In this example we will look at how to use the K-Nearest_Neighbor algorithm for classification. We will use a modified version of the Video Store data set for this example. We will use the "Incidentals" attribute as the target attribute for classification (the class attribute). The goal is to be able to classify an unseen instance as "Yes" or "No" given the values of "Incidentals" from training instances.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [3]:
```
vstable = pd.read_csv("Video_Store_2.csv", index_col=0)

vstable.shape
```

Out[3]:
```
(50, 7)
```

In [4]:
```
vstable.head()
```

Out[4]:

Cust ID	Gender	Income	Age	Rentals	Avg Per Visit	Genre	Incidentals
1	M	45000	25	32	2.5	Action	Yes
2	F	54000	33	12	3.4	Drama	No
3	F	32000	20	42	1.6	Comedy	No
4	F	59000	70	16	4.2	Drama	Yes
5	M	37000	35	25	3.2	Action	Yes

In [5]:
```
vs_names = vstable.columns.values
vs_names
```

Out[5]:
```
array(['Gender', 'Income', 'Age', 'Rentals', 'Avg Per Visit', 'Genre',
       'Incidentals'], dtype=object)
```

We will be splitting the data into a test and training partitions with the test partition to be used for evaluating model error-rate and the training partition to be used to find the K nearest neighbors. Before spliting the data we need to do a random reshuffling to make sure the instances are randomized.

In [6]:
```
vs = vstable.reindex(np.random.permutation(vstable.index))
vs.head(10)
```

Out[6]:

Cust ID	Gender	Income	Age	Rentals	Avg Per Visit	Genre	Incidentals
46	F	57000	43	9	1.1	Drama	No
48	F	52000	47	14	1.6	Drama	No
44	M	35000	24	24	1.7	Drama	No
47	F	69000	35	22	2.8	Drama	Yes
5	M	37000	35	25	3.2	Action	Yes
27	F	62000	47	32	3.6	Drama	No
24	F	79000	35	22	3.8	Drama	Yes
8	M	74000	25	31	2.4	Action	Yes
50	M	24000	20	33	4.7	Action	No
17	M	36000	35	28	3.5	Drama	Yes

In [7]:
```
len(vs)
```

Out[7]:
```
50
```

The target attribute for classification is Incidentals. These are the class labels (in this case "yes" and "no") corresponding to instances in the data.

In [8]:
```
vs_target = vs.Incidentals
```

Before we can compute distances we need to convert the data (excluding the target attribute "incidentals" which contains the class labels) into standard spreadsheet format with binary dummy variables created for each categorical attribute.

In [9]:
```
vs = pd.get_dummies(vs[['Gender','Income','Age','Rentals','Avg Per Visit','Genre']])
```

```
vs.head(10)
```

Out[9]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genre_Drama
46	57000	43	9	1.1	True	False	False	False	True
48	52000	47	14	1.6	True	False	False	False	True
44	35000	24	24	1.7	False	True	False	False	True
47	69000	35	22	2.8	True	False	False	False	True
5	37000	35	25	3.2	False	True	True	False	False
27	62000	47	32	3.6	True	False	False	False	True
24	79000	35	22	3.8	True	False	False	False	True
8	74000	25	31	2.4	False	True	True	False	False
50	24000	20	33	4.7	False	True	True	False	False
17	36000	35	28	3.5	False	True	False	False	True

To be able to evaluate the accuracy of our predictions, we will split the data into training and test sets. In this case, we will use 80% for training and the remaining 20% for testing. Note that we must also do the same split to the target attribute.

In [10]:
```
tpercent = 0.8
tsize = int(tpercent * len(vs))
vs_train = vs[:tsize]
vs_test = vs[tsize:]
```

In [11]:
```
print(vs_train.shape)
print(vs_test.shape)
```

```
(40, 9)
(10, 9)
```

In [12]:
```
vs_train.head(10)
```

Out[12]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genre_Drama
46	57000	43	9	1.1	True	False	False	False	True
48	52000	47	14	1.6	True	False	False	False	True
44	35000	24	24	1.7	False	True	False	False	True
47	69000	35	22	2.8	True	False	False	False	True
5	37000	35	25	3.2	False	True	True	False	False
27	62000	47	32	3.6	True	False	False	False	True
24	79000	35	22	3.8	True	False	False	False	True
8	74000	25	31	2.4	False	True	True	False	False
50	24000	20	33	4.7	False	True	True	False	False
17	36000	35	28	3.5	False	True	False	False	True

In [13]:
```
vs_test
```

Out[13]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genre_Drama
22	25000	33	16	2.9	False	True	False	False	True
18	6000	16	39	1.8	True	False	True	False	False
42	32000	25	26	2.2	False	True	True	False	False
49	31000	25	42	3.4	False	True	True	False	False
38	41000	38	20	3.3	False	True	False	False	True
4	59000	70	16	4.2	True	False	False	False	True
25	1000	16	25	1.4	False	True	False	True	False
26	56000	35	40	2.6	True	False	True	False	False
28	57000	52	22	4.1	False	True	False	True	False
11	41000	22	48	2.3	True	False	False	False	True

Splitting the target attribute ("Incidentals") accordingly:

In [14]:
```
vs_target_train = vs_target[0:int(tsize)]
vs_target_test = vs_target[int(tsize):len(vs)]
```

In [15]:
```
vs_target_train.head()
```

Out[15]:
```
Cust ID
46     No
48     No
44     No
47    Yes
5     Yes
Name: Incidentals, dtype: object
```

In [16]:
```
vs_target_test
```

Out[16]:
```
Cust ID
22    Yes
18    Yes
42    Yes
49    Yes
38    Yes
4     Yes
25    Yes
26    Yes
28     No
11    Yes
Name: Incidentals, dtype: object
```

Next, we normalize the attributes so that everything is in [0,1] scale. We can use the normalization functions we developed in earlier examples. In this case, however, we will use the more flexible and robust scaler function from the preprocessing module of scikit-learn package. *important Note:* ***we train the scaler on the training portion of the data only. Then we use the scaler to transform (normalize) both the training and then test partitions***

In [17]:
```
from sklearn import preprocessing
```

In [18]:
```
# Fit the scaler to the training data
min_max_scaler = preprocessing.MinMaxScaler()
min_max_scaler.fit(vs_train)
```

Out[18]:
```
▾ MinMaxScaler
MinMaxScaler()
```

In [19]:
```
# Use the scaler to transfrom both training and test data sets
vs_train_norm = min_max_scaler.fit_transform(vs_train)
vs_test_norm = min_max_scaler.fit_transform(vs_test)
```

Note that MinMaxScaler returns a Numpy nd-array).

In [20]:
```
np.set_printoptions(precision=2, linewidth=100)
```

```
print(vs_train_norm[:10])
```

```
[[0.63 0.68 0.   0.   1.   0.   0.   0.   1.  ]
 [0.57 0.78 0.13 0.14 1.   0.   0.   0.   1.  ]
 [0.38 0.22 0.38 0.17 0.   1.   0.   0.   1.  ]
 [0.77 0.49 0.33 0.47 1.   0.   0.   0.   1.  ]
 [0.4  0.49 0.41 0.58 0.   1.   1.   0.   0.  ]
 [0.69 0.78 0.59 0.69 1.   0.   0.   0.   1.  ]
 [0.89 0.49 0.33 0.75 1.   0.   0.   0.   1.  ]
 [0.83 0.24 0.56 0.36 0.   1.   1.   0.   0.  ]
 [0.25 0.12 0.62 1.   0.   1.   1.   0.   0.  ]
 [0.39 0.49 0.49 0.67 0.   1.   0.   0.   1.  ]]
```

In [21]:
```
print(vs_test_norm[:10])
```

```
[[0.41 0.31 0.   0.54 0.   1.   0.   0.   1.  ]
 [0.09 0.   0.72 0.14 1.   0.   1.   0.   0.  ]
 [0.53 0.17 0.31 0.29 0.   1.   1.   0.   0.  ]
 [0.52 0.17 0.81 0.71 0.   1.   1.   0.   0.  ]
 [0.69 0.41 0.12 0.68 0.   1.   0.   0.   1.  ]
 [1.   1.   0.   1.   1.   0.   0.   0.   1.  ]
 [0.   0.   0.28 0.   0.   1.   0.   1.   0.  ]
 [0.95 0.35 0.75 0.43 1.   0.   1.   0.   0.  ]
 [0.97 0.67 0.19 0.96 0.   1.   0.   1.   0.  ]
 [0.69 0.11 1.   0.32 1.   0.   0.   0.   1.  ]]
```

For consitency, we'll also convert the training and test target labels into Numpy arrays.

In [22]:
```
vs_target_train = np.array(vs_target_train)
vs_target_test = np.array(vs_target_test)
```

In [23]:
```
print(vs_target_train)
print("\n")
print(vs_target_test)
```

```
['No' 'No' 'No' 'Yes' 'Yes' 'No' 'Yes' 'Yes' 'No' 'Yes' 'No' 'Yes' 'No' 'Yes' 'No' 'No' 'No' 'Yes'
 'No' 'No' 'No' 'Yes' 'No' 'Yes' 'No' 'Yes' 'Yes' 'No' 'Yes' 'Yes' 'No' 'Yes' 'Yes' 'No' 'No' 'No'
 'No' 'No' 'Yes' 'No']


['Yes' 'Yes' 'Yes' 'Yes' 'Yes' 'Yes' 'Yes' 'Yes' 'No' 'Yes']
```

The following function illustrates how we can perform a k-nearest-neighbor search. It takes an instance x (a row in the test data) to be classifed and a data matrix D (assumed to be a 2d Numpy array) as inputs. It also takes K (the desired number of nearest-neighbors to be identified), and "measure" as arguments. The "measure" argument allows us to use either Euclidean distance (measure=0) or (**the inverse of**) Cosine similarity (measure = 1) as the distance function:

In [24]:
```
def knn_search(x, D, K, measure):
    """ find K nearest neighbors of an instance x among the instances in D """
    if measure == 0:
        # euclidean distances from the other points
        dists = np.sqrt(((D - x)**2).sum(axis=1))
    elif measure == 1:
        # first find the vector norm for each instance in D as wel as the norm for vector x
        D_norm = np.array([np.linalg.norm(D[i]) for i in range(len(D))])
        x_norm = np.linalg.norm(x)
        # Compute Cosine: divide the dot product o x and each instance in D by the product of the two norms
        sims = np.dot(D,x)/(D_norm * x_norm)
        # The distance measure will be the inverse of Cosine similarity
        dists = 1 - sims
    idx = np.argsort(dists) # sorting
    # return the indexes of K nearest neighbors
    return idx[:K], dists
```

To test our function, we'll use the first instance in the test data as the instance x as input to the knn_search function and find its K nearest neighbors in the training data.

In [25]:
```
# neigh_idx, distances = knn_search(vs_test_norm[0], vs_train_norm, 5, 1)
```

In [26]:
```
# We'll use K = 5 and Euclidean distance for this example
neigh_idx, distances = knn_search(vs_test_norm[0], vs_train_norm, 5, 0)
```

In [27]:
```
# This was the (non-normalized version of the) test instance used
vs_test.head(1)
```

Out[27]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genre_Drama
22	25000	33	16	2.9	False	True	False	False	True

In [28]:
```
# Let's show the indexes of the 5 nearest neighbors and the neirghbors, themselves, in the original training da

print(neigh_idx)
print("\nNearest Neigbors:")
vs_train.iloc[neigh_idx]
```

```
[17 24  9  2 11]

Nearest Neigbors:
```

