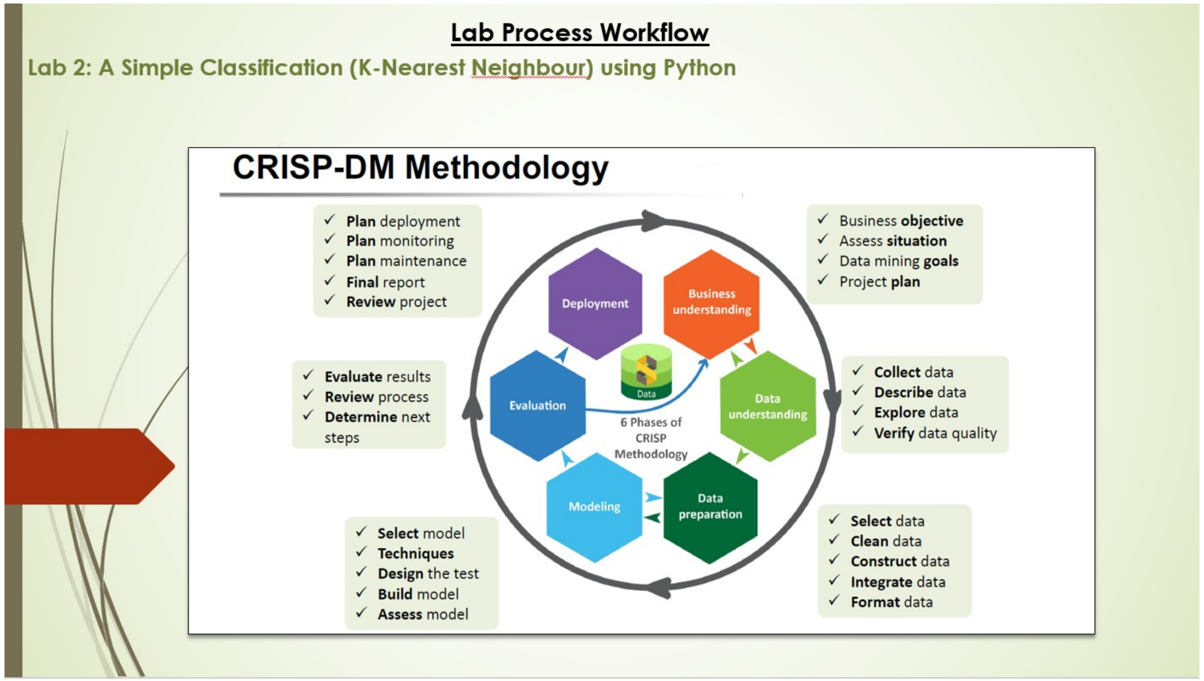
CDS503: Machine Learning

LAB 2: K-Nearest Neighbour (KNN)

A Simple Classification (K-Nearest Neighbour) using Python

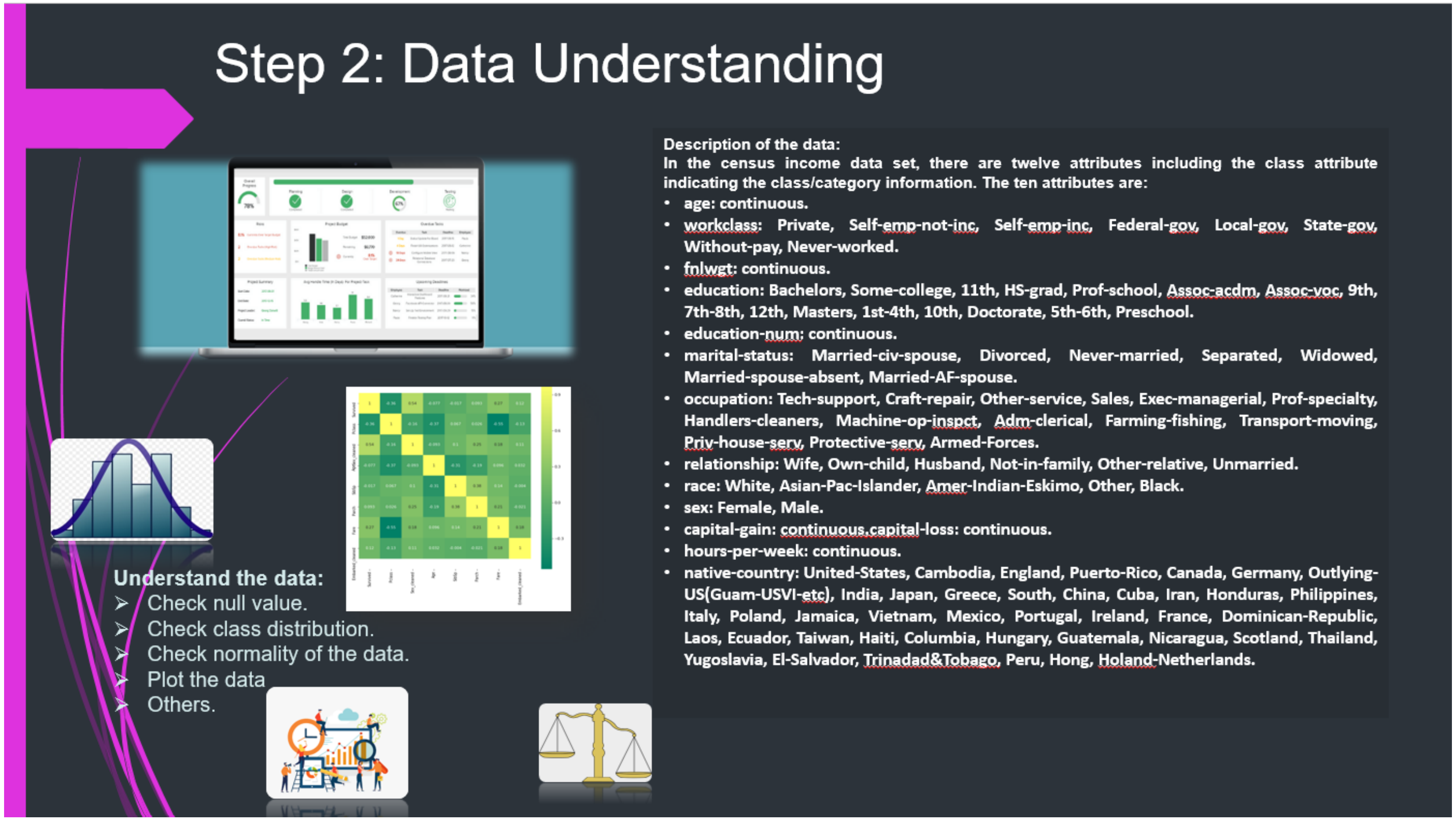
We are going to start demonstrating Python with a simple classification task. In this lab, we are going to explore the Cencus Income data set, It contains the cencus income of the people. They are trying to see the income of more than 50k and less than 50k. You should make a folder in the lab or your own computer to save data and your own work. This data was extracted from the census bureau database found at: [http://www.census.gov/ftp/pub/DES/www/welcome.html (http://www.census.gov/ftp/pub/DES/www/welcome.html)](http://www.census.gov/ftp/pub/DES/www/welcome.html).

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



Step1: Business Understanding

This data was extracted from the census bureau database found at: http://www.census.gov/ftp/pub/DES/www/welcome.html [(http://www.census.gov/ftp/pub/DES/www/welcome.html)](http://www.census.gov/ftp/pub/DES/www/welcome.html). It contains the cencus income of the people. They are trying to see the income of more than 50k and less than 50k. | Class label '>50K' : 23.93% / 24.78% (without unknowns) | Class label '<=50K' : 76.07% / 75.22% (without unknowns)



Step 2: Data Understanding

Description of the data:

In the censuc income data set, there are fifteen attributes including the class attribute indicating the class/category information. The 15 attributes are:

- age: continuous.

- workclass:
  - Private
  - Self-emp-not-inc
  - Self-emp-inc
  - Federal-gov
  - Local-gov
  - State-gov
  - Without-pay
  - Never-worked
- fnlwgt: continuous.
- education:
  - Bachelors
  - Some-college
  - 11th
  - HS-grad
  - Prof-school
  - Assoc-acdm
  - Assoc-voc
  - 9th
  - 7th-8th
  - 12th
  - Masters
  - 1st-4th
  - 10th
  - Doctorate
  - 5th-6th
  - Preschool
- education-num: continuous.
- marital-status:
  - Married-civ-spouse
  - Divorced
  - Never-married
  - Separated
  - Widowed
  - Married-spouse-absent
  - Married-AF-spouse
- occupation:

- Tech-support
- Craft-repair
- Other-service
- Sales
- Exec-managerial
- Prof-specialty
- Handlers-cleaners
- Machine-op-inspct
- Adm-clerical
- Farming-fishing
- Transport-moving
- Priv-house-serv
- Protective-serv
- Armed-Forces

- relationship:
  - Wife
  - Own-child
  - Husband
  - Not-in-family
  - Other-relative
  - Unmarried
- race:
  - White
  - Asian-Pac-Islander
  - Amer-Indian-Eskimo
  - Other
  - Black
- sex:
  - Female
  - Male
- capital-gain: continuous
- capital-loss: continuous
- hours-per-week: continuous.
- native-country:
  - United-States
  - Cambodia
  - England

- Puerto-Rico
- Canada
- Germany
- Outlying-US(Guam-USVI-etc)
- India
- Japan
- Greece
- South
- China
- Cuba
- Iran
- Honduras
- Philippines
- Italy
- Poland
- Jamaica
- Vietnam
- Mexico
- Portugal
- Ireland
- France
- Dominican-Republic
- Laos
- Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands.

  Note: Class label <=50K is encode to 0, Class label >50K is encode to 1

In [1]:
```
import pandas as pd
import numpy as np

train = pd.read_csv("input/adult_train_modified.csv")
test = pd.read_csv("input/adult_test_modified.csv")
```

In [2]:
```
#describe the data
train.describe()
```

Out[2]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-g
count	30162.000000	30162.000000	30162.000000	30162.000000	30162.000000	30162.000000	30162.000000	30162.000000	30162.000000	30162.000000	30162.000
mean	21.435482	2.199324	9825.221504	10.333764	9.121312	2.580134	5.959850	1.418341	3.678602	0.675685	6.5524
std	13.125355	0.953925	5671.017927	3.812292	2.549995	1.498016	4.029566	1.601338	0.834709	0.468126	23.2848
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
25%	11.000000	2.000000	5025.250000	9.000000	8.000000	2.000000	2.000000	0.000000	4.000000	0.000000	0.0000
50%	20.000000	2.000000	9689.500000	11.000000	9.000000	2.000000	6.000000	1.000000	4.000000	1.000000	0.0000
75%	30.000000	2.000000	14520.750000	12.000000	12.000000	4.000000	9.000000	3.000000	4.000000	1.000000	0.0000
max	71.000000	6.000000	20262.000000	15.000000	15.000000	6.000000	13.000000	5.000000	4.000000	1.000000	117.0000

In [3]:
```
#Get the shape/dimension of data
train.shape
```

Out[3]:
```
(30162, 15)
```

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

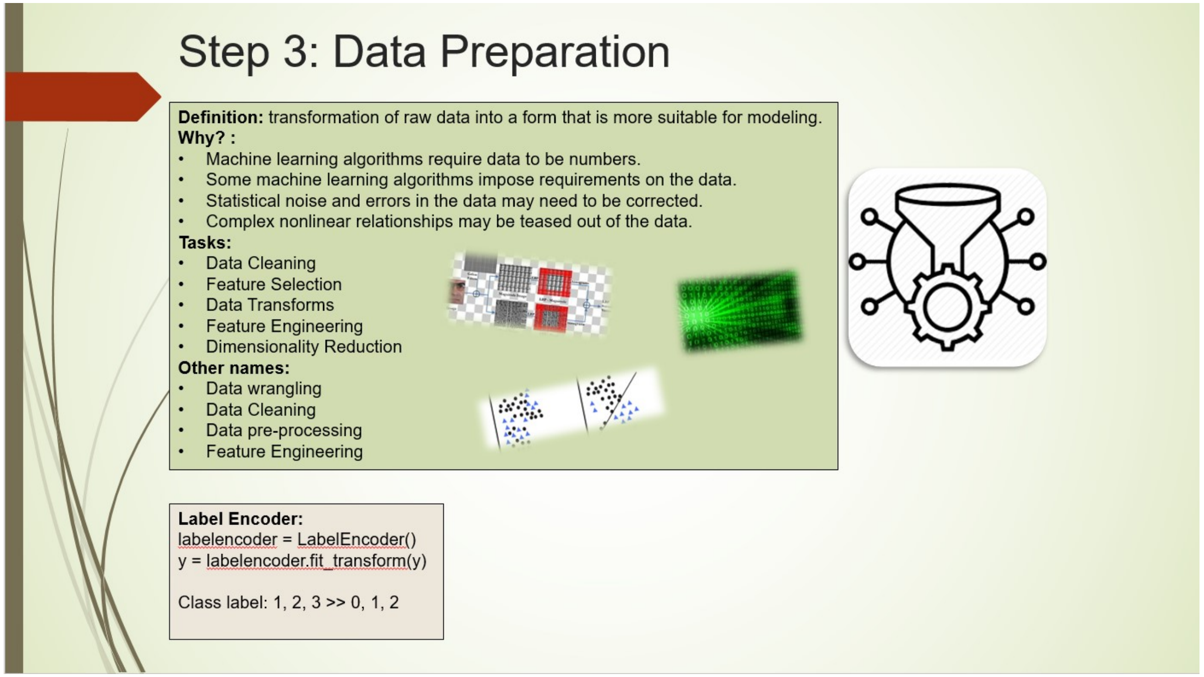
Out[4]:
```
0    22654
1     7508
Name: Class, dtype: int64
```

Check for null data

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

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

```
Out[5]: age               0
        workclass         0
        fnlwgt            0
        education         0
        education-num     0
        marital-status    0
        occupation        0
        relationship      0
        race              0
        sex               0
        capital-gain      0
        capital-loss      0
        hours-per-week    0
        native-country    0
        dtype: int64
```

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.

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 [6]:
```
# select all columns except the last one (the target label)
x_train=train.iloc[:,:-1]
# set target categorical data label (15th attribute)
y_train=train.iloc[:,14]

# select all columns except the last one (the target label)
x_test=test.iloc[:,:-1]
# set target categorical data label (sixth attribute)
y_test=test.iloc[:,14]

#Use line below if want to split data into training and testing
#x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.4,random_state=0)
```

Check the data so far:

In [7]:
```python
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('*******************************')
```

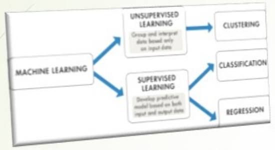
```
-------- x axis test ----------
       age  workclass  fnlwgt  education  education-num  marital-status  \
0        8          2    8315          1              6               4   
1       21          2    1754         11              8               2   
2       11          1   10750          7             11               2   
3       27          2    4780         15              9               2   
4       17          2    7091          0              5               4   
...    ...        ...     ...        ...            ...             ...   
15055   16          2    8927          9             12               4   
15056   22          2    7893          9             12               0   
15057   21          2   11193          9             12               2   
15058   27          2    1593          9             12               0   
15059   18          3    6062          9             12               2   

       occupation  relationship  race  sex  capital-gain  capital-loss  \
0               6             3     2    1             0             0   
1               4             0     4    1             0             0   
2              10             0     4    1             0             0   
3               6             0     2    1            87             0   
4               7             1     4    1             0             0   
...           ...           ...   ...  ...           ...           ...   
15055           9             3     4    1             0             0   
15056           9             1     4    0             0             0   
15057           9             0     4    1             0             0   
15058           0             3     1    1            73             0   
15059           3             0     4    1             0             0   

       hours-per-week  native-country  
0                  39              37  
1                  49              37  
2                  39              37  
3                  39              37  
4                  29              37  
```

