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
```

In [2]:
```
vstable = pd.read_csv("http://facweb.cs.depaul.edu/mobasher/classes/csc478/c

vstable.shape
```

Out[2]:
```
(50, 7)
```

In [3]:
```
vstable.head()
```

Out[3]:

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

We will be splitting the data into a test and training partions 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 [4]:
```
vs = vstable.reindex(np.random.permutation(vstable.index))
vs.head(10)
```

Out[4]:

Cust ID	Gender	Income	Age	Rentals	Avg Per Visit	Genre	Incidentals
48	F	52000	47	14	1.6	Drama	No
12	F	26000	22	32	2.9	Action	Yes
47	F	69000	35	22	2.8	Drama	Yes
44	M	35000	24	24	1.7	Drama	No
42	M	32000	25	26	2.2	Action	Yes
7	F	29000	45	19	3.8	Drama	No
34	F	29000	32	19	2.9	Action	Yes
43	F	49000	28	48	3.3	Drama	Yes
35	M	74000	29	43	4.6	Action	Yes
25	M	1000	16	25	1.4	Comedy	Yes

In [5]:
```
len(vs)
```

Out[5]:
```
50
```

In [6]:
```
vs_names = vs.columns.values
vs_names
```

Out[6]:
```
array(['Gender', 'Income', 'Age', 'Rentals', 'Avg Per Visit', 'Genre',
       'Incidentals'], dtype=object)
```

The target attribute for classification is Incidentals:

In [7]:
```
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 [8]:

```
vs = pd.get_dummies(vs[['Gender','Income','Age','Rentals','Avg Per Visit','G
vs.head(10)
```

Out[8]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genr
48	52000	47	14	1.6	1	0	0	0	
12	26000	22	32	2.9	1	0	1	0	
47	69000	35	22	2.8	1	0	0	0	
44	35000	24	24	1.7	0	1	0	0	
42	32000	25	26	2.2	0	1	1	0	
7	29000	45	19	3.8	1	0	0	0	
34	29000	32	19	2.9	1	0	1	0	
43	49000	28	48	3.3	1	0	0	0	
35	74000	29	43	4.6	0	1	1	0	
25	1000	16	25	1.4	0	1	0	1	

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 [9]:

```
tpercent = 0.8
tsize = int(np.floor(tpercent * len(vs)))
vs_train = vs[:tsize]
vs_test = vs[tsize:]
```

In [10]:

```
print(vs_train.shape)
print(vs_test.shape)
```

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

In [11]:

```
vs_train.head(10)
```

Out[11]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genr
48	52000	47	14	1.6	1	0	0	0	
12	26000	22	32	2.9	1	0	1	0	
47	69000	35	22	2.8	1	0	0	0	
44	35000	24	24	1.7	0	1	0	0	
42	32000	25	26	2.2	0	1	1	0	
7	29000	45	19	3.8	1	0	0	0	
34	29000	32	19	2.9	1	0	1	0	
43	49000	28	48	3.3	1	0	0	0	
35	74000	29	43	4.6	0	1	1	0	
25	1000	16	25	1.4	0	1	0	1	

In [12]:

```
vs_test
```

Out[12]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genr
23	2000	15	30	2.5	1	0	0	1	
27	62000	47	32	3.6	1	0	0	0	
49	31000	25	42	3.4	0	1	1	0	
17	36000	35	28	3.5	0	1	0	0	
31	49000	56	15	3.2	1	0	0	1	
36	29000	21	34	2.3	1	0	0	1	
40	17000	19	32	1.8	0	1	1	0	
4	59000	70	16	4.2	1	0	0	0	
11	41000	22	48	2.3	1	0	0	0	
6	18000	20	29	1.7	0	1	1	0	

Splitting the target attribute ("Incidentals") accordingly:

In [13]:
```
vs_target_train = vs_target[0:int(tsize)]
vs_target_test = vs_target[int(tsize):len(vs)]
```

In [14]:
```
vs_target_train.head()
```

Out[14]:
```
Cust ID
48     No
12    Yes
47    Yes
44     No
42    Yes
Name: Incidentals, dtype: object
```

In [15]:
```
vs_target_test
```

Out[15]:
```
Cust ID
23     No
27     No
49    Yes
17    Yes
31     No
36     No
40     No
4     Yes
11    Yes
6      No
Name: Incidentals, dtype: object
```

Next, we normalize the attributes so that everything is in [0,1] scale. We can use the normalization functions from the kNN module in Ch. 2 of the text. In this case, however, we will use the more flexible and robust scaler function from the preprocessing module of scikit-learn package.

In [16]:
```
from sklearn import preprocessing
```

In [17]:
```
min_max_scaler = preprocessing.MinMaxScaler()
min_max_scaler.fit(vs_train)
```

Out[17]:
```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

In [18]:
```
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 [19]:

```
np.set_printoptions(precision=2, linewidth=100)

print(vs_train_norm[:10])
```

```
[[0.58 0.86 0.13 0.14 1.   0.   0.   0.   1.  ]
 [0.28 0.17 0.59 0.5  1.   0.   1.   0.   0.  ]
 [0.77 0.53 0.33 0.47 1.   0.   0.   0.   1.  ]
 [0.39 0.22 0.38 0.17 0.   1.   0.   0.   1.  ]
 [0.35 0.25 0.44 0.31 0.   1.   1.   0.   0.  ]
 [0.32 0.81 0.26 0.75 1.   0.   0.   0.   1.  ]
 [0.32 0.44 0.26 0.5  1.   0.   1.   0.   0.  ]
 [0.55 0.33 1.   0.61 1.   0.   0.   0.   1.  ]
 [0.83 0.36 0.87 0.97 0.   1.   1.   0.   0.  ]
 [0.   0.   0.41 0.08 0.   1.   0.   1.   0.  ]]
```

In [20]:

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

```
[[0.   0.   0.45 0.32 1.   0.   0.   1.   0.  ]
 [1.   0.58 0.52 0.76 1.   0.   0.   0.   1.  ]
 [0.48 0.18 0.82 0.68 0.   1.   1.   0.   0.  ]
 [0.57 0.36 0.39 0.72 0.   1.   0.   0.   1.  ]
 [0.78 0.75 0.   0.6  1.   0.   0.   1.   0.  ]
 [0.45 0.11 0.58 0.24 1.   0.   0.   1.   0.  ]
 [0.25 0.07 0.52 0.04 0.   1.   1.   0.   0.  ]
 [0.95 1.   0.03 1.   1.   0.   0.   0.   1.  ]
 [0.65 0.13 1.   0.24 1.   0.   0.   0.   1.  ]
 [0.27 0.09 0.42 0.   0.   1.   1.   0.   0.  ]]
```

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

In [21]:

```
vs_target_train = np.array(vs_target_train)
vs_target_test = np.array(vs_target_test)
```

In [22]:

```
print(vs_target_train)
print("\n")
print(vs_target_test)
```

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


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

The following function illustrates how we can perform a k-nearest-neighbor search. It takes an instance x 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 [23]:

```
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 nc
        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
        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
```

In [24]:

```
# Let's use vs_test_norm[0] as a test instance x and find its K nearest neig
neigh_idx, distances = knn_search(vs_test_norm[0], vs_train_norm, 5, 0)
```

In [25]:

```
vs_test.head(1)
```

Out[25]:

	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genr
Cust ID									
23	2000	15	30	2.5	1	0	0	1	

In [26]:

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

```
[12 28 11  9 38]

Nearest Neigbors:
```

Out[26]:

Cust ID	Income	Age	Rentals	Avg Per Visit	Gender_F	Gender_M	Genre_Action	Genre_Comedy	Genr
19	24000	25	41	3.1	1	0	0	1	
3	32000	20	42	1.6	1	0	0	1	
39	68000	35	19	3.9	1	0	0	1	
25	1000	16	25	1.4	0	1	0	1	
29	15000	18	37	2.1	1	0	1	0	

In [27]:

```
print(distances[neigh_idx])
```

```
[0.57 0.57 1.05 1.43 1.45]
```

In [28]:

```
# Let's see how the nearest neighbors of the test instance labeled the targe

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

```
['No' 'No' 'No' 'Yes' '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 [29]:

```
from collections import Counter
print(Counter(neigh_labels))
```

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

In [30]:

```
Counter(neigh_labels).most_common(1)
```

Out[30]:

```
[('No', 3)]
```

Next, we'll use the Knn module from Chapter 2 of Machine Learning in Action. Before importing the whole module, let's illustrate what the code does by stepping through it with some specific input values.

In [31]:

```
dataSetSize = vs_train_norm.shape[0]
print(dataSetSize)
```

```
40
```

In [32]:

```
inX = vs_test_norm[0]   # Again we'll use the first instance in the test dat
diffMat = np.tile(inX, (dataSetSize,1)) - vs_train_norm  # Create dataSetSiz
                                                          # Compute a matrix
print(diffMat[:5,:])
```

```
[[-0.58 -0.86  0.33  0.18  0.    0.    0.    1.   -1.  ]
 [-0.28 -0.17 -0.14 -0.18  0.    0.   -1.    1.    0.  ]
 [-0.77 -0.53  0.12 -0.15  0.    0.    0.    1.   -1.  ]
 [-0.39 -0.22  0.07  0.15  1.   -1.    0.    1.   -1.  ]
 [-0.35 -0.25  0.02  0.01  1.   -1.   -1.    1.    0.  ]]
```

In [33]:

```
sqDiffMat = diffMat**2  # The matrix of squared differences
sqDistances = sqDiffMat.sum(axis=1)  # 1D array of the sum of squared differ
distances = sqDistances**0.5  # and finally the matrix of Euclidean distance
print(distances)
```

```
[1.79 1.47 1.71 2.06 2.05 1.72 1.54 1.67 2.33 1.43 2.14 1.05 0.57 2.14 2.1
 1.95 1.67 1.92 1.81 1.63 1.75 1.68 1.46 2.01 0.57 2.01 2.17 2.21 2.09 2.1
 1.45 2.18]
```

In [34]:

```
sortedDistIndicies = distances.argsort() # the indices of the training insta
print(sortedDistIndicies)
```

```
[12 28 11  9 38 26  1 16  6 23  7 18 20 25  2  5 24  0 35 22 36 21 19 37 2
 17 10 13 30 39 31  8]
```

To see how the test instance should be classified, we need to find the majority class among the neighbors (here we do not use distance weighting; only a simply voting approach)

In [35]:

```
classCount={}
k = 5  # We'll use the top 5 neighbors
for i in range(k):
   voteIlabel = vs_target_train[sortedDistIndicies[i]]
   classCount[voteIlabel] = classCount.get(voteIlabel,0) + 1  # add to the c
   print(sortedDistIndicies[i], voteIlabel, classCount[voteIlabel])
```

```
12 No 1
28 No 2
11 No 3
9 Yes 1
38 Yes 2
```

Now, let's find the predicted class for the test instance.

In [36]:
```
import operator
# Create a dictionary for the class labels with cumulative occurrences acros
# Dictionary will be ordered in decreasing order of the lable values (so the
# be the first dictonary element)
sortedClassCount = sorted(classCount.items(), key=operator.itemgetter(1), re
print(sortedClassCount)
print(sortedClassCount[0][0])
```

```
[('No', 3), ('Yes', 2)]
No
```

A better way to find the majority class given a list of class labels from neighbors is to use a Python Counter:

In [37]:
```
from collections import Counter

k = 5  # We'll use the top 5 neighbors
vote = vs_target_train[sortedDistIndicies[0:k]]
maj_class = Counter(vote).most_common(1)

print(vote)

print(maj_class)

print("Class label for the classified instance: ", maj_class[0][0])
```

```
['No' 'No' 'No' 'Yes' 'Yes']
[('No', 3)]
Class label for the classified instance:  No
```

In [ ]:
```
```

Let's know import a module containing a modified version of the kNN classifier implementation (http://facweb.cs.depaul.edu/mobasher/classes/CSC478/data/kNN_new.py.txt) from Chapter 2 of MLA book. We will step through all test instances, use our Knn classifier to predict a class label for each instance, and in each case we compare the predicted label to the actual value from the target test labels.

In [48]:
```
# kNN_new.py must be in the working folder (or you can specify the path in t

import kNN_new
```

In [50]:
```
numTestVecs = len(vs_target_test)
print(numTestVecs)
```

```
10
```

In [61]:

```
errorCount = 0.0
for i in range(numTestVecs):
    # classify0 function uses Euclidean distance to find k nearest neighbors
    classifierResult = kNN_new.classify0(vs_test_norm[i,:], vs_train_norm, v
    print("Predicted Label: ", classifierResult, "==> Actual Label: ", vs_ta
    print()
    if (classifierResult != vs_target_test[i]):
        errorCount += 1.0

print("the total error rate is: ", errorCount/float(numTestVecs))
```

```
Predicted Label:  No ==> Actual Label:  No

Predicted Label:  Yes ==> Actual Label:  No

Predicted Label:  No ==> Actual Label:  Yes

Predicted Label:  Yes ==> Actual Label:  Yes

Predicted Label:  No ==> Actual Label:  No

Predicted Label:  No ==> Actual Label:  No

Predicted Label:  Yes ==> Actual Label:  No

Predicted Label:  No ==> Actual Label:  Yes

Predicted Label:  Yes ==> Actual Label:  Yes

Predicted Label:  Yes ==> Actual Label:  No

the total error rate is:  0.5
```

I have added a new classifier function to the kNN module that uses Cosine similarity instead of Euclidean distance:

In [63]:

```
errorCount = 0.0
for i in range(numTestVecs):
    # classify1 function uses inverse of Cosine similarity to find k nearest
    classifierResult2 = kNN_new.classify1(vs_test_norm[i,:], vs_train_norm,
    print("Predicted Label: ", classifierResult, "==> Actual Label: ", vs_ta
    print()
    if (classifierResult != vs_target_test[i]):
         errorCount += 1.0

print("the total error rate is: ", errorCount/float(numTestVecs))
```

```
Predicted Label:  Yes ==> Actual Label:  No

Predicted Label:  Yes ==> Actual Label:  No

Predicted Label:  Yes ==> Actual Label:  Yes

Predicted Label:  Yes ==> Actual Label:  Yes

Predicted Label:  Yes ==> Actual Label:  No

Predicted Label:  Yes ==> Actual Label:  No

Predicted Label:  Yes ==> Actual Label:  No

Predicted Label:  Yes ==> Actual Label:  Yes

Predicted Label:  Yes ==> Actual Label:  Yes

Predicted Label:  Yes ==> Actual Label:  No

the total error rate is:  0.6
```

In [ ]: