SQL

connect to IBM Cloud

First, you should install these two packages, then import them

In [ ]:

```
pip install pip install ibm_db_sa
pip install ibm_db
pip install sql
```

Then import them

In [3]:

```
import ibm_db_sa
import ibm_db
import sql
```

Then import sql

In [4]:

```
%load_ext sql
```

After creating an online database on IBM watson cloud, you should get the following information

In [1]:


```
{
  "db": "BLUDB",
  "dsn": "DATABASE=BLUDB;HOSTNAME=dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net;PORT=50000;PR
  "host": "dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net",
  "hostname": "dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net",
  "https_url": "https://dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net",
  "jdbcurl": "jdbc:db2://dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:50000/BLUDB",
  "parameters": {
    "role_crn": "crn:v1:bluemix:public:iam::::serviceRole:Manager"
  },
  "password": "sq8cjvf97v^qwggc",
  "port": 50000,
  "ssldsn": "DATABASE=BLUDB;HOSTNAME=dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net;PORT=50001
  "ssljdbcurl": "jdbc:db2://dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:50001/BLUDB:sslConn
  "uri": "db2://bl111369:sq8cjvf97v%5Eqwggc@dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:500
  "username": "bl111369"
}
```


Out[1]:


```
{'db': 'BLUDB',
 'dsn': 'DATABASE=BLUDB;HOSTNAME=dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.ne
t;PORT=50000;PROTOCOL=TCPIP;UID=bl111369;PWD=sq8cjvf97v^qwggc;',
 'host': 'dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net',
 'hostname': 'dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net',
 'https_url': 'https://dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net',
 'jdbcurl': 'jdbc:db2://dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:50000/B
LUDB',
 'parameters': {'role_crn': 'crn:v1:bluemix:public:iam::::serviceRole:Manager'},
 'password': 'sq8cjvf97v^qwggc',
 'port': 50000,
 'ssldsn': 'DATABASE=BLUDB;HOSTNAME=dashdb-txn-sbox-yp-dal09-14.services.dal.bluemi
x.net;PORT=50001;PROTOCOL=TCPIP;UID=bl111369;PWD=sq8cjvf97v^qwggc;Security=SSL;',
 'ssljdbcurl': 'jdbc:db2://dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:5000
1/BLUDB:sslConnection=true;',
 'uri': 'db2://bl111369:sq8cjvf97v%5Eqwggc@dashdb-txn-sbox-yp-dal09-14.services.dal.
bluemix.net:50000/BLUDB',
 'username': 'bl111369'}
```


We can operate sql database by directly connecting to IBM DB without installing an extra sql on our desktop, To do this, we should:

1. Upload database to IBM db
2. Connect to database through python

The code is shown as follow:

In [5]:

```
# Remember the connection string is of the format:
# %sql ibm_db_sa://my-username:my-password@my-hostname:my-port/my-db-name
# Enter the connection string for your Db2 on Cloud database instance below
%sql ibm_db_sa://bl111369:sq8cjvf97v^qwggc@dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:50(
%sql db2://bl111369:sq8cjvf97v%5Eqwggc@dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:50000/I
```

Then we can try a simple command:

In [6]:

```
%sql SELECT count(*) FROM CHICAGO_CRIME_DATA
```

```
 * db2://bl111369:***@dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:50000/BLU
DB
   ibm_db_sa://bl111369:***@dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:500
00/BLUDB
Done.
```

Out[6]:

1
533

It worked!

In [7]:

```
%sql select * from CHICAGO_PUBLIC_SCHOOLS where community_area_number between 10 and 15
```

```
 * db2://bl111369:***@dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:50000/B
LUDB
   ibm_db_sa://bl111369:***@dashdb-txn-sbox-yp-dal09-14.services.dal.bluemix.net:5
0000/BLUDB
Done.
```

For more detailed operations of sql, please refer to https://www.coursera.org/learn/analytics-mysql#syllabus (https://www.coursera.org/learn/analytics-mysql#syllabus)

Use python to process data

To construct the model, we should do the following steps:

1. Load and clean the data
2. Observe the dataset and choose the right model
3. create train and test set
4. Model evaluation

For data wrangling and model selection, it largely based on personal experience and the requirement.

Creating train and test dataset

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 have been used to train the data. It is more realistic for real world problems.

This means that we know the outcome of each data point in this dataset, making it great to test with! And since this data has not been used to train the model, the model has no knowledge of the outcome of these data points. So, in essence, it is truly an out-of-sample testing.

Model evaluation

There are different model evaluation metrics for Regression analysis.

- Mean absolute error: It is the mean of the absolute value of the errors. This is the easiest of the metrics to understand since it's just average error.
- Mean Squared Error (MSE): Mean Squared Error (MSE) is the mean of the squared error. It's more popular than Mean absolute error because the focus is geared more towards large errors. This is due to the squared term exponentially increasing larger errors in comparison to smaller ones.
- Root Mean Squared Error (RMSE): This is the square root of the Mean Square Error.
- R-squared is not error, but is a popular metric for accuracy of your model. It represents how close the data are to the fitted regression line. The higher the R-squared, the better the model fits your data. Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse).

Jaccard index

We can define jaccard as the size of the intersection divided by the size of the union of two label sets. If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0; otherwise it is 0.0.

Load data

Import essential packages at first

In [1]:

```
import pandas as pd
import pylab as pl
import numpy as np
import scipy.optimize as opt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
%matplotlib inline
import matplotlib.pyplot as plt
```

You could either load csv and excel files from your drive or internet

In [17]:

```
#From internet
other_path = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsN
df = pd.read_csv(other_path, header=None)
headers = ["symboling","normalized-losses","make","fuel-type","aspiration", "num-of-doors","body-sty
           "drive-wheels","engine-location","wheel-base", "length","width","height","curb-weight","eng
           "num-of-cylinders", "engine-size","fuel-system","bore","stroke","compression-ratio","horsep
           "peak-rpm","city-mpg","highway-mpg","price"]
df.columns = headers
```

In [18]:

```
#It can also be written like this
filename = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNet
headers = ["symboling","normalized-losses","make","fuel-type","aspiration", "num-of-doors","body-sty
           "drive-wheels","engine-location","wheel-base", "length","width","height","curb-weight","eng
           "num-of-cylinders", "engine-size","fuel-system","bore","stroke","compression-ratio","horsep
           "peak-rpm","city-mpg","highway-mpg","price"]
df = pd.read_csv(filename, names = headers)
```

In [9]:

```
#From drive
df=pd.read_csv('C:/Users/ronni/IBM data/machine learning/cars_clus.csv')
```

After the operation, we can save the dataset

In [8]:

```
df.to_csv("automobile.csv", index=False)
```

We can then have some basic insight of dataset

Display the first five rows of dataframe

In [19]:

```
df.head()
```

Out[19]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

5 rows × 26 columns

Data Types

Data has a variety of types.
The main types stored in Pandas dataframes are **object**, **float**, **int**, **bool** and **datetime64**. In order to better learn about each attribute, it is always good for us to know the data type of each column. In Pandas:

In [20]:

```
df.dtypes
```

Out[20]:

```
symboling              int64
normalized-losses     object
make                  object
fuel-type             object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                  object
stroke                object
compression-ratio    float64
horsepower            object
peak-rpm              object
city-mpg               int64
highway-mpg            int64
price                 object
dtype: object
```

Describe

If we would like to get a statistical summary of each column, such as count, column mean value, column standard deviation, etc. We use the describe method:

In [21]:

```
df.describe()
```

Out[21]:

	symboling	wheel-base	length	width	height	curb-weight	engine-size	c
count	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	
mean	0.834146	98.756585	174.049268	65.907805	53.724878	2555.565854	126.907317	
std	1.245307	6.021776	12.337289	2.145204	2.443522	520.680204	41.642693	
min	-2.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	
25%	0.000000	94.500000	166.300000	64.100000	52.000000	2145.000000	97.000000	
50%	1.000000	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	
75%	2.000000	102.400000	183.100000	66.900000	55.500000	2935.000000	141.000000	
max	3.000000	120.900000	208.100000	72.300000	59.800000	4066.000000	326.000000	

In [14]:

```
df.describe(include = "all")
```

Out[14]:

	manufact	model	sales	resale	type	price	engine_s	horsepow	wheelbas	width
count	157	159	159	159	159	159	159	159	159	159
unique	30	158	158	118	3	154	32	69	90	80
top	Ford	Neon	*null*	*null*	0.000	18.890	2.000	150.000	112.200	66.700
freq	11	2	2	38	116	2	17	9	8	6
mean	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
std	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
min	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
25%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
50%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
75%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
max	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Info

In [ ]:

```
df.info
```

Data Wrangling

After we have a basic insight of dataset, we should process the data until it can be used for further analysis.

There is a general process to do this job.

- Identify and handle missing values
  - Identify missing values
  - Deal with missing values
  - Correct data format
- Data standardization
- Data Normalization (centering/scaling)
- Binning
- Indicator variable

Identify and handle missing values

Convert "?" to NaN

In this car dataset, missing data comes with the question mark "?". We replace "?" with NaN (Not a Number), which is Python's default missing value marker, for reasons of computational speed and convenience. Here we use the function:

```
.replace(A, B, inplace = True)
```

to replace A by B

In [23]:

```
# replace "?" to NaN
df.replace("?", np.nan, inplace = True)
df.head(5)
```

Out[23]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

5 rows × 26 columns

dentify_missing_values

Evaluating for Missing Data

The missing values are converted to Python's default. We use Python's built-in functions to identify these missing values. There are two methods to detect missing data:

1. **.isnull()**

2. **.notnull()**

The output is a boolean value indicating whether the value that is passed into the argument is in fact missing data.

In [24]:

```
missing_data = df.isnull()
missing_data.head(5)
```

Out[24]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...
0	False	True	False	False	False	False	False	False	False	False	...
1	False	True	False	False	False	False	False	False	False	False	...
2	False	True	False	False	False	False	False	False	False	False	...
3	False	False	False	False	False	False	False	False	False	False	...
4	False	False	False	False	False	False	False	False	False	False	...

5 rows × 26 columns

"True" stands for missing value, while "False" stands for not missing value.

Count missing values in each column

Using a for loop in Python, we can quickly figure out the number of missing values in each column. As mentioned above, "True" represents a missing value, "False" means the value is present in the dataset. In the body of the for loop the method ".value_counts()" counts the number of "True" values.

In [26]:

```
for column in missing_data.columns.values.tolist():
    print(column)
    print (missing_data[column].value_counts())
    print("")
```

```
symboling
False    205
Name: symboling, dtype: int64

normalized-losses
False    164
True      41
Name: normalized-losses, dtype: int64

make
False    205
Name: make, dtype: int64

fuel-type
False    205
Name: fuel-type, dtype: int64

aspiration
False    205
```

Based on the summary above, each column has 205 rows of data, seven columns containing missing data:

1. "normalized-losses": 41 missing data
2. "num-of-doors": 2 missing data
3. "bore": 4 missing data
4. "stroke" : 4 missing data
5. "horsepower": 2 missing data
6. "peak-rpm": 2 missing data
7. "price": 4 missing data

Deal with missing data

How to deal with missing data?

1. drop data
   a. drop the whole row
   b. drop the whole column
2. replace data
   a. replace it by mean
   b. replace it by frequency
   c. replace it based on other functions

Whole columns should be dropped only if most entries in the column are empty. In our dataset, none of the columns are empty enough to drop entirely. We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns:

Replace by mean:

- "normalized-losses": 41 missing data, replace them with mean
- "stroke": 4 missing data, replace them with mean
- "bore": 4 missing data, replace them with mean
- "horsepower": 2 missing data, replace them with mean
- "peak-rpm": 2 missing data, replace them with mean

Replace by frequency:

- "num-of-doors": 2 missing data, replace them with "four".
  - Reason: 84% sedans is four doors. Since four doors is most frequent, it is most likely to occur

Drop the whole row:

- "price": 4 missing data, simply delete the whole row
  - Reason: price is what we want to predict. Any data entry without price data cannot be used for prediction; therefore any row now without price data is not useful to us

Calculate the average of the column

In [27]:

```
avg_norm_loss = df["normalized-losses"].astype("float").mean(axis=0)
print("Average of normalized-losses:", avg_norm_loss)
```

```
Average of normalized-losses: 122.0
```

Replace "NaN" by mean value in "normalized-losses" column

In [28]:

```
df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)
```

We can replace the remaining 'NaN' like this

In [29]:

```
avg_bore=df['bore'].astype('float').mean(axis=0)
avg_horsepower = df['horsepower'].astype('float').mean(axis=0)
avg_peakrpm=df['peak-rpm'].astype('float').mean(axis=0)
avg_stroke = df["stroke"].astype("float").mean(axis = 0)
```

In [30]:

```
df["bore"].replace(np.nan, avg_bore, inplace=True)
df['horsepower'].replace(np.nan, avg_horsepower, inplace=True)
df['peak-rpm'].replace(np.nan, avg_peakrpm, inplace=True)
df["stroke"].replace(np.nan, avg_stroke, inplace = True)
```

Then we replace the 'NaN' by frequency

To find the frequency, we can use code like this:

In [32]:

```
df['num-of-doors'].value_counts()
```

Out[32]:

```
four    114
two      89
Name: num-of-doors, dtype: int64
```

Or like this:

In [33]:

```
df['num-of-doors'].value_counts().idxmax()
```

Out[33]:

```
'four'
```

In [34]:

```
#replace the missing 'num-of-doors' values by the most frequent
df["num-of-doors"].replace(np.nan, "four", inplace=True)
```

Finally, let's drop all rows that do not have price data:

In [35]:

```
# simply drop whole row with NaN in "price" column
df.dropna(subset=["price"], axis=0, inplace=True)

# reset index, because we droped two rows
df.reset_index(drop=True, inplace=True)
```

In [36]:

```
df.head()
```

Out[36]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

5 rows × 26 columns

Correct data format

The last step in data cleaning is checking and making sure that all data is in the correct format (int, float, text or other).

In Pandas, we use

.dtype() to check the data type

.astype() to change the data type

Let's check the data type for each colomns

In [37]:

```
df.dtypes
```

Out[37]:

```
symboling              int64
normalized-losses     object
make                  object
fuel-type             object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                  object
stroke                object
compression-ratio    float64
horsepower            object
peak-rpm              object
city-mpg               int64
highway-mpg            int64
price                 object
dtype: object
```

As we can see above, some columns are not of the correct data type. Numerical variables should have type 'float' or 'int', and variables with strings such as categories should have type 'object'. For example, 'bore' and 'stroke' variables are numerical values that describe the engines, so we should expect them to be of the type 'float' or 'int'; however, they are shown as type 'object'. We have to convert data types into a proper format for each column using the "astype()" method.

In [38]:

```
df[["bore", "stroke"]] = df[["bore", "stroke"]].astype("float")
df[["normalized-losses"]] = df[["normalized-losses"]].astype("int")
df[["price"]] = df[["price"]].astype("float")
df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")
```

List the data type again

In [39]:

```
df.dtypes
```

Out[39]:

```
symboling              int64
normalized-losses      int32
make                  object
fuel-type             object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                 float64
stroke               float64
compression-ratio    float64
horsepower            object
peak-rpm             float64
city-mpg               int64
highway-mpg            int64
price                float64
dtype: object
```

Data Standardization

Data is usually collected from different agencies with different formats. (Data Standardization is also a term for a particular type of data normalization, where we subtract the mean and divide by the standard deviation)

Standardization is the process of transforming data into a common format which allows the researcher to make the meaningful comparison.

Example

Transform mpg to L/100km:

In [40]:

```
# Convert mpg to L/100km by mathematical operation (235 divided by mpg)
df['city-L/100km'] = 235/df["city-mpg"]

# check your transformed data
df.head()
```

Out[40]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

5 rows × 27 columns

In [41]:

```
# transform mpg to L/100km by mathematical operation (235 divided by mpg)
df["highway-mpg"] = 235/df["highway-mpg"]

# rename column name from "highway-mpg" to "highway-L/100km"
df.rename(columns={'"highway-mpg"':'highway-L/100km'}, inplace=True)

# check your transformed data
df.head()
```

Out[41]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

5 rows × 27 columns

Data Normalization

Why normalization?

Normalization is the process of transforming values of several variables into a similar range. Typical normalizations include scaling the variable so the variable average is 0, scaling the variable so the variance is 1, or scaling variable so the variable values range from 0 to 1

Example

To demonstrate normalization, let's say we want to scale the columns "length", "width" and "height"

Target:would like to Normalize those variables so their value ranges from 0 to 1.

Approach: replace original value by (original value)/(maximum value)

In [42]:

```
df['length'] = df['length']/df['length'].max()
df['width'] = df['width']/df['width'].max()
df['height'] = df['height']/df['height'].max()
# show the scaled columns
df[["length","width","height"]].head()
```

Out[42]:

	length	width	height
0	0.811148	0.890278	0.816054
1	0.811148	0.890278	0.816054
2	0.822681	0.909722	0.876254
3	0.848630	0.919444	0.908027
4	0.848630	0.922222	0.908027

In [43]:

```
df['highway-mpg']
```

Out[43]:

```
0       8.703704
1       8.703704
2       9.038462
3       7.833333
4      10.681818
         ...
196     8.392857
197     9.400000
198    10.217391
199     8.703704
200     9.400000
Name: highway-mpg, Length: 201, dtype: float64
```

Binning

Why binning?

Binning is a process of transforming continuous numerical variables into discrete categorical 'bins', for grouped analysis.

Example:

In our dataset, "horsepower" is a real valued variable ranging from 48 to 288, it has 57 unique values. What if we only care about the price difference between cars with high horsepower, medium horsepower, and little horsepower (3 types)? Can we rearrange them into three ‘bins' to simplify analysis?

We will use the Pandas method 'cut' to segment the 'horsepower' column into 3 bins

In [45]:

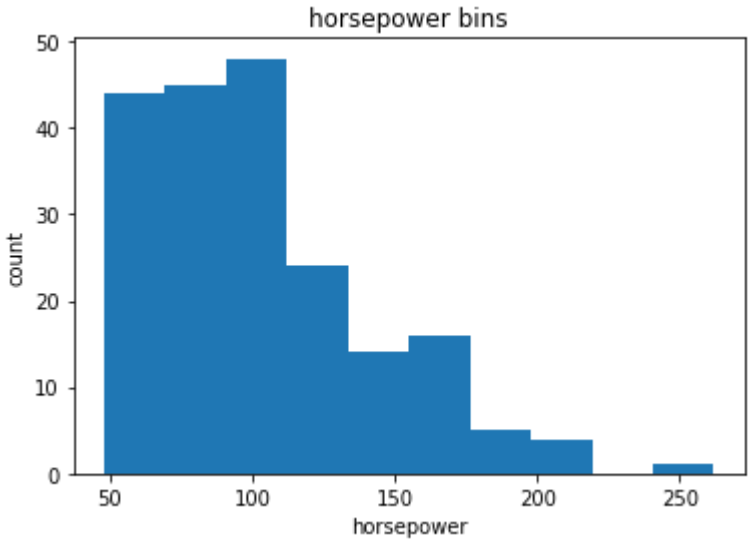
```
df["horsepower"]=df["horsepower"].astype(int, copy=True)
```

In [46]:

```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
plt.pyplot.hist(df["horsepower"])

# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

Out[46]:

```
Text(0.5, 1.0, 'horsepower bins')
```



We would like 3 bins of equal size bandwidth so we use numpy's `linspace(start_value, end_value, numbers_generated` function.

Since we want to include the minimum value of horsepower we want to set start_value=min(df["horsepower"]).

Since we want to include the maximum value of horsepower we want to set end_value=max(df["horsepower"]).

Since we are building 3 bins of equal length, there should be 4 dividers, so numbers_generated=4.

We build a bin array, with a minimum value to a maximum value, with bandwidth calculated above. The bins will be values used to determine when one bin ends and another begins.

In [47]:

```
bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)
bins
```

Out[47]:

```
array([ 48.        , 119.33333333, 190.66666667, 262.        ])
```

Then set the group names.

In [48]:

```
group_names = ['Low', 'Medium', 'High']
```

We apply the function "cut" the determine what each value of "df['horsepower']" belongs to.

In [49]:

```
df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names, include_lowest=True )
df[['horsepower','horsepower-binned']].head(20)
```

Out[49]:

	horsepower	horsepower-binned
0	111	Low
1	111	Low
2	154	Medium
3	102	Low
4	115	Low
5	110	Low
6	110	Low
7	110	Low
8	140	Medium
9	101	Low
10	101	Low
11	121	Medium
12	121	Medium
13	121	Medium
14	182	Medium
15	182	Medium
16	182	Medium
17	48	Low
18	70	Low
19	70	Low

In [50]:

```
df["horsepower-binned"].value_counts()
```

Out[50]:

```
Low       153
Medium     43
High        5
Name: horsepower-binned, dtype: int64
```

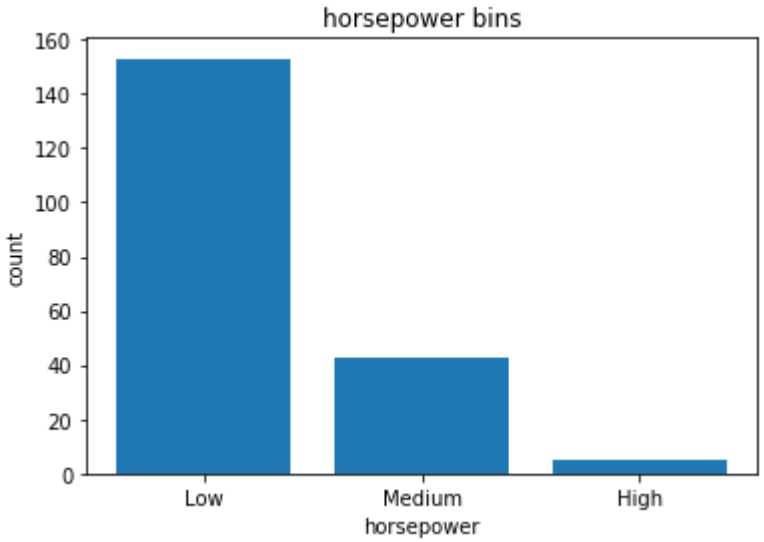
Lets plot the distribution of each bin.

In [51]:

```
pyplot.bar(group_names, df["horsepower-binned"].value_counts())

# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

Out[51]:

```
Text(0.5, 1.0, 'horsepower bins')
```



you will find the last column provides the bins for "horsepower" with 3 categories ("Low","Medium" and "High").

Indicator variable (or dummy variable)

What is an indicator variable?

An indicator variable (or dummy variable) is a numerical variable used to label categories. They are called 'dummies' because the numbers themselves don't have inherent meaning.

Why we use indicator variables?

So we can use categorical variables for regression analysis in the later modules.

Example

We see the column "fuel-type" has two unique values, "gas" or "diesel". Regression doesn't understand words, only numbers. To use this attribute in regression analysis, we convert "fuel-type" into indicator variables.

We will use the panda's method 'get_dummies' to assign numerical values to different categories of fuel type.

In [53]:

```
df.columns
```

Out[53]:

```
Index(['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration',
       'num-of-doors', 'body-style', 'drive-wheels', 'engine-location',
       'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type',
       'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke',
       'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',
       'highway-mpg', 'price', 'city-L/100km', 'horsepower-binned'],
      dtype='object')
```

get indicator variables and assign it to data frame "dummy_variable_1"

In [54]:

```
dummy_variable_1 = pd.get_dummies(df["fuel-type"])
dummy_variable_1.head()
```

Out[54]:

	diesel	gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

change column names for clarity

In [55]:

```
dummy_variable_1.rename(columns={'fuel-type-diesel':'gas', 'fuel-type-diesel':'diesel'}, inplace=Tru
dummy_variable_1.head()
```

Out[55]:

	diesel	gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

We now have the value 0 to represent "gas" and 1 to represent "diesel" in the column "fuel-type". We will now insert this column back into our original dataset.

In [56]:

```
# merge data frame "df" and "dummy_variable_1"
df = pd.concat([df, dummy_variable_1], axis=1)

# drop original column "fuel-type" from "df"
df.drop("fuel-type", axis = 1, inplace=True)
df.head()
```

Out[56]:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	l
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.8
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.8
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.8
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.8
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.8

5 rows × 29 columns

We can do the same thing to aspiration.

In [57]:

```
# get indicator variables of aspiration and assign it to data frame "dummy_variable_2"
dummy_variable_2 = pd.get_dummies(df['aspiration'])

# change column names for clarity
dummy_variable_2.rename(columns={'std':'aspiration-std', 'turbo': 'aspiration-turbo'}, inplace=True

# show first 5 instances of data frame "dummy_variable_1"
dummy_variable_2.head()

#merge the new dataframe to the original datafram
df = pd.concat([df, dummy_variable_2], axis=1)

# drop original column "aspiration" from "df"
df.drop('aspiration', axis = 1, inplace=True)
```

Finally, save the clean csv

In [58]:

```
df.to_csv('clean_df.csv')
```

Models for machine learning

Regression

We will introduce

1. **Simple Linear Regression**
2. **Multiple Linear Regression**
3. **Nonlinear Regression**
4. **Polynominal Regression**.

Simple Linear Regression

We will first read the dataset and have some quick insight

In [63]:

```
import matplotlib.pyplot as plt
import pylab as pl
%matplotlib inline
```

In [59]:

```
df=pd.read_csv('C:/Users/ronni/IBM data/machine learning/FuelConsumption.csv')
```

In [61]:

```
df.head()
```

Out[61]:

	MODELYEAR	MAKE	MODEL	VEHICLECLASS	ENGINESIZE	CYLINDERS	TRANSMISSION
0	2014	ACURA	ILX	COMPACT	2.0	4	AS5
1	2014	ACURA	ILX	COMPACT	2.4	4	M6
2	2014	ACURA	ILX HYBRID	COMPACT	1.5	4	AV7
3	2014	ACURA	MDX 4WD	SUV - SMALL	3.5	6	AS6
4	2014	ACURA	RDX AWD	SUV - SMALL	3.5	6	AS6

In [65]:

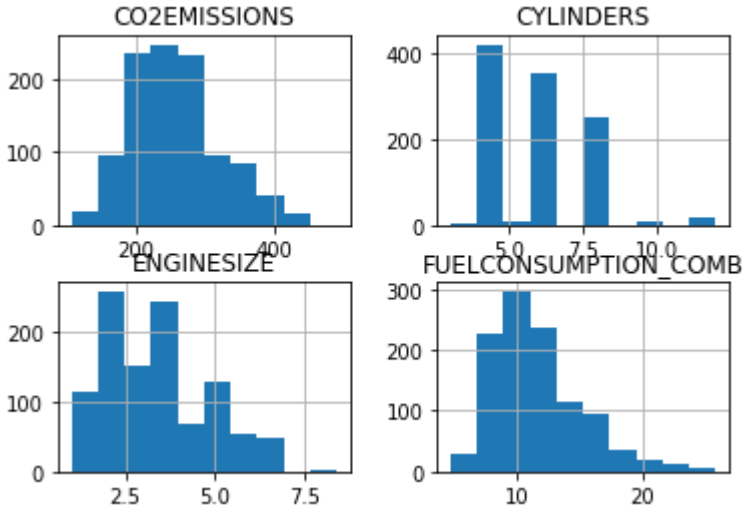
```
cdf = df[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION_COMB','CO2EMISSIONS']]
cdf.head(9)
```

Out[65]:

	ENGINESIZE	CYLINDERS	FUELCONSUMPTION_COMB	CO2EMISSIONS
0	2.0	4	8.5	196
1	2.4	4	9.6	221
2	1.5	4	5.9	136
3	3.5	6	11.1	255
4	3.5	6	10.6	244
5	3.5	6	10.0	230
6	3.5	6	10.1	232
7	3.7	6	11.1	255
8	3.7	6	11.6	267

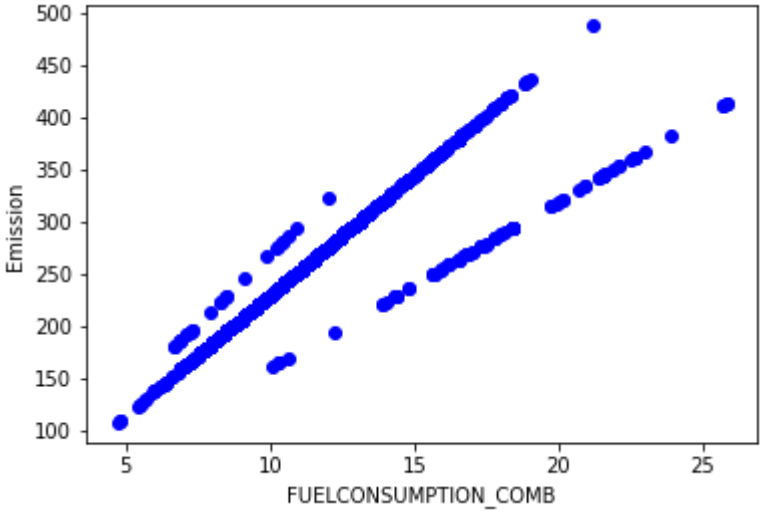
In [66]:

```
viz = cdf[['CYLINDERS','ENGINESIZE','CO2EMISSIONS','FUELCONSUMPTION_COMB']]
viz.hist()
plt.show()
```



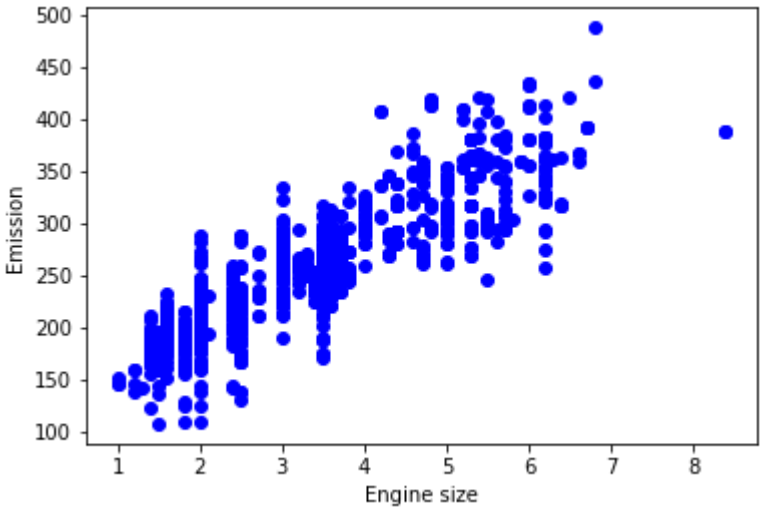
In [67]:

```
plt.scatter(cdf.FUELCONSUMPTION_COMB, cdf.CO2EMISSIONS,  color='blue')
plt.xlabel("FUELCONSUMPTION_COMB")
plt.ylabel("Emission")
plt.show()
```



In [68]:

```
plt.scatter(cdf.ENGINESIZE, cdf.CO2EMISSIONS,  color='blue')
plt.xlabel("Engine size")
plt.ylabel("Emission")
plt.show()
```



Lets split our dataset into train and test sets, 80% of the entire data for training, and the 20% for testing. We create a mask to select random rows using **np.random.rand()** function:

In [69]:

```
msk = np.random.rand(len(df)) < 0.8
train = cdf[msk]
test = cdf[~msk]
```

Use sklearn packages to model a data

In [70]:

```
from sklearn import linear_model
regr = linear_model.LinearRegression()
train_x = np.asanyarray(train[['ENGINESIZE']])
train_y = np.asanyarray(train[['CO2EMISSIONS']])
regr.fit (train_x, train_y)
# The coefficients
print ('Coefficients: ', regr.coef_)
print ('Intercept: ',regr.intercept_)
```

```
Coefficients:  [[38.62646207]]
Intercept:  [126.29302233]
```

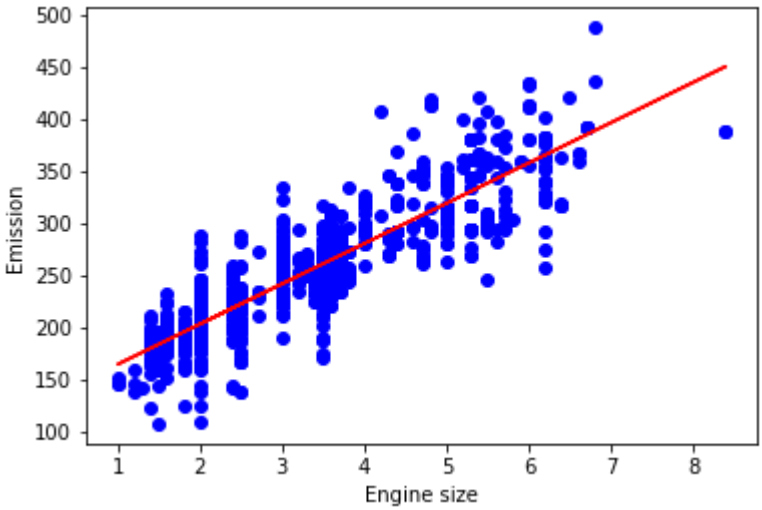
We can plot the fit line over the data.

In [71]:

```
plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS,  color='blue')
plt.plot(train_x, regr.coef_[0][0]*train_x + regr.intercept_[0], '-r')
plt.xlabel("Engine size")
plt.ylabel("Emission")
```

Out[71]:

```
Text(0, 0.5, 'Emission')
```



Then evaluate the model.

In [72]:

```
from sklearn.metrics import r2_score

test_x = np.asanyarray(test[['ENGINESIZE']])
test_y = np.asanyarray(test[['CO2EMISSIONS']])
test_y_hat = regr.predict(test_x)

print("Mean absolute error: %.2f" % np.mean(np.absolute(test_y_hat - test_y)))
print("Residual sum of squares (MSE): %.2f" % np.mean((test_y_hat - test_y) ** 2))
print("R2-score: %.2f" % r2_score(test_y_hat , test_y) )
```

```
Mean absolute error: 25.63
Residual sum of squares (MSE): 1105.40
R2-score: 0.63
```

Multiple Linear Regression

In reality, there are multiple variables that predict the Co2emission. When more than one independent variable is present, the process is called multiple linear regression. For example, predicting co2emission using FUELCONSUMPTION_COMB, EngineSize and Cylinders of cars. The good thing here is that Multiple linear regression is the extension of simple linear regression model.

We will set the train and test sets, just like before.

In [73]:

```
msk = np.random.rand(len(df)) < 0.8
train = cdf[msk]
test = cdf[~msk]
```

Then model them.

In [74]:

```
regr = linear_model.LinearRegression()
x = np.asanyarray(train[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION_COMB']])
y = np.asanyarray(train[['CO2EMISSIONS']])
regr.fit (x, y)
# The coefficients
print ('Coefficients: ', regr.coef_)
```

```
Coefficients:  [[11.11353819  7.24661861  9.4721754 ]]
```

We can also do the prediction

In [75]:

```
y_hat= regr.predict(test[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION_COMB']])
x = np.asanyarray(test[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION_COMB']])
y = np.asanyarray(test[['CO2EMISSIONS']])
print("Residual sum of squares: %.2f"
      % np.mean((y_hat - y) ** 2))

# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % regr.score(x, y))
```

```
Residual sum of squares: 643.13
Variance score: 0.87
```

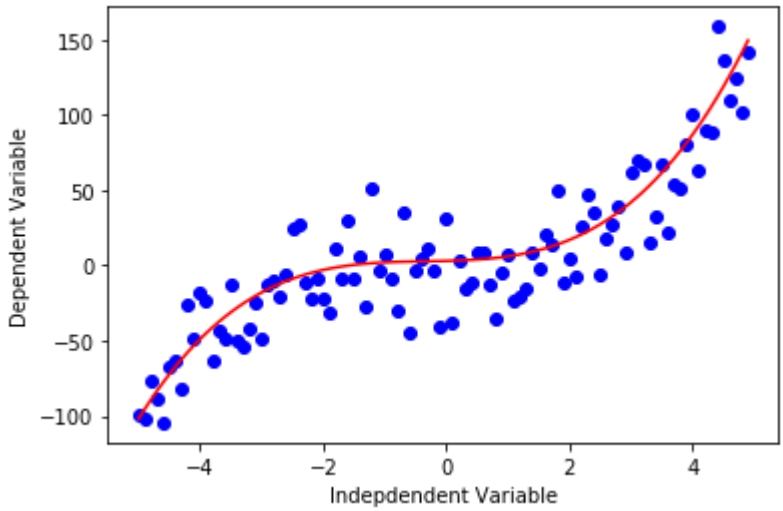
Nonlinear Regression

Let's see some simple Nonlinear Regression

Cubic

In [76]:

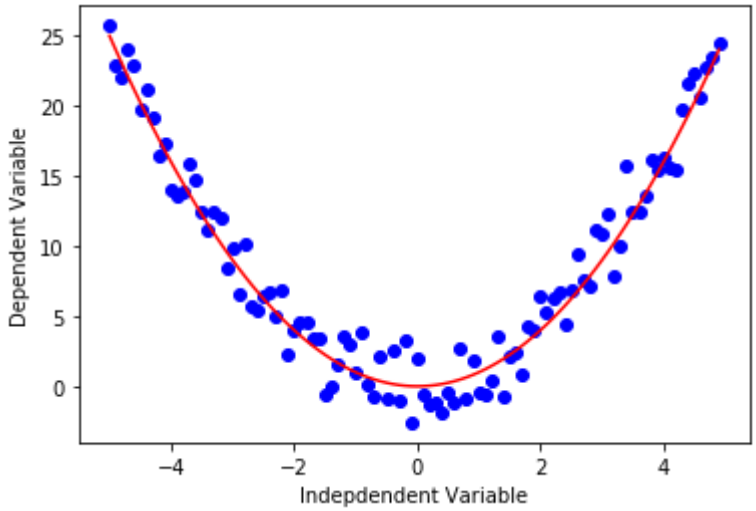
```
x = np.arange(-5.0, 5.0, 0.1)


y = 1*(x**3) + 1*(x**2) + 1*x + 3
y_noise = 20 * np.random.normal(size=x.size)
ydata = y + y_noise
plt.plot(x, ydata,  'bo')
plt.plot(x,y, 'r')
plt.ylabel('Dependent Variable')
plt.xlabel('Indepdendent Variable')
plt.show()
```



Quadratic

In [77]:

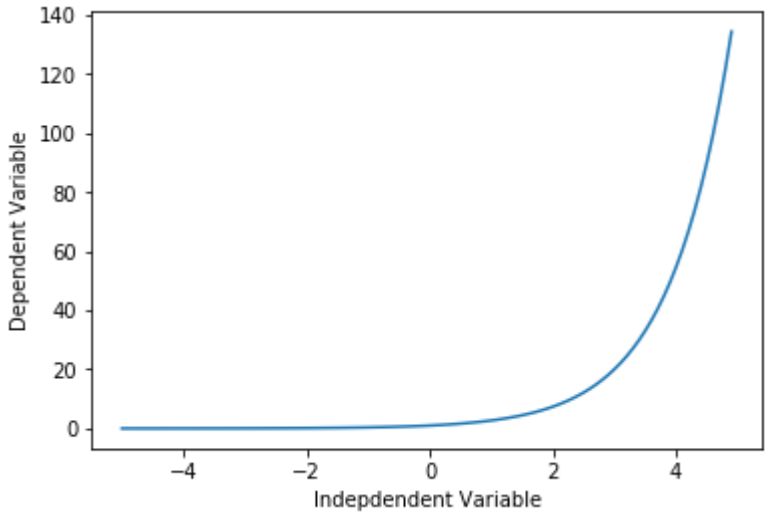
```
x = np.arange(-5.0, 5.0, 0.1)



y = np.power(x,2)
y_noise = 2 * np.random.normal(size=x.size)
ydata = y + y_noise
plt.plot(x, ydata,  'bo')
plt.plot(x,y, 'r')
plt.ylabel('Dependent Variable')
plt.xlabel('Indepdendent Variable')
plt.show()
```



Exponential

In [83]:

```
X = np.arange(-5.0, 5.0, 0.1)

Y= np.exp(X)

plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Indepdendent Variable')
plt.show()
```

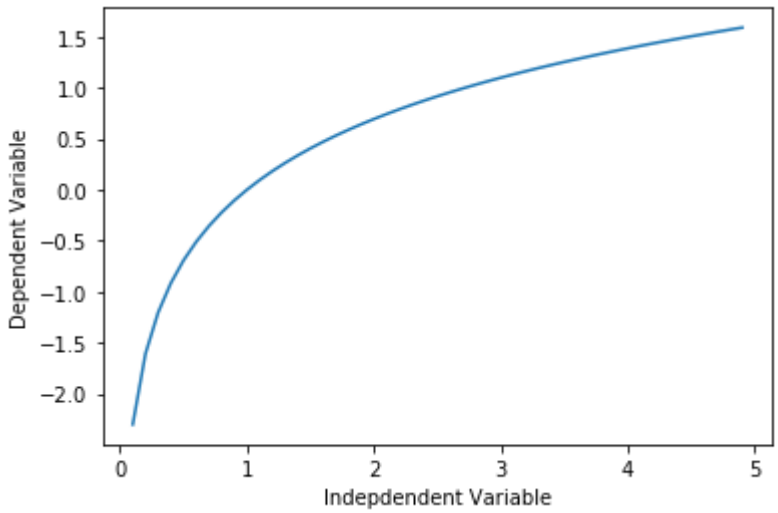


Logarithmic

In [84]:

```
X = np.arange(-5.0, 5.0, 0.1)

Y = np.log(X)

plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Indepdendent Variable')
plt.show()
```

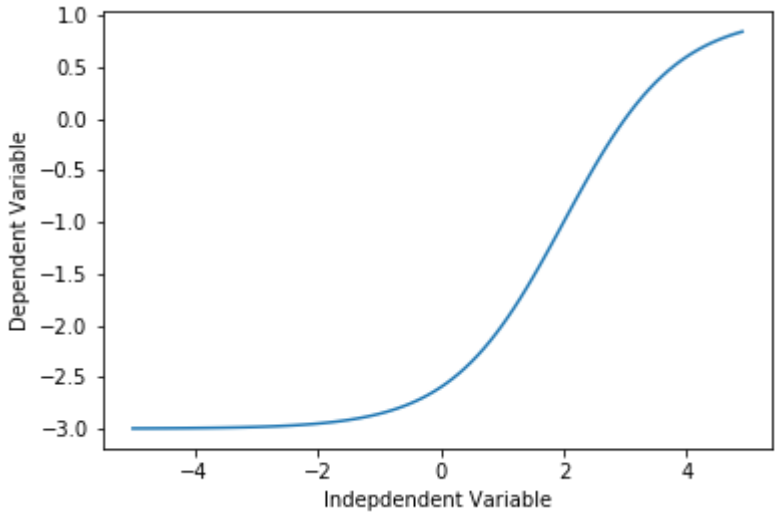
```
C:\Users\ronni\anaconda3\lib\site-packages\ipykernel_launcher.py:3: RuntimeWarning:
invalid value encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
```



Sigmoidal/Logistic

In [85]:

```
X = np.arange(-5.0, 5.0, 0.1)


Y = 1-4/(1+np.power(3, X-2))

plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Indepdendent Variable')
plt.show()
```



First, let's read the dataset.

In [87]:

```
df=pd.read_csv('C:/Users/ronni/IBM data/machine learning/china_gdp.csv')
df.head()
```

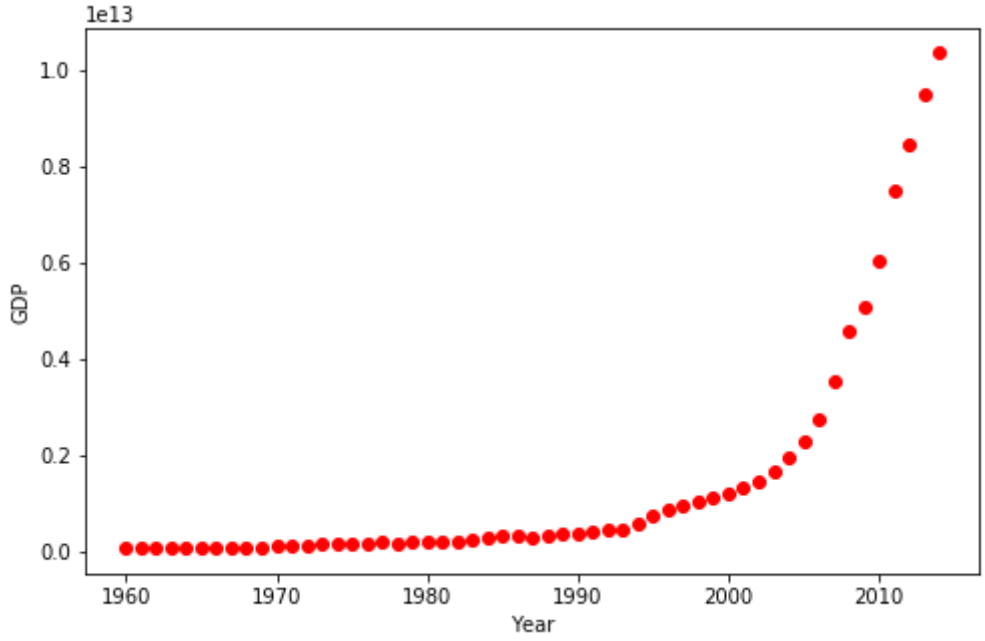
Out[87]:

	Year	Value
0	1960	5.918412e+10
1	1961	4.955705e+10
2	1962	4.668518e+10
3	1963	5.009730e+10
4	1964	5.906225e+10

We can have a overview of dataset

In [88]:

```
plt.figure(figsize=(8,5))
x_data, y_data = (df["Year"].values, df["Value"].values)
plt.plot(x_data, y_data, 'ro')
plt.ylabel('GDP')
plt.xlabel('Year')
plt.show()
```



From an initial look at the plot, we determine that the logistic function could be a good approximation, since it has the property of starting with a slow growth, increasing growth in the middle, and then decreasing again at the end; as illustrated below:

Now, let's build our regression model and initialize its parameters.

In [89]:

```
def sigmoid(x, Beta_1, Beta_2):
    y = 1 / (1 + np.exp(-Beta_1*(x-Beta_2)))
    return y
```

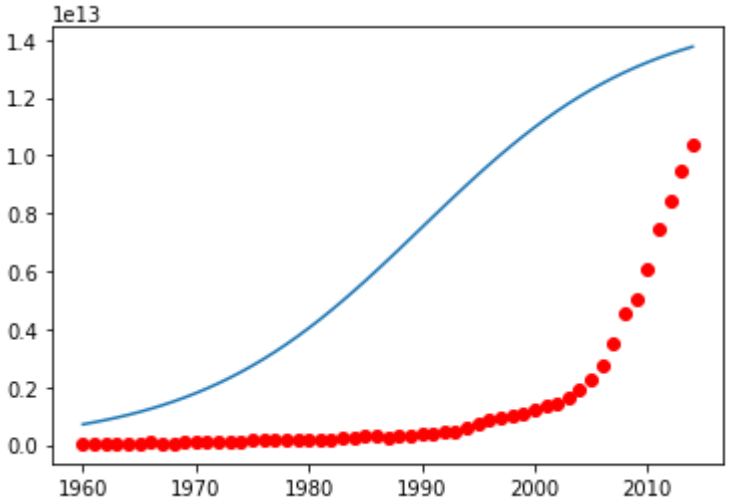
Lets look at a sample sigmoid line that might fit with the data:

In [90]:

```
beta_1 = 0.10
beta_2 = 1990.0

#logistic function
Y_pred = sigmoid(x_data, beta_1 , beta_2)

#plot initial prediction against datapoints
plt.plot(x_data, Y_pred*15000000000000.)
plt.plot(x_data, y_data, 'ro')
```

Out[90]:

```
[<matplotlib.lines.Line2D at 0x15c1bd55908>]
```



Our task here is to find the best parameters for our model. Lets first normalize our x and y:

In [91]:

```
xdata =x_data/max(x_data)
ydata =y_data/max(y_data)
```

How we find the best parameters for our fit line?

we can use **curve_fit** which uses non-linear least squares to fit our sigmoid function, to data. Optimal values for the parameters so that the sum of the squared residuals of sigmoid(xdata, *popt) - ydata is minimized.

popt are our optimized parameters.

In [96]:

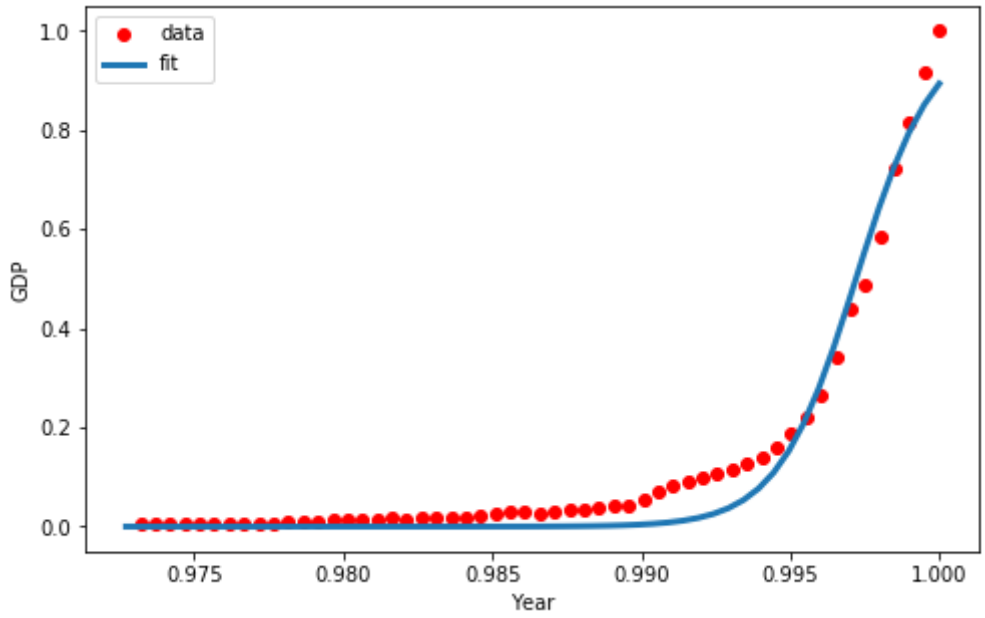
```
# split data into train/test
msk = np.random.rand(len(df)) < 0.8
train_x = xdata[msk]
test_x = xdata[~msk]
train_y = ydata[msk]
test_y = ydata[~msk]

# build the model using train set
popt, pcov = curve_fit(sigmoid, train_x, train_y)

# predict using test set
y_hat = sigmoid(test_x, *popt)

# evaluation
print("Mean absolute error: %.2f" % np.mean(np.absolute(y_hat - test_y)))
print("Residual sum of squares (MSE): %.2f" % np.mean((y_hat - test_y) ** 2))
from sklearn.metrics import r2_score
print("R2-score: %.2f" % r2_score(y_hat , test_y) )
```

```
Mean absolute error: 0.04
Residual sum of squares (MSE): 0.00
R2-score: 0.92
```

Then we can plot the result.

In [98]:

```
x = np.linspace(1960, 2015, 55)
x = x/max(x)
plt.figure(figsize=(8,5))
y = sigmoid(x, *popt)
plt.plot(xdata, ydata, 'ro', label='data')
plt.plot(x,y, linewidth=3.0, label='fit')
plt.legend(loc='best')
plt.ylabel('GDP')
plt.xlabel('Year')
plt.show()
```



Polynomial Regression

Sometimes, the trend of data is not really linear, and looks curvy. In this case we can use Polynomial regression methods. In fact, many different regressions exist that can be used to fit whatever the dataset looks like, such as quadratic, cubic, and so on, and it can go on and on to infinite degrees.

In essence, we can call all of these, polynomial regression, where the relationship between the independent variable x and the dependent variable y is modeled as an nth degree polynomial in x. Lets say you want to have a polynomial regression (let's make 2 degree polynomial):

$$y = b + theta_1 x + theta_2 x^2$$

Now, the question is: how we can fit our data on this equation while we have only x values, such as **Engine Size**? Well, we can create a few additional features: 1, x, and x^2.

PloynomialFeatures() function in Scikit-learn library, drives a new feature sets from the original feature set. That is, a matrix will be generated consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, lets say the original feature set has only one feature, *ENGINESIZE*. Now, if we select the degree of the polynomial to be 2, then it generates 3 features, degree=0, degree=1 and degree=2:

First, we load the data

In [99]:

```
df=pd.read_csv('C:/Users/ronni/IBM data/machine learning/FuelConsumption.csv')
```

In [100]:

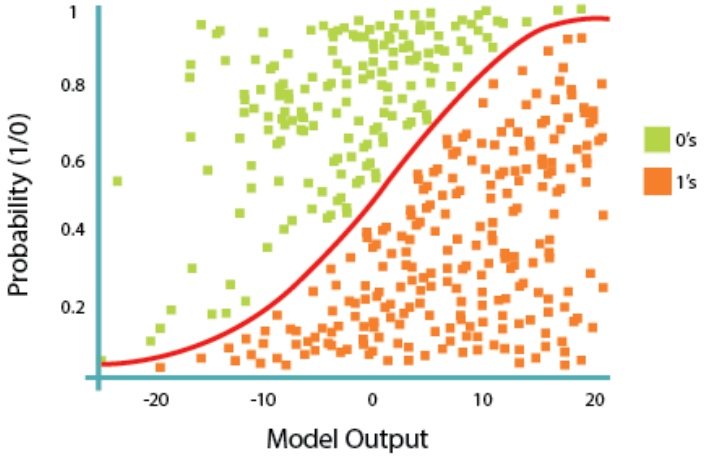
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn import linear_model
train_x = np.asanyarray(train[['ENGINESIZE']])
train_y = np.asanyarray(train[['CO2EMISSIONS']])

test_x = np.asanyarray(test[['ENGINESIZE']])
test_y = np.asanyarray(test[['CO2EMISSIONS']])


poly = PolynomialFeatures(degree=2)
train_x_poly = poly.fit_transform(train_x)
train_x_poly
```

Out[100]:

```
array([[ 1.  ,  2.  ,  4.  ],
       [ 1.  ,  2.4 ,  5.76],
       [ 1.  ,  3.5 , 12.25],
       ...,
       [ 1.  ,  3.2 , 10.24],
       [ 1.  ,  3.  ,  9.  ],
       [ 1.  ,  3.2 , 10.24]])
```

While Linear Regression is suited for estimating continuous values (e.g. estimating house price), it is not the best tool for predicting the class of an observed data point. In order to estimate the class of a data point, we

need some sort of guidance on what would be the **most probable class** for that data point. For this, we use **Logistic Regression**



Let use an example to illustrate this model.

In [7]:

```
churn_df=pd.read_csv('C:/Users/ronni/IBM data/machine learning/ChurnData.csv')
churn_df.head()
```

Out[7]:

	tenure	age	address	income	ed	employ	equip	callcard	wireless	longmon	...	pager
0	11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	1.0	4.40	...	1.0
1	33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	0.0	9.45	...	0.0
2	23.0	30.0	9.0	30.0	1.0	2.0	0.0	0.0	0.0	6.30	...	0.0
3	38.0	35.0	5.0	76.0	2.0	10.0	1.0	1.0	1.0	6.05	...	1.0
4	7.0	35.0	14.0	80.0	2.0	15.0	0.0	1.0	0.0	7.10	...	0.0

5 rows × 28 columns

Let's define X,Y for our dataset

In [8]:

```
churn_df = churn_df[['tenure', 'age', 'address', 'income', 'ed', 'employ', 'equip',   'callcard', 'w
churn_df['churn'] = churn_df['churn'].astype('int')
churn_df.head()
```

Out[8]:

	tenure	age	address	income	ed	employ	equip	callcard	wireless	churn
0	11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	1.0	1
1	33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	0.0	1
2	23.0	30.0	9.0	30.0	1.0	2.0	0.0	0.0	0.0	0
3	38.0	35.0	5.0	76.0	2.0	10.0	1.0	1.0	1.0	0
4	7.0	35.0	14.0	80.0	2.0	15.0	0.0	1.0	0.0	0

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

In [9]:

```
X = np.asarray(churn_df[['tenure', 'age', 'address', 'income', 'ed', 'employ', 'equip']])
X[0:5]
```

Out[9]:

```
array([[ 11.,  33.,   7., 136.,   5.,   5.,   0.],
       [ 33.,  33.,  12.,  33.,   2.,   0.,   0.],
       [ 23.,  30.,   9.,  30.,   1.,   2.,   0.],
       [ 38.,  35.,   5.,  76.,   2.,  10.,   1.],
       [  7.,  35.,  14.,  80.,   2.,  15.,   0.]])
```

In [10]:

```
y = np.asarray(churn_df['churn'])
y [0:5]
```

Out[10]:

```
array([1, 1, 0, 0, 0])
```

Then normalize the dataset

In [11]:

```
X = preprocessing.StandardScaler().fit(X).transform(X)
X[0:5]
```

Out[11]:

```
array([[-1.13518441, -0.62595491, -0.4588971 ,  0.4751423 ,  1.6961288 ,
        -0.58477841, -0.85972695],
       [-0.11604313, -0.62595491,  0.03454064, -0.32886061, -0.6433592 ,
        -1.14437497, -0.85972695],
       [-0.57928917, -0.85594447, -0.261522  , -0.35227817, -1.42318853,
        -0.92053635, -0.85972695],
       [ 0.11557989, -0.47262854, -0.65627219,  0.00679109, -0.6433592 ,
        -0.02518185,  1.16316   ],
       [-1.32048283, -0.47262854,  0.23191574,  0.03801451, -0.6433592 ,
         0.53441472, -0.85972695]])
```

Split the dataset into train and test sets.

In [14]:

```
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=4)
print ('Train set:', X_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (160, 7) (160,)
Test set: (40, 7) (40,)
```

Lets build our model using **LogisticRegression** from Scikit-learn package. This function implements logistic regression and can use different numerical optimizers to find parameters, including ‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’ solvers. You can find extensive information about the pros and cons of these optimizers if you search it in internet.

The version of Logistic Regression in Scikit-learn, support regularization. Regularization is a technique used to solve the overfitting problem in machine learning models. **C** parameter indicates **inverse of regularization strength** which must be a positive float. Smaller values specify stronger regularization. Now lets fit our model with train set:

In [15]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
LR = LogisticRegression(C=0.01, solver='liblinear').fit(X_train,y_train)
LR
```

Out[15]:

```
LogisticRegression(C=0.01, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [16]:

```
yhat = LR.predict(X_test)
yhat
```

Out[16]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0])
```

predict_proba returns estimates for all classes, ordered by the label of classes. So, the first column is the probability of class 1, P(Y=1|X), and second column is probability of class 0, P(Y=0|X):

In [18]:

```
yhat_prob = LR.predict_proba(X_test)
yhat_prob
```

Out[18]:

```
array([[0.54132919, 0.45867081],
       [0.60593357, 0.39406643],
       [0.56277713, 0.43722287],
       [0.63432489, 0.36567511],
       [0.56431839, 0.43568161],
       [0.55386646, 0.44613354],
       [0.52237207, 0.47762793],
       [0.60514349, 0.39485651],
       [0.41069572, 0.58930428],
       [0.6333873 , 0.3666127 ],
       [0.58068791, 0.41931209],
       [0.62768628, 0.37231372],
       [0.47559883, 0.52440117],
       [0.4267593 , 0.5732407 ],
       [0.66172417, 0.33827583],
       [0.55092315, 0.44907685],
       [0.51749946, 0.48250054],
       [0.485743  , 0.514257  ],
       [0.49011451, 0.50988549],
       [0.52423349, 0.47576651],
       [0.61619519, 0.38380481],
       [0.52696302, 0.47303698],
       [0.63957168, 0.36042832],
       [0.52205164, 0.47794836],
       [0.50572852, 0.49427148],
       [0.70706202, 0.29293798],
       [0.55266286, 0.44733714],
       [0.52271594, 0.47728406],
       [0.51638863, 0.48361137],
       [0.71331391, 0.28668609],
       [0.67862111, 0.32137889],
       [0.50896403, 0.49103597],
       [0.42348082, 0.57651918],
       [0.71495838, 0.28504162],
       [0.59711064, 0.40288936],
       [0.63808839, 0.36191161],
       [0.39957895, 0.60042105],
       [0.52127638, 0.47872362],
       [0.65975464, 0.34024536],
       [0.5114172 , 0.4885828 ]])
```

As for the model evaluation, Let's first look at jaccard index. If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0; otherwise it is 0.0.

In [19]:

```
from sklearn.metrics import jaccard_similarity_score
jaccard_similarity_score(y_test, yhat)
```

```
C:\Users\ronni\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:664: F
utureWarning: jaccard_similarity_score has been deprecated and replaced with jaccard
_score. It will be removed in version 0.23. This implementation has surprising behav
ior for binary and multiclass classification tasks.
  FutureWarning)
```

Out[19]:

```
0.75
```

Another way of looking at accuracy of classifier is to look at **confusion matrix**.

In [22]:

```
from sklearn.metrics import classification_report, confusion_matrix
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

In [23]:

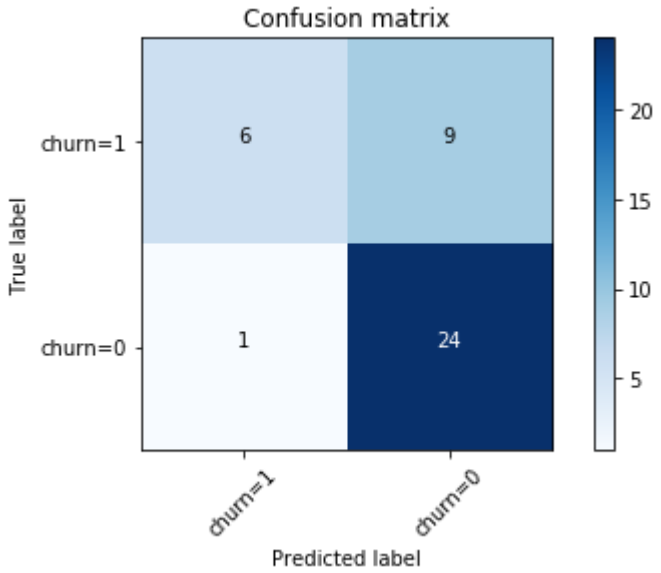
```
print(confusion_matrix(y_test, yhat, labels=[1,0]))
```

```
[[ 6  9]
 [ 1 24]]
```

In [21]:

```
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, yhat, labels=[1,0])
np.set_printoptions(precision=2)


# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=['churn=1','churn=0'],normalize= False,  title='Confusion
```

```
Confusion matrix, without normalization
[[ 6  9]
 [ 1 24]]
```



Look at first row. The first row is for customers whose actual churn value in test set is 1. As you can calculate, out of 40 customers, the churn value of 15 of them is 1. And out of these 15, the classifier correctly predicted 6 of them as 1, and 9 of them as 0.

It means, for 6 customers, the actual churn value were 1 in test set, and classifier also correctly predicted those as 1. However, while the actual label of 9 customers were 1, the classifier predicted those as 0, which is not very good. We can consider it as error of the model for first row.

What about the customers with churn value 0? Lets look at the second row. It looks like there were 25 customers whom their churn value were 0.

The classifier correctly predicted 24 of them as 0, and one of them wrongly as 1. So, it has done a good job in predicting the customers with churn value 0. A good thing about confusion matrix is that shows the model's ability to correctly predict or separate the classes. In specific case of binary classifier, such as this example, we can interpret these numbers as the count of true positives, false positives, true negatives, and false negatives.

In [24]:

```
print (classification_report(y_test, yhat))
```

```
              precision    recall  f1-score   support

           0       0.73      0.96      0.83        25
           1       0.86      0.40      0.55        15

    accuracy                           0.75        40
   macro avg       0.79      0.68      0.69        40
weighted avg       0.78      0.75      0.72        40
```

Based on the count of each section, we can calculate precision and recall of each label:

- **Precision** is a measure of the accuracy provided that a class label has been predicted. It is defined by: precision = TP / (TP + FP)
- **Recall** is true positive rate. It is defined as: Recall = TP / (TP + FN)

So, we can calculate precision and recall of each class.

F1 score: Now we are in the position to calculate the F1 scores for each label based on the precision and recall of that label.

The F1 score is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. It is a good way to show that a classifer has a good value for both recall and precision.

And finally, we can tell the average accuracy for this classifier is the average of the F1-score for both labels, which is 0.72 in our case.

Now, lets try **log loss** for evaluation. In logistic regression, the output can be the probability of customer churn is yes (or equals to 1). This probability is a value between 0 and 1. Log loss(Logarithmic loss) measures the performance of a classifier where the predicted output is a probability value between 0 and 1.

In [25]:

```
from sklearn.metrics import log_loss
log_loss(y_test, yhat_prob)
```

Out[25]:

```
0.6017092478101187
```

K-nearest neighbors

K-Nearest Neighbors is an algorithm for supervised learning. Where the data is 'trained' with data points corresponding to their classification. Once a point is to be predicted, it takes into account the 'K' nearest points to it to determine it's classification.

In [26]:

```
import itertools
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import NullFormatter
import pandas as pd
import numpy as np
import matplotlib.ticker as ticker
from sklearn import preprocessing
%matplotlib inline
```

Let's load the dataset first and have some basic insights of the datasets.

In [33]:

```
df=pd.read_csv('C:/Users/ronni/IBM data/machine learning/teleCust1000t.csv')
df.head()
```

Out[33]:

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

In [34]:

```
df['custcat'].value_counts()
```

Out[34]:

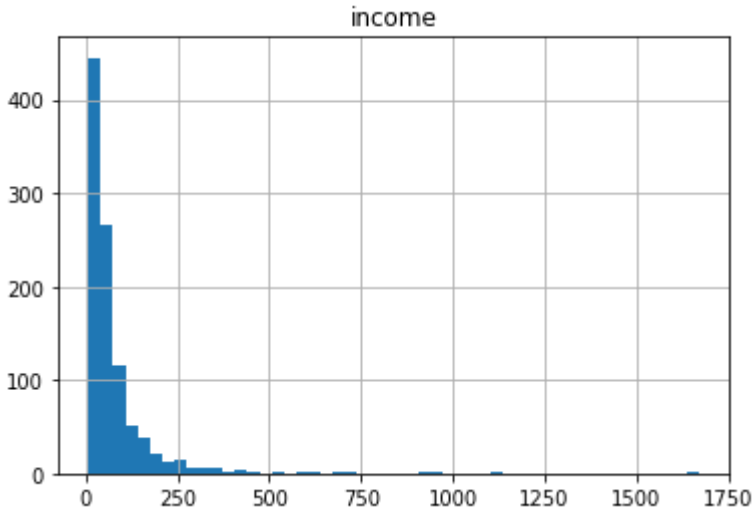
```
3    281
1    266
4    236
2    217
Name: custcat, dtype: int64
```

In [35]:

```
df.hist(column='income', bins=50)
```

Out[35]:

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x0000020434D5A548>]],
      dtype=object)
```



In [36]:

```
X = df[['region', 'tenure','age', 'marital', 'address', 'income', 'ed', 'employ','retire', 'gender',
X[0:5]
```

Out[36]:

```
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.]])
```

In [37]:

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

Out[37]:

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

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

In [38]:

```
X = preprocessing.StandardScaler().fit(X).transform(X.astype(float))
X[0:5]
```

Out[38]:

```
array([[-0.03, -1.06,  0.18,  1.01, -0.25, -0.13,  1.09, -0.59, -0.22,
        -1.03, -0.23],
       [ 1.2 , -1.15, -0.69,  1.01, -0.45,  0.55,  1.91, -0.59, -0.22,
        -1.03,  2.56],
       [ 1.2 ,  1.52,  0.82,  1.01,  1.23,  0.36, -1.37,  1.79, -0.22,
         0.97, -0.23],
       [-0.03, -0.12, -0.69, -0.99,  0.04, -0.42, -0.55, -1.09, -0.22,
         0.97, -0.93],
       [-0.03, -0.59, -0.93,  1.01, -0.25, -0.44, -1.37, -0.89, -0.22,
        -1.03,  1.16]])
```

Then set up train and test sets.

In [39]:

```
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_state=4)
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,)
```

Let's start to construct the model with k=4

In [41]:

```
from sklearn.neighbors import KNeighborsClassifier
k = 4
#Train Model and Predict
neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
neigh
```

Out[41]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=4, p=2,
                     weights='uniform')
```

In [42]:

```
yhat = neigh.predict(X_test)
yhat[0:5]
```

Out[42]:

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

In [44]:

```
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:  0.32
```

We can evaluate the best k by using codes below.

In [46]:

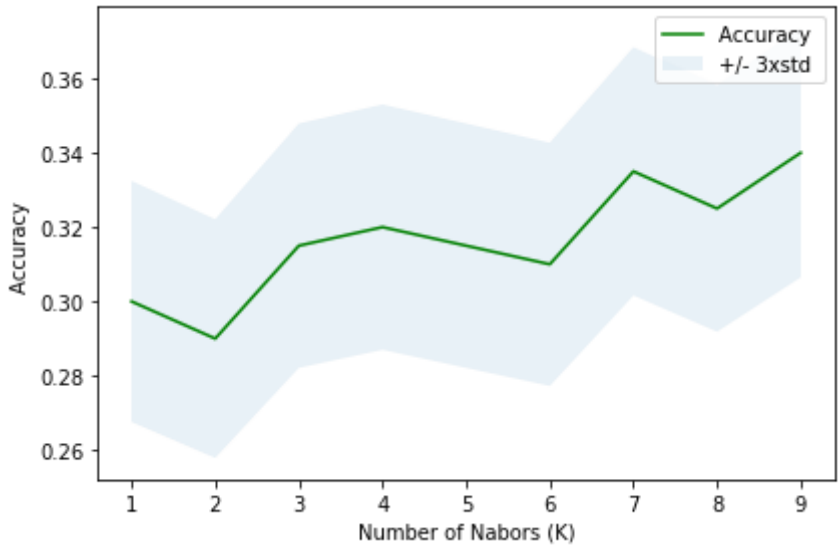
```
Ks = 10
mean_acc = np.zeros((Ks-1))
std_acc = np.zeros((Ks-1))
ConfustionMx = [];
for n in range(1,Ks):

    #Train Model and Predict
    neigh = KNeighborsClassifier(n_neighbors = n).fit(X_train,y_train)
    yhat=neigh.predict(X_test)
    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[46]:

```
array([0.3 , 0.29, 0.32, 0.32, 0.32, 0.31, 0.34, 0.33, 0.34])
```

In [47]:

```
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=0.10)
plt.legend(('Accuracy ', '+/- 3xstd'))
plt.ylabel('Accuracy ')
plt.xlabel('Number of Nabors (K)')
plt.tight_layout()
plt.show()
```



In [48]:

```
print( "The best accuracy was with", mean_acc.max(), "with k=", mean_acc.argmax()+1)
```

```
The best accuracy was with 0.34 with k= 9
```

Decision Tree

First load the data and do some wrangling.

In [3]:

```
my_data=pd.read_csv('C:/Users/ronni/IBM data/machine learning/drug200.csv')
my_data.head()
```

Out[3]:

	Age	Sex	BP	Cholesterol	Na_to_K	Drug
0	23	F	HIGH	HIGH	25.355	drugY
1	47	M	LOW	HIGH	13.093	drugC
2	47	M	LOW	HIGH	10.114	drugC
3	28	F	NORMAL	HIGH	7.798	drugX
4	61	F	LOW	HIGH	18.043	drugY

In [4]:

```
X = my_data[['Age', 'Sex', 'BP', 'Cholesterol', 'Na_to_K']].values
X[0:5]
```

Out[4]:

```
array([[23, 'F', 'HIGH', 'HIGH', 25.355],
       [47, 'M', 'LOW', 'HIGH', 13.093],
       [47, 'M', 'LOW', 'HIGH', 10.113999999999999],
       [28, 'F', 'NORMAL', 'HIGH', 7.797999999999999],
       [61, 'F', 'LOW', 'HIGH', 18.043]], dtype=object)
```

As you may figure out, some features in this dataset are categorical such as **Sex** or **BP**. Unfortunately, Sklearn Decision Trees do not handle categorical variables. But still we can convert these features to numerical values. **pandas.get_dummies()** Convert categorical variable into dummy/indicator variables.

In [5]:

```
from sklearn import preprocessing
le_sex = preprocessing.LabelEncoder()
le_sex.fit(['F','M'])
X[:,1] = le_sex.transform(X[:,1])


le_BP = preprocessing.LabelEncoder()
le_BP.fit([ 'LOW', 'NORMAL', 'HIGH'])
X[:,2] = le_BP.transform(X[:,2])


le_Chol = preprocessing.LabelEncoder()
le_Chol.fit([ 'NORMAL', 'HIGH'])
X[:,3] = le_Chol.transform(X[:,3])

X[0:5]
```

Out[5]:

```
array([[23, 0, 0, 0, 25.355],
       [47, 1, 1, 0, 13.093],
       [47, 1, 1, 0, 10.113999999999999],
       [28, 0, 2, 0, 7.797999999999999],
       [61, 0, 1, 0, 18.043]], dtype=object)
```

In [6]:

```
y = my_data["Drug"]
y[0:5]
```

Out[6]:

```
0    drugY
1    drugC
2    drugC
3    drugX
4    drugY
Name: Drug, dtype: object
```

Now setting up the decision tree.

In [7]:

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
```

Now **train_test_split** will return 4 different parameters. We will name them:
X_trainset, X_testset, y_trainset, y_testset

The **train_test_split** will need the parameters:
X, y, test_size=0.3, and random_state=3.

The **X** and **y** are the arrays required before the split, the **test_size** represents the ratio of the testing dataset, and the **random_state** ensures that we obtain the same splits.

In [8]:

```
X_trainset, X_testset, y_trainset, y_testset = train_test_split(X, y, test_size=0.3, random_state=3)
```

We will first create an instance of the **DecisionTreeClassifier** called **drugTree**.
Inside of the classifier, specify *criterion="entropy"* so we can see the information gain of each node.

In [9]:

```
drugTree = DecisionTreeClassifier(criterion="entropy", max_depth = 4)
drugTree # it shows the default parameters
```

Out[9]:

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='entropy',
                       max_depth=4, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

In [10]:

```
drugTree.fit(X_trainset,y_trainset)
```

Out[10]:

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='entropy',
                       max_depth=4, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

Let's make some **predictions** on the testing dataset and store it into a variable called **predTree**.

In [11]:

```
predTree = drugTree.predict(X_testset)
```

You can print out **predTree** and **y_testset** if you want to visually compare the prediction to the actual values.

In [12]:

```
print (predTree [0:5])
print (y_testset [0:5])
```

```
['drugY' 'drugX' 'drugX' 'drugX' 'drugX']
40     drugY
51     drugX
139    drugX
197    drugX
170    drugX
Name: Drug, dtype: object
```

Then do the evaluation

In [13]:

```
from sklearn import metrics
import matplotlib.pyplot as plt
print("DecisionTrees's Accuracy: ", metrics.accuracy_score(y_testset, predTree))
```

```
DecisionTrees's Accuracy:  0.9833333333333333
```

Visualize it.

In [14]:

```
from sklearn.externals.six import StringIO
import pydotplus
import matplotlib.image as mpimg
from sklearn import tree
%matplotlib inline
```

In [15]:

```
dot_data = StringIO()
filename = "drugtree.png"
featureNames = my_data.columns[0:5]
targetNames = my_data["Drug"].unique().tolist()
out=tree.export_graphviz(drugTree,feature_names=featureNames, out_file=dot_data, class_names= np.un
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png(filename)
img = mpimg.imread(filename)
plt.figure(figsize=(100, 200))
plt.imshow(img,interpolation='nearest')
```

```
---------------------------------------------------------------------------
InvocationException                       Traceback (most recent call last)
<ipython-input-15-8f9eff648245> in <module>
      5 out=tree.export_graphviz(drugTree,feature_names=featureNames, out_file=dot_
data, class_names= np.unique(y_trainset), filled=True,  special_characters=True,ro
tate=False)
      6 graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
----> 7 graph.write_png(filename)
      8 img = mpimg.imread(filename)
      9 plt.figure(figsize=(100, 200))

~\anaconda3\lib\site-packages\pydotplus\graphviz.py in <lambda>(path, f, pr
og)
   1808                 lambda path,
   1809                 f=frmt,
-> 1810                     prog=self.prog: self.write(path, format=f, prog=prog)
   1811             )
   1812

~\anaconda3\lib\site-packages\pydotplus\graphviz.py in write(self, path, pr
og, format)
   1916
   1917             else:
-> 1918                 fobj.write(self.create(prog, format))
   1919         finally:
   1920             if close:

~\anaconda3\lib\site-packages\pydotplus\graphviz.py in create(self, prog, f
ormat)
   1958             if self.progs is None:
   1959                 raise InvocationException(
-> 1960                     'GraphViz\'s executables not found')
   1961
   1962         if prog not in self.progs:

InvocationException: GraphViz's executables not found
```

There are problems happened in using GraphViz, detailed information could be found in
https://blog.csdn.net/weixin_36407399/article/details/87890230
(https://blog.csdn.net/weixin_36407399/article/details/87890230)

Support Vector Machines

SVM works by mapping data to a high-dimensional feature space so that data points can be categorized, even when the data are not otherwise linearly separable. A separator between the categories is found, then the data is transformed in such a way that the separator could be drawn as a hyperplane. Following this, characteristics of new data can be used to predict the group to which a new record should belong.

Let's load and have an insight of the dataset.

In [17]:

```
cell_df=pd.read_csv('C:/Users/ronni/IBM data/machine learning/cell_samples.csv')
cell_df.head()
```

Out[17]:

	ID	Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormN
0	1000025	5	1	1	1	2	1	3	
1	1002945	5	4	4	5	7	10	3	
2	1015425	3	1	1	1	2	2	3	
3	1016277	6	8	8	1	3	4	3	
4	1017023	4	1	1	3	2	1	3	

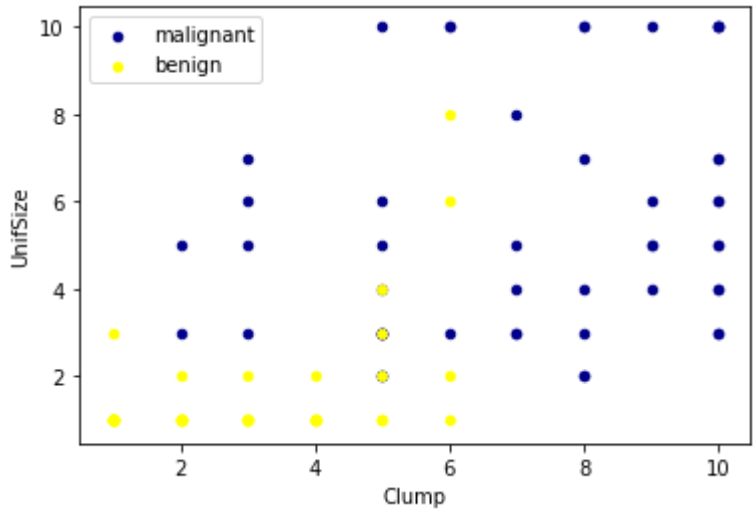
The ID field contains the patient identifiers. The characteristics of the cell samples from each patient are contained in fields Clump to Mit. The values are graded from 1 to 10, with 1 being the closest to benign.

The Class field contains the diagnosis, as confirmed by separate medical procedures, as to whether the samples are benign (value = 2) or malignant (value = 4).

Lets look at the distribution of the classes based on Clump thickness and Uniformity of cell size:

In [18]:

```
ax = cell_df[cell_df['Class'] == 4][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='Dark
cell_df[cell_df['Class'] == 2][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='Yellow',
plt.show()
```



Data pre-processing

In [19]:

```
cell_df.dtypes
```

Out[19]:

```
ID              int64
Clump           int64
UnifSize        int64
UnifShape       int64
MargAdh         int64
SingEpiSize     int64
BareNuc        object
BlandChrom      int64
NormNucl        int64
Mit             int64
Class           int64
dtype: object
```

It looks like the **BareNuc** column includes some values that are not numerical. We can drop those rows:

In [20]:

```
cell_df = cell_df[pd.to_numeric(cell_df['BareNuc'], errors='coerce').notnull()]
cell_df['BareNuc'] = cell_df['BareNuc'].astype('int')
cell_df.dtypes
```

Out[20]:

```
ID              int64
Clump           int64
UnifSize        int64
UnifShape       int64
MargAdh         int64
SingEpiSize     int64
BareNuc         int32
BlandChrom      int64
NormNucl        int64
Mit             int64
Class           int64
dtype: object
```

In [21]:

```
feature_df = cell_df[['Clump', 'UnifSize', 'UnifShape', 'MargAdh', 'SingEpiSize', 'BareNuc', 'BlandC
X = np.asarray(feature_df)
X[0:5]
```

Out[21]:

```
array([[ 5,  1,  1,  1,  2,  1,  3,  1,  1],
       [ 5,  4,  4,  5,  7, 10,  3,  2,  1],
       [ 3,  1,  1,  1,  2,  2,  3,  1,  1],
       [ 6,  8,  8,  1,  3,  4,  3,  7,  1],
       [ 4,  1,  1,  3,  2,  1,  3,  1,  1]], dtype=int64)
```

In [22]:

```
cell_df['Class'] = cell_df['Class'].astype('int')
y = np.asarray(cell_df['Class'])
y [0:5]
```

Out[22]:

```
array([2, 2, 2, 2, 2])
```

Then set up train and test sets.

In [23]:

```
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=4)
print ('Train set:', X_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (546, 9) (546,)
Test set: (137, 9) (137,)
```

The SVM algorithm offers a choice of kernel functions for performing its processing. Basically, mapping data into a higher dimensional space is called kernelling. The mathematical function used for the transformation is known as the kernel function, and can be of different types, such as:

1. Linear
2. Polynomial
3. Radial basis function (RBF)
4. Sigmoid

Each of these functions has its characteristics, its pros and cons, and its equation, but as there's no easy way of knowing which function performs best with any given dataset, we usually choose different functions in turn and compare the results. Let's just use the default, RBF (Radial Basis Function) for this lab.

In [24]:

```
from sklearn import svm
clf = svm.SVC(kernel='rbf')
clf.fit(X_train, y_train)
```

Out[24]:

```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

After being fitted, the model can then be used to predict new values:

In [25]:

```
yhat = clf.predict(X_test)
yhat [0:5]
```

Out[25]:

```
array([2, 4, 2, 4, 2])
```

Then evaluate the model.

In [30]:

```
from sklearn.metrics import classification_report, confusion_matrix
import itertools
```

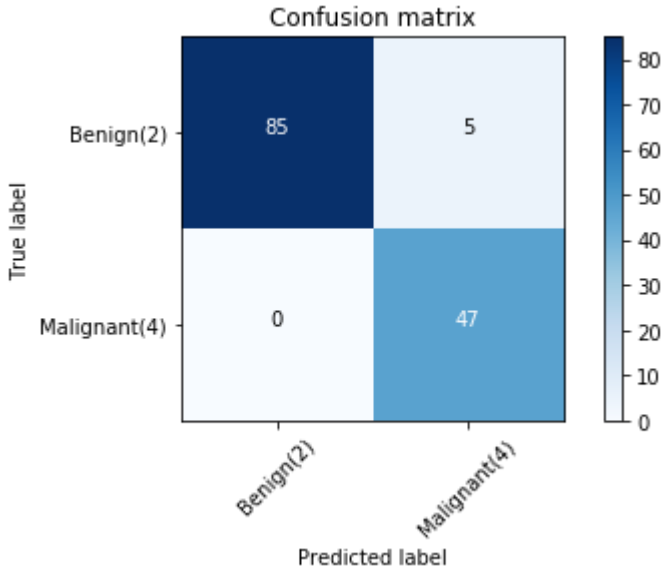
In [31]:

```
#define and plot confustion matrix
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

In [32]:

```
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, yhat, labels=[2,4])
np.set_printoptions(precision=2)

print (classification_report(y_test, yhat))

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=['Benign(2)','Malignant(4)'],normalize= False,  title='Co
```

```
              precision    recall  f1-score   support

           2       1.00      0.94      0.97        90
           4       0.90      1.00      0.95        47

    accuracy                           0.96       137
   macro avg       0.95      0.97      0.96       137
weighted avg       0.97      0.96      0.96       137

Confusion matrix, without normalization
[[85  5]
 [ 0 47]]
```



You can also easily use the **f1_score** from sklearn library:

In [33]:

```
from sklearn.metrics import f1_score
f1_score(y_test, yhat, average='weighted')
```

Out[33]:

0.9639038982104676

Lets try jaccard index for accuracy

In [34]:

```
from sklearn.metrics import jaccard_similarity_score
jaccard_similarity_score(y_test, yhat)
```

```
C:\Users\ronni\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:664: F
utureWarning: jaccard_similarity_score has been deprecated and replaced with jaccard
_score. It will be removed in version 0.23. This implementation has surprising behav
ior for binary and multiclass classification tasks.
  FutureWarning)
```

Out[34]:

0.9635036496350365

K-means clustering

There are many models for **clustering** out there. In this notebook, we will be presenting the model that is considered one of the simplest models amongst them. Despite its simplicity, the **K-means** is vastly used for clustering in many data science applications, especially useful if you need to quickly discover insights from **unlabeled data**. In this notebook, you will learn how to use k-Means for customer segmentation.

Some real-world applications of k-means:

- Customer segmentation
- Understand what the visitors of a website are trying to accomplish
- Pattern recognition
- Machine learning
- Data compression

In this notebook we practice k-means clustering with 2 examples:

- k-means on a random generated dataset
- Using k-means for customer segmentation

First we need to set up a random seed. Use **numpy's random.seed()** function, where the seed will be set to **0**

In [37]:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets.samples_generator import make_blobs
%matplotlib inline
```

```
C:\Users\ronni\anaconda3\lib\site-packages\sklearn\utils\deprecation.py:144: FutureW
arning: The sklearn.datasets.samples_generator module is  deprecated in version 0.22
and will be removed in version 0.24. The corresponding classes / functions should in
stead be imported from sklearn.datasets. Anything that cannot be imported from sklea
rn.datasets is now part of the private API.
  warnings.warn(message, FutureWarning)
```

In [35]:

```
np.random.seed(0)
```

Next we will be making *random clusters* of points by using the **make_blobs** class. The **make_blobs** class can take in many inputs, but we will be using these specific ones.

Input

- **n_samples**: The total number of points equally divided among clusters.
  - Value will be: 5000
- **centers**: The number of centers to generate, or the fixed center locations.
  - Value will be: [[4, 4], [-2, -1], [2, -3],[1,1]]
- **cluster_std**: The standard deviation of the clusters.
  - Value will be: 0.9

Output

- **X**: Array of shape [n_samples, n_features]. (Feature Matrix)
  - The generated samples.
- **y**: Array of shape [n_samples]. (Response Vector)
  - The integer labels for cluster membership of each sample.

In [38]:

```
X, y = make_blobs(n_samples=5000, centers=[[4,4], [-2, -1], [2, -3], [1, 1]], cluster_std=0.9)
```

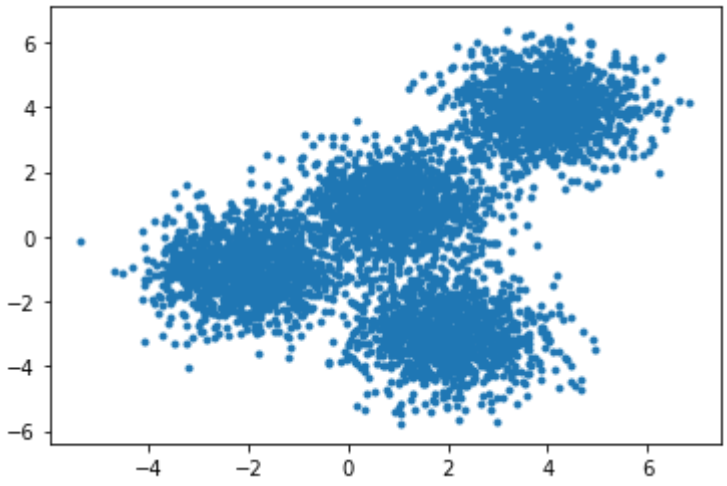
Display the scatter plot of the randomly generated data.

In [39]:

```
plt.scatter(X[:, 0], X[:, 1], marker='.')
```

Out[39]:

```
<matplotlib.collections.PathCollection at 0x1d970a32508>
```



The KMeans class has many parameters that can be used, but we will be using these three:

- **init**: Initialization method of the centroids.
  - Value will be: "k-means++"
  - k-means++: Selects initial cluster centers for k-mean clustering in a smart way to speed up convergence.
- **n_clusters**: The number of clusters to form as well as the number of centroids to generate.
  - Value will be: 4 (since we have 4 centers)
- **n_init**: Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
  - Value will be: 12

Initialize KMeans with these parameters, where the output parameter is called **k_means**.

In [40]:

```
k_means = KMeans(init = "k-means++", n_clusters = 4, n_init = 12)
```

Now let's fit the KMeans model with the feature matrix we created above, **X**

In [41]:

```
k_means.fit(X)
```

Out[41]:

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=4, n_init=12, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
```

Now let's grab the labels for each point in the model using KMeans' **.labels_** attribute and save it as **k_means_labels**

In [42]:

```
k_means_labels = k_means.labels_
k_means_labels
```

Out[42]:

```
array([0, 2, 2, ..., 3, 0, 0])
```

We will also get the coordinates of the cluster centers using KMeans' **.cluster_centers_** and save it as **k_means_cluster_centers**

In [44]:

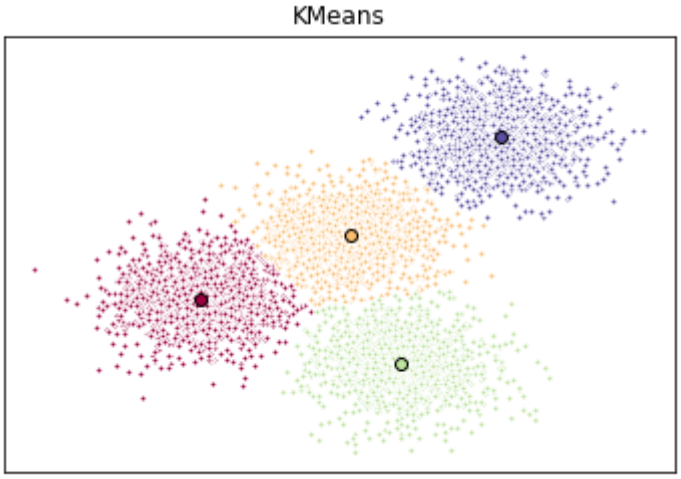
```
k_means_cluster_centers = k_means.cluster_centers_
k_means_cluster_centers
```

Out[44]:

```
array([[-2.04, -1.  ],
       [ 0.97,  0.98],
       [ 2.  , -3.02],
       [ 3.97,  3.99]])
```

In [45]:

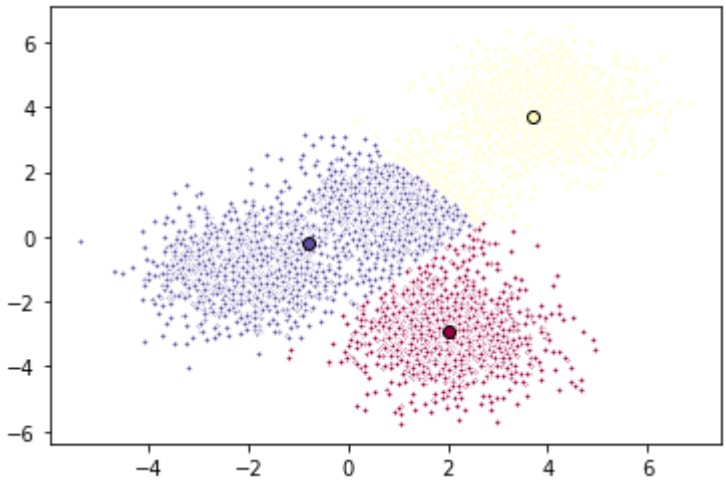
```
# Initialize the plot with the specified dimensions.
fig = plt.figure(figsize=(6, 4))

# Colors uses a color map, which will produce an array of colors based on
# the number of labels there are. We use set(k_means_labels) to get the
# unique labels.
colors = plt.cm.Spectral(np.linspace(0, 1, len(set(k_means_labels))))

# Create a plot
ax = fig.add_subplot(1, 1, 1)

# For loop that plots the data points and centroids.
# k will range from 0-3, which will match the possible clusters that each
# data point is in.
for k, col in zip(range(len([[4,4], [-2, -1], [2, -3], [1, 1]])), colors):

    # Create a list of all data points, where the data poitns that are
    # in the cluster (ex. cluster 0) are labeled as true, else they are
    # labeled as false.
    my_members = (k_means_labels == k)

    # Define the centroid, or cluster center.
    cluster_center = k_means_cluster_centers[k]

    # Plots the datapoints with color col.
    ax.plot(X[my_members, 0], X[my_members, 1], 'w', markerfacecolor=col, marker='.')

    # Plots the centroids with specified color, but with a darker outline
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,  markeredgecolor='k', ma

# Title of the plot
ax.set_title('KMeans')

# Remove x-axis ticks
ax.set_xticks(())

# Remove y-axis ticks
ax.set_yticks(())

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



Try to cluster the above dataset into 3 clusters.

In [46]:

```
k_means3 = KMeans(init = "k-means++", n_clusters = 3, n_init = 12)
k_means3.fit(X)
fig = plt.figure(figsize=(6, 4))
colors = plt.cm.Spectral(np.linspace(0, 1, len(set(k_means3.labels_))))
ax = fig.add_subplot(1, 1, 1)
for k, col in zip(range(len(k_means3.cluster_centers_)), colors):
    my_members = (k_means3.labels_ == k)
    cluster_center = k_means3.cluster_centers_[k]
    ax.plot(X[my_members, 0], X[my_members, 1], 'w', markerfacecolor=col, marker='.')
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,  markeredgecolor='k', ma
plt.show()
```



Imagine that you have a customer dataset, and you need to apply customer segmentation on this historical data. Customer segmentation is the practice of partitioning a customer base into groups of individuals that have similar characteristics. It is a significant strategy as a business can target these specific groups of customers and effectively allocate marketing resources. For example, one group might contain customers who are high-profit and low-risk, that is, more likely to purchase products, or subscribe for a service. A business task is to retaining those customers. Another group might include customers from non-profit organizations. And so on.

In [50]:

```
cust_df=pd.read_csv('c:/users/ronni/IBM data/machine learning/Cust_Segmentation.csv')
cust_df.head()
```

Out[50]:

	Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Defaulted	Address	DebtIncomeRati
0	1	41	2	6	19	0.124	1.073	0.0	NBA001	6.
1	2	47	1	26	100	4.582	8.218	0.0	NBA021	12.
2	3	33	2	10	57	6.111	5.802	1.0	NBA013	20.
3	4	29	2	4	19	0.681	0.516	0.0	NBA009	6.
4	5	47	1	31	253	9.308	8.908	0.0	NBA008	7.

Pre-processing the data

As you can see, **Address** in this dataset is a categorical variable. k-means algorithm isn't directly applicable to categorical variables because Euclidean distance function isn't really meaningful for discrete variables. So, lets drop this feature and run clustering.

In [51]:

```
df = cust_df.drop('Address', axis=1)
df.head()
```

Out[51]:

	Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Defaulted	DebtIncomeRatio
0	1	41	2	6	19	0.124	1.073	0.0	6.3
1	2	47	1	26	100	4.582	8.218	0.0	12.8
2	3	33	2	10	57	6.111	5.802	1.0	20.9
3	4	29	2	4	19	0.681	0.516	0.0	6.3
4	5	47	1	31	253	9.308	8.908	0.0	7.2

Now let's normalize the dataset. But why do we need normalization in the first place? Normalization is a statistical method that helps mathematical-based algorithms to interpret features with different magnitudes and distributions equally. We use **StandardScaler()** to normalize our dataset.

In [52]:

```
from sklearn.preprocessing import StandardScaler
X = df.values[:,1:]
X = np.nan_to_num(X)
Clus_dataSet = StandardScaler().fit_transform(X)
Clus_dataSet
```

Out[52]:

```
array([[ 0.74,  0.31, -0.38, ..., -0.59, -0.52, -0.58],
       [ 1.49, -0.77,  2.57, ...,  1.51, -0.52,  0.39],
       [-0.25,  0.31,  0.21, ...,  0.8 ,  1.91,  1.6 ],
       ...,
       [-1.25,  2.47, -1.26, ...,  0.04,  1.91,  3.46],
       [-0.38, -0.77,  0.51, ..., -0.7 , -0.52, -1.08],
       [ 2.11, -0.77,  1.1 , ...,  0.16, -0.52, -0.23]])
```

In our example (if we didn't have access to the k-means algorithm), it would be the same as guessing that each customer group would have certain age, income, education, etc, with multiple tests and experiments. However, using the K-means clustering we can do all this process much easier.

Lets apply k-means on our dataset, and take look at cluster labels.

In [53]:

```
clusterNum = 3
k_means = KMeans(init = "k-means++", n_clusters = clusterNum, n_init = 12)
k_means.fit(X)
labels = k_means.labels_
print(labels)
```

```
[1 0 1 1 2 0 1 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 1 1
 1 1 0 1 0 1 2 1 0 1 1 1 0 0 1 1 0 0 1 1 1 0 1 0 1 0 0 1 1 0 1 1 1 0 0 0 1
 1 1 1 1 0 1 0 0 2 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1
 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 0 1
 1 1 1 1 1 1 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1
 1 1 1 1 0 1 1 0 1 0 1 1 0 2 1 0 1 1 1 1 1 1 2 0 1 1 1 1 0 1 1 0 0 1 0 1 0
 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 2 0 1 1 1 1 1 1 1 0 1 1 1 1
 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 1 0 1 0 0 1 1 1 1 1 1
 1 1 1 0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 0 1 0 0 1
 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1 1 2 1 1 1 0 1 0 0 0 1
 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
 1 0 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 2
 1 1 1 1 1 1 0 1 1 1 2 1 1 1 1 1 0 1 2 1 1 1 1 0 1 0 0 0 1 1 0 0 1 1 1 1 1 1
 1 0 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 1 0 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1
 1 0 0 1 1 1 1 1 1 1 1 1 1 1 2 0 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 2 1 2 1
 1 2 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 2 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 0
 1 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0
 0 1 1 0 1 0 1 1 0 1 0 1 1 2 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 0 1 1 1 0 0 1 1
 0 1 1 1 0 1 2 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1
 1 1 0 1 1 0 1 0 1 0 0 1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1
 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 0 1 0 0 1 1 0 1 1 1 1 1 0 0
 1 1 1 1 1 1 1 0 1 1 1 1 1 1 2 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0]
```

We assign the labels to each row in dataframe.

In [55]:

```
df["Clus_km"] = labels
df.head(5)
```

Out[55]:

	Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Defaulted	DebtIncomeRatio	Clus_k
0	1	41	2	6	19	0.124	1.073	0.0	6.3	
1	2	47	1	26	100	4.582	8.218	0.0	12.8	
2	3	33	2	10	57	6.111	5.802	1.0	20.9	
3	4	29	2	4	19	0.681	0.516	0.0	6.3	
4	5	47	1	31	253	9.308	8.908	0.0	7.2	

We can easily check the centroid values by averaging the features in each cluster.

In [56]:

```
df.groupby('Clus_km').mean()
```

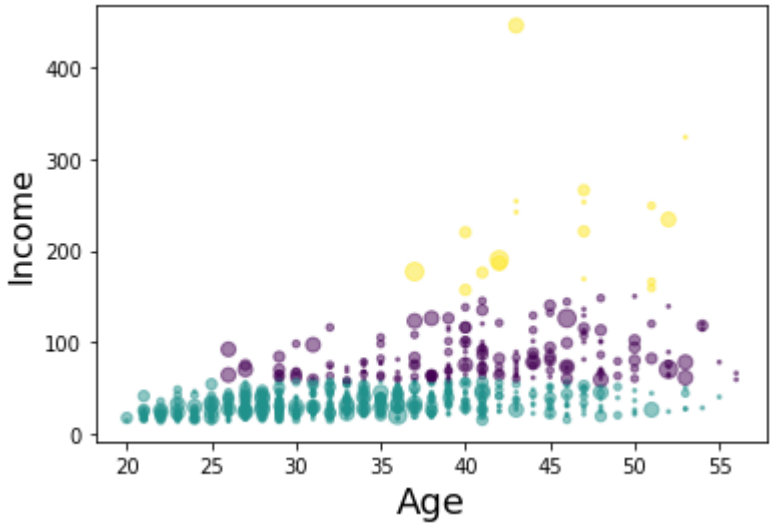
Out[56]:

Clus_km	Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Default
0	403.780220	41.368132	1.961538	15.252747	84.076923	3.114412	5.770352	0.1724
1	432.006154	32.967692	1.613846	6.389231	31.204615	1.032711	2.108345	0.2846
2	410.166667	45.388889	2.666667	19.555556	227.166667	5.678444	10.907167	0.2857

Now, lets look at the distribution of customers based on their age and income:

In [57]:

```
area = np.pi * ( X[:, 1])**2
plt.scatter(X[:, 0], X[:, 3], s=area, c=labels.astype(np.float), alpha=0.5)
plt.xlabel('Age', fontsize=18)
plt.ylabel('Income', fontsize=16)

plt.show()
```

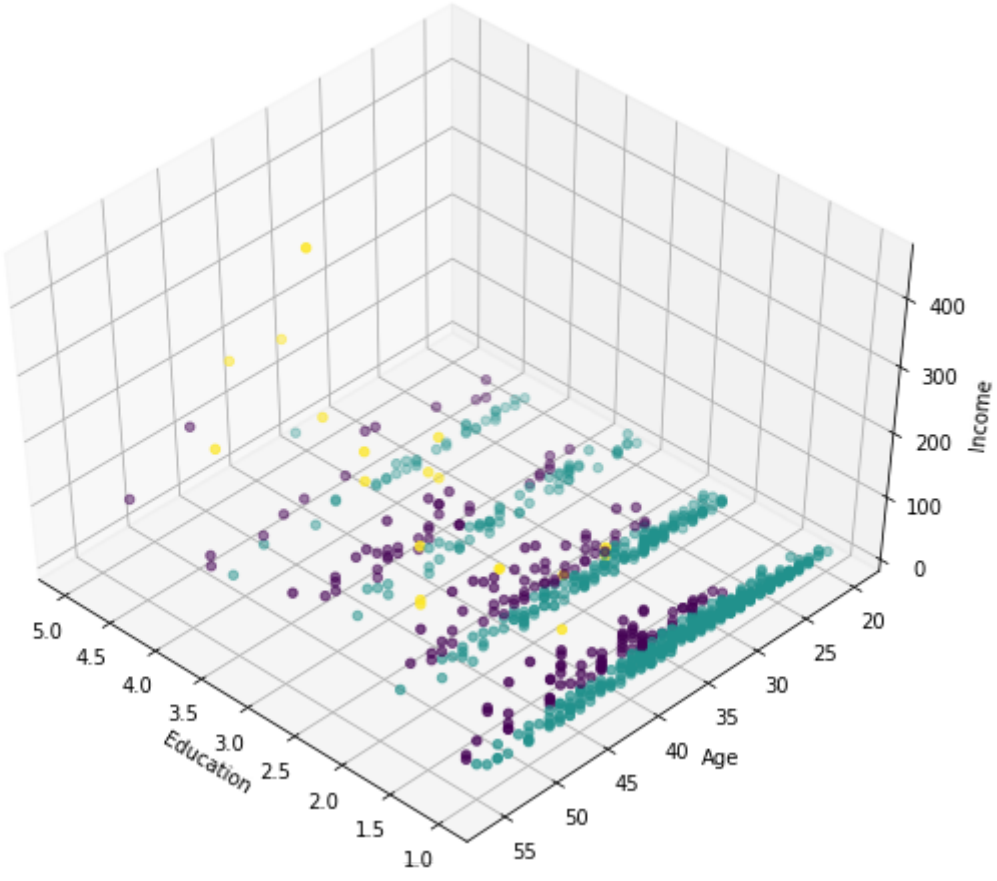


In [58]:

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(1, figsize=(8, 6))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

plt.cla()
# plt.ylabel('Age', fontsize=18)
# plt.xlabel('Income', fontsize=16)
# plt.zlabel('Education', fontsize=16)
ax.set_xlabel('Education')
ax.set_ylabel('Age')
ax.set_zlabel('Income')

ax.scatter(X[:, 1], X[:, 0], X[:, 3], c= labels.astype(np.float))
```

Out[58]:

<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1d97213cb48>



k-means will partition your customers into mutually exclusive groups, for example, into 3 clusters. The customers in each cluster are similar to each other demographically. Now we can create a profile for each group, considering the common characteristics of each cluster. For example, the 3 clusters can be:

- AFFLUENT, EDUCATED AND OLD AGED
- MIDDLE AGED AND MIDDLE INCOME
- YOUNG AND LOW INCOME

Hierarchical Clustering - Agglomerative

In [1]:

```
import numpy as np
import pandas as pd
from scipy import ndimage
from scipy.cluster import hierarchy
from scipy.spatial import distance_matrix
from matplotlib import pyplot as plt
from sklearn import manifold, datasets
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets.samples_generator import make_blobs
%matplotlib inline
```

```
C:\Users\ronni\anaconda3\lib\site-packages\sklearn\utils\deprecation.py:144: FutureW
arning: The sklearn.datasets.samples_generator module is  deprecated in version 0.22
and will be removed in version 0.24. The corresponding classes / functions should in
stead be imported from sklearn.datasets. Anything that cannot be imported from sklea
rn.datasets is now part of the private API.
  warnings.warn(message, FutureWarning)
```

We will be generating a set of data using the **make_blobs** class.

Input these parameters into make_blobs:

- **n_samples**: The total number of points equally divided among clusters.
  - Choose a number from 10-1500
- **centers**: The number of centers to generate, or the fixed center locations.
  - Choose arrays of x,y coordinates for generating the centers. Have 1-10 centers (ex. centers=[[1,1], [2,5]])
- **cluster_std**: The standard deviation of the clusters. The larger the number, the further apart the clusters
  - Choose a number between 0.5-1.5

Save the result to **X1** and **y1**.

In [60]:

```
X1, y1 = make_blobs(n_samples=50, centers=[[4,4], [-2, -1], [1, 1], [10,4]], cluster_std=0.9)
```

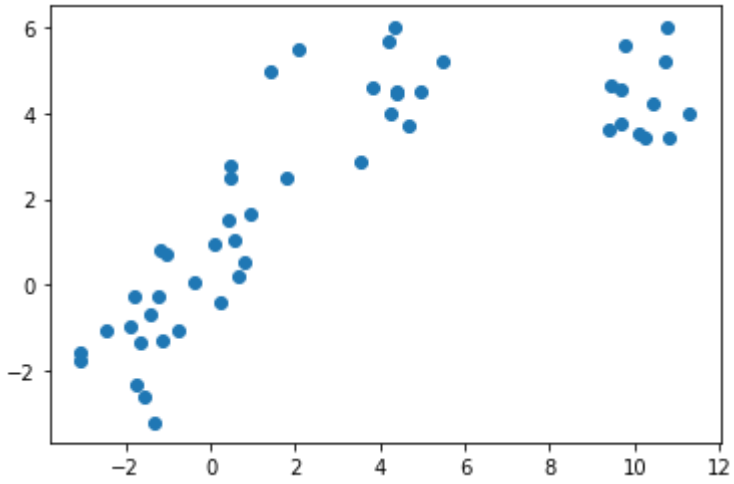
Plot the scatter plot of the randomly generated data

In [61]:

```
plt.scatter(X1[:, 0], X1[:, 1], marker='o')
```

Out[61]:

```
<matplotlib.collections.PathCollection at 0x1d972208488>
```



We will start by clustering the random data points we just created.

The **Agglomerative Clustering** class will require two inputs:

- **n_clusters**: The number of clusters to form as well as the number of centroids to generate.
  - Value will be: 4
- **linkage**: Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.
  - Value will be: 'complete'
  - **Note**: It is recommended you try everything with 'average' as well

Save the result to a variable called **agglom**

In [62]:

```
agglom = AgglomerativeClustering(n_clusters = 4, linkage = 'average')
```

Fit the model with X2 and y2 from the generated data above.

In [63]:

```
agglom.fit(X1,y1)
```

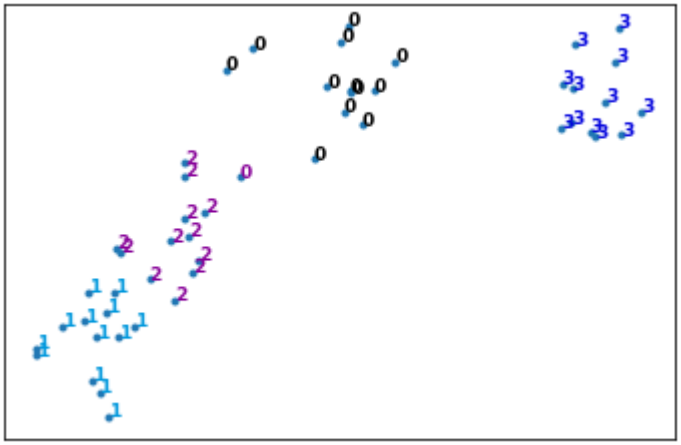
Out[63]:

```
AgglomerativeClustering(affinity='euclidean', compute_full_tree='auto',
                        connectivity=None, distance_threshold=None,
                        linkage='average', memory=None, n_clusters=4)
```

Run the following code to show the clustering! Remember to read the code and comments to gain more understanding on how the plotting works.

In [64]:

```
# Create a figure of size 6 inches by 4 inches.
plt.figure(figsize=(6,4))

# These two lines of code are used to scale the data points down,
# Or else the data points will be scattered very far apart.

# Create a minimum and maximum range of X1.
x_min, x_max = np.min(X1, axis=0), np.max(X1, axis=0)

# Get the average distance for X1.
X1 = (X1 - x_min) / (x_max - x_min)

# This loop displays all of the datapoints.
for i in range(X1.shape[0]):
    # Replace the data points with their respective cluster value
    # (ex. 0) and is color coded with a colormap (plt.cm.spectral)
    plt.text(X1[i, 0], X1[i, 1], str(y1[i]),
             color=plt.cm.nipy_spectral(agglom.labels_[i] / 10.),
             fontdict={'weight': 'bold', 'size': 9})

# Remove the x ticks, y ticks, x and y axis
plt.xticks([])
plt.yticks([])
#plt.axis('off')



# Display the plot of the original data before clustering
plt.scatter(X1[:, 0], X1[:, 1], marker='.')
# Display the plot
plt.show()
```



Dendrogram Associated for the Agglomerative Hierarchical Clustering

Remember that a **distance matrix** contains the **distance from each point to every other point of a dataset** . Use the function **distance_matrix,** which requires **two inputs**. Use the Feature Matrix, **X2** as both inputs and save the distance matrix to a variable called **dist_matrix**

Remember that the distance values are symmetric, with a diagonal of 0's. This is one way of making sure your

matrix is correct.
(print out dist_matrix to make sure it's correct)

In [65]:

```
dist_matrix = distance_matrix(X1,X1)
print(dist_matrix)
```

```
[[0.   1.02 1.02 ... 0.15 0.18 0.14]
 [1.02 0.   0.44 ... 0.97 0.85 0.88]
 [1.02 0.44 0.   ... 1.04 0.87 0.91]
 ...
 [0.15 0.97 1.04 ... 0.   0.17 0.14]
 [0.18 0.85 0.87 ... 0.17 0.   0.04]
 [0.14 0.88 0.91 ... 0.14 0.04 0.  ]]
```

Using the **linkage** class from hierarchy, pass in the parameters:

- The distance matrix
- 'complete' for complete linkage

Save the result to a variable called **Z**

In [66]:

```
Z = hierarchy.linkage(dist_matrix, 'complete')
```

```
C:\Users\ronni\anaconda3\lib\site-packages\ipykernel_launcher.py:1: ClusterWarning:
scipy.cluster: The symmetric non-negative hollow observation matrix looks suspicious
ly like an uncondensed distance matrix
  """Entry point for launching an IPython kernel.
```

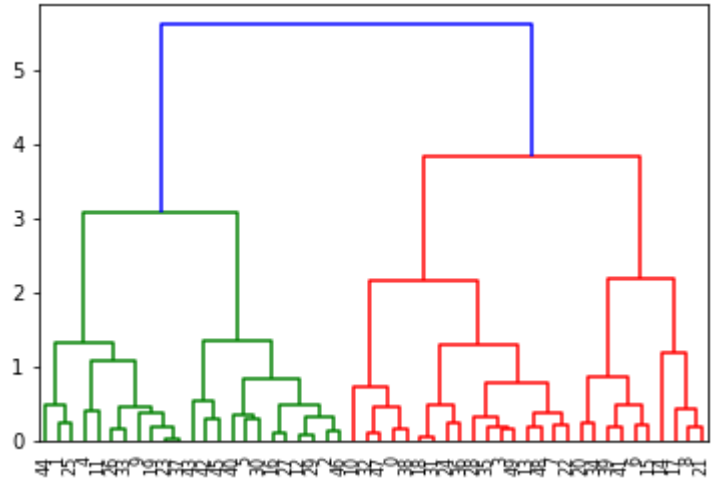
A Hierarchical clustering is typically visualized as a dendrogram as shown in the following cell. Each merge is represented by a horizontal line. The y-coordinate of the horizontal line is the similarity of the two clusters that were merged, where cities are viewed as singleton clusters. By moving up from the bottom layer to the top node, a dendrogram allows us to reconstruct the history of merges that resulted in the depicted clustering.

Next, we will save the dendrogram to a variable called **dendro**. In doing this, the dendrogram will also be displayed. Using the **dendrogram** class from hierarchy, pass in the parameter:

- Z

In [67]:

```
dendro = hierarchy.dendrogram(Z)
```

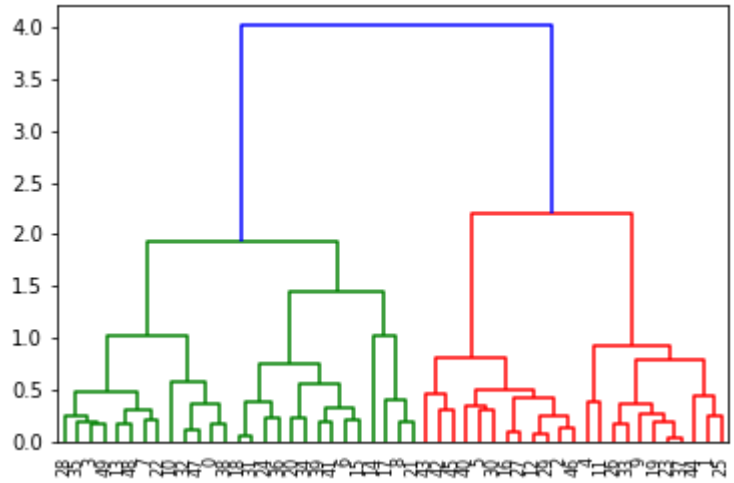


We used **complete** linkage for our case, change it to **average** linkage to see how the dendogram changes.

In [68]:

```
Z = hierarchy.linkage(dist_matrix, 'average')
dendro = hierarchy.dendrogram(Z)
```

```
C:\Users\ronni\anaconda3\lib\site-packages\ipykernel_launcher.py:1: ClusterWarning:
scipy.cluster: The symmetric non-negative hollow observation matrix looks suspicious
ly like an uncondensed distance matrix
  """Entry point for launching an IPython kernel.
```



Imagine that an automobile manufacturer has developed prototypes for a new vehicle. Before introducing the new model into its range, the manufacturer wants to determine which existing vehicles on the market are most like the prototypes--that is, how vehicles can be grouped, which group is the most similar with the model, and therefore which models they will be competing against.

Our objective here, is to use clustering methods, to find the most distinctive clusters of vehicles. It will summarize the existing vehicles and help manufacturers to make decision about the supply of new models.

In [70]:

```
pdf=pd.read_csv('c:/users/ronni/IBM data/machine learning/cars_clus.csv')
pdf.head()
```

Out[70]:

	manufact	model	sales	resale	type	price	engine_s	horsepow	wheelbas	width	le
0	Acura	Integra	16.919	16.360	0.000	21.500	1.800	140.000	101.200	67.300	17
1	Acura	TL	39.384	19.875	0.000	28.400	3.200	225.000	108.100	70.300	19
2	Acura	CL	14.114	18.225	0.000	*null*	3.200	225.000	106.900	70.600	19
3	Acura	RL	8.588	29.725	0.000	42.000	3.500	210.000	114.600	71.400	19
4	Audi	A4	20.397	22.255	0.000	23.990	1.800	150.000	102.600	68.200	17

lets simply clear the dataset by dropping the rows that have null value:

In [71]:

```
print ("Shape of dataset before cleaning: ", pdf.size)
pdf[[ 'sales', 'resale', 'type', 'price', 'engine_s',
       'horsepow', 'wheelbas', 'width', 'length', 'curb_wgt', 'fuel_cap',
       'mpg', 'lnsales']] = pdf[['sales', 'resale', 'type', 'price', 'engine_s',
       'horsepow', 'wheelbas', 'width', 'length', 'curb_wgt', 'fuel_cap',
       'mpg', 'lnsales']].apply(pd.to_numeric, errors='coerce')
pdf = pdf.dropna()
pdf = pdf.reset_index(drop=True)
print ("Shape of dataset after cleaning: ", pdf.size)
pdf.head(5)
```

```
Shape of dataset before cleaning:  2544
Shape of dataset after cleaning:  1872
```

Out[71]:

	manufact	model	sales	resale	type	price	engine_s	horsepow	wheelbas	width	length
0	Acura	Integra	16.919	16.360	0.0	21.50	1.8	140.0	101.2	67.3	172.4
1	Acura	TL	39.384	19.875	0.0	28.40	3.2	225.0	108.1	70.3	192.9
2	Acura	RL	8.588	29.725	0.0	42.00	3.5	210.0	114.6	71.4	196.6
3	Audi	A4	20.397	22.255	0.0	23.99	1.8	150.0	102.6	68.2	178.0
4	Audi	A6	18.780	23.555	0.0	33.95	2.8	200.0	108.7	76.1	192.0

Lets select our feature set:

In [72]:

```
featureset = pdf[['engine_s',  'horsepow', 'wheelbas', 'width', 'length', 'curb_wgt', 'fuel_cap', 'm
```

Now we can normalize the feature set. MinMaxScaler transforms features by scaling each feature to a given range. It is by default (0, 1). That is, this estimator scales and translates each feature individually such that it is between zero and one.

In [73]:

```
from sklearn.preprocessing import MinMaxScaler
x = featureset.values #returns a numpy array
min_max_scaler = MinMaxScaler()
feature_mtx = min_max_scaler.fit_transform(x)
feature_mtx [0:5]
```

Out[73]:

```
array([[0.11, 0.22, 0.19, 0.28, 0.31, 0.23, 0.13, 0.43],
       [0.31, 0.43, 0.34, 0.46, 0.58, 0.5 , 0.32, 0.33],
       [0.36, 0.39, 0.48, 0.53, 0.63, 0.61, 0.35, 0.23],
       [0.11, 0.24, 0.22, 0.34, 0.38, 0.34, 0.28, 0.4 ],
       [0.26, 0.37, 0.35, 0.81, 0.57, 0.52, 0.38, 0.23]])
```

we use Scipy package to cluster the dataset: First, we calculate the distance matrix.

In [74]:

```
import scipy
leng = feature_mtx.shape[0]
D = scipy.zeros([leng,leng])
for i in range(leng):
    for j in range(leng):
        D[i,j] = scipy.spatial.distance.euclidean(feature_mtx[i], feature_mtx[j])
```

```
C:\Users\ronni\anaconda3\lib\site-packages\ipykernel_launcher.py:3: DeprecationWarni
ng: scipy.zeros is deprecated and will be removed in SciPy 2.0.0, use numpy.zeros in
stead
  This is separate from the ipykernel package so we can avoid doing imports until
```

In agglomerative clustering, at each iteration, the algorithm must update the distance matrix to reflect the distance of the newly formed cluster with the remaining clusters in the forest. The following methods are supported in Scipy for calculating the distance between the newly formed cluster and each:

- single
- complete
- average
- weighted
- centroid

We use **complete** for our case, but feel free to change it to see how the results change.

In [75]:

```
import pylab
import scipy.cluster.hierarchy
Z = hierarchy.linkage(D, 'complete')
```

```
C:\Users\ronni\anaconda3\lib\site-packages\ipykernel_launcher.py:3: ClusterWarning:
scipy.cluster: The symmetric non-negative hollow observation matrix looks suspicious
ly like an uncondensed distance matrix
  This is separate from the ipykernel package so we can avoid doing imports until
```

Essentially, Hierarchical clustering does not require a pre-specified number of clusters. However, in some applications we want a partition of disjoint clusters just as in flat clustering. So you can use a cutting line:

In [76]:

```
from scipy.cluster.hierarchy import fcluster
max_d = 3
clusters = fcluster(Z, max_d, criterion='distance')
clusters
```

Out[76]:

```
array([ 1,  5,  5,  6,  5,  4,  6,  5,  5,  5,  5,  5,  4,  4,  5,  1,  6,
        5,  5,  5,  4,  2, 11,  6,  6,  5,  6,  5,  1,  6,  6, 10,  9,  8,
        9,  3,  5,  1,  7,  6,  5,  3,  5,  3,  8,  7,  9,  2,  6,  6,  5,
        4,  2,  1,  6,  5,  2,  7,  5,  5,  5,  4,  4,  3,  2,  6,  6,  5,
        7,  4,  7,  6,  6,  5,  3,  5,  5,  6,  5,  4,  4,  1,  6,  5,  5,
        5,  6,  4,  5,  4,  1,  6,  5,  6,  6,  5,  5,  5,  7,  7,  7,  2,
        2,  1,  2,  6,  5,  1,  1,  1,  7,  8,  1,  1,  6,  1,  1],
      dtype=int32)
```

In [77]:

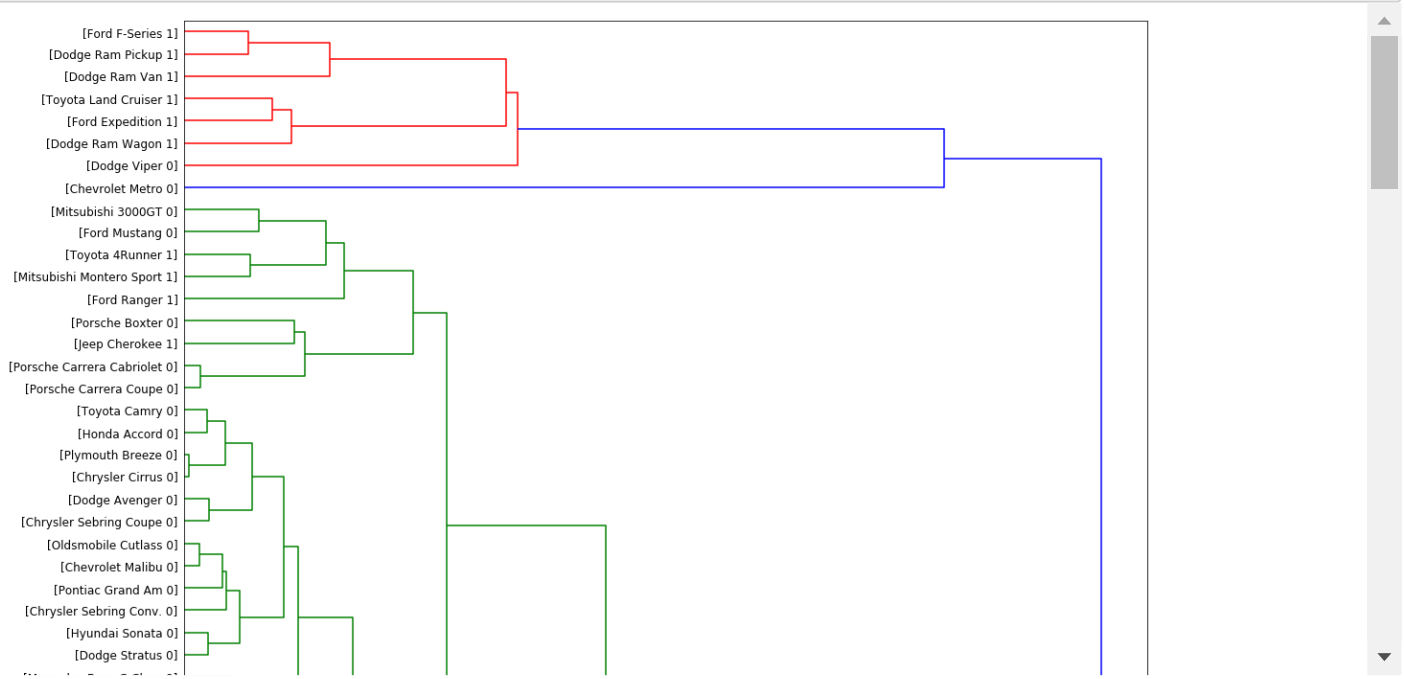
```
#Also, you can determine the number of clusters directly:
from scipy.cluster.hierarchy import fcluster
k = 5
clusters = fcluster(Z, k, criterion='maxclust')
clusters
```

Out[77]:

```
array([1, 3, 3, 3, 3, 2, 3, 3, 3, 3, 3, 3, 2, 2, 3, 1, 3, 3, 3, 3, 2, 1,
       5, 3, 3, 3, 3, 3, 1, 3, 3, 4, 4, 4, 4, 2, 3, 1, 3, 3, 3, 2, 3, 2,
       4, 3, 4, 1, 3, 3, 3, 2, 1, 1, 3, 3, 1, 3, 3, 3, 3, 2, 2, 2, 1, 3,
       3, 3, 3, 2, 3, 3, 3, 3, 2, 3, 3, 3, 3, 2, 2, 1, 3, 3, 3, 3, 3, 2,
       3, 2, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 3, 3, 1, 1, 1,
       3, 4, 1, 1, 3, 1, 1], dtype=int32)
```

In [78]:

```
fig = pylab.figure(figsize=(18,50))
def llf(id):
    return '[%s %s %s]' % (pdf['manufact'][id], pdf['model'][id], int(float(pdf['type'][id])) )

dendro = hierarchy.dendrogram(Z,  leaf_label_func=llf, leaf_rotation=0, leaf_font_size =12, orienta
```



Lets redo it again, but this time using scikit-learn package:

In [79]:

```
dist_matrix = distance_matrix(feature_mtx,feature_mtx)
print(dist_matrix)
```

```
[[0.   0.58 0.75 ... 0.29 0.25 0.19]
 [0.58 0.   0.23 ... 0.36 0.66 0.62]
 [0.75 0.23 0.   ... 0.52 0.82 0.78]
 ...
 [0.29 0.36 0.52 ... 0.   0.42 0.36]
 [0.25 0.66 0.82 ... 0.42 0.   0.15]
 [0.19 0.62 0.78 ... 0.36 0.15 0.  ]]
```

Now, we can use the 'AgglomerativeClustering' function from scikit-learn library to cluster the dataset. The AgglomerativeClustering performs a hierarchical clustering using a bottom up approach. The linkage criteria determines the metric used for the merge strategy:

- Ward minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.
- Maximum or complete linkage minimizes the maximum distance between observations of pairs of clusters.
- Average linkage minimizes the average of the distances between all observations of pairs of clusters.

In [80]:

```
agglom = AgglomerativeClustering(n_clusters = 6, linkage = 'complete')
agglom.fit(feature_mtx)
agglom.labels_
```

Out[80]:

```
array([1, 2, 2, 1, 2, 3, 1, 2, 2, 2, 2, 2, 3, 3, 2, 1, 1, 2, 2, 2, 5, 1,
       4, 1, 1, 2, 1, 2, 1, 1, 1, 5, 0, 0, 0, 3, 2, 1, 2, 1, 2, 3, 2, 3,
       0, 3, 0, 1, 1, 1, 2, 3, 1, 1, 1, 2, 1, 1, 2, 2, 2, 3, 3, 3, 1, 1,
       1, 2, 1, 2, 2, 1, 1, 2, 3, 2, 3, 1, 2, 3, 5, 1, 1, 2, 3, 2, 1, 3,
       2, 3, 1, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,
       2, 0, 1, 1, 1, 1, 1], dtype=int64)
```

And, we can add a new field to our dataframe to show the cluster of each row:

In [81]:

```
pdf['cluster_'] = agglom.labels_
pdf.head()
```

Out[81]:

	manufact	model	sales	resale	type	price	engine_s	horsepow	wheelbas	width	length
0	Acura	Integra	16.919	16.360	0.0	21.50	1.8	140.0	101.2	67.3	172.4
1	Acura	TL	39.384	19.875	0.0	28.40	3.2	225.0	108.1	70.3	192.9
2	Acura	RL	8.588	29.725	0.0	42.00	3.5	210.0	114.6	71.4	196.6
3	Audi	A4	20.397	22.255	0.0	23.99	1.8	150.0	102.6	68.2	178.0
4	Audi	A6	18.780	23.555	0.0	33.95	2.8	200.0	108.7	76.1	192.0

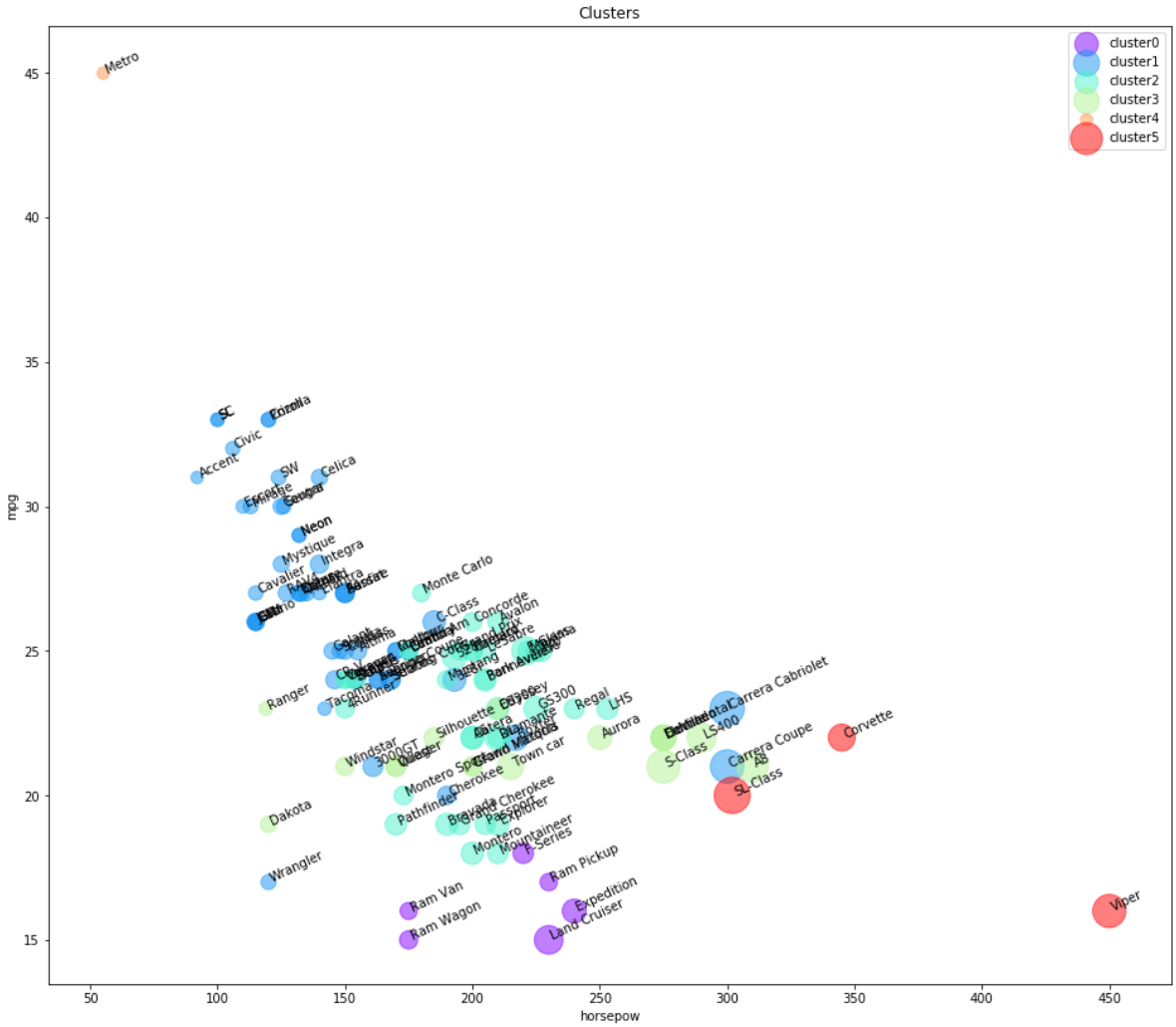
In [82]:

```
import matplotlib.cm as cm
n_clusters = max(agglom.labels_)+1
colors = cm.rainbow(np.linspace(0, 1, n_clusters))
cluster_labels = list(range(0, n_clusters))

# Create a figure of size 6 inches by 4 inches.
plt.figure(figsize=(16,14))

for color, label in zip(colors, cluster_labels):
    subset = pdf[pdf.cluster_ == label]
    for i in subset.index:
            plt.text(subset.horsepow[i], subset.mpg[i],str(subset['model'][i]), rotation=25)
    plt.scatter(subset.horsepow, subset.mpg, s= subset.price*10, c=color, label='cluster'+str(label]
#    plt.scatter(subset.horsepow, subset.mpg)
plt.legend()
plt.title('Clusters')
plt.xlabel('horsepow')
plt.ylabel('mpg')
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
```

Out[82]:

```
Text(0, 0.5, 'mpg')
```



As you can see, we are seeing the distribution of each cluster using the scatter plot, but it is not very clear where is the centroid of each cluster. Moreover, there are 2 types of vehicles in our dataset, "truck" (value of 1 in the type column) and "car" (value of 1 in the type column). So, we use them to distinguish the classes, and summarize the cluster. First we count the number of cases in each group:

In [83]:

```
pdf.groupby(['cluster_','type'])['cluster_'].count()
```

Out[83]:

```
cluster_  type
0         1.0      6
1         0.0     47
          1.0      5
2         0.0     27
          1.0     11
3         0.0     10
          1.0      7
4         0.0      1
5         0.0      3
Name: cluster_, dtype: int64
```

Now we can look at the characteristics of each cluster:

In [84]:

```
agg_cars = pdf.groupby(['cluster_','type'])['horsepow','engine_s','mpg','price'].mean()
agg_cars
```

```
C:\Users\ronni\anaconda3\lib\site-packages\ipykernel_launcher.py:1: FutureWarning: I
ndexing with multiple keys (implicitly converted to a tuple of keys) will be depreca
ted, use a list instead.
  """Entry point for launching an IPython kernel.
```

Out[84]:

		horsepow	engine_s	mpg	price
cluster_	**type**				
0	**1.0**	211.666667	4.483333	16.166667	29.024667
1	**0.0**	146.531915	2.246809	27.021277	20.306128
	1.0	145.000000	2.580000	22.200000	17.009200
2	**0.0**	203.111111	3.303704	24.214815	27.750593
	1.0	182.090909	3.345455	20.181818	26.265364
3	**0.0**	256.500000	4.410000	21.500000	42.870400
	1.0	160.571429	3.071429	21.428571	21.527714
4	**0.0**	55.000000	1.000000	45.000000	9.235000
5	**0.0**	365.666667	6.233333	19.333333	66.010000

It is obvious that we have 3 main clusters with the majority of vehicles in those.

Cars:

- Cluster 1: with almost high mpg, and low in horsepower.
- Cluster 2: with good mpg and horsepower, but higher price than average.
- Cluster 3: with low mpg, high horsepower, highest price.

Trucks:

- Cluster 1: with almost highest mpg among trucks, and lowest in horsepower and price.
- Cluster 2: with almost low mpg and medium horsepower, but higher price than average.
- Cluster 3: with good mpg and horsepower, low price.

Please notice that we did not use **type** , and **price** of cars in the clustering process, but Hierarchical clustering could forge the clusters and discriminate them with quite high accuracy.

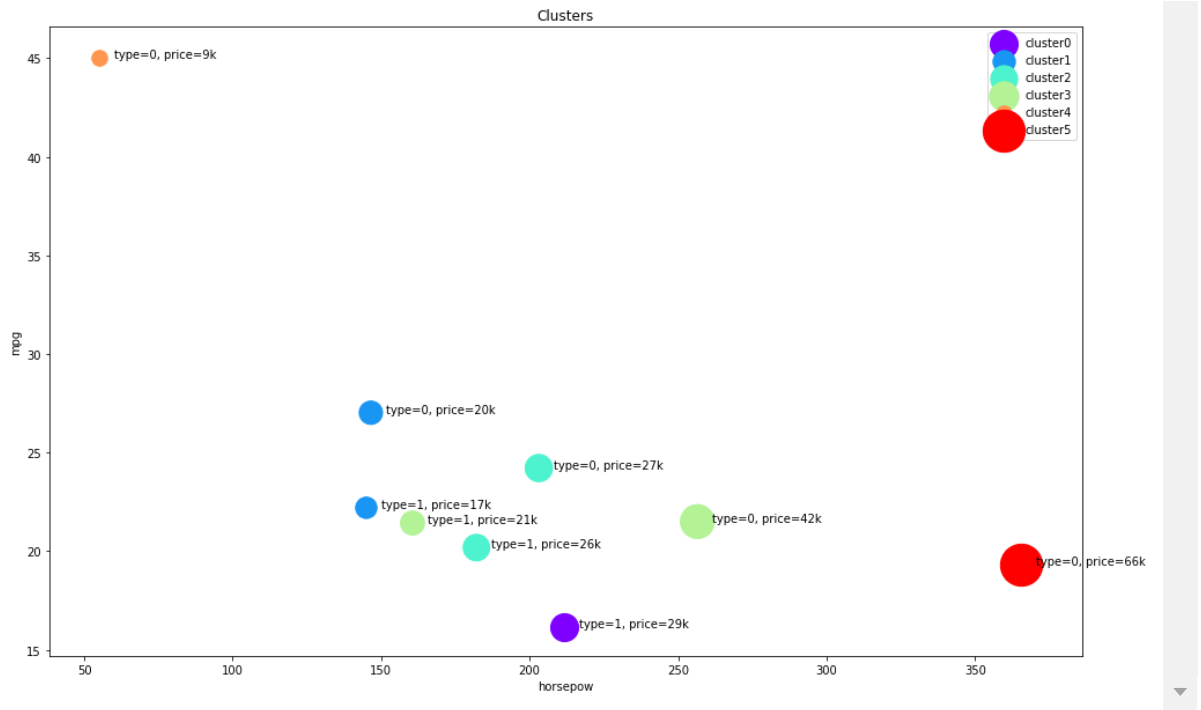
In [86]:

```
plt.figure(figsize=(16,10))
for color, label in zip(colors, cluster_labels):
    subset = agg_cars.loc[(label,),]
    for i in subset.index:
        plt.text(subset.loc[i][0]+5, subset.loc[i][2], 'type='+str(int(i)) + ', price='+str(int(sub
    plt.scatter(subset.horsepow, subset.mpg, s=subset.price*20, c=color, label='cluster'+str(label)
plt.legend()
plt.title('Clusters')
plt.xlabel('horsepow')
plt.ylabel('mpg')
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
```

Out[86]:

```
Text(0, 0.5, 'mpg')
```



Density based clustering

Most of the traditional clustering techniques, such as k-means, hierarchical and fuzzy clustering, can be used to group data without supervision.

However, when applied to tasks with arbitrary shape clusters, or clusters within cluster, the traditional techniques might be unable to achieve good results. That is, elements in the same cluster might not share enough similarity or the performance may be poor. Additionally, Density-based Clustering locates regions of high density that are separated from one another by regions of low density. Density, in this context, is defined as the number of points within a specified radius.

In this section, the main focus will be manipulating the data and properties of DBSCAN and observing the resulting clustering.

In [2]:

```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.datasets.samples_generator import make_blobs
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
%matplotlib inline
```

The function below will generate the data points and requires these inputs:

- **centroidLocation**: Coordinates of the centroids that will generate the random data.
  - Example: input: [[4,3], [2,-1], [-1,4]]
- **numSamples**: The number of data points we want generated, split over the number of centroids (# of centroids defined in centroidLocation)
  - Example: 1500
- **clusterDeviation**: The standard deviation between the clusters. The larger the number, the further the spacing.
  - Example: 0.5

In [88]:

```
def createDataPoints(centroidLocation, numSamples, clusterDeviation):
    # Create random data and store in feature matrix X and response vector y.
    X, y = make_blobs(n_samples=numSamples, centers=centroidLocation,
                                cluster_std=clusterDeviation)

    # Standardize features by removing the mean and scaling to unit variance
    X = StandardScaler().fit_transform(X)
    return X, y
```

Use **createDataPoints** with the **3 inputs** and store the output into variables **X** and **y**.

In [89]:

```
X, y = createDataPoints([[4,3], [2,-1], [-1,4]] , 1500, 0.5)
```

DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. This technique is one of the most common clustering algorithms which works based on density of object. The whole idea is that if a particular point belongs to a cluster, it should be near to lots of other points in that cluster.

It works based on two parameters: Epsilon and Minimum Points
Epsilon determine a specified radius that if includes enough number of points within, we call it dense area
minimumSamples determine the minimum number of data points we want in a neighborhood to define a cluster.

In [90]:

```
epsilon = 0.3
minimumSamples = 7
db = DBSCAN(eps=epsilon, min_samples=minimumSamples).fit(X)
labels = db.labels_
labels
```

Out[90]:

```
array([0, 1, 0, ..., 2, 0, 2], dtype=int64)
```

Lets Replace all elements with 'True' in core_samples_mask that are in the cluster, 'False' if the points are outliers.

In [91]:

```
# Firts, create an array of booleans using the labels from db.
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
core_samples_mask
```

Out[91]:

```
array([ True,  True,  True, ...,  True,  True,  True])
```

In [92]:

```
# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_clusters_
```

Out[92]:

3

In [93]:

```
# Remove repetition in labels by turning it into a set.
unique_labels = set(labels)
unique_labels
```

Out[93]:

{0, 1, 2}

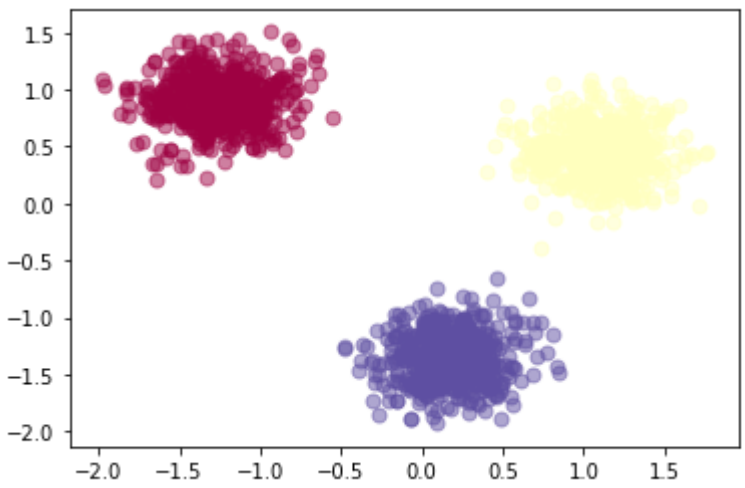
Then do the data visualization

In [94]:

```
# Create colors for the clusters.
colors = plt.cm.Spectral(np.linspace(0, 1, len(unique_labels)))
```

In [95]:

```
# Plot the points with colors
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = 'k'

    class_member_mask = (labels == k)

    # Plot the datapoints that are clustered
    xy = X[class_member_mask & core_samples_mask]
    plt.scatter(xy[:, 0], xy[:, 1],s=50, c=[col], marker=u'o', alpha=0.5)

    # Plot the outliers
    xy = X[class_member_mask & ~core_samples_mask]
    plt.scatter(xy[:, 0], xy[:, 1],s=50, c=[col], marker=u'o', alpha=0.5)
```

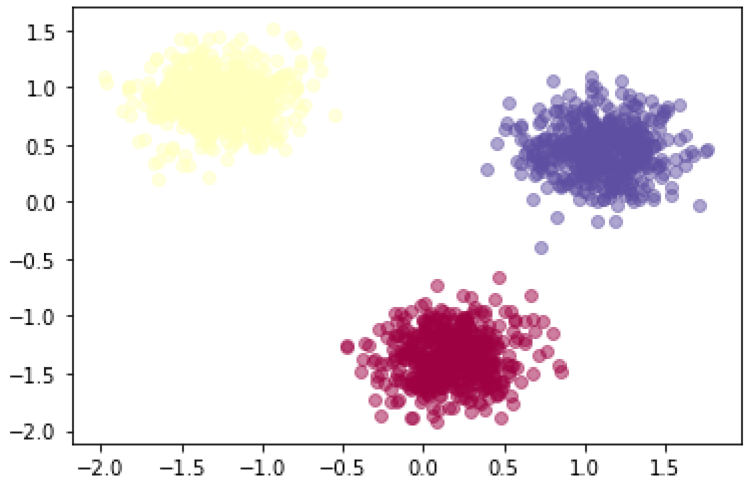


try to cluster the above dataset into 3 clusters using k-Means.

In [96]:

```
from sklearn.cluster import KMeans
k = 3
k_means3 = KMeans(init = "k-means++", n_clusters = k, n_init = 12)
k_means3.fit(X)
fig = plt.figure(figsize=(6, 4))
ax = fig.add_subplot(1, 1, 1)
for k, col in zip(range(k), colors):
    my_members = (k_means3.labels_ == k)
    plt.scatter(X[my_members, 0], X[my_members, 1],  c=col, marker=u'o', alpha=0.5)
plt.show()
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoid
ed as value-mapping will have precedence in case its length matches with 'x' & 'y'.
Please use a 2-D array with a single row if you really want to specify the same RGB
or RGBA value for all points.
```



DBSCAN is specially very good for tasks like class identification on a spatial context. The wonderful attribute of DBSCAN algorithm is that it can find out any arbitrary shape cluster without getting affected by noise. For example, this following example cluster the location of weather stations in Canada. <Click 1> DBSCAN can be used here, for instance, to find the group of stations which show the same weather condition. As you can see, it not only finds different arbitrary shaped clusters, can find the denser part of data-centered samples by ignoring less-dense areas or noises.

In [3]:

```
pdf=pd.read_csv('c:/users/ronni/IBM data/machine learning/weather-stations20140101-20141231.csv')
pdf.head()
```

Out[3]:

	Stn_Name	Lat	Long	Prov	Tm	DwTm	D	Tx	DwTx	Tn	...	DwP	P%N
0	CHEMAINUS	48.935	-123.742	BC	8.2	0.0	NaN	13.5	0.0	1.0	...	0.0	NaN
1	COWICHAN LAKE FORESTRY	48.824	-124.133	BC	7.0	0.0	3.0	15.0	0.0	-3.0	...	0.0	104.0
2	LAKE COWICHAN	48.829	-124.052	BC	6.8	13.0	2.8	16.0	9.0	-2.5	...	9.0	NaN
3	DISCOVERY ISLAND	48.425	-123.226	BC	NaN	NaN	NaN	12.5	0.0	NaN	...	NaN	NaN
4	DUNCAN KELVIN CREEK	48.735	-123.728	BC	7.7	2.0	3.4	14.5	2.0	-1.0	...	2.0	NaN

5 rows × 25 columns

Lets remove rows that dont have any value in the **Tm** field.

In [4]:

```
pdf = pdf[pd.notnull(pdf["Tm"])]
pdf = pdf.reset_index(drop=True)
pdf.head(5)
```

Out[4]:

	Stn_Name	Lat	Long	Prov	Tm	DwTm	D	Tx	DwTx	Tn	...	DwP	P%N	S
0	CHEMAINUS	48.935	-123.742	BC	8.2	0.0	NaN	13.5	0.0	1.0	...	0.0	NaN	
1	COWICHAN LAKE FORESTRY	48.824	-124.133	BC	7.0	0.0	3.0	15.0	0.0	-3.0	...	0.0	104.0	
2	LAKE COWICHAN	48.829	-124.052	BC	6.8	13.0	2.8	16.0	9.0	-2.5	...	9.0	NaN	N
3	DUNCAN KELVIN CREEK	48.735	-123.728	BC	7.7	2.0	3.4	14.5	2.0	-1.0	...	2.0	NaN	N
4	ESQUIMALT HARBOUR	48.432	-123.439	BC	8.8	0.0	NaN	13.1	0.0	1.9	...	8.0	NaN	N

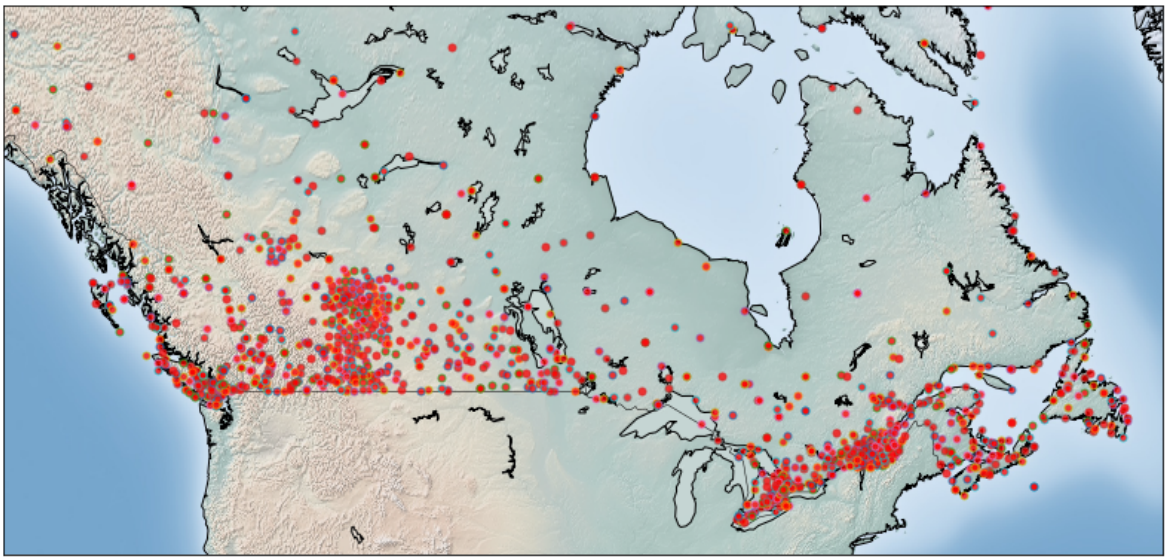
5 rows × 25 columns

Visualization of stations on map using basemap package. The matplotlib basemap toolkit is a library for plotting 2D data on maps in Python. Basemap does not do any plotting on it's own, but provides the facilities to transform coordinates to a map projections.

Please notice that the size of each data points represents the average of maximum temperature for each station in a year.

In [14]:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from pylab import rcParams
%matplotlib inline
rcParams['figure.figsize'] = (14,10)

llon=-140
ulon=-50
llat=40
ulat=65

pdf = pdf[(pdf['Long'] > llon) & (pdf['Long'] < ulon) & (pdf['Lat'] > llat) &(pdf['Lat'] < ulat)]

my_map = Basemap(projection='merc',
            resolution = 'l', area_thresh = 1000.0,
            llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)
            urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
# my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

# To collect data based on stations

xs,ys = my_map(np.asarray(pdf.Long), np.asarray(pdf.Lat))
pdf['xm']= xs.tolist()
pdf['ym'] =ys.tolist()

#Visualization1
for index,row in pdf.iterrows():
#   x,y = my_map(row.Long, row.Lat)
   my_map.plot(row.xm, row.ym,markerfacecolor =([1,0,0]),  marker='o', markersize= 5, alpha = 0.75)
#plt.text(x,y,stn)
plt.show()
```



DBSCAN form sklearn library can runs DBSCAN clustering from vector array or distance matrix. In our case, we pass it the Numpy array Clus_dataSet to find core samples of high density and expands clusters from them.

In [15]:

```
from sklearn.cluster import DBSCAN
import sklearn.utils
from sklearn.preprocessing import StandardScaler
sklearn.utils.check_random_state(1000)
Clus_dataSet = pdf[['xm','ym']]
Clus_dataSet = np.nan_to_num(Clus_dataSet)
Clus_dataSet = StandardScaler().fit_transform(Clus_dataSet)

# Compute DBSCAN
db = DBSCAN(eps=0.15, min_samples=10).fit(Clus_dataSet)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
pdf["Clus_Db"]=labels

realClusterNum=len(set(labels)) - (1 if -1 in labels else 0)
clusterNum = len(set(labels))


# A sample of clusters
pdf[["Stn_Name","Tx","Tm","Clus_Db"]].head(5)
```

Out[15]:

	Stn_Name	Tx	Tm	Clus_Db
0	CHEMAINUS	13.5	8.2	0
1	COWICHAN LAKE FORESTRY	15.0	7.0	0
2	LAKE COWICHAN	16.0	6.8	0
3	DUNCAN KELVIN CREEK	14.5	7.7	0
4	ESQUIMALT HARBOUR	13.1	8.8	0

In [16]:

```
set(labels)
```

Out[16]:

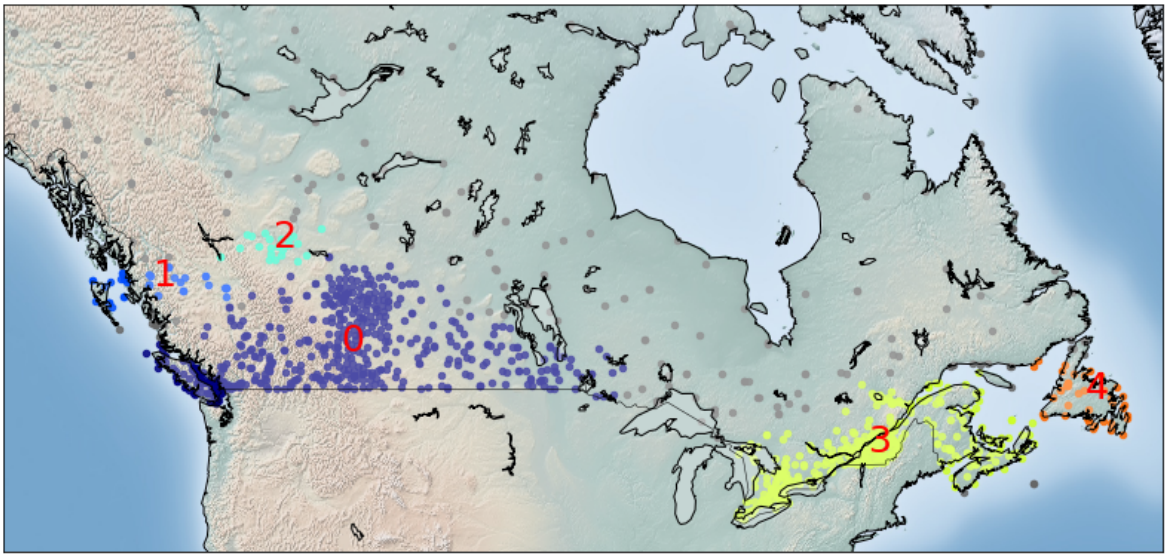
```
{-1, 0, 1, 2, 3, 4}
```

Now, we can visualize the clusters using basemap:

In [17]:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from pylab import rcParams
%matplotlib inline
rcParams['figure.figsize'] = (14,10)

my_map = Basemap(projection='merc',
            resolution = 'l', area_thresh = 1000.0,
            llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)
            urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
#my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

# To create a color map
colors = plt.get_cmap('jet')(np.linspace(0.0, 1.0, clusterNum))



#Visualization1
for clust_number in set(labels):
    c=(([0.4,0.4,0.4]) if clust_number == -1 else colors[np.int(clust_number)])
    clust_set = pdf[pdf.Clus_Db == clust_number]
    my_map.scatter(clust_set.xm, clust_set.ym, color =c,  marker='o', s= 20, alpha = 0.85)
    if clust_number != -1:
        cenx=np.mean(clust_set.xm)
        ceny=np.mean(clust_set.ym)
        plt.text(cenx,ceny,str(clust_number), fontsize=25, color='red',)
        print ("Cluster "+str(clust_number)+', Avg Temp: '+ str(np.mean(clust_set.Tm)))
```

```
Cluster 0, Avg Temp: -5.538747553816051
Cluster 1, Avg Temp: 1.9526315789473685
Cluster 2, Avg Temp: -9.195652173913045
Cluster 3, Avg Temp: -15.300833333333333
Cluster 4, Avg Temp: -7.769047619047619
```



In this section we re-run DBSCAN, but this time on a 5-dimensional dataset:

In [18]:

```
from sklearn.cluster import DBSCAN
import sklearn.utils
from sklearn.preprocessing import StandardScaler
sklearn.utils.check_random_state(1000)
Clus_dataSet = pdf[['xm','ym','Tx','Tm','Tn']]
Clus_dataSet = np.nan_to_num(Clus_dataSet)
Clus_dataSet = StandardScaler().fit_transform(Clus_dataSet)

# Compute DBSCAN
db = DBSCAN(eps=0.3, min_samples=10).fit(Clus_dataSet)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
pdf["Clus_Db"]=labels

realClusterNum=len(set(labels)) - (1 if -1 in labels else 0)
clusterNum = len(set(labels))


# A sample of clusters
pdf[["Stn_Name","Tx","Tm","Clus_Db"]].head(5)
```

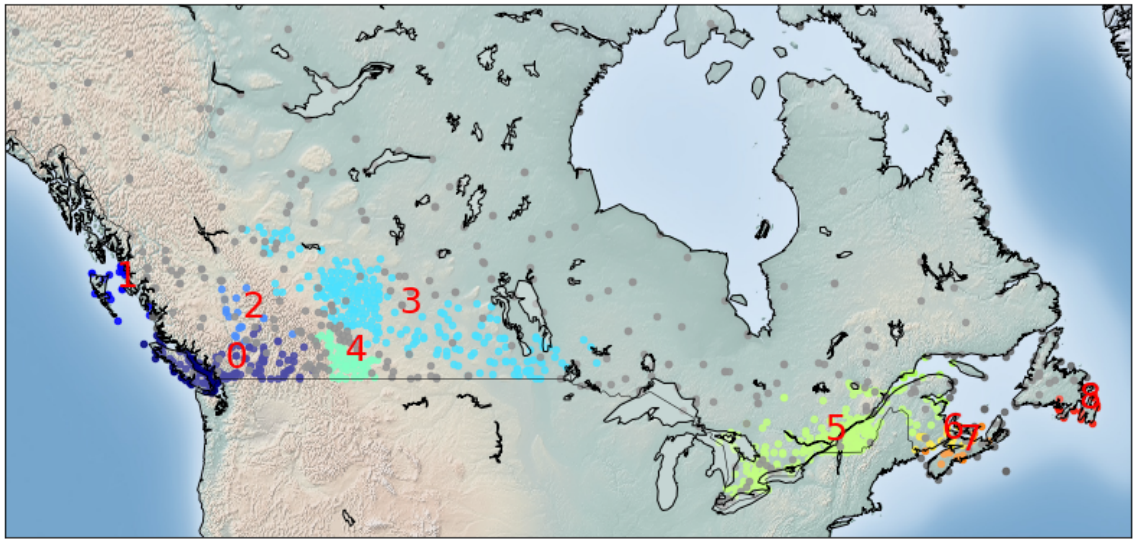
Out[18]:

	Stn_Name	Tx	Tm	Clus_Db
0	CHEMAINUS	13.5	8.2	0
1	COWICHAN LAKE FORESTRY	15.0	7.0	0
2	LAKE COWICHAN	16.0	6.8	0
3	DUNCAN KELVIN CREEK	14.5	7.7	0
4	ESQUIMALT HARBOUR	13.1	8.8	0

In [19]:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from pylab import rcParams
%matplotlib inline
rcParams['figure.figsize'] = (14,10)

my_map = Basemap(projection='merc',
            resolution = 'l', area_thresh = 1000.0,
            llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)
            urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
#my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

# To create a color map
colors = plt.get_cmap('jet')(np.linspace(0.0, 1.0, clusterNum))



#Visualization1
for clust_number in set(labels):
    c=(([0.4,0.4,0.4]) if clust_number == -1 else colors[np.int(clust_number)])
    clust_set = pdf[pdf.Clus_Db == clust_number]
    my_map.scatter(clust_set.xm, clust_set.ym, color =c,  marker='o', s= 20, alpha = 0.85)
    if clust_number != -1:
        cenx=np.mean(clust_set.xm)
        ceny=np.mean(clust_set.ym)
        plt.text(cenx,ceny,str(clust_number), fontsize=25, color='red',)
        print ("Cluster "+str(clust_number)+', Avg Temp: '+ str(np.mean(clust_set.Tm)))
```

```
Cluster 0, Avg Temp: 6.2211920529801334
Cluster 1, Avg Temp: 6.790000000000001
Cluster 2, Avg Temp: -0.49411764705882355
Cluster 3, Avg Temp: -13.877209302325586
Cluster 4, Avg Temp: -4.186274509803922
Cluster 5, Avg Temp: -16.301503759398482
Cluster 6, Avg Temp: -13.599999999999998
Cluster 7, Avg Temp: -9.753333333333334
Cluster 8, Avg Temp: -4.258333333333334
```



Content Based filtering

Recommendation systems are a collection of algorithms used to recommend items to users based on information taken from the user. These systems have become ubiquitous, and can be commonly seen in online stores, movies databases and job finders. In this notebook, we will explore Content-based recommendation systems and implement a simple version of one using Python and the Pandas library.

In [20]:

```
#Dataframe manipulation library
import pandas as pd
#Math functions, we'll only need the sqrt function so let's import only that
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [23]:

```
movies_df=pd.read_csv('F:/dataset/ml-latest/movies.csv')
ratings_df = pd.read_csv('F:/dataset/ml-latest/ratings.csv')
movies_df.head()
```

Out[23]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

Let's also remove the year from the title column by using pandas' replace function and store in a new year column.

In [24]:

```
#Using regular expressions to find a year stored between parentheses
#We specify the parantheses so we don't conflict with movies that have years in their titles
movies_df['year'] = movies_df.title.str.extract('(\(\d\d\d\d\))',expand=False)
#Removing the parentheses
movies_df['year'] = movies_df.year.str.extract('(\d\d\d\d)',expand=False)
#Removing the years from the 'title' column
movies_df['title'] = movies_df.title.str.replace('(\(\d\d\d\d\))', '')
#Applying the strip function to get rid of any ending whitespace characters that may have appeared
movies_df['title'] = movies_df['title'].apply(lambda x: x.strip())
movies_df.head()
```

Out[24]:

	movieId	title	genres	year
0	1	Toy Story	Adventure|Animation|Children|Comedy|Fantasy	1995
1	2	Jumanji	Adventure|Children|Fantasy	1995
2	3	Grumpier Old Men	Comedy|Romance	1995
3	4	Waiting to Exhale	Comedy|Drama|Romance	1995
4	5	Father of the Bride Part II	Comedy	1995

With that, let's also split the values in the Genres column into a list of Genres to simplify future use. This can be achieved by applying Python's split string function on the correct column.

In [25]:

```
#Every genre is separated by a | so we simply have to call the split function on |
movies_df['genres'] = movies_df.genres.str.split('|')
movies_df.head()
```

Out[25]:

	movieId	title	genres	year
0	1	Toy Story	[Adventure, Animation, Children, Comedy, Fantasy]	1995
1	2	Jumanji	[Adventure, Children, Fantasy]	1995
2	3	Grumpier Old Men	[Comedy, Romance]	1995
3	4	Waiting to Exhale	[Comedy, Drama, Romance]	1995
4	5	Father of the Bride Part II	[Comedy]	1995

Since keeping genres in a list format isn't optimal for the content-based recommendation system technique, we will use the One Hot Encoding technique to convert the list of genres to a vector where each column corresponds to one possible value of the feature. This encoding is needed for feeding categorical data. In this case, we store every different genre in columns that contain either 1 or 0. 1 shows that a movie has that genre and 0 shows that it doesn't. Let's also store this dataframe in another variable since genres won't be important for our first recommendation system.

In [26]:

```
#Copying the movie dataframe into a new one since we won't need to use the genre information in our
moviesWithGenres_df = movies_df.copy()

#For every row in the dataframe, iterate through the list of genres and place a 1 into the correspon
for index, row in movies_df.iterrows():
    for genre in row['genres']:
        moviesWithGenres_df.at[index, genre] = 1
#Filling in the NaN values with 0 to show that a movie doesn't have that column's genre
moviesWithGenres_df = moviesWithGenres_df.fillna(0)
moviesWithGenres_df.head()
```

Out[26]:

	movieId	title	genres	year	Adventure	Animation	Children	Comedy	Fantasy	Ro
0	1	Toy Story	[Adventure, Animation, Children, Comedy, Fantasy]	1995	1.0	1.0	1.0	1.0	1.0	
1	2	Jumanji	[Adventure, Children, Fantasy]	1995	1.0	0.0	1.0	0.0	1.0	
2	3	Grumpier Old Men	[Comedy, Romance]	1995	0.0	0.0	0.0	1.0	0.0	
3	4	Waiting to Exhale	[Comedy, Drama, Romance]	1995	0.0	0.0	0.0	1.0	0.0	
4	5	Father of the Bride Part II	[Comedy]	1995	0.0	0.0	0.0	1.0	0.0	

5 rows × 24 columns

In [27]:

```
ratings_df.head()
```

Out[27]:

	userId	movieId	rating	timestamp
0	1	169	2.5	1204927694
1	1	2471	3.0	1204927438
2	1	48516	5.0	1204927435
3	2	2571	3.5	1436165433
4	2	109487	4.0	1436165496

Every row in the ratings dataframe has a user id associated with at least one movie, a rating and a timestamp showing when they reviewed it. We won't be needing the timestamp column, so let's drop it to save on memory.

In [28]:

```
#Drop removes a specified row or column from a dataframe
ratings_df = ratings_df.drop('timestamp', 1)
ratings_df.head()
```

Out[28]:

	userId	movieId	rating
0	1	169	2.5
1	1	2471	3.0
2	1	48516	5.0
3	2	2571	3.5
4	2	109487	4.0

Now, let's take a look at how to implement Content-Based or Item-Item recommendation systems. This technique attempts to figure out what a user's favourite aspects of an item is, and then recommends items that present those aspects. In our case, we're going to try to figure out the input's favorite genres from the movies and ratings given.

Let's begin by creating an input user to recommend movies to:

Notice: To add more movies, simply increase the amount of elements in the userInput. Feel free to add more in! Just be sure to write it in with capital letters and if a movie starts with a "The", like "The Matrix" then write it in like this: 'Matrix, The' .

In [29]:

```
userInput = [
            {'title':'Breakfast Club, The', 'rating':5},
            {'title':'Toy Story', 'rating':3.5},
            {'title':'Jumanji', 'rating':2},
            {'title':"Pulp Fiction", 'rating':5},
            {'title':'Akira', 'rating':4.5}
        ]
inputMovies = pd.DataFrame(userInput)
inputMovies
```

Out[29]:

	title	rating
0	Breakfast Club, The	5.0
1	Toy Story	3.5
2	Jumanji	2.0
3	Pulp Fiction	5.0
4	Akira	4.5

Add movieId to input user

With the input complete, let's extract the input movie's ID's from the movies dataframe and add them into it.

We can achieve this by first filtering out the rows that contain the input movie's title and then merging this subset with the input dataframe. We also drop unnecessary columns for the input to save memory space.

In [30]:

```
#Filtering out the movies by title
inputId = movies_df[movies_df['title'].isin(inputMovies['title'].tolist())]
#Then merging it so we can get the movieId. It's implicitly merging it by title.
inputMovies = pd.merge(inputId, inputMovies)
#Dropping information we won't use from the input dataframe
inputMovies = inputMovies.drop('genres', 1).drop('year', 1)
#Final input dataframe
#If a movie you added in above isn't here, then it might not be in the original
#dataframe or it might spelled differently, please check capitalisation.
inputMovies
```

Out[30]:

	movieId	title	rating
0	1	Toy Story	3.5
1	2	Jumanji	2.0
2	296	Pulp Fiction	5.0
3	1274	Akira	4.5
4	1968	Breakfast Club, The	5.0

We're going to start by learning the input's preferences, so let's get the subset of movies that the input has watched from the Dataframe containing genres defined with binary values.

In [31]:

```
#Filtering out the movies from the input
userMovies = moviesWithGenres_df[moviesWithGenres_df['movieId'].isin(inputMovies['movieId'].tolist()
userMovies
```

Out[31]:

	movieId	title	genres	year	Adventure	Animation	Children	Comedy	Fantasy
0	1	Toy Story	[Adventure, Animation, Children, Comedy, Fantasy]	1995	1.0	1.0	1.0	1.0	1.0
1	2	Jumanji	[Adventure, Children, Fantasy]	1995	1.0	0.0	1.0	0.0	1.0
293	296	Pulp Fiction	[Comedy, Crime, Drama, Thriller]	1994	0.0	0.0	0.0	1.0	0.0
1246	1274	Akira	[Action, Adventure, Animation, Sci-Fi]	1988	1.0	1.0	0.0	0.0	0.0
1885	1968	Breakfast Club, The	[Comedy, Drama]	1985	0.0	0.0	0.0	1.0	0.0

5 rows × 24 columns

We'll only need the actual genre table, so let's clean this up a bit by resetting the index and dropping the movieId, title, genres and year columns.

In [32]:

```
#Resetting the index to avoid future issues
userMovies = userMovies.reset_index(drop=True)
#Dropping unnecessary issues due to save memory and to avoid issues
userGenreTable = userMovies.drop('movieId', 1).drop('title', 1).drop('genres', 1).drop('year', 1)
userGenreTable
```

Out[32]:

	Adventure	Animation	Children	Comedy	Fantasy	Romance	Drama	Action	Crime	Thrille
0	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.(
1	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.(
2	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	1.(
3	1.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.(
4	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.(

Now we're ready to start learning the input's preferences!

To do this, we're going to turn each genre into weights. We can do this by using the input's reviews and multiplying them into the input's genre table and then summing up the resulting table by column. This operation is actually a dot product between a matrix and a vector, so we can simply accomplish by calling Pandas's "dot" function.

In [34]:

```
inputMovies['rating']
```

Out[34]:

```
0    3.5
1    2.0
2    5.0
3    4.5
4    5.0
Name: rating, dtype: float64
```

In [35]:

```
#Dot produt to get weights
userProfile = userGenreTable.transpose().dot(inputMovies['rating'])
#The user profile
userProfile
```

Out[35]:

```
Adventure            10.0
Animation             8.0
Children              5.5
Comedy               13.5
Fantasy               5.5
Romance               0.0
Drama                10.0
Action                4.5
Crime                 5.0
Thriller              5.0
Horror                0.0
Mystery               0.0
Sci-Fi                4.5
IMAX                  0.0
Documentary           0.0
War                   0.0
Musical               0.0
Western               0.0
Film-Noir             0.0
(no genres listed)    0.0
dtype: float64
```

Now, we have the weights for every of the user's preferences. This is known as the User Profile. Using this, we can recommend movies that satisfy the user's preferences.

Let's start by extracting the genre table from the original dataframe:

In [36]:

```
#Now let's get the genres of every movie in our original dataframe
genreTable = moviesWithGenres_df.set_index(moviesWithGenres_df['movieId'])
#And drop the unnecessary information
genreTable = genreTable.drop('movieId', 1).drop('title', 1).drop('genres', 1).drop('year', 1)
genreTable.head()
```

Out[36]:

movieId	Adventure	Animation	Children	Comedy	Fantasy	Romance	Drama	Action	Crime
1	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0
2	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0
4	0.0	0.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0
5	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0

In [37]:

```
genreTable.shape
```

Out[37]:

```
(34208, 20)
```

With the input's profile and the complete list of movies and their genres in hand, we're going to take the weighted average of every movie based on the input profile and recommend the top twenty movies that most satisfy it.

In [38]:

```
#Multiply the genres by the weights and then take the weighted average
recommendationTable_df = ((genreTable*userProfile).sum(axis=1))/(userProfile.sum())
recommendationTable_df.head()
```

Out[38]:

```
movieId
1    0.594406
2    0.293706
3    0.188811
4    0.328671
5    0.188811
dtype: float64
```

In [39]:

```
#Sort our recommendations in descending order
recommendationTable_df = recommendationTable_df.sort_values(ascending=False)
#Just a peek at the values
recommendationTable_df.head()
```

Out[39]:

```
movieId
5018      0.748252
26093     0.734266
27344     0.720280
148775    0.685315
6902      0.678322
dtype: float64
```

Now here's the recommendation table!

In [40]:

```
#The final recommendation table
movies_df.loc[movies_df['movieId'].isin(recommendationTable_df.head(20).keys())]
```

Out[40]:

	movieId	title	genres	year
664	673	Space Jam	[Adventure, Animation, Children, Comedy, Fanta...	1996
1824	1907	Mulan	[Adventure, Animation, Children, Comedy, Drama...	1998
2902	2987	Who Framed Roger Rabbit?	[Adventure, Animation, Children, Comedy, Crime...	1988
4923	5018	Motorama	[Adventure, Comedy, Crime, Drama, Fantasy, Mys...	1991
6793	6902	Interstate 60	[Adventure, Comedy, Drama, Fantasy, Mystery, S...	2002
8605	26093	Wonderful World of the Brothers Grimm, The	[Adventure, Animation, Children, Comedy, Drama...	1962
8783	26340	Twelve Tasks of Asterix, The (Les douze travau...	[Action, Adventure, Animation, Children, Comed...	1976
9296	27344	Revolutionary Girl Utena: Adolescence of Utena...	[Action, Adventure, Animation, Comedy, Drama, ...	1999
9825	32031	Robots	[Adventure, Animation, Children, Comedy, Fanta...	2005
11716	51632	Atlantis: Milo's Return	[Action, Adventure, Animation, Children, Comed...	2003
11751	51939	TMNT (Teenage Mutant Ninja Turtles)	[Action, Adventure, Animation, Children, Comed...	2007
13250	64645	The Wrecking Crew	[Action, Adventure, Comedy, Crime, Drama, Thri...	1968
16055	81132	Rubber	[Action, Adventure, Comedy, Crime, Drama, Film...	2010
18312	91335	Gruffalo, The	[Adventure, Animation, Children, Comedy, Drama]	2009
22778	108540	Ernest & Célestine (Ernest et Célestine)	[Adventure, Animation, Children, Comedy, Drama...	2012
22881	108932	The Lego Movie	[Action, Adventure, Animation, Children, Comed...	2014
25218	117646	Dragonheart 2: A New Beginning	[Action, Adventure, Comedy, Drama, Fantasy, Th...	2000
26442	122787	The 39 Steps	[Action, Adventure, Comedy, Crime, Drama, Thri...	1959
32854	146305	Princes and Princesses	[Animation, Children, Comedy, Drama, Fantasy, ...	2000
33509	148775	Wizards of Waverly Place: The Movie	[Adventure, Children, Comedy, Drama, Fantasy, ...	2009

Advantages and Disadvantages of Content-Based Filtering

Advantages

- Learns user's preferences
- Highly personalized for the user

Disadvantages

- Doesn't take into account what others think of the item, so low quality item recommendations might happen
- Extracting data is not always intuitive
- Determining what characteristics of the item the user dislikes or likes is not always obvious

COLLABORATIVE FILTERING

Recommendation systems are a collection of algorithms used to recommend items to users based on information taken from the user. These systems have become ubiquitous can be commonly seen in online stores, movies databases and job finders.

In [46]:

```
movies_df=pd.read_csv('F:/dataset/ml-latest/movies.csv')
ratings_df = pd.read_csv('F:/dataset/ml-latest/ratings.csv')
movies_df.head()
```

Out[46]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

Let's remove the year from the title column by using pandas' replace function and store in a new year column.

In [47]:

```
#Using regular expressions to find a year stored between parentheses
#We specify the parantheses so we don't conflict with movies that have years in their titles
movies_df['year'] = movies_df.title.str.extract('(\(\d\d\d\d\))',expand=False)
#Removing the parentheses
movies_df['year'] = movies_df.year.str.extract('(\d\d\d\d)',expand=False)
#Removing the years from the 'title' column
movies_df['title'] = movies_df.title.str.replace('(\(\d\d\d\d\))', '')
#Applying the strip function to get rid of any ending whitespace characters that may have appeared
movies_df['title'] = movies_df['title'].apply(lambda x: x.strip())
```

In [48]:

```
movies_df.head()
```

Out[48]:

	movieId	title	genres	year
0	1	Toy Story	Adventure|Animation|Children|Comedy|Fantasy	1995
1	2	Jumanji	Adventure|Children|Fantasy	1995
2	3	Grumpier Old Men	Comedy|Romance	1995
3	4	Waiting to Exhale	Comedy|Drama|Romance	1995
4	5	Father of the Bride Part II	Comedy	1995

In [49]:

```
movies_df = movies_df.drop('genres', 1)
```

In [50]:

```
movies_df.head()
```

Out[50]:

	movieId	title	year
0	1	Toy Story	1995
1	2	Jumanji	1995
2	3	Grumpier Old Men	1995
3	4	Waiting to Exhale	1995
4	5	Father of the Bride Part II	1995

In [45]:

```
ratings_df.head()
```

Out[45]:

	userId	movieId	rating
0	1	169	2.5
1	1	2471	3.0
2	1	48516	5.0
3	2	2571	3.5
4	2	109487	4.0

Next, let's look at the ratings dataframe.

In [51]:

```
ratings_df.head()
```

Out[51]:

	userId	movieId	rating	timestamp
0	1	169	2.5	1204927694
1	1	2471	3.0	1204927438
2	1	48516	5.0	1204927435
3	2	2571	3.5	1436165433
4	2	109487	4.0	1436165496

Every row in the ratings dataframe has a user id associated with at least one movie, a rating and a timestamp showing when they reviewed it. We won't be needing the timestamp column, so let's drop it to save on memory.

In [52]:

```
#Drop removes a specified row or column from a dataframe
ratings_df = ratings_df.drop('timestamp', 1)
```

Here's how the final ratings Dataframe looks like:

In [53]:

```
ratings_df.head()
```

Out[53]:

	userId	movieId	rating
0	1	169	2.5
1	1	2471	3.0
2	1	48516	5.0
3	2	2571	3.5
4	2	109487	4.0

Now, time to start our work on recommendation systems.

The first technique we're going to take a look at is called **Collaborative Filtering**, which is also known as **User-User Filtering**. As hinted by its alternate name, this technique uses other users to recommend items to the input user. It attempts to find users that have similar preferences and opinions as the input and then recommends items that they have liked to the input. There are several methods of finding similar users (Even some making use of Machine Learning), and the one we will be using here is going to be based on the **Pearson Correlation Function**.

The process for creating a User Based recommendation system is as follows:

- Select a user with the movies the user has watched

- Based on his rating to movies, find the top X neighbours
- Get the watched movie record of the user for each neighbour.
- Calculate a similarity score using some formula
- Recommend the items with the highest score

Let's begin by creating an input user to recommend movies to:

Notice: To add more movies, simply increase the amount of elements in the userInput. Feel free to add more in! Just be sure to write it in with capital letters and if a movie starts with a "The", like "The Matrix" then write it in like this: 'Matrix, The' .

In [54]:

```
userInput = [
            {'title':'Breakfast Club, The', 'rating':5},
            {'title':'Toy Story', 'rating':3.5},
            {'title':'Jumanji', 'rating':2},
            {'title':"Pulp Fiction", 'rating':5},
            {'title':'Akira', 'rating':4.5}
        ]
inputMovies = pd.DataFrame(userInput)
inputMovies
```

Out[54]:

	title	rating
0	Breakfast Club, The	5.0
1	Toy Story	3.5
2	Jumanji	2.0
3	Pulp Fiction	5.0
4	Akira	4.5

Add movieId to input user

With the input complete, let's extract the input movies's ID's from the movies dataframe and add them into it.

We can achieve this by first filtering out the rows that contain the input movies' title and then merging this subset with the input dataframe. We also drop unnecessary columns for the input to save memory space.

In [55]:

```
#Filtering out the movies by title
inputId = movies_df[movies_df['title'].isin(inputMovies['title'].tolist())]
#Then merging it so we can get the movieId. It's implicitly merging it by title.
inputMovies = pd.merge(inputId, inputMovies)
#Dropping information we won't use from the input dataframe
inputMovies = inputMovies.drop('year', 1)
#Final input dataframe
#If a movie you added in above isn't here, then it might not be in the original
#dataframe or it might spelled differently, please check capitalisation.
inputMovies
```

Out[55]:

	movieId	title	rating
0	1	Toy Story	3.5
1	2	Jumanji	2.0
2	296	Pulp Fiction	5.0
3	1274	Akira	4.5
4	1968	Breakfast Club, The	5.0

Now with the movie ID's in our input, we can now get the subset of users that have watched and reviewed the movies in our input.

In [56]:

```
#Filtering out users that have watched movies that the input has watched and storing it
userSubset = ratings_df[ratings_df['movieId'].isin(inputMovies['movieId'].tolist())]
userSubset.head()
```

Out[56]:

	userId	movieId	rating
19	4	296	4.0
441	12	1968	3.0
479	13	2	2.0
531	13	1274	5.0
681	14	296	2.0

We now group up the rows by user ID.

In [57]:

```
#Groupby creates several sub dataframes where they all have the same value in the column specified a
userSubsetGroup = userSubset.groupby(['userId'])
```

lets look at one of the users, e.g. the one with userID=1130

In [58]:

```
userSubsetGroup.get_group(1130)
```

Out[58]:

	userId	movieId	rating
104167	1130	1	0.5
104168	1130	2	4.0
104214	1130	296	4.0
104363	1130	1274	4.5
104443	1130	1968	4.5

Let's also sort these groups so the users that share the most movies in common with the input have higher priority. This provides a richer recommendation since we won't go through every single user.

In [59]:

```
#Sorting it so users with movie most in common with the input will have priority
userSubsetGroup = sorted(userSubsetGroup,  key=lambda x: len(x[1]), reverse=True)
```

Now lets look at the first user

In [60]:

```
userSubsetGroup[0:3]
```

Out[60]:

```
[(75,
        userId  movieId  rating
  7507      75        1     5.0
  7508      75        2     3.5
  7540      75      296     5.0
  7633      75     1274     4.5
  7673      75     1968     5.0),
 (106,
        userId  movieId  rating
  9083     106        1     2.5
  9084     106        2     3.0
  9115     106      296     3.5
  9198     106     1274     3.0
  9238     106     1968     3.5),
 (686,
         userId  movieId  rating
  61336     686        1     4.0
  61337     686        2     3.0
  61377     686      296     4.0
  61478     686     1274     4.0
  61569     686     1968     5.0)]
```

Similarity of users to input user

Next, we are going to compare all users (not really all !!!) to our specified user and find the one that is most similar.
we're going to find out how similar each user is to the input through the **Pearson Correlation Coefficient**. It is used to measure the strength of a linear association between two variables. The formula for finding this coefficient between sets X and Y with N values can be seen in the image below.

Why Pearson Correlation?

Pearson correlation is invariant to scaling, i.e. multiplying all elements by a nonzero constant or adding any constant to all elements. For example, if you have two vectors X and Y,then, pearson(X, Y) == pearson(X, 2 * Y + 3). This is a pretty important property in recommendation systems because for example two users might rate two series of items totally different in terms of absolute rates, but they would be similar users (i.e. with similar ideas) with similar rates in various scales .

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

The values given by the formula vary from r = -1 to r = 1, where 1 forms a direct correlation between the two entities (it means a perfect positive correlation) and -1 forms a perfect negative correlation.

In our case, a 1 means that the two users have similar tastes while a -1 means the opposite.

In [61]:

```
userSubsetGroup = userSubsetGroup[0:100]
```

Now, we calculate the Pearson Correlation between input user and subset group, and store it in a dictionary, where the key is the user Id and the value is the coefficient

In [62]:

```
#Store the Pearson Correlation in a dictionary, where the key is the user Id and the value is the co
pearsonCorrelationDict = {}

#For every user group in our subset
for name, group in userSubsetGroup:
    #Let's start by sorting the input and current user group so the values aren't mixed up later on
    group = group.sort_values(by='movieId')
    inputMovies = inputMovies.sort_values(by='movieId')
    #Get the N for the formula
    nRatings = len(group)
    #Get the review scores for the movies that they both have in common
    temp_df = inputMovies[inputMovies['movieId'].isin(group['movieId'].tolist())]
    #And then store them in a temporary buffer variable in a list format to facilitate future calcul
    tempRatingList = temp_df['rating'].tolist()
    #Let's also put the current user group reviews in a list format
    tempGroupList = group['rating'].tolist()
    #Now let's calculate the pearson correlation between two users, so called, x and y
    Sxx = sum([i**2 for i in tempRatingList]) - pow(sum(tempRatingList),2)/float(nRatings)
    Syy = sum([i**2 for i in tempGroupList]) - pow(sum(tempGroupList),2)/float(nRatings)
    Sxy = sum( i*j for i, j in zip(tempRatingList, tempGroupList)) - sum(tempRatingList)*sum(tempGr

    #If the denominator is different than zero, then divide, else, 0 correlation.
    if Sxx != 0 and Syy != 0:
        pearsonCorrelationDict[name] = Sxy/sqrt(Sxx*Syy)
    else:
        pearsonCorrelationDict[name] = 0
```

In [63]:

```
pearsonCorrelationDict.items()
```

Out[63]:

```
dict_items([(75, 0.8272781516947562), (106, 0.5860090386731182), (686, 0.8320502943378437), (815, 0.5765566601970551), (1040, 0.9434563530497265), (1130, 0.2891574659831201), (1502, 0.8770580193070299), (1599, 0.4385290096535153), (1625, 0.7161148740394432), (1950, 0.179028718509858), (2065, 0.4385290096535153), (2128, 0.5860090386731196), (2432, 0.1386750490563073), (2791, 0.8770580193070299), (2839, 0.8204126541423674), (2948, -0.11720180773462392), (3025, 0.45124262819713973), (3040, 0.8951435925492929), (3186, 0.6784622064861935), (3271, 0.26989594817970664), (3429, 0.0), (3734, -0.15041420939904673), (4099, 0.05860090386731196), (4208, 0.29417420270727607), (4282, -0.4385290096535115), (4292, 0.6564386345361464), (4415, -0.11183835382312353), (4586, -0.9024852563942795), (4725, -0.08006407690254357), (4818, 0.4885967564883424), (5104, 0.7674257668936507), (5165, -0.4385290096535153), (5547, 0.17200522903844556), (6082, -0.04728779924109591), (6207, 0.9615384615384616), (6366, 0.6577935144802716), (6482, 0.0), (6530, -0.3516054232038709), (7235, 0.6981407669689391), (7403, 0.11720180773462363), (7641, 0.7161148740394331), (7996, 0.626600514784504), (8008, -0.22562131409856986), (8086, 0.6933752452815365), (8245, 0.0), (8572, 0.8600261451922278), (8675, 0.5370861555295773), (9101, -0.08600261451922278), (9358, 0.6921787383584485), (9663, 0.193972725041952), (9994, 0.5030272728659587), (10248, -0.2480694691784693), (10315, 0.537086155529574), (10368, 0.4688072309384945), (10607, 0.4160251471689218 6), (10707, 0.9615384615384616), (10863, 0.6020183016345595), (11314, 0.8204126541423654), (11399, 0.517260600111872), (11769, 0.9376144618769914), (11827, 0.4902903378454601), (12069, 0.0), (12120, 0.9292940047327363), (12211, 0.8600261451922278), (12325, 0.9616783115081544), (12916, 0.5860090386731196), (12921, 0.6611073566849309), (13053, 0.9607689228305227), (13142, 0.6016568375961863), (13260, 0.7844645405527362), (13366, 0.8951435925492911), (13768, 0.8770580193070289), (13888, 0.2508726030021272), (13923, 0.3516054232038718), (13934, 0.17200522903844556), (14529, 0.7417901772340937), (14551, 0.537086155529574), (14588, 0.21926450482675766), (14984, 0.716114874039432), (15137, 0.5860090386731196), (15157, 0.9035841064985974), (15466, 0.7205766921228921), (15670, 0.516015687115336), (15834, 0.22562131409856986), (16292, 0.6577935144802716), (16456, 0.7161148740394331), (16506, 0.5481612620668942), (17246, 0.48038446141526137), (17438, 0.7093169886164387), (17501, 0.8168748513121271), (17502, 0.8272781516947562), (17666, 0.7689238340176859), (17735, 0.7042381820123422), (17742, 0.3922322702763681), (17757, 0.64657575013984), (17854, 0.537086155529574), (17897, 0.8770580193070289), (17944, 0.2713848825944774), (18301, 0.2983811975164301 6), (18509, 0.1322214713369862)])
```

In [64]:

```
pearsonDF = pd.DataFrame.from_dict(pearsonCorrelationDict, orient='index')
pearsonDF.columns = ['similarityIndex']
pearsonDF['userId'] = pearsonDF.index
pearsonDF.index = range(len(pearsonDF))
pearsonDF.head()
```

Out[64]:

	similarityIndex	userId
0	0.827278	75
1	0.586009	106
2	0.832050	686
3	0.576557	815
4	0.943456	1040

Now let's get the top 50 users that are most similar to the input.

In [65]:

```
topUsers=pearsonDF.sort_values(by='similarityIndex', ascending=False)[0:50]
topUsers.head()
```

Out[65]:

	similarityIndex	userId
64	0.961678	12325
34	0.961538	6207
55	0.961538	10707
67	0.960769	13053
4	0.943456	1040

Now, let's start recommending movies to the input user.

Rating of selected users to all movies

We're going to do this by taking the weighted average of the ratings of the movies using the Pearson Correlation as the weight. But to do this, we first need to get the movies watched by the users in our **pearsonDF** from the ratings dataframe and then store their correlation in a new column called _similarityIndex". This is achieved below by merging of these two tables.

In [66]:

```
topUsersRating=topUsers.merge(ratings_df, left_on='userId', right_on='userId', how='inner')
topUsersRating.head()
```

Out[66]:

	similarityIndex	userId	movieId	rating
0	0.961678	12325	1	3.5
1	0.961678	12325	2	1.5
2	0.961678	12325	3	3.0
3	0.961678	12325	5	0.5
4	0.961678	12325	6	2.5

Now all we need to do is simply multiply the movie rating by its weight (The similarity index), then sum up the new ratings and divide it by the sum of the weights.

We can easily do this by simply multiplying two columns, then grouping up the dataframe by movieId and then dividing two columns:

It shows the idea of all similar users to candidate movies for the input user:

In [68]:

```
#Multiplies the similarity by the user's ratings
topUsersRating['weightedRating'] = topUsersRating['similarityIndex']*topUsersRating['rating']
topUsersRating.head()
```

Out[68]:

	similarityIndex	userId	movieId	rating	weightedRating
0	0.961678	12325	1	3.5	3.365874
1	0.961678	12325	2	1.5	1.442517
2	0.961678	12325	3	3.0	2.885035
3	0.961678	12325	5	0.5	0.480839
4	0.961678	12325	6	2.5	2.404196

In [69]:

```
#Applies a sum to the topUsers after grouping it up by userId
tempTopUsersRating = topUsersRating.groupby('movieId').sum()[['similarityIndex','weightedRating']]
tempTopUsersRating.columns = ['sum_similarityIndex','sum_weightedRating']
tempTopUsersRating.head()
```

Out[69]:

movieId	sum_similarityIndex	sum_weightedRating
1	38.376281	140.800834
2	38.376281	96.656745
3	10.253981	27.254477
4	0.929294	2.787882
5	11.723262	27.151751

In [70]:

```
#Creates an empty dataframe
recommendation_df = pd.DataFrame()
#Now we take the weighted average
recommendation_df['weighted average recommendation score'] = tempTopUsersRating['sum_weightedRating'
recommendation_df['movieId'] = tempTopUsersRating.index
recommendation_df.head()
```

Out[70]:

movieId	weighted average recommendation score	movieId
1	3.668955	1
2	2.518658	2
3	2.657941	3
4	3.000000	4
5	2.316058	5

Now let's sort it and see the top 20 movies that the algorithm recommended!

In [71]:

```
recommendation_df = recommendation_df.sort_values(by='weighted average recommendation score', ascend
recommendation_df.head(10)
```

Out[71]:

movieId	weighted average recommendation score	movieId
5073	5.0	5073
3329	5.0	3329
2284	5.0	2284
26801	5.0	26801
6776	5.0	6776
6672	5.0	6672
3759	5.0	3759
3769	5.0	3769
3775	5.0	3775
90531	5.0	90531

In [72]:

```
movies_df.loc[movies_df['movieId'].isin(recommendation_df.head(10)['movieId'].tolist())]
```

Out[72]:

	movieId	title	year
2200	2284	Bandit Queen	1994
3243	3329	Year My Voice Broke, The	1987
3669	3759	Fun and Fancy Free	1947
3679	3769	Thunderbolt and Lightfoot	1974
3685	3775	Make Mine Music	1946
4978	5073	Son's Room, The (Stanza del figlio, La)	2001
6563	6672	War Photographer	2001
6667	6776	Lagaan: Once Upon a Time in India	2001
9064	26801	Dragon Inn (Sun lung moon hak chan)	1992
18106	90531	Shame	2011

Advantages and Disadvantages of Collaborative Filtering

Advantages

- Takes other user's ratings into consideration

- Doesn't need to study or extract information from the recommended item
- Adapts to the user's interests which might change over time

Disadvantages

- Approximation function can be slow
- There might be a low of amount of users to approximate
- Privacy issues when trying to learn the user's preferences

In [ ]: