Chapter 1 - Data Preprocessing

1. Import libraries.

2. Import dataset. Split into **features** (x-var) and **dependent variable vector** (y-var):

    - To **locate indexes**, we use *iloc(row,col)*.
    - We use .values because we want to turn that pandas DF into a numpy array.

3. Handle missing data. There are several options:
    - Deletion. Okay for large datasets and few missing.
    - Replace with mean / median for numbers, and mode for categories. Use scikitlearn.

4. Encode data.
    - For categorical data, we do this so that models will have a easier time to process them as numbers. But cannot just mere numbers or the machine will misinterpret wrong relationships and assume there is an order (1 2 3 got order). Prof's methods are to use **one hot encoding** into binary forms.
    - *ColumnTransformer(transformers=[(what_transformation, what_transformation_exactly, indexes)], remainder="do_what?")* where apply at index but don't touch remainder.

5. Split into train and test set.

6. Feature scaling. There are a lot of debates if should feature scale before or after train-test split. But there's only one answer: **only do this after splitting**. Feature scaling is done to prevent one feature from dominating the others **as many models are based on Euclidean distance**. The test set is meant to be separate from the train set to properly test it; it is akin to new data in the future. If you feature scale everything together, then it would cause **information leakage** on the test set. Additionally, you use the trainset's mean and SD to scale for each new testset; your testset needs to be scaled by the same scaler used on the training set. This is because the model is trained with a particular scaler, in order to make predictions that are congruent with how it was trained, we need to apply the trainset's scalar on any future testsets.

    - Standardisation's numbers fall between -3 and 3.
    - Normalisation's numbers fall between 0 and 1.
    - Normalisation better if most features have normal dist. Standardisation works well all the time. Since all the time and not needing to care about the features, use standardisation.
    - You do not need to apply standardisation to the dummy variables (aka the one hot encoding). Since dummy variables are already between -3 and 3, then no need. In fact, applying standardisation to them makes it worse as it'll stretch them to -3 and 3, losing the information / interpretation of what they stand for.
    - Not all models need feature scaling. A lot of them incorporate inside.

$x_{stand} = \frac{x - mean(x)}{standard\ deviation\ (x)}$	$x_{norm} = \frac{x - min(x)}{max(x) - min(x)}$

Further Notes

- fit() only calculates mean and sd for each col. It is a function that **fits an equation/model to data**. It extracts out some information of the data on which the object is applied.
- transform() actually does the transformation. It is a function that **applies the fit() to a dataset**.

In [76]:

```
### 1. Libraries

import numpy as np # to work with arrays
import matplotlib.pyplot as plt # to make nice charts
import pandas as pd # to import datasets and as matrices

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values # matrix of features
Y = data.iloc[:,-1].values # dependent variable vector

### 3. Missing Data

from sklearn.impute import SimpleImputer # import class
imputer = SimpleImputer(missing_values=np.nan, strategy='mean') # cre
imputer.fit(X[:,1:3]) # to apply the function to the dataset, calcula
X[:,1:3] = imputer.transform(X[:,1:3]) # the actual replacement

### 4. Encoding

# Categorical Data (One-Hot Encoding)
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder',OneHotEncoder(),[0])]
X = np.array(ct.fit_transform(X)) # fit and transform together. retur

# Dependent Variable (turn into 1s and 0s)
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
Y = le.fit_transform(Y)

# print(data,"\n",X,"\n",Y)

### 5. Split (into 4 sets)

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0
# print(X_train,"\n###\n",X_test,"\n###\n",Y_train,"\n###\n",Y_test)
```

```
40
41 ### 6. Feature Scaling (prevent dominating features)
42
43 from sklearn.preprocessing import StandardScaler
44 sc = StandardScaler()
45 X_train[:,3:] = sc.fit_transform(X_train[:,3:])
46 X_test[:,3:] = sc.transform(X_test[:,3:]) # no fitting!!!
47
48 print(X_train, "\n", X_test)
```

```
[[0.0 0.0 1.0 -0.19159184384578545 -1.0781259408412425]
 [0.0 1.0 0.0 -0.014117293757057777 -0.07013167641635372]
 [1.0 0.0 0.0 0.566708506533324 0.633562432710455]
 [0.0 0.0 1.0 -0.30453019390224867 -0.30786617274297867]
 [0.0 0.0 1.0 -1.9018011447007988 -1.420463615551582]
 [1.0 0.0 0.0 1.1475343068237058 1.232653363453549]
 [0.0 1.0 0.0 1.4379472069688968 1.5749910381638885]
 [1.0 0.0 0.0 -0.7401495441200351 -0.5646194287757332]]
 [[0.0 1.0 0.0 -1.4661817944830124 -0.9069571034860727]
 [1.0 0.0 0.0 -0.44973664397484414 0.2056403393225306]]
```

Chapter 2 - Simple Linear Regression

Basically represented as:

```
y = b0 + b1x1
```

It creates a **best fit line** based on the least sum of squared error / min SSE. Also note that **p-values only apply to linear models**.

In [16]:

```
1  ### 1. Libraries
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import pandas as pd
6
7  ### 2. Dataset
8
9  data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
10 X = data.iloc[:,:-1].values
11 Y = data.iloc[:,-1].values
12
13 ### 3. Split (into 4 sets)
14
15 from sklearn.model_selection import train_test_split
16 X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0
17 # print(X_train,"\n###\n",X_test,"\n###\n",Y_train,"\n###\n",Y_test)
18
19 ### 4. Training the Model
20
21 from sklearn.linear_model import LinearRegression
22 regressor = LinearRegression()
23 regressor.fit(X_train, Y_train) # at this point, fit() will take the
24
25 ### 5. Predict
```

```
26
27 Y_predictions = regressor.predict(X_test)
28
29 ### 6. Visualisations
30
31 plt.scatter(X_train, Y_train, color='red')
32 plt.plot(X_train, regressor.predict(X_train), color='blue')
33 plt.title("Salary vs Experience (Training Set)")
34 plt.xlabel("Years of Experience")
35 plt.ylabel("Salary")
36 plt.show()
37
38 plt.scatter(X_test, Y_test, color='red')
39 plt.plot(X_train, regressor.predict(X_train), color='blue') # don't n
40 plt.plot(X_test, Y_predictions, color='yellow') # see? exact same lin
41 plt.title("Salary vs Experience (Test Set)")
42 plt.xlabel("Years of Experience")
43 plt.ylabel("Salary")
44 plt.show()
45
46 ### Bonus
47
48 # Predict an employee with 12 years experience:
49 print(regressor.predict([[12]])) # double brackets cos it expects a 2
50
51 # Get the coeffecients
52 print(regressor.coef_, regressor.intercept_)
```

```
[137605.23485427]
[9332.94473799] 25609.89799835482
```

Chapter 3 - Multiple Linear Regression

$$y = b_0 + b_1*x_1 + b_2*x_2 + \dots + b_n*x_n$$

Assumptions of All Kinds of Linear Regressions

- Linearity
- Homoscedasticity (the same variance at every X value)
- Multivariate normality (all residuals, aka y - yhat, are normally distributed)
- Independence of errors (all residuals, aka y - yhat, are independent)
- Lack of multicollinearity

Before using any LR model, need to check these are true. See here (https://www.statisticssolutions.com/assumptions-of-multiple-linear-regression/).

Notes

- For MLR, **no need any feature scaling** because how big or small, the betas will adjust themselves and put everything on the same scale.
- You don't need to check for the 5 assumptions, because if they do not hold, you will have bad performance results anyway.
- For the model we are importing, we don't have to worry about **dummy variable trap**. Can just keep everything.
- The model we are importing also automatically takes care of selecting the best features. Hence, no need to do any backward eliminations etc.

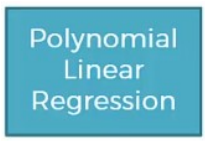
In [9]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

### 3. Encoding

# Categorical Data (One-Hot Encoding)
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder',OneHotEncoder(),[3])]
```

```
X = np.array(ct.fit_transform(X)) # fit and transform together. retur

### 4. Split (into 4 sets)

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

### 5. Modelling

from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train,Y_train)

### 6. Predictions

Y_pred = regressor.predict(X_test)

### 7. Comparisons

np.set_printoptions(precision=2) # 2 dp only
print(np.concatenate((Y_pred.reshape(len(Y_pred),1), Y_test.reshape(l
# since numbers are close, the model is probably good

### 8. Bonus

# Predict profit if R&D 160K, Admin 130K, Mkt 300K, State California
print(regressor.predict([[1,0,0,160000,130000,300000]])) # need 2D

# Model?
print(regressor.coef_, regressor.intercept_)
```

```
[[114664.42 105008.31]
 [ 90593.16  96479.51]
 [ 75692.84  78239.91]
 [ 70221.89  81229.06]
 [179790.26 191050.39]
 [171576.92 182901.99]
 [ 49753.59  35673.41]
 [102276.66 101004.64]
 [ 58649.38  49490.75]
 [ 98272.03  97483.56]]
[180892.25]
[-2.85e+02  2.98e+02 -1.24e+01  7.74e-01 -9.44e-03  2.89e-02] 49834.88
507321703
```

Chapter 4 - Polynomial Regression



$$y = b_0 + b_1 x_1 + b_2 x_1^2 + \ldots + b_n x_1^n$$

Q: Why is it still called LINEAR? When you're talking about the class of regression, you're actually referring to the coefficients b0 b1 bn etc.!!! You're actually asking "can this function be

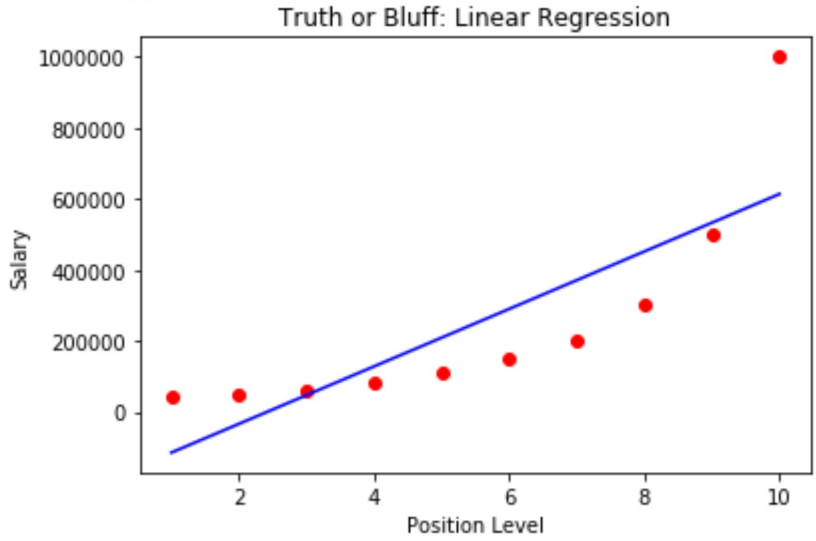
expressed as a linear combination of these coefficients?" An example of a non-linear regression would be the equation y = 0 + b1X1 / b2+x2 etc, where you cannot replace the coefficients with other coefficients to turn the equation into a linear one in regards to the coefficients not the x-values.

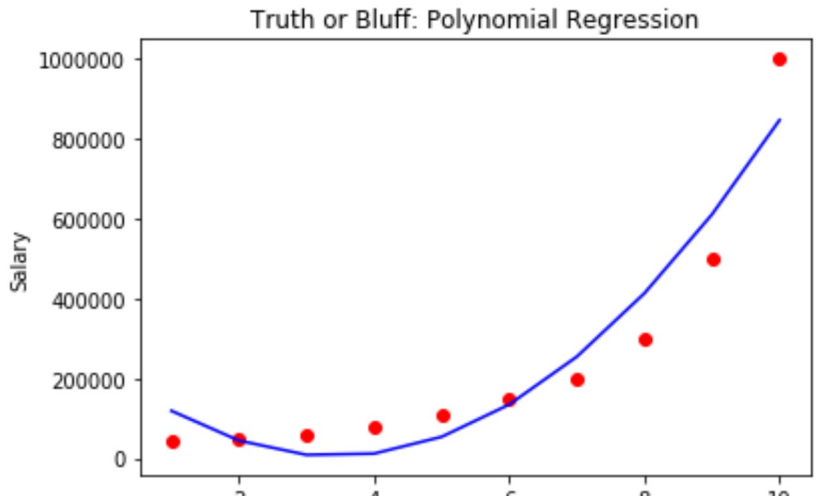
But note that while PR is linear on its coefficients, but it is a non-linear function of X.

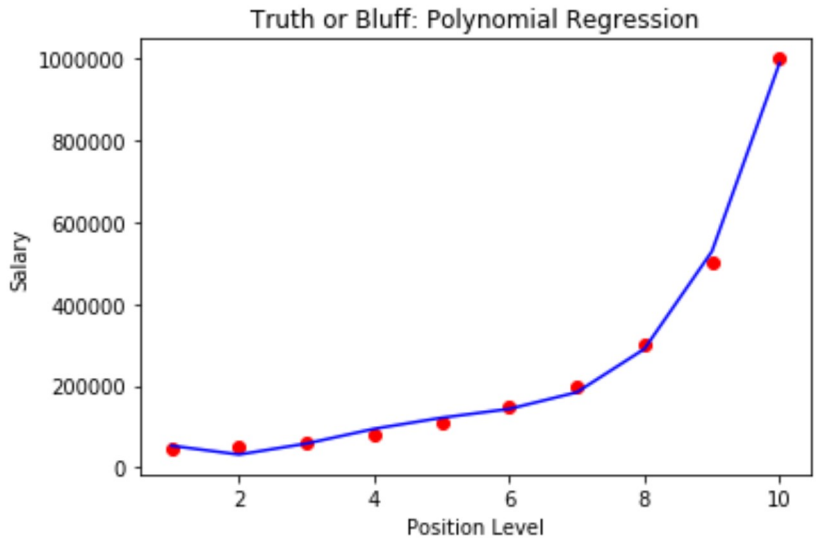
In [55]:

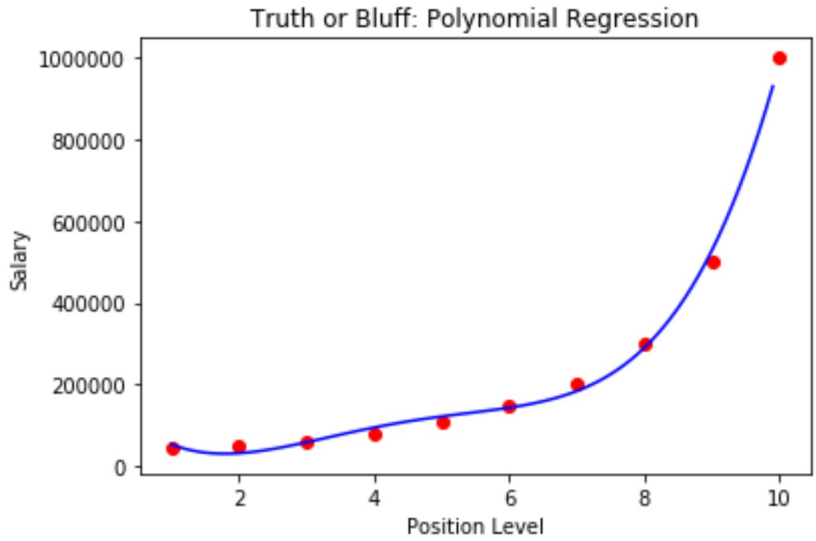
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,1:-1].values # still need : for 2D. X must always be
Y = data.iloc[:,-1].values

### 3. Modelling

# Simple Linear Regression (for testing)
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X,Y)

# Polynomial Linear Regression
from sklearn.preprocessing import PolynomialFeatures # to create poly
poly_reg = PolynomialFeatures(degree = 2) # max is X^2
X_poly = poly_reg.fit_transform(X)
lin_reg2 = LinearRegression()
lin_reg2.fit(X_poly, Y)

### 4. Visualisations

# Simple Linear Regression
plt.scatter(X,Y, color='red')
plt.plot(X,lin_reg.predict(X),color = 'blue')
plt.title("Truth or Bluff: Linear Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
# red = real, blue = predicted. prediction is far-off.

# Polynomial Linear Regression
plt.scatter(X,Y, color='red')
plt.plot(X,lin_reg2.predict(X_poly),color = 'blue') # note which ones
plt.title("Truth or Bluff: Polynomial Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()

# Polynomial Linear Regression (when up to X^4)
poly_reg4 = PolynomialFeatures(degree=4)
X_poly4 = poly_reg4.fit_transform(X)
```

```
lin_reg4 = LinearRegression()
lin_reg4.fit(X_poly4,Y)

plt.scatter(X,Y, color='red')
plt.plot(X,lin_reg4.predict(X_poly4),color = 'blue') # note which one
plt.title("Truth or Bluff: Polynomial Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()

# Smoother Curve for X^4 where position levels are continuous instead
X_grid = np.arange(min(X),max(X), 0.1)
X_grid = X_grid.reshape((len(X_grid),1))
plt.scatter(X,Y, color='red')
plt.plot(X_grid,lin_reg4.predict(poly_reg4.fit_transform(X_grid)),col
plt.title("Truth or Bluff: Polynomial Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()

### 5. Predictions

print(f'LR Prediction: {lin_reg.predict([[6.5]])}')
print(f'Poly LR Prediction X^4: {lin_reg4.predict(poly_reg4.fit_trans
poly_reg4.fit_transform([[6.5]])
```





