Predicting Car Prices

In this project, we'll practice the machine learning workflow and explore the fundamentals of machine learning using the k-nearest neighbors algorithm to predict a car's market price using its attributes.

The data set we will be working with contains information on various cars. For each car, we have information about the technical aspects of the vehicle such as the motor's displacement, the weight of the car, the miles per gallon, how fast the car accelerates, and more.

Introduction to the Data Set

In [1080]:

```
import pandas as pd
import numpy as np

pd.options.display.max_columns = 99
```

In [1081]:

```
cols = ['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration', 'num-of-doors', 'body-style',
        'drive-wheels', 'engine-location', 'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type
',
        'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke', 'compression-rate', 'horsepower', 'pe
ak-rpm', 'city-mpg', 'highway-mpg', 'price']
cars = pd.read_csv('imports-85.data', names=cols)

cars.head()
```

Out[1081]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	width	height	curb-weight	engine-type
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	168.8	64.1	48.8	2548	dohc
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	168.8	64.1	48.8	2548	dohc
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	171.2	65.5	52.4	2823	ohcv
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	176.6	66.2	54.3	2337	ohc
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	176.6	66.4	54.3	2824	ohc

In [1082]:

```
# Select only the columns with continuous values from - https://archive.ics.uci.edu/ml/machine-learning-databases
/autos/imports-85.names
continuous_values_cols = ['normalized-losses', 'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-
size', 'bore', 'stroke', 'compression-rate', 'horsepower', 'peak-rpm', 'city-mpg', 'highway-mpg', 'price']
numeric_cars = cars[continuous_values_cols]
```

In [1083]:

```
numeric_cars.head(5)
```

Out[1083]:

	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-rate	horsepower	peak-rpm	city-mpg	highway-mpg	price
0	?	88.6	168.8	64.1	48.8	2548	130	3.47	2.68	9.0	111	5000	21	27	13495
1	?	88.6	168.8	64.1	48.8	2548	130	3.47	2.68	9.0	111	5000	21	27	16500
2	?	94.5	171.2	65.5	52.4	2823	152	2.68	3.47	9.0	154	5000	19	26	16500
3	164	99.8	176.6	66.2	54.3	2337	109	3.19	3.40	10.0	102	5500	24	30	13950
4	164	99.4	176.6	66.4	54.3	2824	136	3.19	3.40	8.0	115	5500	18	22	17450

Data Cleaning

We usually shouldn't have any missing values if we want to use them for predictive modeling. We can tell that the normalized-losses column contains missing values represented using "?". Let's replace these values with the numpy.nan missing value and look for the presence of missing values in other numeric columns. Let's also rescale the values in the numeric columns so they all range from 0 to 1.

In [1084]:

```
numeric_cars = numeric_cars.replace('?', np.nan)
numeric_cars.head(5)
```

Out[1084]:

	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-rate	horsepower	peak-rpm	city-mpg	highway-mpg	price
0	NaN	88.6	168.8	64.1	48.8	2548	130	3.47	2.68	9.0	111	5000	21	27	13495
1	NaN	88.6	168.8	64.1	48.8	2548	130	3.47	2.68	9.0	111	5000	21	27	16500
2	NaN	94.5	171.2	65.5	52.4	2823	152	2.68	3.47	9.0	154	5000	19	26	16500
3	164	99.8	176.6	66.2	54.3	2337	109	3.19	3.40	10.0	102	5500	24	30	13950
4	164	99.4	176.6	66.4	54.3	2824	136	3.19	3.40	8.0	115	5500	18	22	17450

In [1085]:

```
numeric_cars = numeric_cars.astype('float')
numeric_cars.isnull().sum()
```

Out[1085]:

```
normalized-losses    41
wheel-base            0
length                0
width                 0
height                0
curb-weight           0
engine-size           0
bore                  4
stroke                4
compression-rate      0
horsepower            2
peak-rpm              2
city-mpg              0
highway-mpg           0
price                 4
dtype: int64
```

In [1086]:

```
# Because `price` is the column we want to predict, let's remove any rows with missing `price` values.
numeric_cars = numeric_cars.dropna(subset=['price'])
numeric_cars.isnull().sum()
```

Out[1086]:

```
normalized-losses    37
wheel-base            0
length                0
width                 0
height                0
curb-weight           0
engine-size           0
bore                  4
stroke                4
compression-rate      0
horsepower            2
peak-rpm              2
city-mpg              0
highway-mpg           0
price                 0
dtype: int64
```

In [1087]:

```
# Replace missing values in other columns using column means.
numeric_cars = numeric_cars.fillna(numeric_cars.mean())
```

In [1088]:

```
# Confirm that there are no more missing values!
numeric_cars.isnull().sum()
```

Out[1088]:

```
normalized-losses    0
wheel-base           0
length               0
width                0
height               0
curb-weight          0
engine-size          0
bore                 0
stroke               0
compression-rate     0
horsepower           0
peak-rpm             0
city-mpg             0
highway-mpg          0
price                0
dtype: int64
```

In [1089]:

```
# Normalize all columnns to range from 0 to 1 except the target column.
price_col = numeric_cars['price']
numeric_cars = (numeric_cars - numeric_cars.min())/(numeric_cars.max() - numeric_cars.min())
numeric_cars['price'] = price_col
```

Univariate Model

Let's begin with some univariate k-nearest neighbors models. Starting with simple models before moving to more complex models helps us structure our code workflow and understand the features better.

A function named knn_train_test() is created that encapsulates the training and simple validation process. The function should have 3 parameters - training column name, target column name, and the dataframe object.

This function should split the data set into a training and test set. Then, it should instantiate the KNeighborsRegressor class, fit the model on the training set, and make predictions on the test set. Finally, it should calculate the root-mean-square error(RMSE) and return that value.

The RMSE is a standard way to measure the error of model in predicting quantitative data. A lower RMSE is better as it indicates there are less differences between the model's predictions when compared to the actual observed values.

In [1090]:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error

def knn_train_test(train_col, target_col, df):
    knn = KNeighborsRegressor()
    np.random.seed(1)

    # Randomize order of rows in data frame.
    shuffled_index = np.random.permutation(df.index)
    rand_df = df.reindex(shuffled_index)

    # Divide number of rows in half and round.
    last_train_row = int(len(rand_df) / 2)

    # Select the first half and set as training set.
    # Select the second half and set as test set.
    train_df = rand_df.iloc[0:last_train_row]
    test_df = rand_df.iloc[last_train_row:]

    # Fit a KNN model using default k value.
    knn.fit(train_df[[train_col]], train_df[target_col])

    # Make predictions using model.
    predicted_labels = knn.predict(test_df[[train_col]])

    # Calculate and return RMSE.
    mse = mean_squared_error(test_df[target_col], predicted_labels)
    rmse = np.sqrt(mse)
    return rmse

rmse_results = {}
train_cols = numeric_cars.columns.drop('price')

# For each column (minus `price`), train a model, return RMSE value
# and add to the dictionary `rmse_results`.
for col in train_cols:
    rmse_val = knn_train_test(col, 'price', numeric_cars)
    rmse_results[col] = rmse_val

# Create a Series object from the dictionary so
# we can easily view the results, sort, etc
rmse_results_series = pd.Series(rmse_results)
rmse_results_series.sort_values()
```

Out[1090]:

```
engine-size          3238.462830
horsepower           4037.037713
curb-weight          4401.118255
highway-mpg          4630.026799
width                4704.482590
city-mpg             4766.422505
length               5427.200961
wheel-base           5461.553998
compression-rate     6610.812153
bore                 6780.627785
normalized-losses    7330.197653
peak-rpm             7697.459696
stroke               8006.529545
height               8144.441043
dtype: float64
```

The knn_train_test() function should be modified to accept different k-parameter values. For each numeric column, we create, train, and test a univariate model using the following k-values: 1, 3, 5, 7 and 9.

In [1091]:

```
def knn_train_test(train_col, target_col, df):
    np.random.seed(1)

    # Randomize order of rows in data frame.
    shuffled_index = np.random.permutation(df.index)
    rand_df = df.reindex(shuffled_index)

    # Divide number of rows in half and round.
    last_train_row = int(len(rand_df) / 2)

    # Select the first half and set as training set.
    # Select the second half and set as test set.
    train_df = rand_df.iloc[0:last_train_row]
    test_df = rand_df.iloc[last_train_row:]

    k_values = [1,3,5,7,9]
    k_rmses = {}

    for k in k_values:
        # Fit model using k nearest neighbors.
        knn = KNeighborsRegressor(n_neighbors=k)
        knn.fit(train_df[[train_col]], train_df[target_col])

        # Make predictions using model.
        predicted_labels = knn.predict(test_df[[train_col]])

        # Calculate and return RMSE.
        mse = mean_squared_error(test_df[target_col], predicted_labels)
        rmse = np.sqrt(mse)

        k_rmses[k] = rmse
    return k_rmses

k_rmse_results = {}

# For each column (minus `price`), train a model, return RMSE value
# and add to the dictionary `rmse_results`.
train_cols = numeric_cars.columns.drop('price')
for col in train_cols:
    rmse_val = knn_train_test(col, 'price', numeric_cars)
    k_rmse_results[col] = rmse_val

k_rmse_results
```

Out[1091]:

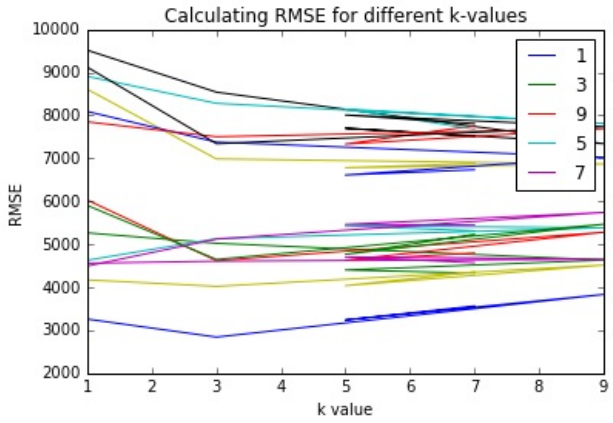

```
{'bore': {1: 8602.58848450066,
  3: 6984.239489480916,
  5: 6780.627784685976,
  7: 6878.097965921532,
  9: 6866.808502038413},
 'city-mpg': {1: 5901.143574354764,
  3: 4646.746408727155,
  5: 4766.422505090134,
  7: 5232.523034167316,
  9: 5465.209492527533},
 'compression-rate': {1: 8087.205346523092,
  3: 7375.063685578359,
  5: 6610.812153159129,
  7: 6732.801282941515,
  9: 7024.485525463435},
 'curb-weight': {1: 5264.290230758878,
  3: 5022.318011757233,
  5: 4401.118254793124,
  7: 4330.608104418053,
  9: 4632.044474454401},
 'engine-size': {1: 3258.4861059962027,
  3: 2840.562805643501,
  5: 3238.4628296477176,
  7: 3563.086774256415,
  9: 3831.8244149840766},
 'height': {1: 8904.04645636071,
  3: 8277.609643045525,
  5: 8144.441042663747,
  7: 7679.598124393773,
  9: 7811.03606291223},
 'highway-mpg': {1: 6025.594966720739,
  3: 4617.305019788554,
  5: 4630.026798588056,
  7: 4796.061440186946,
  9: 5278.358056953987},
 'horsepower': {1: 4170.054848037801,
  3: 4020.8492630885394,
  5: 4037.0377131537603,
  7: 4353.811860277134,
  9: 4515.135617419103},
 'length': {1: 4628.45550121557,
  3: 5129.8358210721635,
  5: 5427.2009608367125,
  7: 5313.427720847974,
  9: 5383.054514833446},
 'normalized-losses': {1: 7846.750605148984,
  3: 7500.5698123109905,
  5: 7330.197653434445,
  7: 7756.421586234123,
  9: 7688.096096891432},
 'peak-rpm': {1: 9511.480067750124,
  3: 8537.550899973421,
  5: 7697.4596964334805,
  7: 7510.294160083481,
  9: 7340.041341263401},
 'stroke': {1: 9116.495955406906,
  3: 7338.68466990294,
  5: 8006.529544647101,
  7: 7803.937796804327,
  9: 7735.554366079291},
 'wheel-base': {1: 4493.734068810494,
  3: 5120.161506064513,
  5: 5461.553997873057,
  7: 5448.1070513823315,
  9: 5738.405685192312},
 'width': {1: 4559.257297950061,
  3: 4606.413692169901,
  5: 4704.482589704386,
  7: 4571.485046194653,
  9: 4652.914172067787}}
```

In [1092]:

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

for k,v in k_rmse_results.items():
    x = list(v.keys())
    y = list(v.values())

    plt.title('Calculating RMSE for different k-values')
    plt.plot(x,y)
    plt.xlabel('k value')
    plt.ylabel('RMSE')
    plt.legend(x)
```



Multivariate Model

Let's modify the knn_train_test() function we wrote previously to work with multiple columns. We modify the knn_train_test() function to accept a list of column names(instead of just a string). We modify the rest of the function logic to use this parameter:

- Instead of using just a single column for train and test, use all of the columns passed in.
- Use a default k-value from scikit-learn for now

In [1093]:

```
# Compute average RMSE across different `k` values for each feature.
feature_avg_rmse = {}
for k,v in k_rmse_results.items():
    avg_rmse = np.mean(list(v.values()))
    feature_avg_rmse[k] = avg_rmse
series_avg_rmse = pd.Series(feature_avg_rmse)
sorted_series_avg_rmse = series_avg_rmse.sort_values()
print(sorted_series_avg_rmse)

sorted_features = sorted_series_avg_rmse.index
```

```
engine-size          3346.484586
horsepower           4219.377860
width                4618.910560
curb-weight          4730.075815
highway-mpg          5069.469256
length               5176.394904
city-mpg             5202.409003
wheel-base           5252.392462
compression-rate     7166.073599
bore                 7222.472445
normalized-losses    7624.407151
stroke               8000.240467
peak-rpm             8119.365233
height               8163.346266
dtype: float64
```

Use the 2, 3, 4, and 5 best features to train and test a multivariate k-nearest neighbors model using the default k-value. Also, display all of the RMSE values.

In [1094]:

```
def knn_train_test(train_cols, target_col, df):
    np.random.seed(1)

    # Randomize order of rows in data frame.
    shuffled_index = np.random.permutation(df.index)
    rand_df = df.reindex(shuffled_index)

    # Divide number of rows in half and round.
    last_train_row = int(len(rand_df) / 2)

    # Select the first half and set as training set.
    # Select the second half and set as test set.
    train_df = rand_df.iloc[0:last_train_row]
    test_df = rand_df.iloc[last_train_row:]

    k_values = [5]
    k_rmses = {}

    for k in k_values:
        # Fit model using k nearest neighbors.
        knn = KNeighborsRegressor(n_neighbors=k)
        knn.fit(train_df[train_cols], train_df[target_col])

        # Make predictions using model.
        predicted_labels = knn.predict(test_df[train_cols])

        # Calculate and return RMSE.
        mse = mean_squared_error(test_df[target_col], predicted_labels)
        rmse = np.sqrt(mse)

        k_rmses[k] = rmse
    return k_rmses

k_rmse_results = {}

for nr_best_feats in range(2,7):
    k_rmse_results['{} best features'.format(nr_best_feats)] = knn_train_test(
        sorted_features[:nr_best_feats],
        'price',
        numeric_cars
    )

k_rmse_results
```

Out[1094]:

```
{'2 best features': {5: 2949.8817277180374},
 '3 best features': {5: 3580.7376651928435},
 '4 best features': {5: 3487.340917327035},
 '5 best features': {5: 3410.2170133901805},
 '6 best features': {5: 3478.510890118539}}
```

Hyperparameter Tuning

Let's now optimize the model that performed the best in the multivariate model. For the top 3 models in the multivariate model, the hyperparameter value is varied from 1 to 25 and the resulting RMSE values are plotted.

We examine which k-value is optimal for each model and plot the resulting RMSE values, and look at the differences in the k-values to account for the differences.

A lower RMSE is better as it indicates there are less differences between the model's predictions when compared to the actual observed values.

In [1095]:

```
def knn_train_test(train_cols, target_col, df):
    np.random.seed(1)

    # Randomize order of rows in data frame.
    shuffled_index = np.random.permutation(df.index)
    rand_df = df.reindex(shuffled_index)

    # Divide number of rows in half and round.
    last_train_row = int(len(rand_df) / 2)

    # Select the first half and set as training set.
    # Select the second half and set as test set.
    train_df = rand_df.iloc[0:last_train_row]
    test_df = rand_df.iloc[last_train_row:]

    k_values = [i for i in range(1, 25)]
    k_rmses = {}

    for k in k_values:
        # Fit model using k nearest neighbors.
        knn = KNeighborsRegressor(n_neighbors=k)
        knn.fit(train_df[train_cols], train_df[target_col])

        # Make predictions using model.
        predicted_labels = knn.predict(test_df[train_cols])

        # Calculate and return RMSE.
        mse = mean_squared_error(test_df[target_col], predicted_labels)
        rmse = np.sqrt(mse)

        k_rmses[k] = rmse
    return k_rmses

k_rmse_results = {}

for nr_best_feats in range(2,6):
    k_rmse_results['{} best features'.format(nr_best_feats)] = knn_train_test(
        sorted_features[:nr_best_feats],
        'price',
        numeric_cars
    )

k_rmse_results
```

Out[1095]:


```
{'2 best features': {1: 2783.6204237227344,
  2: 2657.7963807419765,
  3: 2792.586573031673,
  4: 2891.5329686923255,
  5: 2949.8817277180374,
  6: 3096.402601694776,
  7: 3164.681969020496,
  8: 3413.228359192009,
  9: 3748.6716603306486,
  10: 4080.7125057341937,
  11: 4215.6372280600335,
  12: 4275.421524277872,
  13: 4373.901683035496,
  14: 4424.285137239815,
  15: 4539.505493095937,
  16: 4667.307671446768,
  17: 4729.605305844226,
  18: 4790.556632159094,
  19: 4824.3866193292615,
  20: 4840.850914693829,
  21: 4837.429062000271,
  22: 4831.16988267597,
  23: 4861.679492959275,
  24: 4903.346008862579},
 '3 best features': {1: 3399.8148100410203,
  2: 3497.191103423058,
  3: 3333.6966577570593,
  4: 3355.8842294742026,
  5: 3580.7376651928435,
  6: 3732.943016673517,
  7: 3639.9439408462786,
  8: 3747.4209132113137,
  9: 3986.593913133887,
  10: 4005.354888715163,
  11: 4121.687230061635,
```

```
 12: 4255.700651624227,
 13: 4328.476829895253,
 14: 4332.216494947217,
 15: 4388.225713011904,
 16: 4408.838883583756,
 17: 4404.781029718083,
 18: 4447.577705091259,
 19: 4537.049753345422,
 20: 4592.444230865941,
 21: 4636.731219491763,
 22: 4721.248544133379,
 23: 4787.943506313775,
 24: 4802.894378990491},
'4 best features': {1: 2952.725686581471,
 2: 3131.704952720018,
 3: 3129.692821910155,
 4: 3241.4320776448717,
 5: 3487.340917327035,
 6: 3637.0381471429987,
 7: 3606.195077860286,
 8: 3809.9307026308247,
 9: 3875.274902378068,
 10: 3997.1583055842293,
 11: 4162.564050411074,
 12: 4289.486490995821,
 13: 4368.061602779942,
 14: 4416.304772968801,
 15: 4434.013914355171,
 16: 4441.4634909198785,
 17: 4512.996303789127,
 18: 4523.575629742228,
 19: 4534.834065236792,
 20: 4620.211598150367,
 21: 4688.356509517293,
 22: 4731.46717779913,
 23: 4763.535312989311,
 24: 4751.601375872476},
'5 best features': {1: 2824.7061233282866,
 2: 2915.6731645496975,
 3: 3012.4204546509704,
 4: 3202.8876051367483,
 5: 3410.2170133901805,
 6: 3618.4509432660384,
 7: 3622.6290209234803,
 8: 3848.635835654326,
 9: 3977.8149139381726,
 10: 3994.8132211260104,
 11: 4159.843526607947,
 12: 4294.3389473154875,
 13: 4380.848359486949,
 14: 4466.368754416089,
 15: 4522.420711094978,
 16: 4536.427578452413,
 17: 4587.098443664006,
 18: 4622.107837952761,
 19: 4612.890107622797,
 20: 4632.693976139521,
 21: 4712.917548435062,
 22: 4676.301064518744,
 23: 4691.189310956096,
 24: 4755.990767231825}}
```

In [1096]:

```
for k,v in k_rmse_results.items():
    x = list(v.keys())
    y = list(v.values())
    plt.plot(x,y, label="{}".format(k))

plt.title('Examining which k value is optimal for each model')
plt.xlabel('k value')
plt.ylabel('RMSE')

# reordering the labels
handles, labels = plt.gca().get_legend_handles_labels()

# specify order
order = [2, 1, 3, 0]

# pass handle & labels lists along with order as below
plt.legend([handles[i] for i in order], [labels[i] for i in order], loc='lower right')
```

Out[1096]:

```
<matplotlib.legend.Legend at 0x7fd781374710>
```