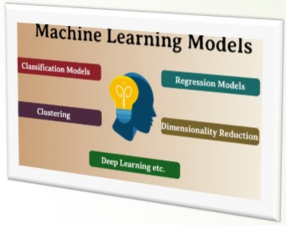
```
...                ...              ...
15055               39               37
15056               35               37
15057               49               37
15058               39               37
15059               59               37

[15060 rows x 14 columns]
-------- x axis train ---------
       age  workclass  fnlwgt  education  education-num  marital-status  \
0       22          5    2491          9             12               4
1       33          4    2727          9             12               2
2       21          2   13188         11              8               0
3       36          2   14354          1              6               2
4       11          2   18120          9             12               2
...    ...        ...     ...        ...            ...             ...
30157   10          2   15471          7             11               2
30158   23          2    7555         11              8               2
30159   41          2    7377         11              8               6
30160    5          2   12060         11              8               4
30161   35          3   16689         11              8               2

       occupation  relationship  race  sex  capital-gain  capital-loss  \
0               0             1     4    1            24             0
1               3             0     4    1             0             0
2               5             1     4    1             0             0
3               5             0     2    1             0             0
4               9             5     2    0             0             0
...           ...           ...   ...  ...           ...           ...
30157          12             5     4    0             0             0
30158           6             0     4    1             0             0
30159           0             4     4    0             0             0
30160           0             3     4    1             0             0
30161           3             5     4    0           107             0

       hours-per-week  native-country
0                  39              38
1                  12              38
2                  39              38
3                  39              38
4                  39               4
...               ...             ...
30157              37              38
30158              39              38
```

```
30159             39             38
30160             19             38
30161             39             38

[30162 rows x 14 columns]
-------- y axis test ----------
0        0
1        0
2        1
3        1
4        0
        ..
15055    0
15056    0
15057    0
15058    0
15059    1
Name: Class, Length: 15060, dtype: int64
-------- y axis train ---------
0        0
1        0
2        0
3        0
4        0
        ..
30157    0
30158    1
30159    0
30160    0
30161    1
Name: Class, Length: 30162, dtype: int64
*******************************
```

Step 4: Modelling

Step 4: Modelling

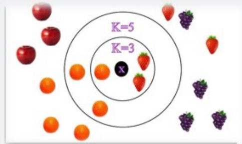




Machine Learning using K-nearest neighbour

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:

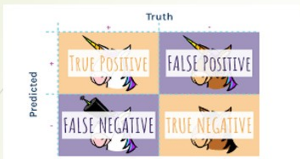
- **Weighting** functions used in prediction
  - uniform: all points in each neighborhood are weighted equally.
  - distance: weight points by the inverse of their distance.

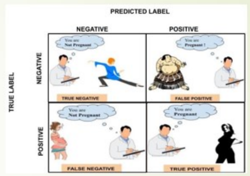


- **Algorithm** to compute the neighbours
  - ball_tree: binary tree search in *D* dimensional hyperspheres.
  - kd_tree: binary tree search in *k* dimensional planes.
  - brute: a brute-force search.
  - auto: the most appropriate algorithm is decided based on the values passed to fit method.

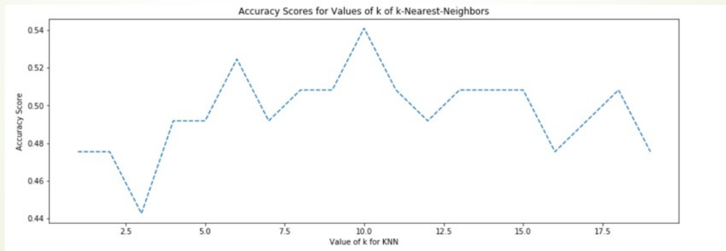
Step 5: Evaluation

Step 5: Evaluation





Confusion matrix



Accuracy Score

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)](http://scikit-learn.org/stable/index.html)) In this lab, the K nearest neighbour (or KNN) will be used to classify the data.

Then, we apply the **KNN models** (refer to: [http://scikit-learn.org/stable/modules/neighbors.html (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 [8]:
```
# import KNN model as 'KNeighborsClassifier'
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

#Define k-value
knn=KNeighborsClassifier(n_neighbors=1)
knn.fit(x_train,y_train)

#Estimate the accuracy of the classifier on test data
y_pred=knn.predict(x_test)
score = metrics.accuracy_score(y_test,y_pred)
score
```

Out[8]:
```
0.7201859229747676
```

The result showed that the highest accuracy obtained by KNN when **k = 1 (accuracy = 0.720)**

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:

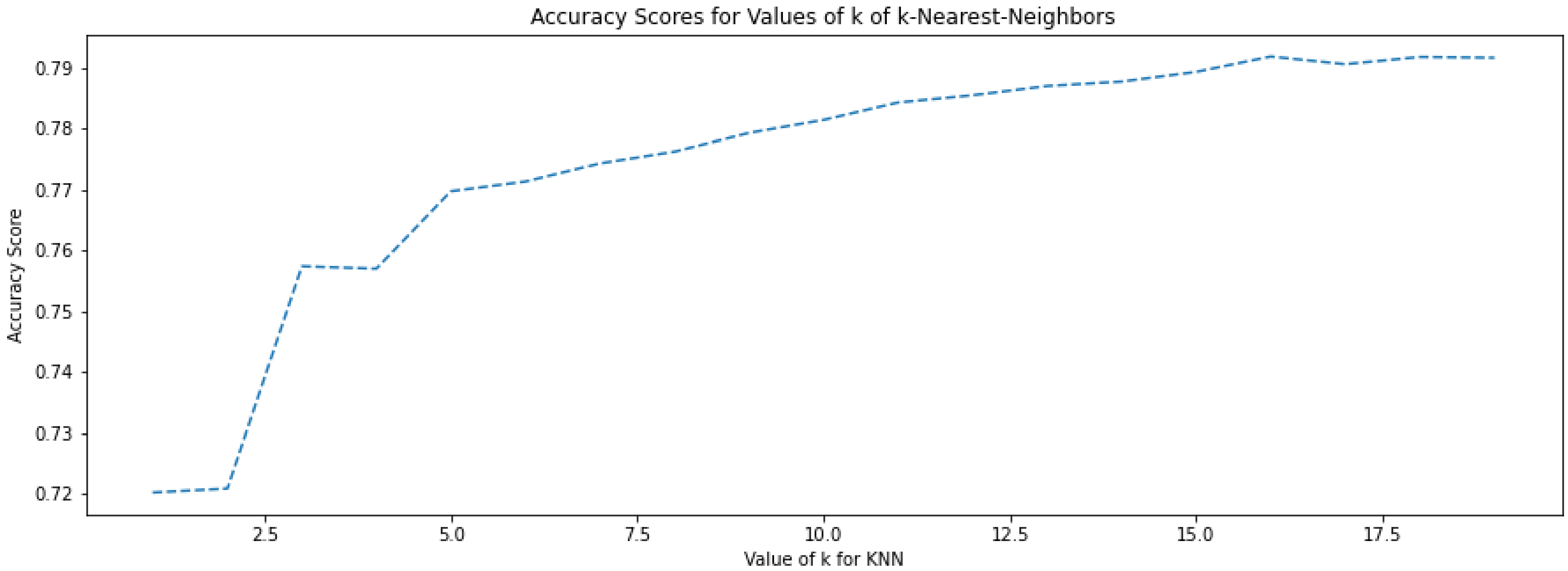
- **Weighting** functions used in prediction
  - `uniform` : all points in each neighborhood are weighted equally.
  - `distance` : weight points by the inverse of their distance.
- **Algorithm** to compute the neighbours
  - `ball_tree` : binary tree search in *D* dimensional hyperspheres.
  - `kd_tree` : 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 [9]:
```
import matplotlib.pyplot as plt # library for plotting
import warnings # to hide unnecesary warning
warnings.filterwarnings('ignore')
# line required for inline charts/plots
%matplotlib inline

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

# We try different values of k for the KNN (from k=1 up to k=20)
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))
    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 16
The optimal score is 0.79
```