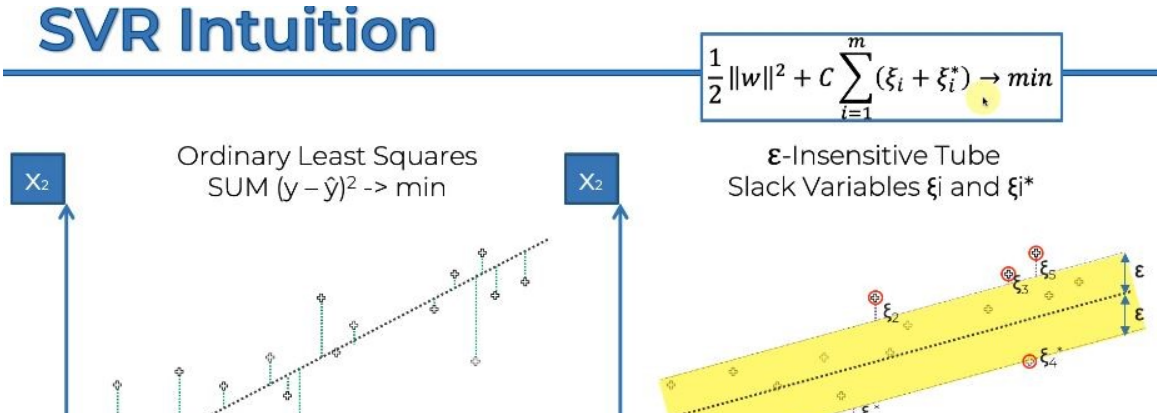


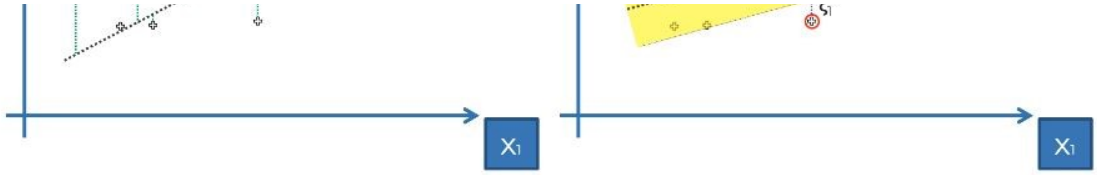



```
LR Prediction: [330378.78787879]
Poly LR Prediction X^4: [158862.45265153]
```

Out[55]:
```
array([[1.0000000e+00, 6.5000000e+00, 4.2250000e+01, 2.7462500e+02,
        1.7850625e+03]])
```

Chapter 5 - Support Vector Regression SVR





This is just a simple LINEAR SVR model. There are other types to be covered later.

Instead of just a single line, there is a **tube** with a **width of epsilon** measured vertically. It is called the **epsilon-insensitive tube**, which means that any points in the dataset that fall inside the tube will not be disregarding the error. Think of it as allowing a margin of error, and any data point inside it we will consider it as not having any error. Hence, it gives some sort of buffer to the model.

For the data points outside the tube, their error is their distance from the tube - ε* if below the tube and ε if above it.

The name **SVR** is because the outside data points are **support vectors** that dictate how the tube is created. They are supporting the structure / formation of the tube.

Kernels

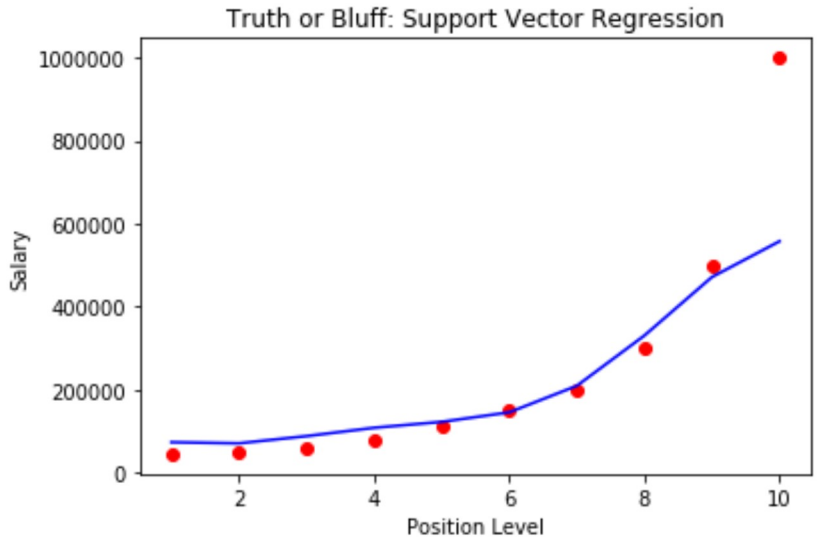
There are linear kernels that learn linear relationships, and nonlinear kernels (RBF radial basis) that learn nonlinear relationships. See here (https://data-flair.training/blogs/svm-kernel-functions/). We will be using the RBF model which is the most popular version.

Application

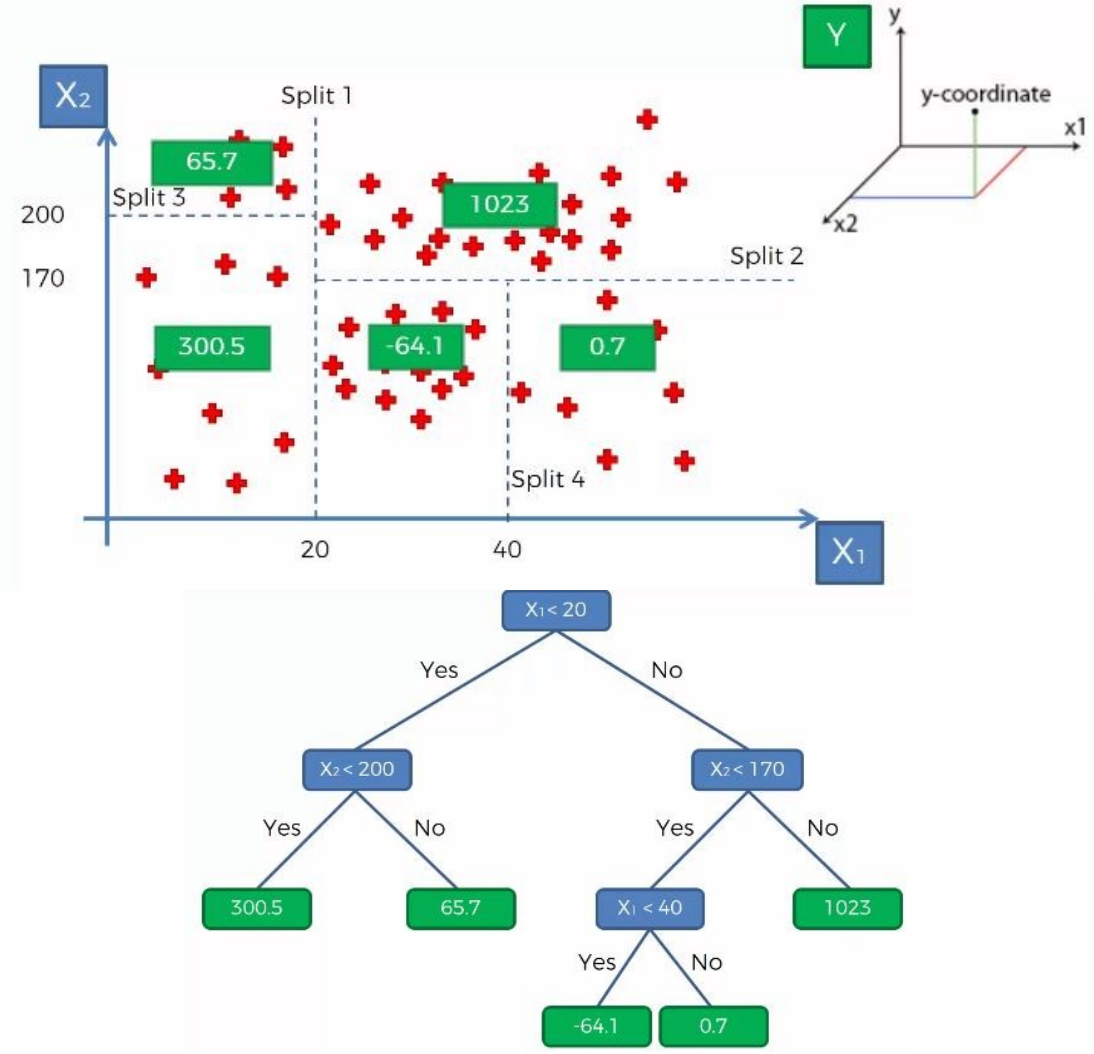
- We have to apply feature scaling. SVR model has no explicit equation of Y wrt Xs. There are not those coefficients multiplying each X and therefore no compensation of lower values for features that take high values. In fact, the SVR model has an implicit equation of Y and Xs where there are no such coefficients around.
- AKA if there is only an implicit equation btw Y and Xs, then probably need features scaling.
- We also feature scale Y here. Previously we didn't do it because it was just 0 and 1. Furthermore, **explicit equations have parameters that will take care of the high feature values by reducing their value**. Now we do it here because **Y is extremely high compared to X** AND **it is an implicit equation**.
- Feature scaling is done separately on X and Y because they are separate datasets. Unless you merge and do it, then unmerge.

In [38]:
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,1:-1].values # still need : for 2D. X must always be
Y = data.iloc[:,-1].values
```

```
Y = Y.reshape(len(Y),1) # Y must be 2D to do feature scaling

### 3. Feature Scaling

from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X = sc_X.fit_transform(X)
sc_Y = StandardScaler()
Y = sc_Y.fit_transform(Y)

### 4. Modelling

from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf') # there are many types of SVR kernels
regressor.fit(X,Y.ravel()) # ravel() turns it into a 1D array

## 5. Predictions (inverse scaling)
regressor.predict([[6.5]]) # this is WRONG! cos you must first scale
y_predicted = regressor.predict(sc_X.transform([[6.5]])) # this is ri
sc_Y.inverse_transform(y_predicted)

### 6. Visualisation

plt.scatter(sc_X.inverse_transform(X),sc_Y.inverse_transform(Y), colo
plt.plot(sc_X.inverse_transform(X),sc_Y.inverse_transform(regressor.p
plt.title("Truth or Bluff: Support Vector Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()

# Smoother Curve
X_grid = np.arange(min(sc_X.inverse_transform(X)),max(sc_X.inverse_tr
X_grid = X_grid.reshape((len(X_grid),1))
plt.scatter(sc_X.inverse_transform(X),sc_Y.inverse_transform(Y), colo
plt.plot(X_grid,sc_Y.inverse_transform(regressor.predict(sc_X.transfo
plt.title("Truth or Bluff: Support Vector Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
```



Chapter 6 - Decision Tree Regression



Ignoring Y, we will first create a scatterplot based on the Xs. We will then slowly split the dataset based on **information entropy** (math not covered here). Each split is only for the dataset it covers, ie Split 4 is only splitting dataset that is for X2 < 170. The splits will keep on carrying on until a terminal leaf hits some yardstick (eg each terminal leaf must have at least 5% of total population) and then the splits will stop.

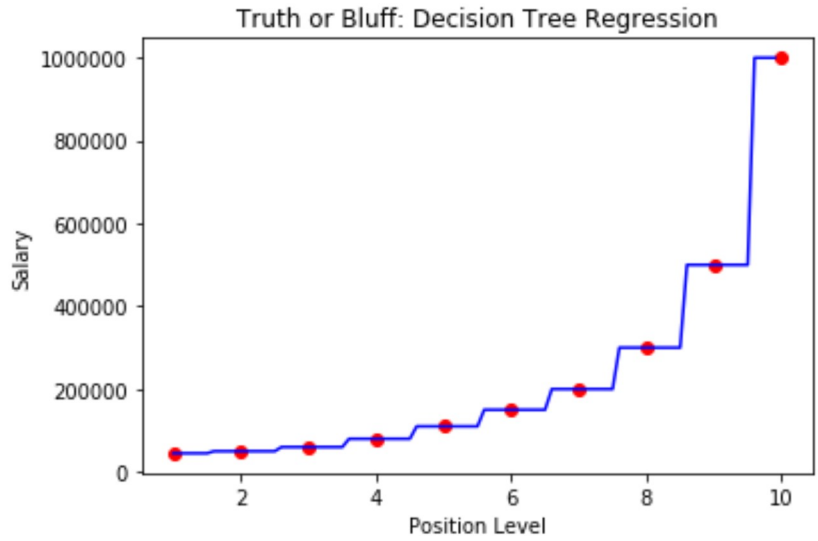
To predict a new value, we will take its X values and insert it into a terminal leaf. Its Y-value will then be the average of that leaf node's Y-value.

Note: No need feature scaling because there are no equations we are working here, just splits. Still works with original scaling of features.

Note 2: Decision trees are better for models with many features. With just one feature, it doesn't do as well.

In [17]:
```
1 ### 1. Libraries
2
```

```
 3 import numpy as np
 4 import matplotlib.pyplot as plt
 5 import pandas as pd
 6
 7 ### 2. Dataset
 8
 9 data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
10 X = data.iloc[:,1:-1].values # still need : for 2D. X must always be
11 Y = data.iloc[:,-1].values
12
13 ### 3. Modelling
14
15 from sklearn.tree import DecisionTreeRegressor # to predict a continu
16 regressor = DecisionTreeRegressor(random_state=0) # set seed
17 regressor.fit(X,Y)
18
19 ### 4. Predictions
20
21 regressor.predict([[6.5]])
22
23 ### 5. Visualisation
24
25 X_grid = np.arange(min(X),max(X),0.1)
26 X_grid = X_grid.reshape(len(X_grid),1)
27 plt.scatter(X,Y,color='red')
28 plt.plot(X_grid,regressor.predict(X_grid),color='blue')
29 plt.title("Truth or Bluff: Decision Tree Regression")
30 plt.xlabel("Position Level")
31 plt.ylabel("Salary")
32 plt.show()
```



Chapter 7 - Random Forest

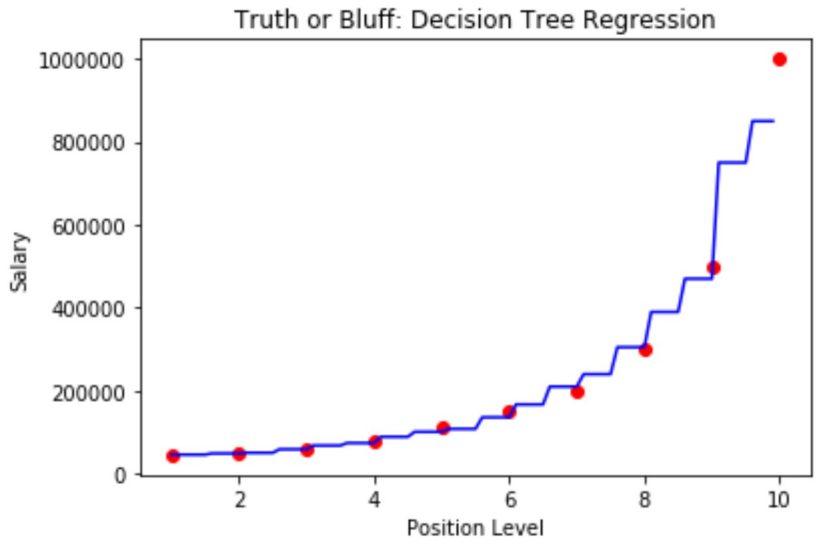
RF is a version of **ensemble learning**. There are many other versions of ensemble learning such as gradient boosting etc. Ensemble learning is when you take multiple algorithms OR the same algorithm multiple times and you put them together to make something much more powerful than the original.

- Step 1: Pick randomly K data points from the train set.
- Step 2: Build the decision tree associated to these K data points.
- Step 3: Choose Ntree, the number of trees you want to build. Repeat Steps 1 and 2. Hence you build a lot of regression decision trees.
- Step 4: For a new data point, make each one of your Ntree trees predict the value of Y, and assign the new data point the average of all the predicted Y values.

This improves the accuracy of the prediction as you are taking the average of many predictions. Also it is more stable, as any changes in your data set could normally impact one tree but less so on an entire forest.

In [21]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,1:-1].values # still need : for 2D. X must always be
Y = data.iloc[:,-1].values

### 3. Modelling

from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators = 10, random_state=0)
regressor.fit(X,Y)

### 4. Predictions

regressor.predict([[6.5]])

### 5. Visualisation

X_grid = np.arange(min(X),max(X),0.1)
X_grid = X_grid.reshape(len(X_grid),1)
plt.scatter(X,Y,color='red')
plt.plot(X_grid,regressor.predict(X_grid),color='blue')
plt.title("Truth or Bluff: Decision Tree Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
```



Regression: Summary Thus Far

In [53]:

```
### 1. Libraries
```

```
 2
 3 import numpy as np
 4 import matplotlib.pyplot as plt
 5 import pandas as pd
 6
 7 ### 2. Dataset
 8
 9 data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
10 X = data.iloc[:,:-1].values
```

In [37]:

```
 1 ##### Multiple Linear Regression #####
 2
 3 ### 3. Train Test Split
 4 from sklearn.model_selection import train_test_split
 5 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
 6
 7 ### 4. Modelling
 8 from sklearn.linear_model import LinearRegression
 9 regressor = LinearRegression()
10 regressor.fit(X_train, y_train)
11
12 ### 5. Predictions
13 y_pred = regressor.predict(X_test)
14 np.set_printoptions(precision=2)
15 print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(l
16
17 ### 6. Evaluation
18 from sklearn.metrics import r2_score
```

```
[[2.25 2.  ]
 [2.03 2.  ]
 [3.52 4.  ]
 [3.7  4.  ]
 [1.89 2.  ]
 [2.09 2.  ]
 [2.26 2.  ]
 [3.94 4.  ]
 [2.07 2.  ]
 [1.94 2.  ]]
```

Out[37]: 0.8354489501242135

In [39]:

```
 1 ##### Polynomial Linear Regression #####
 2
 3 ### 3. Train Test Split
 4 from sklearn.model_selection import train_test_split
 5 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
 6
 7 ### 4. Modelling
 8 from sklearn.preprocessing import PolynomialFeatures
 9 from sklearn.linear_model import LinearRegression
10 poly_reg = PolynomialFeatures(degree = 4)
11 X_poly = poly_reg.fit_transform(X_train)
12 regressor = LinearRegression()
13 regressor.fit(X_poly, y_train)
14
```

```
### 5. Predictions
y_pred = regressor.predict(poly_reg.transform(X_test))
np.set_printoptions(precision=2)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(l
### 6. Evaluation
from sklearn.metrics import r2_score
```

```
[[2.4  2.  ]
 [2.   2.  ]
 [3.68 4.  ]
 [3.49 4.  ]
 [2.29 2.  ]
 [1.9  2.  ]
 [2.24 2.  ]
 [4.39 4.  ]
 [2.55 2.  ]
 [2.18 2.  ]]
```

Out[39]: 0.6085482039668688

In [45]:

```
##### Support Vector Regression #####

### 2. Additional Reshape (because feature scaler takes in 2D)
y = y.reshape(len(y),1)

### 3. Train Test Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =

### 4. Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
sc_y = StandardScaler()
X_train = sc_X.fit_transform(X_train)
y_train = sc_y.fit_transform(y_train)

### 5. Modelling
from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf')
regressor.fit(X_train, y_train.ravel())

### 6. Predictions
y_pred = sc_y.inverse_transform(regressor.predict(sc_X.transform(X_te
np.set_printoptions(precision=2)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(l
### 7. Evaluation
from sklearn.metrics import r2_score
```

```
[[2.1  2.  ]
 [1.91 2.  ]
 [3.97 4.  ]
```

Out[45]: 0.8671506597422487

In [47]:

```
1  ##### Decision Tree Regression #####
2
3  ### 3. Train Test Split
4  from sklearn.model_selection import train_test_split
5  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
6
7  ### 4. Modelling
8  from sklearn.tree import DecisionTreeRegressor
9  regressor = DecisionTreeRegressor(random_state = 0)
10 regressor.fit(X_train, y_train)
11
12 ### 5. Predictions
13 y_pred = regressor.predict(X_test)
14 np.set_printoptions(precision=2)
15 print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(l
16
17 ### 6. Evaluation
18 from sklearn.metrics import r2_score
```

```
[[2. 2.]
 [2. 2.]
 [4. 4.]
 [4. 4.]
 [2. 2.]
 [2. 2.]
 [2. 2.]
 [2. 4.]
 [2. 2.]
 [2. 2.]]
```

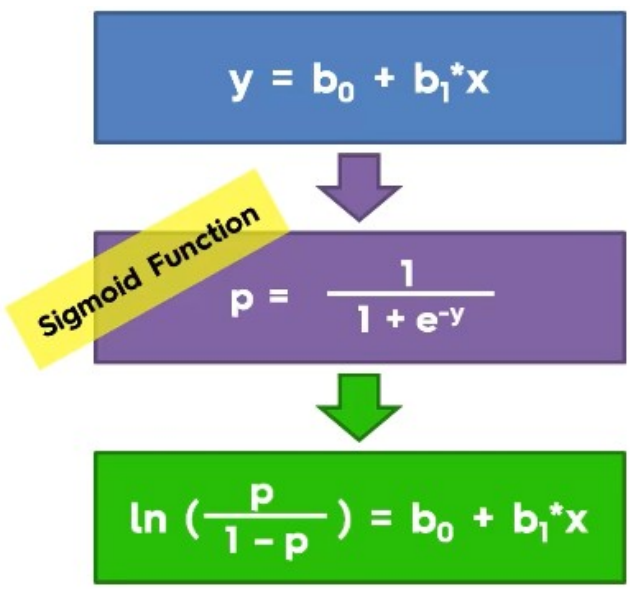
Out[47]: 0.6535632183908047

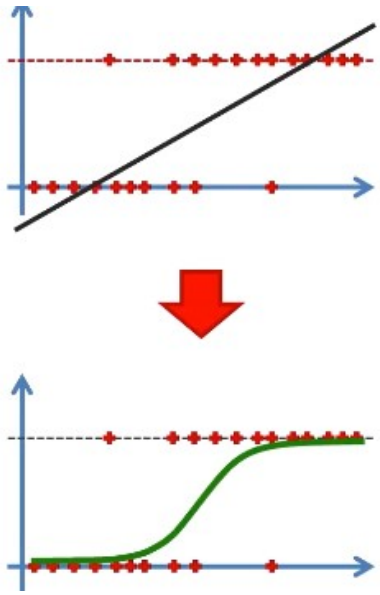
In [54]:
```
##### Random Forest #####

### 3. Train Test Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =

### 4. Modelling
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators = 10, random_state = 0
regressor.fit(X_train, y_train)

### 5. Predictions
y_pred = regressor.predict(X_test)
np.set_printoptions(precision=2)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(l

### 6. Evaluation
from sklearn.metrics import r2_score
```

```
[[2.  2. ]
 [2.  2. ]
 [4.  4. ]
 [4.  4. ]
 [2.  2. ]
 [2.  2. ]
 [2.4 2. ]
 [3.8 4. ]
 [2.  2. ]
 [2.  2. ]]
```

Out[54]: 0.8447333333333333

Chapter 8 - Logistic Regression





For logistic regression, **feature scaling is not compulsory but it improves results**.

In [41]:

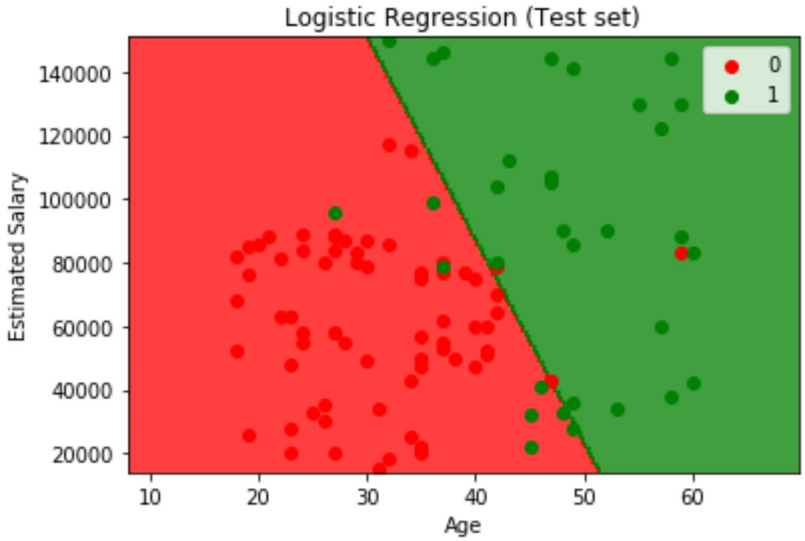
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Modelling

from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state=0)
classifier.fit(X_train, Y_train) # smaller the C, stronger the regula

### 4. Predictions

print(classifier.predict(sc.transform([[30,87000]])))

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Logistic Regression (Training set)')
```

```
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_test), Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Logistic Regression (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

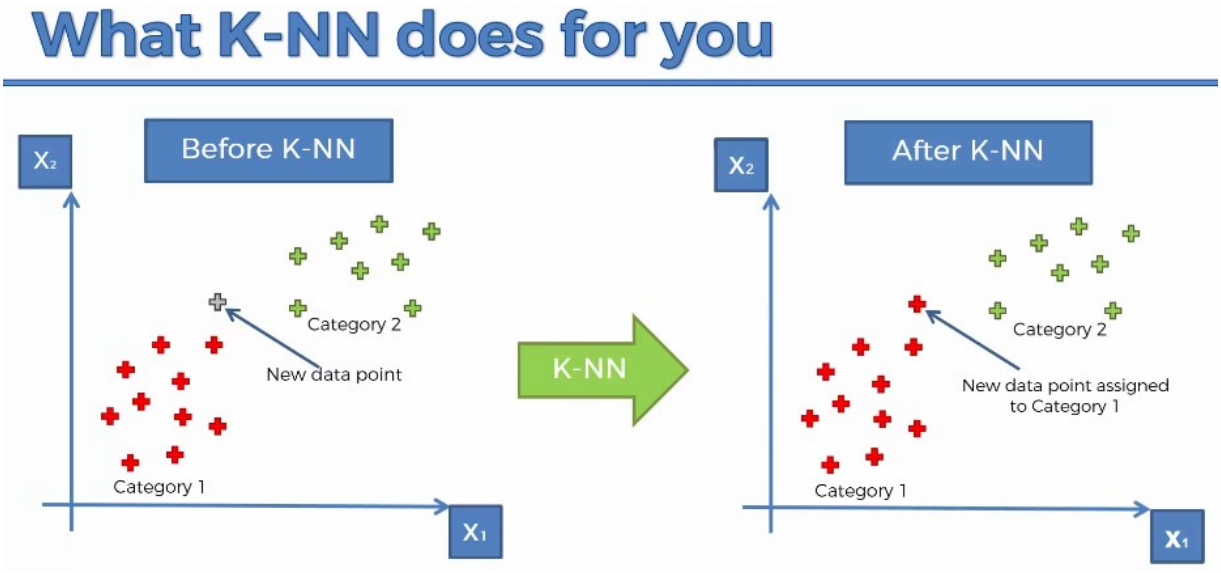
```
[0]
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [1 1]]
[[65  3]
 [ 8 24]] 0.89
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```



Chapter 9 K-Nearest Neighbour KNN



- Step 1: Choose the number K of neighbours. Common default K = 5.
- Step 2: Take the K nearest neighbours of the new data point based on Euclidean distance.
- Step 3: Among these K neighbours, count the number of data points in each category.
- Step 4: Assign the new data point to the category with the most neighbours.

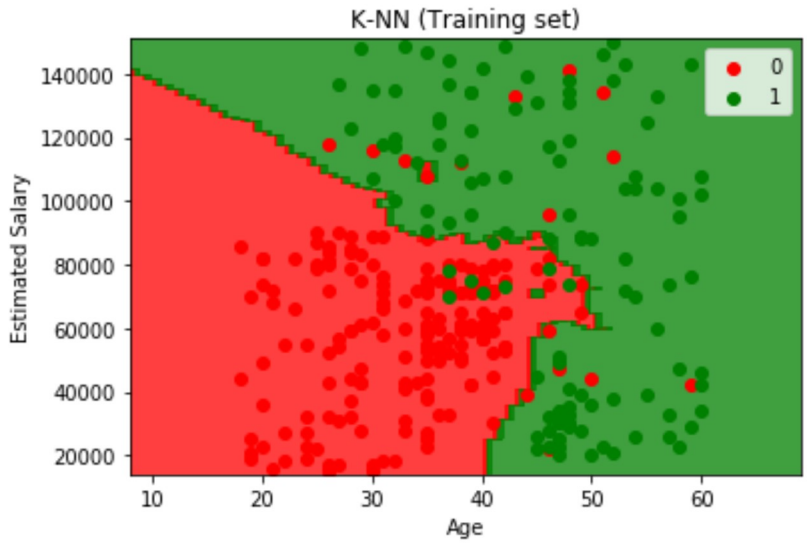
In [47]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Modelling

from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=5, metric='minkowski', 
classifier.fit(X_train, Y_train)

### 4. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('K-NN (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
```

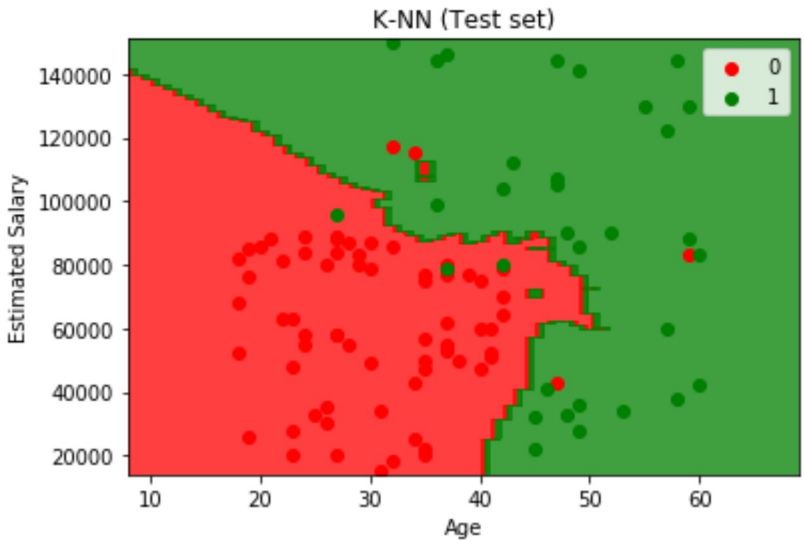
```
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_test), Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('K-NN (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

```
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [1 1]]
[[64  4]
 [ 3 29]] 0.93
```

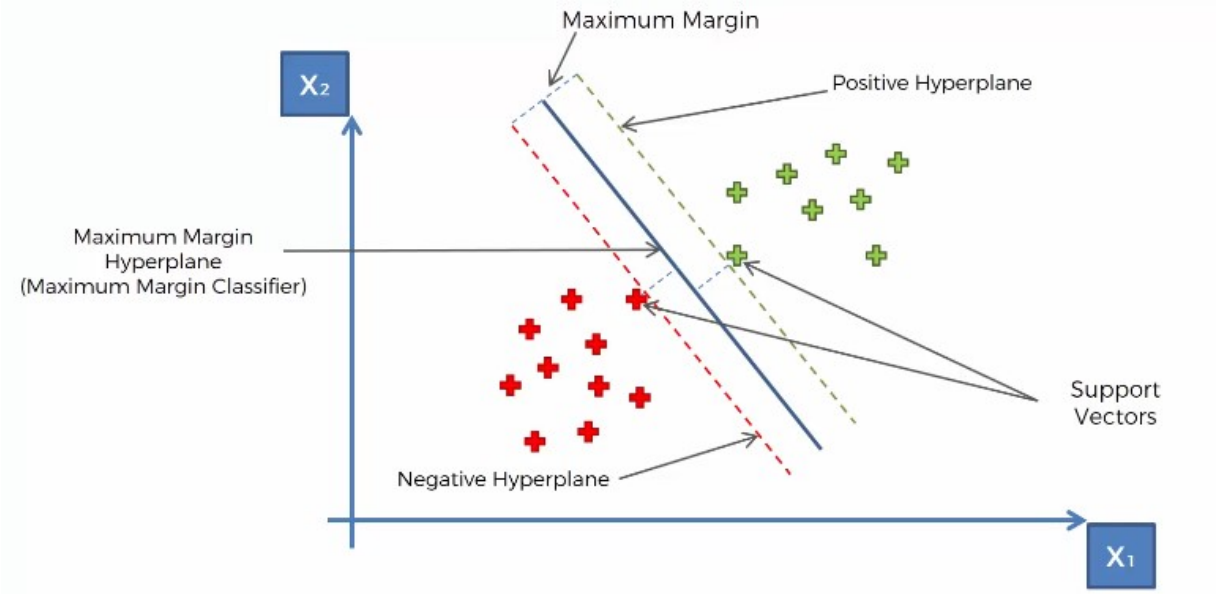
```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth matches with 'x' & 'y'.  Please use a 2-D array with a single row
if you really want to specify the same RGB or RGBA value for all point
s.
'c' argument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



```
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```



Chapter 10 - Support Vector Machine SVM



Basically it's drawing a line that separates groups, and the distance is at a **maximum margin**. Both **support vector** points are equidistant from the line. Only those two points contribute to / support the entire algorithm; the rest of the points do nothing at all.

Points are called vectors because if not in 2D space anymore, hard to visualise. In such a multidimensional space, they are called vectors.

The line is also called the **maximum margin hyperplane / maximum margin classifier**. Then arbitrarily, one line is the positive hyperplane and the other is negative hyperplane; whatever is left of the negative hyperplane is the negative category, vice versa.

The SVM is popular in the sense that it uses extreme data points that are close to each other as the support vectors. These data points are very similar, ie an oranged-color apple vs a green-colored orange. This is very different from the other ML algorithms and that is why sometimes it

In [51]:

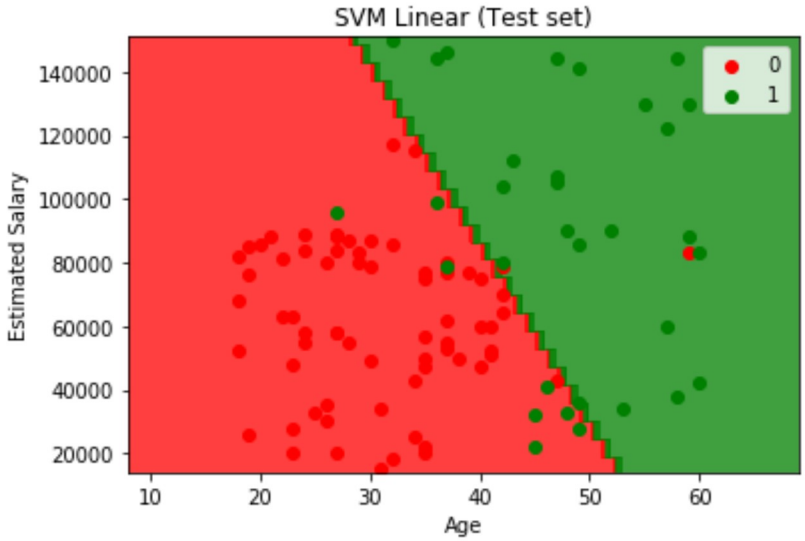
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Modelling

from sklearn.svm import SVC
classifier = SVC(kernel='linear', random_state=0) # trying out a line
classifier.fit(X_train,Y_train)

### 4. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
```

```
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('SVM Linear (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_test), Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('SVM Linear (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

```
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [1 1]]
[[66  2]
 [ 8 24]] 0.9
```

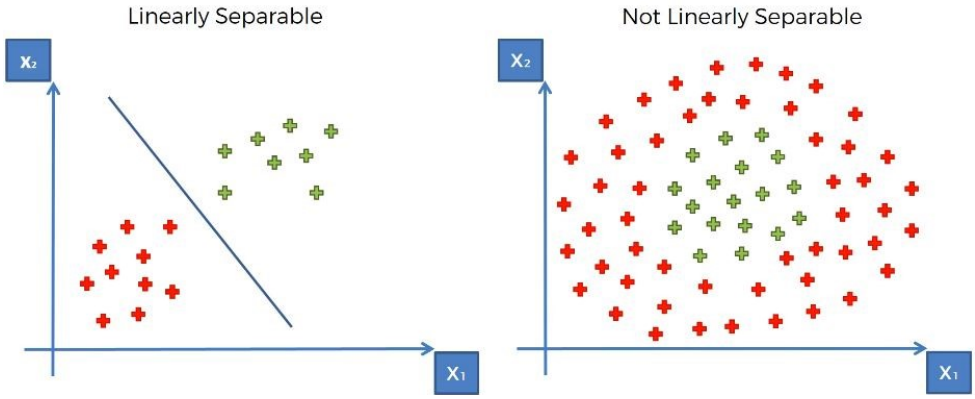
```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth matches with 'x' & 'y'.  Please use a 2-D array with a single row
if you really want to specify the same RGB or RGBA value for all point
s.
'c' argument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

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```



Chapter 11 - Kernel SVM

Sometimes, linear SVM does not work because the data is **not linearly separable**.



