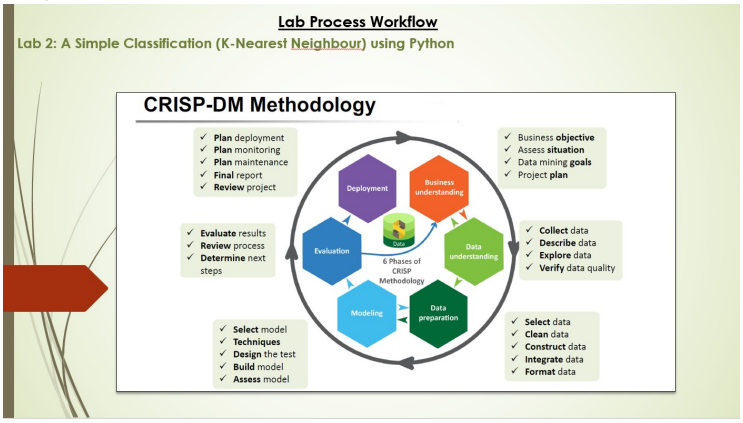
CDS503: Machine Learning

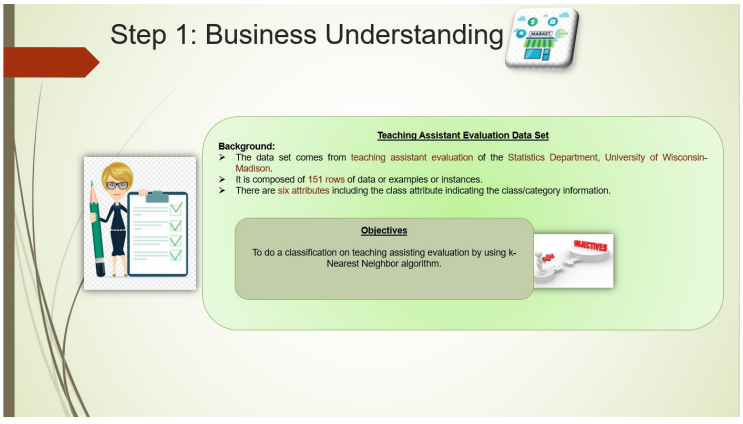
LAB 2: A Simple Classification (K-Nearest Neighbour) using Python

We are going to start demonstrating Python with a simple classification task. In this lab, we are going to explore the Teaching Assistant Evaluation (TAE) data set, which will be used to train a model to decide what evaluation a teaching assistant (TA) would get. You should make a folder in the lab or your own computer to save data and your own work. The data set is freely available from: http://archive.ics.uci.edu/ml/datasets/Teaching+Assistant+Evaluation (http://archive.ics.uci.edu/ml/datasets/Teaching+Assistant+Evaluation).

In this Lab we are going to use the CRISP-DM model.



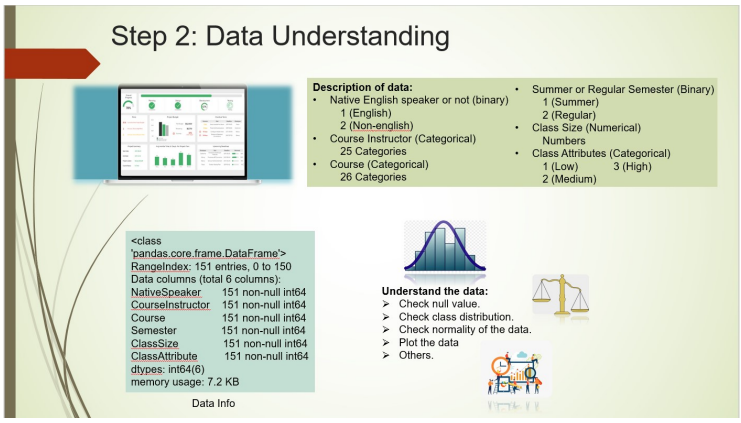
Step 1: Business Understanding



About:

The data set comes from teaching assistant evaluation of the Statistics Department, University of Wisconsin-Madison. The data set is composed of 151 rows of data or examples or instances. Each instance corresponds to a teaching assistant evaluation from a course.

Step 2: Data Understanding



Description of the data:

Each instance describes features/attributes of an object or entity, which in our case here is a teaching assistant evaluation. In the TEA data set, there are six attributes including the class attribute indicating the class/category information. The six attributes are:

- Native English speaker or not (binary)
  - 1 (English)
  - 2 (Non-english)
- Course Instructor (Categorical)
  - 25 Categories
- Course (Categorical)
  - 26 Categories
- Summer or Regular Semester (Binary)
  - 1 (Summer)
  - 2 (Regular)
- Class Size (Numerical)
  - Numbers
- Class Attributes (Categorical)
  - 1 (Low)
  - 2 (Medium)
  - 3 (High)

Each instance describes values on the six attributes for a teaching assistant evaluation object. The sixth attribute is the class attribute, which means the class/category this object belongs to. There are 3 categories for the class attribute, and they are “Low”, “Medium” and “High”. For example, the first instance is (1, 23, 3, 1, 19, 3), which tells the teaching assistant is an English speaker (attribute 1) and gets a high evaluation (attribute 6).

In [20]:
```
import pandas as pd
import numpy as np

df = pd.read_csv('input/tae.csv')
```

In [19]:
```
#describe the data
df.describe()
```

Out[19]:

	NativeSpeaker	CourseInstructor	Course	Semester	ClassSize	ClassAttribute
count	151.000000	151.000000	151.000000	151.000000	151.000000	151.000000
mean	1.807947	13.642384	8.105960	1.847682	27.867550	2.019868
std	0.395225	6.825779	7.023914	0.360525	12.893758	0.820327
min	1.000000	1.000000	1.000000	1.000000	3.000000	1.000000
25%	2.000000	8.000000	3.000000	2.000000	19.000000	1.000000
50%	2.000000	13.000000	4.000000	2.000000	27.000000	2.000000
75%	2.000000	20.000000	15.000000	2.000000	37.000000	3.000000
max	2.000000	25.000000	26.000000	2.000000	66.000000	3.000000

In [3]:
```
#Get tha shape/dimension of data
df.shape
```

Out[3]:
```
(151, 6)
```

In [2]:
```
colNames = ['NativeSpeaker', 'CourseInstructor', 'Course','Semester',
            'ClassSize','ClassAttribute']
# read again to put the column headers of the data
df = pd.read_csv('input/tae.csv', names=colNames)

df.describe()
```

Out[2]:

	NativeSpeaker	CourseInstructor	Course	Semester	ClassSize	ClassAttribute
count	151.000000	151.000000	151.000000	151.000000	151.000000	151.000000
mean	1.807947	13.642384	8.105960	1.847682	27.867550	2.019868
std	0.395225	6.825779	7.023914	0.360525	12.893758	0.820327
min	1.000000	1.000000	1.000000	1.000000	3.000000	1.000000
25%	2.000000	8.000000	3.000000	2.000000	19.000000	1.000000
50%	2.000000	13.000000	4.000000	2.000000	27.000000	2.000000
75%	2.000000	20.000000	15.000000	2.000000	37.000000	3.000000
max	2.000000	25.000000	26.000000	2.000000	66.000000	3.000000

In [4]:
```
# Count observations based on attribute
df['Course'].value_counts()
```

Out[4]:
```
3     45
2     16
1     14
15    10
17    10
11     9
7      7
5      5
8      4
9      3
25     3
13     3
16     3
21     3
22     3
18     2
6      2
14     1
12     1
10     1
19     1
20     1
4      1
23     1
24     1
26     1
Name: Course, dtype: int64
```

Check for null data

In [5]:
```
# select rows from dataframe
x=df.iloc[:,:-1]

# sum of null data based on attributes
x.isnull().sum()
```

Out[5]:
```
NativeSpeaker       0
CourseInstructor    0
Course              0
Semester            0
ClassSize           0
dtype: int64
```

Data Plots

To better understand the data set with working, some visualization may be required. For the purpose of this lab, lets try plotting using Scatter plot and Histogram.

In [6]:
```
import matplotlib.pyplot as plt # library for plotting

# line required for inline charts/plots
%matplotlib inline

# library for evaluation metrics
from sklearn import metrics

# library for sampling the data sets
from sklearn.model_selection import train_test_split
```

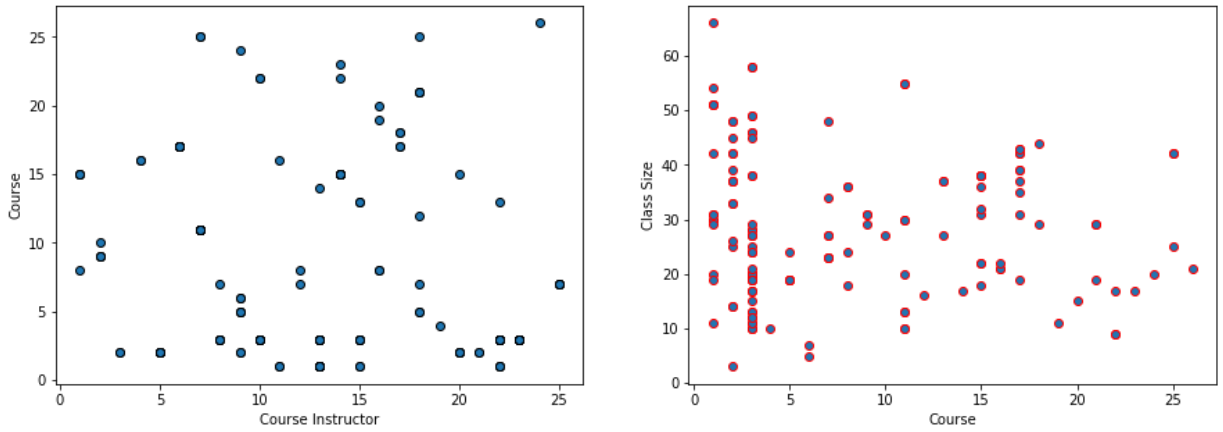
By plotting the scatter plot, we can display values for two variables for the data. In this case, the first plot represent the interaction patterns of the second attribute (course instructors) and the third attribute (courses). The first plot represent the interaction patterns of the third attribute (courses) and the fifth attribute (class size).

The choice of attributes and the type of data also played a *crucial* role here. The first plot cannot tell much information between the two attributes. However, from the second plot, we know that course 1 until 5 were among the classes that have high number of students.

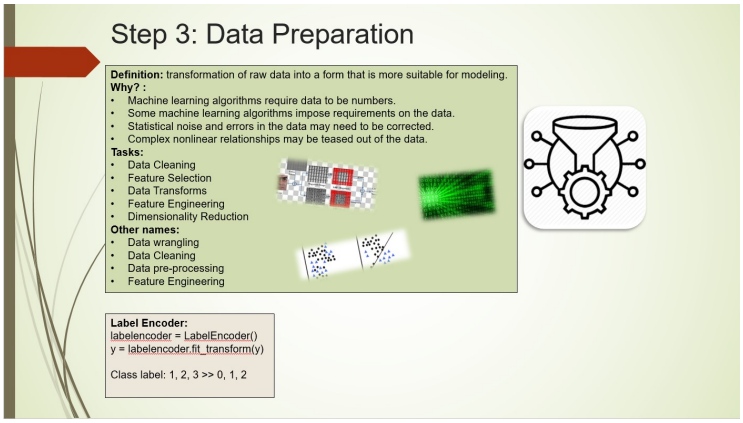
Note: scatter plot only can represent interaction patterns between two or more numerical or categorical attributes.

Learn more on the plots: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplot.html (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplot.html)

In [7]:
```
# size the figure to fit 2 figures on the same line
plt.figure(2,figsize=(15,5))

# subplot 1
plt.subplot(1,2,1)
# scatter plot 1
plt.scatter(x=x.iloc[:,1],y=x.iloc[:,2] ,edgecolor='k')
plt.xlabel('Course Instructor')
plt.ylabel('Course')

# subplot 1
plt.subplot(1,2,2)
# scatter plot 2
plt.scatter(x=x.iloc[:,2],y=x.iloc[:,4] ,edgecolor='r')
plt.xlabel('Course')
plt.ylabel('Class Size')

# show the resulting plot
plt.show()
```



Step 3: Data Preparation



Data preparation is required for transformation of raw data into a form that is more suitable for modeling. In this lab, since the target label (or attribute) for the classification is categorical (class attribute = 1, 2, or 3), then *label encoder* is used.

In [17]:
```
# import label encoder library to deal with categorical data
from sklearn.preprocessing import LabelEncoder

# select all columns except the last one (the target label)
x=df.iloc[:,:-1]
# set target categorical data label (sixth attribute)
y=df.iloc[:,5]

labelencoder = LabelEncoder()
y = labelencoder.fit_transform(y)
print (x)
print(y)
```

```
     NativeSpeaker  CourseInstructor  Course  Semester  ClassSize
0                1                23       3         1         19
1                2                15       3         1         17
2                1                23       3         2         49
3                1                 5       2         2         33
4                2                 7      11         2         55
..             ...               ...     ...       ...        ...
146              2                 3       2         2         26
147              2                10       3         2         12
148              1                18       7         2         48
149              2                22       1         2         51
150              2                 2      10         2         27

[151 rows x 5 columns]
[2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
 0 0 0]
```

Then, the data is split into train and test sets. The train set is used to train the classifier and validate its accuracy. Then, the classifier will be evaluated by the test set to determine its performance. The following codes are used for splitting the data into train and test sets:

In [10]:
```
# import machine learning library to sample the data
from sklearn.model_selection import train_test_split

x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.4,random_state=0)
```

Then, we standardize the scalability of the data

In [18]:
```
# import machine learning library to standardize the data
from sklearn.preprocessing import StandardScaler

sc=StandardScaler()
sc.fit_transform(x_train,y_train)
sc.fit(x_test,y_test)
```

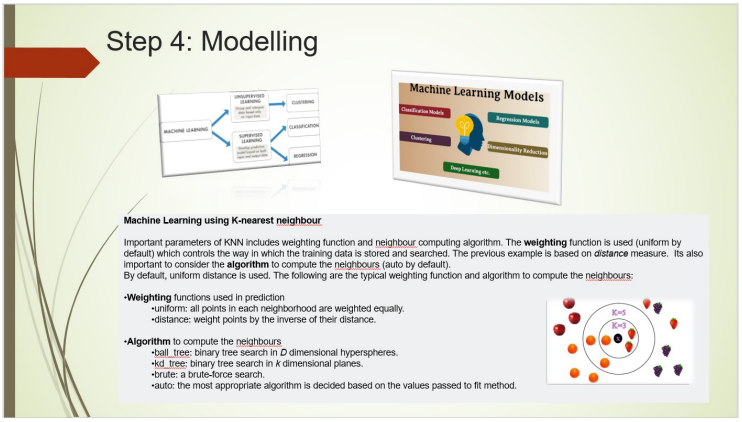
Out[18]:
```
StandardScaler(copy=True, with_mean=True, with_std=True)
```

Check the data so far:

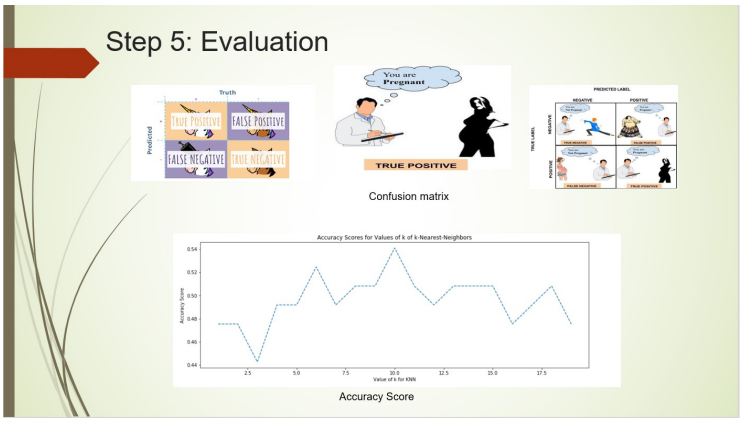
In [12]:
```
print('-------- x axis test ----------')
print(x_test)
print('-------- x axis train ---------')
print(x_train)
print('-------- y axis test ----------')
print(y_test)
print('-------- y axis train ---------')
print(y_train)
print('******************************')
```

```
-------- x axis test ----------
     NativeSpeaker  CourseInstructor  Course  Semester  ClassSize
114              2                12       7         2         34
62               2                 1      15         1         22
33               1                13       3         1         13
107              2                20       2         2         14
7                2                10       3         2         27
..             ...               ...     ...       ...        ...
106              2                23       3         2         10
147              2                10       3         2         12
50               2                13       1         2         30
148              1                18       7         2         48
85               2                 7      11         1         20

[61 rows x 5 columns]
-------- x axis train ---------
     NativeSpeaker  CourseInstructor  Course  Semester  ClassSize
30               2                11       1         2         51
101              2                23       3         1         20
94               2                 1      15         2         31
64               2                 7      11         2         13
89               2                14      22         2         17
..             ...               ...     ...       ...        ...
9                2                15       3         1         20
103              2                 6      17         2         37
67               2                21       2         2         42
117              2                23       3         2         12
47               1                22       3         2         58

[90 rows x 5 columns]
-------- y axis test ----------
[0 1 0 0 2 1 2 2 0 0 1 2 0 1 1 0 2 2 2 1 2 1 1 2 2 1 2 1 2 1 1 2 1 2 1 1 1
 2 1 2 0 2 0 2 2 1 0 0 0 0 0 1 1 2 2 1 1 0 2 0 2]
-------- y axis train ---------
[0 1 2 1 2 2 2 2 2 0 1 1 1 2 2 0 1 2 0 0 1 2 1 0 1 1 2 2 0 2 2 0 2 0 0 2 2
 1 1 0 2 1 0 1 2 0 0 1 0 0 1 0 0 0 1 1 1 0 0 1 2 1 0 1 2 1 1 2 2 0 0 0 0 1
 2 0 0 2 1 0 2 0 2 0 1 2 1 0 0 2]
******************************
```

Step 4: Modelling



Step 5: Evaluation



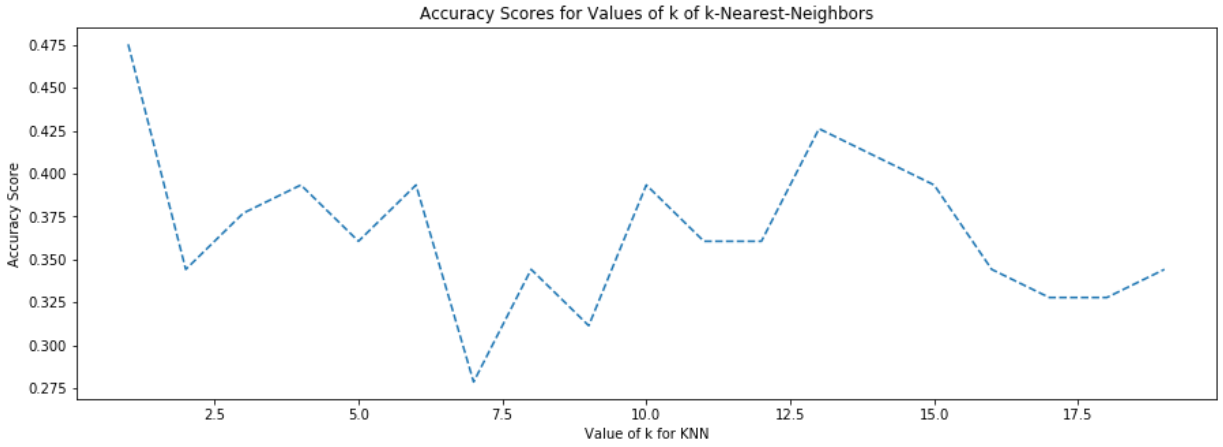
Machine Learning using K-nearest neighbour

A simple classification task will be used to demonstrate the workings of a machine learning algorithm. There are many classification and regression algorithms that can be directly implemented using Python `sklearn` library (check this link for more: http://scikit-learn.org/stable/index.html (http://scikit-learn.org/stable/index.html)) In this lab, the K nearest neighbour (or KNN) will be used to classify the data. Since the target label (or attribute) for the classification is categorical (class attribute = 1, 2, or 3), then *label encoder* is used.

Then, we apply the **KNN models** (refer to: http://scikit-learn.org/stable/modules/neighbors.html (http://scikit-learn.org/stable/modules/neighbors.html)).

The size of the neighborhood is controlled by the *k* parameter. For example, if set to 1, then predictions are made using the single most similar training instance to a given new pattern for which a prediction is requested. Larger data set commonly uses larger *k* value. However, larger *k* doesn't always give better result. The following code shows different accuracy results for different values of *k*. The following code shows an example of *default* application of KNN.

In [13]:
```
# import KNN model as 'KNeighborsClassifier'
from sklearn.neighbors import KNeighborsClassifier

# empty variable for storing the KNN metrics
scores=[]

# We try different values of k for the KNN (from k=1 up to k=26)
lrange=list(range(1,20))

# loop the KNN process
for k in lrange:
    # input the k value and 'distance' measure
    knn=KNeighborsClassifier(n_neighbors=k)
    # input the train data to train KNN
    knn.fit(x_train,y_train)
    # see KNN prediction by inputting the test data
    y_pred=knn.predict(x_test)
    # append the performance metric (accuracy)
    scores.append(metrics.accuracy_score(y_test,y_pred))

plt.figure(2,figsize=(15,5))

# plot the results
plt.plot(lrange, scores,ls='dashed')
plt.xlabel('Value of k for KNN')
plt.ylabel('Accuracy Score')
plt.title('Accuracy Scores for Values of k of k-Nearest-Neighbors')
plt.show()
```



The result showed that the highest accuracy obtained by KNN is when **k = 1 (accuracy = 0.475)**

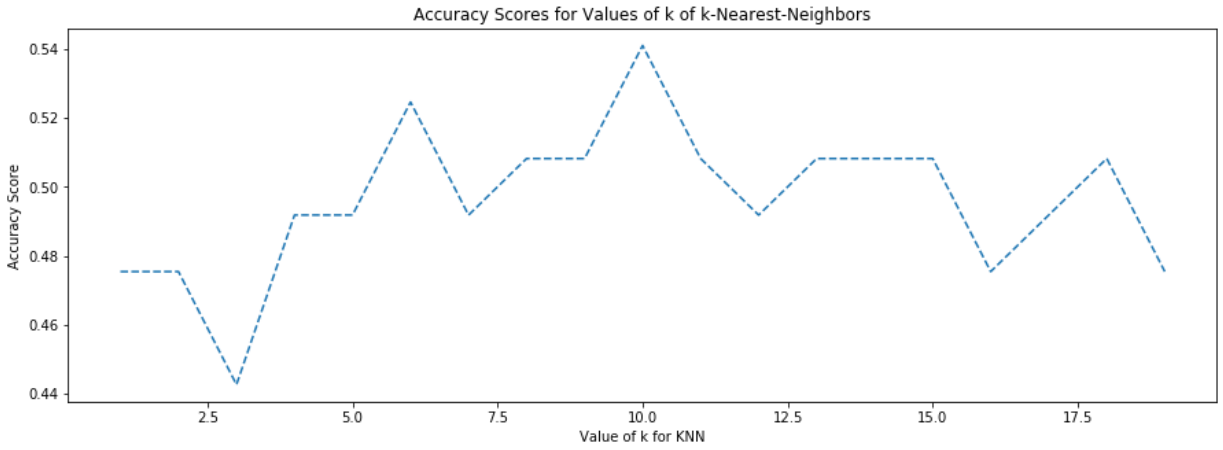
Important parameters of KNN includes weighting function and neighbour computing algorithm. The **weighting** function is used (`uniform` by default). which controls the way in which the training

data is stored and searched. The previous example is based on *distance* measure. Its also important to consider the **algorithm** to compute the neighbours (`auto` by default). By default, `uniform` distance is used. The following are the typical weighting function and algorithm to compute the neighbours:

- **Weighting** functions used in prediction
  - `uniform` : all points in each neighborhood are weighted equally.
  - `distance` : weight points by the inverse of their distance.
- **Algorithm** to compute the neighbours
  - `ball_tree` : binary tree search in *D* dimensional hyperspheres.
  - `kd_tree` : binary tree search in *k* dimensional planes.
  - `brute` : a brute-force search.
  - `auto` : the most appropriate algorithm is decided based on the values passed to `fit` method.

The following is code for application of KNN with specific parameters:

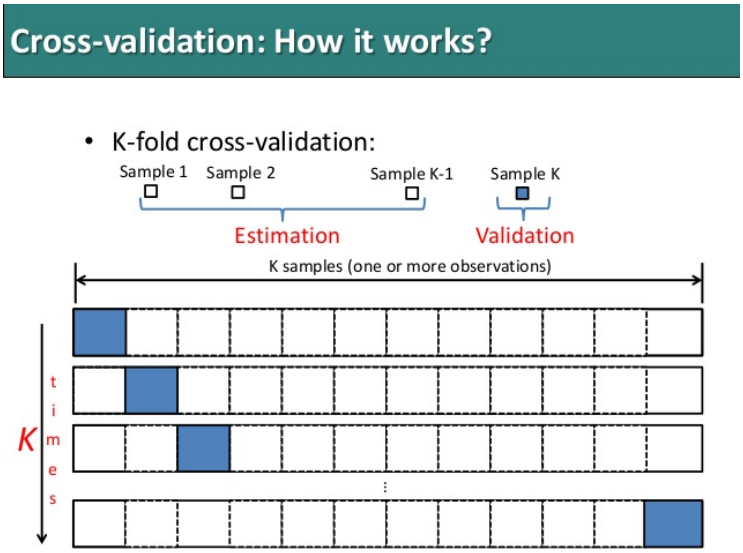
In [14]:
```
# empty variable for storing the KNN metrics
scores=[]

# We try different values of k for the KNN (from k=1 up to k=26)
lrange=list(range(1,20))

# loop the KNN process
for k in lrange:
    # input the k value and 'distance' measure
    knn=KNeighborsClassifier(n_neighbors=k, weights='distance', algorithm='auto'
    # input the train data to train KNN
    knn.fit(x_train,y_train)
    # see KNN prediction by inputting the test data
    y_pred=knn.predict(x_test)
    # append the performance metric (accuracy)
    scores.append(metrics.accuracy_score(y_test,y_pred))

plt.figure(2,figsize=(15,5))

# plot the results
plt.plot(lrange, scores,ls='dashed')
plt.xlabel('Value of k for KNN')
plt.ylabel('Accuracy Score')
plt.title('Accuracy Scores for Values of k of k-Nearest-Neighbors')
plt.show()
```



The result showed that the highest accuracy obtained by KNN using *distance* weighting function is when **k = 10 (accuracy = 0.54)**. Although it is *better* than the default KNN, the *accuracy* of the classifier is still **poor**.

To address this, we need to conduct some *tuning* on the KNN parameter by using **cross-validation**. Obviously, the best *k* is the one that corresponds to the lowest test error rate, so let's suppose we carry out repeated measurements of the test error for different values of *k*. Inadvertently, what we are doing is using the *test set* as a *training set*! This means that we are underestimating the true error rate since our model has been forced to fit the test set in the best possible manner. Our model is then *incapable* of generalizing to newer observations, a process known as **overfitting**. Hence, touching the test set is out of the question and must only be done at the very end of our pipeline.

An alternative and smarter approach involves estimating the *test error rate* by holding out a subset of the training set from the fitting process. This subset, called the *validation* set, can be used to select the appropriate level of flexibility of our algorithm! There are different validation approaches that are used in practice, and we will be exploring one of the more popular ones called **k-fold cross validation**.

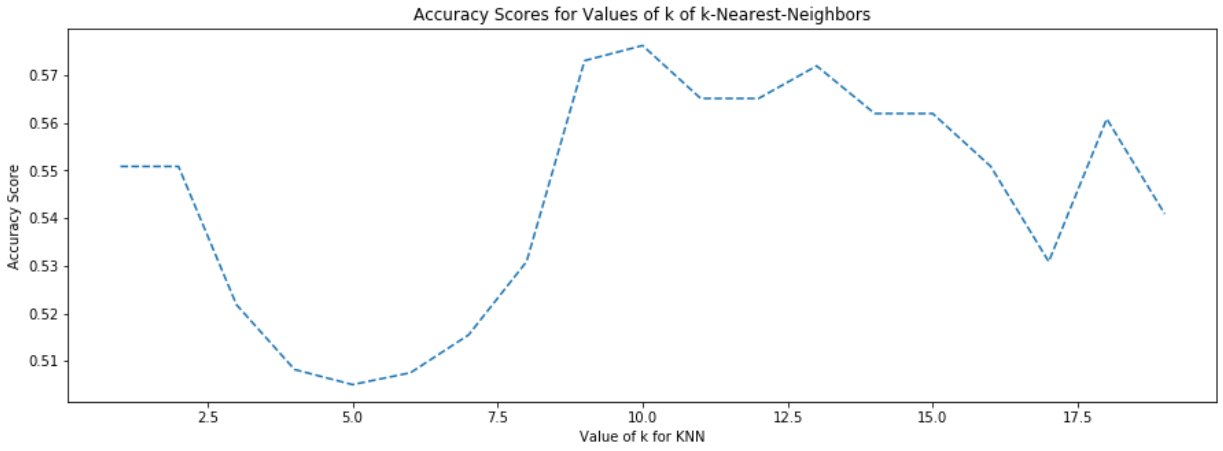


K-fold cross validation (the *k* is totally unrelated to *K* of KNN) involves randomly dividing the training set into *k* groups, or folds, of approximately equal size. The first fold is treated as a *validation set*, and the method is fit on the remaining *k−1* folds. The misclassification rate is then computed on the observations in the held-out fold. This procedure is repeated *k* times; each time, a different group of observations is treated as a *validation set*. This process results in *k* estimates of the test error which are then averaged out.

Cross-validation can be used to estimate the test error associated with a learning method in order to **evaluate** its performance, or to select the appropriate level of *flexibility*. *Scikit* learn comes in handy with its `cross_val_score()` method. We specifiy that we are performing *10* folds with the *cv=10* parameter and that our scoring metric should be **accuracy** since we are in a classification setting.

In [15]:
```
# import library for cross validation scoring
from sklearn.model_selection import cross_val_score

# empty variable for storing the KNN metrics
scores=[]

# We try different values of k for the KNN (from k=1 up to k=26)
lrange=list(range(1,20))

# loop the KNN process
for k in lrange:
    # input the k value and 'distance' measure
    knn=KNeighborsClassifier(n_neighbors=k, weights='distance', algorithm='auto'
    # get score for the 10 fold cross validation
    score = cross_val_score(knn, x_train, y_train, cv=10, scoring='accuracy')
    scores.append(score.mean())

optimal_k = lrange[scores.index(max(scores))]
print("The optimal number of neighbors is %d" % optimal_k)
print("The optimal score is %.2f" % max(scores))

plt.figure(2,figsize=(15,5))

# plot the results
plt.plot(lrange, scores,ls='dashed')
plt.xlabel('Value of k for KNN')
plt.ylabel('Accuracy Score')
plt.title('Accuracy Scores for Values of k of k-Nearest-Neighbors')
plt.show()
```

```
The optimal number of neighbors is 10
The optimal score is 0.58
```



Lab 2 Exercise

Use the dataset that you select in your Assignment 1. Look at each attribute and see what type of data it has.

Question 1: Do any **preprocessing** to data as *necessary*. Then, answer the following questions:

- What are the **types** of the attributes?
- Is there any **empty or null** values? What approach you use to address them (remove, replace, etc.)? and why?
- Any **unused** or **irrelevant** columns/attributes? What do you do to them?
- What attribute(s) might be **useful**?

Question 2:

Experiment with KNN machine learning algorithm to *predict* your **Class label** based on your selected data. Use *default* KNN configurations and try **at least** two different values of *k*. Try conduct also with *custom* KNN configurations with **at least** 5 fold cross-validation. Compare the two KNN and specify your findings. Do higher values of *k* lead to better performance? Do cross-validation effect KNN performance?

You may *discuss* with your friends and *complete* the lab exercise. Post your solution on Lab 02 Submission on elearn@usm. Make sure you include your name on the submission post.

In [ ]: