Problem Set 3

Before You Start

Make sure the following libraries load correctly (hit Ctrl-Enter). Note that while you are loading several powerful libraries, including machine learning libraries, the goal of this problem set is to implement several algorithms from scratch. In particular, you should *not* be using any built-in libraries for nearest neighbors, distance metrics, or cross-validation -- your mission is to write those algorithms in Python! Part 1 will be relatively easy; Part 2 will take more time.

In [1]:
```
import IPython
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib
import sklearn
import math
import time
```

In [2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

Introduction to the assignment

For this assignment, you will be using the Boston Housing Prices Data Set. Please read about the dataset carefully before continuing. Use the following commands to load the dataset:

NOTE - This dataset is similar to the one you used in PS1; we are just using a different method to load it this time. The column names and their order will remain the same for this dataset as was in PS1.

In [3]:
```
# load Boston housing data set
data = np.loadtxt('data.txt')
target = np.loadtxt('target.txt')
```

In [4]:
```
print(data.shape)
print(target.shape)
```

```
(506, 13)
(506,)
```

In [5]:
```
# adding column names and turning data into a dataframe
col = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', '
 'B', 'LSTAT']
```

```
df = pd.DataFrame(data, columns = col)
df['MEDV'] = target
```

Part 1: Experimental Setup

The goal of the next few sections is to design an experiment to predict the median home value for an instance in the data. Before beginning the "real" work, refamiliarize yourself with the dataset.

In [6]:

```
# investigating dataframe
print(df.head())
print(df.isnull().sum())
df.dtypes
```

```
       CRIM    ZN     INDUS  CHAS       NOX        RM   AGE       DIS  RAD  \
0  0.218960  18.0  2.629288   0.0  0.869420  6.875396  65.2  4.347275  1.0
1  0.141576   0.0  7.315612   0.0  0.549711  6.499894  78.9  5.315684  2.0
2  0.380457   0.0  7.340354   0.0  0.697928  7.263489  61.1  5.356935  2.0
3  0.313563   0.0  2.562407   0.0  0.599629  7.209732  45.8  6.103983  3.0
4  0.330105   0.0  2.497337   0.0  0.476077  7.184111  54.2  6.264372  3.0

     TAX    PTRATIO           B     LSTAT  MEDV
0  307.0  15.534711  397.462329  5.715647  24.0
1  255.0  17.914131  397.012611  9.338417  21.6
2  243.0  17.919989  396.628236  4.142473  34.7
3  226.0  18.979527  398.564784  3.239272  33.4
4  234.0  18.708888  399.487766  6.115159  36.2
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
MEDV       0
dtype: int64
```

Out[6]:

```
CRIM       float64
ZN         float64
INDUS      float64
CHAS       float64
NOX        float64
RM         float64
AGE        float64
DIS        float64
RAD        float64
TAX        float64
PTRATIO    float64
B          float64
LSTAT      float64
MEDV       float64
dtype: object
```

1.1 Begin by writing a function to compute the Root Mean Squared Error for a list of numbers

You can find the sqrt function in the Numpy package. Furthermore the details of RMSE can be found on Wikipedia. Do not use a built-in function to compute RMSE, other than numpy functions like `sqrt` and if needed, `sum` or other relevant ones.

In [7]:

```
"""
Function
--------
compute_rmse

Given two arrays, one of actual values and one of predicted values,
compute the Roote Mean Squared Error

Parameters
----------
predictions : array
    Array of numerical values corresponding to predictions for each of the N obs

yvalues : array
    Array of numerical values corresponding to the actual values for each of the

Returns
-------
rmse : int
    Root Mean Squared Error of the prediction

Example
-------
>>> print(compute_rmse((4,6,3),(2,1,4)))
3.16
"""

def compute_rmse(predictions, yvalues):
    # taking the difference between the 2 arrays
    diffs = (np.array(yvalues)-np.array(predictions))
    # squaring the differences
    squares = np.square(diffs)
    # summing the squares
    s = np.sum(squares)
    # dividing by the length
    inside = s/len(diffs)
    # taking the square root
    rmse = math.sqrt(inside)
    return rmse
```

In [8]:

```
# testing the compute_rmse function
print(compute_rmse((4,6,3),(2,1,4)))
```

```
3.1622776601683795
```

1.2 Divide your data into training and testing datasets

Randomly select 75% of the data and put this in a training dataset (call this "bdata_train"), and place the remaining 25% in a testing dataset (call this "bdata_test"). Do not use built-in functions.

To perform any randomized operation, only use functions in the *numpy library (np.random)*. Do not use other packages for random functions.

In [9]:
```
# leave the following line untouched, it will help ensure that your "random" spl
np.random.seed(seed=13579)

# splits data into training and testing sets
train_percent = .75
train_number = int(train_percent*len(data))
print('Total examples: %i' % len(data))
print('Number of training examples: %i' % train_number)
print('Number of testing examples: %i' % (len(data) - train_number))

ids = np.arange(0, len(df), 1)
ids = np.random.permutation(ids)
df_shuffled = df.iloc[ids]
bdata_train = df_shuffled[:train_number]
bdata_test = df_shuffled[train_number:]
```

```
Total examples: 506
Number of training examples: 379
Number of testing examples: 127
```

1.3 Use a very bad baseline for prediction, and compute RMSE

Let's start by creating a very bad baseline model that predicts median home values as the averages of `MEDV` based on adjacency to Charles River.

Specifically, create a model that predicts, for every observation X_i, the median home value as the average of the median home values of all houses in the **training set** that have the same adjacency value as the observation.

For example - For an input observation where `CHAS==1`, the model should predict the `MEDV` as the mean of all `MEDV` values in the training set that also have `CHAS==1`.

Once the model is built, do the following:

1. Compute the RMSE of the training set.
2. Now compute the RMSE of the test data set (but use the model you trained on the training set!).
3. How does RMSE compare for training vs. testing datasets? Is this what you expected, and why?
4. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. Color the training instances in red and the test instances in blue. Make sure to label your axes appropriately, and add a legend to your figure to make clear which dots are which.

5. Add code to your function to measure the running time of your algorithm. How long does it take to compute the predicted values for the test data?

NOTE - Be careful while dealing with floats and integers. Additionally, the `groupby` operation might come handy here.

In [10]:

```
def model(dataset):
    start_time = time.time() # 1.3.5
    # sets prediction values
    pred = bdata_train['MEDV'].groupby(bdata_train['CHAS']).mean()

    # assigns predictions for each value of the dataset
    predicted = []
    for i in dataset['CHAS']:
        if i == 0:
            predicted.append(pred[0])
        else:
            predicted.append(pred[1])
    t1 = time.time() - start_time # 1.3.5
    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return predicted
```

1.3.1 Compute the RMSE of the training set.

In [11]:

```
pred_train = model(bdata_train)
print(compute_rmse(pred_train, bdata_train['MEDV']))
```

```
Time taken: 0.00 seconds
8.963434181280334
```

1.3.2 Now compute the RMSE of the test data set (but use the model you trained on the training set!).

In [12]:

```
pred_test = model(bdata_test)
print(compute_rmse(pred_test, bdata_test['MEDV']))
```

```
Time taken: 0.00 seconds
9.292691153610612
```

1.3.3 How does RMSE compare for training vs. testing datasets? Is this what you expected, and why?

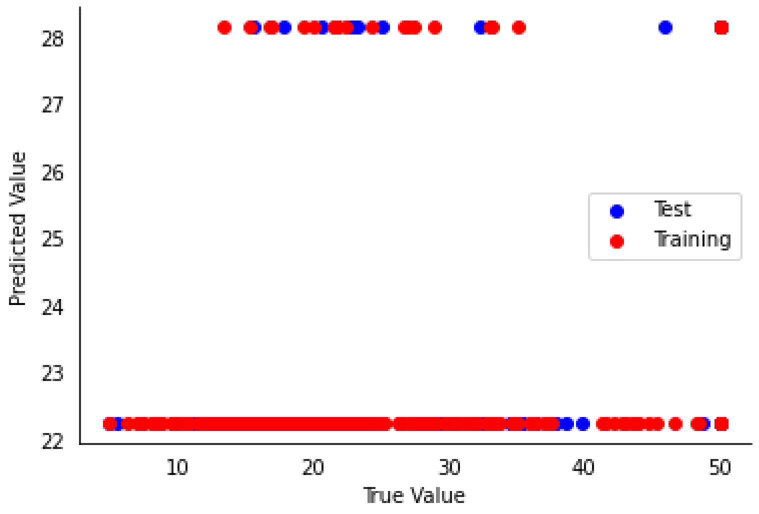
The RMSE for the training set is 8.96, while the RMSE for the testing set is 9.29. It makes sense that the RMSE would be worse for the testing set since the RMSE for the training set was trained with the same data, while the testing set is new to the model.

1.3.4 Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. Color the training instances in red and the test instances in blue. Make sure to label your axes appropriately, and add a legend to your figure to make clear which dots are which.

In [13]:

```
fig, ax = plt.subplots(1)
ax.scatter(bdata_test['MEDV'].values, pred_test, c = 'blue', label= 'Test')
ax.scatter(bdata_train['MEDV'].values, pred_train, c = 'red', label = 'Training'
plt.xlabel('True Value')
plt.ylabel('Predicted Value')

ax = plt.gca()
ax.legend(loc="best")
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.yaxis.set_ticks_position('none')
ax.xaxis.set_ticks_position('none')
plt.show()
```



1.3.5 Add code to your function to measure the running time of your algorithm. How long does it take to compute the predicted values for the test data?

Time taken: 0.01 seconds to compute the predicted values for the test data.

Part 2: Nearest Neighbors

2.1 Nearest Neighbors: Distance function

Let's try and build a machine learning algorithm to beat the "Average Value" baseline that you computed above. Soon you will implement the Nearest Neighbor algorithm, but first you need to create a distance metric to measure the distance (and similarity) between two instances. Write a generic function to compute the L-Norm distance (called the *p*-norm distance on Wikipedia). Verify that your function works by computing the Euclidean distance between the points (2,7) and (5,11), and then compute the Manhattan distance between (4,4) and (12,10).

In [14]:
```
"""
Function
--------
distance

Given two instances and a value for L, return the L-Norm distance between them

Parameters
```

```
----------
x1, x2 : array
    Array of numerical values corresponding to predictions for each of the N obs

L: int
    Value of L to use in computing distances

Returns
-------
dist : int
    The L-norm distance between instances

Example
-------
>>> print(distance((2,7),(5,11),2))
5

"""
# distance function

def distance(x1, x2, L):
    x1 = np.array(x1)
    x2 = np.array(x2)
    dist = (np.sum((np.absolute((np.array(x1) - np.array(x2))))**L, axis =-1))**
    return dist
```

In [15]:
```
# testing Euclidean distance
print(distance((2,7),(5,11),2))
```

```
5.0
```

In [16]:
```
# testing Manhattan distance
print(distance((4,4),(12,10),1))
```

```
14.0
```

2.2 Basic Nearest Neighbor algorithm

Your next task is to implement a basic nearest neighbor algorithm from scratch. Your simple model will use three input features (`CRIM, RM and ZN`) and a single output (`MEDV`). In other words, you are modelling the relationship between median home value and crime rates, house size and the proportion of residential land zoned for lots.

Use your training data (bdata_train) to "fit" your model, although as you know, with Nearest Neighbors there is no real training, you just need to keep your training data in memory. Write a function that predicts the median home value using the nearest neighbor algorithm we discussed in class. Since this is a small dataset, you can simply compare your test instance to every instance in the training set, and return the `MEDV` value of the closest training instance. Have your function take L as an input, where L is passed to the distance function. Use L=2 for all questions henceforth unless explicitly stated otherwise.

Make sure to do the following -

1. Fill in the function specification below
2. Use your algorithm to predict the median home value of every instance in the test set. Report the RMSE ("test RMSE")
3. Use your algorithm to predict the median home value of every instance in the training set and report the training RMSE.
4. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. Color the training instances in red and the test instances in blue.
5. Report an estimate of the total time taken by your code to predict the nearest neighbors for all the values in the test data set.
6. How does the performance (test RMSE and total runtime) of your nearest neighbors algorithm compare to the baseline in part 1.3?

2.2.1 Fill in the function specification below

In [17]:
```python
"""
Function
--------
Nearest Neighbors

Implementation of nearest neighbors algorithm.

Parameters
----------
x_train: array
    Array of numerical feature values for training the model.
y_train: array
    Array of numerical output values for training the model.
x_test: array
    Array of numerical feature values for testing the model.
y_test: array
    Array of numerical output values for testing the model.
L: int
    Order of L-norm function used for calculating distance.

Returns
-------
rmse : int
    Value of the RMSE from data.
"""

# function predicts the median home value using the nearest neighbor algorithm
def nneighbor(x_train, y_train, x_test, y_test, L):
    start_time = time.time()

    pred = []

    for i in range(len(x_test)):
        # finds the distance
        dists = distance(np.array(x_test[i]), np.array(x_train), L)
        # finds the smallest distance
        closest_point = np.argmin(dists)
        # appends closest point to the predictions array
        pred.append(np.array(y_train)[closest_point])
    # computes RMSE of the predictions and the test values
    rmse = compute_rmse(pred, y_test)
    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return rmse, pred
```

2.2.2 Use your algorithm to predict the median home value of every instance in the test set. Report the RMSE ("test RMSE")

In [18]:

```
# specifying the data to pass into the nneighbor function
x_train = bdata_train[['CRIM','RM','ZN']].values
y_train = bdata_train[['MEDV']]
x_test = bdata_test[['CRIM','RM','ZN']].values
y_test = bdata_test[['MEDV']]

# using algorithm to predict median home value of every instance in the test set
rmse_test, test_pred = nneighbor(x_train,y_train,x_test,y_test , 2)
print(rmse_test)
```

```
Time taken: 0.00 seconds
7.11504450215995
```

2.2.3 Use your algorithm to predict the median home value of every instance in the training set and report the training RMSE.

In [19]:

```
rmse_train, train_pred = nneighbor(x_train,y_train,x_train,y_train , 2)
print(rmse_train)
```

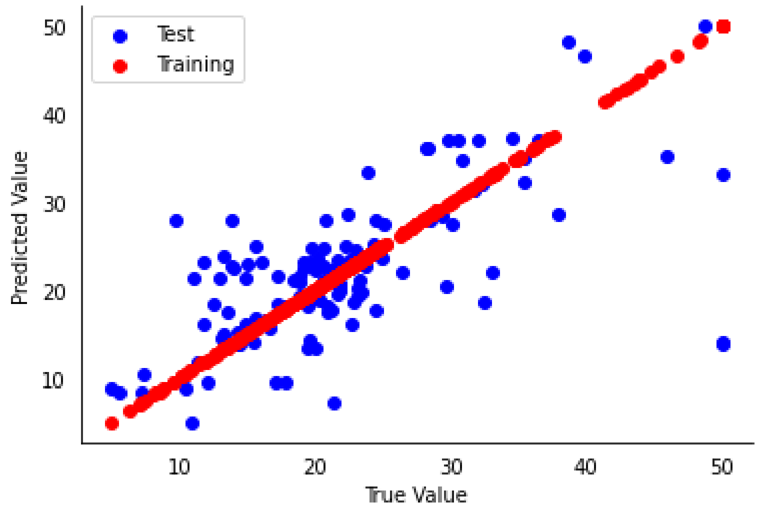
```
Time taken: 0.01 seconds
0.0
```

2.2.4 Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. Color the training instances in red and the test instances in blue.

In [20]:

```
# creating the scatter plot
fig, ax = plt.subplots(1)
ax.scatter(bdata_test[['MEDV']], test_pred, c = 'blue', label = 'Test')
ax.scatter(bdata_train[['MEDV']], train_pred, c = 'red', label = 'Training')
plt.xlabel('True Value')
plt.ylabel('Predicted Value')

ax = plt.gca()
ax.legend(loc="best")
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.yaxis.set_ticks_position('none')
ax.xaxis.set_ticks_position('none')
plt.show()
```



2.2.5 Report an estimate of the total time taken by your code to predict the nearest neighbors for all the values in the test data set.

The time taken to predict the nearest neighbors for all the values in the test data set was calculated in 2.2.2 and is:

Time taken: 0.01 seconds

2.2.6 How does the performance (test RMSE and total runtime) of your nearest neighbors algorithm compare to the baseline in part 1.3?

NN Test RMSE: 7.11504450215995 Time taken: 0.01 seconds

Baseline Test RMSE: 9.292691153610612 Time taken: 0.01 seconds

The RMSE for the nearest neighbors algorithm is better than that of the baseline. The run times appear to be equivalent.

2.3 Results and Normalization

If you were being astute, you would have noticed that we never normalized our features -- a big no-no with Nearest Neighbor algorithms. Write a generic normalization function that takes as input an array of values for a given feature, and returns the standardized array (subtract the mean and divide by the standard deviation).

Re-run the Nearest Neighbor algorithm on the normalized dataset (still just using `CRIM`, `RM` and `ZN` as input), and compare the RMSE from this method with your previous RMSE evaluations. What do you observe?

NOTE: To normalize properly, you should compute the mean and standard deviation on the training set, and use the same values to normalize both the training and the testing dataset.

NOTE 2: In this case, the normalization may or may not reduce the RMSE; don't get confused if you find that to be the case.

```
In [21]:
"""
```

```
Function
--------
Normalize data

Normalize all of the features in a data frame.

Parameters
----------
raw_data: array
    Array of numerical values to normalize.

Returns
-------
normalized_data : array
    The array with normalized values for all features
"""
# normalization function - takes in data, standard deviation, and mean
def normalize(raw_data,s,m):
    raw_data = np.array(raw_data)
    normalized_data = (raw_data - m)/s
    return normalized_data
```

In [22]:

```
# calculating standard deviation and mean
s = np.std(np.array(bdata_train[['CRIM','RM','ZN']]), axis=0)
m = np.mean(np.array(bdata_train[['CRIM','RM','ZN']]), axis=0)

# normalizing data
x_train = normalize(bdata_train[['CRIM','RM','ZN']], s,m)
x_test = normalize(bdata_test[['CRIM','RM','ZN']], s,m)

# rerunning nearest neighbors with normalized data on test set
rmse_test, test_pred = nneighbor(x_train,bdata_train[['MEDV']],x_test,bdata_test
print(rmse_test)

# rerunning nearest neighbors with normalized data on training set
rmse_train, train_pred = nneighbor(x_train,bdata_train[['MEDV']],x_train,bdata_t
print(rmse_train)
```

```
Time taken: 0.00 seconds
7.455732878524116
Time taken: 0.01 seconds
0.0
```

Training results without normalization: Time taken: 0.02 seconds 0.0

Training results with normalization: Time taken: 0.02 seconds 0.0

Test results without normalization: Time taken: 0.01 seconds 7.11504450215995

Test results with normalization: Time taken: 0.01 seconds 7.455732878524116

We can see that the training RMSE and the time taken does not change before and after normalizing the data. This makes sense because the nearest neighbor algorithm was trained with the training data and should not be calculating any difference in distance between the closest point, as there should be a corresponding point because the training set is the same.

We can see that the test RMSE increased after normalization, reflecting the changes in distance after the data has been made proportional in our nearest neighbors algorithm. The time taken between for the test data between the normalized and non-normalized data is equivalent.

2.4 Optimization

A lot of the decisions we've made so far have been arbitrary. Try to increase the performance of your nearest neighbor algorithm by adding features that you think might be relevant, and by using different values of L in the distance function. Try a model that uses a different set of 2 features, then try at least one model that uses more than 4 features, then try using a different value of L. If you're having fun, try a few different combinations of features and L! Use the test set to report the RMSE values.

What combination of features and distance function provide the lowest RMSE on the test set? Do your decisions affect the running time of the algorithm?

NOTE: For this and all subsequent questions, you should use normalized features.

In PS1, we saw a saw the strongest correlations between MEDV, RM, and ZN. So I will first attempt a model with these features

In [23]:

```
s1 = np.std(np.array(bdata_train[['RM','ZN']]), axis=0)
m1 = np.mean(np.array(bdata_train[['RM','ZN']]), axis=0)

x_train1 = normalize(bdata_train[['RM','ZN']], s1,m1)
x_test1 = normalize(bdata_test[['RM','ZN']], s1,m1)

rmse_test1, test_pred1 = nneighbor(x_train1,bdata_train[['MEDV']],x_test1,bdata_
print(rmse_test1)

rmse_train1, train_pred1 = nneighbor(x_train1,bdata_train[['MEDV']],x_train1,bda
print(rmse_train1)
```

```
Time taken: 0.00 seconds
9.346160102492217
Time taken: 0.01 seconds
0.0
```

I also want to try features with the lowest correlation, so I will try LSTAT and INDUS

In [24]:

```
s2 = np.std(np.array(bdata_train[['LSTAT','INDUS']]), axis=0)
m2 = np.mean(np.array(bdata_train[['LSTAT','INDUS']]), axis=0)

x_train2 = normalize(bdata_train[['LSTAT','INDUS']], s2,m2)
x_test2 = normalize(bdata_test[['LSTAT','INDUS']], s2,m2)

rmse_test2, test_pred2 = nneighbor(x_train2,bdata_train[['MEDV']],x_test2,bdata_
print(rmse_test2)

rmse_train2, train_pred2 = nneighbor(x_train2,bdata_train[['MEDV']],x_train2,bda
print(rmse_train2)
```

```
Time taken: 0.00 seconds
7.1875110407309935
```

```
Time taken: 0.01 seconds
0.0
```

Now I will try all 4 features above and include TAX with different distances

In [25]:
```
s3 = np.std(np.array(bdata_train[['LSTAT','INDUS','RM','ZN','TAX']]), axis=0)
m3 = np.mean(np.array(bdata_train[['LSTAT','INDUS','RM','ZN','TAX']]), axis=0)

x_train3 = normalize(bdata_train[['LSTAT','INDUS','RM','ZN','TAX']], s3,m3)
x_test3 = normalize(bdata_test[['LSTAT','INDUS','RM','ZN','TAX']], s3,m3)

# Euclidean distance
rmse_test3, test_pred3 = nneighbor(x_train3,bdata_train[['MEDV']],x_test3,bdata_
print(rmse_test3)

rmse_train3, train_pred3 = nneighbor(x_train3,bdata_train[['MEDV']],x_train3,bda
print(rmse_train3)

# L=3
rmse_test_m4, test_pred_m4 = nneighbor(x_train3,bdata_train[['MEDV']],x_test3,bd
print(rmse_test_m4)

# Manhattan Distance
rmse_test_m5, test_pred_m5 = nneighbor(x_train3,bdata_train[['MEDV']],x_test3,bd
print(rmse_test_m5)
```

```
Time taken: 0.00 seconds
5.981658316927876
Time taken: 0.01 seconds
0.0
Time taken: 0.01 seconds
5.988006385470956
Time taken: 0.00 seconds
5.910187387945489
```

What combination of features and distance function provide the lowest RMSE on the test set? Do your decisions affect the running time of the algorithm?

The combination of features that provided the lowest RMSE (5.910187387945489) on the test set was the features: LSTAT, INDUS, RM, ZN, and TAX with the manhattan distance (L=1). My decisions did not seem to affect the running time of the algorithm, at least not within one hundreth of a second.

2.5 Cross-Validation

The more you tinkered with your features and distance function, the higher the risk that you overfit your training data. One solution to this sort of overfitting is to use cross-validation (see K-fold cross-validation. Here you must implement a simple k-fold cross-validation algorithm yourself. The function you write here will be used several more times in this problem set, so do your best to write efficient code! (Note that the sklearn package has a built-in K-fold iterator -- you should *not* be invoking that or any related algorithms in this section of the problem set.)

Use 25-fold cross-validation and report the average RMSE for Nearest Neighbors using Euclidean distance with `CRIM,RM` and `ZN` input features, as well as the total running time for the full run of 25 folds. In other words, randomly divide your training dataset (created in 1.2) into 25 equally-sized samples.

For each of the 25 iterations (the "folds"), use 24 samples as "training data" (even though there is no training in k-NN!), and the remaining 1 sample for validation. Compute the RMSE of that particular validation set, then move on to the next iteration.

- Report the average cross-validated RMSE across the 25 iterations. What do you observe?
- Create a histogram of the RMSEs for the folds (there should be 25 of these). Additionally, use a horizontal line to mark the average cross-validated RMSE.

NOTE: To perform any randomized operation, only use functions in the *numpy library (np.random)*. Do not use other packages for random functions.

HINT: Running 25-fold cross validation might be time-consuming. Try starting with 5 folds.

In [26]:
```
def crossval(data,k,model,K):
    start_time_total = time.time()

    # splits the data into separate partitions
    partitions = [int(x) for x in np.linspace(0, len(data), k+1)]
    x_splits = [data[partitions[i]:partitions[i+1]][model] for i in range(len(pa
    y_splits = [data[partitions[i]:partitions[i+1]][['MEDV']] for i in range(len

    # caluculated the mean and standard deviation to be used for normalization a
    s = np.std(np.array(data[model]), axis=0)
    m = np.mean(np.array(data[model]), axis=0)

    rmses = []
    for i in range(k): # runs k folds
        # defines data splits
        x_train = np.concatenate(x_splits[:i] + x_splits[i+1:])
        y_train = np.concatenate(y_splits[:i] + y_splits[i+1:])
        x_val = np.array(x_splits[i])
        y_val = np.array(y_splits[i])

        # normalizes data
        x_train_norm = normalize(np.array(x_train),s,m)
        x_val_norm = normalize(np.array(x_val),s,m)


        if K == 0: # runs nearest neighbor function if K is 0
            rmse_test, test_pred = nneighbor(x_train_norm,y_train,x_val_norm,y_v
        else: # runs knn otherwise
            rmse_test, test_pred = knn(x_train_norm, y_train, x_val_norm, y_val,
        rmses.append(rmse_test)

    print("Total time taken: {:.2f} seconds".format(time.time() - start_time_tot
    return rmses
```

In [27]:
```
# Report the average cross-validated RMSE across the 25 iterations. What do you
rmses = crossval(bdata_train,25,['CRIM','RM','ZN'],0)
```

```
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.03 seconds
```

In [28]:

```
# Report the average cross-validated RMSE across the 25 iterations. What do you
avg = np.mean(np.array(rmses))
print(avg)
```

```
6.800408907307364
```

Running nearest neighbors under cross validation seems to return a lower average RMSE (6.80) across the 25 iterations than running nearest neighbors alone on the test set. Could indicate that we might get a lower RMSE when we train with cross validation before running an algorithm on our test set.

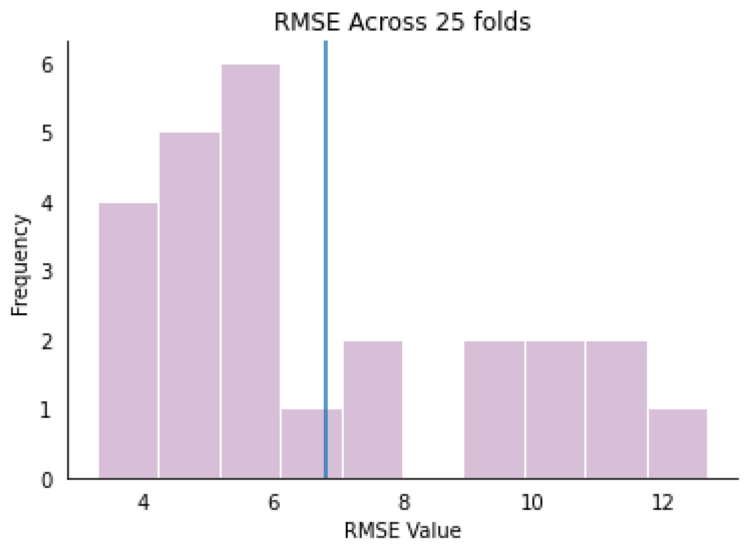
In [29]:

```
#Create a histogram of the RMSEs for the folds (there should be 25 of these).
#Additionally, use a horizontal line to mark the average cross-validated RMSE.

plt.hist(rmses, facecolor='thistle', edgecolor='white')
plt.axvline(x=avg)

plt.xlabel("RMSE Value")
plt.ylabel("Frequency")
plt.title("RMSE Across 25 folds")

#polishing
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.yaxis.set_ticks_position('none')
ax.xaxis.set_ticks_position('none')
```



2.6 K-Nearest Neighbors Algorithm

Implement the K-Nearest Neighbors algorithm. Use 10-fold cross validation and L2 normalization, and the same features as in 2.5. Report the RMSE for K=5 and the running time of the algorithm. What do you observe?

In [30]:
```
"""
Function
--------
K-Nearest Neighbors

Implementation of nearest neighbors algorithm.

Parameters
----------
x_train: array
    Array of numerical feature values for training the model.
y_train: array
    Array of numerical output values for training the model.
x_test: array
    Array of numerical feature values for testing the model.
y_test: array
    Array of numerical output values for testing the model.
L: int
    Order of L-norm function used for calculating distance.
K: int
    Neighbors to include in algorithm

Returns
-------
rmse : int
    Value of the RMSE from data.
"""

def knn(x_train, y_train, x_test, y_test, L, K):
    start_time = time.time()
    pred = []
    for i in range(len(x_test)):
        # finds the distances
        dists = distance(np.array(x_test[i]), np.array(x_train), L)
        # finds the average of the K nearest distances
        avg = [np.array(y_train[np.argsort(dists)[:K]].mean())]
        pred.append(avg)
    # computes RMSE
    rmse = compute_rmse(np.array(pred), np.array(y_test))
    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return rmse, pred
```

In [31]:

```
# Report the RMSE for K=5 and the running time of the algorithm. What do you obs
rmses = crossval(bdata_train,10,['CRIM','RM','ZN'],5)
```

```
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.03 seconds
```

In [32]:

```
# Report the RMSE for K=5 and the running time of the algorithm. What do you obs
avg = np.mean(np.array(rmses))
print(avg)
```

```
5.23747187990516
```

We can see that the average RMSE for KNN (5.23) is lower than the average RMSE for NN (6.80), which implies KNN will be a better algorithm for our data.

2.7 Using cross validation to find K

Compute the cross-validated RMSE for values of K between 1 and 25 using 10-fold cross-validation and L2 normalization. Use the following features in your model: `CRIM, ZN, RM, AGE, DIS, TAX`. Create a graph that shows how cross-validated RMSE changes as K increases from 1 to 25. Label your axes, and summarize what you see. What do you think is a reasonable choice of K for this model?

Finally, report the test RMSE using the value of K that minimized the cross-validated RMSE. (Continue to use L2 normalization and the same set of features). How does the test RMSE compare to the cross-validated RMSE, and is this what you expected? How does the test RMSE compare to the test RMSE from 2.4, and is this what you expected?

In [33]:

```
rmses = []
for i in range(1,26):
    rmse = crossval(bdata_train,10,['CRIM','ZN','RM','AGE','DIS','TAX'],i)
    rmses.append(np.mean(np.array(rmse)))
```

```
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.03 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.03 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.03 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
```

```
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
```

```
Total time taken: 0.02 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Time taken: 0.00 seconds
Total time taken: 0.02 seconds
```

In [34]:

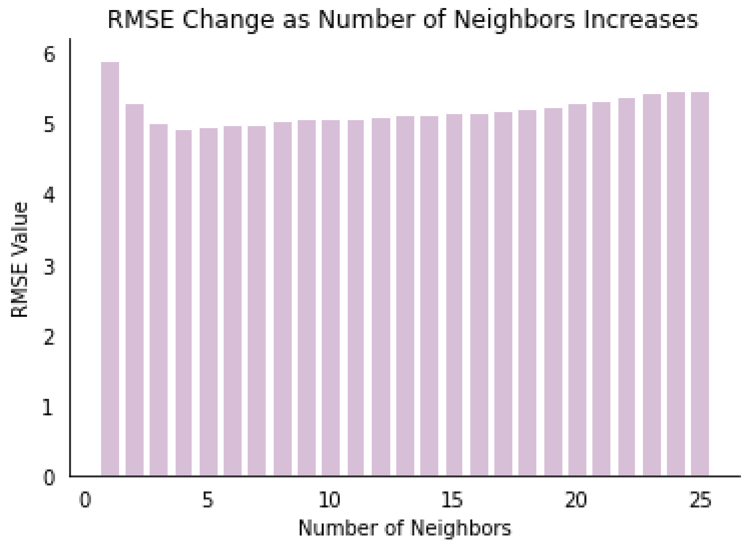
```python
# Create a graph that shows how cross-validated RMSE changes as K increases from
# Label your axes, and summarize what you see. What do you think is a reasonable
y = np.arange(1, 26, 1)
plt.figure()
plt.bar(y, rmses, facecolor='thistle', edgecolor='white')

plt.xlabel("Number of Neighbors")
plt.ylabel("RMSE Value")
plt.title("RMSE Change as Number of Neighbors Increases")

#polishing
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.yaxis.set_ticks_position('none')
ax.xaxis.set_ticks_position('none')

plt.show()

print(min(rmses))
print("Best choice of K: " + str(rmses.index(min(rmses))+1))
```



```
4.924371375718152
Best choice of K: 4
```

It looks like the minimum RMSE value occurs at K = 4, which is verified by taking the index of the minimum value of our rmses array returned by our cross validation function

In [35]:

```python
# report the test RMSE using the value of K that minimized the cross-validated R
```

```
s = np.std(np.array(bdata_train[['CRIM','ZN','RM','AGE','DIS','TAX']]), axis=0)
m = np.mean(np.array(bdata_train[['CRIM','ZN','RM','AGE','DIS','TAX']]), axis=0)

x_train = normalize(bdata_train[['CRIM','ZN','RM','AGE','DIS','TAX']], s,m)
x_test = normalize(bdata_test[['CRIM','ZN','RM','AGE','DIS','TAX']], s,m)

rmse, pred = knn(np.array(x_train), np.array(bdata_train[['MEDV']]), np.array(x_
print(rmse)
```

```
Time taken: 0.01 seconds
6.001738183265948
```

How does the test RMSE compare to the cross-validated RMSE, and is this what you expected? How does the test RMSE compare to the test RMSE from 2.4, and is this what you expected?

Our test RMSE (6.00) is higher than our cross-validated RMSE (4.92) and . This is to be expected, as we are using more data to find the neighbors and we are using every value in our training set to validate, thus we have a lower RMSE in cross validation.

Our test RMSE (6.00) is also higher than our best RMSE from 2.4 (5.91). This could be because the features chosen LSTAT, INDUS, RM, ZN, and TAX perform better than the features selected here: CRIM, ZN, RM, AGE, DIS, TAX. We also used a different distance measure in 2.4 (Manhattan difference) where we used the Euclidean difference here. This could also potentially indicate that nearest neighbors performs better than KNN for our data.

Extra-Credit: Forward selection

Thus far the choice of predictor variables has been rather arbitrary. For extra credit, implement a basic forward selection algorithm to progressively include features that decrease the cross-validated RMSE of the model. Note that the optimal value of K may be different for each model, so you may want to use cross-validation to choose K each time (but it is also fine if you fix K at the optimal value from 2.7). Create a graph that shows RMSE as a function of the number of features in the model. Label each point on the x-axis with the name of the feature that is added at that step in the forward selection algorithm. *(For instance, if the optimal single-feature model has CRIM with RMSE = 10, and the optimal two-feature model has CRIM+ZN with RMSE=9, the first x-axis label will say CRIM and the second x-axis lable with say ZN)*

In [36]:
```
# enter your code here
```