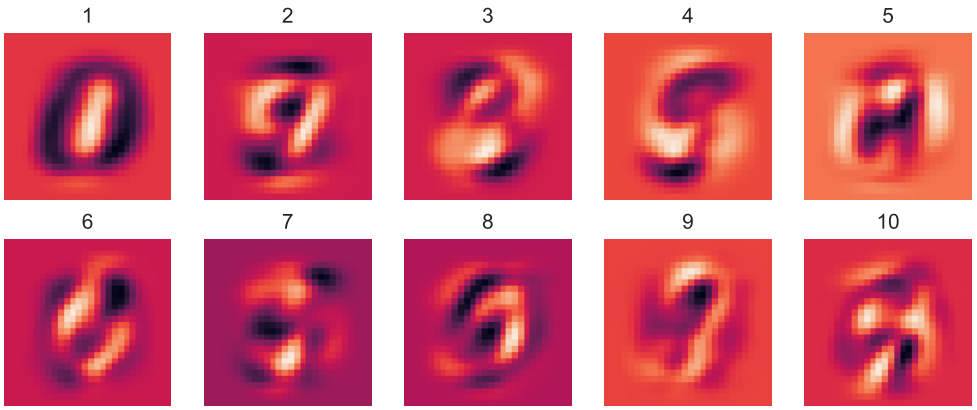
We are essentially transforming $x - \mu$ into the k dimensional eigenvector space, transforming it back, then correcting the mean subtraction by adding in μ.

c. *[5 points]*

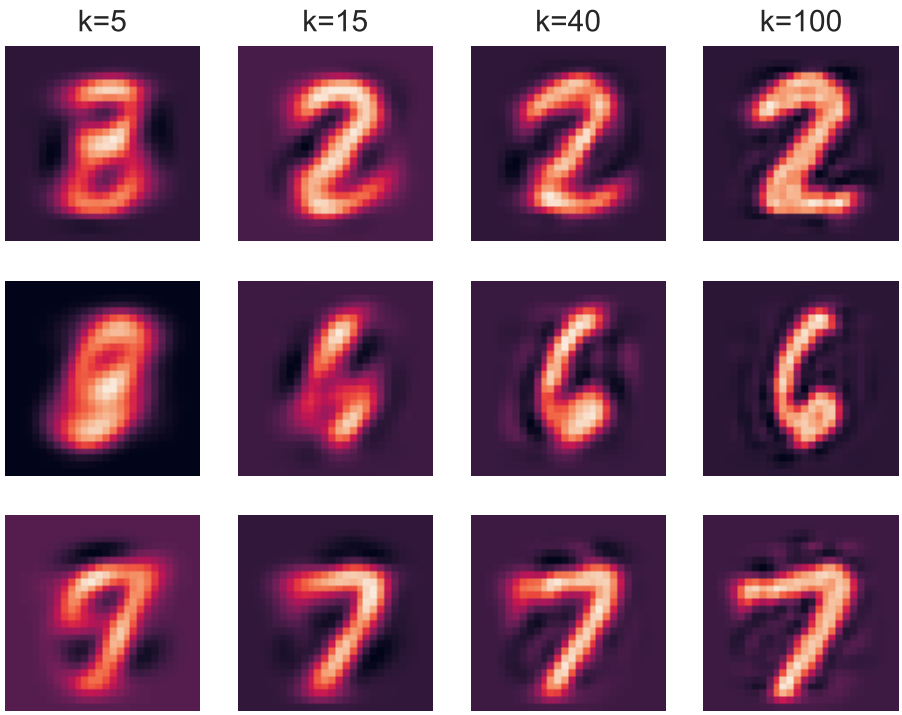




d. *[3 points]* The eigenvectors are shown below. Some of them look like numbers (the first eigenvector looks like a 0, the third an 8, etc.) but most of them seem to be focusing on edges and the differences in brightness, rather than specific lines, to capture the characteristics of each digit.



e. *[3 points]* The digits are already fairly distinguishable at $k = 15$. By $k = 40$ they are distinct and $k = 100$ simply makes them clearer. Thus we need very few dimensions (less than 10%) to accurately portray all digits.



Source Code A.6

```
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST
import seaborn as sns

sns.set()

# Load data
mndata = MNIST('../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

# Define mu and sigma
mu = np.mean(X_train, axis=0)
Sig = np.matmul(np.transpose(X_train-mu), X_train-mu)/60000

# Eigenvalue decomposition. Need to flip as the order is wrong
D, U = np.linalg.eigh(Sig)
```

```
D = np.flip(D)
U = np.fliplr(U)

def reconst(x, m, u, k):
    temp = np.matmul(u[:, 0:k], np.transpose(u[:, 0:k]))
    return np.transpose(np.matmul(temp, np.transpose(x-m))) + m

def mse(a, b):
    temp = np.square(a-b)
    return np.mean(np.sum(temp, axis=1))

# Get reconstruction errors
train_err = []
test_err = []
for k in range(1, 101):
    print(k)
    pred = reconst(X_train, mu, U, k)
    train_err.append(mse(X_train, pred))
    pred = reconst(X_test, mu, U, k)
    test_err.append(mse(X_test, pred))

# a
temp = [1, 2, 10, 30, 50]
for i in temp:
    print(D[i-1])
print(np.sum(D))

# c
x = np.arange(1, 101, 1)
plt.plot(x, train_err)
plt.plot(x, test_err)
plt.xlabel('k')
plt.ylabel('Mean Squared Error')
plt.legend(('Train', 'Test'))

plt.figure()
valsum = np.zeros(100)
for i in range(100):
    valsum[i] = 1-(np.sum(D[0:i])/np.sum(D))
plt.plot(x, valsum)
plt.xlabel('k')
plt.ylabel('1 - ratio of eigenvalues')

# d
plt.figure()
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(np.reshape(U[:, i], (28, 28)))
    plt.title(i+1)
    plt.axis('off')

# e
plt.figure()
digs = [2, 6, 7]
ks = [5, 15, 40, 100]
```

```
for d in range(len(digs)):
    ind = np.nonzero(labels_train == digs[d])
    ind = np.squeeze(np.asarray(ind))
    xtemp = X_train[np.random.choice(ind), :]
    for k in range(len(ks)):
        pred = reconst(xtemp, mu, U, ks[k])
        pred = np.reshape(pred, (28, 28))
        plt.subplot(len(digs), len(ks), d*len(ks)+k+1)
        plt.imshow(pred)
        plt.axis('off')
        if d == 0:
            plt.title('k='+str(ks[k]))
```