ibm_data_science_specialization

January 6, 2021

0.0.1 ANOVA: Analysis of Variance

- The Analysis of Variance (ANOVA) is a statistical method used to test whether there are significant differences between the means of two or more groups. ANOVA returns two parameters:
- **F-test score:** ANOVA assumes the means of all groups are the same, calculates how much the actual means deviate from the assumption, and reports it as the F-test score. A larger score means there is a larger difference between the means.
- **P-value:** P-value tells how statistically significant is our calculated score value.
- If our price variable is strongly correlated with the variable we are analyzing, expect ANOVA to return a sizeable F-test score and a small p-value.

[1]:
```
from scipy import stats
import pandas as pd
```

[2]:
```
df = pd.read_csv('data/car_dataset.csv')
```

[3]:
```
grouped = df[['drive-wheels', 'price']].groupby('drive-wheels')
```

[4]:
```
f_val, p_val = stats.f_oneway(
    grouped.get_group('fwd')['price'],
    grouped.get_group('rwd')['price'],
    grouped.get_group('4wd')['price']
)

print('ANOVA results: F=', f_val, ', P=', p_val)
```

```
ANOVA results: F= 67.95406500780399 , P= 3.3945443577151245e-23
```

This is a great result, with a large F test score showing a strong correlation and a P value of almost 0 implying almost certain statistical significance. But does this mean all three tested groups are all this highly correlated?

[5]:
```
f_val, p_val = stats.f_oneway(
    grouped.get_group('fwd')['price'],
    grouped.get_group('rwd')['price']
)
```

```
print('ANOVA results: F=', f_val, ', P=', p_val)
```

```
ANOVA results: F= 130.5533160959111 , P= 2.2355306355677845e-23
```

[6]:
```
f_val, p_val = stats.f_oneway(
    grouped.get_group('4wd')['price'],
    grouped.get_group('rwd')['price']
)

print('ANOVA results: F=', f_val, ', P=', p_val)
```

```
ANOVA results: F= 8.580681368924756 , P= 0.004411492211225333
```

[7]:
```
f_val, p_val = stats.f_oneway(
    grouped.get_group('4wd')['price'],
    grouped.get_group('fwd')['price']
)

print('ANOVA results: F=', f_val, ', P=', p_val)
```

```
ANOVA results: F= 0.665465750252303 , P= 0.41620116697845666
```

There is a not statistical significance between 4wd and fwd, even so, that metric can be useful for machine learning algorithms.

0.0.2 Residual Plot

A good way to visualize the variance of the data is to use a residual plot.

What is a residual?

The difference between the observed value (y) and the predicted value (Yhat) is called the residual (e). When we look at a regression plot, the residual is the distance from the data point to the fitted regression line.

So what is a residual plot?

A residual plot is a graph that shows the residuals on the vertical y-axis and the independent variable on the horizontal x-axis.

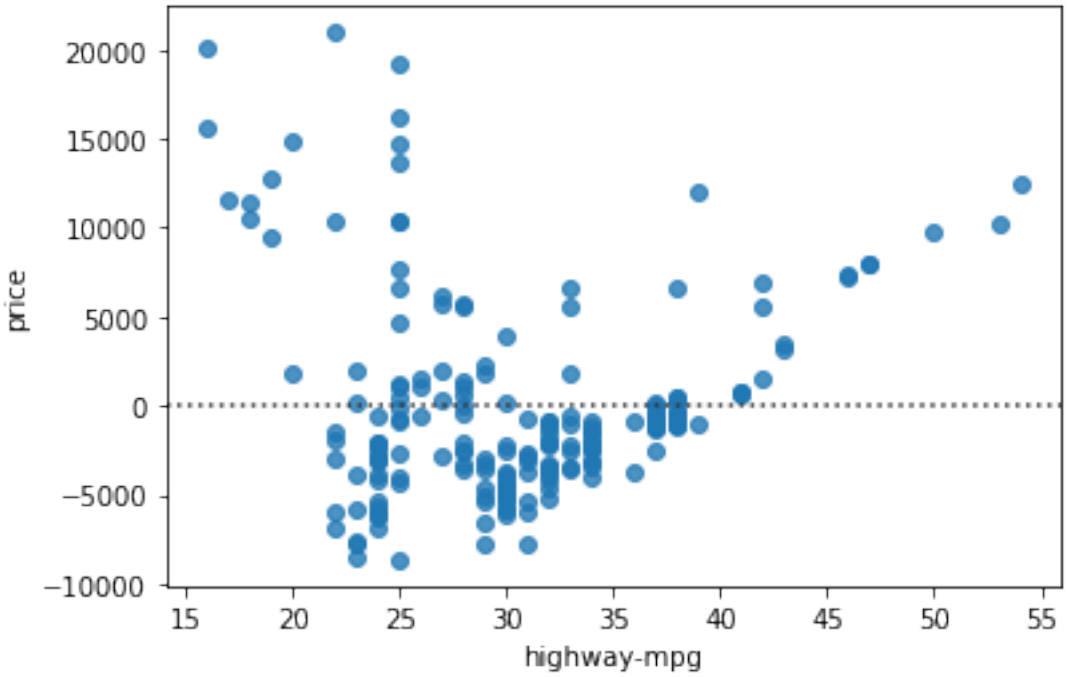
What do we pay attention to when looking at a residual plot?

We look at the spread of the residuals: If the points in a residual plot are randomly spread out around the x-axis, then a linear model is appropriate for the data. Why is that? Randomly spread out residuals means that the variance is constant, and thus the linear model is a good fit for this data.

[1]:
```
import pandas as pd
import seaborn as sns
```

[2]:
```
df = pd.read_csv('data/car_dataset.csv')
```

[3]:
```
sns.residplot(x='highway-mpg', y='price', data=df);
```



What is this plot telling us?

We can see from this residual plot that the residuals are not randomly spread around the x-axis, which leads us to believe that maybe a non-linear model is more appropriate for this data.

0.0.3 Polynomial Regression

Polynomial regression is a particular case of the general linear regression model or multiple linear regression models. We get non-linear relationships by squaring or setting higher-order terms of the predictor variables.

There are different orders of polynomial regression:

Quadratic - 2nd order

$$Yhat = a + b_1 X + b_2 X^2$$

Cubic - 3rd order

$$Yhat = a + b_1 X + b_2 X^2 + b_3 X^3$$

Higher order:

$$Y = a + b_1 X + b_2 X^2 + b_3 X^3 ....$$

We saw earlier in `residplot` that a linear model did not provide the best fit while using highway-mpg as the predictor variable. Let's see if we can try fitting a polynomial model to the data instead.

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

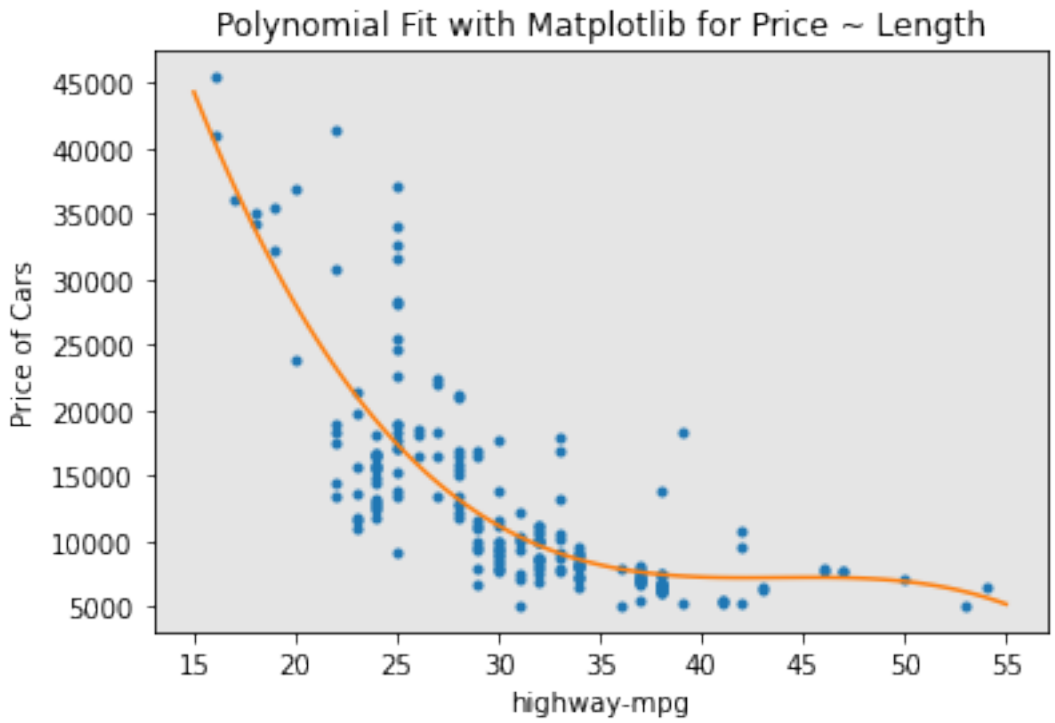
[2]:
```
df = pd.read_csv('data/car_dataset.csv')
```

[3]:
```
X = df['highway-mpg']
y = df['price']
```

[4]:
```
# here we use a polynomial of the 3rd order (cubic)
f = np.polyfit(X, y, 3)
p = np.poly1d(f)
print(p)
```

```
        3         2
-1.557 x + 204.8 x - 8965 x + 1.379e+05
```

[5]:
```
X_new = np.linspace(15, 55, 100)
y_new = p(X_new)

plt.plot(X, y, '.', X_new, y_new, '-')
plt.title('Polynomial Fit with Matplotlib for Price ~ Length')

ax = plt.gca()
ax.set_facecolor((0.898, 0.898, 0.898))
fig = plt.gcf()

plt.xlabel('highway-mpg')
plt.ylabel('Price of Cars')

plt.show()
plt.close()
```



We can already see from plotting that this polynomial model performs better than the linear model. This is because the generated polynomial function "hits" more of the data points.

Also, you can use `from sklearn.preprocessing import PolynomialFeatures` to apply.

[6]:
```
from sklearn.preprocessing import PolynomialFeatures
```

0.0.4 Area Plot

[1]:
```
%%capture
!pip3 install xlrd
```

[2]:
```
import matplotlib.pyplot as plt
import pandas as pd
```

[3]:
```
df_can = pd.read_excel(
    'data/canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2
)
```

[4]:
```
df_can.columns = list(map(lambda x: str(x), df_can.columns))
```

[5]:
```
drops = [
    'AREA',
    'REG',
    'DEV',
    'Type',
    'Coverage'
]
df_can.drop(columns=drops, inplace=True)
```

[6]:
```
columns = {
    'OdName': 'Country',
    'AreaName': 'Continent',
    'RegName': 'Region'
}

df_can.rename(columns=columns, inplace=True)
```

[7]:
```
df_can.set_index('Country', inplace=True)
```

[8]:
```
df_can['Total'] = df_can.sum(axis=1)
```

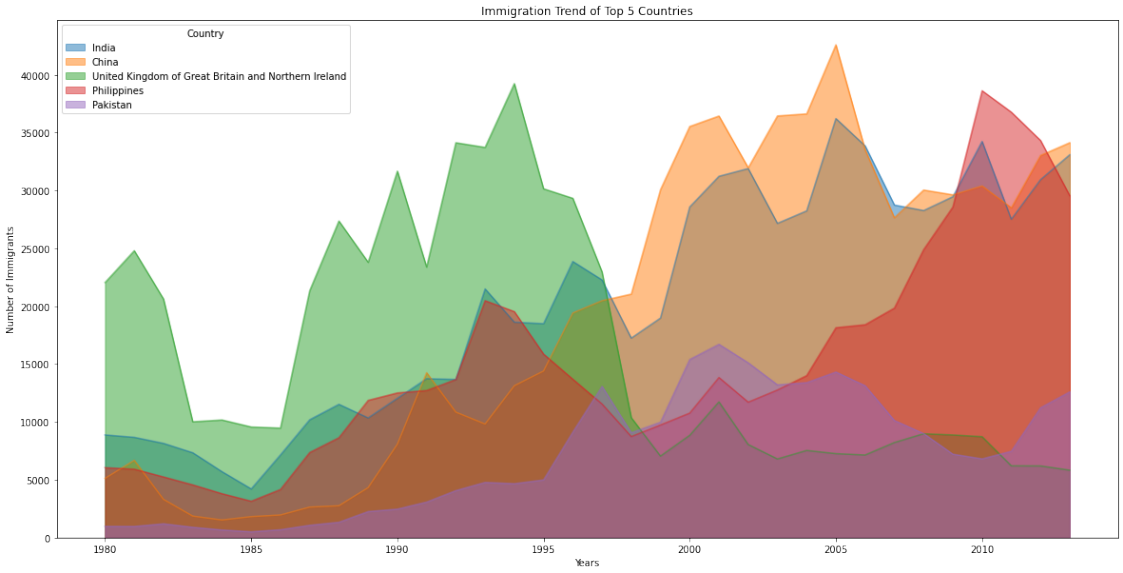
[9]:
```
years = list(map(str, range(1980, 2014)))
```

[10]:
```
df_can.sort_values('Total', ascending=False, axis=0, inplace=True)

df_top5 = df_can.head()
df_top5 = df_top5[years].transpose()

df_top5.head()
```

[10]:
```
Country  India  China  United Kingdom of Great Britain and Northern Ireland  \
1980      8880   5123                                                 22045
1981      8670   6682                                                 24796
1982      8147   3308                                                 20620
1983      7338   1863                                                 10015
1984      5704   1527                                                 10170

Country  Philippines  Pakistan
1980            6051       978
1981            5921       972
1982            5249      1201
1983            4562       900
1984            3801       668
```

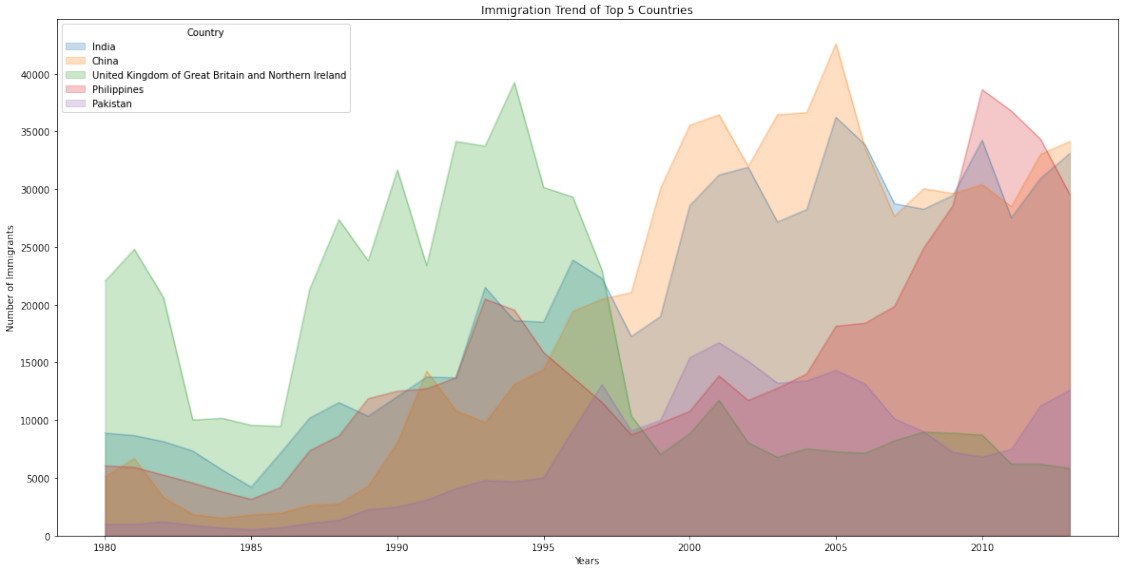
[11]:
```
df_top5.index = df_top5.index.map(int)
df_top5.plot(
    kind='area',
```

```
    stacked=False,
    figsize=(20, 10)
)

plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```



The unstacked plot has a default transparency (alpha value) at 0.5. We can modify this value by passing in the `alpha` parameter.

[12]:
```
df_top5.plot(
    kind='area',
    alpha=0.25,
    stacked=False,
    figsize=(20, 10),
)

plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```



0.0.5 Two types of plotting

There are two styles/options of ploting with `matplotlib`. Plotting using the Artist layer and plotting using the scripting layer.

Option 1: Scripting layer (procedural method) - using matplotlib.pyplot as `plt`

You can use `plt` i.e. `matplotlib.pyplot` and add more elements by calling different methods procedurally; for example, `plt.title(...)` to add title or `plt.xlabel(...)` to add label to the x-axis.

```
# option 1: this is what we have been using so far
df_top5.plot(kind='area', alpha=0.35, figsize=(20, 10))

plt.title('Immigration trend of top 5 countries')
plt.ylabel('Number of immigrants')
plt.xlabel('Years')
```

Option 2: Artist layer (Object oriented method) - using an `Axes` instance from Matplotlib (preferred)

You can use an `Axes` instance of your current plot and store it in a variable (eg. `ax`). You can add more elements by calling methods with a little change in syntax (by adding `*set_*` to the previous methods). For example, use `ax.set_title()` instead of `plt.title()` to add title, or `ax.set_xlabel()` instead of `plt.xlabel()` to add label to the x-axis.

This option sometimes is more transparent and flexible to use for advanced plots.

```
# option 2: preferred option with more flexibility
ax = df_top5.plot(kind='area', alpha=0.35, figsize=(20, 10))

ax.set_title('Immigration Trend of Top 5 Countries')
```

```
    ax.set_ylabel('Number of Immigrants')
    ax.set_xlabel('Years')
```

0.0.6 Bubble Plot

A `bubble plot` is a variation of the `scatter plot` that displays three dimensions of data (x, y, z). The datapoints are replaced with bubbles, and the size of the bubble is determined by the third variable 'z', also known as the weight. In `maplotlib`, we can pass in an array or scalar to the keyword `s` to `plot()`, that contains the weight of each point.

[1]:
```
%%capture
!pip3 install xlrd
```

[2]:
```
import matplotlib.pyplot as plt
import pandas as pd
```

[3]:
```
df_can = pd.read_excel(
    'data/canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2
)
```

[4]:
```
df_can.columns = list(map(lambda x: str(x), df_can.columns))
```

[5]:
```
drops = [
    'AREA',
    'REG',
    'DEV',
    'Type',
    'Coverage'
]
df_can.drop(columns=drops, inplace=True)
```

[6]:
```
columns = {
    'OdName': 'Country',
    'AreaName': 'Continent',
    'RegName': 'Region'
}

df_can.rename(columns=columns, inplace=True)
```

[7]:
```
df_can.set_index('Country', inplace=True)
```

[8]:
```
df_can['Total'] = df_can.sum(axis=1)
```

[9]:
```
years = list(map(str, range(1980, 2014)))
```

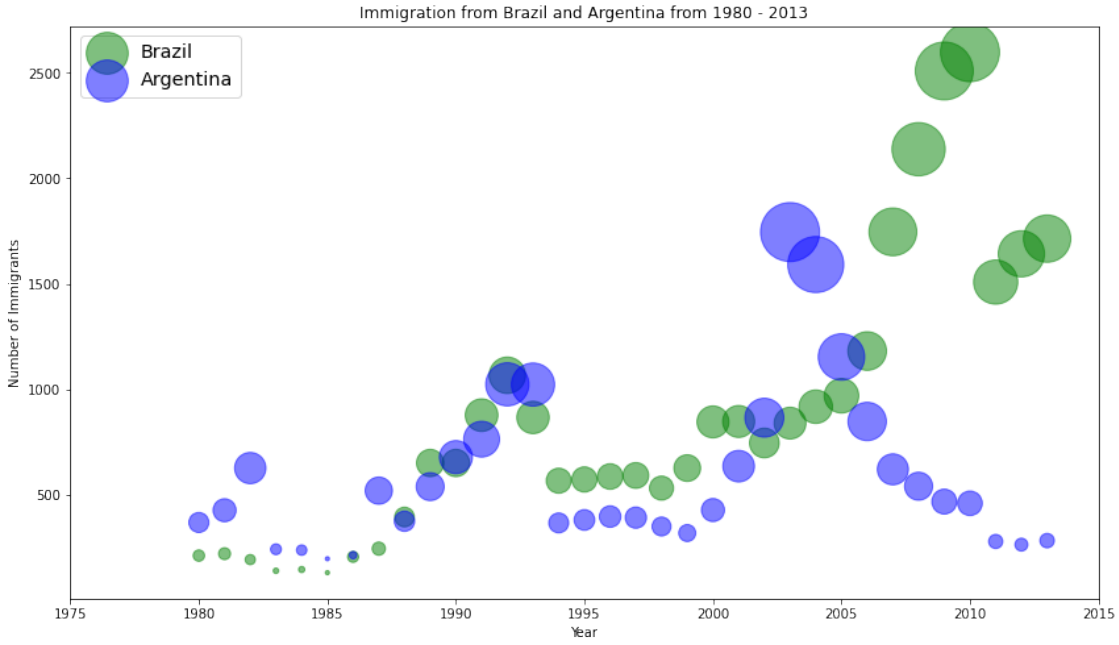
[10]:
```
df_can_t = df_can[years].transpose()
df_can_t.index = map(int, df_can_t.index)

df_can_t.index.name = 'Year'
df_can_t.reset_index(inplace=True)
```

[11]:
```
def normalize(data, col):
    return (data[col] - data[col].min()) / (data[col].max() - data[col].min())
```

[12]:
```
norm_brazil = normalize(df_can_t, 'Brazil')
norm_argentina = normalize(df_can_t, 'Argentina')
```

[13]:
```
ax0 = df_can_t.plot(
    kind='scatter',
    x='Year',
    y='Brazil',
    figsize=(14, 8),
    alpha=0.5,
    color='green',
    s=norm_brazil * 2000 + 10,
    xlim=(1975, 2015)
)

ax1 = df_can_t.plot(
    kind='scatter',
    x='Year',
    y='Argentina',
    alpha=0.5,
    color='blue',
    s=norm_argentina * 2000 + 10,
    ax = ax0
)

ax0.set_ylabel('Number of Immigrants')
ax0.set_title('Immigration from Brazil and Argentina from 1980 - 2013')
ax0.legend(['Brazil', 'Argentina'], loc='upper left', fontsize='x-large')
```

[13]:
```
<matplotlib.legend.Legend at 0x116f1da20>
```



The size of the bubble corresponds to the magnitude of immigrating population for that year, compared to the 1980 - 2013 data. The larger the bubble, the more immigrants in that year.

From the plot above, we can see a corresponding increase in immigration from Argentina during the 1998 - 2002 great depression. We can also observe a similar spike around 1985 to 1993. In fact, Argentina had suffered a great depression from 1974 - 1990, just before the onset of 1998 - 2002 great depression.

On a similar note, Brazil suffered the *Samba Effect* where the Brazilian real (currency) dropped nearly 35% in 1999. There was a fear of a South American financial crisis as many South American countries were heavily dependent on industrial exports from Brazil. The Brazilian government subsequently adopted an austerity program, and the economy slowly recovered over the years, culminating in a surge in 2010. The immigration data reflect these events.

0.0.7 Histogram

A histogram is a way of representing the *frequency* distribution of numeric dataset. The way it works is it partitions the x-axis into *bins*, assigns each data point in our dataset to a bin, and then counts the number of data points that have been assigned to each bin. So the y-axis is the frequency or the number of data points in each bin. Note that we can change the bin size and usually one needs to tweak it so that the distribution is displayed nicely.

[1]:
```
%%capture
!pip3 install xlrd
```

[2]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

```
import numpy as np
```

[3]:
```
df_can = pd.read_excel(
    'data/canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2
)
```

[4]:
```
df_can.columns = list(map(lambda x: str(x), df_can.columns))
```

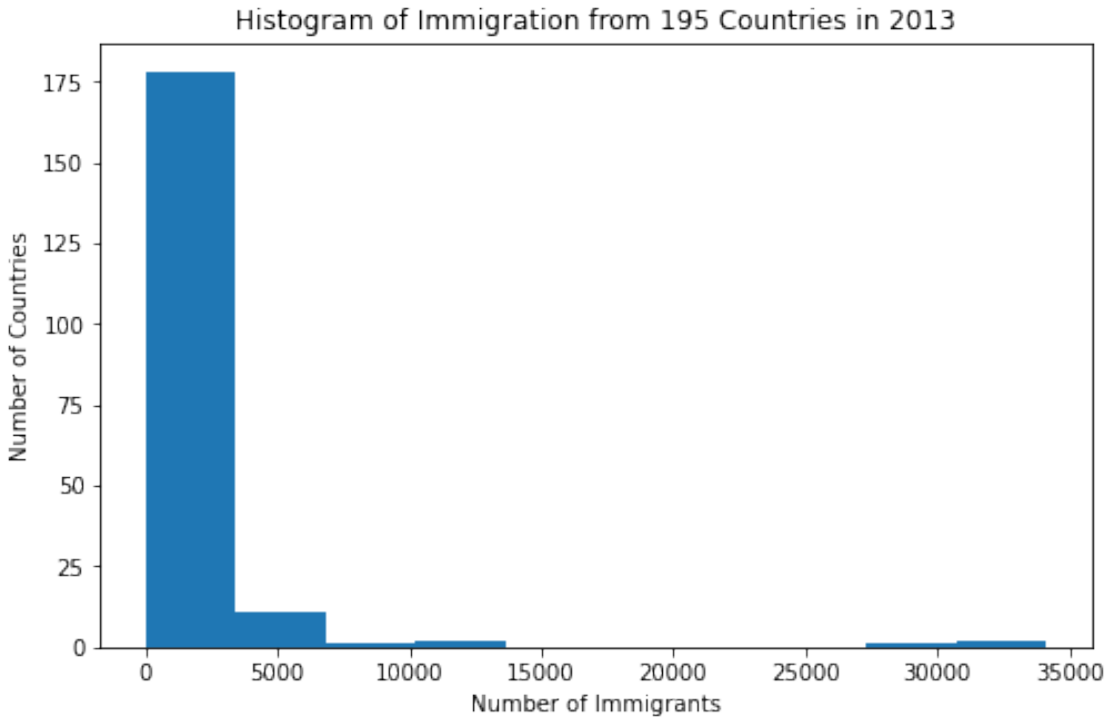
[5]:
```
drops = [
    'AREA',
    'REG',
    'DEV',
    'Type',
    'Coverage'
]
df_can.drop(columns=drops, inplace=True)
```

[6]:
```
columns = {
    'OdName': 'Country',
    'AreaName': 'Continent',
    'RegName': 'Region'
}

df_can.rename(columns=columns, inplace=True)
```

[7]:
```
df_can.set_index('Country', inplace=True)
```

[8]:
```
df_can['Total'] = df_can.sum(axis=1)
```

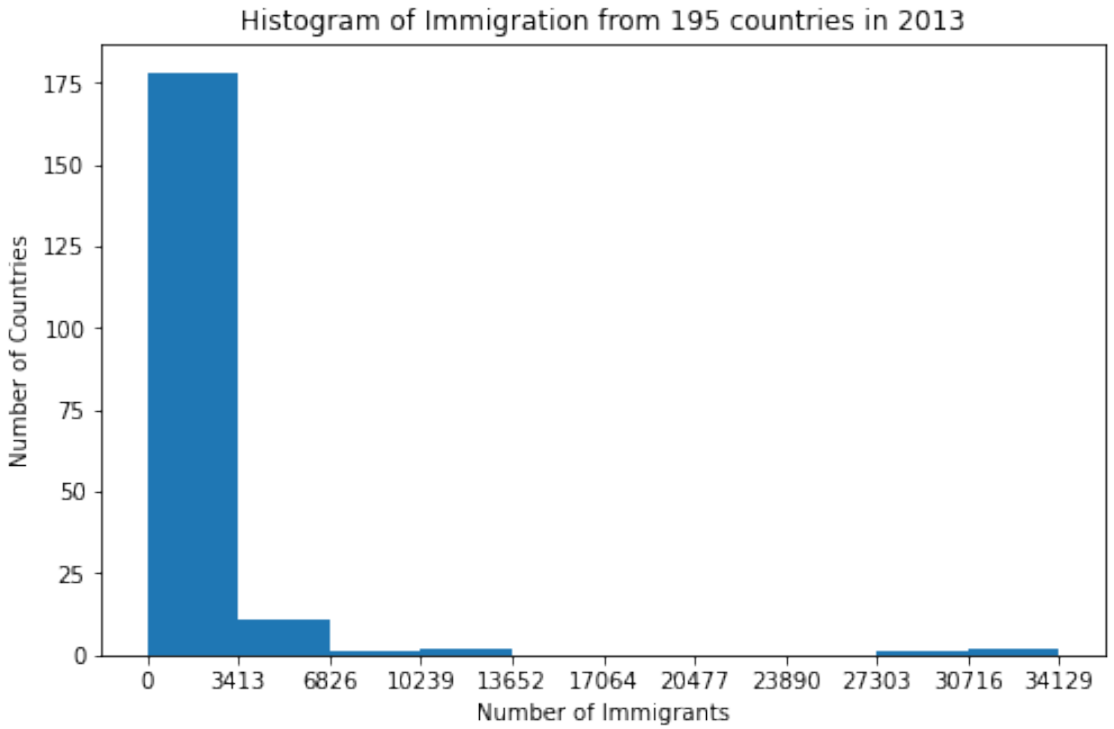
[9]:
```
df_can['2013'].plot(kind='hist', figsize=(8, 5))

plt.title('Histogram of Immigration from 195 Countries in 2013')
plt.ylabel('Number of Countries')
plt.xlabel('Number of Immigrants')

plt.show()
```



In the above plot, the x-axis represents the population range of immigrants in intervals of 3412.9. The y-axis represents the number of countries that contributed to the aforementioned population.

Notice that the x-axis labels do not match with the bin size. This can be fixed by passing in a `xticks` keyword that contains the list of the bin sizes, as follows:

[10]:
```
count, bin_edges = np.histogram(df_can['2013'])

df_can['2013'].plot(kind='hist', figsize=(8, 5), xticks=bin_edges)

plt.title('Histogram of Immigration from 195 countries in 2013')
plt.ylabel('Number of Countries')
plt.xlabel('Number of Immigrants')

plt.show()
```



0.0.8 Pie Chart

A `pie chart` is a circualr graphic that displays numeric proportions by dividing a circle (or pie) into proportional slices. You are most likely already familiar with pie charts as it is widely used in business and media. We can create pie charts in Matplotlib by passing in the `kind=pie` keyword.

[1]:
```
%%capture
!pip3 install xlrd
```

[2]:
```
import matplotlib.pyplot as plt
import pandas as pd
```

[3]:
```
df_can = pd.read_excel(
    'data/canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2
)
```

[4]:
```
df_can.columns = list(map(lambda x: str(x), df_can.columns))
```

[5]:
```
drops = [
    'AREA',
```

```
    'REG',
    'DEV',
    'Type',
    'Coverage'
]
df_can.drop(columns=drops, inplace=True)
```

[6]:
```
columns = {
    'OdName': 'Country',
    'AreaName': 'Continent',
    'RegName': 'Region'
}

df_can.rename(columns=columns, inplace=True)
```

[7]:
```
df_can.set_index('Country', inplace=True)
```

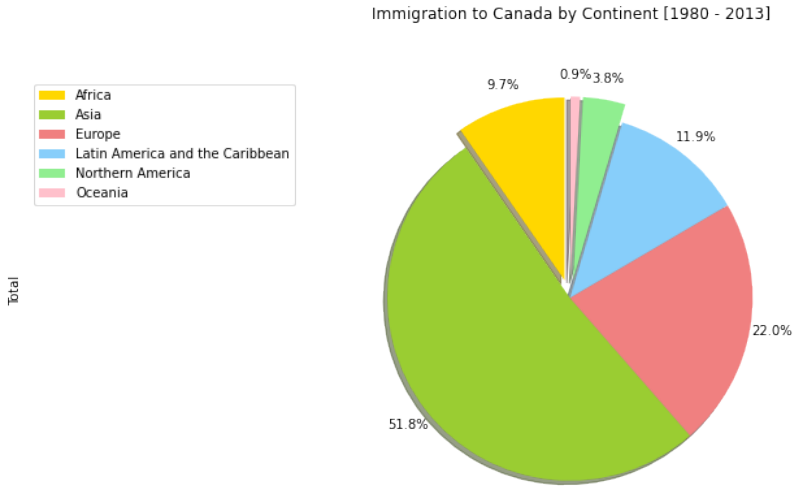
[8]:
```
df_can['Total'] = df_can.sum(axis=1)
```

[9]:
```
df_continents = df_can.groupby('Continent', axis=0).sum()
```

- Remove the text labels on the pie chart by passing in `legend` and add it as a seperate legend using `plt.legend()`.
- Push out the percentages to sit just outside the pie chart by passing in `pctdistance` parameter.
- Pass in a custom set of colors for continents by passing in `colors` parameter.
- **Explode** the pie chart to emphasize the lowest three continents (Africa, North America, and Latin America and Carribbean) by pasing in `explode` parameter.

[10]:
```
colors_list = [
    'gold',
    'yellowgreen',
    'lightcoral',
    'lightskyblue',
    'lightgreen',
    'pink'
]
explode_list = [0.1, 0, 0, 0, 0.1, 0.1]

df_continents['Total'].plot(
    kind='pie',
    figsize=(15, 6),
    autopct='%1.1f%%',
    startangle=90,
    shadow=True,
    labels=None,
    pctdistance=1.12,
```

```
    colors=colors_list,
    explode=explode_list
)

plt.title('Immigration to Canada by Continent [1980 - 2013]', y=1.12)
plt.axis('equal')

plt.legend(labels=df_continents.index, loc='upper left')
plt.show()
```



0.0.9 Subplots

Often times we might want to plot multiple plots within the same figure. For example, we might want to perform a side by side comparison of the box plot with the line plot of China and India's immigration.

To visualize multiple plots together, we can create a **`figure`** (overall canvas) and divide it into **`subplots`**, each containing a plot. With **subplots**, we usually work with the **artist layer** instead of the **scripting layer**.

Typical syntax is :

```
fig = plt.figure()
ax = fig.add_subplot(nrows, ncols, plot_number)
```

Where - `nrows` and `ncols` are used to notionally split the figure into (`nrows` * `ncols`) sub-axes,
- `plot_number` is used to identify the particular subplot that this function is to create within the notional grid. `plot_number` starts at 1, increments across rows first and has a maximum of `nrows` * `ncols` as shown below.

[1]:
```
%%capture
!pip3 install xlrd
```

[2]:
```
import matplotlib.pyplot as plt
import pandas as pd
```

[3]:
```
df_can = pd.read_excel(
    'data/canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2
)
```

[4]:
```
df_can.columns = list(map(lambda x: str(x), df_can.columns))
```

[5]:
```
drops = [
    'AREA',
    'REG',
    'DEV',
    'Type',
    'Coverage'
]
df_can.drop(columns=drops, inplace=True)
```

[6]:
```
columns = {
    'OdName': 'Country',
    'AreaName': 'Continent',
    'RegName': 'Region'
}

df_can.rename(columns=columns, inplace=True)
```

[7]:
```
df_can.set_index('Country', inplace=True)
```

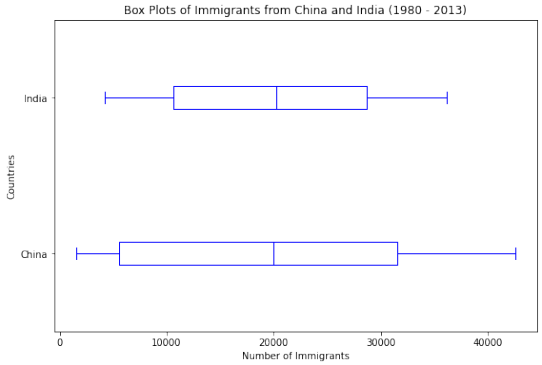
[8]:
```
df_can['Total'] = df_can.sum(axis=1)
```

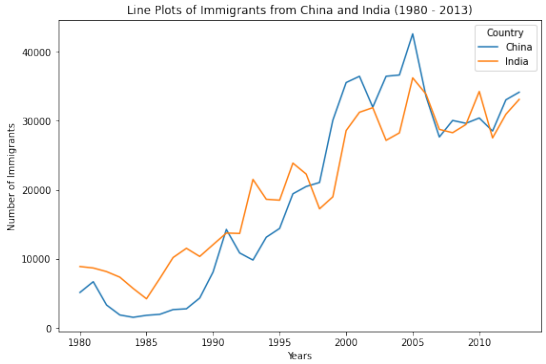
[9]:
```
years = list(map(str, range(1980, 2014)))
```

[10]:
```
df_ci = df_can.loc[['China', 'India'], years].transpose()
```

[11]:
```
fig = plt.figure()

ax0 = fig.add_subplot(1, 2, 1)
ax1 = fig.add_subplot(1, 2, 2)

df_ci.plot(kind='box', color='blue', vert=False, figsize=(20, 6), ax=ax0)
ax0.set_title('Box Plots of Immigrants from China and India (1980 - 2013)')
ax0.set_xlabel('Number of Immigrants')
ax0.set_ylabel('Countries')
```

```
df_ci.plot(kind='line', figsize=(20, 6), ax=ax1)
ax1.set_title ('Line Plots of Immigrants from China and India (1980 - 2013)')
ax1.set_ylabel('Number of Immigrants')
ax1.set_xlabel('Years')

plt.show()
```





Tip regarding subplot convention

In the case when `nrows`, `ncols`, and `plot_number` are all less than 10, a convenience exists such that the a 3 digit number can be given instead, where the hundreds represent `nrows`, the tens represent `ncols` and the units represent `plot_number`. For instance,

```
subplot(211) == subplot(2, 1, 1)
```

produces a subaxes in a figure which represents the top plot (i.e. the first) in a 2 rows by 1 column notional grid (no grid actually exists, but conceptually this is how the returned subplot has been positioned).

0.0.10 Waffle Chart

A `waffle chart` is an interesting visualization that is normally created to display progress toward goals. It is commonly an effective option when you are trying to add interesting visualization features to a visual that consists mainly of cells, such as an Excel dashboard.

[1]:
```
%%capture
!pip3 install xlrd
!pip3 install pywaffle
```

[2]:
```
from pywaffle import Waffle
import matplotlib.pyplot as plt
import pandas as pd
```

[3]:
```
df_can = pd.read_excel(
    'data/canada.xlsx',
    sheet_name='Canada by Citizenship',
```

```
    skiprows=range(20),
    skipfooter=2
)
```

[4]:
```
df_can.columns = list(map(lambda x: str(x), df_can.columns))
```

[5]:
```
drops = [
    'AREA',
    'REG',
    'DEV',
    'Type',
    'Coverage'
]
df_can.drop(columns=drops, inplace=True)
```

[6]:
```
columns = {
    'OdName': 'Country',
    'AreaName': 'Continent',
    'RegName': 'Region'
}

df_can.rename(columns=columns, inplace=True)
```

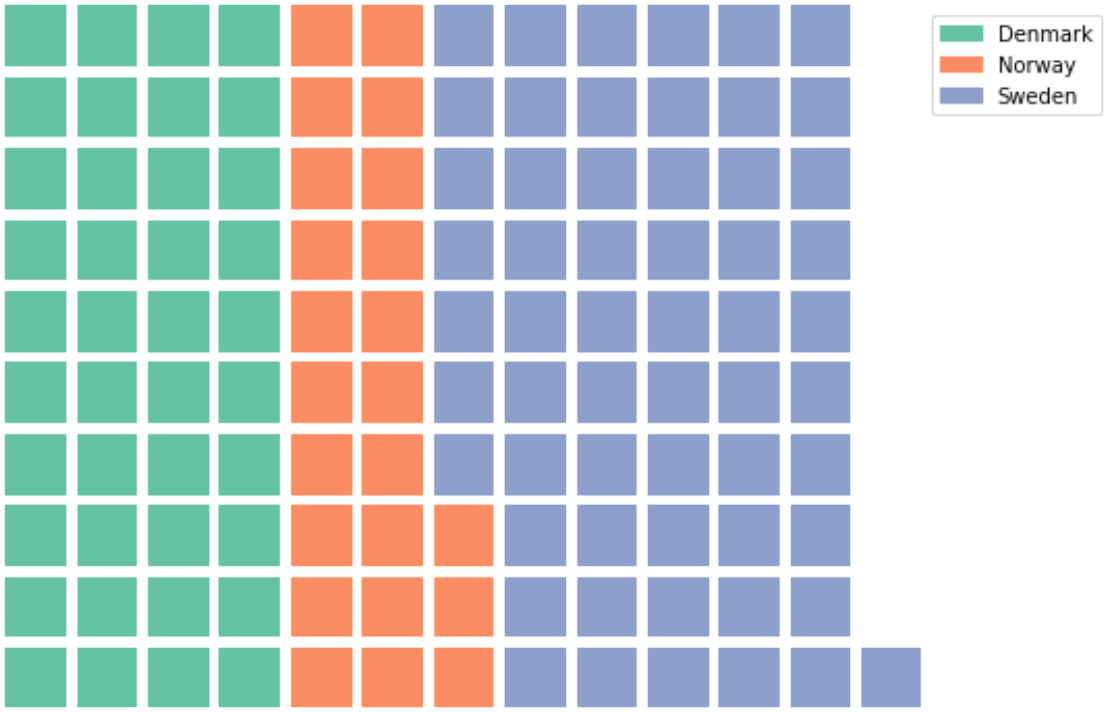
[7]:
```
df_can.set_index('Country', inplace=True)
```

[8]:
```
df_can['Total'] = df_can.sum(axis=1)
```

[9]:
```
df_dsn = df_can.loc[['Denmark', 'Norway', 'Sweden'], 'Total'].reset_index()
```

[10]:
```
totals = dict(zip(df_dsn.Country, df_dsn.Total / 100))
```

[11]:
```
fig = plt.figure(
    FigureClass=Waffle,
    rows=10,
    legend={'loc': 'upper left', 'bbox_to_anchor': (1, 1)},
    values=totals,
    figsize=(9, 5)
)
plt.show()
```



[12]:
```
fig = plt.figure(
    FigureClass=Waffle,
    rows=10,
    legend={'loc': 'upper left', 'bbox_to_anchor': (1, 1)},
    values=totals,
    icons='child',
    icon_legend=True,
    figsize=(9, 5)
)
plt.show()
```

In the charts as above, 1 block = 100 individuals.

[1]:
```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd
```

0.0.11 Ordinary Least Squares (OLS)

Coefficient and **Intercept**, are the parameters of the fit line. Given that it is a multiple linear regression, with 3 parameters, and knowing that the parameters are the intercept and coefficients of hyperplane, sklearn can estimate them from our data. Scikit-learn uses plain Ordinary Least Squares method to solve this problem.

OLS is a method for estimating the unknown parameters in a linear regression model. OLS chooses the parameters of a linear function of a set of explanatory variables by minimizing the sum of the squares of the differences between the target dependent variable and those predicted by the linear function. In other words, it tries to minimizes the sum of squared errors (SSE) or mean squared error (MSE) between the target variable (y) and our predicted output (yhat) over all samples in the dataset.

OLS can find the best parameters using of the following methods:

```
- Solving the model parameters analytically using closed-form equations
- Using an optimization algorithm (Gradient Descent, Stochastic Gradient Descent, Newton's Meth
```

[2]:
```
df = pd.read_csv('data/fuel_consumption_co2.csv')
```

[3]:
```
columns = [
    'ENGINESIZE',
    'CYLINDERS',
    'FUELCONSUMPTION_CITY',
    'FUELCONSUMPTION_HWY',
    'FUELCONSUMPTION_COMB',
    'CO2EMISSIONS'
]
cdf = df[columns]
```

[4]:
```
# train test split
msk = np.random.rand(len(df)) < 0.8
train = cdf[msk]
test = cdf[~msk]
```

[5]:
```
independents = [
    'ENGINESIZE',
    'CYLINDERS',
    'FUELCONSUMPTION_CITY',
    'FUELCONSUMPTION_HWY'
]
dependent = ['CO2EMISSIONS']
```

[6]:
```
regr = LinearRegression()
X_train = np.asanyarray(train[independents])
y_train = np.asanyarray(train[dependent])
```

[7]:
```
regr.fit(X_train, y_train)
print('Coefficients: ', regr.coef_)
```

```
Coefficients:  [[11.85159853  6.22788704  6.49985737  3.01636028]]
```

[8]:
```
y_hat = regr.predict(test[independents])
X_test = np.asanyarray(test[independents])
y_test = np.asanyarray(test[dependent])
```

[9]:
```
print('Residual sum of squares: %.2f' % np.mean((y_hat - y_test) ** 2))
print('Variance score: %.2f' % regr.score(X_test, y_test))
```

```
Residual sum of squares: 602.09
Variance score: 0.84
```

0.0.12 Gradient Descent

When the number of rows in your data set is less than 10,000, you can think of OLS as an option. However, for greater values, you should try other faster approaches. The second option is to use an

optimization algorithm to find the best parameters. That is, you can use a process of optimizing the values of the coefficients by iteratively minimizing the error of the model on your training data. For example, you can use gradient descent which starts optimization with random values for each coefficient, then calculates the errors and tries to minimize it through y's changing of the coefficients in multiple iterations. Gradient descent is a proper approach if you have a large data set. Please understand however, that there are other approaches to estimate the parameters of the multiple linear regression that you can explore on your own.

[10]:
```
# normalize independent variables
X_norm = X_train.copy()

min_x = np.min(X_train)
max_x = np.max(X_train)

X_norm = (X_train - min_x) / (max_x - min_x)
```

[11]:
```
# normalize dependent variables
y_norm = y_train.copy()

max_y = np.max(y_train)
min_y = np.min(y_train)

y_norm = (y_train - min_y) / (max_y - min_y)
```

[12]:
```
M = len(y_train)
```

[13]:
```
def grad(theta, X, y):
    return 1 / M * np.sum((X.dot(theta) - y) * X, axis=0).reshape(-1, 1)
```

[14]:
```
def cost_func(theta, X, y):
    return np.sum((X.dot(theta) - y) ** 2, axis=0)[0]
```

[15]:
```
def gradient_descent(theta0, X, y, learning_rate=0.5, epochs=1000, TOL=1e-7):
    theta_history = [theta0]
    j_history = [cost_func(theta0, X, y)]

    theta_new = theta0 * 10000

    for epoch in range(epochs):
        if epoch % 100 == 0:
            print(f'{epoch:5d}\t{j_history[-1]:7.4f}\t')

        dj = grad(theta0, X, y)
        j = cost_func(theta0, X, y)

        theta_new = theta0 - learning_rate * dj
        theta_history.append(theta_new)
```

```
        j_history.append(j)

        if np.sum((theta_new - theta0) ** 2) < TOL:
            print('Convergence achieved.')
            break

        theta0 = theta_new

    return theta_new, theta_history, j_history
```

[16]:
```
def predict(X, theta):
    X_norm = (X - min_x) / (max_x - min_x)
    y_pred_norm = X_norm.dot(theta)

    return y_pred_norm * (max_y - min_y) + min_y
```

[17]:
```
theta0 = np.zeros((X_norm.shape[1], 1)) + 0.4
```

[18]:
```
theta, theta_history, j_history = gradient_descent(theta0, X_norm, y_norm)
```

```
    0    3.8723
  100    3.6385
  200    3.5909
  300    3.5504
Convergence achieved.
```

[19]:
```
y_hat = predict(X_test, theta)
```

[20]:
```
print('Residual sum of squares: %.2f' % np.mean((y_hat - y_test) ** 2))
```

```
Residual sum of squares: 626.66
```

More information about gradient descent method is here.

0.0.13 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:

[1]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.preprocessing import PolynomialFeatures
import numpy as np
import pandas as pd
```

[2]:
```
df = pd.read_csv('data/fuel_consumption_co2.csv')
```

[3]:
```
columns = [
    'ENGINESIZE',
    'CYLINDERS',
    'FUELCONSUMPTION_CITY',
    'FUELCONSUMPTION_HWY',
    'FUELCONSUMPTION_COMB',
    'CO2EMISSIONS'
]
cdf = df[columns]
```

[4]:
```
# train test split
msk = np.random.rand(len(df)) < 0.8
train = cdf[msk]
test = cdf[~msk]
```

[5]:
```
independent = ['ENGINESIZE']
dependent = ['CO2EMISSIONS']
```

[6]:
```
X_train = np.asanyarray(train[independent])
y_train = np.asanyarray(train[dependent])

X_test = np.asanyarray(test[independent])
y_test = np.asanyarray(test[dependent])


poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train)
```

[7]:
```
regr = LinearRegression()
```

[8]:
```
regr.fit(X_train_poly, y_train)
print('Coefficients: ', regr.coef_)
```

```
Coefficients:  [[ 0.         50.89709271 -1.61364228]]
```

[9]:
```
X_test_poly = poly.fit_transform(X_test)
y_hat = regr.predict(X_test_poly)

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

```
Mean absolute error: 21.25
Residual sum of squares (MSE): 765.96
R2-score: 0.76
```

[1]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    confusion_matrix, classification_report, jaccard_score, log_loss
)
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd
```

[2]:
```
df = pd.read_csv('data/churn_data.csv')
```

[3]:
```
columns = [
    'tenure',
    'age',
    'address',
    'income',
    'ed',
    'employ',
    'equip',
    'callcard',
    'wireless',
    'churn'
]
```

[4]:
```
churn = df[columns].copy()
churn['churn'] = churn['churn'].astype('int')
```

[5]:
```
scaler = StandardScaler()
```

[6]:
```
columns = [
    'tenure',
```

```
    'age',
    'address',
    'income',
    'ed',
    'employ',
    'equip'
]
```

[7]:
```
X = np.asarray(churn[columns])
X = scaler.fit_transform(X)
```

[8]:
```
y = np.asarray(churn['churn'])
```

[9]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=4)
```

[10]:
```
lr = LogisticRegression(C=0.01, solver='liblinear').fit(X_train,y_train)
```

[11]:
```
y_hat = lr.predict(X_test)
y_hat_prob = lr.predict_proba(X_test)
```

[13]:
```
confusion_matrix(y_test, y_hat)
```

[13]:
```
array([[24,  1],
       [ 9,  6]])
```

[14]:
```
print(classification_report(y_test, y_hat))
```

```
              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
```

0.0.14 Log loss

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.

[15]:
```
log_loss(y_test, y_hat_prob)
```

[15]:
```
0.6017092478101185
```

0.0.15 Hierarchical Clustering - Agglomerative

[1]:
```
from matplotlib.axes._axes import _log as matplotlib_axes_logger
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from scipy.spatial import distance_matrix
from scipy.spatial.distance import euclidean
from sklearn.cluster import AgglomerativeClustering
from sklearn.preprocessing import MinMaxScaler
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pylab
import scipy
import warnings

matplotlib_axes_logger.setLevel('ERROR')
warnings.filterwarnings('ignore')
```

[2]:
```
df = pd.read_csv('data/cars_clus.csv')
```

[3]:
```
columns = [
    'sales',
    'resale',
    'type',
    'price',
    'engine_s',
    'horsepow',
    'wheelbas',
    'width',
    'length',
    'curb_wgt',
    'fuel_cap',
    'mpg',
    'lnsales'
]
df[columns] = df[columns].apply(pd.to_numeric, errors='coerce')
```

[4]:
```
df = df.dropna()
df = df.reset_index(drop=True)
```

[5]:
```
columns = [
    'engine_s',
    'horsepow',
    'wheelbas',
    'width',
    'length',
```

```
    'curb_wgt',
    'fuel_cap',
    'mpg'
]
dataset = df[columns]
```

[6]:
```
X = dataset.values
mms = MinMaxScaler()
features = mms.fit_transform(X)
```

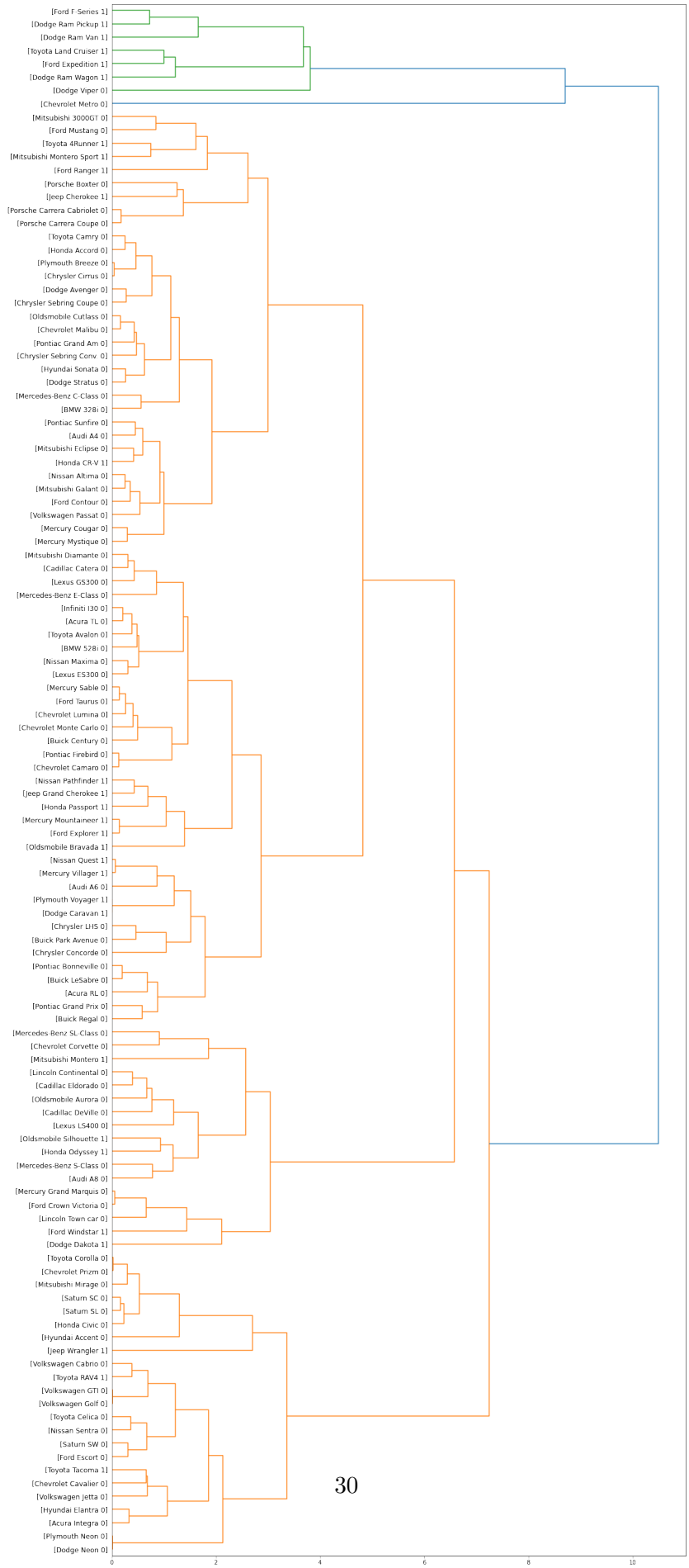
[7]:
```
leng = features.shape[0]
D = scipy.zeros([leng, leng])

for i in range(leng):
    for j in range(leng):
        D[i,j] = euclidean(features[i], features[j])
```

[8]:
```
Z = linkage(D, 'complete')
```

[9]:
```
k = 3
clusters = fcluster(Z, k, criterion='distance')
```

[10]:
```
def llf(id):
    return '[%s %s %s]' % (df['manufact'][id], df['model'][id],␣
 ↪int(float(df['type'][id])))
```

[11]:
```
fig = pylab.figure(figsize=(18,50))
dendro = dendrogram(Z, leaf_label_func=llf, leaf_rotation=0, leaf_font_size=12,␣
 ↪orientation='right')
```


[Ford F-Series 1]
[Dodge Ram Pickup 1]
[Dodge Ram Van 1]
[Toyota Land Cruiser 1]
[Ford Expedition 1]
[Dodge Ram Wagon 1]
[Dodge Viper 0]
[Chevrolet Metro 0]
[Mitsubishi 3000GT 0]
[Ford Mustang 0]
[Toyota 4Runner 1]
[Mitsubishi Montero Sport 1]
[Ford Ranger 1]
[Porsche Boxter 0]
[Jeep Cherokee 1]
[Porsche Carrera Cabriolet 0]
[Porsche Carrera Coupe 0]
[Toyota Camry 0]
[Honda Accord 0]
[Plymouth Breeze 0]
[Chrysler Cirrus 0]
[Dodge Avenger 0]
[Chrysler Sebring Coupe 0]
[Oldsmobile Cutlass 0]
[Chevrolet Malibu 0]
[Pontiac Grand Am 0]
[Chrysler Sebring Conv 0]
[Hyundai Sonata 0]
[Dodge Stratus 0]
[Mercedes-Benz C-Class 0]
[BMW 328i 0]
[Pontiac Sunfire 0]
[Audi A4 0]
[Mitsubishi Eclipse 0]
[Honda CR-V 1]
[Nissan Altima 0]
[Mitsubishi Galant 0]
[Ford Contour 0]
[Volkswagen Passat 0]
[Mercury Cougar 0]
[Mercury Mystique 0]
[Mitsubishi Diamante 0]
[Cadillac Catera 0]
[Lexus GS300 0]
[Mercedes-Benz E-Class 0]
[Infiniti I30 0]
[Acura TL 0]
[Toyota Avalon 0]
[BMW 528i 0]
[Nissan Maxima 0]
[Lexus ES300 0]
[Mercury Sable 0]
[Ford Taurus 0]
[Chevrolet Lumina 0]
[Chevrolet Monte Carlo 0]
[Buick Century 0]
[Pontiac Firebird 0]
[Chevrolet Camaro 0]
[Nissan Pathfinder 1]
[Jeep Grand Cherokee 1]
[Honda Passport 1]
[Mercury Mountaineer 1]
[Ford Explorer 1]
[Oldsmobile Bravada 1]
[Nissan Quest 1]
[Mercury Villager 1]
[Audi A6 0]
[Plymouth Voyager 1]
[Dodge Caravan 1]
[Chrysler LHS 0]
[Buick Park Avenue 0]
[Chrysler Concorde 0]
[Pontiac Bonneville 0]
[Buick LeSabre 0]
[Acura RL 0]
[Pontiac Grand Prix 0]
[Buick Regal 0]
[Mercedes-Benz SL Class 0]
[Chevrolet Corvette 0]
[Mitsubishi Montero 1]
[Lincoln Continental 0]
[Cadillac Eldorado 0]
[Oldsmobile Aurora 0]
[Cadillac DeVille 0]
[Lexus LS400 0]
[Oldsmobile Silhouette 1]
[Honda Odyssey 1]
[Mercedes-Benz S-Class 0]
[Audi A8 0]
[Mercury Grand Marquis 0]
[Ford Crown Victoria 0]
[Lincoln Town car 0]
[Ford Windstar 1]
[Dodge Dakota 1]
[Toyota Corolla 0]
[Chevrolet Prizm 0]
[Mitsubishi Mirage 0]
[Saturn SC 0]
[Saturn SL 0]
[Honda Civic 0]
[Hyundai Accent 0]
[Jeep Wrangler 1]
[Volkswagen Cabrio 0]
[Toyota RAV4 1]
[Volkswagen GTI 0]
[Volkswagen Golf 0]
[Toyota Celica 0]
[Nissan Sentra 0]
[Saturn SW 0]
[Ford Escort 0]
[Toyota Tacoma 1]
[Chevrolet Cavalier 0]
[Volkswagen Jetta 0]
[Hyundai Elantra 0]
[Acura Integra 0]
[Plymouth Neon 0]
[Dodge Neon 0]
0
2
4
6
8
10

0.0.16 Clustering using scikit-learn

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.

[12]:
```
dist_matrix = distance_matrix(features, features)
```

[13]:
```
agglom = AgglomerativeClustering(n_clusters=6, linkage='complete')
```

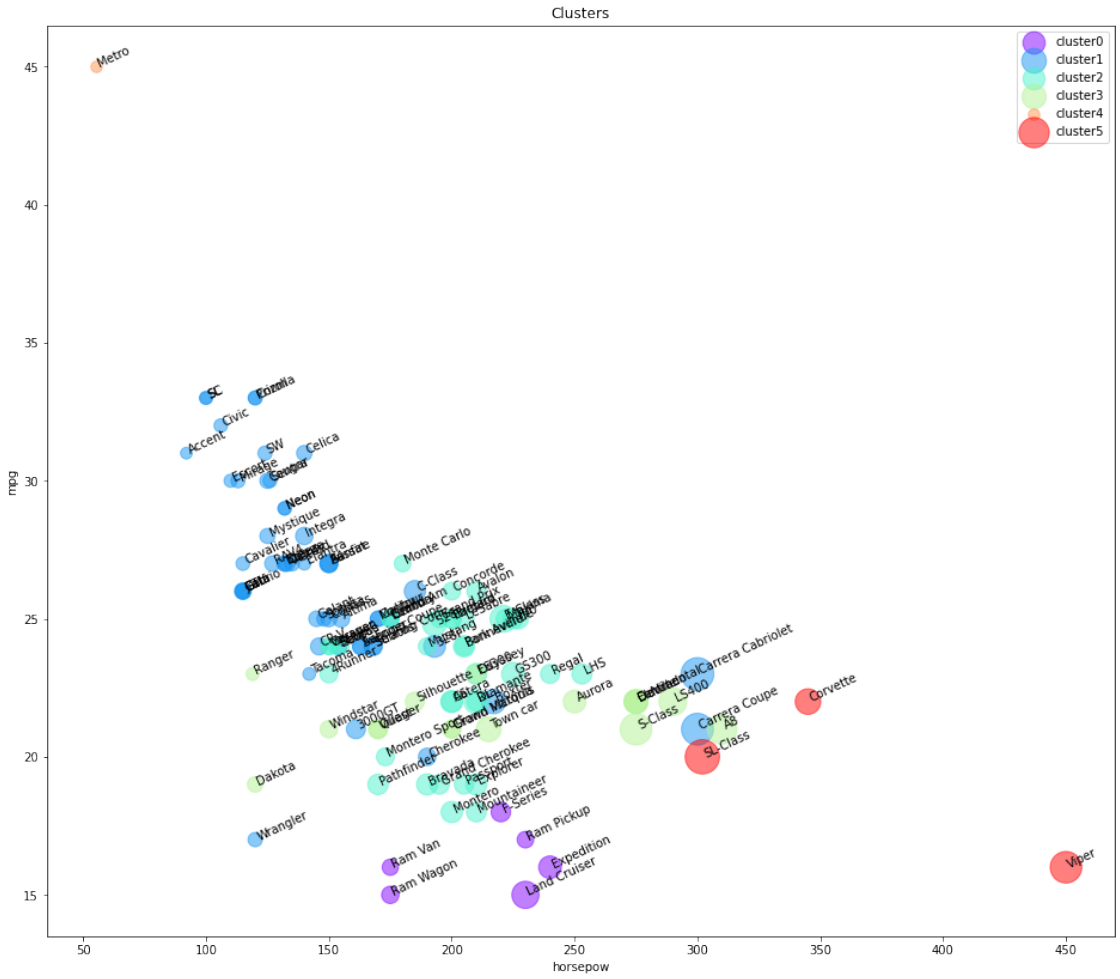
[14]:
```
agglom.fit(features)
```

[14]:
```
AgglomerativeClustering(linkage='complete', n_clusters=6)
```

[15]:
```
df['cluster'] = agglom.labels_
```

[16]:
```
n_clusters = max(agglom.labels_) + 1
colors = cm.rainbow(np.linspace(0, 1, n_clusters))
cluster_labels = list(range(0, n_clusters))

plt.figure(figsize=(16, 14))

for color, label in zip(colors, cluster_labels):
    subset = df[df.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),
        alpha=0.5
    )

plt.legend()
plt.title('Clusters')
plt.xlabel('horsepow')
plt.ylabel('mpg');
```



[17]:
```
cols = [
    'horsepow',
    'engine_s',
    'mpg',
    'price'
]
agg_cars = df.groupby(['cluster', 'type'])[cols].mean()
agg_cars
```

[17]:

cluster	type	horsepow	engine_s	mpg	price
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:

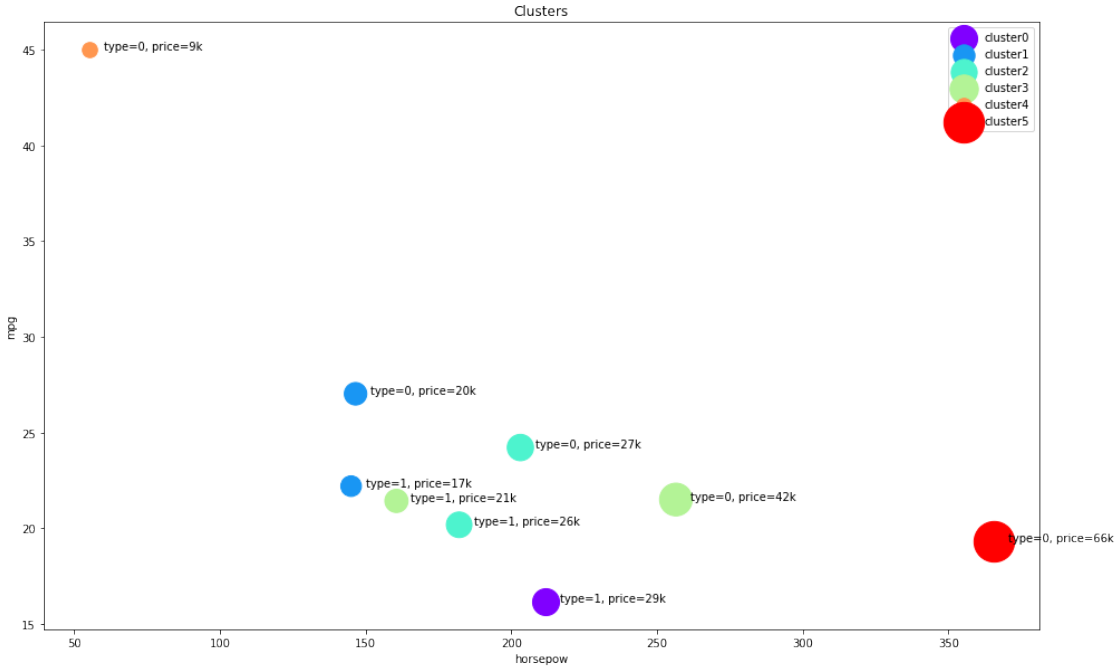
- Cluster 1: with almost high mpg, and low in horsepower.
- Cluster 2: with good mpg and horsepower, but higher price than average.
- Cluster 3: with low mpg, high horsepower, highest price.

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.

[18]:
```
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(subset.loc[i][3])) +␣
↪'k'
        )

    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');
```



0.0.17 Customer Segmentation with K-Means

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.

[1]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

[2]:
```
df = pd.read_csv('data/customer_segmentation.csv')
```

[3]:
```
customers = df.drop('Address', axis=1)
```

[4]:
```
X = customers.values[:,1:]
X = np.nan_to_num(X)
dataset = StandardScaler().fit_transform(X)
```

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

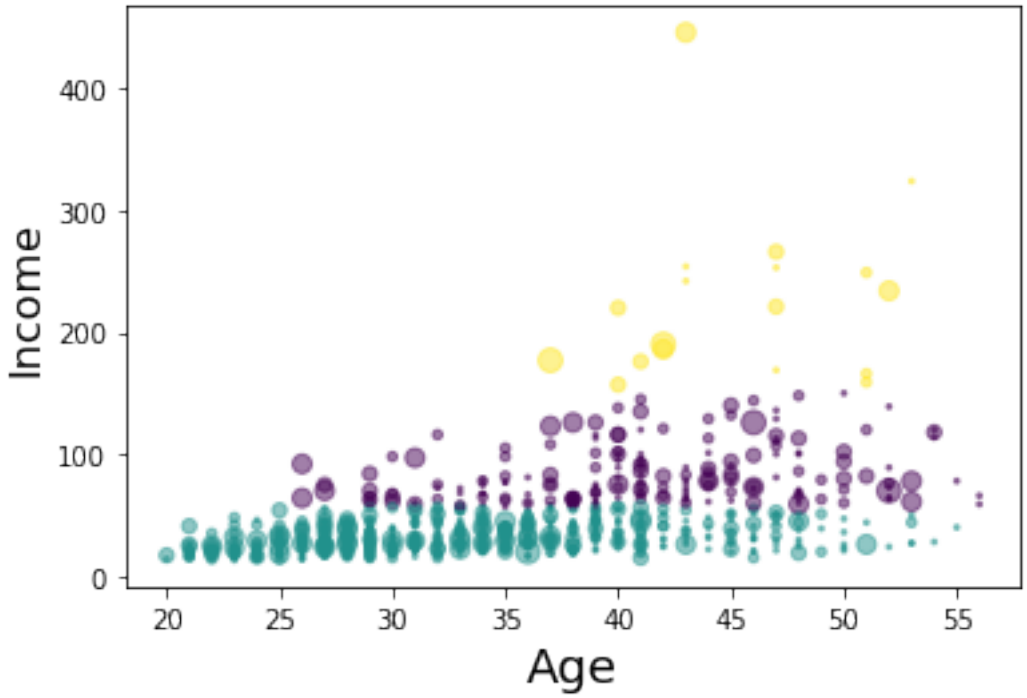
experiments. However, using the K-means clustering we can do all this process much easier.

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

[5]:
```
num_of_clusters = 3
k_means = KMeans(init='k-means++', n_clusters=num_of_clusters, n_init=12)
k_means.fit(X)
labels = k_means.labels_
```

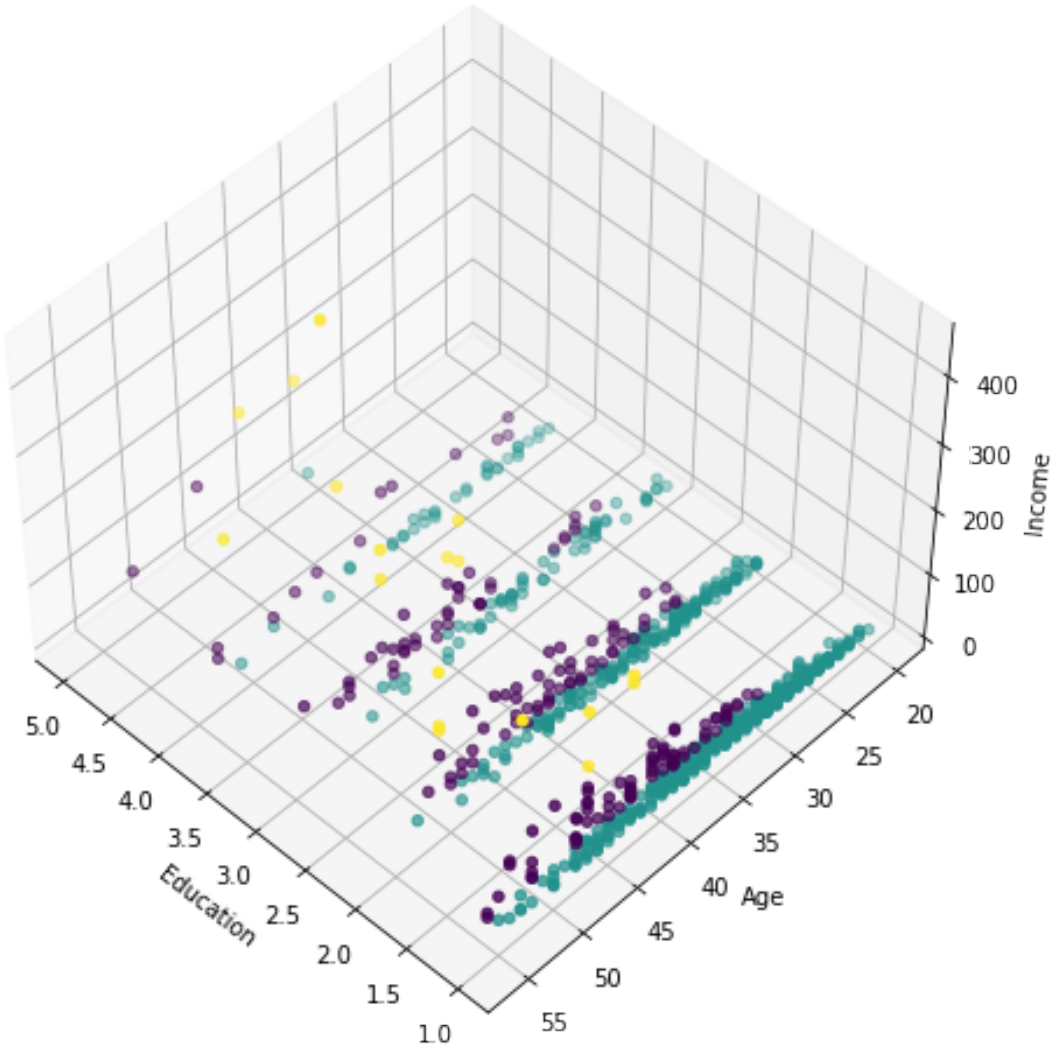
[6]:
```
customers['cluster'] = labels
```

[7]:
```
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()
```



[8]:
```
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()
```

```
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));
```



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

[1]:
```
%%capture
!unzip data/data.zip -d data/
```

[2]:
```
from math import sqrt
from IPython.display import Image
import pandas as pd
```

0.0.18 Collaborative Filtering

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

The first step is to discover how similar the active user is to the other users. How do we do this? Well, this can be done through several different statistical and vectorial techniques such as distance or similarity measurements including Euclidean Distance, Pearson Correlation, Cosine Similarity, and so on. To calculate the level of similarity between two users, we use the three movies that both the users have rated in the past. Regardless of what we use for similarity measurement, let's say for example, the similarity could be 0.7, 0.9, and 0.4 between the active user and other users. These numbers represent similarity weights or proximity of the active user to other users in the dataset.

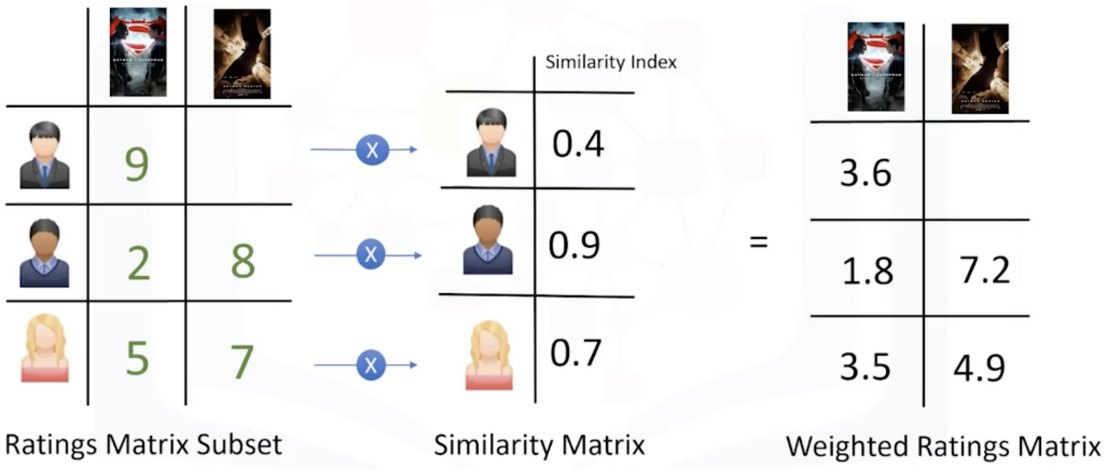
[3]:
```
Image('img/form-6.png', width=800, height=400)
```

[3]:

The next step is to create a weighted rating matrix. We just calculated the similarity of users to our active user in the previous slide. Now, we can use it to calculate the possible opinion of the active user about our two target movies. This is achieved by multiplying the similarity weights to the user ratings. It results in a weighted ratings matrix, which represents the user's neighbors opinion about are two candidate movies for recommendation. In fact, it incorporates the behavior of other users and gives more weight to the ratings of those users who are more similar to the active user.

[4]:
```
Image('img/form-7.png', width=800, height=400)
```

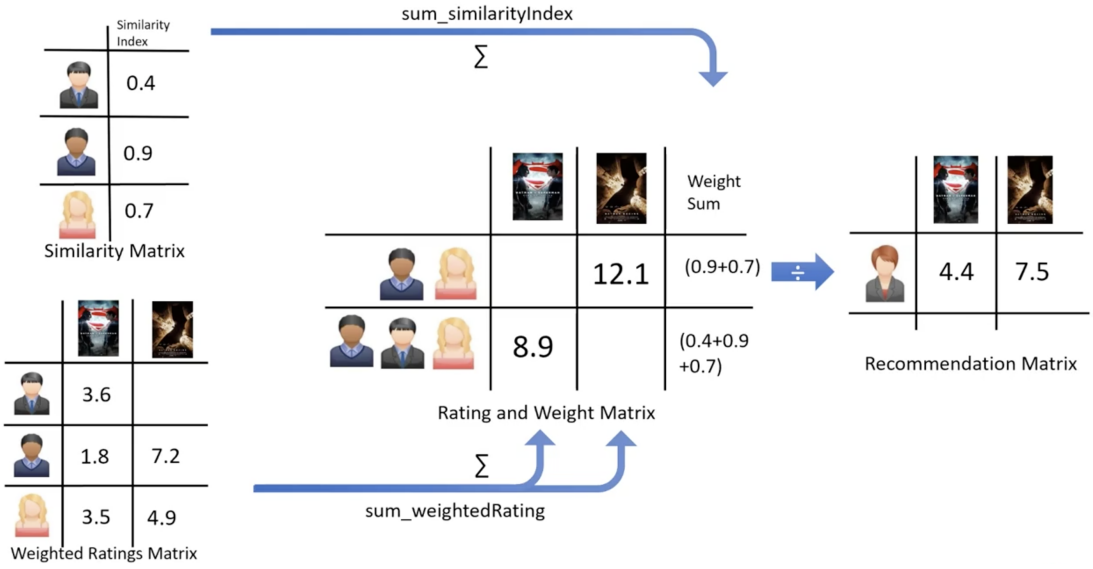
[4]:



Now, we can generate the recommendation matrix by aggregating all of the weighted rates. However, as three users rated the first potential movie and two users rated the second movie, we have to normalize the weighted rating values. We do this by dividing it by the sum of the similarity index for users. The result is the potential rating that our active user will give to these movies based on her similarity to other users. It is obvious that we can use it to rank the movies for providing recommendation to our active user.

[5]:
```
Image('img/form-8.png', width=800, height=400)
```

[5]:



[6]:
```
movies = pd.read_csv('data/movies.csv')
ratings = pd.read_csv('data/ratings.csv')
```

[7]:
```
movies['year'] = movies.title.str.extract('(\(\d\d\d\d\))', expand=False)
movies['year'] = movies.year.str.extract('(\d\d\d\d)', expand=False)
movies['title'] = movies.title.str.replace('(\(\d\d\d\d\))', '')
movies['title'] = movies['title'].apply(lambda x: x.strip())
```

[8]:
```
movies = movies.drop('genres', axis=1)
ratings = ratings.drop('timestamp', axis=1)
```

[9]:
```
user_input = [
    {
        'movieId': 1968,
        'title': 'Breakfast Club, The',
        'rating': 5
    },
    {
        'movieId': 1,
```

```
        'title': 'Toy Story',
        'rating': 3.5
    },
    {
        'movieId': 2,
        'title': 'Jumanji',
        'rating': 2
    },
    {
        'movieId': 296,
        'title': 'Pulp Fiction',
        'rating': 5
    },
    {
        'movieId': 1274,
        'title': 'Akira',
        'rating': 4.5
    }
]
input_movies = pd.DataFrame(user_input).sort_values('movieId').
 ↪reset_index(drop=True)
```

[10]:
```
user_movies = movies[movies['movieId'].isin(input_movies['movieId'].tolist())]
user_movies.head()
```

[10]:

	movieId	title	year
0	1	Toy Story	1995
1	2	Jumanji	1995
293	296	Pulp Fiction	1994
1246	1274	Akira	1988
1885	1968	Breakfast Club, The	1985

[11]:
```
user_subset = ratings[ratings['movieId'].isin(input_movies['movieId'].tolist())]
user_subset.head()
```

[11]:

	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

[12]:
```
grouped = user_subset.groupby(['userId'])
```

[13]:
```
grouped = sorted(grouped,  key=lambda x: len(x[1]), reverse=True)
```

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.

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.

We will select a subset of users to iterate through. This limit is imposed because we don't want to waste too much time going through every single user.

[14]:
```
grouped = grouped[0: 100]
```

[15]:
```
pearson_corr = {}

for name, group in grouped:
    group = group.sort_values(by='movieId')
    input_movies = input_movies.sort_values(by='movieId')
    n_ratings = len(group)

    temp = input_movies[input_movies['movieId'].isin(group['movieId'].tolist())]
    temp_ratings = temp['rating'].tolist()
    temp_groups = group['rating'].tolist()

    sxx = sum([i ** 2 for i in temp_ratings]) - pow(sum(temp_ratings), 2) /␣
↪float(n_ratings)
    syy = sum([i ** 2 for i in temp_groups]) - pow(sum(temp_groups), 2) /␣
↪float(n_ratings)
    sxy = sum(i * j for i, j in zip(temp_ratings, temp_groups)) -␣
↪sum(temp_ratings) * sum(temp_groups) / float(n_ratings)

    if sxx != 0 and syy != 0:
        pearson_corr[name] = sxy / sqrt(sxx * syy)
    else:
        pearson_corr[name] = 0
```

[16]:
```
pearson_df = pd.DataFrame.from_dict(pearson_corr, orient='index')
pearson_df.columns = ['similarity']
pearson_df['userId'] = pearson_df.index
pearson_df.index = range(len(pearson_df))
pearson_df.head()
```

[16]:

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

[17]:
```
top_users = pearson_df.sort_values(by='similarity', ascending=False)[0:50]
top_users.head()
```

[17]:

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

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 **pearson_df** 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.

[18]:
```
top_ratings = top_users.merge(ratings, left_on='userId', right_on='userId')
```

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:

[19]:
```
top_ratings['weighted_rating'] = top_ratings['similarity'] *␣
↪top_ratings['rating']
top_ratings.head()
```

[19]:

	similarity	userId	movieId	rating	weighted_rating
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
```

[20]:
```
temp = top_ratings.groupby('movieId').sum()[['similarity','weighted_rating']]
temp.columns = ['sum_similarity','sum_weighted_rating']
temp.head()
```

[20]:
```
         sum_similarity  sum_weighted_rating
movieId
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
```

[21]:
```
rec_df = pd.DataFrame()
rec_df['recommendation_score'] = temp['sum_weighted_rating'] /␣
↪temp['sum_similarity']
rec_df['movieId'] = temp.index
rec_df.head()
```

[21]:
```
   recommendation_score  movieId
1              3.668955        1
2              2.518658        2
3              2.657941        3
4              3.000000        4
5              2.316058        5
```

[22]:
```
rec_df = rec_df.sort_values(by='recommendation_score', ascending=False)
```

[23]:
```
movies.loc[movies['movieId'].isin(rec_df.head(10)['movieId'].tolist())]
```

[23]:
```
       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

[1]:
```
%%capture
!unzip data/data.zip -d data/
```

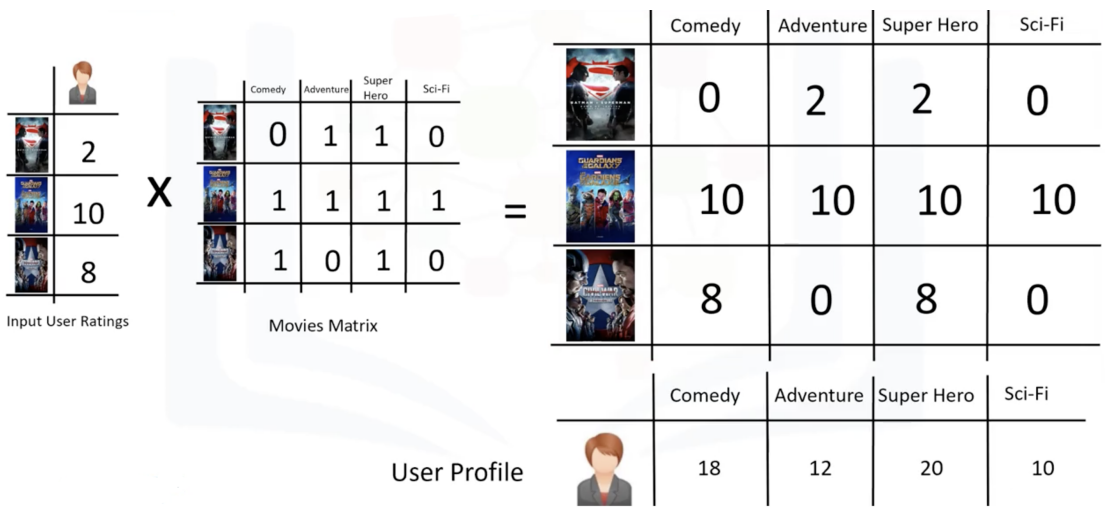
[2]:
```
from IPython.display import Image
import pandas as pd
```

0.0.19 Content Based Filtering

The task of the recommender engine is to recommend one of the three candidate movies to this user, or in other, words we want to predict what the user's possible rating would be of the three candidate movies if she were to watch them. To achieve this, we have to build the user profile. First, we create a vector to show the user's ratings for the movies that she's already watched. We call it Input User Ratings. Then, we encode the movies through the one-hot encoding approach. Genre of movies are used here as a feature set. We use the first three movies to make this matrix, which represents the movie feature set matrix. If we multiply these two matrices we can get the weighted feature set for the movies. Let's take a look at the result. This matrix is also called the Weighted Genre matrix and represents the interests of the user for each genre based on the movies that she's watched. Now, given the Weighted Genre Matrix, we can shape the profile of our active user.

[3]:
```
Image('img/form-1.png', width=800, height=400)
```

[3]:



Essentially, we can aggregate the weighted genres and then normalize them to find the user profile. It clearly indicates that she likes superhero movies more than other genres. We use this profile to figure out what movie is proper to recommend to this user.

[4]:
```
Image('img/form-2.png', width=800, height=400)
```

[4]:

What would the user think of new movies?

[5]:
```
Image('img/form-3.png', width=400, height=100)
```

[5]:

Now we're in the position where we have to figure out which of the above movies is most suited to be recommended to the user. To do this, we simply multiply the User Profile matrix by the candidate Movie Matrix, which results in the Weighted Movies Matrix. It shows the weight of each genre with respect to the User Profile. Now, if we aggregate these weighted ratings, we get the active user's possible interest level in these three movies. In essence, it's our recommendation lists, which we can sort to rank the movies and recommend them to the user.

[6]:
```
Image('img/form-4.png', width=800, height=400)
```

[6]:

[7]:
```
Image('img/form-5.png', width=800, height=400)
```

[7]:

Let's coding this!

[8]:
```
movies = pd.read_csv('data/movies.csv')
ratings = pd.read_csv('data/ratings.csv')
```

[9]:
```
movies['year'] = movies.title.str.extract('(\(\d\d\d\d\))', expand=False)
movies['year'] = movies.year.str.extract('(\d\d\d\d)', expand=False)
movies['title'] = movies.title.str.replace('(\(\d\d\d\d\))', '')
movies['title'] = movies['title'].apply(lambda x: x.strip())
```

[10]:
```
movies['genres'] = movies.genres.str.split('|')
```

[11]:
```
ratings = ratings.drop('timestamp', axis=1)
```

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.

[12]:
```
dummies = pd.get_dummies(movies['genres'].apply(pd.Series).stack()).sum(level=0)
dfs = [
    movies.drop('genres', axis=1),
    dummies
]
movies = pd.concat(dfs, axis=1)
```

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:

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.

[13]:
```
user_input = [
    {
        'movieId': 1968,
        'title': 'Breakfast Club, The',
        'rating': 5
    },
    {
        'movieId': 1,
        'title': 'Toy Story',
        'rating': 3.5
    },
    {
        'movieId': 2,
        'title': 'Jumanji',
        'rating': 2
    },
    {
        'movieId': 296,
        'title': 'Pulp Fiction',
        'rating': 5
```

```
    },
    {
        'movieId': 1274,
        'title': 'Akira',
        'rating': 4.5
    }
]
input_movies = pd.DataFrame(user_input).sort_values('movieId').
↪reset_index(drop=True)
```

[14]:
```
input_movies
```

[14]:

	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.

[15]:
```
user_movies = movies[movies['movieId'].isin(input_movies['movieId'].tolist())]
```

[16]:
```
user_movies.head()
```

[16]:

	movieId	title	year	(no genres listed)	Action	\
0	1	Toy Story	1995	0	0	
1	2	Jumanji	1995	0	0	
293	296	Pulp Fiction	1994	0	0	
1246	1274	Akira	1988	0	1	
1885	1968	Breakfast Club, The	1985	0	0	

	Adventure	Animation	Children	Comedy	Crime	…	Film-Noir	Horror	\
0	1	1	1	1	0	…	0	0	
1	1	0	1	0	0	…	0	0	
293	0	0	0	1	1	…	0	0	
1246	1	1	0	0	0	…	0	0	
1885	0	0	0	1	0	…	0	0	

	IMAX	Musical	Mystery	Romance	Sci-Fi	Thriller	War	Western
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
293	0	0	0	0	0	1	0	0
1246	0	0	0	0	1	0	0	0
1885	0	0	0	0	0	0	0	0

```
[5 rows x 23 columns]
```

[17]:
```
user_movies = user_movies.reset_index(drop=True)
user_genres = user_movies.drop(columns=['movieId', 'title', 'year'])
```

[18]:
```
user_genres
```

[18]:

	(no genres listed)	Action	Adventure	Animation	Children	Comedy	Crime	\
0	0	0	1	1	1	1	0	
1	0	0	1	0	1	0	0	
2	0	0	0	0	0	1	1	
3	0	1	1	1	0	0	0	
4	0	0	0	0	0	1	0	

	Documentary	Drama	Fantasy	Film-Noir	Horror	IMAX	Musical	Mystery	\
0	0	0	1	0	0	0	0	0	
1	0	0	1	0	0	0	0	0	
2	0	1	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	
4	0	1	0	0	0	0	0	0	

	Romance	Sci-Fi	Thriller	War	Western
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	0	0

[19]:
```
input_movies['rating']
```

[19]:
```
0    3.5
1    2.0
2    5.0
3    4.5
4    5.0
Name: rating, dtype: float64
```

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' `dot` function.

[20]:
```
profile = user_genres.transpose().dot(input_movies['rating'])
```

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.

[21]:
```
genre_table = movies.set_index('movieId').drop(columns=['title', 'year'])
```

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.

[22]:
```
recommendations = ((genre_table * profile).sum(axis=1)) / (profile.sum())
```

[23]:
```
recommendations = recommendations.sort_values(ascending=False)
```

[24]:
```
movies.loc[movies['movieId'].isin(recommendations.head(20).keys())]
```

[24]:

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

	(no genres listed)	Action	Adventure	Animation	Children	Comedy	\
664	0	0	1	1	1	1	
1824	0	0	1	1	1	1	
2902	0	0	1	1	1	1	
4923	0	0	1	0	0	1	
6793	0	0	1	0	0	1	
8605	0	0	1	1	1	1	
8783	0	1	1	1	1	1	
9296	0	1	1	1	0	1	
9825	0	0	1	1	1	1	
11716	0	1	1	1	1	1	
11751	0	1	1	1	1	1	
13250	0	1	1	0	0	1	

```
16055                  0       1          1          0         0       1
18312                  0       0          1          1         1       1
22778                  0       0          1          1         1       1
22881                  0       1          1          1         1       1
25218                  0       1          1          0         0       1
26442                  0       1          1          0         0       1
32854                  0       0          0          1         1       1
33509                  0       0          1          0         1       1

       Crime  …  Film-Noir  Horror  IMAX  Musical  Mystery  Romance  Sci-Fi  \
664        0  …          0       0     0        0        0        0       1
1824       0  …          0       0     0        1        0        1       0
2902       1  …          0       0     0        0        1        0       0
4923       1  …          0       0     0        0        1        0       1
6793       0  …          0       0     0        0        1        0       1
8605       0  …          0       0     0        1        0        1       0
8783       0  …          0       0     0        0        0        0       0
9296       0  …          0       0     0        0        0        1       0
9825       0  …          0       0     1        0        0        0       1
11716      0  …          0       0     0        0        0        0       0
11751      0  …          0       0     0        0        0        0       0
13250      1  …          0       0     0        0        0        0       0
16055      1  …          1       1     0        0        1        0       0
18312      0  …          0       0     0        0        0        0       0
22778      0  …          0       0     0        0        0        1       0
22881      0  …          0       0     0        0        0        0       0
25218      0  …          0       0     0        0        0        0       0
26442      1  …          0       0     0        0        0        0       0
32854      0  …          0       0     0        0        0        1       1
33509      0  …          0       0     0        0        0        0       1

       Thriller  War  Western
664           0    0        0
1824          0    0        0
2902          0    0        0
4923          1    0        0
6793          1    0        0
8605          0    0        0
8783          0    0        0
9296          0    0        0
9825          0    0        0
11716         0    0        0
11751         0    0        0
13250         1    0        0
16055         1    0        1
18312         0    0        0
22778         0    0        0
```

```
22881          0    0         0
25218          1    0         0
26442          1    0         0
32854          0    0         0
33509          0    0         0

[20 rows x 23 columns]
```

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