K-Nearest Neighbors

Estimated time needed: **25** minutes

Objectives

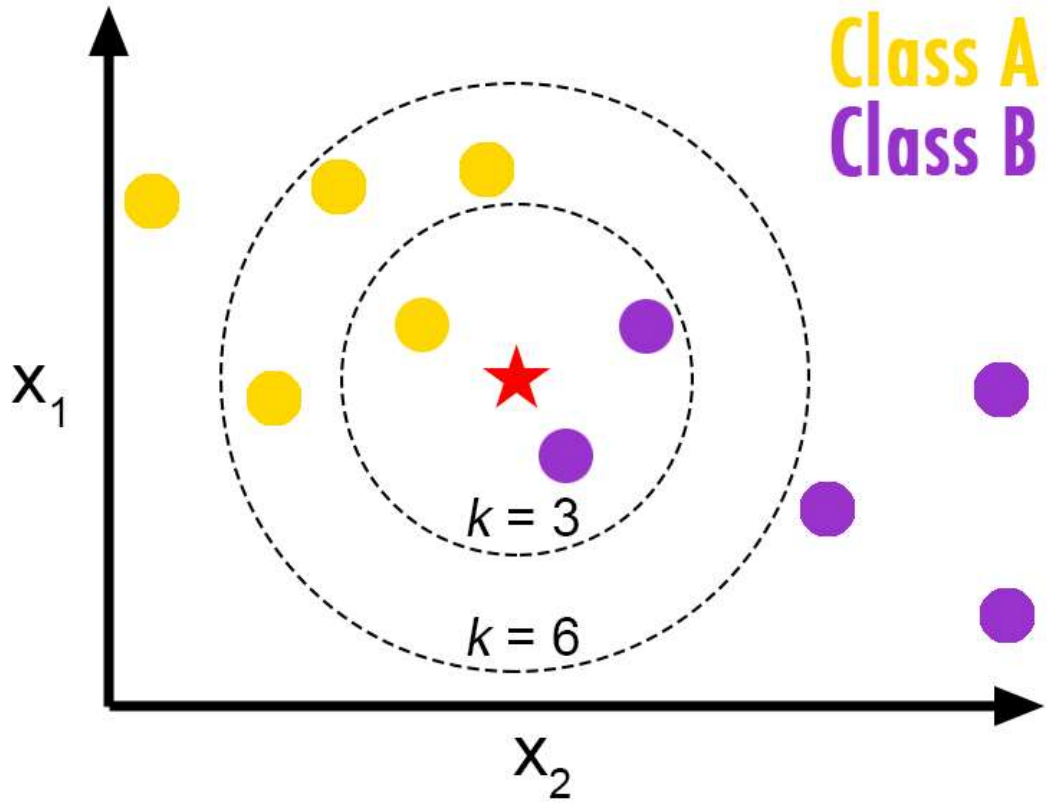
After completing this lab you will be able to:

- Use K Nearest neighbors to classify data

In this Lab you will load a customer dataset, fit the data, and use K-Nearest Neighbors to predict a data point. But what is **K-Nearest Neighbors**?

K-Nearest Neighbors is a supervised learning algorithm. Where the data is 'trained' with data points corresponding to their classification. To predict the class of a given data point, it takes into account the classes of the 'K' nearest data points and chooses the class in which the majority of the 'K' nearest data points belong to as the predicted class.

Here's an visualization of the K-Nearest Neighbors algorithm.



In this case, we have data points of Class A and B. We want to predict what the star (test data point) is. If we consider a k value of 3 (3 nearest data points), we will obtain a prediction of Class B. Yet if we consider a k value of 6, we will obtain a prediction of Class A.

In this sense, it is important to consider the value of k. Hopefully from this diagram, you should get a sense of what the K-Nearest Neighbors algorithm is. It considers the 'K' Nearest Neighbors (data points) when it predicts the classification of the test point.

Table of contents

In [1]:
```
import piplite
await piplite.install(['pandas'])
await piplite.install(['matplotlib'])
await piplite.install(['numpy'])
await piplite.install(['scikit-learn'])
await piplite.install(['scipy'])
```

Let's load required libraries

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn import preprocessing
%matplotlib inline
```

About the dataset

Imagine a telecommunications provider has segmented its customer base by service usage patterns, categorizing the customers into four groups. If demographic data can be used to predict group membership, the company can customize offers for individual prospective customers. It is a classification problem. That is, given the dataset, with predefined labels, we need to build a model to be used to predict class of a new or unknown case.

The example focuses on using demographic data, such as region, age, and marital, to predict usage patterns.

The target field, called **custcat**, has four possible values that correspond to the four customer groups, as follows: 1- Basic Service 2- E-Service 3- Plus Service 4- Total Service

Our objective is to build a classifier, to predict the class of unknown cases. We will use a specific type of classification called K nearest neighbour.

Let's download the dataset.

In [3]:
```
from pyodide.http import pyfetch

async def download(url, filename):
    response = await pyfetch(url)
    if response.status == 200:
        with open(filename, "wb") as f:
            f.write(await response.bytes())
```

In [4]:
```
path="https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDevelop
```

Load Data From CSV File

In [5]:
```
await download(path, 'teleCust1000t.csv')
```

In [6]:
```
df = pd.read_csv('teleCust1000t.csv')
df.head()
```

Out[6]:

	region	tenure	age	marital	address	income	ed	employ	retire	gender	reside	cus
0	2	13	44	1	9	64.0	4	5	0.0	0	2	
1	3	11	33	1	7	136.0	5	5	0.0	0	6	
2	3	68	52	1	24	116.0	1	29	0.0	1	2	
3	2	33	33	0	12	33.0	2	0	0.0	1	1	
4	2	23	30	1	9	30.0	1	2	0.0	0	4	

Data Visualization and Analysis

Let's see how many of each class is in our data set

In [7]:
```
df['custcat'].value_counts()
```

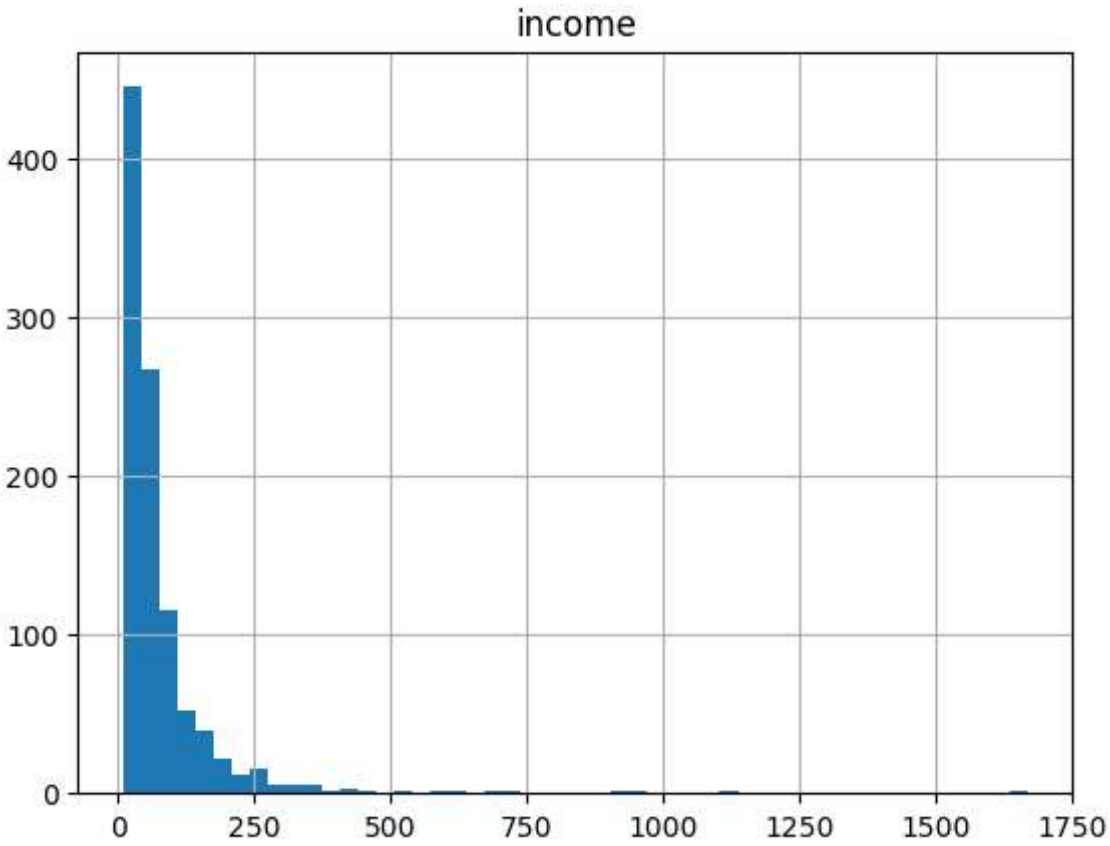
Out[7]:
```
3    281
1    266
4    236
2    217
Name: custcat, dtype: int64
```

281 Plus Service,
266 Basic-service,
236 Total Service and
217 E-Service customers

You can easily explore your data using visualization techniques:

In [8]:
```
df.hist(column='income', bins=50)
```

Out[8]:
```
array([[<AxesSubplot:title={'center':'income'}>]], dtype=object)
```



Feature set

Let's define feature sets, X:

In [9]:
```
df.columns
```

Out[9]:
```
Index(['region', 'tenure', 'age', 'marital', 'address', 'income', 'ed',
       'employ', 'retire', 'gender', 'reside', 'custcat'],
      dtype='object')
```

To use scikit-learn library, we have to convert the Pandas data frame to a Numpy array:

In [25]:
```
X = df[['region', 'tenure','age', 'marital', 'address', 'income', \
        'ed', 'employ','retire', 'gender', 'reside']] .values  #.astype(float)
```

```
X[0:5]
```

Out[25]:
```
array([[  2.,  13.,  44.,   1.,   9.,  64.,   4.,   5.,   0.,   0.,   2.],
       [  3.,  11.,  33.,   1.,   7., 136.,   5.,   5.,   0.,   0.,   6.],
       [  3.,  68.,  52.,   1.,  24., 116.,   1.,  29.,   0.,   1.,   2.],
       [  2.,  33.,  33.,   0.,  12.,  33.,   2.,   0.,   0.,   1.,   1.],
       [  2.,  23.,  30.,   1.,   9.,  30.,   1.,   2.,   0.,   0.,   4.]])
```

What are our labels?

In [11]:
```
y = df['custcat'].values
y[0:5]
```

Out[11]:
```
array([1, 4, 3, 1, 3], dtype=int64)
```

Train Test Split

Out of Sample Accuracy is the percentage of correct predictions that the model makes on data that the model has NOT been trained on. Doing a train and test on the same dataset will most likely have low out-of-sample accuracy, due to the likelihood of our model overfitting.

It is important that our models have a high, out-of-sample accuracy, because the purpose of any model, of course, is to make correct predictions on unknown data. So how can we improve out-of-sample accuracy? One way is to use an evaluation approach called Train/Test Split. Train/Test Split involves splitting the dataset into training and testing sets respectively, which are mutually exclusive. After which, you train with the training set and test with the testing set.

This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is not part of the dataset that has been used to train the model. It is more realistic for the real world problems.

In [12]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_st
print ('Train set:', X_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (800, 11) (800,)
Test set: (200, 11) (200,)
```

The code above is using the `train_test_split` function from the `sklearn.model_selection` module, which is typically used in machine learning for splitting a dataset into training and testing sets. Let me explain each part of the code and its output:

1. `from sklearn.model_selection import train_test_split`: This line imports the `train_test_split` function from the `sklearn.model_selection` module. This

function is used to split a dataset into two subsets, one for training a machine learning model and the other for testing the model's performance.

2. `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=4)` : This line of code performs the actual split of the data. Here's what each part does:

   - `X` and `y` are assumed to be your feature matrix and target variable, respectively.
   - `test_size=0.2` specifies that 20% of the data will be used for testing (i.e., the test set will be 20% of the total dataset).
   - `random_state=4` sets a random seed, ensuring that the same split can be reproduced if needed.

   The function returns four sets of data:

   - `X_train` : The training data features, which will be used to train your machine learning model.
   - `X_test` : The testing data features, which will be used to evaluate your model's performance.
   - `y_train` : The corresponding target values for the training data.
   - `y_test` : The corresponding target values for the testing data.
3. `print('Train set:', X_train.shape, y_train.shape)` : This line prints the shape (number of rows and columns) of the training set. In this case, the training set has 800 rows and 11 columns.

4. `print('Test set:', X_test.shape, y_test.shape)` : This line prints the shape of the testing set. The testing set has 200 rows and 11 columns, which is consistent with the 20% split specified earlier.

So, the output of this code provides you with the dimensions (shape) of both the training and testing sets:

- The training set (`X_train` and `y_train`) has 800 samples (rows) and 11 features (columns).
- The testing set (`X_test` and `y_test`) has 200 samples (rows) and 11 features (columns).

Why does X_train and X_test have rows and columns in shape whereas y_train and y_test doesn't?

The reason `X_train` and `X_test` have rows and columns in their shape while `y_train` and `y_test` do not is due to the nature of the data they represent in the context of machine learning.

1. **`X_train` and `X_test` (Feature Data)**:

- `X_train` and `X_test` represent the feature data, which are the inputs or independent variables used to train and test a machine learning model.
- In a typical machine learning problem, each row in the feature data (`X`) corresponds to a separate data point or observation, and each column corresponds to a feature or attribute of that data point.
- So, `X_train.shape` and `X_test.shape` provide information about the number of data points (rows) and the number of features (columns) in the training and testing sets.

2. **`y_train` and `y_test` (Target Data)**:

   - `y_train` and `y_test` represent the target data, which are the labels or dependent variables associated with the corresponding feature data points.
   - In many machine learning problems, the target data (`y`) is typically represented as a one-dimensional array or vector.
   - Each element of the `y` array corresponds to the target or label for a specific data point in the feature data.
   - The shape of `y_train` and `y_test` is often expressed as `(n,)` , indicating that it is a one-dimensional array. The number `n` represents the total number of target values in the respective sets.

So, in summary:

- `X_train` and `X_test` have a shape of `(number of rows, number of columns)` because they represent the feature data with multiple rows (data points) and multiple columns (features).
- `y_train` and `y_test` have a shape of `(number of elements,)` because they represent the target data as a one-dimensional array with one element per data point.

The difference in their shapes reflects the distinct roles of feature data and target data in machine learning tasks.

Normalize Data

Data Standardization gives the data zero mean and unit variance, it is good practice, especially for algorithms such as KNN which is based on the distance of data points:

In [13]:
```
X_train_norm = preprocessing.StandardScaler().fit(X_train).transform(X_train.astype
X_train_norm[0:5]
```

Out[13]:
```
array([[-1.28618818e+00, -1.53085556e+00, -8.49354628e-01,
         1.00752834e+00, -7.46393214e-01, -4.83608776e-01,
         1.12305195e+00, -7.02283455e-01, -2.23313158e-01,
        -1.01005050e+00,  1.83304333e+00],
       [-4.79805457e-02,  2.51253999e-01,  3.44957012e-01,
         1.00752834e+00,  2.45326718e-01, -2.71543263e-01,
        -5.40728715e-01, -1.25385370e-04, -2.23313158e-01,
        -1.01005050e+00,  1.83304333e+00],
       [-1.28618818e+00, -7.80493639e-01, -6.90113076e-01,
         1.00752834e+00, -5.21892614e-02, -4.92444839e-01,
        -1.37261905e+00, -8.02591750e-01, -2.23313158e-01,
        -1.01005050e+00,  1.83304333e+00],
       [-4.79805457e-02, -1.53085556e+00, -3.71629972e-01,
        -9.92527915e-01,  1.46154725e-01, -4.74772713e-01,
        -1.37261905e+00, -8.02591750e-01, -2.23313158e-01,
         9.90049504e-01, -9.33814526e-01],
       [ 1.19022709e+00, -2.17722200e-01, -6.10492300e-01,
        -9.92527915e-01, -2.50533248e-01,  2.40948394e-01,
         1.12305195e+00, -4.01358568e-01, -2.23313158e-01,
         9.90049504e-01,  1.14132887e+00]])
```

The code above is performing feature scaling on the `X_train` dataset using the `StandardScaler` from scikit-learn's `preprocessing` module. Feature scaling is a common preprocessing step in machine learning to standardize the range of independent variables or features. Let's break down the code and the output:

1. `preprocessing.StandardScaler().fit(X_train)`: This part of the code creates an instance of the `StandardScaler` and fits it to the `X_train` dataset. Fitting means that it computes the mean and standard deviation of each feature in `X_train`, which will be used for scaling.

2. `.transform(X_train.astype(float))`: After fitting the scaler to `X_train`, this part of the code transforms (scales) the `X_train` dataset using the mean and standard deviation computed during fitting. It converts `X_train` to a floating-point type (if it's not already) and then applies the scaling transformation.

3. `X_train_norm[0:5]`: This code prints the first five rows of the scaled `X_train` dataset, which is stored in the variable `X_train_norm`.

Classification

K nearest neighbor (KNN)

Import library

Classifier implementing the k-nearest neighbors vote.

In [14]:
```
from sklearn.neighbors import KNeighborsClassifier
```

Training

Let's start the algorithm with k=4 for now:

In [15]:
```
k = 4
#Train Model and Predict
neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train_norm,y_train)
neigh
```

Out[15]:
```
KNeighborsClassifier(n_neighbors=4)
```

In [16]:
```
X_test_norm = preprocessing.StandardScaler().fit(X_test).transform(X_test.astype(fl
X_test_norm[0:5]
```

Out[16]:
```
array([[-1.13100942, -0.93533648, -0.93817795,  1.02020406, -0.36233308,
         0.17102716, -0.58336825,  0.00666863, -0.21707238, -1.13967126,
         1.97590703],
       [-1.13100942, -1.44823286, -0.13972863, -0.98019606,  0.03474427,
        -0.50017376, -0.58336825, -0.56492821, -0.21707238, -1.13967126,
        -0.90337462],
       [-1.13100942, -0.51569399,  1.53701494,  1.02020406, -0.56087175,
         0.45499678,  0.9619781 , -0.75546048, -0.21707238,  0.87744601,
         1.25608662],
       [-1.13100942, -0.74882871, -0.85833302,  1.02020406, -1.15648777,
        -0.70669712,  0.9619781 , -0.85072662, -0.21707238, -1.13967126,
         0.53626621],
       [-1.13100942,  0.51009876, -1.01802288,  1.02020406, -0.75941043,
         0.17102716,  1.73465128, -0.75546048, -0.21707238, -1.13967126,
         1.25608662]])
```

In the above, you are using the k-Nearest Neighbors (k-NN) algorithm for classification. Let's break down the code and what each part does:

1. `k = 4`: You are setting the value of `k` to 4, which means that the k-NN algorithm will consider the 4 nearest neighbors when making predictions.

2. `neigh = KNeighborsClassifier(n_neighbors=k).fit(X_train_norm, y_train)`: In this line, you are creating an instance of the `KNeighborsClassifier` with `n_neighbors` set to the value of `k` (which is 4). Then, you are fitting the model to the normalized training data `X_train_norm` and the corresponding target labels `y_train`. This step trains the k-NN classifier on your training data.

3. `X_test_norm = preprocessing.StandardScaler().fit(X_test).transform(X_test.astype(float))`: You are standardizing (scaling) the test data `X_test` using the mean and standard

deviation computed from the training data. This ensures that the test data is on the same scale as the training data, which is essential for making predictions with k-NN.

4. `X_test_norm[0:5]`: This code displays the first five rows of the scaled test data `X_test_norm`. It shows the standardized feature values for the first five data points in your test dataset.

After running this code, you have trained a k-NN classifier with `k = 4` using the normalized training data, and you have also standardized the test data for prediction. The `neigh` variable now holds the trained k-NN model, which you can use to make predictions on new data points by passing them to the `neigh.predict()` method.

If you have any further questions or need assistance with the next steps in your machine learning project, please feel free to ask.

Predicting

We can use the model to make predictions on the test set:

In [17]:
```
yhat = neigh.predict(X_test_norm)
yhat[0:5]
```

Out[17]:
```
array([3, 1, 3, 2, 4], dtype=int64)
```

Accuracy evaluation

In multilabel classification, **accuracy classification score** is a function that computes subset accuracy. This function is equal to the jaccard_score function. Essentially, it calculates how closely the actual labels and predicted labels are matched in the test set.

In [18]:
```
from sklearn import metrics
print("Train set Accuracy: ", metrics.accuracy_score(y_train, neigh.predict(X_train
print("Test set Accuracy: ", metrics.accuracy_score(y_test, yhat))
```

```
Train set Accuracy:  0.5475
Test set Accuracy:  0.33
```

Practice

Can you build the model again, but this time with k=6?

In [31]:
```
# write your code here
k6=6
neigh6 = KNeighborsClassifier(n_neighbors = k6).fit(X_train_norm,y_train)
# yhat6 = neigh.predict(X_test_norm)
print("Train set accuracy:",metrics.accuracy_score(y_train,neigh6.predict(X_train_n
print("Test set accuracy:",metrics.accuracy_score(y_test,neigh6.predict(X_test_norm
```

```
Train set accuracy: 0.5125
Test set accuracy: 0.335
```

▶ Click here for the solution

What about other K?

K in KNN, is the number of nearest neighbors to examine. It is supposed to be specified by the user. So, how can we choose right value for K? The general solution is to reserve a part of your data for testing the accuracy of the model. Then choose k =1, use the training part for modeling, and calculate the accuracy of prediction using all samples in your test set. Repeat this process, increasing the k, and see which k is the best for your model.

We can calculate the accuracy of KNN for different values of k.

In [20]:
```
Ks = 10
mean_acc = np.zeros((Ks-1))
std_acc = np.zeros((Ks-1))

for n in range(1,Ks):

    #Train Model and Predict
    neigh = KNeighborsClassifier(n_neighbors = n).fit(X_train_norm,y_train)
    yhat=neigh.predict(X_test_norm)
    mean_acc[n-1] = metrics.accuracy_score(y_test, yhat)


    std_acc[n-1]=np.std(yhat==y_test)/np.sqrt(yhat.shape[0])

mean_acc
```

Out[20]:
```
array([0.32 , 0.315, 0.335, 0.33 , 0.34 , 0.335, 0.35 , 0.34 , 0.335])
```

The code above is calculating the mean accuracy of a k-NN classifier for different values of `n` (the number of neighbors) and storing those accuracies in the `mean_acc` array. Let's break down what this code does step by step:

1. `Ks = 10`: You've set `Ks` to 10, indicating that you want to iterate through values of `n` from 1 to 9 (inclusive).
2. `mean_acc = np.zeros((Ks-1))` and `std_acc = np.zeros((Ks-1))`: These lines initialize two NumPy arrays, `mean_acc` and `std_acc`, both with a length of `Ks-1`. These arrays will store the mean accuracy and standard deviation of accuracy for each value of `n`.
3. `for n in range(1, Ks)`: This loop iterates through values of `n` from 1 to 9 (inclusive).
4. `neigh = KNeighborsClassifier(n_neighbors = n).fit(X_train_norm, y_train)`: Inside the loop, you create a k-NN classifier with `n_neighbors` set to the

current value of `n`. Then, you fit the classifier to the normalized training data `X_train_norm` and the corresponding training labels `y_train`.

5. `yhat = neigh.predict(X_test_norm)`: After training the classifier, you use it to make predictions on the normalized test data `X_test_norm`, and store the predictions in `yhat`.

6. `mean_acc[n-1] = metrics.accuracy_score(y_test, yhat)`: You calculate the accuracy score by comparing the predicted values `yhat` to the true labels `y_test` for the test set, and store the result in the `mean_acc` array at the corresponding index (adjusted by -1 to match Python's 0-based indexing).

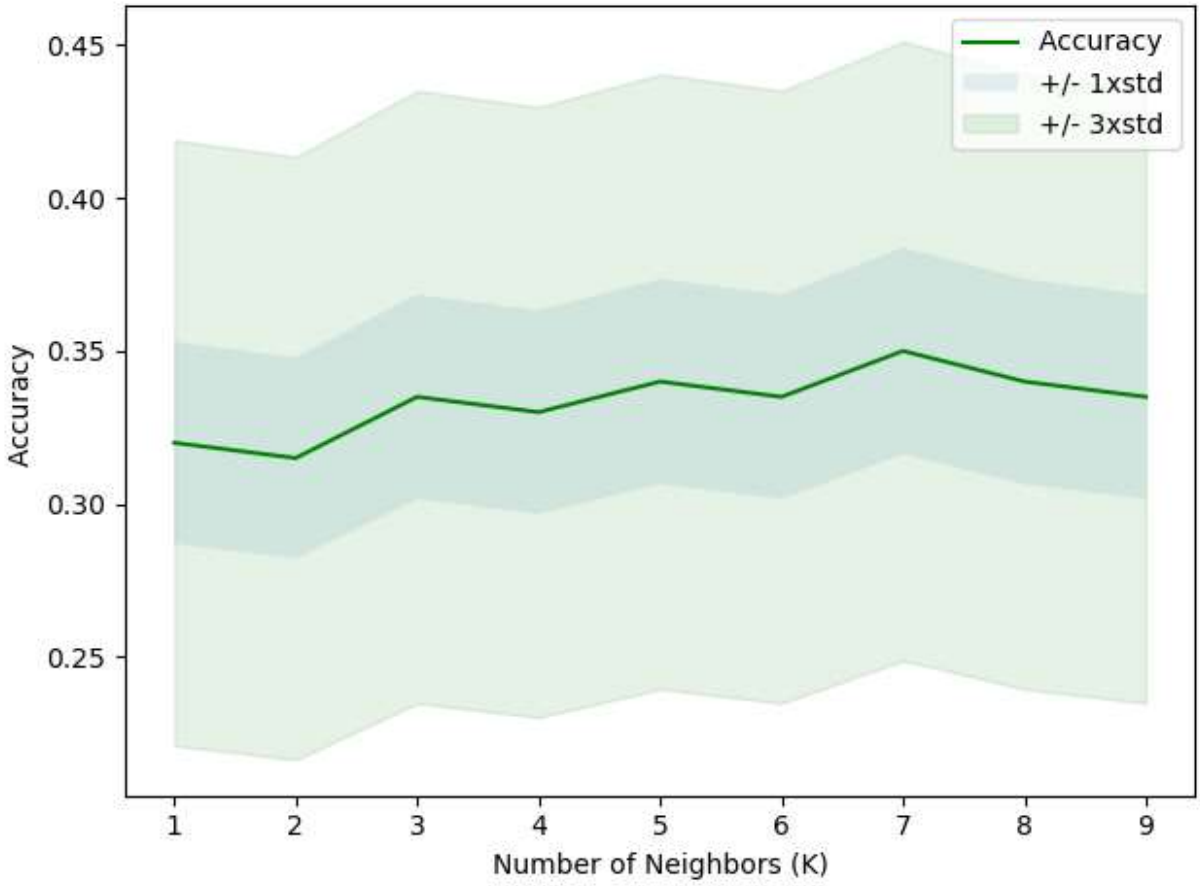
7. `std_acc[n-1] = np.std(yhat == y_test) / np.sqrt(yhat.shape[0])`: You calculate the standard deviation of accuracy by comparing the predicted values `yhat` to the true labels `y_test`, and then divide by the square root of the number of samples in `yhat`. This is often used to estimate the standard error of the accuracy.