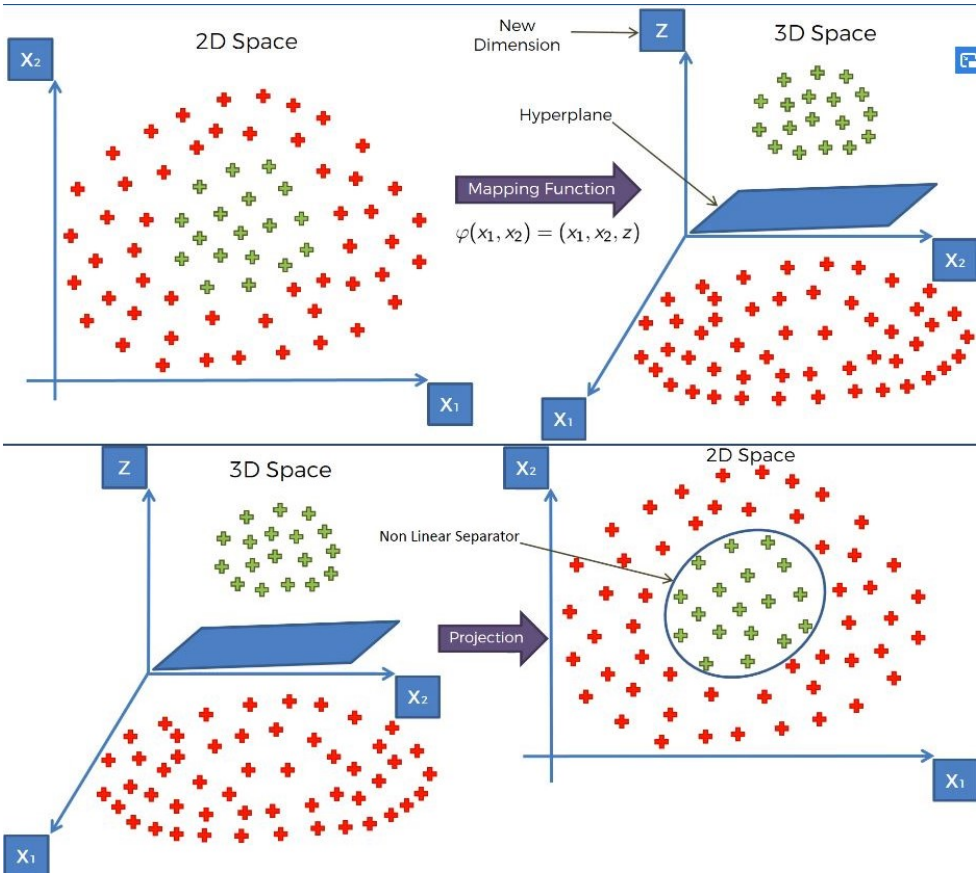
There are several ways to go about this.

Mapping into a Higher-Dimensional Space

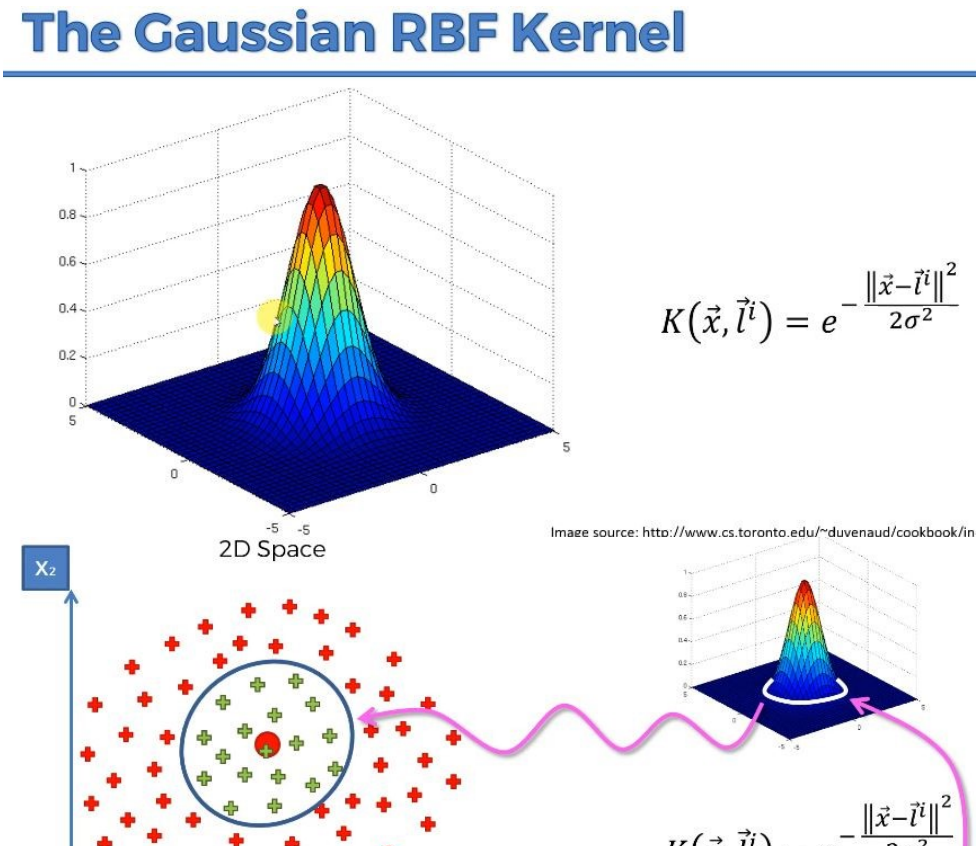
Doing so will map the nonlinear dataset and turn it into a linearly separable dataset. For an original 2D dataset, it will have a new dimension Z, with a hyperplane between it. After that, we project it back into 2D, and that hyperplane would become a circle.

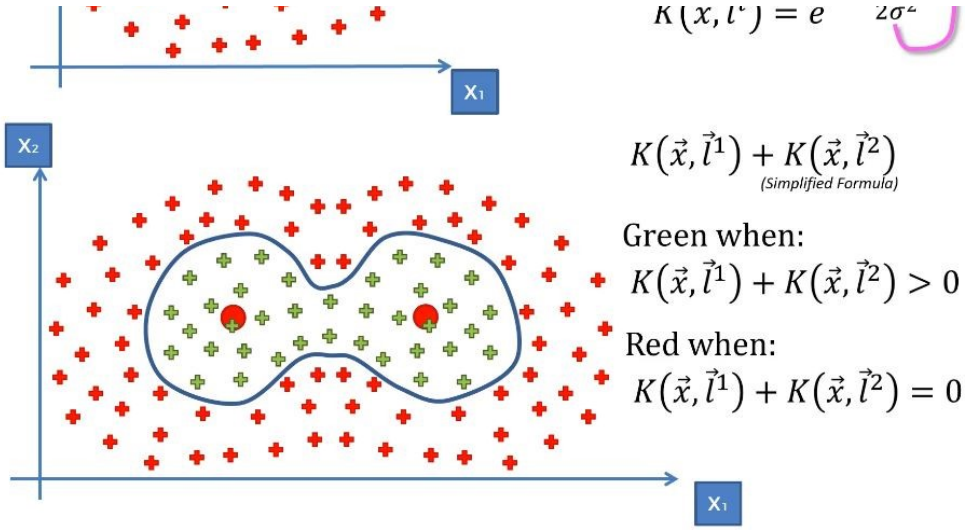
The disadvantage is that mapping into a higher dimensional space is highly compute-intensive,

which is why it's not a good strategy for big data.



The Kernel Trick



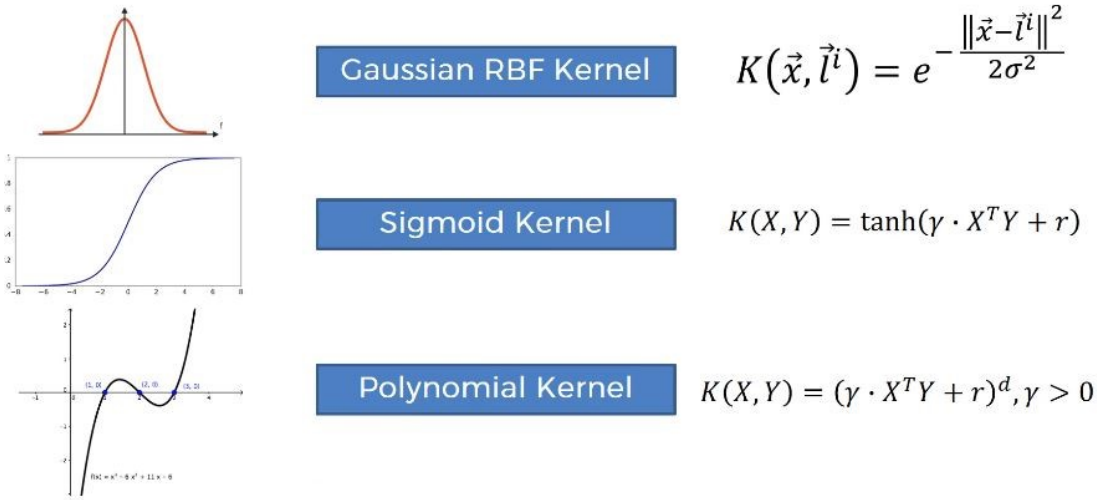


The K equation is called the **kernel function**. l is landmark. It is in the middle of the 2D space. x is any other point on that 2D space. Any vertical is the result of the equation. The top part of the equation basically means "distance from x to the landmark, squared". Sigma is some fixed parameter. If you think about it, e^0=1 which is why it's the peak, and e^-inf=0.

The optimal landmark is not necessarily the center, and it is found via some algorithm. The base of the cone (the circumference) is then projected onto the 2D dataset: anything in the circumference is assigned a value above 0 while the rest are close to 0. The sigma's role is to define how wide the circumference is: the higher it is, the larger is the circle.

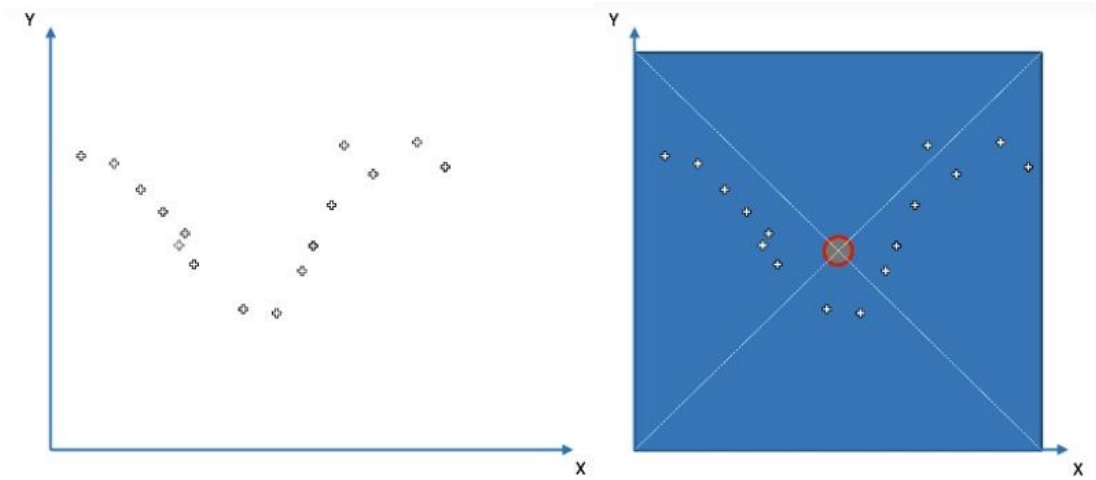
Hence the kernel function allows you to separate the points by adjusting the function itself, and sigma. All of this without going into a higher dimensional space. You can even make even more complex equations which do not interfere with each other.

Types of Kernel Functions

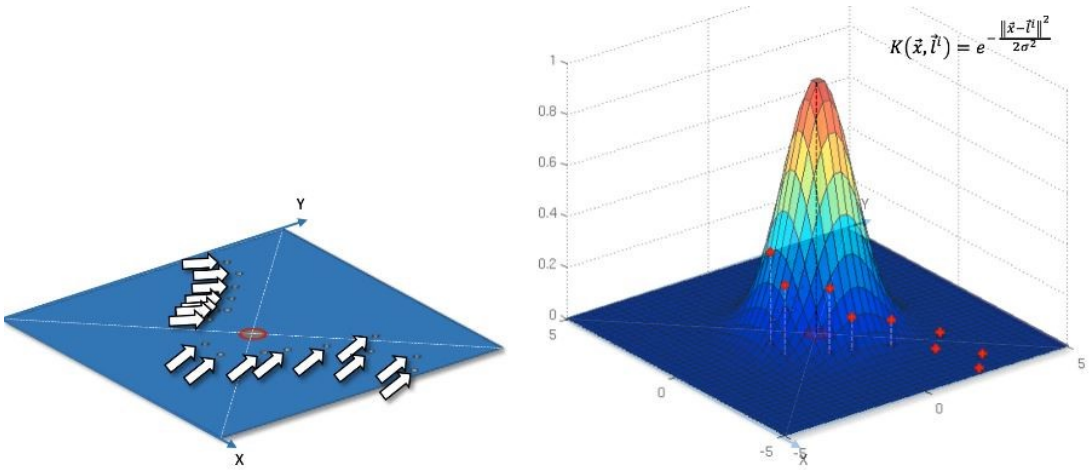


- Gaussian RBF Kernel: for when data in the middle are different.
- Sigmoid Kernel: for when data on the right side are different.
- Polynomial Kernel: lol
- Read more here (https://thatcher.dev/MLKernels.jl/dev/) and here (https://github.com/trthatcher/MLKernels.jl).

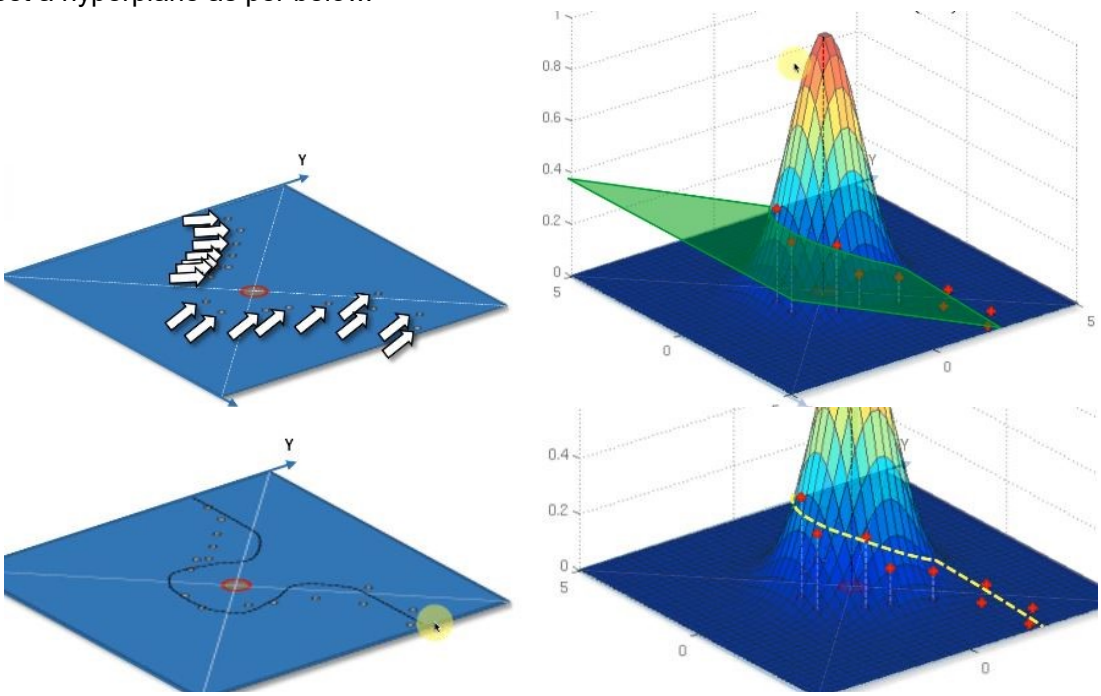
Showcase



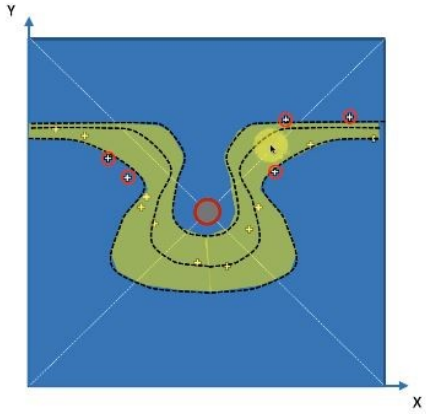
We got the above and want to use SVR. But a linear SVR doesn't work. We can project into 3D and fit it with a kernel function.

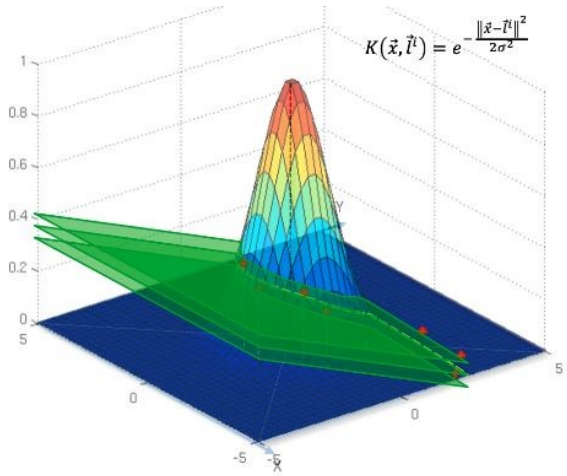


We set a hyperplane as per below.



Then instead of an epsilon tube, we have an epsilon space. The intersections between the outer planes of the epsilon space and the kernel functions are the support vectors.





Note: due to the kernel trick, you don't really map into a new dimension as the above pictures

In [7]:

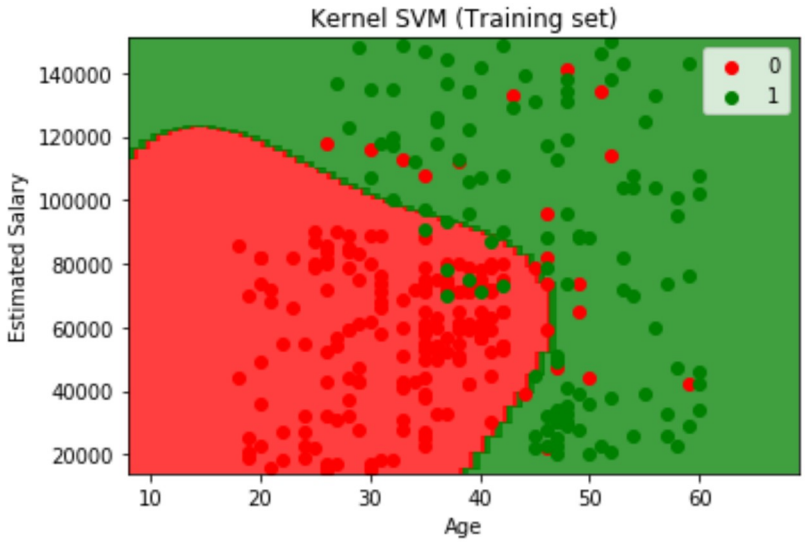
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Modelling

from sklearn.svm import SVC
classifier = SVC(kernel='rbf', random_state=0)
classifier.fit(X_train,Y_train)

### 4. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
```

```
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Kernel SVM (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_test), Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Kernel SVM (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

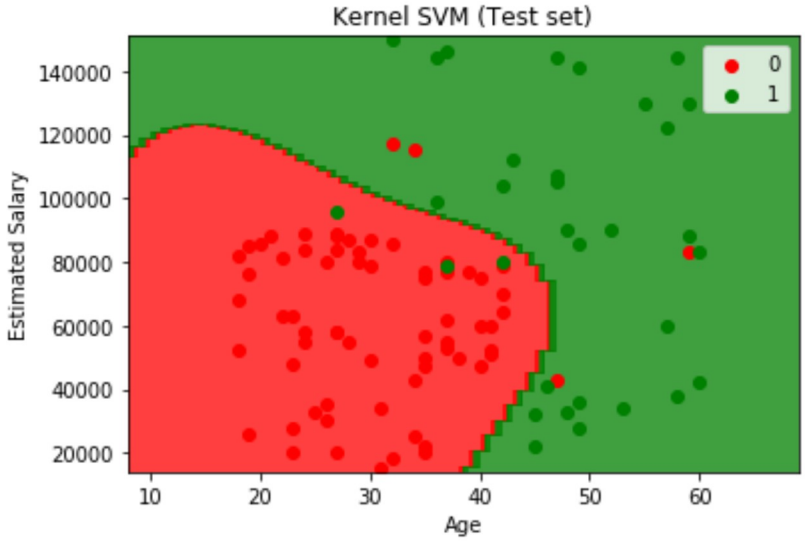
```
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [1 1]]
[[64  4]
 [ 3 29]] 0.93
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
```

```
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```




```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```



Chapter 12 - Bayes Theorem

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

Mach1: 30 wrenches / hr -> P(Mach1) = 30/50 = 0.6
Mach2: 20 wrenches / hr -> P(Mach2) = 20/50 = 0.4

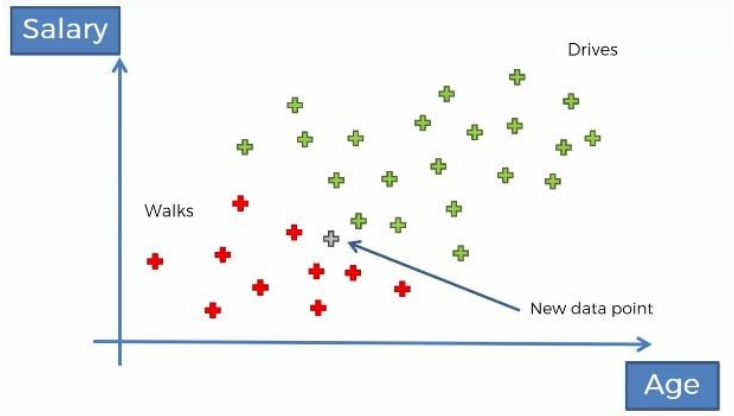
Out of all produced parts:
We can SEE that 1% are defective -> P(Defect) = 1%

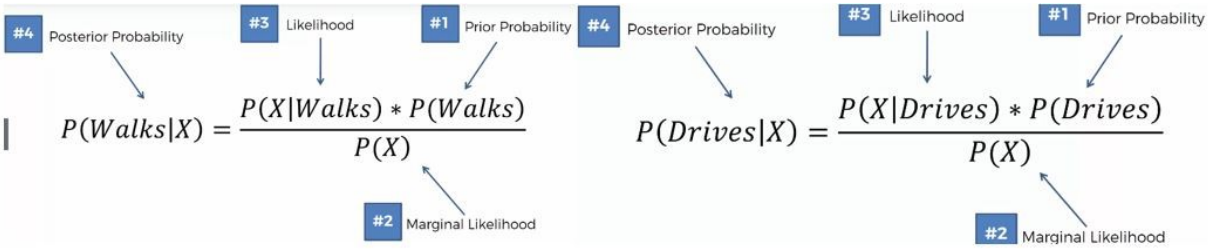
Out of all defective parts:
We can SEE that 50% came from mach1 -> P(Mach1 | Defect) = 50%
And 50% came from mach2 -> P(Mach2 | Defect) = 50%

Question:
What is the probability that a part
produced by mach2 is defective = ? -> P(Defect | Mach2) = ?
Mach1: 30 wrenches / hr -> P(Mach2) = 20/50 = 0.4
Mach2: 20 wrenches / hr -> P(Defect) = 1%
Out of all produced parts: -> P(Mach2 | Defect) = 50%
We can SEE that 1% are defective -> P(Defect | Mach2) = ?
Out of all defective parts:
We can SEE that 50% came from mach1
And 50% came from mach2
Question:
What is the probability that a part
produced by mach2 is defective = ?

$$P(\text{Defect} \mid \text{Mach2}) = \frac{0.5 * 0.01}{0.4} = 0.0125 = 1.25\%$$

BT in Machine Learning





Marginal Probability Likelihood

For **marginal likelihood**, draw a circle (size depends on formula) where the new point would be. All points in that circle will be deemed similar to the new point. Hence, the no. of points in that circle out of total will be the marginal probability. AKA it is the probability of any new random variable being in that circle.

For **likelihood**, it is similar. Look at the circle again. You are asking what is the probability that the new point belongs to one of the classes. In the example, P(X|Walks) means we ignore all Drives, then count no. of red dots out of all the red dots only.

In the end, choose the one with the higher percentage.

Extra Notes

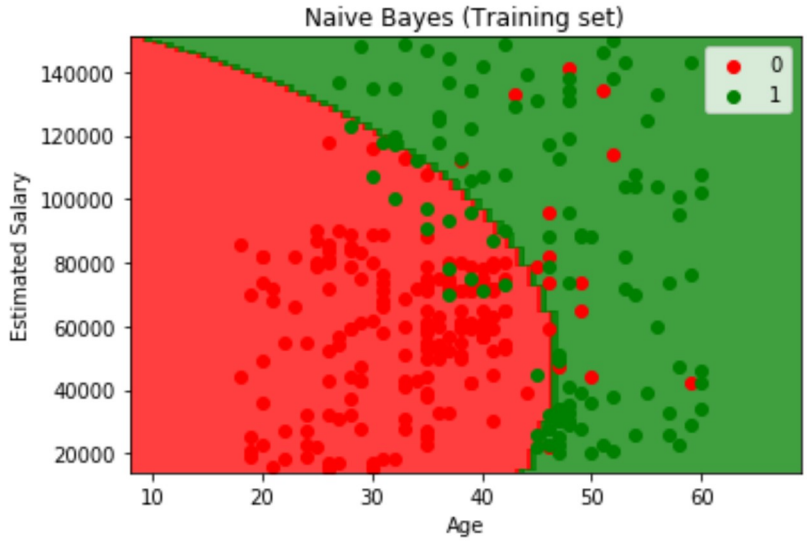
- **Why "Naive"?** Because it uses Bayes' Theorem which has its own assumptions, and these assumptions are naive. For eg, it assumes features are independent, which is extremely hard in the real world (eg "age" and "salary" probably quite interlinked).
- Calculating marginal likelihood P(X) is not *necessary* when comparing two features. Because both denominators are P(X) so essentially it is just comparing the top. However when there are more features, this cannot apply.
- The above also applies if only 2 classes. If you calculate one, you know the other. But if > 2, need to calculate all.

In [6]:
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Modelling

from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB() # we use the Gaussian NB one
```

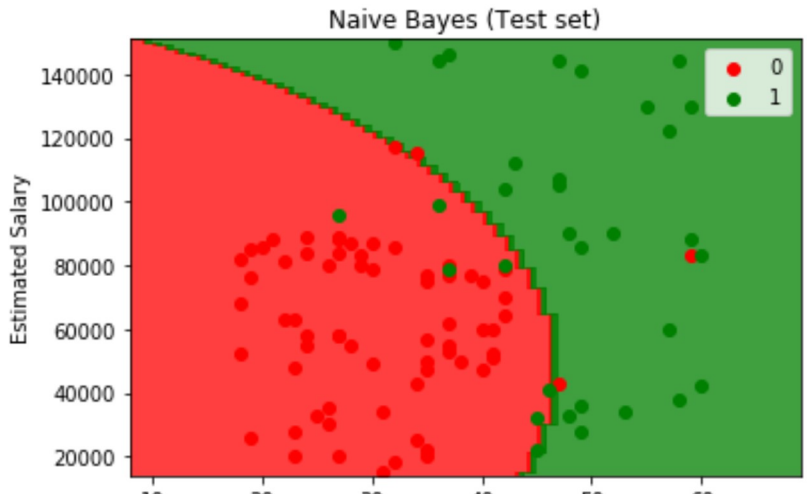
```
classifier.fit(X_train,Y_train)

### 4. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Naive Bayes (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_test), Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Naive Bayes (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

```
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
```

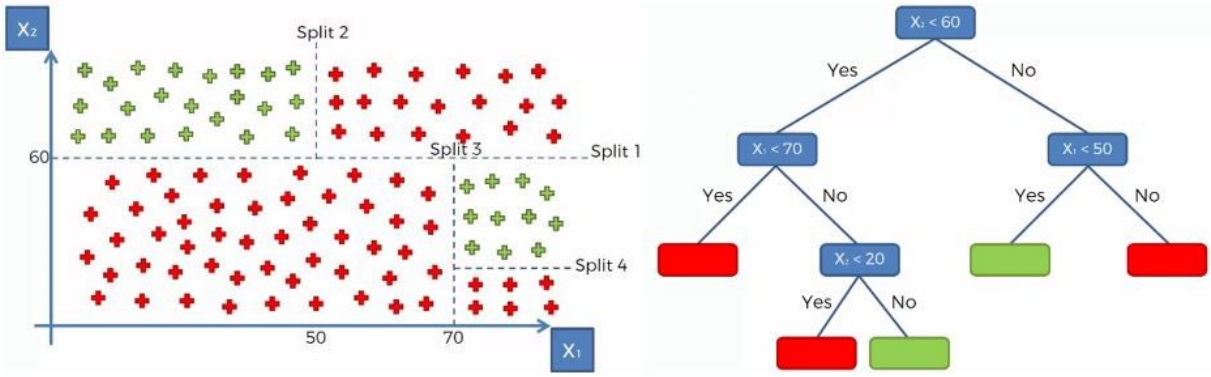
```
 [1 1]]
[[65  3]
 [ 7 25]] 0.9
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth matches with 'x' & 'y'.  Please use a 2-D array with a single row
if you really want to specify the same RGB or RGBA value for all point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```




```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```



Chapter 13 - Decision Trees (CART)



Splits are made at points that best minimises information entropy.

Decision trees are not as popular now and instead have been overtaken by Random Forest, Gradient Boosting etc.

In [5]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Modelling

from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion='entropy', random_state
classifier.fit(X_train,Y_train)

### 4. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

```

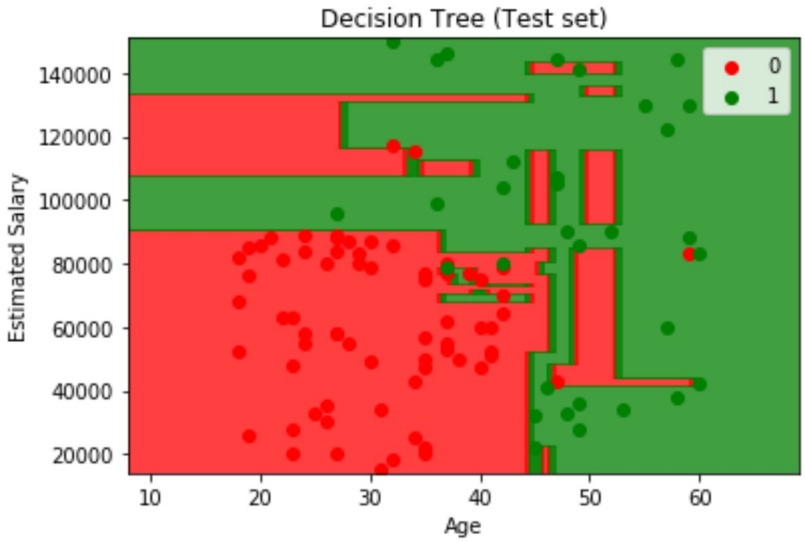
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Decision Tree (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_test), Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Decision Tree (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

```
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [1 1]]
[[62  6]
 [ 3 29]] 0.91
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```


```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```



Chapter 14 - Random Forest

Ensemble learning is when you use many ML algorithms and put them together to create one bigger ML algorithm. RF uses this by combining many smaller trees into one forest.

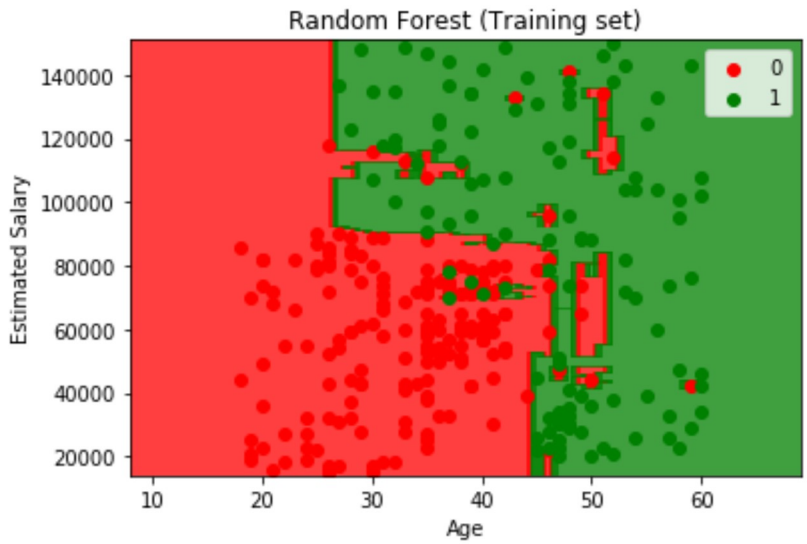
In [12]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Modelling

from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators=40, criterion='gini'
classifier.fit(X_train,Y_train)

### 4. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Random Forest (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
```

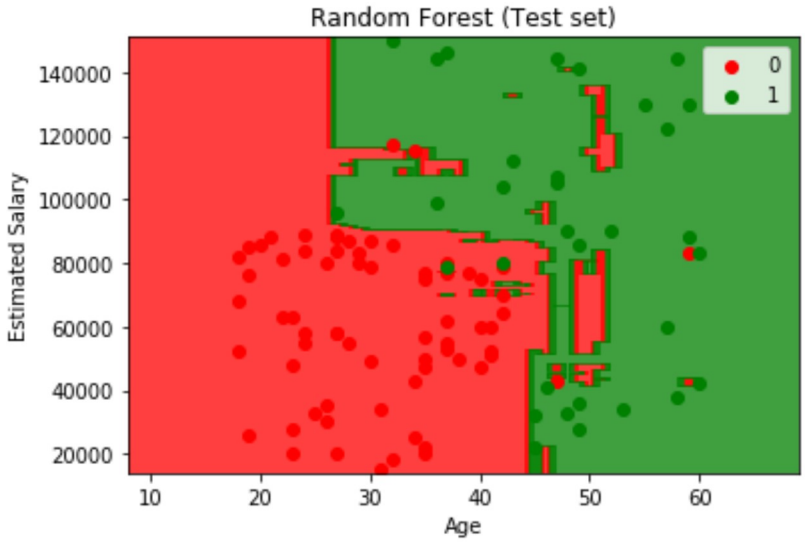
```
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_test), Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
                     np.arange(start = X_set[:, 1].min() - 1000, stop
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.rav
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Random Forest (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

```
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [1 1]]
[[64  4]
 [ 2 30]] 0.94
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```