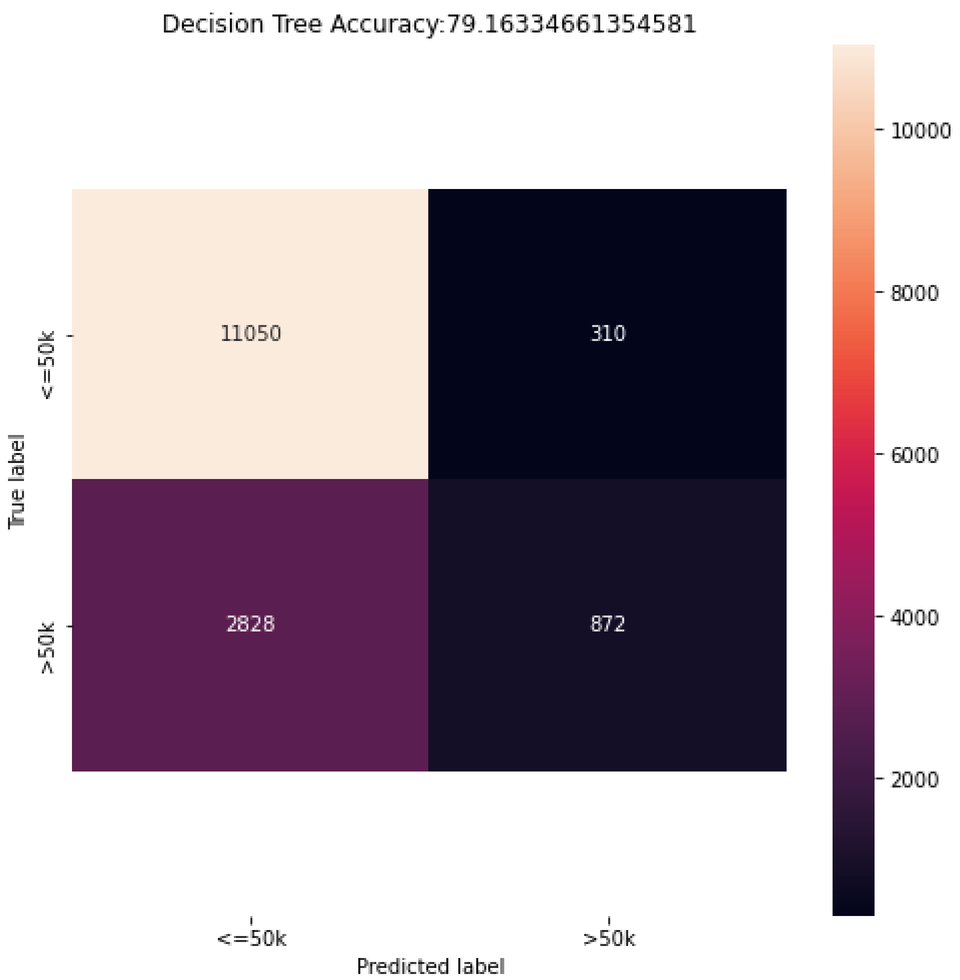


Accuracy Scores for Values of k of k-Nearest-Neighbors
Accuracy Score
0.79
0.78
0.77
0.76
0.75
0.74
0.73
0.72
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Value of k for KNN

In [10]:
```python
from sklearn.metrics import accuracy_score, confusion_matrix, precision_recall_fscore_support
import seaborn as sns

# predict the classes of new, unseen data
predict = knn.predict(x_test)

print("The prediction accuracy is: {0:2.2f}{1:s}".format(knn.score(x_test,y_test)*100,"%"))
# Creates a confusion matrix
cm = confusion_matrix(y_test, predict)


# Transform to dataframe for easier plotting
cm_df = pd.DataFrame(cm, index = ['<=50k','>50k'],
                     columns = ['<=50k','>50k'])

# plot the confusion matrix
plt.figure(figsize=(8,8))
ax= sns.heatmap(cm_df, annot=True, fmt='g')
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
plt.title("Decision Tree Accuracy:" + str(knn.score(x_test,y_test)*100))
plt.ylabel('True label')
plt.xlabel('Predicted label')
```

```
The prediction accuracy is: 79.16%
```

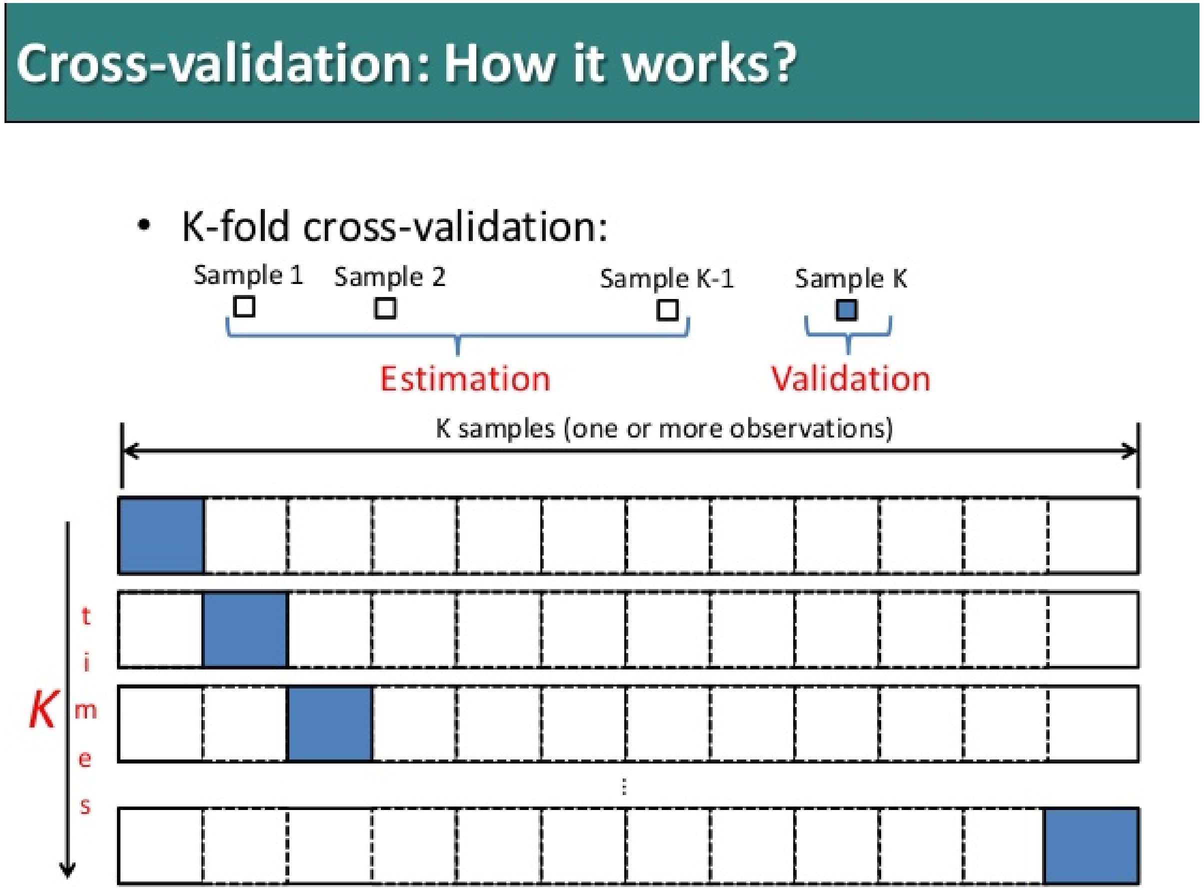
Out[10]:
```
Text(0.5, 51.0, 'Predicted label')
```



The result showed that the highest accuracy obtained by KNN using *distance* weighting function is when **k = 16 (accuracy = 0.79)**. 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 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**.



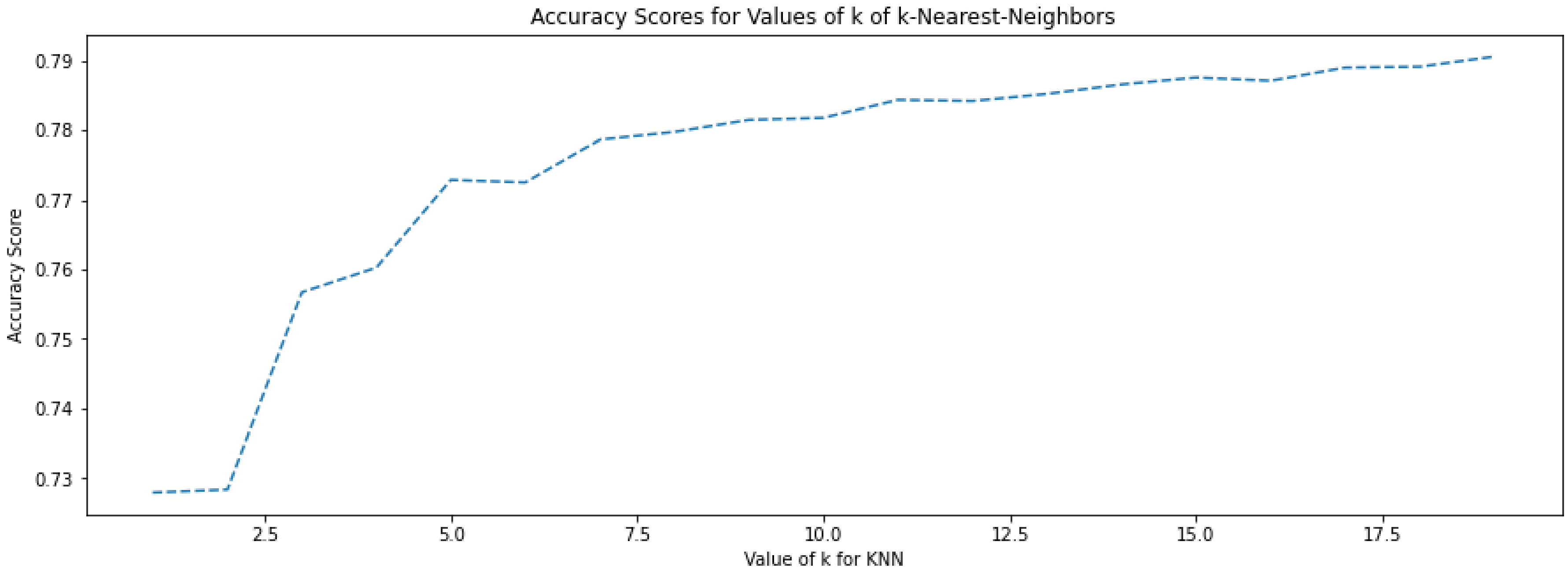
K-fold cross validation (the *k* is totally unrelated to *K* of KNN) involves randomly dividing the training set into *k* groups, or folds, of approximately equal size. The first fold is treated as a *validation set*, and the method is fit on the remaining *k−1* folds. The misclassification rate is then 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 [11]:

```python
# 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))
print(score)
# 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()
```

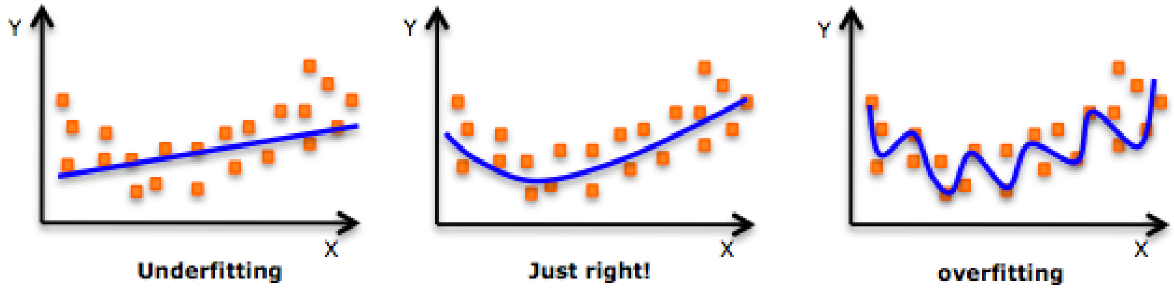
```
The optimal number of neighbors is 19
The optimal score is 0.79
[0.79118329 0.78588001 0.78183024 0.79011936 0.78481432 0.79807692
 0.78580902 0.79675066 0.79343501 0.79807692]
```



Splitting Data into Training Data and Testing Data

These are two rather important concepts in *data science* and *data analysis* and are used as *tools* to prevent (or at least minimize) **overfitting/underfitting**. For example, using a statistical model like linear regression, we usually fit the model on a training set in order to make predications on a data that wasn't trained (*general data*).

Overfitting means that we've fit the model too much to the training data. This model will be very accurate on the training data but will probably be very not accurate on untrained or new data. **Underfitting** means that we've fit the model too little to the training data. This model does not fit the training data and therefore misses the trends in the data.



It is worth noting the underfitting is not as prevalent as overfitting. These two problems makes the the model cannot be generalized to new data. Nevertheless, we want to avoid both of those problems in data analysis. You might say we are trying to find the middle ground between under and overfitting our model. As you will see, train/test split and cross validation help to avoid overfitting more than underfitting.

As such, the data are separated into the *training* data for modelling and *test* data for validating that model. Since the algorithms are **supervised** learning algorithms, the data *label* (**target class**) and the *data* itself need to be defined.

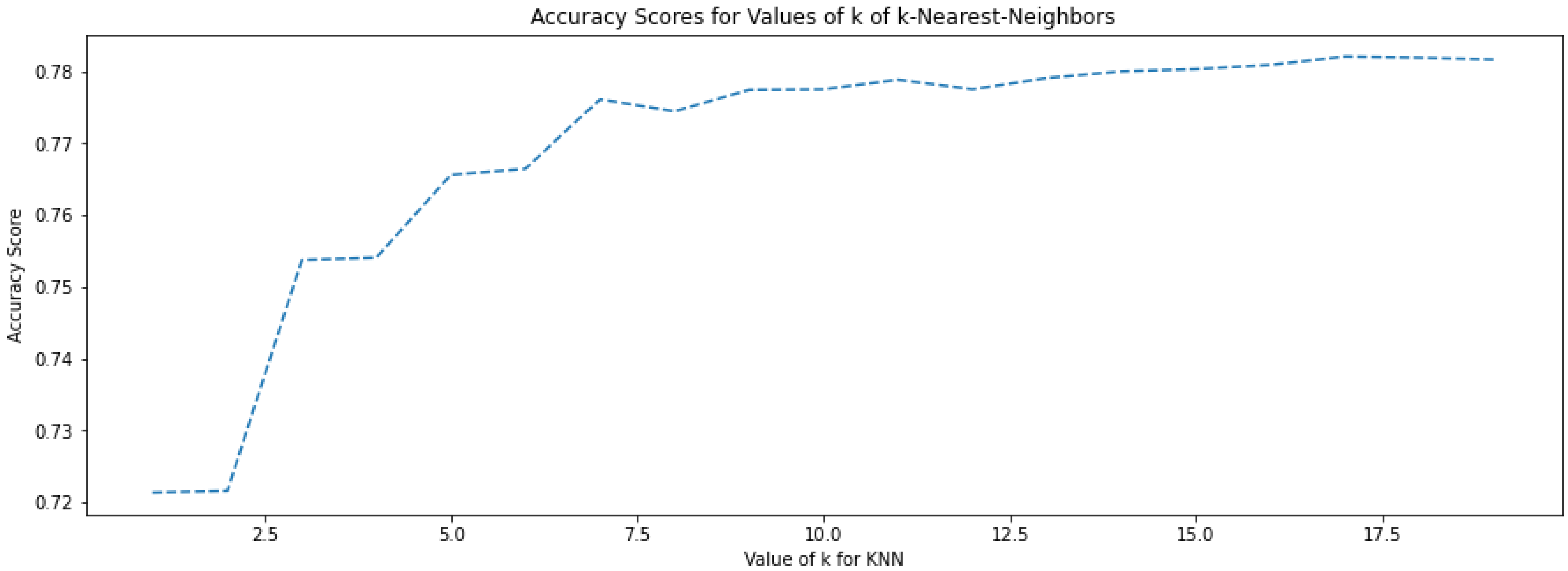
In [12]:
```
#Read data
df = pd.read_csv("input/adult_train_modified.csv")
```

In [13]:
```
from sklearn.model_selection import train_test_split

X = df.iloc[:,:-1]
Y = df.iloc[:, -1]

x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.40, random_state = 0)
```

In [14]:

```python
import matplotlib.pyplot as plt # library for plotting
import warnings # to hide unnecesary warning
warnings.filterwarnings('ignore')
# line required for inline charts/plots
%matplotlib inline

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

# We try different values of k for the KNN (from k=1 up to k=20)
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))
    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 17
The optimal score is 0.78
```



Accuracy Scores for Values of k of k-Nearest-Neighbors
Accuracy Score
0.72
0.73
0.74
0.75
0.76
0.77
0.78
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Value of k for KNN

In [ ]: