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.

```
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


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


# code in this cell from:
# https://stackoverflow.com/questions/27934885/how-to-hide-code-from-cells-in-ipython-
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 disp
```

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.

```
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.

```
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@N07/54
```

```
df.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrc
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.00
mean	-119.569704	35.631861	28.639486	2635.763081	537.87
std	2.003532	2.135952	12.585558	2181.615252	421.38
min	-124.350000	32.540000	1.000000	2.000000	1.00
25%	-121.800000	33.930000	18.000000	1447.750000	296.00
50%	-118.490000	34.260000	29.000000	2127.000000	435.00
75%	-118.010000	37.710000	37.000000	3148.000000	647.00
max	-114.310000	41.950000	52.000000	39320.000000	6445.00

Missing Data

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

```
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
290	-122.16	37.77	47.0	1256.0	NaN

Let's drop these instances for now

696	-122.10	37.69	41.0	746.0	NaN

```
df = df.dropna()
```

20267	-119.19	34.20	18.0	3620.0	NaN

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.

```
# for repeatability
np.random.seed(42)
```

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

```
# 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]
```

```
# 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=

# 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.

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


mse = rmse(y.mean(), y_test)
print('test, rmse baseline: {0:.0f}'.format(mse))
```

```
test, rmse baseline: 112913
```

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.

```
kreg = KNeighborsRegressor(algorithm='brute')
```

```
kreg.fit(X_train, y_train)
predictions = kreg.predict(X_test)
mse = rmse(predictions, y_test)
print('test RMSE, default hyperparameters: {0:.1f}'.format(mse))
```

```
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.

```
def get_train_test_rmse(regr, X_train, X_test, y_train, y_test):
    regr.fit(X_train, y_train)
    predictions = regr.predict(X_train)
    rmse_tr = rmse(predictions, y_train)
    predictions = regr.predict(X_test)
    rmse_te = rmse(predictions, y_test)
    return rmse_tr, rmse_te
```

```
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
```

```
# 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.

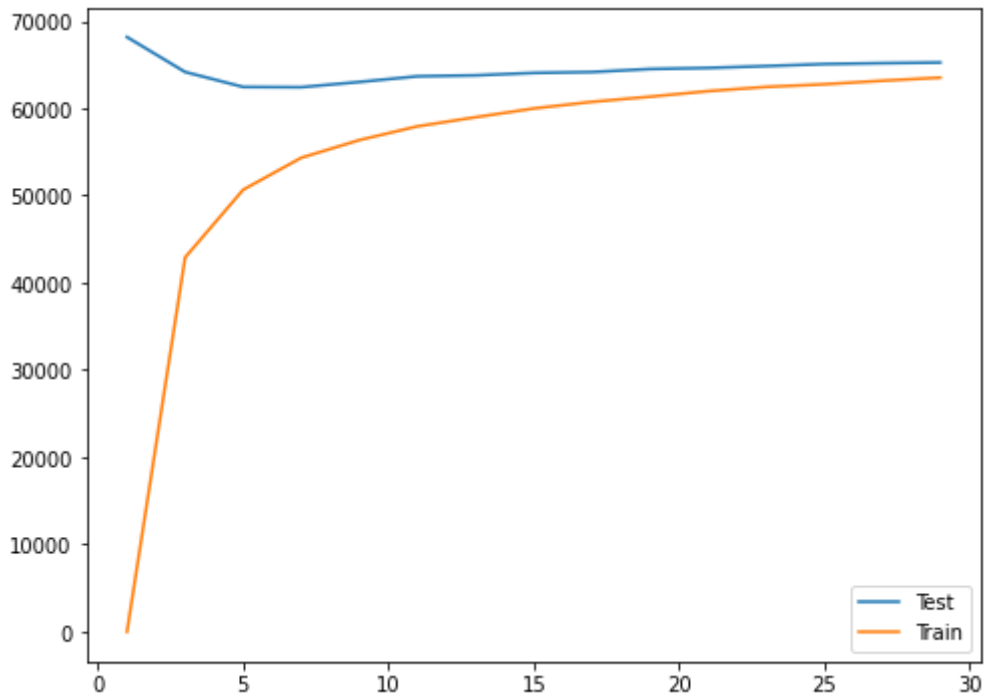
```
def get_best(ks, rmse):
    i = rmse.index(min(rmse))
    best_rmse = rmse[i]
    best_k = ks[i]
    return best_k, best_rmse

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.

```
plt.plot(ks, test_rmse)
plt.plot(ks, train_rmse)
plt.legend(labels=('Test','Train'))
```

```
<matplotlib.legend.Legend at 0x7f241950c490>
```



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.

The k hyperparameter indicates the number of nearest neighbors that will be considered in the classification algorithm. We can see that as this increases, the accuracy of the trained data set increases. More points of classification means more context for each point of data, which in turn means increased accuracy up to a certain point. In this data set, there appears to be a linear progression of results. However, the increase in accuracy comes with gradually diminishing returns as k goes up.

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.

```
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.

```
ns = [0,1,2,3,4]
X_train_noise = X_train.copy()
X_test_noise = X_test.copy()
train_rmse_noise = []
test_rmse_noise = []
train_rmse_base = []
test_rmse_base = []
for n in ns:
    #noise calculation
    kreg_noise = KNeighborsRegressor(n_neighbors=10, algorithm='brute')
    kreg_noise.fit(X_train_noise, y_train)
    predictions_noise = kreg_noise.predict(X_train_noise)
    rmse_tr_noise = rmse(predictions_noise, y_train)
```

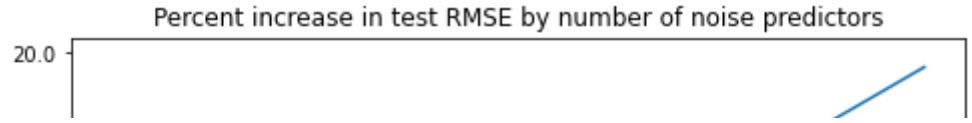
```
    predictions_noise = kreg_noise.predict(X_test_noise)
    rmse_te_noise = rmse(predictions_noise, y_test)
    train_rmse_noise.append(rmse_tr_noise)
    test_rmse_noise.append(rmse_te_noise)
    #base calculation
    kreg_base = KNeighborsRegressor(n_neighbors=10, algorithm='brute')
    kreg_base.fit(X_train, y_train)
    predictions_base = kreg_base.predict(X_train)
    rmse_tr_base = rmse(predictions_base, y_train)
    predictions_base = kreg_base.predict(X_test)
    rmse_te_base = rmse(predictions_base, y_test)
    train_rmse_base.append(rmse_tr_base)
    test_rmse_base.append(rmse_te_base)
    X_train_noise = add_noise_predictor(X_train_noise)
    X_test_noise = add_noise_predictor(X_test_noise)
```

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.*

```
perc_inc_rmse_noise = []
for n in ns:
    perc_inc_rmse_noise.append((100 * (test_rmse_noise[n] - test_rmse_base[n])/test_rm
plt.plot(ns,perc_inc_rmse_noise)
plt.ylabel('% increase in test RMSE')
plt.xlabel('# noise predictors')
plt.title('Percent increase in test RMSE by number of noise predictors')
```

```
Text(0.5, 1.0, 'Percent increase in test RMSE by number of noise predictors
```



Comments

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

There is a steady increase in test RMSE when noise predictors are added. The random, useless data throws off the alogrithm and makes prediction less accurate. This exemplifies the need to hunt and remove such data before fitting and training a model.

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.

```
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_t
    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.

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

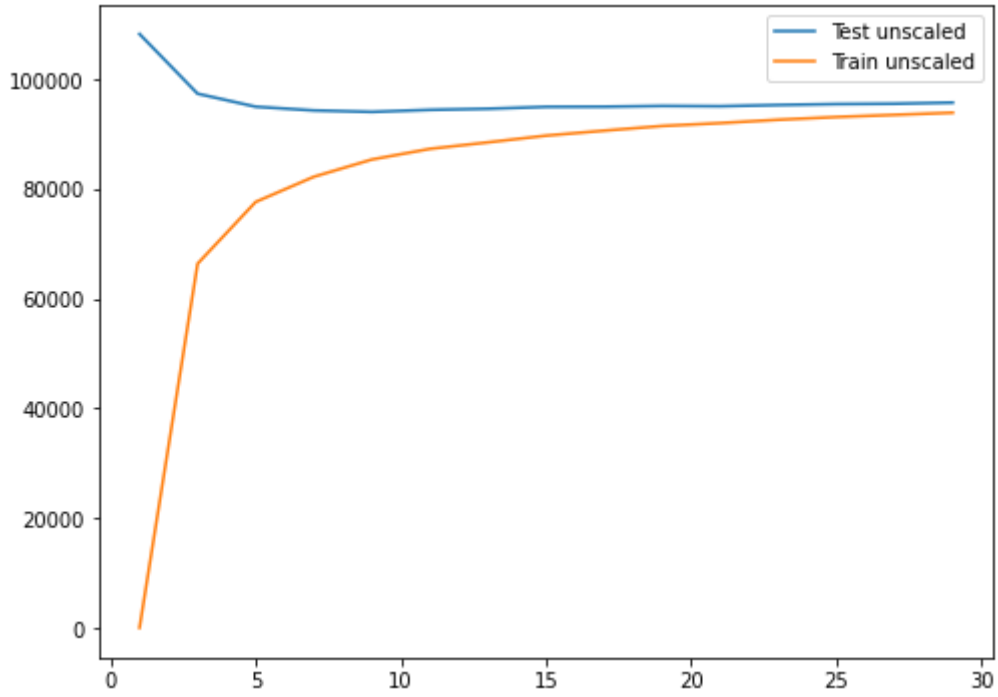
```
    return best_k, best_rmse

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

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

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

```
plt.plot(ks, test_rmse)
plt.plot(ks, train_rmse)
plt.legend(labels=('Test unscaled','Train unscaled'))
```

```
    <matplotlib.legend.Legend at 0x7f2416c3b7d0>
```



Comments

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

With unscaled data like this, certain features are dominating the distance algorithm and shifting the results. The classification accuracing has somewhat decreased, and it is taking a higher number of neighbors to get the same training RMSE/accuracy as before.

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.

```
algs = ['ball_tree','kd_tree','brute']
test_rmse_algorithms = []
for a in algs:
    kreg_algorithms = KNeighborsRegressor(n_neighbors=10, algorithm=a)
    kreg_algorithms.fit(X_train, y_train)
    predictions_algorithms = kreg_algorithms.predict(X_test)
    rmse_te_algorithms = rmse(predictions_algorithms, y_test)
    test_rmse_algorithms.append(rmse_te_algorithms)
```

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

```
def get_best_alg(algs, rmse):
    i = rmse.index(min(rmse))
    best_rmse = rmse[i]
    best_alg = algs[i]
    return best_alg, best_rmse

best_alg, best_rmse = get_best_alg(algs, test_rmse_algorithms)

if ((test_rmse_algorithms[0] == test_rmse_algorithms[1]) and (test_rmse_algorithms[0]
    i = 0
    for a in algs:
        print('algorithm = {}, test RMSE = {:0.1f}'.format(a, test_rmse_algorithms[i])
        i = i+1
    print('All algorithms have the same result.')
else:
    print('best algorithm = {}, best test RMSE = {:0.1f}'.format(best_alg, best_rmse))
```

```
 algorithm = ball_tree, test RMSE = 63254.8
 algorithm = kd_tree, test RMSE = 63254.8
 algorithm = brute, test RMSE = 63254.8
 All algorithms have the same result.
```

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

```
plt.bar(algs, test_rmse_algorithms)
```

```
<BarContainer object of 3 artists>
```

Comments

As usual, reflect on the results and add comments.

With this data set and with k=10, all three classification algorithms have the same accuracy rate. However, I also tested with other values of k with the same results. Either this data set is not one which would exemplify the differences in the algorithms, or perhaps there is no meaningful differnce in their outcomes.

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.

```
weights = ['uniform','distance']
train_rmse_weights = []
test_rmse_weights = []
for w in weights:
    kreg_weights = KNeighborsRegressor(n_neighbors=10, algorithm='brute')
    kreg_weights.fit(X_train, y_train)
```

```
    predictions_weights = kreg_weights.predict(X_test)
    rmse_te_weights = rmse(predictions_weights, y_test)
    test_rmse_weights.append(rmse_te_weights)
```

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

```
def get_best_weight(weights, rmse):
    i = rmse.index(min(rmse))
    best_rmse = rmse[i]
    best_weight = algs[i]
    return best_weight, best_rmse

best_weight, best_rmse = get_best_weight(weights, test_rmse_weights)

if (test_rmse_weights[0] == test_rmse_weights[1]):
    i = 0
    for w in weights:
        print('weight = {}, test RMSE = {:0.1f}'.format(w, test_rmse_weights[i]))
        i = i+1
    print('All weights have the same result.')
else:
    print('best weight = {}, best test RMSE = {:0.1f}'.format(best_weight, best_rmse))
```

```
weight = uniform, test RMSE = 63254.8
weight = distance, test RMSE = 63254.8
All weights have the same result.
```

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

```
plt.bar(weights, test_rmse_weights)
```

```
<BarContainer object of 2 artists>
```

Comments

As usual, reflect and comment.

Like with the various algorithms, the various weights available have no effect on the RMSE for this data set. Again, this is the case for multiple values of k tested. It seems that like algorithms, either this data set won't show the differences or there are no meaningful differences in RMSE with the weight options.

Conclusions

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

The value of k chosen for knn has a drastic effect on RMSE and classification accuracy. It is key for a data scientist to carefully choose this value for the best balance between performance and prediction accuracy.

The presence of noise predictors, especially if several, significantly affects RMSE. In this data set, the presence of 4 noise predictors led to a 20% drop in accuracy. Noise predictors, or those features that are random and not relevant to prediction, should always be filtered out before scaling, training and fitting a knn model.

Using unscaled data certainly has an affect on RMSE and prediction accuracy, however it's likely that the larger the difference in raw values between difference features, the larger the difference will be. In this data set, unscaled data somewhat skewed the results but not drastically.

However, it seems that various algorithms and weights available with the knn function lead to little or no difference in RMSE, at least for this data set.

✓ 0s completed at 1:45 PM