Out[28]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genre_Drama
32	47000	30	21	3.1	False	True	False	False	True
14	45000	36	24	2.7	False	True	False	False	True
17	36000	35	28	3.5	False	True	False	False	True
44	35000	24	24	1.7	False	True	False	False	True
45	56000	38	30	3.5	False	True	False	False	True

In [29]:
```
# And here are the distances of the above neighbors to the test instance
print(distances[neigh_idx])
```

```
[0.33 0.45 0.53 0.54 0.64]
```

In [30]:
```
# Let's see how the nearest neighbors of the test instance labeled the target attribute "incidentals"

neigh_labels = vs_target_train[neigh_idx]
print(neigh_labels)
```

```
['Yes' 'No' 'Yes' 'No' 'Yes']
```

Now that we know the nearest neighbors, we need to find the majority class label among them. The majority class would be the class assgined to the new instance x.

In [31]:
```
from collections import Counter
print(Counter(neigh_labels))
```

```
Counter({'Yes': 3, 'No': 2})
```

In [32]:
```
Counter(neigh_labels).most_common(1)
```

Out[32]:
```
[('Yes', 3)]
```

Let's now put everything together into a function that calls our knn_search function and then returns that majority class among the K nearest neighbors of the instance to be classified. This is our "classifier function."

In [33]:
```
def knn_classify(x, D, K, labels, measure):
    from collections import Counter
    neigh_idx, distances = knn_search(x, D, K, measure)
    neigh_labels = labels[neigh_idx]
    count = Counter(neigh_labels)
    print("Labels for top ", K, "neighbors: ", count.most_common())
    return count.most_common(1)[0][0]
```

We can now use our KNN classifer to evaluate it's classification accuracy. Here we will run the classifier on each test instance in our test data and compare the predicted class to the actual class for each. We will maintain the number of disagreements which allows us to compute the final error rate across all test instances.

In [34]:
```
numTestVecs = len(vs_target_test)
print(numTestVecs)
```

```
10
```

In [35]:
```
errorCount = 0.0
for i in range(numTestVecs):
    classifierResult = knn_classify(vs_test_norm[i,:], vs_train_norm, 5, vs_target_train, 0)
    print("Predicted Label: ", classifierResult, "==> Actual Label: ", vs_target_test[i])
    print()
    if (classifierResult != vs_target_test[i]):
        errorCount += 1.0

print("Classification Accuracy: ", 1 - (errorCount/float(numTestVecs)))
```

```
Labels for top  5 neighbors:  [('Yes', 3), ('No', 2)]
Predicted Label:  Yes ==> Actual Label:  Yes

Labels for top  5 neighbors:  [('Yes', 3), ('No', 2)]
Predicted Label:  Yes ==> Actual Label:  Yes

Labels for top  5 neighbors:  [('Yes', 4), ('No', 1)]
Predicted Label:  Yes ==> Actual Label:  Yes

Labels for top  5 neighbors:  [('Yes', 3), ('No', 2)]
Predicted Label:  Yes ==> Actual Label:  Yes

Labels for top  5 neighbors:  [('Yes', 3), ('No', 2)]
Predicted Label:  Yes ==> Actual Label:  Yes

Labels for top  5 neighbors:  [('No', 4), ('Yes', 1)]
Predicted Label:  No ==> Actual Label:  Yes

Labels for top  5 neighbors:  [('No', 3), ('Yes', 2)]
Predicted Label:  No ==> Actual Label:  Yes

Labels for top  5 neighbors:  [('Yes', 5)]
Predicted Label:  Yes ==> Actual Label:  Yes

Labels for top  5 neighbors:  [('No', 4), ('Yes', 1)]
Predicted Label:  No ==> Actual Label:  No

Labels for top  5 neighbors:  [('Yes', 3), ('No', 2)]
Predicted Label:  Yes ==> Actual Label:  Yes

Classification Accuracy:  0.8
```

Let's put this evaluation code into a function that we can resuse easily with different parameters of KNN classifier. We'll also create a new version of the classifier without the extraneous output which returns predicted label and the top K neighbors.

In [36]:
```
def knn_classify(x, D, K, labels, measure):
    from collections import Counter
    neigh_idx, distances = knn_search(x, D, K, measure)
    neigh_labels = labels[neigh_idx]
    count = Counter(neigh_labels)
    # print("Labels for top", K, "neighbors: ", count)
    predicted_label = count.most_common(1)[0][0]
    return neigh_idx, predicted_label
```

In [37]:
```
def knn_evaluate(test, test_labs, train, train_labs, K, measure):
    # Inputs:
    # test: an array or list of test instances
    # test_labs: an array or list of class labels for the corresponding test instances in test
    # train: the training instances
    # train_labs: class labels for the corresponding training instances in train
    # K: number of neighbors
    # measure: 0 = Euclidean distance; 1 = Cosine distance

    T=0 # no. of correctly classified instances
    F=0 # no. of incorrectly classified instances
    for i in range(len(test)):
        actual=test_labs[i]
        top_K_neighbors, predicted = knn_classify(test[i], train, K, train_labs, measure)
        if actual == predicted:
            T += 1
        else:
            F += 1
    accuracy = float(T)/float(T+F)
    return accuracy
```

In [38]:
```
# Testing the evaluation function with K =5 and Euclidean distance on the full test set
accuracy = knn_evaluate(vs_test_norm, vs_target_test, vs_train_norm, vs_target_train, 5, 0)

print("Classification Accuracy: ", accuracy)
```

```
Classification Accuracy:  0.8
```

In [39]:
```
# Let's compare this to the accuracy on the training data, itself
accuracy = knn_evaluate(vs_train_norm, vs_target_train, vs_train_norm, vs_target_train, 5, 0)

print("Classification Accuracy (Train): ", accuracy)
```

```
Classification Accuracy (Train):  0.8
```

Let's repeat with the distance metric based on Cosine similarity (instead of Euclidean distance):

In [40]:
```
# Testing the evaluation function with K =5 and Cosine distance on the full test set
```

```
accuracy = knn_evaluate(vs_test_norm, vs_target_test, vs_train_norm, vs_target_train, 5, 1)

print("Classification Accuracy: ", accuracy)
```

```
Classification Accuracy:  0.8
```

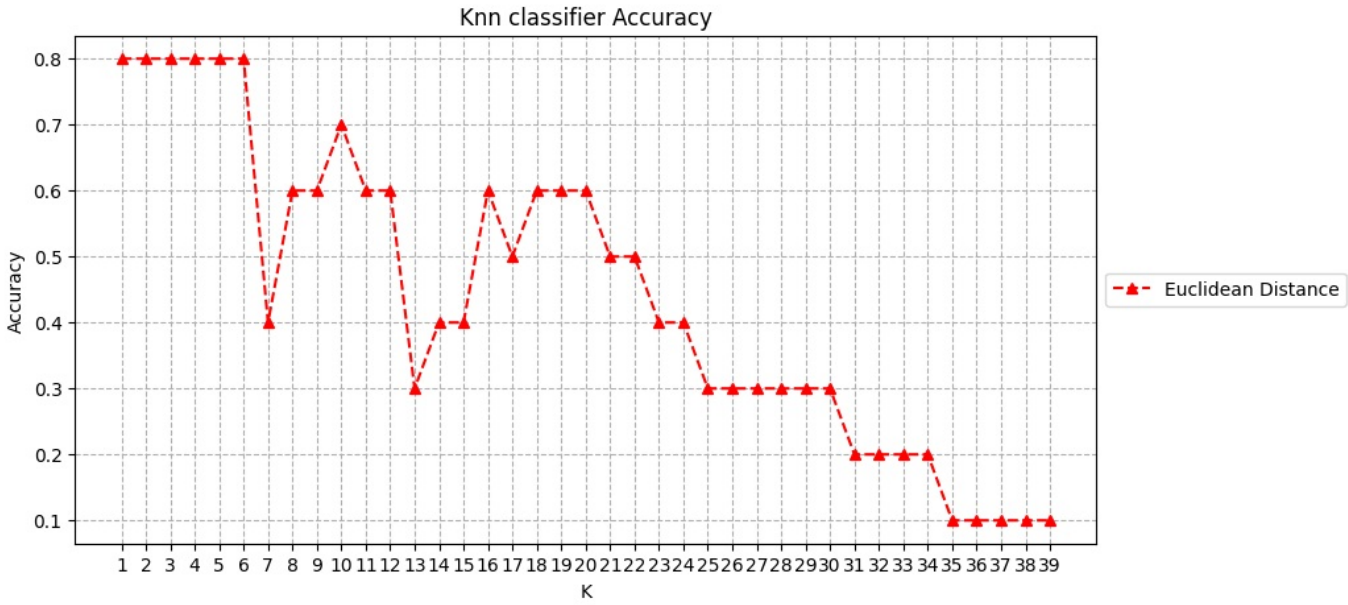
In [41]:
```
Euclid=[]
for K in range(1, 40):
    Euclid.append(knn_evaluate(vs_test_norm, vs_target_test, vs_train_norm, vs_target_train, K, 0))

print(Euclid)
```

```
[0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.4, 0.6, 0.6, 0.7, 0.6, 0.6, 0.3, 0.4, 0.4, 0.6, 0.5, 0.6, 0.6, 0.6, 0.5, 0.5, 0
.4, 0.4, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.2, 0.2, 0.2, 0.2, 0.1, 0.1, 0.1, 0.1, 0.1]
```

In [42]:
```
Ks=list(range(1, 40))
plt.figure(figsize=(10,5))
plt.plot(Ks, Euclid, 'r^--', label='Euclidean Distance')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.title('Knn classifier Accuracy')
plt.grid(linestyle='--')
plt.xticks(Ks)
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```



A better way to split the data into training and test sets

In [43]:
```
from sklearn.model_selection import train_test_split
vs_train2, vs_test2, vs_target_train2, vs_target_test2 = train_test_split(vs, vs_target, test_size=0.2, random_s

print (vs_test2.shape)
print (vs_train2.shape)
```

```
(10, 9)
(40, 9)
```

In [44]:
```
vs_test2
```

Out[44]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genre_Drama
3	32000	20	42	1.6	True	False	False	True	False
24	79000	35	22	3.8	True	False	False	False	True
26	56000	35	40	2.6	True	False	True	False	False
19	24000	25	41	3.1	True	False	False	True	False
27	62000	47	32	3.6	True	False	False	False	True
38	41000	38	20	3.3	False	True	False	False	True
33	23000	25	28	2.7	False	True	True	False	False
10	65000	40	21	3.3	True	False	False	False	True
22	25000	33	16	2.9	False	True	False	False	True
32	47000	30	21	3.1	False	True	False	False	True

In [45]:
```
vs_target_test2
```

Out[45]:
```
Cust ID
3      No
24    Yes
26    Yes
19     No
27     No
38    Yes
33     No
10     No
22    Yes
32    Yes
Name: Incidentals, dtype: object
```

In [46]:
```
vs_train2.head(10)
```

Out[46]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genre_Drama
1	45000	25	32	2.5	False	True	True	False	False
35	74000	29	43	4.6	False	True	True	False	False
6	18000	20	29	1.7	False	True	True	False	False
20	12000	16	23	2.2	False	True	True	False	False
45	56000	38	30	3.5	False	True	False	False	True
44	35000	24	24	1.7	False	True	False	False	True
13	83000	46	14	3.6	False	True	False	True	False
21	47000	52	11	3.1	True	False	False	False	True
18	6000	16	39	1.8	True	False	True	False	False
9	38000	21	18	2.1	False	True	False	True	False