Name:

Karan Pandya

Netid:

karandp2

CS 441 - HW1: Instance-based Methods

Complete the sections below. You do not need to fill out the checklist.

Total Points Available **[ ] / 145**

1. Retrieval, K-means, 1-NN on MNIST
   a. Retrieval [ ] / 5
   b. K-means [ ] / 15
   c. 1-NN [ ] / 10
2. Make it fast
   a. K-means plot [ ] / 15
   b. 1-NN error plots [ ] / 8
   c. 1-NN time plots [ ] / 7
   d. Most confused label [ ] / 5
3. Temperature Regression
   a. RMSE Tables [ ] / 20
4. Conceptual questions [ ] / 15
5. Stretch Goals
   a. Evaluate effect of K for MNIST [ ] / 15
   b. Evaluate effect of K for Temp Reg. [ ] / 15
   c. Compare Kmeans more iterations vs. restarts [ ] / 15

1. Retrieval, K-means, 1-NN on MNIST

a. What index is returned for x_test[1]?

28882

b. Paste the display of clusters after the 1st and 10th iteration for K=30.

After 1st Iteration:

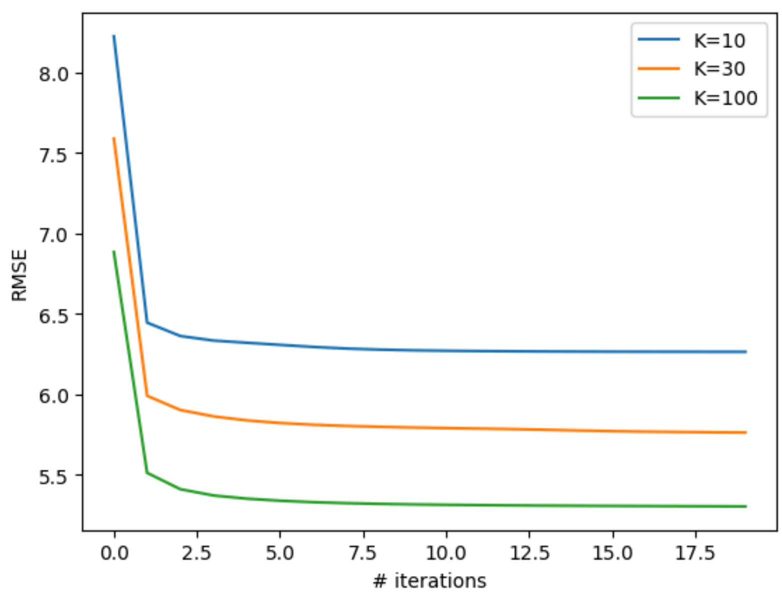
After 10th Iteration:

c. Error rate for first 100 test samples, using first 10,000 training samples (x.x)

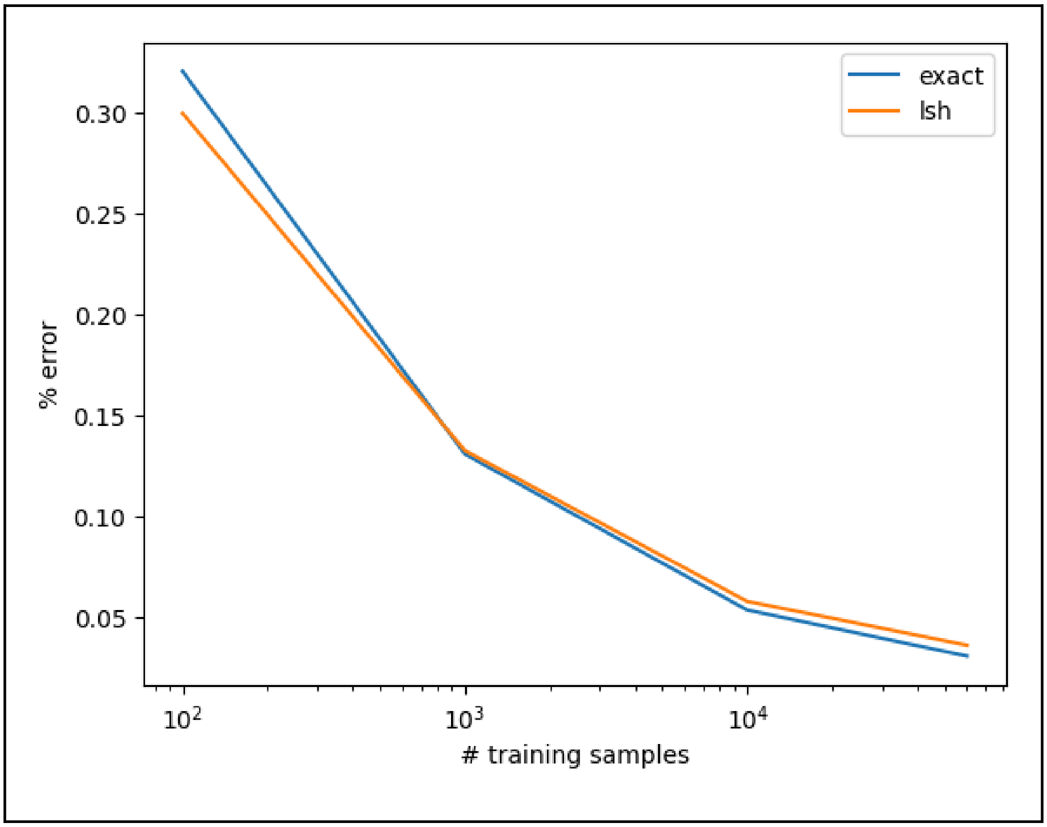
8.0%

2. Make it fast

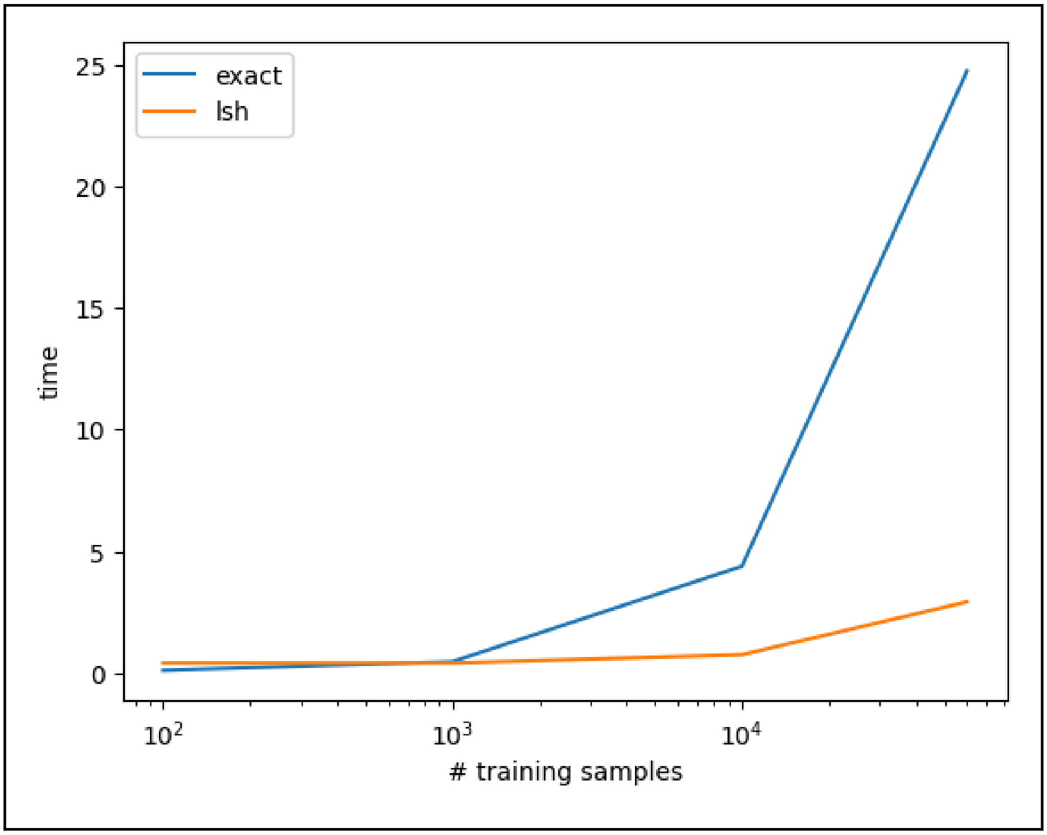
a. KMeans plot of RMSE vs iterations for K=10, 30, 100



b. Nearest neighbor error vs training size plot



c. Nearest neighbor time vs training size plot



d. What label is most commonly confused with ‘2’?

7

3. Temperature Regression

a. Table of RMSE for KNN with K=5   (x.xx)

	KNN (K=5)
Original Features	3.249
Normalized Features	2.932

4. Test your understanding

Fill in the letter corresponding to the answer. If you’re not sure, you can sometimes run small experiments to check.

1. Is K-means guaranteed to decrease RMSE between nearest cluster and samples at each iteration until convergence?

a. Yes
b. No

b

2. If you increase K, is K-means expected or guaranteed to achieve lower RMSE?
    a. Guaranteed
    b. Expected but not guaranteed
    c. Not expected

a

3. In K-NN regression, for training labels y, what is the lowest target value that can possibly be predicted for any query?
    a. Min(y)
    b. Mean(y)
    c. Can't be determined

a

4. Would you expect the "training error" for 1-NN to be higher or lower than 3-NN for classification? Training error is the error if you test on the training data.
    a. Higher
    b. Lower
    c. It's problem-dependent

b

5. Would you expect the test error for 1-NN to be higher or lower than for 3-NN for regression?
    a. Higher
    b. Lower
    c. It's problem-dependent

a

5. Stretch Goals (optional)

a. Select best K parameter for K-NN MNIST classification in K=1, 3, 5, 11, 25. (x.xx)

Validation Set Performance	K=1	K=3	K=5	K=11	K=25

% error	2.88	2.71	2.70	2.97	3.72

Best K:

5

Test % error (x.xx)

3.03

b. Select best K parameter for K-NN temperature regression in K=1, 3, 5, 11, 25. (x.xx)

Validation Set RMSE	K=1	K=3	K=5	K=11	K=25
Original Features	4.33	3.22	3.09	3.05	3.06
Normalized Features	3.866	3.174	3.03	2.89	2.91

Best Setting (K, feature type):

25

Test RMSE (x.xx)

2.76

c. K Means, MNIST: compare average and standard deviation RMSE based on number of iterations and number of restarts

(4 digit precision)

K=30	RMSE avg	RMSE std
20 iterations, 1 restart	61.88	0.5988
4 iterations, 5 restarts	61.47	0.16
50 iterations, 1 restart	61.71	0.42
10 iterations, 5 restarts	60.90	0.19

CS441_SP24_HW1_karandp2

February 5, 2024

0.1 CS441: Applied ML - HW 1

0.1.1 Parts 1-2: MNIST

Include all the code for generating MNIST results below

[ ]:
```
# initialization code
import numpy as np
from keras.datasets import mnist
%matplotlib inline
from matplotlib import pyplot as plt
from scipy import stats


def load_mnist():
  '''
  Loads, reshapes, and normalizes the data
  '''
  (x_train, y_train), (x_test, y_test) = mnist.load_data() # loads MNIST data
  x_train = np.reshape(x_train, (len(x_train), 28*28))  # reformat to 768-d␣
↪vectors
  x_test = np.reshape(x_test, (len(x_test), 28*28))
  maxval = x_train.max()
  x_train = x_train/maxval  # normalize values to range from 0 to 1
  x_test = x_test/maxval
  return (x_train, y_train), (x_test, y_test)

def display_mnist(x, subplot_rows=1, subplot_cols=1):
  '''
  Displays one or more examples in a row or a grid
  '''
  if subplot_rows>1 or subplot_cols>1:
    fig, ax = plt.subplots(subplot_rows, subplot_cols, figsize=(15,15))
    for i in np.arange(len(x)):
      ax[i].imshow(np.reshape(x[i], (28,28)), cmap='gray')
      ax[i].axis('off')
  else:
      plt.imshow(np.reshape(x, (28,28)), cmap='gray')
      plt.axis('off')
```

```
plt.show()
```

```
# example of using MNIST load and display  functions
(x_train, y_train), (x_test, y_test) = load_mnist()
display_mnist(x_train[:30],1,30)
print('Total size: train={}, test ={}'.format(len(x_train), len(x_test)))
```

1. Retrieval, Clustering, and NN Classification

```
from re import L
# Retrieval

def get_nearest(X_query, X):
  ''' Return the index of the sample in X that is closest to X_query according
      to L2 distance '''
  # TO DO
  min = 9999999
  pos = 0
  for i in range(len(X)) :
    L2 = (np.sum((X[i]-X_query)**2))**0.5

    if (L2<min):
      min=L2
      # print(min)
      pos = i
  return pos


j = get_nearest(x_test[0], x_train)
print(j)
j = get_nearest(x_test[1], x_train)
print(j)
```

```
X= x_train[:1000]
print(X[0])
```

```
#kmeans

clusters = []
def kmeans(X, K, niter=10):

    for i in range(K):
      clusters.append(X[i].tolist())

    for point in X[K:]:
      nearest = get_nearest(point, X[:K])
      print(nearest)
```

```
        clusters[nearest].append(point)

    for i in range(k):
        clusters[k] = np.mean(clusters[k])
    print("clusters shape", np.shape(clusters[0]))
    return clusters
    print("cm", clusters)

K= 30
centers = kmeans(x_train[:1000], K)
```

link text

```
[ ]: # K-means

def kmeans(X, K, niter):
    '''
    Starting with the first K samples in X as cluster centers, iteratively assign␣
  ↪each
    point to the nearest cluster and compute the mean of each cluster.
    Input: X[i] is the ith sample, K is the number of clusters, niter is the␣
  ↪number of iterations
    Output: K cluster centers
    '''
    Cluster_mean = X[:K].copy()
    rmse_values = []
    for n in range(niter):

      display_mnist(Cluster_mean,1,30)
      Cluster_centers = np.array([get_nearest(point, Cluster_mean) for point in␣
  ↪X])
      rmse = np.sqrt(np.mean(np.sum((X -␣
  ↪Cluster_mean[Cluster_centers])**2,axis=1)))
      for k in range(K):
        if len(X[Cluster_centers == k]) > 0:
            Cluster_mean[k] = np.mean(X[Cluster_centers == k], axis=0)

      display_mnist(Cluster_mean, 1, 30)
      print("iter", n)

      rmse_values.append(rmse)
      print(f"Iteration {n+1}/{niter}, RMSE: {rmse}")

    return Cluster_mean, rmse_values

K= 30
centers = kmeans(x_train[:1000], K, 20)
```

```
# K=10
# centers, rmse_values = kmeans(x_train[:1000], K, 20)
# plt.plot(np.arange(len(rmse_values)), rmse_values, label='K=10')

# K=30
# centers, rmse_values = kmeans(x_train[:1000], K, 20)
# plt.plot(np.arange(len(rmse_values)), rmse_values, label='K=30')

# K=100
# centers, rmse = kmeans(x_train[:1000], K, 20)
# plt.plot(np.arange(len(rmse)), rmse, label='K=100')
# plt.legend(), plt.ylabel('RMSE'), plt.xlabel('# iterations')
# plt.show()
```

[ ]:
```
# 1-NN
error_count = 0
for sample_idx in range(100):
  index = get_nearest(x_test[sample_idx],x_train[:10000])
  if(y_train[index] != y_test[sample_idx]):
    error_count += 1

error = error_count/100
print("error percentage is", error*100,"%")

# TO DO
```

2. Make it fast

[ ]:
```
# install libraries you need for part 2
!apt install libomp-dev
!pip install faiss-cpu
import faiss
import time
```

[ ]:
```
# retrieval
index = faiss.IndexFlatL2(x_train[:5].shape[1])  # set for exact search
centers = x_train[:5].copy()
index.add(centers) # add the data
dist, idx = index.search(x_test[10:30],1)
idx=idx.flatten()
X = x_test[10:30].copy()
print("idx is",idx)
for k in range(5):
    cluster_points = X[idx == k]
    print("cluster points for k=", k, " are", cluster_points)
    if len(cluster_points) > 0:
```

```python
        centers[k] = np.mean(cluster_points, axis=0)
        # print("mean of cluster points is ", centers[0])
# print(idx,"done")
# print("getnearest funct", get_nearest(x_test[12],x_train[:10]))
# TO DO (check that you're using FAISS correctly)
```

[ ]:
```python
# K-means

def kmeans_fast(X, K, niter=10):
  '''
  Starting with the first K samples in X as cluster centers, iteratively assign␣
 ↪each
  point to the nearest cluster using faiss and compute the mean of each cluster.
  Input: X[i] is the ith sample, K is the number of clusters, niter is the␣
 ↪number of iterations
  Output: K cluster centers
  '''
  rmse_values = []
  cluster_centers = X[50:K+50].copy()

  for n in range(niter):
   #   display_mnist(X[:30],1,30)
     index = faiss.IndexFlatL2(784)  # set for exact search
     index.add(cluster_centers)
     dist_curr, idx = index.search(X,1)
     print("idx shape", np.shape(idx))
     idx = idx.flatten()
     print("idx flat shape", np.shape(idx))

    #  rmse = np.sqrt(np.sum((X - cluster_centers[idx])**2)/len(X))
     rmse = np.sqrt(np.mean(np.sum((X - cluster_centers[idx])**2,axis=1)))

     for k in range(K):
        cluster_points = X[idx==k]
        cluster_centers[k] = np.mean(cluster_points, axis=0)
   #   display_mnist(cluster_centers[idx[0:30]],1,30)
     rmse_values.append(rmse)
     print(f"Iteration {n+1}/{niter}, RMSE: {rmse}")
  return cluster_centers, rmse_values

K=10
centers, rmse_values = kmeans_fast(x_train, K, niter=20)
plt.plot(np.arange(len(rmse_values)), rmse_values, label='K=10')

K=30
centers, rmse_values = kmeans_fast(x_train, K, niter=20)
plt.plot(np.arange(len(rmse_values)), rmse_values, label='K=30')
```

```
K=100
centers, rmse = kmeans_fast(x_train, K, niter=20)
plt.plot(np.arange(len(rmse)), rmse, label='K=100')
plt.legend(), plt.ylabel('RMSE'), plt.xlabel('# iterations')
plt.show()
```

[ ]:
```
a =np.array([[ 1,2,3], [4,5,7]])
sum = np.mean(np.sum(a, axis=1))
print(a)
print(np.sum(a, axis=1))
sum
```

[ ]:
```
# 1-NN
import time
nsample = [100, 1000, 10000, 60000]
# TO DO
# 1-NN
error_count = 0
error_lsh= []
timing_lsh = []
error_exact= []
timing_exact = []
for n in nsample:
  start_time = time.time()
  X = x_train
  dim = X.shape[1]
  index = faiss.IndexLSH(dim, dim)
  index.add(x_train[:n])
  dist, idx = index.search(x_test,1)

  for i in range(len(x_test)):
    if(y_train[idx[i]] != y_test[i]):
      error_count += 1

  error_lsh.append( error_count/len(x_test))
  print("error percentage is", error_count/len(x_test),"%")
  error_count = 0
  elapsed_time = time.time() - start_time
  timing_lsh.append(elapsed_time)

for n in nsample:
  start_time = time.time()
  X = x_train
  dim = X.shape[1]
  index = faiss.IndexFlatL2(x_train.shape[1])
  index.add(x_train[:n])
```

```
    dist, idx = index.search(x_test,1)

    for i in range(len(x_test)):
      if(y_train[idx[i]] != y_test[i]):
        error_count += 1

    error_exact.append(error_count/len(x_test))
    print("error percentage is", error_count/len(x_test),"%")
    error_count = 0
    elapsed_time = time.time() - start_time
    timing_exact.append(elapsed_time)

plt.semilogx(nsample, error_exact, label='exact')
plt.semilogx(nsample, error_lsh, label='lsh')
plt.legend(), plt.ylabel('% error'), plt.xlabel('# training samples')
plt.show()

plt.semilogx(nsample, timing_exact, label='exact')
plt.semilogx(nsample, timing_lsh, label='lsh')
plt.legend(), plt.ylabel('time'), plt.xlabel('# training samples')
plt.show()
```

[ ]:

```
# Confusion matrix
import sklearn
from sklearn.metrics import confusion_matrix

X = x_train.copy()
dim = X.shape[1]
index = faiss.IndexFlatL2(dim)
index.add(x_train)
dist, idx = index.search(x_test,1)
confusion_mat = confusion_matrix(y_test, y_train[idx])
print(confusion_mat)
print(np.sum(confusion_mat, axis=1))
print(np.shape(y_test))
print(len(y_test[y_test==2]))
```

0.2 Part 3: Temperature Regression

Include all your code used for part 2 in this section.

[ ]:

```
import numpy as np
%matplotlib inline
from matplotlib import pyplot as plt
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
```

```python
# load data (modify to match your data directory or comment)
def load_temp_data():
  datadir = "temperature_data.npz"
  T = np.load(datadir)
  x_train, y_train, x_val, y_val, x_test, y_test, dates_train, dates_val,␣
↪dates_test, feature_to_city, feature_to_day = \
  T['x_train'], T['y_train'], T['x_val'], T['y_val'], T['x_test'], T['y_test'],␣
↪T['dates_train'], T['dates_val'], T['dates_test'], T['feature_to_city'],␣
↪T['feature_to_day']
  return (x_train, y_train, x_val, y_val, x_test, y_test, dates_train,␣
↪dates_val, dates_test, feature_to_city, feature_to_day)

# plot one data point for listed cities and target date
def plot_temps(x, y, cities, feature_to_city, feature_to_day, target_date):
  nc = len(cities)
  ndays = 5
  xplot = np.array([-5,-4,-3,-2,-1])
  yplot = np.zeros((nc,ndays))
  for f in np.arange(len(x)):
    for c in np.arange(nc):
      if cities[c]==feature_to_city[f]:
        yplot[feature_to_day[f]+ndays,c] = x[f]
  plt.plot(xplot,yplot)
  plt.legend(cities)
  plt.plot(0, y, 'b*', markersize=10)
  plt.title('Predict Temp for Cleveland on ' + target_date)
  plt.xlabel('Day')
  plt.ylabel('Avg Temp (C)')
  plt.show()
```

[ ]:
```python
# load data
(x_train, y_train, x_val, y_val, x_test, y_test, dates_train, dates_val,␣
↪dates_test, feature_to_city, feature_to_day) = load_temp_data()
''' Data format:
      x_train, y_train: features and target value for each training sample␣
↪(used to fit model)
      x_val, y_val: features and target value for each validation sample (used␣
↪to select hyperparameters, such as regularization and K)
      x_test, y_test: features and target value for each test sample (used to␣
↪evaluate final performance)
      dates_xxx: date of the target value for the corresponding sample
      feature_to_city: maps from a feature number to the city
      feature_to_day: maps from a feature number to a day relative to the␣
↪target value, e.g. -2 means two days before
      Note: 361 is the temperature of Cleveland on the previous day
'''
```

```
f = 361
print('Feature {}: city = {}, day= {}'.format(f,feature_to_city[f],␣
 ↪feature_to_day[f]))
baseline_rmse = np.sqrt(np.mean((y_val[1:]-y_val[:-1])**2)) # root mean squared␣
 ↪error example
print('Baseline - prediction using previous day: RMSE={}'.format(baseline_rmse))

# plot first two x/y for val
plot_temps(x_val[0], y_val[0], ['Cleveland', 'New York', 'Chicago', 'Denver',␣
 ↪'St. Louis'], feature_to_city, feature_to_day, dates_val[0])
plot_temps(x_val[1], y_val[1], ['Cleveland', 'New York', 'Chicago', 'Denver',␣
 ↪'St. Louis'], feature_to_city, feature_to_day, dates_val[1])
```

[ ]:
```
# install libraries you need for part 2
import faiss
import time

# K-NN Regression

def regress_KNN(X_trn, y_trn, X_tst, y_test, K):
  '''
  Predict the target value for each data point in X_tst using a
  K-nearest neighbor regressor based on (X_trn, y_trn), with L2 distance.
  Input: X_trn[i] is the ith training data. y_trn[i] is the ith training label.␣
 ↪K is the number of closest neighbors to use.
  Output: return y_pred, where y_pred[i] is the predicted ith test value
  '''
  # TO DO
  Y_pred = np.zeros(X_tst.shape[0])
  index = faiss.IndexFlatL2(X_trn.shape[1])  # set for exact search
  index.add(X_trn)
  for i in range(X_tst.shape[0]):
    query_vector = X_tst[i].reshape(1, -1)
    dist, idx = index.search(query_vector, K)
    Y_pred[i] = np.mean(y_trn[idx])

  # rmse = np.sqrt(np.mean((y_test - Y_pred)**2))
  rmse = np.sqrt(np.mean((y_test - Y_pred)**2))

  return rmse


def normalize_features(x, y, fnum):
  ''' Normalize the features in x and y.
      For each data sample i:
        x2[i] = x[i]-x[i,fnum]
        y2[i] = y[i]-x[i,fnum]
```

```
  '''

  # TO DO
  x_normalized = np.copy(x)
  y_normalized = np.copy(y)

  # Normalize features for each data sample
  for i in range(x.shape[0]):
    x_normalized[i, :] = x[i, :] - x[i, fnum]
    y_normalized[i] = y[i] - x[i, fnum]

  return x_normalized, y_normalized

# KNN with original features

# TO DO
# print(np.shape(x_train))
K = 3
print(x_train.shape, y_train.shape)

rmse =regress_KNN(x_train, y_train, x_test, y_test, K)
print("rmse", rmse)

# KNN with normalized features
fnum = 361 # previous day temp in Cleveland

# KNN with normalized features
x_norm, y_norm = normalize_features(x_train, y_train, fnum)
xtest_norm, ytest_norm = normalize_features(x_test, y_test, fnum)

rmse =regress_KNN(x_norm, y_norm, xtest_norm , ytest_norm, K)
print("rmse", rmse)
```

0.3 Part 5: Stretch Goals

Include all your code used for part 5 in this section. You can copy-paste code from parts 1-3 if it is re-usable.

[ ]:
```
# Stretch: KNN classification (Select K)
# install libraries you need for part 2
import faiss
import time

# K-NN Regression
def regress_KNN(X_trn, y_trn, X_tst, Y_tst, K):
  '''
  Predict the target value for each data point in X_tst using a
```

```
    K-nearest neighbor regressor based on (X_trn, y_trn), with L2 distance.
    Input: X_trn[i] is the ith training data. y_trn[i] is the ith training label.␣
↪K is the number of closest neighbors to use.
    Output: return y_pred, where y_pred[i] is the predicted ith test value
    '''
    # TO DO
    dimrow, dimcol = X_tst.shape
    Y_pred = np.zeros(dimrow)
    index = faiss.IndexFlatL2(X_trn.shape[1])  # set for exact search
    index.add(X_trn)
    error_count = 0
    error = 0
    for i in range(dimrow):
      query_vector = X_tst[i].reshape(1, -1)
      dist, idx = index.search(query_vector, K)
      # print(y_trn[idx].flatten())
      Y_pred[i] = most_common_and_closest(y_trn[idx].flatten(), Y_tst[i], X_tst,␣
↪X_trn, i, idx.flatten())
      # print(Y_pred[i])
      if Y_pred[i] != Y_tst[i]:
         error_count += 1
    error = error_count/dimrow
    return error

def most_common_and_closest(sequence, sample, X_tst, X_trn,i, idx):
    # if not sequence:
    #     return None  # Handle the case when the sequence is empty

    unique_elements, counts = np.unique(sequence, return_counts=True)

    # Find indices of maximum count(s)
    max_count_indices = np.where(counts == np.max(counts))[0]
    # print(np.where(counts == np.max(counts))[0])
    # If there's only one element with the maximum count, return it
    if len(max_count_indices) == 1:
        return unique_elements[max_count_indices[0]]
    else:
    # If there's a tie, find the index of the element closest to the sample
      closest_index = np.argmin(np.abs(unique_elements[max_count_indices] -␣
↪sample))
      print("tie casee", "sequence",sequence, "unique_elements",␣
↪unique_elements,"counts", counts, "mci", max_count_indices,"closesst index",␣
↪closest_index,"Sample", sample)
      display_mnist(X_tst[i],1,1)
      display_mnist(X_trn[idx],1,11)
```

```python
    return unique_elements[max_count_indices[closest_index]]

def normalize_features(x, y, fnum):
  ''' Normalize the features in x and y.
      For each data sample i:
        x2[i] = x[i]-x[i,fnum]
        y2[i] = y[i]-x[i,fnum]
  '''

  # TO DO
  x_normalized = np.copy(x)
  y_normalized = np.copy(y)

  # Normalize features for each data sample
  for i in range(x.shape[0]):
    x_normalized[i, :] = x[i, :] - x[i, fnum]
    y_normalized[i] = y[i] - x[i, fnum]

  return x_normalized, y_normalized

# KNN with original features

# TO DO
# print(np.shape(x_train))
K = [1,3,5,11,25]
x_t = x_train[:50000]
y_t = y_train[:50000]
x_v = x_train[50000:]
y_v = y_train[50000:]

print(x_t.shape, y_t.shape, x_v.shape, y_v.shape)
rmse =regress_KNN(x_t, y_t, x_v, y_v, 11)
print("Error for K", i, "is ", rmse)

# # KNN with normalized features
# fnum = 361 # previous day temp in Cleveland

# # KNN with normalized features
# x_norm, y_norm = normalize_features(x_train, y_train, fnum)
# xtest_norm, ytest_norm = normalize_features(x_test, y_test, fnum)

# rmse =regress_KNN(x_norm, y_norm, xtest_norm , ytest_norm, K)
# print("rmse", rmse)
```

[ ]:
```python
# Stretch: KNN regression (Select K)
display_mnist(x_v[:20], 1,20)
```

[ ]:
```python
# Stretch: KNN classification (Select K)
# install libraries you need for part 2
import faiss
import time

# K-NN Regression
def regress_KNN(X_trn, y_trn, X_tst, Y_tst, K):
  '''
  Predict the target value for each data point in X_tst using a
  K-nearest neighbor regressor based on (X_trn, y_trn), with L2 distance.
  Input: X_trn[i] is the ith training data. y_trn[i] is the ith training label.␣
↪K is the number of closest neighbors to use.
  Output: return y_pred, where y_pred[i] is the predicted ith test value
  '''
  # TO DO
  dimrow, dimcol = X_tst.shape
  Y_pred = np.zeros(dimrow)
  index = faiss.IndexFlatL2(X_trn.shape[1])  # set for exact search
  index.add(X_trn)
  error_count = 0
  error = 0
  for i in range(dimrow):
    query_vector = X_tst[i].reshape(1, -1)
    dist, idx = index.search(query_vector, K)
    # print(y_trn[idx].flatten())
    Y_pred[i] = most_common_and_closest(y_trn[idx].flatten(), Y_tst[i], X_tst,␣
↪X_trn, i, idx.flatten(), dist.flatten())

    # print(Y_pred[i])
    if Y_pred[i] != Y_tst[i]:
       error_count += 1
  error = error_count/dimrow
  return error

def most_common_and_closest(sequence, sample, X_tst, X_trn,i, idx, dist):
    # if not sequence:
    #     return None  # Handle the case when the sequence is empty

    unique_elements, counts = np.unique(sequence, return_counts=True)

    # Find indices of maximum count(s)
    max_count_indices = np.where(counts == np.max(counts))[0]
    # print(np.where(counts == np.max(counts))[0])
    # If there's only one element with the maximum count, return it
    if len(max_count_indices) == 1:
        return unique_elements[max_count_indices[0]]
    else:
```

```python
      min=99999999999
      for i in range(len(sequence)):
         if np.any(np.isin( unique_elements[max_count_indices], sequence)):
            if dist[i]<min:
              min=dist[i]
              pos = i

    # If there's a tie, find the index of the element closest to the sample

      # print("tie casee", "sequence",sequence, "unique_elements",␣
↪unique_elements,"counts", counts, "mci", max_count_indices,"closest",␣
↪sequence[pos],"Sample", sample)
      # display_mnist(X_tst[i],1,1)
      # display_mnist(X_trn[idx],1,5)


    return sequence[pos]

def normalize_features(x, y, fnum):
  ''' Normalize the features in x and y.
      For each data sample i:
        x2[i] = x[i]-x[i,fnum]
        y2[i] = y[i]-x[i,fnum]
  '''

  # TO DO
  x_normalized = np.copy(x)
  y_normalized = np.copy(y)

  # Normalize features for each data sample
  for i in range(x.shape[0]):
    x_normalized[i, :] = x[i, :] - x[i, fnum]
    y_normalized[i] = y[i] - x[i, fnum]

  return x_normalized, y_normalized

# KNN with original features

# TO DO
# print(np.shape(x_train))
K = [1,3,5,11,25]
x_t = x_train[:50000]
y_t = y_train[:50000]
x_v = x_train[50000:]
y_v = y_train[50000:]
for i in K:
  print(x_t.shape, y_t.shape, x_v.shape, y_v.shape)
```

```
  rmse =regress_KNN(x_t, y_t, x_v, y_v, i)
  print("Error for K", i, "is ", rmse)

# # KNN with normalized features
# fnum = 361 # previous day temp in Cleveland

# # KNN with normalized features
# x_norm, y_norm = normalize_features(x_train, y_train, fnum)
# xtest_norm, ytest_norm = normalize_features(x_test, y_test, fnum)

# rmse =regress_KNN(x_norm, y_norm, xtest_norm , ytest_norm, K)
# print("rmse", rmse)
```

[ ]:
```
#temp regression
# install libraries you need for part 2
import faiss
import time

# K-NN Regression

def regress_KNN(X_trn, y_trn, X_tst, y_test, K):
  '''
  Predict the target value for each data point in X_tst using a
  K-nearest neighbor regressor based on (X_trn, y_trn), with L2 distance.
  Input: X_trn[i] is the ith training data. y_trn[i] is the ith training label.
 ↪K is the number of closest neighbors to use.
  Output: return y_pred, where y_pred[i] is the predicted ith test value
  '''
  # TO DO
  Y_pred = np.zeros(X_tst.shape[0])
  index = faiss.IndexFlatL2(X_trn.shape[1])  # set for exact search
  index.add(X_trn)
  for i in range(X_tst.shape[0]):
    query_vector = X_tst[i].reshape(1, -1)
    dist, idx = index.search(query_vector, K)
    Y_pred[i] = np.mean(y_trn[idx])

  # rmse = np.sqrt(np.mean((y_test - Y_pred)**2))
  rmse = np.sqrt(np.mean((y_test - Y_pred)**2))

  return rmse


def normalize_features(x, y, fnum):
  ''' Normalize the features in x and y.
      For each data sample i:
        x2[i] = x[i]-x[i,fnum]
```

```
        y2[i] = y[i]-x[i,fnum]
    '''

    # TO DO
    x_normalized = np.copy(x)
    y_normalized = np.copy(y)

    # Normalize features for each data sample
    for i in range(x.shape[0]):
      x_normalized[i, :] = x[i, :] - x[i, fnum]
      y_normalized[i] = y[i] - x[i, fnum]

    return x_normalized, y_normalized

# KNN with original features

# TO DO
# print(np.shape(x_train))
K = [1,3,5,11,25]
k=25
# for k in K:
print(x_train.shape, y_train.shape)
rmse =regress_KNN(x_train, y_train, x_test, y_test, k)
print("rmse", "unnormalized", rmse, "k", k)

# KNN with normalized features
fnum = 361 # previous day temp in Cleveland

# KNN with normalized features
x_norm, y_norm = normalize_features(x_train, y_train, fnum)
xtest_norm, ytest_norm = normalize_features(x_test, y_test, fnum)

rmse =regress_KNN(x_norm, y_norm, xtest_norm , ytest_norm, k)
print("rmse","normalized", rmse, "k", k)
```

[ ]:
```
# Compare (niter=10, nredo=5) vs. (niter=50, nredo=1) for K=30.
#Repeat this test five times and report the mean and standard deviation of the
 ↪RMSE.
import time
t = int(time.time())
rmse_vals = []
for i in range(5):
    kmeans = faiss.Kmeans(x_train.shape[1], 30, niter=20, nredo=1,
 ↪seed=int(i*10000))
    kmeans.train(x_train)
    dist, idx = kmeans.index.search(x_train, 1)
    rmse = np.sqrt(np.sum(dist) / x_train.shape[0])
```

```
    rmse_vals.append(rmse)
mean = np.mean(rmse_vals)
std = np.std(rmse_vals)
print("mean", mean, "std", std)


    # kmeans = faiss.Kmeans(x_train.shape[1], 30, niter=50, nredo=1,␣
 ↪seed=int(t))
    # kmeans.train(x_train)
    # dist, idx = kmeans.index.search(x_train, 1)
    # rmse = np.sqrt(np.sum(dist) / x_train.shape[0])
    # print("rmse", rmse, "i", i)
# Compare (niter=4, nredo=5) vs. (niter=20, nredo=1) for K=30.
#Repeat this test five times and report the mean and standard deviation of the␣
 ↪RMSE.
```