8. Finally, `mean_acc` contains the mean accuracy values for different values of `n`.

This code allows you to evaluate and compare the performance of the k-NN classifier for different values of `n` and provides insights into how the choice of the number of neighbors affects model accuracy.

Plot the model accuracy for a different number of neighbors.

In [21]:
```
plt.plot(range(1,Ks),mean_acc,'g')
plt.fill_between(range(1,Ks),mean_acc - 1 * std_acc,mean_acc + 1 * std_acc, alpha=0
plt.fill_between(range(1,Ks),mean_acc - 3 * std_acc,mean_acc + 3 * std_acc, alpha=0
plt.legend(('Accuracy ', '+/- 1xstd','+/- 3xstd'))
plt.ylabel('Accuracy ')
plt.xlabel('Number of Neighbors (K)')
plt.tight_layout()
plt.show()
```



The code above is related to plotting the accuracy of a k-NN classifier with different values of k. Let's break down what this code does step by step:

1. `plt.plot(range(1, Ks), mean_acc, 'g')` : This line of code creates a line plot. It plots the values of `mean_acc` (which presumably contains the mean accuracy scores for different values of k) against a range of values from 1 to `Ks` (10). The `'g'` parameter specifies that the line color is green.

2. `plt.fill_between(range(1, Ks), mean_acc - 1 * std_acc, mean_acc + 1 * std_acc, alpha=0.10)` : This line of code is used to fill the area between two horizontal lines, which are `mean_acc - 1 * std_acc` and `mean_acc + 1 * std_acc` , for each value of k in the specified range. It's often used to represent the standard deviation of accuracy. The `alpha=0.10` parameter controls the transparency of the fill.

3. `plt.fill_between(range(1, Ks), mean_acc - 3 * std_acc, mean_acc + 3 * std_acc, alpha=0.10, color="green")` : This line does the same as the previous one but with a larger range, filling the area between `mean_acc - 3 * std_acc` and `mean_acc + 3 * std_acc` . It's also colored green.

4. `plt.legend(('Accuracy', '+/- 1xstd', '+/- 3xstd'))` : This line adds a legend to the plot, labeling the lines. It explains that the green line represents accuracy, the

light green area represents accuracy within one standard deviation, and the lighter green area represents accuracy within three standard deviations.

5. `plt.ylabel('Accuracy')` and `plt.xlabel('Number of Neighbors (K)')`: These lines set labels for the y-axis and x-axis of the plot, respectively.

6. `plt.tight_layout()`: This function adjusts the subplot parameters to make sure everything fits nicely in the figure.

7. `plt.show()`: This line displays the plot to the user.

Overall, this code is used to visualize how the accuracy of a k-NN classifier changes as you vary the number of neighbors (`Ks`). It provides insights into how different values of `K` affect the model's performance, with shaded regions indicating the level of uncertainty due to standard deviation. It's a useful visualization for model selection and tuning.

In [22]:
```
print( "The best accuracy was with", mean_acc.max(), "with k=", mean_acc.argmax()+1
```

```
The best accuracy was with 0.35 with k= 7
```