# House Price Prediction Model Report

**Amrinder Sehmbi**

## Introduction

In this assignment, we will be predicting the prices of houses using features like the house's age, distance to public transportation, and latitude/longitude coordinates. The data that we are using is the Real Estate Valuation Data Set from https://archive.ics.uci.edu/ml/datasets/Real+estate+valuation+data+set .

We will be exploring both k Nearest Neighbour models and Linear Regression models.

## Data, Indexing, and Vectorized Code

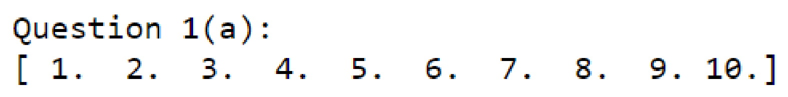
Before beginning a machine learning task, one of the first thing that you should do is to understand the data that you are working with. That is where we should start: with the data. Along the way, we will illustrate how to use Python's numpy package to vectorize computation.

```
import numpy as np
import numpy.random as rnd

# Read the data
# Download "data.csv" from the course website
data = np.genfromtxt('data.csv', delimiter=',', skip_header=1)

# Display the *shape* of the data matrix
print(data.shape)

# Please leave these print statements, to help your TAs grade quickly.
print('\n\nQuestion 1')
print('----------')
```

### Part (a)

Print the first column and the first 10 rows of the data. Recall that you should not add loops that are not already in the starter code. Note that data is a 2D numpy array (i.e. a matrix), and its elements can be indexed. For examples data[0, 0] indexes the first row and column. Additionally, similar to python lists, numpy arrays support slicing: e.g. data[1:3, 0] and data[200:, :].

```
print('\nQuestion 1(a):') # Please leave print statements like these

#print('\nFirst column and the first 10 rows of the data')
print(data[0:10,0])
```



```
Question 1(a):
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

**Part (b)**

Print the second column and the first 10 rows of the data.

```
print('\nQuestion 1(b):')

#print('\nSecond column and the first 10 rows of the data')
print(data[0:10,1])
```

```
Question 1(b):
[2012.917 2012.917 2013.583 2013.5   2012.833 2012.667 2012.667 2013.417
 2013.5   2013.417]
```

**Part (c)**

What do you think the columns in parts (a) and (b) represent? Find the answer by reading the data set information and "Attribute Information" in https://archive.ics.uci.edu/ml/datasets/Real+estate+valuation+data+set. You should understand the meaning of the remaining fields as well.

We will be predicting the housing price (last column) using the some of the remaining features.

The first column of the data set indicates the number of the house in the data set and the second column in the data set indicates the transaction date of the house, for example, 2013.250=2013 March.

**Part (d)**

Remove the first column from the data, and overwrite the variable data with the result. Print the shape of the resulting matrix data.

```
print('\nQuestion 1(d):')

data = data[:, 1: ]

#print('\nShape of data')
print(data.shape)

test = data[data[:, 0] > 2013.417]
```

```
Question 1(d):
(414, 7)
```

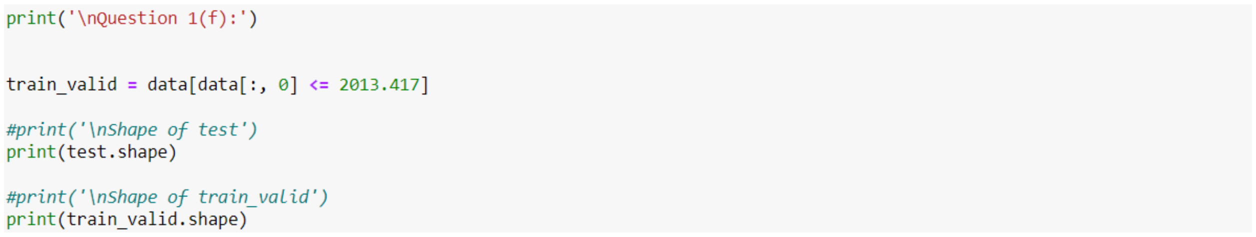
**Part (e)**

We will first separate the data into training, validation, and test sets. Rather than choosing a random percentage of data points to leave out in our test set, we will instead place the *most recent* data points in our test set. In particular, any data point with date *larger* than 2013.417 will be placed in our test set. The code to select the test set element is written for you below. Pay attention to the way a boolean numpy array like data[ : , 0] > 2013.417 can be used to index elements of another numpy array. Explain why this is a better strategy than randomly selecting data points in our test set.

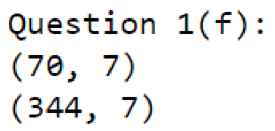
If we randomly group, the data then we could have data from before and after the date 2013.417 in the test set and training set. This would not help us test the accuracy of predicting future prices of the model since the model could be trained to predict prices for date range in the data. If we instead place the most recent data points in the test set and place older data points in the training set, then we could test the accuracy of the predictions from our model by giving it unseen data. This simulates the idea of predicting future prices which we could compare with prices in the test set to measure accuracy.

**Part (f)**

Create a matrix train_valid that contains the data points that will be in the training or validation set. Then, print the shape of both new matrices from parts (e) and (f).



```
print('\nQuestion 1(f):')


train_valid = data[data[:, 0] <= 2013.417]

#print('\nShape of test')
print(test.shape)

#print('\nShape of train_valid')
print(train_valid.shape)
```



```
Question 1(f):
(70, 7)
(344, 7)
```

**Part (g)**

We will use the variable randarray, given below, to separate our training and validation sets. This array assigns a random integer (0, 1, 2, 3, 4) to every element of the train_valid dictionary. For each data point in train_valid, if its corresponding value in randarray is 0, place that data point in the validation set. Otherwise, place that data point in the training set. To earn credit, you should do this without using any loops. (Hint: consider the way we indexed numpy arrays in parts e and f) Print the shape of the training and validation matrices

```
print('\nQuestion 1(g):')

# Below array was generated by calling
# randarray = rnd.randint(0, 5, train_valid.shape[0])
# Do NOT uncomment the above line of code. Instead, we are including
# the values you should use below.
randarray = np.array([2, 0, 1, 3, 0, 0, 0, 3, 2, 3, 1, 1, 2, 0, 4, 4, 0, 2, 1, 2, 2, 2,
                      4, 1, 3, 2, 0, 1, 2, 0, 3, 0, 3, 1, 3, 0, 4, 1, 4, 4, 0, 0, 1, 2,
                      4, 0, 0, 1, 1, 1, 2, 3, 4, 4, 3, 3, 0, 0, 0, 0, 2, 2, 3, 0, 0, 1,
                      4, 1, 4, 2, 2, 4, 4, 2, 0, 4, 0, 3, 2, 0, 4, 3, 1, 1, 0, 0, 0, 0,
                      1, 1, 4, 0, 3, 4, 2, 0, 0, 4, 4, 4, 4, 3, 3, 0, 0, 2, 2, 1, 3, 2,
                      4, 1, 2, 2, 3, 4, 1, 4, 3, 1, 1, 3, 0, 4, 4, 4, 0, 3, 3, 0, 4, 0,
                      0, 4, 3, 4, 1, 2, 2, 4, 4, 1, 2, 1, 1, 0, 4, 4, 4, 2, 0, 4, 2, 0,
                      4, 4, 1, 4, 0, 4, 0, 0, 1, 4, 3, 2, 4, 3, 1, 4, 1, 3, 4, 1, 0, 0,
                      4, 4, 2, 0, 4, 4, 4, 2, 3, 3, 4, 1, 0, 1, 2, 3, 1, 0, 1, 3, 4, 0,
                      0, 1, 2, 2, 2, 2, 4, 3, 1, 1, 4, 4, 1, 4, 2, 4, 0, 2, 4, 1, 3, 0,
                      4, 2, 3, 0, 4, 2, 3, 2, 2, 0, 2, 0, 2, 3, 2, 3, 4, 2, 4, 2, 2, 4,
                      3, 4, 0, 4, 4, 0, 1, 4, 2, 4, 2, 4, 0, 3, 4, 2, 1, 1, 3, 0, 1, 0,
                      3, 1, 3, 2, 4, 3, 1, 3, 0, 3, 0, 4, 2, 1, 2, 3, 2, 2, 4, 1, 4, 2,
                      1, 3, 1, 2, 2, 3, 1, 4, 2, 2, 4, 4, 1, 3, 2, 4, 1, 2, 4, 4, 0, 3,
                      1, 2, 1, 3, 3, 3, 2, 1, 3, 0, 2, 4, 2, 0, 3, 1, 0, 4, 2, 4, 1, 0,
                      4, 2, 2, 0, 1, 4, 3, 4, 0, 3, 2, 0, 2, 0])


train = train_valid[randarray[0:] != 0]
valid = train_valid[randarray[0:] == 0]


#print('\nShape of train')
print(train.shape)

#print('\nShape of valid')
print(valid.shape)
```

```
Question 1(g):
(273, 7)
(71, 7)
```

**Part (h)**

Separate the input features and target values for each of the train, valid, and test sets. In particular, we will use the following columns as features: 1, 2, 3, 4, 5 (but not the date column 0). We will predict the housing price, which is in column 6. We will refer to the five feature columns as x, and the housing price as t. We will need training, validation and testing versions of both x and t, for a total of 6 arrays. You should build these 6 arrays using the starter code below. Print the first 2 rows of each of these six new numpy arrays.

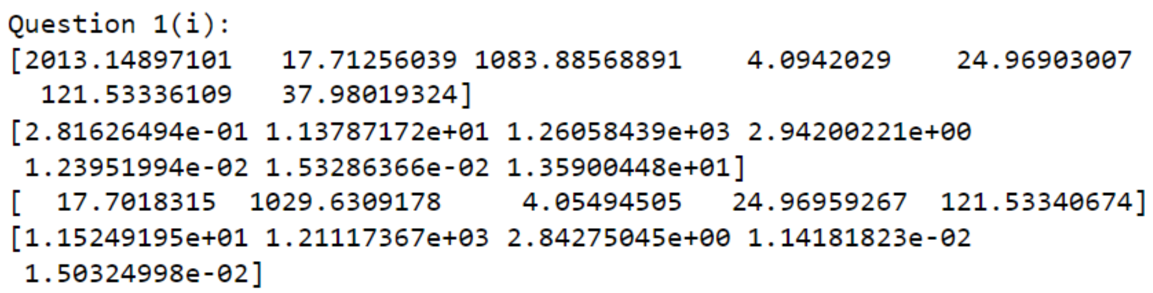
```
print('\nQuestion 1(h):')

train_x = train[0: ,1:6]
train_t = train[0: ,6]
valid_x = valid[0: ,1:6]
valid_t = valid[0: ,6]
test_x = test[0: ,1:6]
test_t = test[0: ,6]


#print('\nFirst 2 rows of train_x')
print(train_x[:2])

#print('\nFirst 2 rows of train_t')
print(train_t[:2])

#print('\nFirst 2 rows of valid_x')
print(valid_x[:2])

#print('\nFirst 2 rows of valid_t')
print(valid_t[:2])

#print('\nFirst 2 rows of test_x')
print(test_x[:2])

#print('\nFirst 2 rows of test_t')
print(test_t[:2])
```

```
Question 1(h):
[[ 32.        84.87882  10.         24.98298 121.54024]
 [  5.       390.5684    5.         24.97937 121.54245]]
[37.9 43.1]
[[ 19.5      306.5947    9.         24.98034 121.53951]
 [ 34.5      623.4731    7.         24.97933 121.53642]]
[42.2 40.3]
[[ 13.3      561.9845    5.         24.98746 121.54391]
 [ 13.3      561.9845    5.         24.98746 121.54391]]
[47.3 54.8]
```

## Part (i)

Compute the mean and standard deviation of each column in data. Then, compute the mean and standard deviation of each column in train_x, saving the results in x_mean and x_std. Print both sets of means and standard deviations. You may find the functions np.mean and np.std helpful. Find and read their documentations, and pay particular attention to the parameter axis.

```
print('\nQuestion 1(i):')

#print('\nMean of each column in data')
print(np.mean(data, axis=0))

#print('\nStandard deviation of each column in data')
print(np.std(data, axis=0))


x_mean = np.mean(train_x, axis=0)
x_std = np.std(train_x, axis=0)

#print('\nMean of each column in train_x')
print(x_mean)

#print('\nStandard deviation of each column in train_x')
print(x_std)
```



```
Question 1(i):
[2013.14897101   17.71256039 1083.88568891    4.0942029    24.96903007
  121.53336109   37.98019324]
[2.81626494e-01 1.13787172e+01 1.26058439e+03 2.94200221e+00
 1.23951994e-02 1.53286366e-02 1.35900448e+01]
[  17.7018315  1029.6309178     4.05494505   24.96959267  121.53340674]
[1.15249195e+01 1.21117367e+03 2.84275045e+00 1.14181823e-02
 1.50324998e-02]
```

**Part (j)**

For some of the models that we work with, we will be working with a normalized version of the features. In other words, we subtract the mean, and divide by the std, so that the features have zero mean and unit variance. Explain why using normalized data may be useful for some models, like the k-nearest neighbor model.

For the k-nearest neighbour model we need to calculate the distance between a data point and other data points. The distance calculated between the points could be affected by the scale of the features of a data point. Suppose a feature in the data set has high scale compared to other features. If we calculate the Euclidean distance, we will notice that the feature with high scale causes bias toward features with low scale in the distance which could impact the k-nearest neighbours. So, normalizing the data is useful to calculate the Euclidean distance to prevent bias towards certain features in the distance.

**Part (k)**

Explain why we should compute the mean and standard deviation using the training data, rather than across the entire labeled data (including the validation/test sets).

For the k-nearest neighbour model we calculate the Euclidean distance between a point we are trying to classify and other points in training set. To prevent bias toward features with high scale in the distance we need to normalize the data. Since we only classify with the training set, hence we need to only normalize the training set so we should only need to calculate the mean and standard deviation for the training data.

**Part (l)**

This part is meant to help you understand broadcasting, a method that numpy uses to perform vectorized computation. You will need to use broadcasting to complete part (m). Consider the following computation:

```
print('\nQuestion 1(l):')

tmp_a = np.array([[1.4, 2.5, 3.0], [9.1, 3.4, 2.3]])
tmp_b = np.sum(tmp_a, axis=0)
print(tmp_a)
print(tmp_b)
print(tmp_a - tmp_b)
```

```
Question 1(l):
[[1.4 2.5 3. ]
 [9.1 3.4 2.3]]
[10.5  5.9  5.3]
[[-9.1 -3.4 -2.3]
 [-1.4 -2.5 -3. ]]
```

Explain what computation was done to obtain the result.

tmp_a = np.array([[1.4, 2.5, 3.0], [9.1, 3.4, 2.3]]) creates a 2x3 array.
tmp_b = np.sum(tmp_a, axis=0) creates a new 1x3 array where the indices hold the respective sum of each column in tmp_a.
print(tmp_a - tmp_b) prints a 2x3 array where each row is the difference between the respective row in tmp_a and the single row in $tmp_b.

**Part (m)**

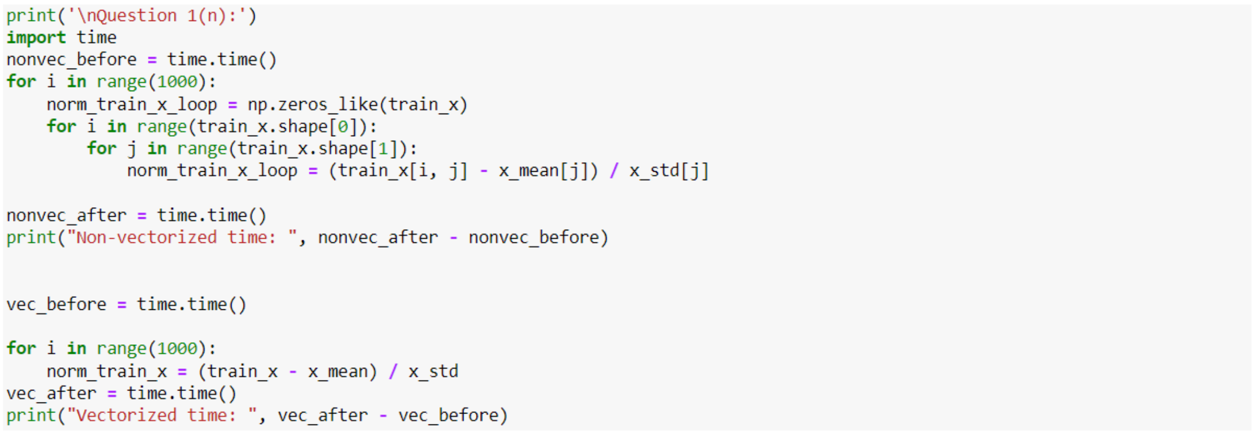
Using broadcasting (which you learned in the previous part), create the numpy array norm_train_x, which is the normalized version of the training data. Each column of this new matrix should have zero mean and unit variance. Print the mean of the columns of the new matrix. Print the first 2 rows of norm_train_x.

```
print('\nQuestion 1(m):')
norm_train_x = (train_x - x_mean) / x_std

#print('\nFirst 2 rows of norm_train_x')
print(norm_train_x[:2])

#print('\nMean and STD of the columns of norm_train_x')
print(np.mean(norm_train_x, axis=0))
#print(np.std(norm_train_x, axis=0))
```

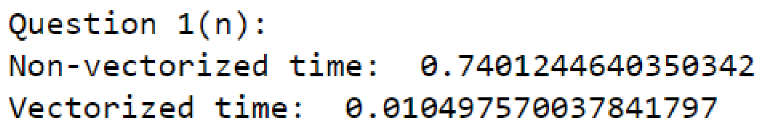
```
Question 1(m):
[[ 1.24063066 -0.78003025  2.09130384  1.17245685  0.45456578]
 [-1.10211889 -0.52763904  0.33244386  0.85629444  0.60158059]]
[ 1.08760172e-15 -1.28509332e-16  1.01668775e-16 -4.21720046e-13
 -4.74405822e-13]
```

**Part (n)**

Consider the computation below, which is an alternative way of computing norm_train_x that uses loops rather than vectorized code. How much slower is this code compared to your code in the previous part? Include your response in your writeup.



```python
print('\nQuestion 1(n):')
import time
nonvec_before = time.time()
for i in range(1000):
    norm_train_x_loop = np.zeros_like(train_x)
    for i in range(train_x.shape[0]):
        for j in range(train_x.shape[1]):
            norm_train_x_loop = (train_x[i, j] - x_mean[j]) / x_std[j]

nonvec_after = time.time()
print("Non-vectorized time: ", nonvec_after - nonvec_before)


vec_before = time.time()

for i in range(1000):
    norm_train_x = (train_x - x_mean) / x_std
vec_after = time.time()
print("Vectorized time: ", vec_after - vec_before)
```



```
Question 1(n):
Non-vectorized time:  0.7401244640350342
Vectorized time:  0.010497570037841797
```

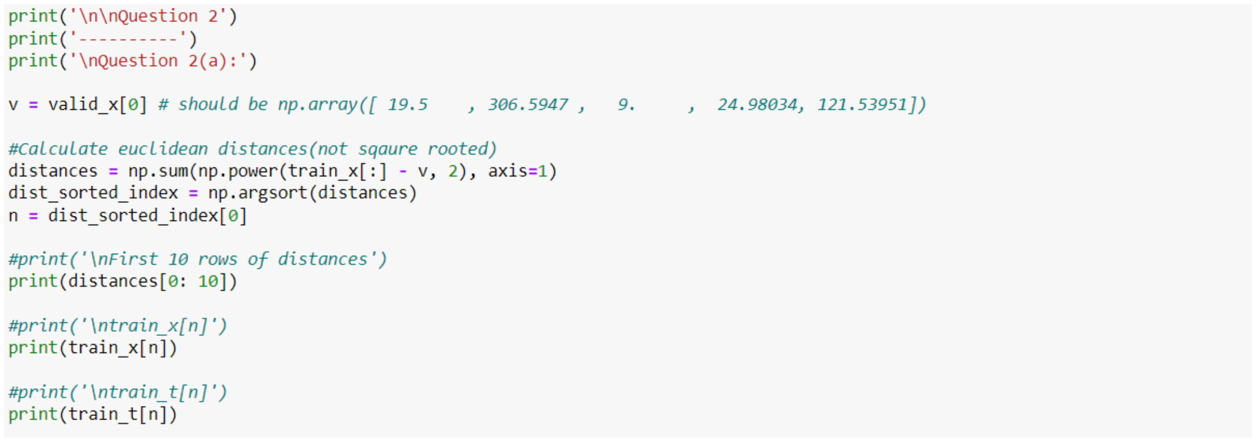
To compare the speed of normalizing the training data, I ran the vectorized and non-vectorized code 1000 times to get more accurate run time. The approximate time it took to normalize using vectorization and not using vectorization was 0.014 and 1.125, respectively. With these results we see that the non vectorized code took roughly 80 times more time to normalize the data than the vectorized code. The results also showcase roughly a 98 percent decrease in time to normalized data using vectorized method. This portrays that the vectorized code was significantly faster than the non-vectorized code.

# Nearest Neighbour for Regression

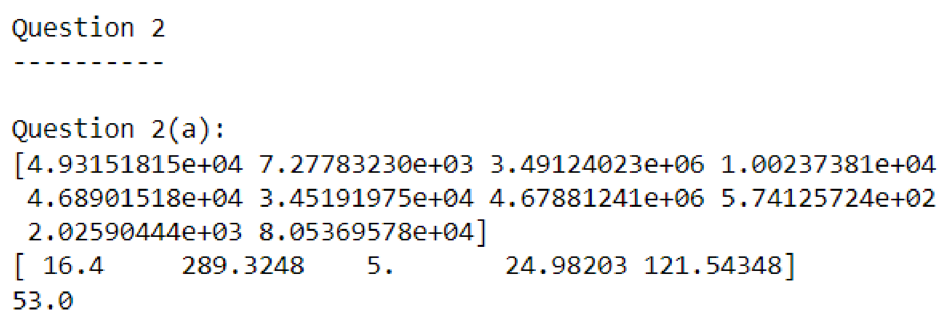
Here we will use nearest neighbours for regression. In particular, we will use the nearest neighbor method to predict the housing prices, given the other features. Instead of taking a majority vote of the discrete target in the nearest neighbours, we will take the average of the continuous target (the housing prices). We will explore using both the normalized and unnormalized features.

### Part (a)

First, let's consider using a 1-nearest neighbour approach to predict the house price of the first data point v in the validation set. We will use the unnormalized version of the dataset. Without using loops, compute the Euclidean distance between v and every data point in the training set train_x. Save the result in the numpy array distances. Print the first 10 rows of distances. Then, find the index n with the minimum value of distances. Print the row train_x[n], which is the closest data point to v, and the prediction train_t[n].



```
print('\n\nQuestion 2')
print('----------')
print('\nQuestion 2(a):')

v = valid_x[0] # should be np.array([ 19.5    , 306.5947 ,   9.      ,  24.98034, 121.53951])

#Calculate euclidean distances(not sqaure rooted)
distances = np.sum(np.power(train_x[:] - v, 2), axis=1)
dist_sorted_index = np.argsort(distances)
n = dist_sorted_index[0]

#print('\nFirst 10 rows of distances')
print(distances[0: 10])

#print('\ntrain_x[n]')
print(train_x[n])

#print('\ntrain_t[n]')
print(train_t[n])
```



```
Question 2
----------

Question 2(a):
[4.93151815e+04 7.27783230e+03 3.49124023e+06 1.00237381e+04
 4.68901518e+04 3.45191975e+04 4.67881241e+06 5.74125724e+02
 2.02590444e+03 8.05369578e+04]
[ 16.4      289.3248     5.        24.98203 121.54348]
53.0
```

### Part (b)

Now, let's consider using a 3-nearest neighbour model to make a prediction for the same v from part (a), again using unnormalized data. In other words, we find the 3 smallest elements of distances, and average their corresponding values in train_t to obtain a prediction. Print this prediction.

```
print('\nQuestion 2(b):')
#Compute 3NN for training data set with vector
three_nn = np.mean(train_t[dist_sorted_index[0:3]])
print(three_nn)
```

```
Question 2(b):
48.4
```

**Part (c)**

Complete the function unnorm_knn(v, k) that takes a feature vector v, and uses the k-nearest neighbour algorithm to make a prediction. Your code should be nearly identical to those from part (b), except k is now a parameter. Print the prediction for v=valid_x[1], with k=5.

```
print('\nQuestion 2(c):')

def unnorm_knn(v, k, features=train_x, labels=train_t):
    """
    Returns the k Nearest Neighbour prediction of housing prices for an input
    vector v.

    Parameters:
        v - The input vector to make predictions for
        k - The hyperparameter "k" in kNN
        features - The input features of the training data; a numpy array of shape [N, D]
                   (By default, `train_x` is used)
        labels - The target labels of the training data; a numpy array of shape [N]
                 (By default, `train_t` is used)
    """
    #Calculate euclidean distances
    distances = np.sum(np.power(v - features[:], 2), axis=1)
    #Sort distance indexes
    dist_sorted_index = np.argsort(distances)
    #Return KNN
    return np.mean(labels[dist_sorted_index[0:k]])


print(unnorm_knn(v=valid_x[1], k=5))
```

```
Question 2(c):
36.5
```

**Part (d)**

We wrote most of the function compute_mse for you below. This function takes a parameter predict, which is itself a function that makes a prediction given a feature vector. This function is intended to compute the mean squared error of the predictions made using the predict method, across the provided dataset (by default, the entire unnormalized validation set is supplied). Complete the Mean Square Error (MSE) computation. The Mean Square Error is another term for the average square loss across a data set. When you have done so, the code below will print the training and validation MSE for a (very simple) model that always predicts the average house price across the entire training set. We will call this a baseline model. Such a model is often used for sanity checking, and as a point of comparison. Your kNN model should be better than this baseline model.

```python
print('\nQuestion 2(d):')
def compute_mse(predict, data_x=valid_x, data_t=valid_t):
    """
    Returns the Mean Squared Error of a model across a dataset

    Parameters:
        predict - A Python *function* that takes an input vector and produces a
                  prediction for that vector.
        data_x - The input features of the data set to make predictions for
                 (By default, `valid_x` is used)
        data_t - The target labels of the dataset to make predictions for
                 (By default, `train_t` is used)
    """
    errors = []
    for i in range(data_t.shape[0]):
        #Compute difference between actual and predicted label
        error = (predict(data_x[i]) - data_t[i])**2
        errors.append(error)
    return np.mean(errors)

def baseline(v):
    """
    Returns the average housing price given an input vector v.
    """
    return np.mean(train_t)

# compute and print the training and validation MSE
print(compute_mse(baseline, data_x=train_x, data_t=train_t))
print(compute_mse(baseline, data_x=valid_x, data_t=valid_t))
```

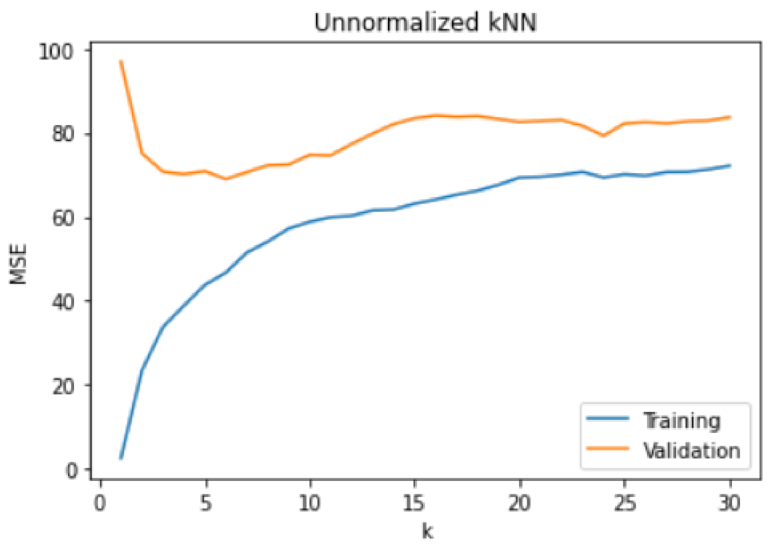
```
Question 2(d):
174.77563525607482
200.6520555038695
```

### Part (e)

For each choice of k (1, 2, up to 30), compute the MSE on the training and validation sets for the corresponding kNN model. Store these values in the two lists train_mse and valid_mse. Print these two lists. We include code below that plots the values in these two lists. Include the plot in your writeup. From the plot, what is the optimal value of k?

```python
print('\nQuestion 2(e):')

train_mse = []
valid_mse = []
for k in range(1, 31):
    # create a temporary function `predict_fn` that computes the knn
    # prediction for the current value of the loop variable `k`
    def predict_fn(new_v):
        return unnorm_knn(new_v, k)

    # compute the training and validation MSE for this kNN model
    mse = compute_mse(predict_fn, data_x=train_x, data_t=train_t)
    train_mse.append(mse)
    mse = compute_mse(predict_fn, data_x=valid_x, data_t=valid_t)
    valid_mse.append(mse)


from matplotlib import pyplot as plt
plt.plot(range(1, 31), train_mse)
plt.plot(range(1, 31), valid_mse)
plt.xlabel("k")
plt.ylabel("MSE")
plt.title("Unnormalized kNN")
plt.legend(["Training", "Validation"])
plt.show()

#print('\ntrain_mse')
print(train_mse)

#print('\nvalid_mse')
print(valid_mse)
```

```
Question 2(e):
```



```
[2.545677655677656, 23.402637362637364, 33.75391127391128, 38.81804258241759, 43.769038827838834, 46.693162393162396, 51.
464432981984, 54.095642742673995, 57.21185727852395, 58.80321098901099, 59.89691278418552, 60.23546245421245, 61.54160933
7408154, 61.737446736936526, 63.134434839234856, 64.11130394345237, 65.2640393170843, 66.2138081671415, 67.6325706980000
7, 69.35244203296705, 69.51671019079184, 70.0110742288015, 70.6711153811532, 69.35548000610501, 70.10270833699634, 69.764
35994971499, 70.63859956687118, 70.68841420909024, 71.27089793678377, 72.13229401709401]
[96.90380281690139, 75.1262323943662, 70.73532081377151, 70.16405809859154, 70.86134647887324, 68.98659624413145, 70.6134
1189997124, 72.24152288732395, 72.42198226395408, 74.74239014084505, 74.60658479804447, 77.32889964788733, 79.78338611550
961, 82.05927421672892, 83.43915492957747, 84.09591549295774, 83.79711584385204, 83.97218266388454, 83.24658577503804, 8
2.5664683098915, 82.811104404203, 83.0167332091724, 81.641431880508, 79.27247603677621, 82.21265239436615, 82.5237938578
2145, 82.23400142970301, 82.70106567979299, 82.88870677094673, 83.67681095461657]
```

We would like to generalize our model to data that has not been seen before. Hence, we must choose k from the validation set since the set has not been used to train the model. Also, we must choose a k which is not large or small and minimizes the mean squared error. Based on the graph above, the most optimal k would be approximately 6.

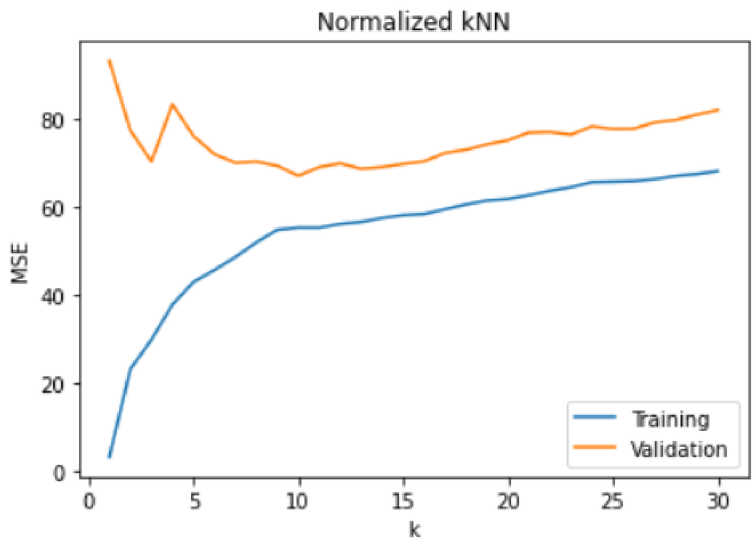
**Part (f)**

Let's consider the effect of normalization on a kNN model. Complete the function norm_knn below, that is identical to unnorm_knn, but uses normalized distances instead. In other words, you should use the matrix norm_train_x form Question 1 to compute distances. Be careful about what validation data you use. In particular, should you normalized the validation data? If so, how? (Hint: How would you normalize the test data? What if you only have a single test point?) Construct the same plot as in Part(e), but for norm_knn. Construct and print the lists train_mse and valid_mse.

```python
print('\nQuestion 2(f):')

def norm_knn(v, k, features=norm_train_x, means=x_mean, stds=x_std, labels=train_t):
    """
    Returns the k Nearest Neighbour prediction of housing prices for an input
    vector v.

    Parameters:
        v - The input vector to make predictions for
        k - The hyperparameter "k" in kNN
        features - The normalized input features of the training data
                   (By default, `norm_train_x` is used)
        means - The means over the training data (By default, `x_mean` is used)
        stds - The standard deviations of the training data (By default, `x_std` is used)
        labels - The target labels of the training data; a numpy array of shape [N]
                 (By default, `train_t` is used)
    """
    #normalize vector v in terms of features data
    v_norm = (v - means) / stds
    #Calculate distances
    distances = np.sum(np.power(features[:] - v_norm, 2), axis=1)
    #Sort distance indexes
    dist_sorted_index = np.argsort(distances)
    #return knn
    return np.mean(labels[dist_sorted_index[0:k]])
```

```python
train_mse = []
valid_mse = []
for k in range(1, 31):
    # create a temporary function `predict_fn` that computes the knn
    # prediction for the current value of the loop variable `k`
    def predict_fn(new_v):
        return norm_knn(new_v, k)

    # compute the training and validation MSE for this kNN model
    mse = compute_mse(predict_fn, data_x=train_x, data_t=train_t)
    train_mse.append(mse)
    mse = compute_mse(predict_fn, data_x=valid_x, data_t=valid_t)
    valid_mse.append(mse)

#print('\ntrain_mse_norm')
print(train_mse)

#print('\nvalid_mse_norm')
print(valid_mse)

#Plot MSE for normalized KNN
from matplotlib import pyplot as plt
plt.plot(range(1, 31), train_mse)
plt.plot(range(1, 31), valid_mse)
plt.xlabel("k")
plt.ylabel("MSE")
plt.title("Normalized kNN")
plt.legend(["Training", "Validation"])
plt.show()
```

```
Question 2(f):
[3.0816117216117225, 22.992408424908422, 29.71118437118437, 37.67307005494506, 42.79117509157509, 45.536583231583215, 48.
48223592733796, 51.83131066849816, 54.687386605164384, 55.19630256410255, 55.18164229709686, 56.01946377696378, 56.474021
93467284, 57.373671226732455, 58.022286039886026, 58.28442980196886, 59.379906713816744, 60.46701499118166, 61.3481005144
44006, 61.71432683150183, 62.57499921091757, 63.56482063391153, 64.39252151754985, 65.53403095746845, 65.65184797069597,
65.79180012788001, 66.22803810729737, 66.95946526687598, 67.38634056787446, 68.06599320309321]
[93.08704225352112, 77.31038732394366, 70.22596244131452, 83.25283450704225, 76.05908732394364, 71.91145148669798, 69.944
9123311296, 70.2127420774648, 69.26568422882978, 67.00365633802818, 68.94937725526714, 69.8208313771518, 68.5677756479706
6, 68.94750718597301, 69.68439999999998, 70.24938160211268, 72.1741020517569, 72.91705007824726, 74.13080059303186, 75.07
856654929576, 76.83044105905273, 76.9609597252939, 76.34524987353231, 78.20031274452268, 77.62949746478873, 77.6806252604
3838, 79.15300566085125, 79.70214752802531, 80.89930063137449, 81.92216338028169]
```



If you have normalized the training set data, then you should normalize the validation set data. To normalize the validation set, we should use the mean and standard deviation from the training set and normalize the individual points in the validation set since we use the validation set to tune hyperparameters for our model which is based on the training data. To make any decisions about our model and/or comparison of our model with another data set, the features should be on the same scale as the training set of our model. For the mean square error for validation set to tune hyperparameter k in the plot above, we needed to compute the prediction for the features of each data point in the validation set. We use the kNN method with a normalized training data set to predict for each data point in the validation set so the points must be normalized to the same scale as the normalized training set to find k smallest distances for reasons discussed earlier in questions one.

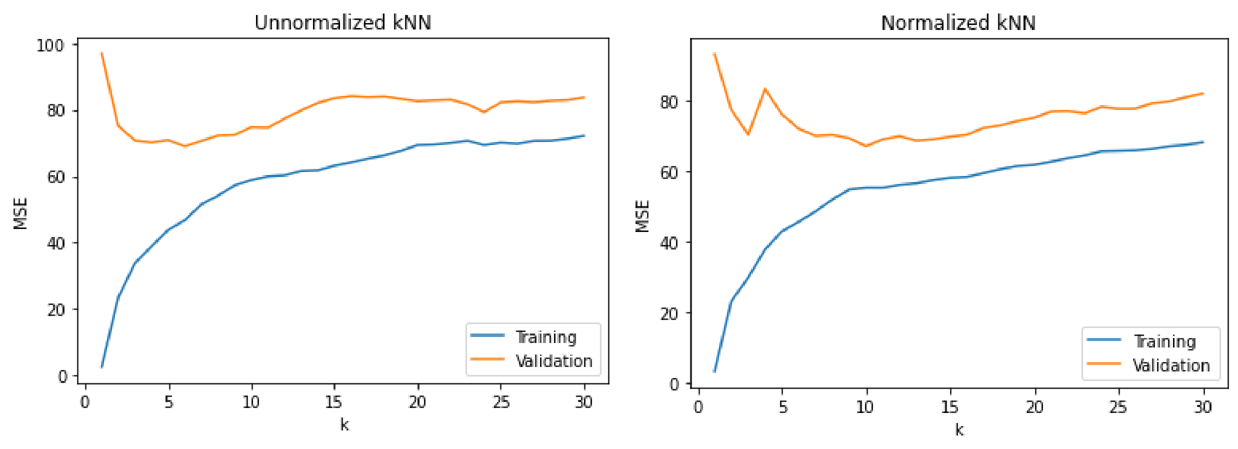
**Part (g)**

Is it true that the training MSE of a kNN model is always 0, for any k? If so, prove it. If not, construct a small counter example.

Training MSE of a kNN model is not always 0, for any k? For example, suppose we have the training set with the features x = [1, 2, 1] and their respective labels as t = [1, 2, 3]. Using the kNN model with k = 1, the MSE for the training set is the following:
MSE = $\frac{1}{n}\sum_{i=1}^{n}(y^{(x\_i)} - t^{(x\_i)})^2$ where t is the actual value and y is the prediction value for from kNN model with k = 1,
MSE = $(y^{(1)} - t^{(1)})^2 + (y^{(2)} - t^{(2)})^2 + (y^{(1)} - t^{(1)})^2$
MSE = $(1-1)^2 + (2-2)^2 + (1-3)^2$
MSE = 4
Notice that the train set has two different labels for the same feature. Hence, when the kNN model predicts the label for x = 1 with k = 1, it can choose any one of two labels as the closest point. In our example, the kNN model chooses to predict the same value for x = 1, which cause the training error to not be zero.

**Part (h)**

For this data set, does normalization improve the performance of kNN models? Justify your answer using the two plots from Part (f).



Based on the graphs above, normalization does not seem to improve the overall performance of the kNN model. For the validation set, normalization does not seem to reduce the minimum mean square error significantly but does seem to lower the mean square error for some values of k (example: k =15) and increases the mean square error for other values of k (example k = 4). For the training set, the kNN model seems to perform very similar for both unnormalized and normalized kNN models.

**Part (i)**

So far, we have discussed two different ways of computing distances for the kNN model: Euclidean distance over the normalized and unnormalized features. What other ways of computing distances do you expect would improve performance? In a few sentences, describe your proposal, and explain why you would expect performance to improve. You do not need to implement your idea.

The distance metric I propose is the following:

$$\left|\left|x^{(a)} - x^{(b)}\right|\right| = \sqrt{\sum_{i=1}^{n} |x_i^{(a)} - x_i^{(b)}|}$$

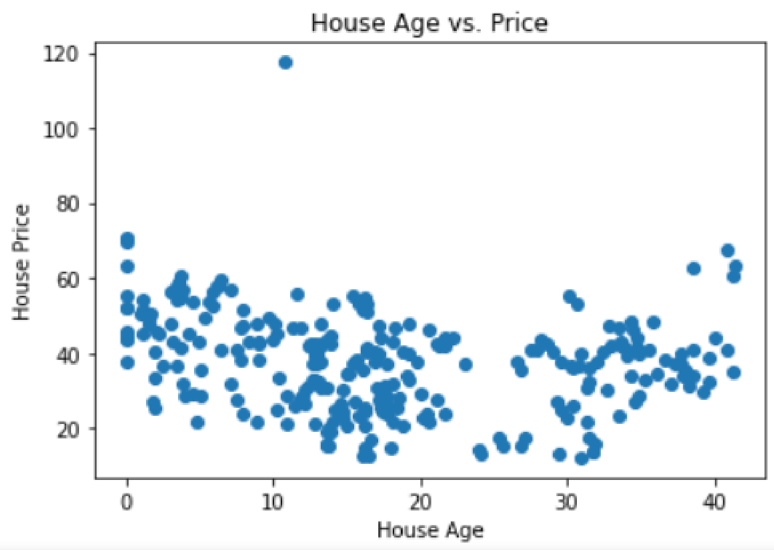
The distance metric above takes the sum of absolute difference for features in the two data points and then takes the overall square root. It is like the Euclidean distance except we are not squaring the differences, instead we are taking the absolute differences. I expect this would improve the performance of the model in the sense of producing better prediction. Earlier, we discussed how Euclidean distance could be impacted by features with large scales because we square the differences, so this led to bias toward features with low scale. With this distance metric, the high scale features do not impact the distance as much as in the Euclidean distances. This could lead more robust distances measures with out needing to normalize the data and hence more accurate prediction by the kNN model.

## Polynomial Regression

In this part, we will use polynomial regression to predict the housing prices. We will use only one of the features, namely the house's age.

```
print('\n\nQuestion 3')
print('----------')

train_d = train_x[:, 0] # house age

# Plot this feature against the target (house price)
plt.scatter(train_d, train_t)
plt.xlabel("House Age")
plt.ylabel("House Price")
plt.title("House Age vs. Price")
plt.show()
```

```
Question 3
----------
```



## Part (a)

As a warm-up, we will fit a straight line to the data (a polynomial of degree 1).

Your job is to estimate the coefficients $w = [a, b]$ of the model

$$y(x) = a + bx$$

where $x$ is the house's age and $y$ is the predicted house price. To do this, find the values of $a$ and $b$ that minimize the cost function

$$\mathcal{J}(a, b) = \frac{1}{2N} \sum_{n=1}^{N} (t^{(n)} - y(x^{(n)}))^2$$

where the sum is over all the training points $(x^{(n)}, t^{(n)})$. Recall that the values of $a$ and $b$ that minimize this loss are given by the following equation:

$$w = (X^T X)^{-1} X^T t$$

where $w = [a, b]$ is the weight vector, $t$ is a column vector of target values, and $X$ is the data matrix (design matrix). Here, $X$ should have two columns: the first column is all 1's, and the second column consists of the values in `train_d` (created for you).

You can construct the design matrix `X` by using functions like `np.ones`, `np.ones_like`, `np.stack` and `np.concatenate`, which you can look up in the numpy user guide. Do not use any loops. To avoid loops, you will also need to use matrix multiplication `@`, `np.linalg.inv`, and the transpose operation `X.T`, which you can also look up.

Store your weights (coefficients) $[a, b]$ in the variable `linear_coef`.

Print `linear_coef`.

```
print('\nQuestion 3(a):')

#Create design matrix for linear polynomal regression
ones = np.ones(train_d.shape[0])
X = np.stack((ones, train_d), axis=-1)

#Compute weights for linear polynomal regression
linear_coef = np.matmul(np.linalg.inv(np.matmul(X.T,X)), np.matmul(X.T,train_t[:]))
print(linear_coef)
```

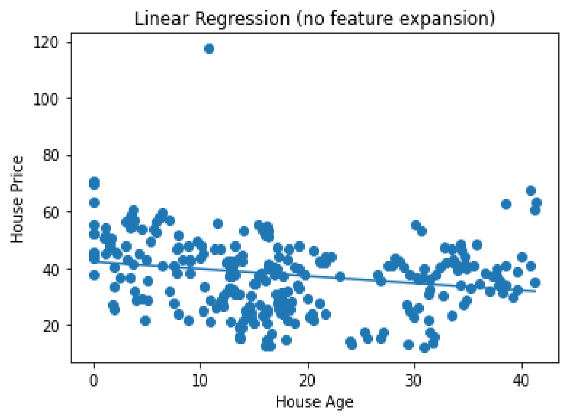
```
Question 3(a):
[42.25538722 -0.2532634 ]
```

## Part (b)

Write a function pred_linear that uses your weights from Part (c) and makes predictions for a numpy vector v consisting of the ages of several houses that we would like to make predictions for. Note that unlike the prediction functions for KNN that you wrote in Question 2, which returned a single prediction for a single input point, the function predict_linear here returns a vector of many predictions given a vector of many input points. Your code for predict_linear should be completely vectorized and should not use any loops. The function plot_prediction visualizes the house price predictions of a model. It is written for you. Uncomment the call to plot_prediction(...) to visualize the resulting linear model function. Include this plot in your write up.

```python
def plot_prediction(predict, title):
    """
    Display a plot that superimposes the model predictions on a scatter
    plot of the training data (train_d, train_t)

    Parameters:
        predict - A Python *function* that takes an input vector and produces a
                  prediction for that vector.
        title - A title to display on the figure.
    """
    # start with a scatter plot
    plt.scatter(train_d, train_t)

    # create several "house age" values to make predictions for
    min_age = np.min(train_d)
    max_age = np.max(train_d)
    v = np.arange(min_age, max_age, 0.1)

    # make predictions for those values
    y = predict(v)

    # plot the result
    plt.plot(v, y)
    plt.title(title)
    plt.xlabel("House Age")
    plt.ylabel("House Price")
    plt.show()

#Create plot for linear polynomial regression
plot_prediction(pred_linear, title="Linear Regression (no feature expansion)")
```

```python
def pred_linear(v, coef=linear_coef):
    """
    Returns the linear regression predictions of house prices, given
    a vector consisting of the ages of several houses that we would
    like to make predictions for.

    Parameters:
        v - A vector of house ages
        coef - The linear regression coefficient
    """
    return (coef[1]*v) + coef[0]
```



## Part (c)

Complete the function compute_mse_vectorized. This function takes a parameter predict, which is itself a function that makes a prediction given a vector of house ages.

```
print('\nQuestion 3(c):')

def compute_mse_vectorized(predict, data_x=valid_x, data_t=valid_t):
    """
    Returns the Mean Squared Error of a model across a dataset

    Parameters:
        predict - A Python *function* that takes a vector of house ages,
                  and produces a vector of house price predictions
        data_x - The input features of the data set to make predictions for
                 (By default, `valid_x` is used)
        data_t - The target labels of the dataset to make predictions for
                 (By default, `valid_t` is used)
    """
    v = data_x[:, 0] # house age
    mse = np.square(predict(v) - data_t[:]).mean()
    return mse
```

## Part (d)

In the rest of this question, you will fit models of increasing complexity to the data. Here, for instance, you will fit a quadratic model (a polynomial of degree 2). That is, you should estimate the coefficients [a, b, c] to the model

$$y(x) = a + bx + cx^2$$

where, like in part (a), x is the age of the house and y is its predicted price. Find the coefficients that minimize the same cost function as in Part (a). Once again, do not use any loops. Store your weights (coefficients) [a, b, c] in the variable quad_coef.

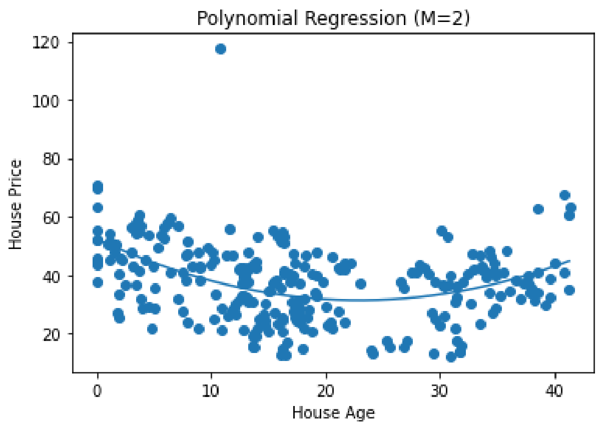
```
print('\nQuestion 3(d):')
#Create design matrix for quadratic polynomial regression
ones = np.ones(train_d.shape[0])
train_d_sqr = np.square(train_d)
X = np.stack((ones, train_d, train_d_sqr), axis=-1)

#Compute weights for quadratic polynomial regression
quad_coef = np.matmul(np.linalg.inv(np.matmul(X.T,X)), np.matmul(X.T,train_t[:]))

print(quad_coef)
```

```
Question 3(d):
[ 5.24237889e+01 -1.83379775e+00  3.99166404e-02]
```

## Part (e)

Write a function pred_quad that uses your model from part (d) and makes predictions for a numpy vector v consisting of the ages of houses that we would like to make predictions for. As in part (b), do not use loops. Uncomment the call to plot_prediction(...) to visualize the predictions of the model from part (d). Include this plot in your write up.

```
def pred_quad(v, coef=quad_coef):
    """
    Returns the degree 2 polynomial regression predictions of
    house prices, given a vector consisting of the ages of several houses
    that we would like to make predictions for.

    Parameters:
        v - A vector of house ages
        coef - The linear regression coefficient
    """

    v_sqr = np.square(v)
    return (coef[2]*v_sqr) + (coef[1]*v) + coef[0]

#Create plot for quadratic polynomial regression
plot_prediction(pred_quad, title="Polynomial Regression (M=2)")
```



## Part (f)

Use the function compute_mse_vectorized and the pred_quad function to compute the training and validation MSE of your model in Part (d).

```
print('\nQuestion 3(f):')

#Compute training set MSE for quadratic polynomial regression
mse_train = compute_mse_vectorized(pred_quad, data_x=train_x, data_t=train_t)
#print('\nTraining MSE')
print(mse_train)

#Compute validation set MSE for quadratic polynomial regression
mse_valid = compute_mse_vectorized(pred_quad)
#print('\nValidation MSE')
print(mse_valid)
```

```
Question 3(f):
139.0357747767251
153.4632471105083
```

**Part (g)**

Compute the training and validation MSEs for polynomial regression, for polynomial degrees 0, 1, 2, . . . up to M=10. In other words, for each of the polynomial degrees (0 to 10), compute the optimal, least-squares coefficients directly like you did in parts (a) and (d). Then, compute the training and validation MSE using the compute_mse_vectorized like in parts (b-c) and (e-f). Store these MSE values in the two lists train_mse_r and valid_mse_r. Print these two lists. Each list should have length 11.

```
print('\nQuestion 3(g):')


def vector_multinomial(v, M):
    """
    Return design matrix for polynomial regression with M dimension

    Parameters
    ----------
    v : input vector(N)
    M : dimension size

    Returns
    -------
    X : design matrix(N,M+1)

    """
    X = np.ones((v.shape[0],1))

    for i in range(0,M):
        pwr = np.power(v, i + 1)
        pwr.shape = (v.shape[0],1)
        X = np.concatenate((X, pwr), axis=1)

    return X


train_mse_r = []
valid_mse_r = []


for M in range(0, 11):

    #Compute design matrix X
    X = vector_multinomial(train_d, M)
    #Compute weights for polynomial regression function
    coef = np.matmul(np.linalg.inv(np.matmul(X.T,X)), np.matmul(X.T,train_t[:]))

    #Compute prediction given inputs v and coeffcients c
    def predict(v, c= coef):
        v = vector_multinomial(v, M)

        pred = np.dot(v, c.T)
        return pred

    #Compute validation set MSE
    mse_valid = compute_mse_vectorized(predict)
    #Compute training set MSE
    mse_train = compute_mse_vectorized(predict, data_x=train_x, data_t=train_t)

    train_mse_r.append(mse_train)
    valid_mse_r.append(mse_valid)


#print('\ntrain_msr_r')
print(train_mse_r)
#print('\nvalid_msr_r')
print(valid_mse_r)
```

```
Question 3(g):
[174.77563525607482, 166.25600652655075, 139.0357747767251, 137.92733345645456, 137.9183442488486, 137.91306291158125, 13
7.17426920413578, 137.06614129883937, 133.54686060299068, 133.480832440462, 133.4547753316347]
[200.6520555038694, 191.3222052423672, 153.4632471105083, 157.89940575013276, 158.5852433378963, 159.25025813604552, 175.
260343065075, 164.50045733403945, 349.8361201781444, 425.4498195276298, 338.217151425373]
```

## Part (h)

Explain why train_mse_r[0] and valid_mse_r[0] should be identical to your result from Question 2 (d).

The train_mse_r[0] and valid_mse_r[0] should be identical to the result from Question 2 (d) because the predicted value for Polynomial Feature Mapping with M= 0 and the predicted value for 2(d) are the same such that for each data point in the training and validation data the predicted value is the mean of the labels in the training data. Recall, the baseline prediction function returns the mean of the training data labels and similarly, the coefficient for the regression model y = b is also the mean of the training data labels. The following will prove the coefficient for the regression model y = b is also the mean of the training data labels.

For polynomial feature mapping, we use the formula w = $(X^T X)^{-1} X^T t$ to calculate the polynomial coefficients where X is the design matrix and t = $[y_1, \dots \dots, y_N]^T$ is a vector of training data labels. For M=0, the design matrix is
X = $[1_1, \dots \dots, 1_N]^T$ where N is the number of data point in the training data. The coefficient is calculated using the training data and hence is the following:

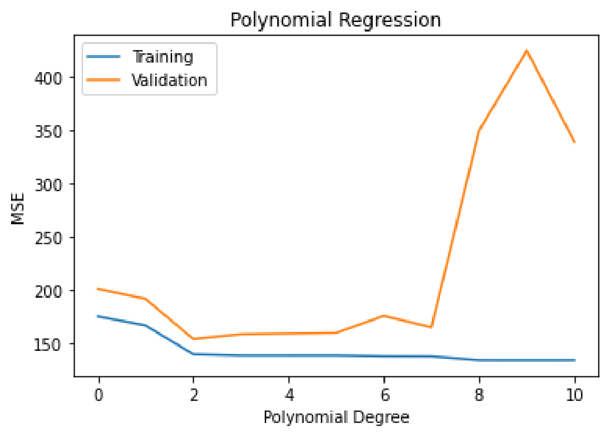
$$
\begin{aligned}
w &= (X^T X)^{-1} X^T t \\
&= [N]^{-1}[y_1 + \cdots + y_N] \\
&= [\tfrac{1}{N}][y_1 + \cdots + y_N] \\
&= [\tfrac{y_1 + \cdots + y_N}{N}]
\end{aligned}
$$

So, the coefficient for the regression model y = b is also the mean of the training data labels. The prediction function returns a vector of predicted values which is the mean of the labels in the training data in this case. This leads to the Polynomial Feature Mapping with M= 0 to having the same mean square error for the validation and training data as in Question 2 (d).

## Part (i)

Use the code below to plot the training and validation MSEs that you computed in part (h). Describe the shape of this graph, and briefly explain why we see this shape.

```python
plt.plot(range(0, 11), train_mse_r)
plt.plot(range(0, 11), valid_mse_r)
plt.xlabel("Polynomial Degree")
plt.ylabel("MSE")
plt.title("Polynomial Regression")
plt.legend(["Training", "Validation"])
plt.show()
```

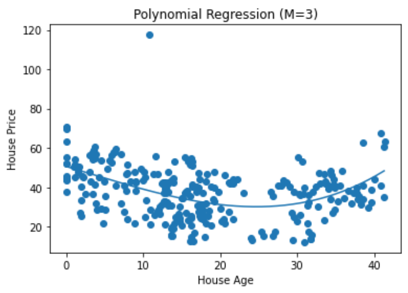


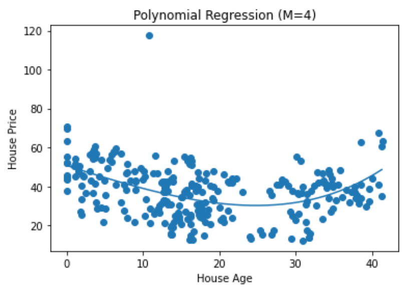
To begin with, the MSE of the training set starts of high at approximately 175 at M = 0 and then starts to decrease somewhat linearly and eventually reaches approximately 130 at M=10. The MSE of the training set starts of high because the regression model underfits that data at M = 0 and as M increases, the complexity of the model increases such that the fit of the model increases, which causes the MSE to decrease. Next, the MSE of the validation set is higher than the MSE of the training set for all M, as expected. The MSE of the validation set starts of at approximately 200 at M = 0 and then linearly decrease from M = 0 to M = 2 because the validation set starts to increasingly fit the model created using the training set, so the MSE starts to decrease. From M = 2 to M = 7, the MSE for validation set slightly balances out because the model fits the validation set similarly in this range. From M = 7 to M=10 the MSE spikes up as M increases because the model seems to underfit the validation data significantly, causing the MSE for the validation set to increase significantly.

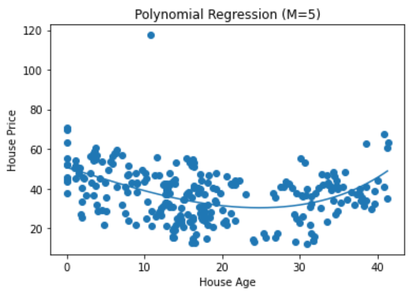
**Part (j)**

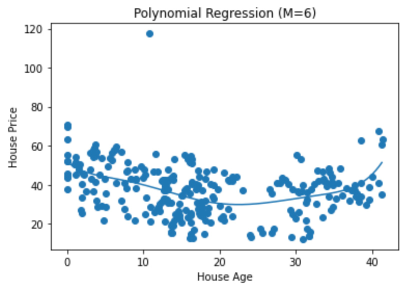
Use the function plot_prediction to plot the model predictions for polynomial degrees 3-10. These plots are analogous to those in Part (b) and Part (e), and should have appropriate titles. Which model shows an example of overfitting? Which model shows an example of underfitting? Justify your reasoning.

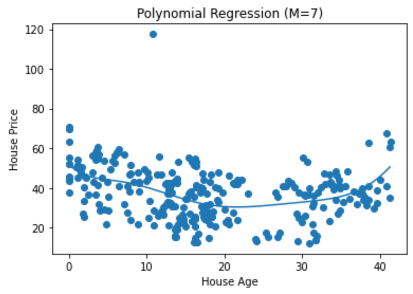
```
for M in range(3, 11):

    #Compute design matrix X
    X = vector_multinomial(train_d, M)
    #Compute weights for polynomial regression function
    coef = np.matmul(np.linalg.inv(np.matmul(X.T,X)), np.matmul(X.T,train_t[:]))

    #Compute prediction given inputs v and coefficents c
    def predict(v, c= coef):
        v = vector_multinomial(v, M)

        pred = np.dot(v, c.T)
        return pred

    #Create plot of polynomial regression with M
    plot_prediction(predict, title="Polynomial Regression (M=%d)" % M)
```

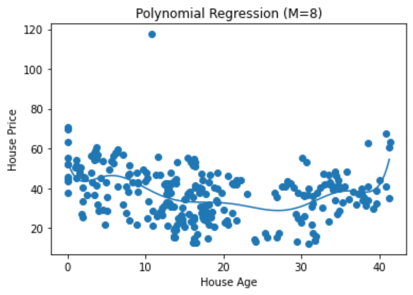


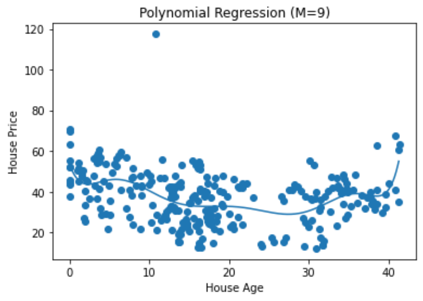


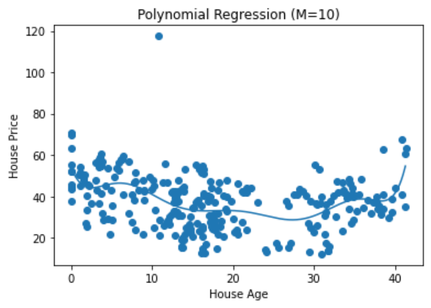












Given the set of plots for M = 0 to M=10, an example of a model underfitting training data is the polynomial regression model with M = 0. This model very simple because it does not follow the data at all such that it is just represents the mean of labels in training data set. From the graph from the previous question, we see the mean square error for the training set with this model is high amongst other models, which also justifies the underfit with the training data. An example of overfit from this set of models is the polynomial regression model with M = 10. This model is very complex because it follows the training data set too closely. From the graph from the previous question, we see the mean square error for the training set with this model is low amongst other models and the mean square error for the validation set with this model, which also justifies the overfit with the training data.

**Part (k)**

Will normalization affect the linear regression validation MSE when M = 1? If so, provide an example where the validation MSE differs. If not, provide a proof.

Normalization does not affect the linear regression validation MSE when M = 1. The MSE is calculated as $MSE = \frac{1}{N}1\sum_{i=1}^{N}[t^i - y(x^i)]^2$ and the reason it does not change is because the predictor function $y(x^i) = a + bx$ gives the same prediction for normalized and unnormalized validation set. When we normalize the data, we use the mean and standard deviation for the training set to normalize the training set and validation set. Hence, by normalization we are just normalizing data by applying a linear transformation to the data set. The following will prove that the predictor function for unnormalized and normalized data will predict the same values for unnormalized and normalized data, respectively:

Let define the following vectors
Training set feature vector**:** $\boldsymbol{train_x} = [\boldsymbol{x_1} \dots \boldsymbol{x_N}]$
Validation set feature vector: $\boldsymbol{valid_x} = [\boldsymbol{v_1} \dots \boldsymbol{v_M}]$
Training set label vector: $\boldsymbol{train_t} = [\boldsymbol{t_1} \dots \boldsymbol{t_N}]$
Validation set label vector: $\boldsymbol{valid_t} = [\boldsymbol{p_1} \dots \boldsymbol{p_M}]$

Note the least squares estimator coefficients for linear equation $y(x^i) = a + bx$:

$$[\hat{a}, \hat{b}] = (X^T X)^{-1} X^T t$$

Where:

$$t = train_t = [t_1 \dots t_N]^T$$

$$X = \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix}$$

$$\hat{a} = \bar{t} - \hat{b}\bar{x} \qquad\qquad \hat{b} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(t_i - \bar{t})}{\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

If we are to normalize the training and validation data sets features with mean ($\bar{x}$) and standard deviation (sd) of training set, the mean of normalized training set will be 0 ($\bar{x}_{norm} = 0$) and the standard deviation of the training set will be 1. Hence the new estimators for coefficients of linear equation $y(x^i) = a + bx$ using the normalized data will be the following:

$$\widehat{b^*} = \frac{\sum_{i=1}^{N}(x_i - \bar{x}_{norm})(t_i - \bar{t})}{\sum_{i=1}^{N}(x_i - \bar{x}_{norm})^2}$$

$$\widehat{b^*} = \frac{\sum_{i=1}^{N}(\left(\frac{x_i - \bar{x}}{sd}\right) - \bar{x}_{norm})(t_i - \bar{t})}{\sum_{i=1}^{N}\left(\left(\frac{x_i - \bar{x}}{sd}\right) - \bar{x}_{norm}\right)^2}$$

$$\widehat{b^*} = \frac{\sum_{i=1}^{N}(\left(\frac{x_i - \bar{x}}{sd}\right) - 0)(t_i - \bar{t})}{\sum_{i=1}^{N}\left(\left(\frac{x_i - \bar{x}}{sd}\right) - 0\right)^2}$$

$$\widehat{b^*} = \frac{\sum_{i=1}^{N}\left(\frac{x_i - \bar{x}}{sd}\right)(t_i - \bar{t})}{\sum_{i=1}^{N}\left(\frac{x_i - \bar{x}}{sd}\right)^2}$$

$$\widehat{b^*} = \frac{\frac{1}{sd}\sum_{i=1}^{N}(x_i - \bar{x})(t_i - \bar{t})}{\frac{1}{sd^2}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

$$\widehat{b^*} = \frac{sd\sum_{i=1}^{N}(x_i - \bar{x})(t_i - \bar{t})}{\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

$$\widehat{b^*} = sd\hat{b}$$

$$\widehat{a^*} = \bar{t} - \widehat{b^*}\bar{x}_{norm}$$

$$\widehat{a^*} = \bar{t} - \widehat{b^*}(0)$$

$$\widehat{a^*} = \bar{t}$$

Hence the predicted linear equation $y(x^i) = a + bx$ for the unnormalized and normalized validation data are the following:

$$y_{unorm}(v_i) = \hat{a} + \hat{b}v_i$$

$$y_{norm}\left(\frac{v_i - \bar{x}}{sd}\right) = \widehat{a^*} + \widehat{b^*}(\frac{v_i - \bar{x}}{sd})$$

$$y_{norm}\left(\frac{v_i - \bar{x}}{sd}\right) = \bar{t} + sd\hat{b}(\frac{v_i - \bar{x}}{sd})$$

$$y_{norm}\left(\frac{v_i - \bar{x}}{sd}\right) = \bar{t} + \hat{b}(v_i - \bar{x})$$

$$y_{norm}\left(\frac{v_i - \bar{x}}{sd}\right) = (\bar{t} - \hat{b}\bar{x}) + \hat{b}v_i$$

$$y_{norm}\left(\frac{v_i - \bar{x}}{sd}\right) = \hat{a} + \hat{b}v_i$$

Since the predicted value for unnormalized and normalized validation will be the same such that $y_{unorm}(v_i) = y_{norm}\left(\frac{v_i - \bar{x}}{sd}\right)$. Hence, $MSE = \frac{1}{N}1\sum_{i=1}^{N}[p^i - y_{unorm}(v^i)]^2$ = $\frac{1}{N}1\sum_{i=1}^{N}[p^i - y_{norm}\left(\frac{v_i - \bar{x}}{sd}\right)]^2$. Therefore, Normalization does not affect the linear regression validation MSE when M = 1.

**Part (l)**

Will normalization affect the linear regression validation MSE when M > 1? If so, provide an example where the validation MSE differs. If not, provide a proof.

Normalization does not affect the linear regression validation MSE when M >1. Similar to linear regression with M=1, the MSE does not change because the predictor function for M > 1, $y(x^i) = b_0 + b_1 x + b_2 x^2 + \cdots + b_M x^M$ gives the same prediction for normalized and unnormalized validation set. When we normalize the data, we use the mean and standard deviation for the training set to normalize the training set and validation set. Hence, by normalization we are just normalizing data by applying a linear transformation to the data set. So, the coefficient for the predictor function for normalized data are coefficients of unnormalized predictor function scaled by some factor. The scaled factors cancel out once we input the normalized validation set data to calculated predicted value, hence giving the same prediction for normalized and unnormalized validation set. Therefore, normalization does not affect the linear regression validation MSE when M >1.

# Gradient Descent

In this part, we will solve the polynomial regression problem by optimizing the square error cost function via gradient descent.

**Part (a)**

Derive, by hand, an equation for the derivative $\frac{\partial \mathcal{J}}{\partial w_0}$.

$$\frac{\partial \tau}{\partial w_0} = \frac{\partial \tau}{\partial y}\frac{\partial y}{\partial w_0}$$

$$\frac{\partial \tau}{\partial w_0} = \frac{1}{2N}\sum_{i=1}^{N}\frac{\partial[(t^{(x_i)} - y^{(x_i)})^2]}{\partial y}\frac{\partial[w_0 + w_1 x_i + \cdots + w_M x_i^M]}{\partial w_0}$$

$$\frac{\partial \tau}{\partial w_0} = \frac{1}{2N}\sum_{i=1}^{N}\frac{\partial[(t^{(x_i)} - y^{(x_i)})^2]}{\partial y}\left[\frac{\partial}{\partial w_0}w_0 + \frac{\partial}{\partial w_0}w_1 x_i + \cdots + \frac{\partial}{\partial w_0}w_M x_i^M\right]$$

$$\frac{\partial \tau}{\partial w_0} = \frac{1}{2N}\sum_{i=1}^{N} -2\left(t^{(x_i)} - y^{(x_i)}\right)[1 + 0 + \cdots + 0]$$

$$\frac{\partial \tau}{\partial w_0} = \frac{2}{2N}\sum_{i=1}^{N} -\left(t^{(x_i)} - y^{(x_i)}\right)[1]$$

$$\frac{\partial \tau}{\partial w_0} = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(x_i)} - t^{(x_i)}\right)$$

**Part (b)**

Derive, by hand, an equation for the derivative $\frac{\partial \mathcal{J}}{\partial w_1}$ .

$$\frac{\partial \tau}{\partial w_1} = \frac{\partial \tau}{\partial y}\frac{\partial y}{\partial w_1}$$

$$\frac{\partial \tau}{\partial w_1} = \frac{1}{2N}\sum_{i=1}^{N}\frac{\partial[(t^{(x_i)} - y^{(x_i)})^2]}{\partial y}\frac{\partial[w_0 + w_1 x_i + \cdots + w_M x_i^M]}{\partial w_1}$$

$$\frac{\partial \tau}{\partial w_1} = \frac{1}{2N}\sum_{i=1}^{N}\frac{\partial[(t^{(x_i)} - y^{(x_i)})^2]}{\partial y}\left[\frac{\partial}{\partial w_1}w_0 + \frac{\partial}{\partial w_1}w_1 x_i + \cdots + \frac{\partial}{\partial w_1}w_M x_i^M\right]$$

$$\frac{\partial \tau}{\partial w_1} = \frac{1}{2N}\sum_{i=1}^{N} -2\left(t^{(x_i)} - y^{(x_i)}\right)[0 + \ x_i + \cdots + 0]$$
$$\frac{\partial \tau}{\partial w_1} = \frac{2}{2N}\sum_{i=1}^{N} -\left(t^{(x_i)} - y^{(x_i)}\right)[x_i]$$
$$\frac{\partial \tau}{\partial w_1} = \frac{1}{N}\sum_{i=1}^{N} \left(y^{(x_i)} - t^{(x_i)}\right) x_i$$

**Part (c)**

More generally, derive, by hand, the derivative $\frac{\partial \mathcal{J}}{\partial w_j}$.

$$\frac{\partial \tau}{\partial w_j} = \frac{\partial \tau}{\partial y}\frac{\partial y}{\partial w_j}$$

$$\frac{\partial \tau}{\partial w_j} = \frac{1}{2N}\sum_{i=1}^{N} \frac{\partial[(t^{(x_i)} - y^{(x_i)})^2]}{\partial y}\frac{\partial[w_0 + w_1 x_i + \cdots + w_M x_i^M]}{\partial w_j}$$

$$\frac{\partial \tau}{\partial w_j} = \frac{1}{2N}\sum_{i=1}^{N} \frac{\partial[(t^{(x_i)} - y^{(x_i)})^2]}{\partial y}\left[\frac{\partial}{\partial w_j} w_0 + \frac{\partial}{\partial w_j} w_1 x_i + \cdots + \frac{\partial}{\partial w_j} w_M x_i^M\right]$$
$$\frac{\partial \tau}{\partial w_j} = \frac{1}{2N}\sum_{i=1}^{N} -2\left(t^{(x_i)} - y^{(x_i)}\right)\left[0 + \ \ldots + x_i^j + \cdots + 0\right]$$
$$\frac{\partial \tau}{\partial w_j} = \frac{2}{2N}\sum_{i=1}^{N} -\left(t^{(x_i)} - y^{(x_i)}\right)\left[x_i^j\right]$$
$$\frac{\partial \tau}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N} \left(y^{(x_i)} - t^{(x_i)}\right) x_i^j$$

**Part (d)**

Using your result from Part (c), show that the gradient vector $\nabla\tau(w) = \left[\frac{\partial \tau}{\partial w_0} \quad \frac{\partial \tau}{\partial w_1} \quad \ldots \quad \frac{\partial \tau}{\partial w_M}\right]^T$ has $\nabla\tau(w) = \frac{1}{N}[\boldsymbol{Y} - \boldsymbol{t}]^T\boldsymbol{X}$ , where $\boldsymbol{X}$ is the design matrix.

$$\nabla\tau(w) = \begin{bmatrix} \dfrac{\partial\tau}{\partial w_0} \\ \dfrac{\partial\tau}{\partial w_1} \\ \vdots \\ \dfrac{\partial\tau}{\partial w_M} \end{bmatrix}$$

$$\nabla\tau(w) = \begin{bmatrix} \dfrac{1}{N}\sum_{i=1}^{N}\left(y^{(x_i)} - t^{(x_i)}\right) \\ \dfrac{1}{N}\sum_{i=1}^{N}\left(y^{(x_i)} - t^{(x_i)}\right)x_i \\ \vdots \\ \dfrac{1}{N}\sum_{i=1}^{N}\left(y^{(x_i)} - t^{(x_i)}\right)x_i^M \end{bmatrix}$$

$$\nabla\tau(w) = \frac{1}{N}\begin{bmatrix} \sum_{i=1}^{N}\left(y^{(x_i)} - t^{(x_i)}\right) \\ \sum_{i=1}^{N}\left(y^{(x_i)} - t^{(x_i)}\right)x_i \\ \vdots \\ \sum_{i=1}^{N}\left(y^{(x_i)} - t^{(x_i)}\right)x_i^M \end{bmatrix}$$

$$\nabla\tau(w) = \frac{1}{N}\begin{bmatrix} \left(y^{(1)} - t^{(1)}\right) + \left(y^{(2)} - t^{(2)}\right) + \cdots + \left(y^{(N)} - t^{(N)}\right) \\ \left(y^{(1)} - t^{(1)}\right)x_1^1 + \left(y^{(2)} - t^{(2)}\right)x_2^1 + \cdots + \left(y^{(N)} - t^{(N)}\right)x_N^1 \\ \vdots \\ \left(y^{(1)} - t^{(1)}\right)x_1^M + \left(y^{(2)} - t^{(2)}\right)x_2^M + \cdots + \left(y^{(N)} - t^{(N)}\right)x_N^M \end{bmatrix}$$

$$\nabla\tau(w) = \frac{1}{N}\left[\left(y^{(1)} - t^{(1)}\right) \quad \left(y^{(2)} - t^{(2)}\right) \quad . \; . \; . \quad \left(y^{(N)} - t^{(N)}\right)\right]\begin{bmatrix} 1 & x_1 & \cdots & x_1^M \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & \cdots & x_N^M \end{bmatrix}$$

$$\nabla\tau(w) = \frac{1}{N}\begin{bmatrix} \left(y^{(1)} - t^{(1)}\right) \\ \left(y^{(2)} - t^{(2)}\right) \\ \vdots \\ \left(y^{(N)} - t^{(N)}\right) \end{bmatrix}^T \begin{bmatrix} 1 & x_1 & \cdots & x_1^M \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & \cdots & x_N^M \end{bmatrix}$$

$$\nabla\tau(w) = \frac{1}{N}\left[\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} - \begin{bmatrix} t^{(1)} \\ t^{(2)} \\ \vdots \\ t^{(N)} \end{bmatrix}\right]^T \begin{bmatrix} 1 & x_1 & \cdots & x_1^M \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & \cdots & x_N^M \end{bmatrix}$$

$$\nabla\tau(w) = \frac{1}{N}[\boldsymbol{Y} - \boldsymbol{t}]^T \boldsymbol{X}$$

## Part (e)

Complete the function grad that takes the a weight vector weight, the training data X and t, and computes the gradient ∇J (w) at that weight.

```python
print('\n\nQuestion 4')
print('----------')
print('\nQuestion 4(e):')

def grad(weight, X, t):
    '''
    Return gradient of each weight evaluated at the current value

    Parameters:
        `weight` - a current "guess" of what our weights should be,
                   a numpy array of shape (D)
        `X` - matrix of shape (N,D) of input features
        `t` - target y values of shape (N)

    '''
    #Number of data points in data set
    n = X.shape[0]

    #Compute vector of prediction Y
    predict_y = np.matmul(X, weight.T)

    #Compute error vector Y-t
    error = predict_y - t

    #Return gradient vector
    return np.matmul(error, X) / n


# Please leave this print statement for grading:
print(grad(np.array([1]), np.array([[1], [1]]), np.array([2, 2])))
```

```
Question 4
----------

Question 4(e):
[-1.]
```

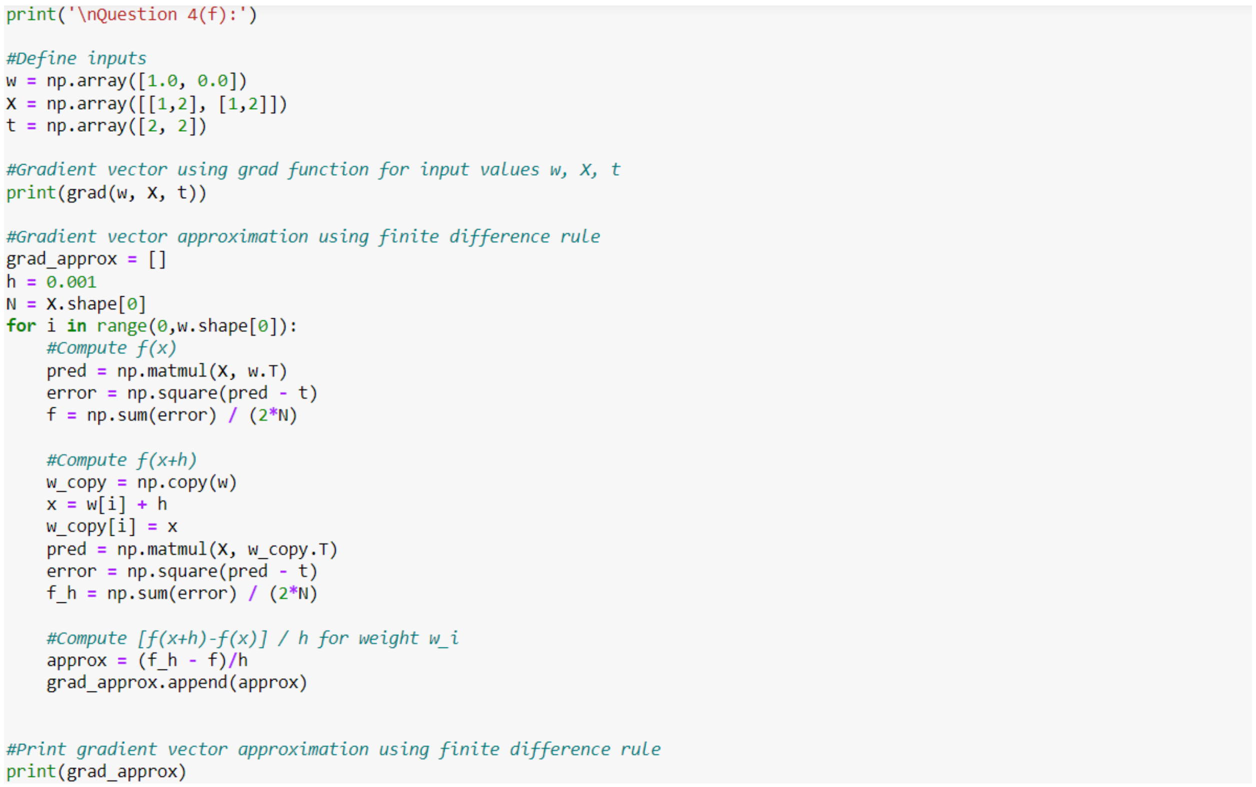
**Part (f)**

We can check that our grad function in part (e) is implemented correctly using the finite difference rule. In 1D, the

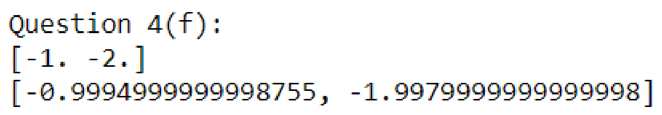
finite difference rule tells us that for small h, we should have

$$\frac{f(x + h) - f(x)}{h} \approx f'(x)$$

Prove to yourself (and your TA) that grad is implement correctly by comparing the result from grad and a gradient estimation obtained from computing J (w) and using the finite difference rule. Briefly justify your choice(s) of w, b, and X.



```python
print('\nQuestion 4(f):')

#Define inputs
w = np.array([1.0, 0.0])
X = np.array([[1,2], [1,2]])
t = np.array([2, 2])

#Gradient vector using grad function for input values w, X, t
print(grad(w, X, t))

#Gradient vector approximation using finite difference rule
grad_approx = []
h = 0.001
N = X.shape[0]
for i in range(0,w.shape[0]):
    #Compute f(x)
    pred = np.matmul(X, w.T)
    error = np.square(pred - t)
    f = np.sum(error) / (2*N)

    #Compute f(x+h)
    w_copy = np.copy(w)
    x = w[i] + h
    w_copy[i] = x
    pred = np.matmul(X, w_copy.T)
    error = np.square(pred - t)
    f_h = np.sum(error) / (2*N)

    #Compute [f(x+h)-f(x)] / h for weight w_i
    approx = (f_h - f)/h
    grad_approx.append(approx)


#Print gradient vector approximation using finite difference rule
print(grad_approx)
```



```
Question 4(f):
[-1. -2.]
[-0.9994999999998755, -1.9979999999999998]
```

In Python, I computed the gradient vector with grad function and estimation obtained from using the finite difference rule. I used the following inputs for w, X, t:

$$w = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} \quad t = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

The reason I chose the w above is because I wanted to showcase the grad function works for weight vector with multiple weights. The reason I chose the X above is because I wanted to show that the grad function works for design matrix with multiple data points and multiple features. The reason I chose the t above is because I wanted to show grad function works with

vector with multiple training labels. The gradient vector that I got with my grad function and finite difference rule approximation are the following:
Grad function output: [-1. -2.]
Finite difference rule output: [-0.9994999999998755, -1.9979999999999998]
We see that the approximation output and grad function output for inputs above is very close and as h approaches zero, approximation will match grad function output. Therefore, the grad function has been implemented correctly.

## Part (g)

Complete the function solve_via_gradient_descent that takes the maximum degree k of a polynomial regression, the learning rate alpha, and the number of iterations niter. This function begins by initializing all of your polynomial regression weights to 0, and returns the value of the weights after niter updates of gradient descent. You should use the grad function you wrote in the Part (e) as a helper function. Print the value of solve_via_gradient_descent(M, alpha=0.0025, niter=10000)

```python
print('\nQuestion 4(g):')

def solve_via_gradient_descent(M, alpha=0.0025, niter=10000):
    '''
    Given `M` - maximum degree of the polynomial regression model
          `alpha` - the learning rate
          `niter` - the number of iterations of gradient descent to run
    Solves for linear regression weights.
    Return weights after `niter` iterations.
    '''
    #Initialize all the weights to zeros
    w = np.zeros([M + 1])

    train_d = train_x[:, 0] # house age

    #Construct the design matrix
    X = vector_multinomial(train_d, M)

    #Compute and update weights using grad function for niter iterations
    for i in range(niter):

        gradient = grad(w, X, train_t[:])
        w = w - alpha*gradient
    return w

print(solve_via_gradient_descent(M=1, alpha=0.0025, niter=10000))
```

```
Question 4(g):
[42.23043921 -0.25227295]
```

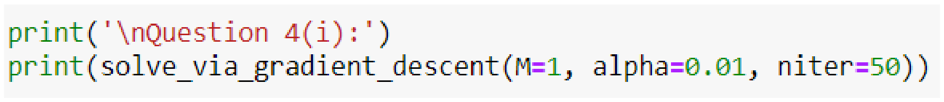
## Part (h)

How do your gradient descent weights from Part (g) compare to the weights that you got from Question 3(a)?

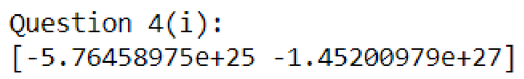
Weights 3(a) : [42.25538722      -0.2532634 ]
Weights 4(g): [42.23043921      -0.25227295]
The weights form question 3(a) are very close compared to the weights from gradient descent such that they differ by a few decimals, respectively.

**Part (i)**

Print the value of *solve_via_gradient_descent(M=1, alpha=0.01, niter=50).*



```
print('\nQuestion 4(i):')
print(solve_via_gradient_descent(M=1, alpha=0.01, niter=50))
```



```
Question 4(i):
[-5.76458975e+25 -1.45200979e+27]
```

In your writeup, explain why this result differs from the one in Part(g).

The results differ from the one in Part(g) because the learning rate (*alpha=0.01*) is larger and the number of iterations of gradient descent to run (*niter=50)* is less then Part(g). The larger learning rate cause the update in the gradient descent to be large such that weights have more variation from update to the next, compared to a smaller learning rate. The smaller number of iterations of gradient descent to run decreases the number of times to update the gradient descent such that the weights change less times compare to large number of iterations. Hence, these factors caused the weights to differ after each update of the gradient descent and eventually lead to different final weights given by the algorithm.

# Cross Validation and Model Selection

When we have a small amount of data, evaluating models is challenging. We may not have the luxury of leaving out data like we did in Questions 2-4 for the purposes of validation. In this part, we will explore a more data efficient way of evaluating machine learning models called cross validation. Recall that in Question 1, we first set aside a test set, then we split the remaining data into 80% training and 20% validation. The validation performance (e.g. MSE) helps us select hyper-parameters. The idea behind cross validation is to repeat this process of train/validation split multiple times, so that we get several different estimates of the validation accuracy. In particular, in Question 1(g), we gave each element of the (non-test) data set a random label (0, 1, 2, 3, 4). This way, we effectively split the data set into 5 groups, and decided that group 0 was the validation set. In this question, we will perform k-Fold Cross Validation. In other words, we will split our data set into k random groups, and repeat the training, allowing each of the k groups a chance to be the validation set. In our case, since we have 5 different groups, what we are proposing is called "5-Fold Cross Validation". In each of the 5 training iterations, we will choose a different group (out of the 5) to be the validation set. The remaining 4 groups will be used for training.

**Part (a)**

By repeating training 5 different times for 5 different training/validation splits, we will get 5 different estimates for the validation MSE, which can be averaged together. What is the advantage of averaging 5 MSE measures, rather than just using a single MSE measure, especially when the data set size is small?

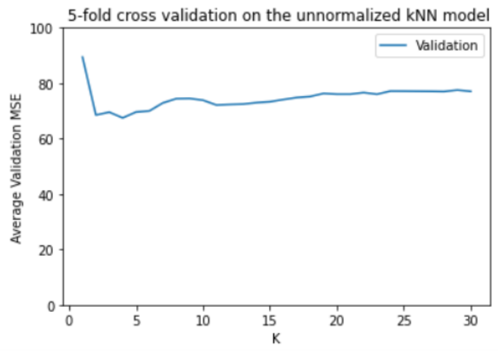
An advantage of averaging 5 MSE measures, rather than just using a single MSE measure is that it allows us to evaluate the model more accurately, especially when the data set size is small. When our data set is small and we use a single MSE measure, we must use an even smaller test set, which could possibly not represent all types of data points. Hence there is a higher chance of MSE for test to vary depending on how we choose the test set. If we use 5 MSE measures, then we would we able to train and test with 5 different training and test data sets which allows us to test and train our model given the limited data more thoroughly.

**Part (b)**

We will perform 5-fold cross validation on the unnormalized kNN model. Produce a plot of the average validation MSE across the 5 folds, similar to the plot from Quesiton 2(e). Your plot should have values of "k" on the x-axis (ranging from 1 to 30 like before), and "Average validation MSE" on the y-axis. Clearly label your plot.

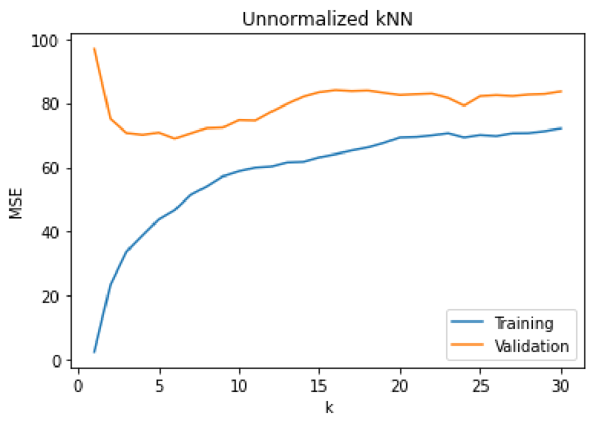
```
print('\n\nQuestion 5')
print('----------')

print('\nQuestion 5(b):')

#Number of data point
n = train_valid.shape[0]

#Number of data points per fold for 5-fold CV
fold_size = round(n/5)

#List of average MSE for each k in range 1-30
cv_avgs = []

for k in range(1,31):
    total = 0
    for L in range(0,5):
        #print(L*fold_size,(L+1)*fold_size)
        #Validation Set for fold
        v = train_valid[L*fold_size:(L+1)*fold_size,:]
        #print(v.shape)

        #Training set for fold
        t = np.concatenate((train_valid[0:L*fold_size,:], train_valid[(L+1)*fold_size:,:]), axis=0)
        #print(t.shape)

        #Seperate training and validation sets by features and labels
        t_x = t[0: ,1:6]
        t_t = t[0: ,6]
        v_x = v[0: ,1:6]
        v_t = v[0: ,6]
        #print(train_x.shape, train_t.shape, valid_x.shape, valid_t.shape)

        #Compute unnormalized KNN prediction
        def predict_fn(new_v):
            return unnorm_knn(new_v, k, features=t_x, labels=t_t)

        #Compute and add validation set MSE to total
        total += compute_mse(predict_fn, data_x=v_x, data_t=v_t)

    #Compute average MSE for validation set for given k in KNN
    cv_avgs.append(total/5)

#Create plot
from matplotlib import pyplot as plt
plt.plot(range(1, 31), cv_avgs)
plt.xlabel("K")
plt.ylabel("Average Validation MSE")
plt.title(" 5-fold cross validation on the unnormalized kNN model")
plt.legend(["Validation"])
plt.axis([None, None, 0, 100])
plt.show()
```

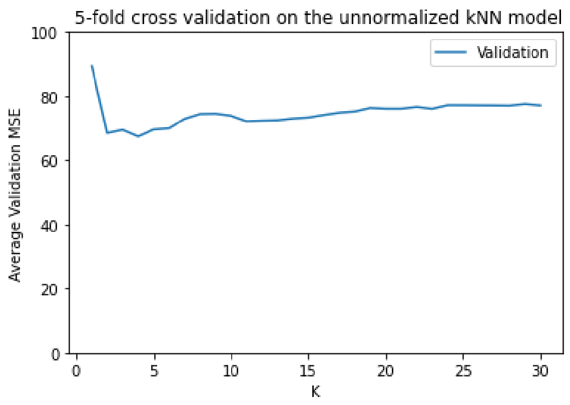
```
Question 5(b):
```



**Part (c)**

Compare your plot from Question 5(b) with the plot from Question 2(e). Which plot appears more smooth (less noisy)? Explain why you think that is.





The plot that appears to be smoother is the 5-fold cross validation plot. I think it is reasonable because for single validation set approach, the training set is smaller so this could lead to more variation in the prediction, hence more variation in the MSE for different models. For the 5-fold cross-validation approach, the training sets are larger, and we use all the data train the model so there is less variation the prediction, hence, less variation in the MSE. Thus, less variation in the MSE causes the plot for 5-fold cross-validation to be smoother.

**Part (d)**

What are some advantages and disadvantages of using k-fold cross validation?

An advantage to using k-fold cross validation is that it eliminates the randomness element in dividing the data for training and validation set . For the one validation set approach, the MSE could depend on how we split the data, hence there is randomness involved in our MSE. With the k-fold cross validation, we effectively use the data available such that all the data point are in the training or validation set at some point, which reduces variability due to no random division of the data. Another advantage of using k-fold cross validation is that it reduces the chance of overfitting. For the one validation set approach, we train data with only one set, so it is possible that we overfit the model to the single training set. With the k-fold cross validation, we have multiple training and validation sets which allows us to generalize the model to multiple training and validation sets. A disadvantage to use k-fold cross validation is that it increases the amount of time it takes to train the model. For the one validation set approach, we trained our model with just one training and validation set. With the k-fold cross validation, we have to train model with multiple training and validation sets which increases the number of computations, hence increased training time.

## Part (e)

If you were to choose a model from questions 2-5 to deploy, which model will you deploy? Compute the test MSE for this optimal model.

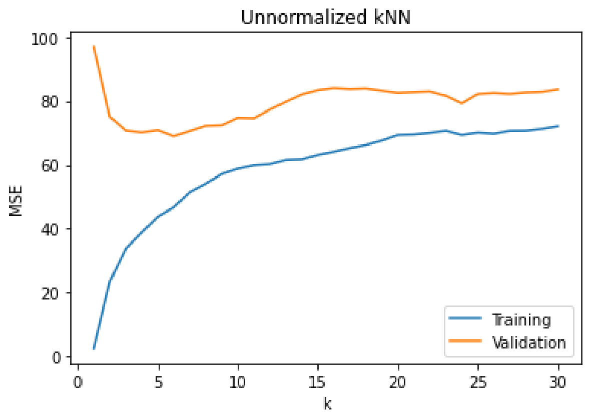
```
print('\nQuestion 5(e):')
#Compute and print test set MSE for optimal model
def predict_unnorm(new_v):
        return unnorm_knn(new_v, 6)


print(compute_mse(predict_unnorm, data_x=test_x, data_t=test_t))
```

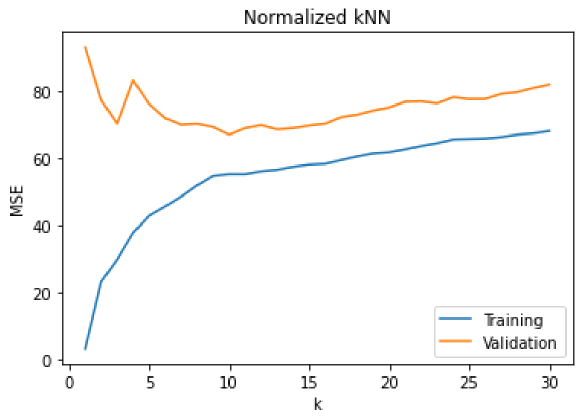
```
Question 5(e):
90.92383333333329
```

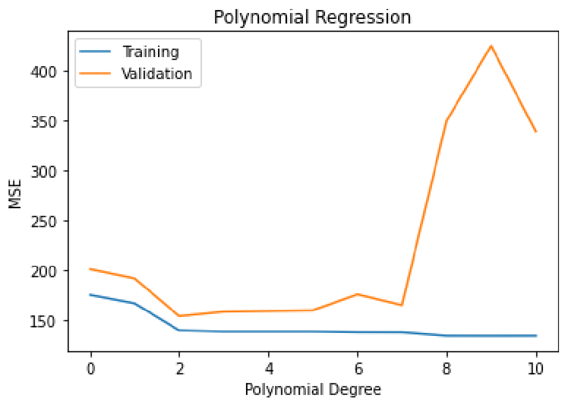
The model I choose to deploy is the unnormalized KNN with k = 6
Test MSE: 90.92383333333329

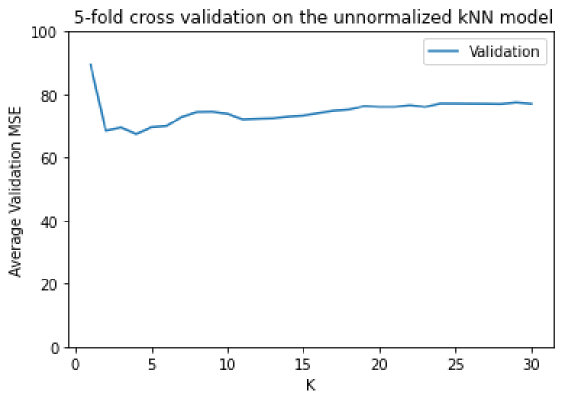
## Part (f)

Explain why you chose the model from the previous part.









I choose to deploy the unnormalized KNN with k = 6. One of the reasons why I choose the unnormalized KNN model over normalized KNN model is because the normalization does not seem to significantly decrease the overall mean square error for the validation set compared to unnormalized model, as shown in the charts above. Since there is no major difference in prediction for validation set, I choose the model which take less time to train since unnormalized model does not need the data to be normalized. Another reason why I choose the unnormalized KNN model over normalized KNN model is because the cross-validation approach provides more evidence that the unnormalized KNN model generalizes with various training and validation sets. In the charts above, we see that the optimal value for k and mean square error trained using cross validation is close to the unnormalized model trained using single validation approach which showcases the generalization of the model. I choose k = 6 because the graphs for unnormalized kNN showcase the optimal value of k is in the range approximately of 3-7. The reason I choose unnormalized KNN model over polynomial feature model is because we only use one feature for training our model. I believe there are other features in the date other than house age which impact house prices like longitude and latitude that were used to train the model. Besides, the MSE for polynomial regression above does not give a lower MSE for validation set than other models which also prevent me from deploying this model.