Student Name: Raymond Shum

Class: CST383-30_SP22

Assignment: Module 6 - HW6

Due Date: February 15, 2022

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.

In [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
```

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

In [3]:
```
# code in this cell from:
# https://stackoverflow.com/questions/27934885/how-to-hide-code-from-cells-in-ipython-notebook-visualized-with-
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."></for
```

Out[3]: 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.

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

In [5]:
```
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.

In [6]:
```
from IPython.display import Image
from IPython.core.display import HTML
Image(url= "
#Flickr source for "Houses going down" : https://www.flickr.com/photos/59937401@N07/5474464467
```

Out[6]:

In [7]:
```
df.describe()
```

Out[7]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_ho
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	206
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	2068
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	1153
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	149
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	1196
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	1797
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	2647
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	5000

Missing Data

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

In [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
```

Out[8]:

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

207 rows × 10 columns

Let's drop these instances for now

In [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.

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

In [11]:
```
# select the predictor variables and target variables to be used with regression
predictors = ['longitude','latitude','housing_median_age','total_rooms', 'total_bedrooms', 'population', 'house
#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
```

In [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]
```

In [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)
```

In [14]:
```
# 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.

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

In [16]:
```
baseline = rmse(y_train.mean(), y_test).round(1)

print(f"test, rmse baseline: {baseline}")
```

```
test, rmse baseline: 112909.3
```

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.

In [17]:
```
knn = KNeighborsRegressor(algorithm="brute")
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
default = rmse(predictions, y_test).round(1)

print(f"test RMSE, default hyperparameters: {default}")
```

```
test RMSE, default hyperparameters: 62448.9
```

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.*

In [18]:
```
def get_train_test_rmse(regr, X_train, X_test, y_train, y_test):
    regr.fit(X_train, y_train)

    test = regr.predict(X_test)
    rmse_te = rmse(test, y_test)

    train = regr.predict(X_train)
    rmse_tr = rmse(train, y_train)

    return rmse_tr, rmse_te
```

In [19]:
```
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
```

In [20]:
```
# sanity check
print('Test RMSE when k = 3: {:0.1f}'.format(test_rmse[1]))
```

```
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.

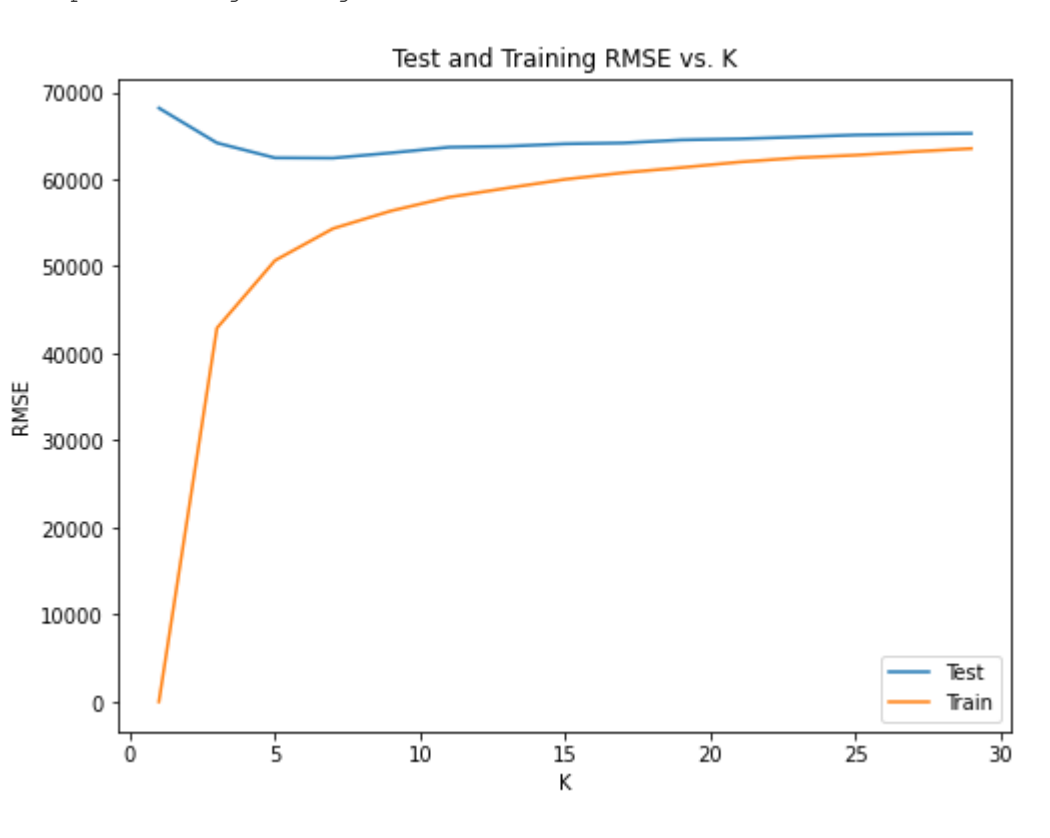
In [21]:
```
def get_best(ks, rmse):
    best_rmse = min(rmse)
    index = rmse.index(best_rmse)
    best_k = ks[index]
    return best_k, best_rmse

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.

In [22]:
```
plt.plot(ks, test_rmse, label="Test")
plt.plot(ks, train_rmse, label="Train")
plt.title("Test and Training RMSE vs. K")
plt.xlabel("K")
plt.ylabel("RMSE")
plt.legend()
```

Out[22]:
```
<matplotlib.legend.Legend at 0x2ab012d1340>
```



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.

RMSE for training data seems to always be lower than error for test RMSE. Professor Memo confirmed that this is because test has not previously seen by the model. RMSE for both the training and test predictions appear to converge as K increases. The test RMSE starts high, falls until it reaches the best k value at k = 7, and slowly increases afterwards.

Impact of noise predictors

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.*

In [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.*

In [24]:
```
X_te = X_test.copy()
X_tr = X_train.copy()
k = 10
test_rmse = []
train_rmse = []

for i in range(5):
    print(i, ' ', end='')
    regr = KNeighborsRegressor(n_neighbors=k, algorithm="brute", p=2)
    rmse_tr, rmse_te = get_train_test_rmse(regr, X_tr, X_te, y_train, y_test)
    train_rmse.append(rmse_tr)
    test_rmse.append(rmse_te)
    X_te = add_noise_predictor(X_te)
    X_tr = add_noise_predictor(X_tr)
print("done")
```

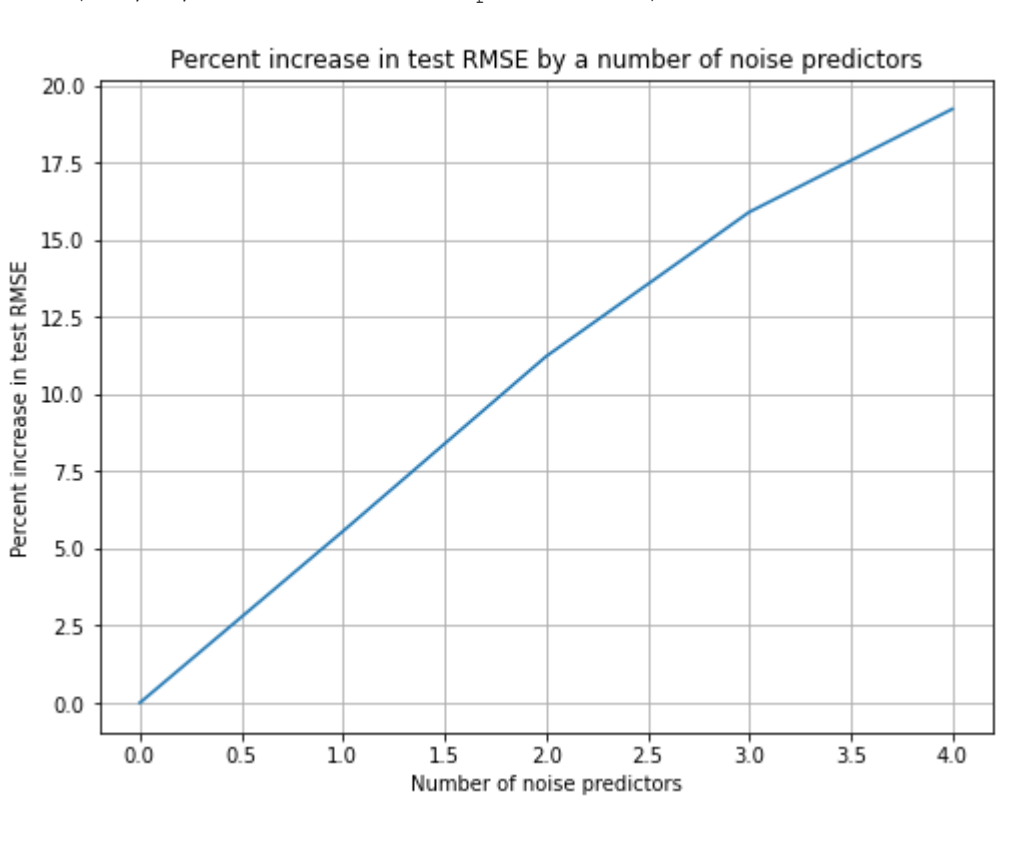
```
0  1  2  3  4  done
```

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.*

In [25]:
```
percent_increase = []
for i in test_rmse:
    percent_increase.append(100 * (i - test_rmse[0]) / test_rmse[0])

plt.plot(range(5), percent_increase)
plt.grid(visible=True)
plt.title("Percent increase in test RMSE by a number of noise predictors")
plt.ylabel("Percent increase in test RMSE")
plt.xlabel("Number of noise predictors")
```

Out[25]:
```
Text(0.5, 0, 'Number of noise predictors')
```



Comments

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

Test RMSE appears to increase as noisy predictors are added to the data. In our sample, it seems that each additional noisy predictor increases the RMSE by between 4%-7% above the base with the largest increases occuring with the least number of noisy predictors already present. This illustrates the importance of identifying noisy predictors through data exploration and removing them during the data preprocessing stage.

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.

In [26]:
```
n = 30
raw_test_rmse = []
raw_train_rmse = []
ks = np.arange(1, n+1, 2)
for k in ks:
    print(k, ' ', end='')
    regr = KNeighborsRegressor(n_neighbors=k, algorithm='brute',p=2)
    rmse_tr, rmse_te = get_train_test_rmse(regr, X_train_raw, X_test_raw, y_train, y_test)
    raw_train_rmse.append(rmse_tr)
    raw_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.

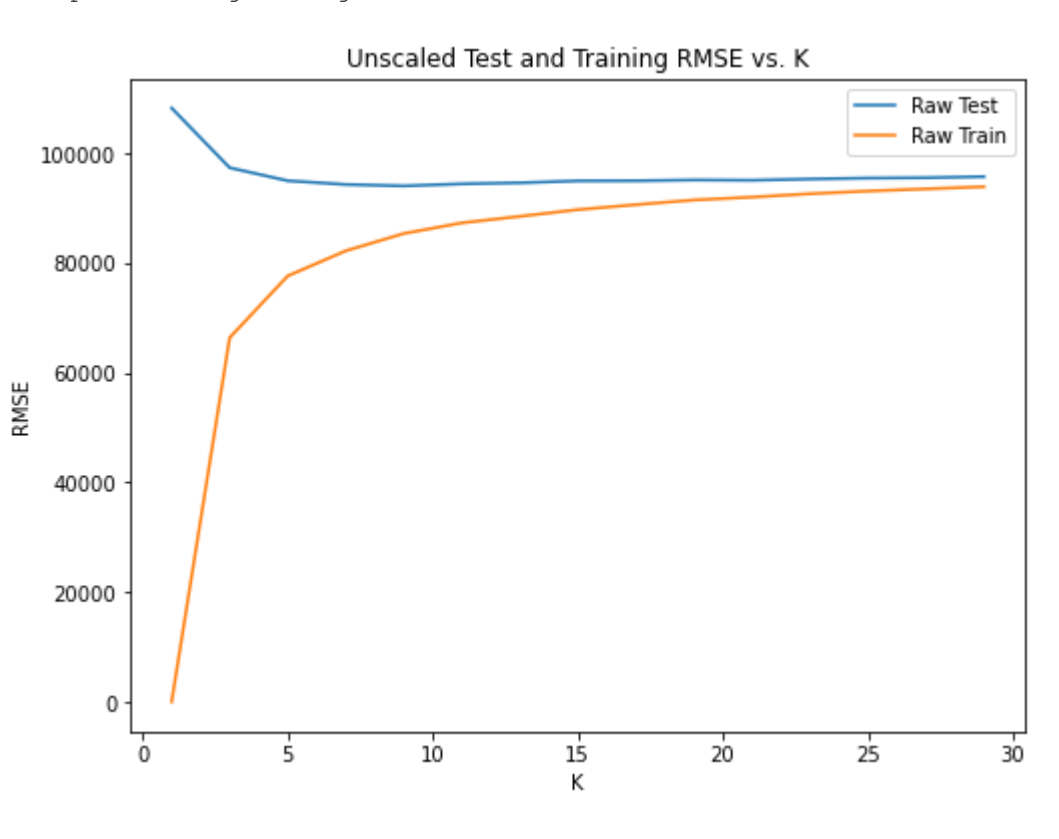
In [27]:
```
best_k, best_rmse = get_best(ks, raw_test_rmse)
print('best k = {}, best test RMSE: {:0.1f}'.format(best_k, best_rmse))
```

```
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.

In [28]:
```
plt.plot(ks, raw_test_rmse, label="Raw Test")
plt.plot(ks, raw_train_rmse, label="Raw Train")
plt.title("Unscaled Test and Training RMSE vs. K")
plt.xlabel("K")
plt.ylabel("RMSE")
plt.legend()
```

Out[28]:
```
<matplotlib.legend.Legend at 0x2ab7bc73ca0>
```



Comments

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

The shape of the graph appears to be similar to the scaled version, but with higher values of RMSE. The best K value is also higher, at 9 instead of 7. I feel that this probably impacts performance as higher values of K means higher burden in terms of calculation to find distances per point. Higher values in the unscaled training data may also mean more memory is being used.

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.*

In [29]:
```
algorithm = ['ball_tree', 'kd_tree', 'brute']
results = []
for algo in algorithm:
    n = 10
    test_rmse = []
    train_rmse = []
    ks = np.arange(1, n+1, 2)
    for k in ks:
        regr = KNeighborsRegressor(n_neighbors=k, algorithm=algo,p=2)
        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(k, ' ', end='')
    best_k, best_rmse = get_best(ks, test_rmse)
    results.append((algo,best_k,best_rmse))
    print(f'{algo} 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.

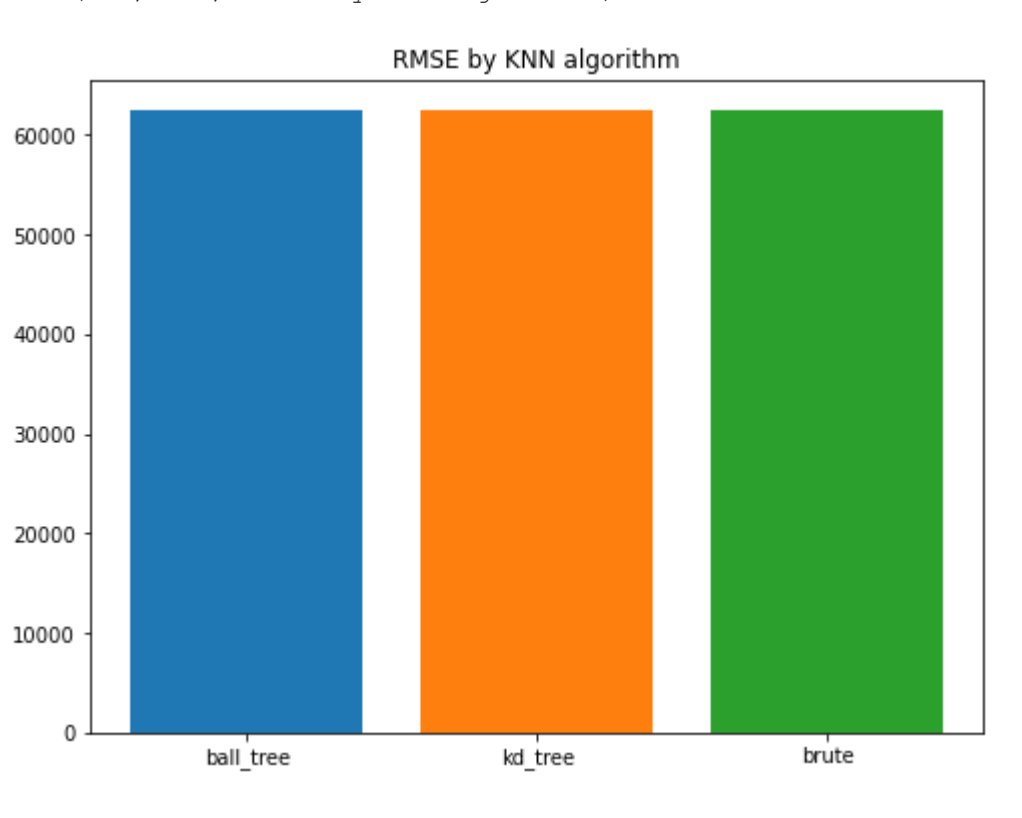
In [30]:
```
for res in results:
    print(f"best {res[0]} k = {res[1]}, best {res[0]} test RMSE: {res[2].round(3)}")

print(f"All 3 have the same best rmse with k = {res[1]}")
```

```
best ball_tree k = 7, best ball_tree test RMSE: 62421.498
best kd_tree k = 7, best kd_tree test RMSE: 62421.498
best brute k = 7, best brute test RMSE: 62421.498
All 3 have the same best rmse with k = 7
```

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

In [31]:
```
for res in results:
    plt.bar(res[0], res[2])

plt.title("RMSE by KNN algorithm")
```

Out[31]:
```
Text(0.5, 1.0, 'RMSE by KNN algorithm')
```



Comments

As usual, reflect on the results and add comments.

It seems as though each algorithm is equally accurate even before rounding the RMSE values. The primary visible difference seems to be time for execution, with brute being the slowest. According to this medium article: https://outline.com/YU7nSM, the difference in speed is due to the use of a binary tree data structure in the KD and Ball tree algorithms and possibly the dimensionality and structure of data (when considering performance differences between the two tree algorithms).

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.

In [32]:
```
weights = ['uniform', 'distance']
results = []
for weight in weights:
    n = 10
    test_rmse = []
    train_rmse = []
    ks = np.arange(1, n+1, 2)
    for k in ks:
        regr = KNeighborsRegressor(n_neighbors=k, weights=weight,p=2)
        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(k, ' ', end='')
    best_k, best_rmse = get_best(ks, test_rmse)
    results.append((weight,best_k,best_rmse))
    print(f'{weight} weight done')
```

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

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

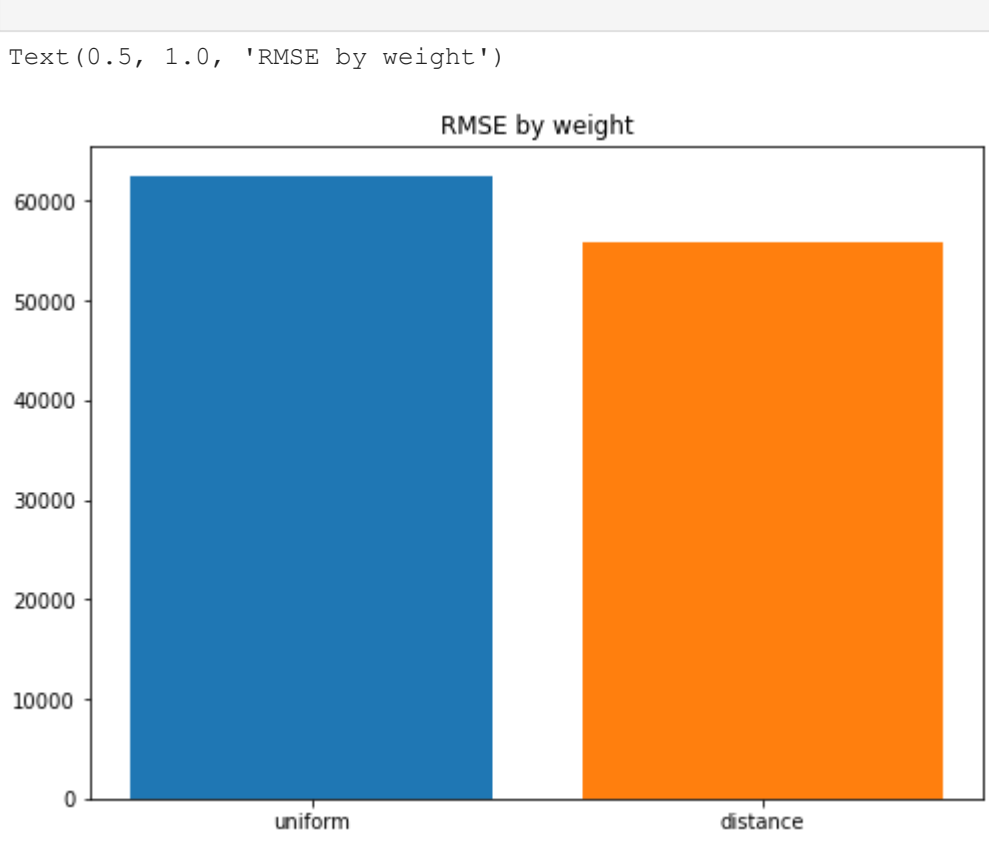
In [33]:
```
for res in results:
    print(f"best {res[0]} weight k = {res[1]}, best {res[0]} weight test RMSE: {res[2].round(3)}")
```

```
best uniform weight k = 7, best uniform weight test RMSE: 62421.498
best distance weight k = 7, best distance weight test RMSE: 55800.71
```

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

In [34]:
```
for res in results:
    plt.bar(res[0], res[2])

plt.title("RMSE by weight")
```

Out[34]:
```
Text(0.5, 1.0, 'RMSE by weight')
```



Comments

As usual, reflect and comment.

Weighting by distance resulted in lower RMSE compared to weighting uniformly. All neighbors have equal influence over a prediction when using uniform weighting while closer neighbors are given more weight when using the distance option. It appears that in our experiment, the model was slightly more accurate when giving more weight to closer neighbors.

Conclusions

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

By comparing the performance of tuned data and parameters against baseline values, this homework illustrates the importance of data preprocessing, scaling and tuning of hyperparameters in the accuracy and performance of the model. We found that k=7 provided the test RMSE. The default K value is 5, so we would only have understood this through tuning the model. We found that the inclusion of noisy predictors led to an increase in test RMSE, which shows us the importance of identifying and dropping these features during data exploration. We saw that training with unscaled data had an overall higher RMSE compared to the results using scaled data, also with higher k values representing best test RMSE. We know from lecture that unscaled data may have an outsized effect on the resulting predictions. We saw that all algorithms appear to have the same best k and test RMSE values, but the tree algorithms appeared to have better performance. Finally, we saw that weighting by distance appeared to provide lower test RMSE results compared to uniform weighting.