KNN regression experiments

In class we learned about how KNN regression works, and tips for using KNN. For example, we learned that data should be scaled when using KNN, and that extra, useless predictors should not be used with KNN. Are these tips really correct?

In this notebook we run a bunch of tests to see how KNN is affect by the choice of k, scaling of the predictors, presence of useless predictors, and other things.

One experiment we do not run, and which would be interesting, is to see how KNN performance changes as a function of the size of the training set.

INSTRUCTIONS

Enter code wherever you see # YOUR CODE HERE in code cells, or YOU TEXT HERE in markup cells.

[1]
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
import matplotlib.pyplot as plt
```

[2]
```
# set default figure size
plt.rcParams['figure.figsize'] = [8.0, 6.0]
```

[3]
```
# code in this cell from:
# https://stackoverflow.com/questions/27934885/how-to-hide-code-from-cells-in-ipython-notebook-visualized-with-nbviewer
from IPython.display import HTML

HTML('''<script>
code_show=true;
function code_toggle() {
 if (code_show){
 $('div.input').hide();
 } else {
 $('div.input').show();
 }
 code_show = !code_show
}
$( document ).ready(code_toggle);
</script>
<form action="javascript:code_toggle()"><input type="submit" value="Click here to display/hide the code."></form>''')
```

Click here to display/hide the code.

Read the data and take a first look at it

The housing dataset is good for testing KNN because it has many numeric features. See Aurélien Géron's book titled 'Hands-On Machine learning with Scikit-Learn and TensorFlow' for information on the dataset.

```
[4] df = pd.read_csv("https://raw.githubusercontent.com/grbruns/cst383/master/housing.csv")
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Note that numeric features have different ranges. For example, the mean value of *'total_rooms'* is over 2,500, while the mean value of *'median_income'* is about 4. *'median_house_value'* has a much greater mean value, over $200,000, but we will be using it as the target variable.

```
[6] from IPython.display import Image
    from IPython.core.display import HTML
    Image(url= "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD/2wCEAAoHCBIVFRgVFRYYGBgYGBgZGBgYGBgaGBoaGBgZGRgYGBgcIS4lHB4rHxgYJj
    #Flickr source for "Houses going down" : https://www.flickr.com/photos/59937401@N07/5474464467
```

```
df.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

Missing Data

Notice that 207 houses are missing their *total_bedroom* info:

```
[8] print(df.isnull().sum())
    df[df['total_bedrooms'].isnull()]
```

```
longitude               0
latitude                0
housing_median_age      0
total_rooms             0
total_bedrooms        207
population              0
households              0
median_income           0
median_house_value      0
ocean_proximity         0
dtype: int64
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
290	-122.16	37.77	47.0	1256.0	NaN	570.0	218.0	4.3750	161900.0	NEAR BAY
341	-122.17	37.75	38.0	992.0	NaN	732.0	259.0	1.6196	85100.0	NEAR BAY
538	-122.28	37.78	29.0	5154.0	NaN	3741.0	1273.0	2.5762	173400.0	NEAR BAY
563	-122.24	37.75	45.0	891.0	NaN	384.0	146.0	4.9489	247100.0	NEAR BAY
696	-122.10	37.69	41.0	746.0	NaN	387.0	161.0	3.9063	178400.0	NEAR BAY
...	...	...	...	...	...	...	...	...	...	...
20267	-119.19	34.20	18.0	3620.0	NaN	3171.0	779.0	3.3409	220500.0	NEAR OCEAN
20268	-119.18	34.19	19.0	2393.0	NaN	1938.0	762.0	1.6953	167400.0	NEAR OCEAN
20372	-118.88	34.17	15.0	4260.0	NaN	1701.0	669.0	5.1033	410700.0	<1H OCEAN
20460	-118.75	34.29	17.0	5512.0	NaN	2734.0	814.0	6.6073	258100.0	<1H OCEAN
20484	-118.72	34.28	17.0	3051.0	NaN	1705.0	495.0	5.7376	218600.0	<1H OCEAN

207 rows × 10 columns

Let's drop these instances for now

```
[9] df = df.dropna()
```

Prepare data for machine learning

We will use KNN regression to predict the price of a house from its features, such as size, age and location.

We use a subset of the data set for our training and test data. Note that we keep an unscaled version of the data for one of the experiments we will run.

[10]
```
# for repeatability
np.random.seed(42)
```

[11]
```
# select the predictor variables and target variables to be used with regression
predictors = ['longitude','latitude','housing_median_age','total_rooms', 'total_bedrooms', 'population', 'households', 'median_income']
#dropping categortical features, such as ocean_proximity, including spatial ones such as long/lat.
target = 'median_house_value'
X = df[predictors].values
y = df[target].values
```

[12]
```
# KNN can be slow, so get a random sample of the full data set
indexes = np.random.choice(y.size, size=10000)
X_mini = X[indexes]
y_mini = y[indexes]
```

[13]
```
# Split the data into training and test sets, and scale
scaler = StandardScaler()

# unscaled version (note that scaling is only used on predictor variables)
X_train_raw, X_test_raw, y_train, y_test = train_test_split(X_mini, y_mini, test_size=0.30, random_state=42)

# scaled version
X_train = scaler.fit_transform(X_train_raw)
X_test = scaler.transform(X_test_raw)
```

```
# sanity check
print(X_train.shape)
print(X_train[:3])
```

```
(7000, 8)
[[ 1.22783551 -1.3492796   0.34639424 -0.16627017  0.11697691 -0.15874461
   0.18687025 -0.74984935]
 [ 0.62095726 -0.82169566  0.58720859 -0.11584049 -0.22077651 -0.0770853
  -0.14171346  1.12877289]
 [-1.16983102  0.7563873  -0.45632025 -0.32112946  0.02736886 -0.37395092
  -0.04890738 -0.10303138]]
```

Baseline performance

For regression problems, our baseline is the "blind" prediction that is just the average value of the target variable. The blind prediction must be calculated using the training data. Calculate and print the test set root mean squared error (test RMSE) using this blind prediction. I have provided a function you can use for RMSE.

[15]
```
def rmse(predicted, actual):
    return np.sqrt(((predicted - actual)**2).mean())
```

Double-click (or enter) to edit

[16]
```
y_train_mean = np.mean(y_train)
y_pred_baseline = np.full_like(y_test, y_train_mean)
baseline_rmse = rmse(y_pred_baseline, y_test)
print(baseline_rmse)
```

```
112909.28341439406
```

Performance with default hyperparameters

Using the training set, train a KNN regression model using the ScikitLearn KNeighborsRegressor, and report on the test RMSE. The test RMSE is the RMSE computed using the test data set.

When using the KNN algorithm, use algorithm='brute' to get the basic KNN algorithm.

```
knn_regressor = KNeighborsRegressor(algorithm='brute')
knn_regressor.fit(X_train, y_train)
y_pred_knn = knn_regressor.predict(X_test)
knn_rmse = rmse(y_pred_knn, y_test)
print(knn_rmse)
```

```
62448.852862157735
```

Impact of K

In class we discussed the relationship of the hyperparameter k to overfitting.

I provided code to test KNN on k=1, k=3, k=5, ..., k=29. For each value of k, compute the training RMSE and test RMSE. The training RMSE is the RMSE computed using the training data. Use the 'brute' algorithm, and Euclidean distance, which is the default. You need to add the get_train_test_rmse() function.

[17]

[18]
```python
def get_train_test_rmse(regr, X_train, X_test, y_train, y_test):
    regr.fit(X_train, y_train)
    y_train_pred = regr.predict(X_train)
    y_test_pred = regr.predict(X_test)
    train_rmse = rmse(y_train_pred, y_train)
    test_rmse = rmse(y_test_pred, y_test)
    return train_rmse, test_rmse
```

[19]
```python
n = 30
test_rmse = []
train_rmse = []
ks = np.arange(1, n+1, 2)
for k in ks:
    print(k, ' ', end='')
    regr = KNeighborsRegressor(n_neighbors=k, algorithm='brute')
    rmse_tr, rmse_te = get_train_test_rmse(regr, X_train, X_test, y_train, y_test)
    train_rmse.append(rmse_tr)
    test_rmse.append(rmse_te)
print('done')
```

```
1  3  5  7  9  11  13  15  17  19  21  23  25  27  29  done
```

[20]
```python
# sanity check
print('Test RMSE when k = 3: {:0.1f}'.format(np.array(test_rmse)[ks==3][0]))
```

```
Test RMSE when k = 3: 64167.1
```

Using the training and test RMSE values you got for each value of k, find the k associated with the lowest test RMSE value. Print this k value and the associated lowest test RMSE value. In other words, if you found that k=11 gave the lowest test RMSE, then print the value 11 and the test RMSE value obtained when k=11.

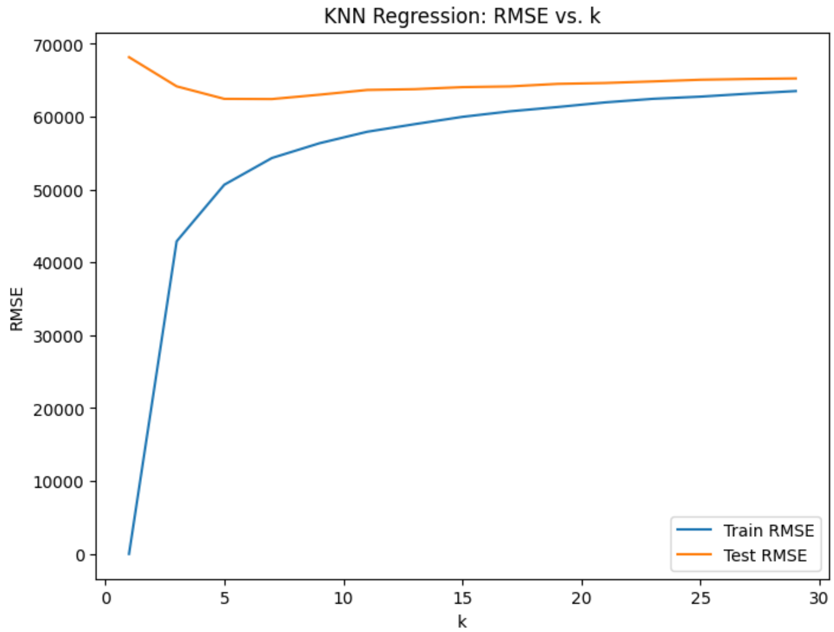
```python
def get_best(ks, rmse):
    min_rmse = float('inf')  # Initialize min_rmse to positive infinity
    best_k = None  # Initialize best_k to None

    # Iterate through the values of k and their corresponding RMSE values
    for k, rmse_value in zip(ks, rmse):
        if rmse_value < min_rmse:
            min_rmse = rmse_value
            best_k = k
    return best_k, min_rmse

best_k, best_rmse = get_best(ks, test_rmse)
print('best k = {}, best test RMSE: {:0.1f}'.format(best_k, best_rmse))
```

```
best k = 7, best test RMSE: 62421.5
```

Plot the test and training RMSE as a function of k, for all the k values you tried.

```
plt.plot(ks, train_rmse, label='Train RMSE')
plt.plot(ks, test_rmse, label='Test RMSE')
plt.xlabel('k')
plt.ylabel('RMSE')
plt.title('KNN Regression: RMSE vs. k')
plt.legend()
```

<matplotlib.legend.Legend at 0x797f1fddacb0>



Comments

In the markup cell below, write about what you learned from your plot. I would expect two or three sentences, but what's most important is that you write something thoughtful.

[22]

Training plot increases compared the test. The test decreased. The larger the value of K the closer both plots get closer to each other.

Impact of noise predictors

Double-click (or enter) to edit

In class we heard that the KNN performance goes down if useless "noisy predictors" are present. These are predictor that don't help in making predictions. In this section, run KNN regression by adding one noise predictor to the data, then 2 noise predictors, then three, and then four. For each, compute the training and test RMSE. In every case, use k=10 as the k value and use the default Euclidean distance as the distance function.

The add_noise_predictor() method makes it easy to add a predictor variable of random values to X_train or X_test.

[23]
```
def add_noise_predictor(X):
    """ add a column of random values to 2D array X """
    noise = np.random.normal(size=(X.shape[0], 1))
    return np.hstack((X, noise))
```

Hint: In each iteration of your loop, add a noisy predictor to both X_train and X_test. You don't need to worry about rescaling the data, as the new noisy predictor is already scaled. Don't modify X_train and X_test however, as you will be using them again.

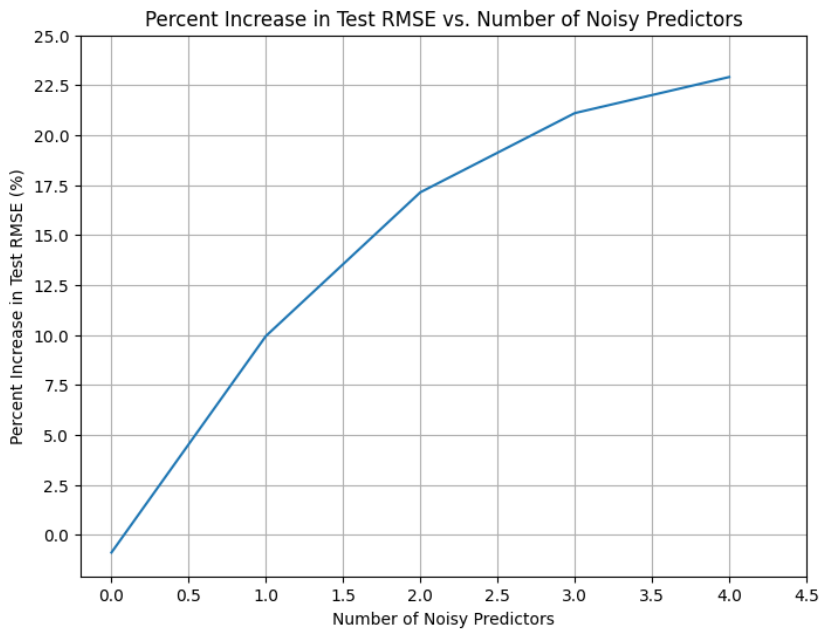
```
train_rmse_noisy = []
test_rmse_noisy = []
percent_increase_rmse = []

X_train_noisy = X_train.copy()
X_test_noisy = X_test.copy()

baseline_rmse = test_rmse[0]

for i in range(0, 5):
    print(i, ' ', end='')

    baseline_rmse = test_rmse[i]

    #if i > 0:
    X_train_noisy = add_noise_predictor(X_train_noisy)
    X_test_noisy = add_noise_predictor(X_test_noisy)

    regr = KNeighborsRegressor(n_neighbors=10)
    rmse_tr, rmse_te = get_train_test_rmse(regr, X_train_noisy, X_test_noisy, y_train, y_test)
    train_rmse_noisy.append(rmse_tr)
    test_rmse_noisy.append(rmse_te)

    percent_increase_rmse.append(100 * (rmse_te - baseline_rmse) / baseline_rmse)

print('done')
```

```
0  1  2  3  4  done
```

Plot the percent increase in test RMSE as a function of the number of noise predictors. The x axis will range from 0 to 4. The y axis will show a percent increase in test RMSE.

*To compute percent increase in RMSE for n noise predictors, compute 100 * (rmse - base_rmse)/base_rmse, where base_rmse is the test RMSE with no noise predictors, and rmse is the test RMSE when n noise predictors have been added.*

```
plt.plot(range(5), percent_increase_rmse)
plt.xlabel('Number of Noisy Predictors')
plt.ylabel('Percent Increase in Test RMSE (%)')
plt.title('Percent Increase in Test RMSE vs. Number of Noisy Predictors')
plt.xticks(np.arange(0, 5, 0.5))
plt.yticks(np.arange(0, max(percent_increase_rmse) + 2.5, 2.5))
plt.grid(True)
```



Comments

Look at the results you obtained and add some thoughtful commentary.

Looks like it increases but slows down aver 2nd noise was added.

Impact of scaling

In class we learned that we should scaled the training data before using KNN. How important is scaling with KNN? Repeat the experiments you ran before (like in the impact of distance metric section), but this time use unscaled data.

Run KNN as before but use the unscaled version of the data. You will vary k as before. Use algorithm='brute' and Euclidean distance.

```
#X_train_raw, X_test_raw
n = 30
test_rmse = []
train_rmse = []
ks = np.arange(1, n+1, 2)
for k in ks:
    print(k, ' ', end='')
    regr = KNeighborsRegressor(n_neighbors=k, algorithm='brute')
    rmse_tr, rmse_te = get_train_test_rmse(regr, X_train_raw, X_test_raw, y_train, y_test)
    train_rmse.append(rmse_tr)
    test_rmse.append(rmse_te)
print('done')
```

```
1  3  5  7  9  11  13  15  17  19  21  23  25  27  29  done
```

Print the best k and the test RMSE associated with the best k.

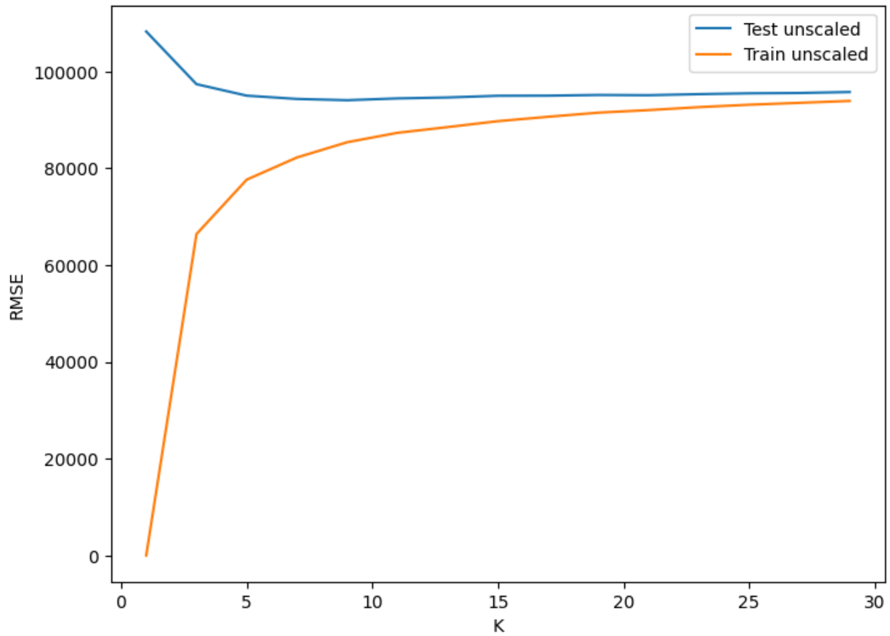
```
[27] best_k, best_rmse = get_best(ks, test_rmse)
     print('best k = {}, best test RMSE: {:0.1f}'.format(best_k, best_rmse))
```

```
best k = 9, best test RMSE: 94057.4
```

Plot training and test RMSE as a function of k. Your plot title should note the use of unscaled data.

```
# YOUR CODE HERE
plt.plot(ks, test_rmse)
plt.plot(ks, train_rmse)
plt.xlabel('K')
plt.ylabel('RMSE')
plt.legend(['Test unscaled','Train unscaled'])
```

```
<matplotlib.legend.Legend at 0x797ee34df460>
```



Comments

Reflect on what happened and provide some short commentary, as in previous sections.

Looks like the graph gab between both plots close in a lot closer. This can be because the values are a lot larger than the scaled version.

Impact of algorithm

We didn't discuss in class that there are variants of the KNN algorithm. The main purpose of the variants is to be faster and to reduce that amount of training data that needs to be stored.

Run experiments where you test each of the three KNN algorithms supported by Scikit-Learn: ball_tree, kd_tree, and brute. In each case, use k=10 and use Euclidean distance.

[29]

```
n = 9
test_rmse_ball_tree= np.array([])
test_rmse_kd= np.array([])
test_rmse_brute= np.array([])
algs = ['ball_tree', 'kd_tree', 'brute']
ks = np.arange(1, n+1, 2)
for alg in algs:
  train_rmse_alg= []
  test_rmse_alg= []

  for k in ks:
    print(k, ' ', end='')
    regr = KNeighborsRegressor(n_neighbors=k, algorithm= alg)
    rmse_tr, rmse_te = get_train_test_rmse(regr, X_train, X_test, y_train, y_test)

    if alg == 'ball_tree':
      test_rmse_ball_tree = np.append(test_rmse_ball_tree, rmse_te)
    elif alg == 'kd_tree':
      test_rmse_kd = np.append(test_rmse_kd,rmse_te)
    elif alg == 'brute':
      test_rmse_brute = np.append(test_rmse_brute, rmse_te)
    else:
      print('invalid algorithm')
  print(alg, 'done')
```

```
1  3  5  7  9  ball_tree done
1  3  5  7  9  kd_tree done
1  3  5  7  9  brute done
```

Print the name of the best algorith, and the test RMSE achieved with the best algorithm.

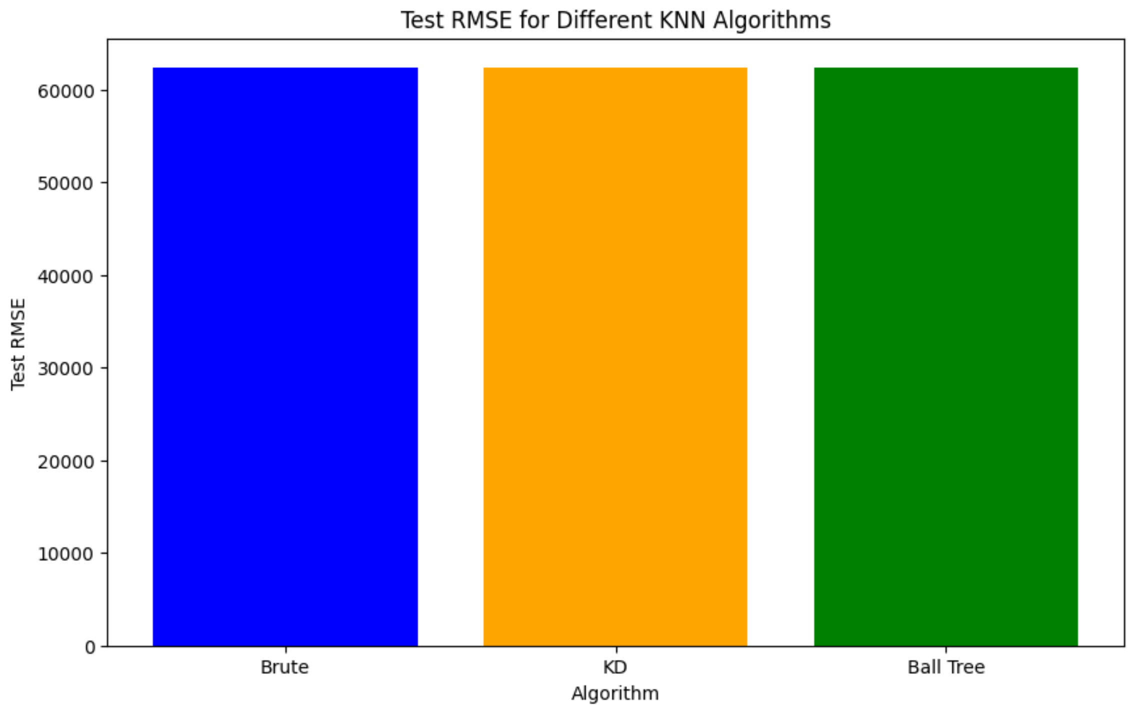
```python
b_t_min = test_rmse_ball_tree.min()
b_t_min_index = np.where(test_rmse_ball_tree == b_t_min)[0][0]
k_d_min = test_rmse_kd.min()
k_d_min_index = np.where(test_rmse_kd == k_d_min)[0][0]
brute_min = test_rmse_brute.min()
brute_min_index = np.where(test_rmse_brute == brute_min)[0][0]

print('Best ball tree k=', ks[b_t_min_index], 'best ball tree test RMSE: {:0.3f}'.format(b_t_min))
print('Best kd tree k=', ks[k_d_min_index], 'best kd test RMSE: {:0.3f}'.format(k_d_min))
print('Best brute k=', ks[brute_min_index], 'best brute test RMSE: {:0.3f}'.format(brute_min))

min_rmses = np.array([b_t_min, k_d_min, brute_min])
min_of_all = min_rmses.min()

if (min_of_all == b_t_min) and (min_of_all == k_d_min) and (min_of_all == brute_min):
    print('All 3 algorithms have the best RMSE with k=', ks[brute_min_index])
elif (min_of_all == b_t_min) and (min_of_all == k_d_min):
    print('Ball tree and kd tree have the best RMSE with k=', ks[b_t_min_index])
elif (min_of_all == b_t_min) and (min_of_all == brute_min):
    print('Ball tree and brute force have the best RMSE with k=', ks[b_t_min_index])
elif (min_of_all == k_d_min) and (min_of_all == brute_min):
    print('Kd tree and brute force have the best RMSE with k=', ks[k_d_min_index])
elif min_of_all == b_t_min:
    print('Ball tree has the best RMSE with k=', ks[b_t_min_index])
elif min_of_all == k_d_min:
    print('Kd tree has the best RMSE with k=', ks[k_d_min_index])
elif min_of_all == brute_min:
    print('Brute force has the best RMSE with k=', ks[brute_min_index])
```

```
Best ball tree k= 7 best ball tree test RMSE: 62421.498
Best kd tree k= 7 best kd test RMSE: 62421.498
Best brute k= 7 best brute test RMSE: 62421.498
All 3 algorithms have the best RMSE with k= 7
```

Plot the test RMSE for each of the three algorithms as a bar plot.

[31]
```
algorithms = ['Brute', 'KD', 'Ball Tree']
#rmse_values = [b_t_min, k_d_min, brute_min]
rmse_values = [brute_min, k_d_min, b_t_min]

plt.figure(figsize=(10, 6))
bars = plt.bar(algorithms, rmse_values, color=['blue', 'orange', 'green'])

for bar, rmse in zip(bars, rmse_values):
    yval = bar.get_height()

plt.xlabel('Algorithm')
plt.ylabel('Test RMSE')
plt.title('Test RMSE for Different KNN Algorithms')
```

```
Text(0.5, 1.0, 'Test RMSE for Different KNN Algorithms')
```



Comments

As usual, reflect on the results and add comments.

Looks like the results are all the same. Using knn finds the best solution for each neighbor of numbers. Once established they all preform the same.

Impact of weighting

It was briefly mentioned in lecture that there is a variant of KNN in which training points are given more weight when they are closer to the point for which a prediction is to be made. The 'weight' parameter of KNeighborsRegressor() has two possible values: 'uniform' and 'distance'. Uniform is the basic algorithm.

Run an experiment similar to the previous one. Compute the test RMSE for uniform and distance weighting. Using k = 10, the brute algorithm, and Euclidean distance.

```
test_rmse_uniform = []
test_rmse_distance = []

ks = np.arange(1, 10, 2)
weights = ['uniform', 'distance']

for weight in weights:
    print(weight, ' ', end='')
    train_rmse_alg = []
    test_rmse_alg = []

    for k in ks:
        print(k, ' ', end='')
        regr = KNeighborsRegressor(n_neighbors=k, algorithm='brute', weights=weight)
        regr.fit(X_train, y_train)

        y_train_pred = regr.predict(X_train)
        y_test_pred = regr.predict(X_test)

        train_rmse = np.sqrt(((y_train_pred - y_train)**2).mean())
        test_rmse = np.sqrt(((y_test_pred - y_test)**2).mean())

        if weight == 'uniform':
            test_rmse_uniform.append(test_rmse)
        elif weight == 'distance':
            test_rmse_distance.append(test_rmse)
        else:
            print('Invalid weight')
    print(weight, 'done')
```

```
uniform  1  3  5  7  9  uniform done
distance  1  3  5  7  9  distance done
```

Print the weighting the gave the lowest test RMSE, and the test RMSE it achieved.

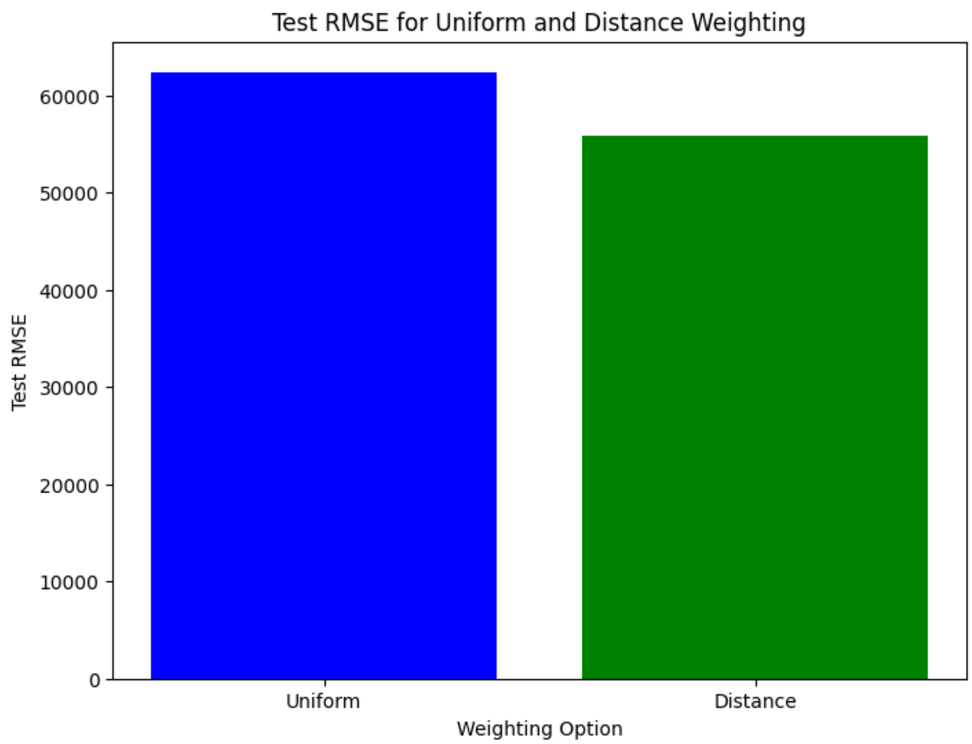
```
uniform_min_index = np.argmin(test_rmse_uniform)
best_uniform_k = ks[uniform_min_index]
best_uniform_rmse = test_rmse_uniform[uniform_min_index]

distance_min_index = np.argmin(test_rmse_distance)
best_distance_k = ks[distance_min_index]
best_distance_rmse = test_rmse_distance[distance_min_index]

print('Best uniform k =', best_uniform_k, ', Best uniform test RMSE: {:0.3f}'.format(best_uniform_rmse))
print('Best distance k =', best_distance_k, ', Best distance test RMSE: {:0.3f}'.format(best_distance_rmse))

if best_uniform_rmse == best_distance_rmse:
    print('Both have the best RMSE with k =', best_uniform_k)
elif best_uniform_rmse < best_distance_rmse:
    print('Uniform has the best RMSE with k =', best_uniform_k)
else:
    print('Distance has the best RMSE with k =', best_distance_k)
```

```
Best uniform k = 7 , Best uniform test RMSE: 62421.498
Best distance k = 7 , Best distance test RMSE: 55800.710
Distance has the best RMSE with k = 7
```

Create a bar plot showing the test RMSE for the uniform and distance weighting options.

```
weight_options = ['Uniform', 'Distance']

test_rmse_values = [best_uniform_rmse, best_distance_rmse]

#plt.figure(figsize=(8, 6))
plt.bar(weight_options, test_rmse_values, color=['blue', 'green'])

plt.xlabel('Weighting Option')
plt.ylabel('Test RMSE')
plt.title('Test RMSE for Uniform and Distance Weighting')
```

```
Text(0.5, 1.0, 'Test RMSE for Uniform and Distance Weighting')
```



Comments

As usual, reflect and comment.

The optimal number is still 7 but it does seem to have more precision than before. This can be factor of the weights of the distance.

Conclusions

Please provide at least a few sentences of commentary on the main things you've learned from the experiments you've run.

kNN is very important. Helps you get the precision you need for algorithms. It gave me a better understanding that a small decimal difference can alter the precision by a good amount. Reminds me of engineers that put satellites into space and a trajectory of a decimal can have the mission be a failure.