# 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 [2]:

```
import IPython
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib
```

In [3]:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

---

# Introduction to the assignment

For this assignment, you will be using the Boston Housing Prices Data Set (http://www.kellogg.northwestern.edu/faculty/weber/emp/_session_3/boston.htm). 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 [4]:

```
# load Boston housing data set
data = np.loadtxt('data.txt')
target = np.loadtxt('target.txt')
```

---

# 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.

## 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 (http://en.wikipedia.org/wiki/Root-mean-square_deviation). 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 [5]:

```
"""
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 observa

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

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):
    temp = list(map(lambda i, j : (i-j)**2, predictions, yvalues))
    rmse = np.sqrt(sum(temp)/len(predictions))
    return rmse
```

In [6]:

```
compute_rmse((4,6,3),(2,1,4))
```

Out[6]:

```
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 [7]:

```
bdata = pd.concat([pd.DataFrame(data), pd.DataFrame(target)], axis = 1)
bdata.columns = ['CRIM','ZN','INDUS','CHAS','NOX','RM','AGE','DIS','RAD','TAX','PTRA
bdata.head()
```

Out[7]:

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.218960 | 18.0 | 2.629288 | 0.0 | 0.869420 | 6.875396 | 65.2 | 4.347275 | 1.0 | 307.0 | 15.534711 |
| 1 | 0.141576 | 0.0 | 7.315612 | 0.0 | 0.549711 | 6.499894 | 78.9 | 5.315684 | 2.0 | 255.0 | 17.914131 |
| 2 | 0.380457 | 0.0 | 7.340354 | 0.0 | 0.697928 | 7.263489 | 61.1 | 5.356935 | 2.0 | 243.0 | 17.919989 |
| 3 | 0.313563 | 0.0 | 2.562407 | 0.0 | 0.599629 | 7.209732 | 45.8 | 6.103983 | 3.0 | 226.0 | 18.979527 |
| 4 | 0.330105 | 0.0 | 2.497337 | 0.0 | 0.476077 | 7.184111 | 54.2 | 6.264372 | 3.0 | 234.0 | 18.708888 |

In [8]:

```
# leave the following line untouched, it will help ensure that your "random" split i
np.random.seed(seed=13579)

# enter your code here
bdata = pd.concat([pd.DataFrame(data), pd.DataFrame(target)], axis = 1)
train_percent = .75
train_number = int(train_percent*len(bdata))
print('Total examples: %i' % len(bdata))
print('Number of training examples: %i' % train_number)
print('Number of testing examples: %i' % (len(bdata) - train_number))

ids = np.arange(0, len(bdata), 1)
ids = np.random.permutation(ids)
df_shuffled = bdata.iloc[ids]
bdata_train = df_shuffled[:train_number]
bdata_test = df_shuffled[train_number:]
bdata_train.columns = ['CRIM','ZN','INDUS','CHAS','NOX','RM','AGE','DIS','RAD','TAX'
bdata_test.columns = ['CRIM','ZN','INDUS','CHAS','NOX','RM','AGE','DIS','RAD','TAX',
```

```
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 [9]:

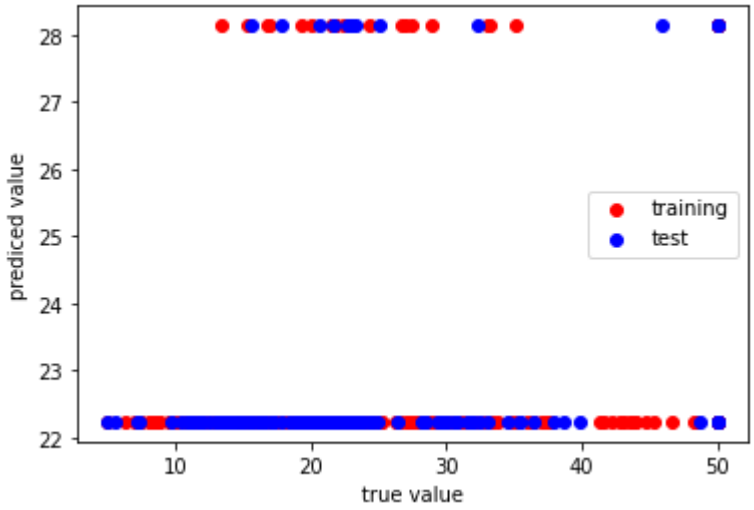
```
import time

# enter your code here
# 1.3.1  Compute the RMSE of the training set
start_time = time.time()
tr = bdata_train[['CHAS','MEDV']]
model = tr.groupby('CHAS').mean()
tr['predict_MEDV'] = tr['CHAS'].apply(lambda x: model.loc[1,'MEDV'] if x == 1 else m
print('=======1.3.1=======')
print(' rmse = %.4f ' % compute_rmse(tr['MEDV'],tr['predict_MEDV']))
print(" runtime = %.4f seconds " % (time.time() - start_time))

#1.3.2 compute the RMSE of the test data set
start_time = time.time()
te = bdata_test[['CHAS','MEDV']]
te['predict_MEDV'] = te['CHAS'].apply(lambda x: model.loc[1,'MEDV'] if x == 1 else m
print('=======1.3.2=======')
print('rmse = %.4f ' % compute_rmse(te['MEDV'],te['predict_MEDV']))
print("runtime = %.4f seconds " % (time.time() - start_time))

# 1.3.4 scatter plot
fig, ax = plt.subplots()

plt.scatter(tr['MEDV'], tr['predict_MEDV'], color = 'red')
plt.scatter(te['MEDV'], te['predict_MEDV'], color = 'blue')

ax.set_ylabel('prediced value')
ax.set_xlabel('true value')

ax.legend(['training', 'test'])
```

```
=======1.3.1=======
 rmse = 8.9634
 runtime = 0.0155 seconds
=======1.3.2=======
rmse = 9.2927
runtime = 0.0026 seconds

/var/folders/2h/4fcst_952sn2rw58_smpkpzw0000gn/T/ipykernel_26347/41226
29022.py:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas
-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
 (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
  tr['predict_MEDV'] = tr['CHAS'].apply(lambda x: model.loc[1,'MEDV']
 if x == 1 else model.loc[0,'MEDV'])
/var/folders/2h/4fcst_952sn2rw58_smpkpzw0000gn/T/ipykernel_26347/41226
29022.py:16: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas
-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
 (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
```

```
    te['predict_MEDV'] = te['CHAS'].apply(lambda x: model.loc[1,'MEDV']
   if x == 1 else model.loc[0,'MEDV'])
```

Out[9]:

```
<matplotlib.legend.Legend at 0x11ed36490>
```




- 1.3.1 rmse = 8.9634
- 1.3.2 rmse = 9.2927
- 1.3.3 The RMSE of the test set is slightly lower (0.3) than the training set. Because the model was build by the data from the trainig set, it is not surprising that the test set has a smaller rmse.

---

# 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 (https://en.wikipedia.org/wiki/Norm_(mathematics)#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 [11]:

```
"""
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 observa

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

"""
def distance(x1, x2, L):
    # your code here
    temp = list(map(lambda i, j : (i-j)**L, x1, x2))
    dist = np.sum(temp)**(1./L)
    return dist
```

In [10]:

```
print(distance((2,7),(5,11),2))
```

```
5.0
```

## 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?

In [12]:

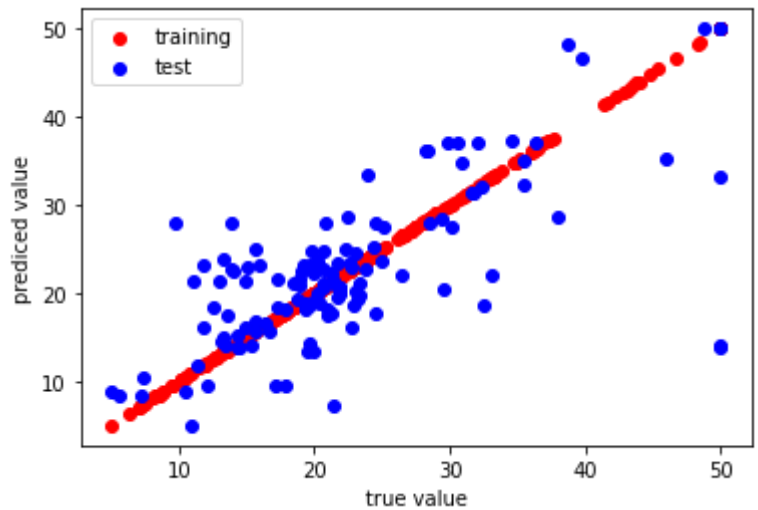
```
"""
Function
--------
Nearest Neighbors

Implementation of nearest neighbors algorithm.

Parameters
----------
x_train: array
    Array of numerical feature values for training the model.   #CRIM, RM, ZN of tra
y_train: array
    Array of numerical output values for training the model.    #MEDV of training se
x_test: array
    Array of numerical feature values for testing the model.   #CRIM, RM, ZN of test
y_test: array
    Array of numerical output values for testing the model.   #MEDV of test set
L: int
    Order of L-norm function used for calculating distance.

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

def nneighbor(x_train, y_train, x_test, y_test, L):
    start_time = time.time()
    #your code here
    closest_points = []
    for i in range(len(x_test)):
        min_dist = np.inf
        min_idx = 0
        #dists = np.sqrt(np.sum((x_test[i] - x_train[:])**2, axis=1))
        for j in range(len(x_train)):
            temp = distance(x_test[i], x_train[j], L)
            #print(j)
            if temp < min_dist:
                min_dist = temp
                min_idx = j
        closest_points.append(y_train[min_idx])

    rmse = compute_rmse(closest_points, y_test)

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return rmse

#your additional code here

x_test = np.array(bdata_test[['CRIM', 'RM', 'ZN']])
y_test = np.array(bdata_test['MEDV'])
x_train = np.array(bdata_train[['CRIM', 'RM', 'ZN']])
y_train = np.array(bdata_train['MEDV'])

# 2.2.2
print('=======2.2.2=======')
print('rmse = %.4f' % nneighbor(x_train, y_train, x_train, y_train, 2))

# 2.2.3
```

```
print('=======2.2.3=======')
print('rmse = %.4f' % nneighbor(x_train, y_train, x_test, y_test, 2))

# 2.2.4 scatter plot
def nneighbor_array(x_train, y_train, x_test, L): # the function returns a list
    #your code here
    closest_points = []
    for i in range(len(x_test)):
        min_dist = np.inf
        min_idx = 0
        for j in range(len(x_train)):
            temp = distance(x_test[i], x_train[j], L)
            if temp < min_dist:
                min_dist = temp
                min_idx = j
        closest_points.append(y_train[min_idx])
    return closest_points

fig, ax = plt.subplots()
plt.scatter(y_train, nneighbor_array(x_train, y_train, x_train, 2), color = 'red')
plt.scatter(y_test, nneighbor_array(x_train, y_train, x_test, 2), color = 'blue')

ax.set_ylabel('prediced value')
ax.set_xlabel('true value')

ax.legend(['training', 'test'])
```

```
=======2.2.2=======
Time taken: 0.54 seconds
rmse = 0.0000
=======2.2.3=======
Time taken: 0.17 seconds
rmse = 7.1150
```

Out[12]:

```
<matplotlib.legend.Legend at 0x11ed382e0>
```




- 2.2.5 0.57 seconds for the training set and 0.19 for the test set
- 2.2.6 The performance of the nneighbors is much better than the baseline. In the nnneighbors, the predicted value is equal to the true value in the traning set (rmse=0). A more dioganol distribution and a smaller rmse (7.115) of the test set also indicates a better predcition compared with the baseline.

## 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 [13]:

```
"""
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
"""
# function returns the mean and std of the input array
def MS(raw_data):
    a=np.mean(raw_data) #compute the mean of each col
    b=np.std(raw_data) #compute the std of each col
    df = pd.DataFrame(np.vstack((a,b))) #concat
    df.columns = raw_data.columns #add col names
    return df

def normalize(raw_data, meanstd):
    normalized_data = pd.DataFrame()
    for col in raw_data.columns:
        normalized_data[col] = (raw_data[col] - meanstd.loc[0, col])/meanstd.loc[1,
    return normalized_data

#your additional code here
MeanStd = MS(bdata_train[['CRIM', 'RM', 'ZN']])
x_train_norm = np.array(normalize(bdata_train[['CRIM', 'RM', 'ZN']], MeanStd))
x_test_norm = np.array(normalize(bdata_test[['CRIM', 'RM', 'ZN']], MeanStd))

print('=======2.3 training set=======')
print('rmse  = %.4f' % nneighbor(x_train_norm, y_train, x_train_norm, y_train, 2))
print('=======2.3 test set=======')
print('rmse  = %.4f' % nneighbor(x_train_norm, y_train, x_test_norm, y_test, 2))
```

```
=======2.3 training set=======
Time taken: 0.54 seconds
rmse  = 0.0000
=======2.3 test set=======
Time taken: 0.17 seconds
rmse  = 7.4557
```

- 2.3 The result doesn't change much. The RMSE slightly increases from 7.115 to 7.4557 after normalization.

## 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 [14]:

```
# model using two different features ('DIS', 'AGE')
features = ['DIS', 'AGE']
MS_m2 = MS(bdata_train[features])
x_train_m2_norm = np.array(normalize(bdata_train[features], MS_m2))
x_test_m2_norm = np.array(normalize(bdata_test[features], MS_m2))
print('=======model using two different features (\'DIS', 'AGE\')=======')
print('rmse  = %.4f' % nneighbor(x_train_m2_norm, y_train, x_test_m2_norm, y_test, 2

# model using 4 features ('DIS', 'AGE','CRIM','RM')
features = ['DIS', 'AGE','CRIM','RM']
MS_m3 = MS(bdata_train[features])
x_train_m3_norm = np.array(normalize(bdata_train[features], MS_m3))
x_test_m3_norm = np.array(normalize(bdata_test[features], MS_m3))
print('======= model using 4 features (\'DIS', 'AGE','CRIM','RM\') =======')
print('rmse  = %.4f' % nneighbor(x_train_m3_norm, y_train, x_test_m3_norm, y_test, 2

# model using L=4 & 4 features('DIS', 'AGE','CRIM','RM')
print('======= model using L=4 & 4 features(\'DIS', 'AGE','CRIM','RM\') =======')
print('rmse  = %.4f' % nneighbor(x_train_m3_norm, y_train, x_test_m3_norm, y_test, 4
```

```
=======model using two different features ('DIS AGE')=======
Time taken: 0.17 seconds
rmse  = 12.0547
======= model using 4 features ('DIS AGE CRIM RM') =======
Time taken: 0.18 seconds
rmse  = 6.5063
======= model using L=4 & 4 features('DIS AGE CRIM RM') =======
Time taken: 0.18 seconds
rmse  = 6.3786
```

- The combination of 4 features('DIS', 'AGE','CRIM','RM') and L=4 provide the lowest RMSE (6.3786).
- The runtime observed here does not change a lot as the number of features and L change.

## 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 (http://en.wikipedia.org/wiki/Cross-validation_(statistics)). 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 (http://scikit-learn.org/stable/modules/cross_validation.html#cross-validation) 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.

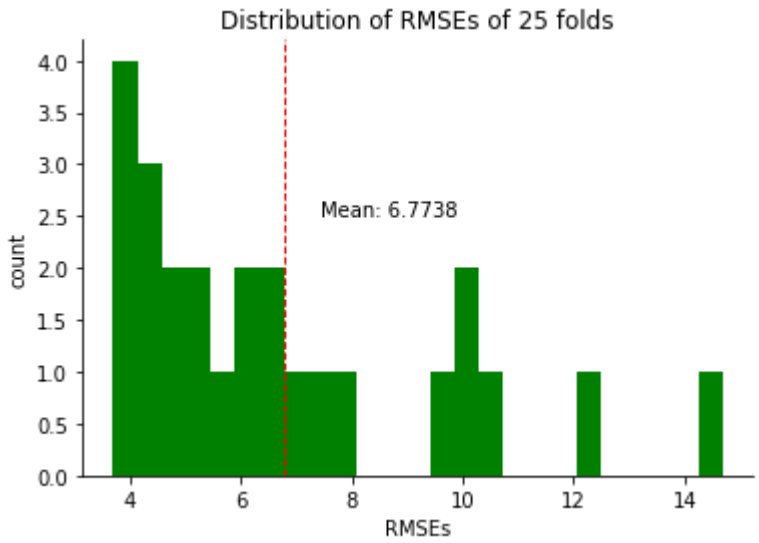
NOTE: To perform any randomized operation, only use functions in the *numpy library (np.random)*. Do not use

In [15]:

```
# the function extracts cross validation splits
def cross_validation_split(df, folds):
    ids = np.arange(0, len(df), 1)
    np.random.shuffle(ids) # What's weird about this syntax?
    df_shuffled = df.iloc[ids]

    partitions = [int(x) for x in np.linspace(0, len(df), folds+1)]
    splits = [df_shuffled[partitions[i]:partitions[i+1]] for i in range(len(partitic

    return splits

# 25-fold cross validation
nfolds = 25
RMSE = []
splits = cross_validation_split(bdata_train[['CRIM', 'RM', 'ZN','MEDV']],nfolds)

start_time = time.time()

for i in range(nfolds):
    tr = [x for j,x in enumerate(splits) if j!=i]
    train = pd.concat(tr)[['CRIM', 'RM', 'ZN']]
    test = splits[i][['CRIM', 'RM', 'ZN']]
    MeanStd = MS(train)

    xtrain_norm = np.array(normalize(train, MeanStd))
    ytrain =  np.array(pd.concat(tr)['MEDV'])
    xtest_norm =  np.array(normalize(test, MeanStd))
    ytest = np.array(splits[i]['MEDV'])

    RMSE.append(nneighbor(xtrain_norm, ytrain, xtest_norm, ytest, 2))


# compute mean RMSE
print('======= 2.5 =======')
print('average cross-validated RMSE = %.4f' %np.mean(RMSE))
print('total runtime = %.4f' % (time.time() - start_time)) #total runtime

#histogram
fig, ax = plt.subplots()
num_of_bins = len(RMSE)
plt.hist(RMSE, color='green', bins = int(num_of_bins))
plt.axvline(np.mean(RMSE), color='r', linestyle='dashed', linewidth=1) #add average
plt.text(np.mean(RMSE)*1.1, 2.5, 'Mean: {:.4f}'.format(np.mean(RMSE))) #add average
plt.title('Distribution of RMSEs of 25 folds')
ax.set_ylabel('count')
ax.set_xlabel('RMSEs')
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
```

```
Time taken: 0.04 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
```

```
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
Time taken: 0.02 seconds
======= 2.5 =======
average cross-validated RMSE = 6.7738
total runtime = 0.6127
```




- 2.5 The average RMSE of my cross validation is 6.7738 and the total runtime is 0.6733 seconds. The distribution is not very equal, probably because there are only about 15 data in each fold.

## 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 [16]:

```
"""
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()
    # enter your code here
    closest_points = []
    for i in range(len(x_test)):
        temp = []
        for j in range(len(x_train)):
            temp.append(distance(x_test[i], x_train[j], L)) #list contain all the di
        idxs = pd.Series(temp).sort_values().index[:K] #sort the list and pick the f
        dist = np.mean(y_train[idxs]) # compute the mean distance of the k nearest p
        closest_points.append(dist)

    rmse = compute_rmse(closest_points, y_test)
    runtime = time.time() - start_time
    #print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return rmse, runtime

# enter your additional code here
nfolds = 10
RMSE =[]
RT = []
splits = cross_validation_split(bdata_train[['CRIM', 'RM', 'ZN','MEDV']],nfolds)

start_time = time.time()

for i in range(nfolds):
    tr = [x for j,x in enumerate(splits) if j!=i]
    train = pd.concat(tr)[['CRIM', 'RM', 'ZN']]
    test = splits[i][['CRIM', 'RM', 'ZN']]
    MeanStd = MS(train)
```

```
    xtrain_norm = np.array(normalize(train, MeanStd))
    ytrain =  np.array(pd.concat(tr)['MEDV'])
    xtest_norm =  np.array(normalize(test, MeanStd))
    ytest = np.array(splits[i]['MEDV'])

    a, b = knn(xtrain_norm, ytrain, xtest_norm, ytest, 2, 5)
    RMSE.append(a)
    RT.append(b)

print('======= 2.6 =======')
print('average cross-validated RMSE = %.4f' %np.mean(RMSE))
print('average cross-validated runtime = %.4f' %np.mean(RT))
print('total runtime = %.4f' % (time.time() - start_time)) #total runtime
```

```
======= 2.6 =======
average cross-validated RMSE = 5.2110
average cross-validated runtime = 0.0523
total runtime = 0.5614
```

- 2.6 Compared with the result of nneighbor algorithm, the RMSE of knn is smaller(from 6.7738 to 5.2110) and the total runtime (from 0.6733 to 0.5862 sec) is smaller.

## 2.7 Using cross validation to find K

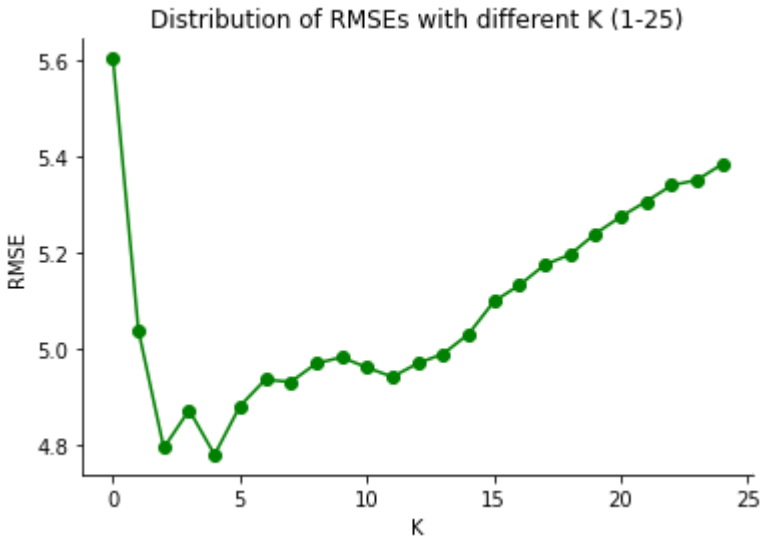
Compute the cross-validated RMSE for values of K between 1 and 25 using 10-fold cross-validation and L2 normalization. Use the following features in your model: `CRIM, ZN, RM, AGE, DIS, TAX` . Create a graph that shows how cross-validated RMSE changes as K increases from 1 to 25. Label your axes, and 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 [23]:

```
# enter your code here
nfolds = 10
AVE_RMSE = []
AVE_RT = []
features = ['CRIM', 'RM', 'ZN', 'AGE', 'DIS', 'TAX']
splits = cross_validation_split(bdata_train[['CRIM', 'RM', 'ZN', 'AGE', 'DIS', 'TAX'

start_time = time.time()

for k in range(1, 26): # k from 1 to 25
    RMSE = []
    RT = []
    for i in range(nfolds):
        tr = [x for j,x in enumerate(splits) if j!=i]
        train = pd.concat(tr)[features]
        test = splits[i][features]
        MeanStd = MS(train)

        xtrain_norm = np.array(normalize(train, MeanStd))
        ytrain =  np.array(pd.concat(tr)['MEDV'])
        xtest_norm =  np.array(normalize(test, MeanStd))
        ytest = np.array(splits[i]['MEDV'])

        a, b = knn(xtrain_norm, ytrain, xtest_norm, ytest, 2, k)
        RMSE.append(a)
        RT.append(b)

    AVE_RMSE.append(np.mean(RMSE))
    AVE_RT.append(np.mean(RT))

print('======= cross-validated RMSE for values of K between 1 and 25 =======')
print('average cross-validated RMSE = %.4f' %np.mean(AVE_RMSE))
print('average cross-validated runtime = %.4f' %np.mean(AVE_RT))
print('total runtime = %.4f' % (time.time() - start_time)) #total runtime

#lineplot
fig, ax = plt.subplots()
plt.plot(pd.Series(AVE_RMSE).index,pd.Series(AVE_RMSE),'-go')
plt.title('Distribution of RMSEs with different K (1-25)')
ax.set_ylabel('RMSE')
ax.set_xlabel('K')
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
```

```
======= cross-validated RMSE for values of K between 1 and 25 =======
average cross-validated RMSE = 5.0870
average cross-validated runtime = 0.0640
total runtime = 17.1319
```




In [22]:

```
features = ['CRIM', 'RM', 'ZN', 'AGE', 'DIS', 'TAX']
L = 2
K = 5
ms = MS(bdata_train[features])
x_train_norm = np.array(normalize(bdata_train[features],ms))
y_train = np.array(bdata_train['MEDV'])
x_test_norm = np.array(normalize(bdata_test[features],ms))
y_test = np.array(bdata_test['MEDV'])

rmse, rt = knn(x_train_norm, y_train, x_test_norm, y_test, L, K)

print('======= test set =======')
print('RMSE = %.4f' % rmse)
print('runtime = %.4f' % rt)
```

```
======= test set =======
RMSE = 5.9130
runtime = 0.2658
```

- k = 5 is the reasonable choice of k for this model because it has the lowest RMSE.
- Compared with the average cross-validated RMSE, the test RMSE is higher (5.913). The result is expected because the rmse values shown on the line plot are the average rmse values of all 10 folds of each k.
- Compared with the minimun rmse I got in 2.4 (6.3786), the test RMSE is much smaller(5.9130). The lower rmse is expected because the value of K that minimized the cross-validated RMSE and knn rather than nneighbor is used here .

## Extra-Credit: Forward selection

Thus far the choice of predictor variables has been rather arbitrary. For extra credit, implement a basic [forward selection (https://see.stanford.edu/materials/aimlcs229/cs229-notes5.pdf)](https://see.stanford.edu/materials/aimlcs229/cs229-notes5.pdf) 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 [18]:

```
# enter your code here
```