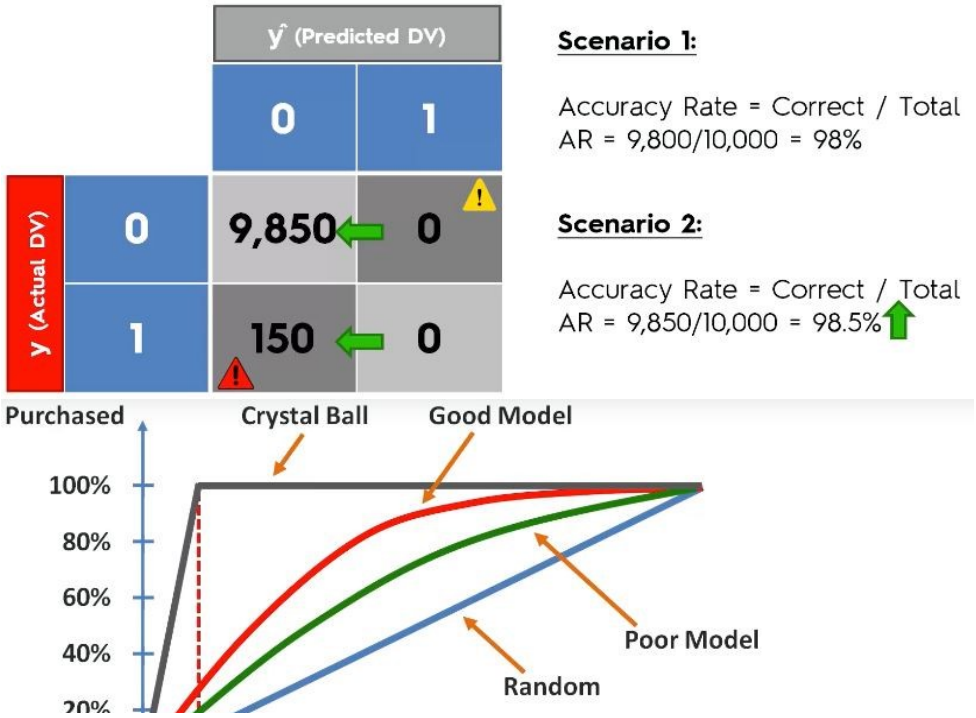


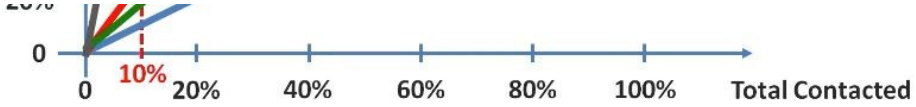
```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth matches with 'x' & 'y'.  Please use a 2-D array with a single row
if you really want to specify the same RGB or RGBA value for all point
s.
'c' argument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



Classification: Further Notes

Accuracy Paradox

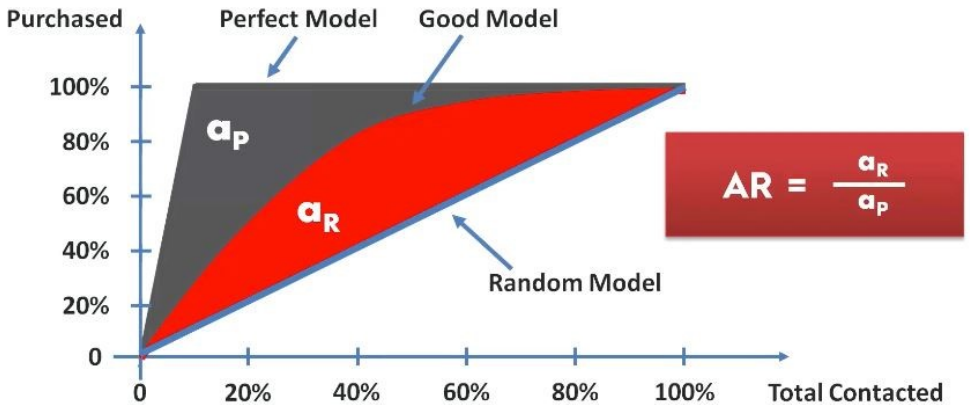




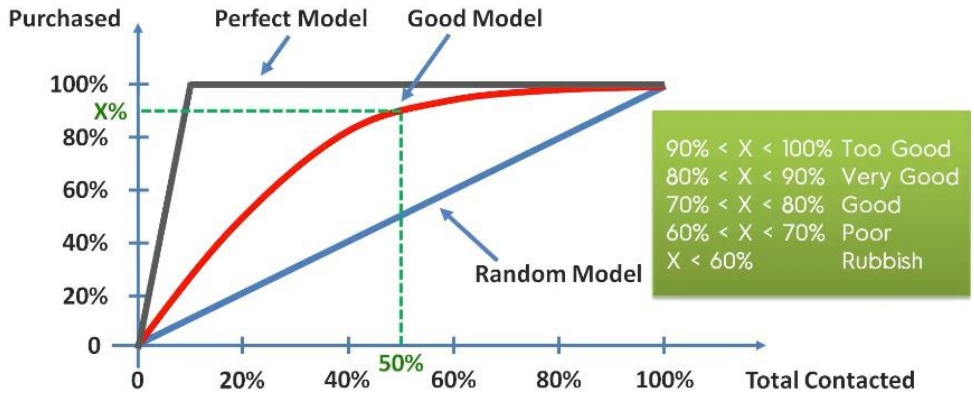
Accuracy is not a good enough parameter on its own. Use the **cumulative accuracy profile (CAP)** to help analyse. The bigger the gap between the blue and red line, the better.

The red line is basically at first aiming all the confirm people first. That is why huge growth.

Note that the CAP curve =/= the ROC curve (receiver operating characteristic). Many people see similar then think they are similar.



To calculate the accuracy ratio, you take the area under the perfect model (everything until the blue line) divided by the area under the red model. The number will be between 0 and 1. The closer to 1, the better. However, while the method is calculable using statistical tools, it is hard to quantify by looking at it visually.



This is the other method. Count from 50%. But note that if "too good", there may be overfitting, or **forward looking variables**.

Which Model?

Classification Model	Pros	Cons
Logistic Regression	Probabilistic approach, gives informations about statistical significance of features	The Logistic Regression Assumptions
K-NN	Simple to understand, fast and efficient	Need to choose the number of neighbours k

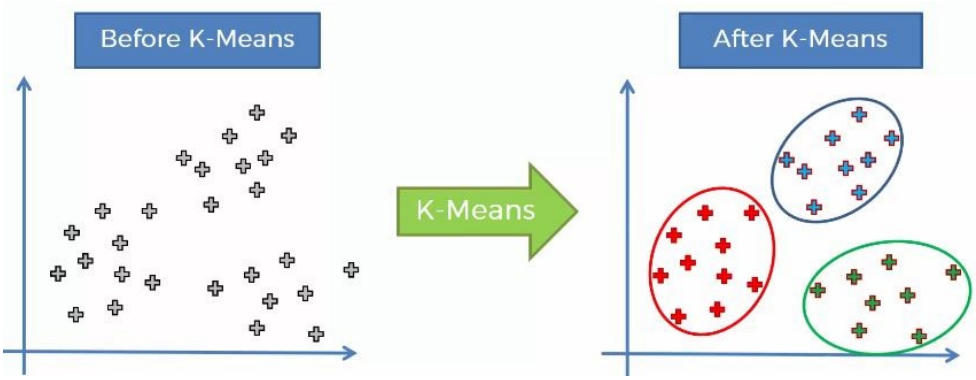
SVM	Performant, not biased by outliers, not sensitive to overfitting	Not appropriate for non linear problems, not the best choice for large number of features
Kernel SVM	High performance on nonlinear problems, not biased by outliers, not sensitive to overfitting	Not the best choice for large number of features, more complex
Naive Bayes	Efficient, not biased by outliers, works on nonlinear problems, probabilistic approach	Based on the assumption that features have same statistical relevance
Decision Tree Classification	Interpretability, no need for feature scaling, works on both linear / nonlinear problems	Poor results on too small datasets, overfitting can easily occur
Random Forest Classification	Powerful and accurate, good performance on many problems, including non linear	No interpretability, overfitting can easily occur, need to choose the number of trees

- If linear, use logistic or SVM.
- If nonlinear, use KNN, NB, decision tree, or RF.

From a business POV:

- If you want to rank probabilities that your customer will buy, use logistic (linear) or NB (nonlinear) as they can provide probabilities.
- Segmentations of customers can use SVM.
- DT for clear interpretation of model results.
- RF for high performance model but less on interpretation.

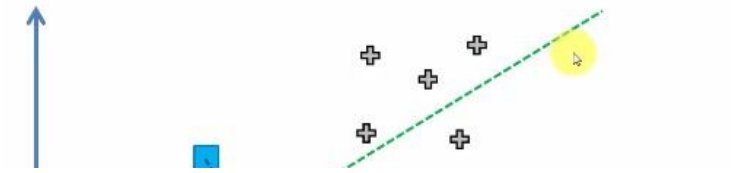
Chapter 14 - K Means Clustering

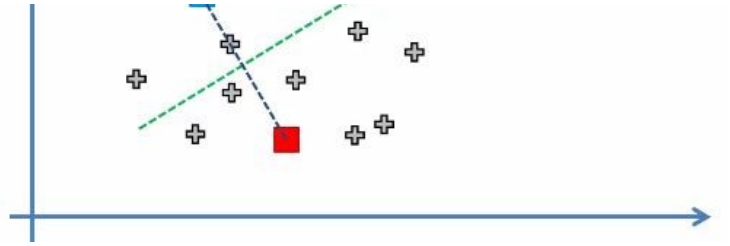


Clustering is different from classification because you don't know what you are looking for - you are simply segmenting them.

- Step 1: Choose K number of clusters
- Step 2: Select random K points as centroids (no need to be from dataset)
- Step 3: Assign each data point to the closest centroid
- Step 4: Compute and place the new centroid of each cluster
- Step 5: Repeat from step 3 until no more reassignment

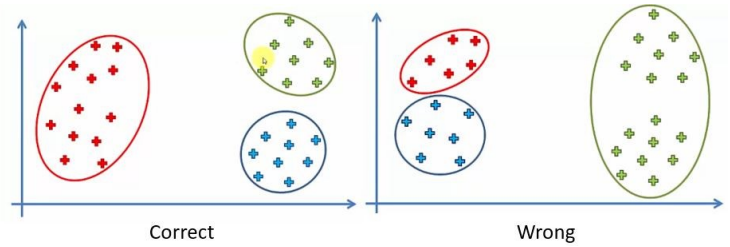
Note for step 3: can use the **perpendicular distance hack**, where you draw a line between two centroids and draw a perpendiccular line at its center. Anything on its side is closer to it.





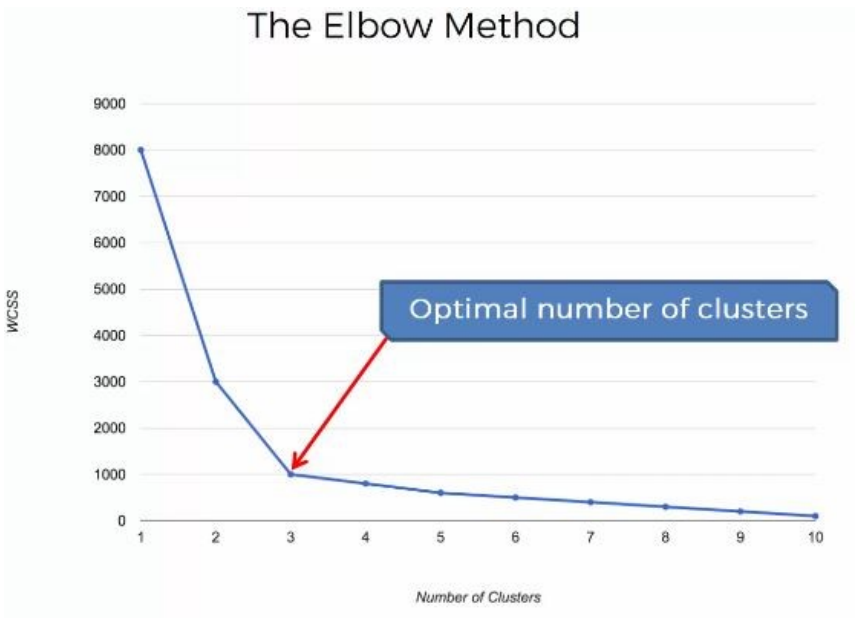
Note that there is **NO** train test split because no Y!

Problems



The **random initialisation trap** where it will lead to wrong clusters formed. The solution is the **K-Means++ algorithm**.

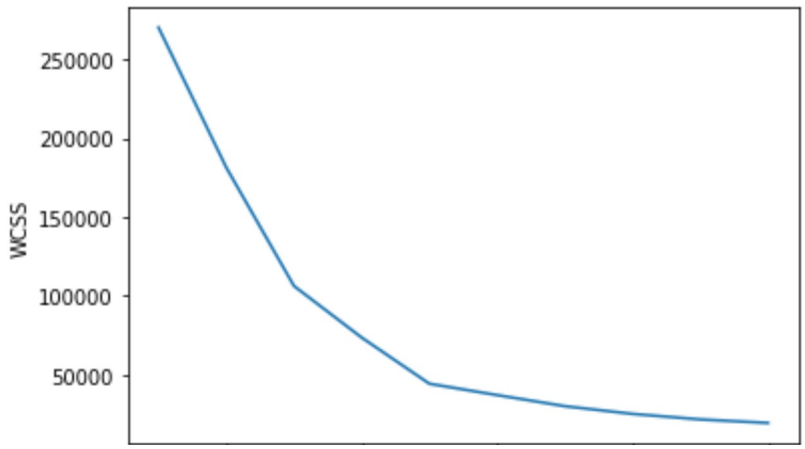
$$\text{WCSS} = \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster 2}} \text{distance}(P_i, C_2)^2 + \sum_{P_i \text{ in Cluster 3}} \text{distance}(P_i, C_3)^2$$



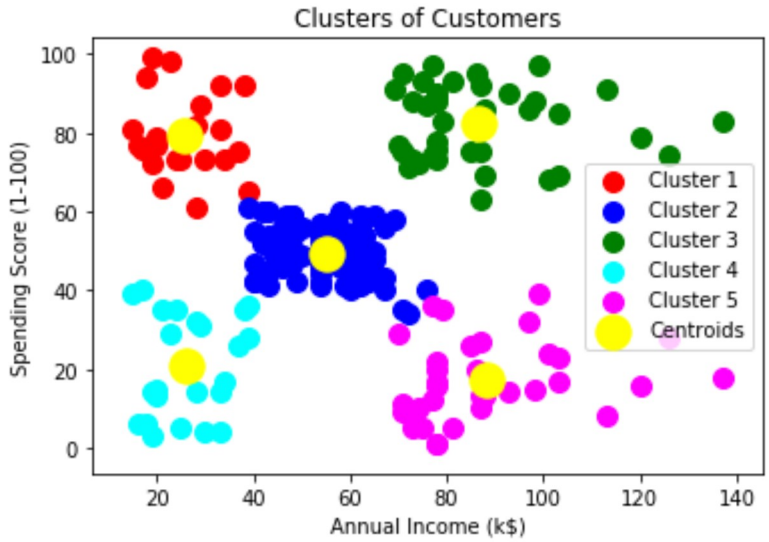
Furthermore, how to choose the correct number of clusters? We use the **within cluster sum of squares (WCSS)**. It may look complicated, but actually each part is just summing the square of distance difference from the centroid for each data point, aka distance btw point and centroid + squared + sum all these. Then compare WCSS when calculated for 1 cluster, for 2, for 3 etc. When every single point is its own centroid, then WCSS = 0. Plot the WCSS, and the graph will be like an elbow, ie **the elbow method**. Choose arbitrarily the one that you think is optimal.

In [11]:
```
### 1. Libraries

```

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,1:].values # correct version actually
X = data.iloc[:,[3,4]].values # for vosualisation so only 2D 2 featur

### 3. Modelling

# Elbow Method
from sklearn.cluster import KMeans
wcss = [] # stores all the wcss results
for i in range(1,11): # do 10 times
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1,11), wcss) # x-axis is times, y-axis is wcss
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.show()

# Actual
kmeans = KMeans(n_clusters=5, init='k-means++', random_state=42) # 42
y_kmeans = kmeans.fit_predict(X) # train AND predict
print(y_kmeans)

### 4. Visualisation

# plt.scatter(X[:,0]) # x-axis is annual income, choose all rows, onl
plt.scatter(X[y_kmeans==0,0], X[y_kmeans==0,1], s=100, c='red', label
plt.scatter(X[y_kmeans==1,0], X[y_kmeans==1,1], s=100, c='blue', labe
plt.scatter(X[y_kmeans==2,0], X[y_kmeans==2,1], s=100, c='green', lab
plt.scatter(X[y_kmeans==3,0], X[y_kmeans==3,1], s=100, c='cyan', labe
plt.scatter(X[y_kmeans==4,0], X[y_kmeans==4,1], s=100, c='magenta', l
plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1
plt.title('Clusters of Customers')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.show()
```



```
[3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3
 0 3 0 3 0 3 1 3 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 2 4 2 1 2 4 2 4 2 1 2 4 2 4 2 4 2 4 2 1 2 4 2 4
 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4
 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2]
```



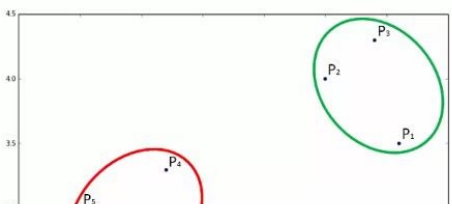
Chapter 15 - Hierarchical Clustering

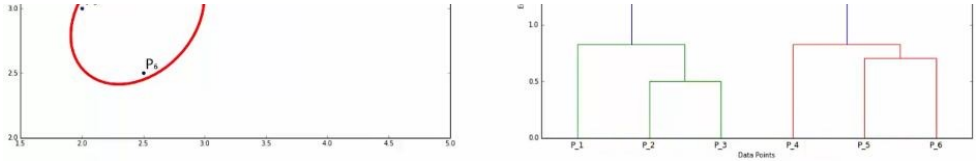
HC is like K-Means, just that they use different methods to get the final results. There are 2 types: **agglomerative (bottom-up approach) and divisive (top-down approach)**. We will focus more on the agglomerative approach, while divisive is the same but reversed.

- Step 1: Make each data point a single-point cluster.
- Step 2: Take the two closest clusters and make them one cluster.
- Step 3: Repeat until there is 1 huge cluster left.

Take note that "distance between two clusters" can be any of the below. Each can impact your output differently:

- Closest points
- Furthest points
- Average distance (average of every distance between each point and the next)
- Distance between centroids





After you do all the steps, the algorithm memorises the process of what we did and stores it in a **dendrogram**. The height you see is the Euclidean distance between the points; the distance between each cluster is measured by **dissimilarity**, the more dissimilar the higher the height. Then set a dissimilarity level, and cut off at there.

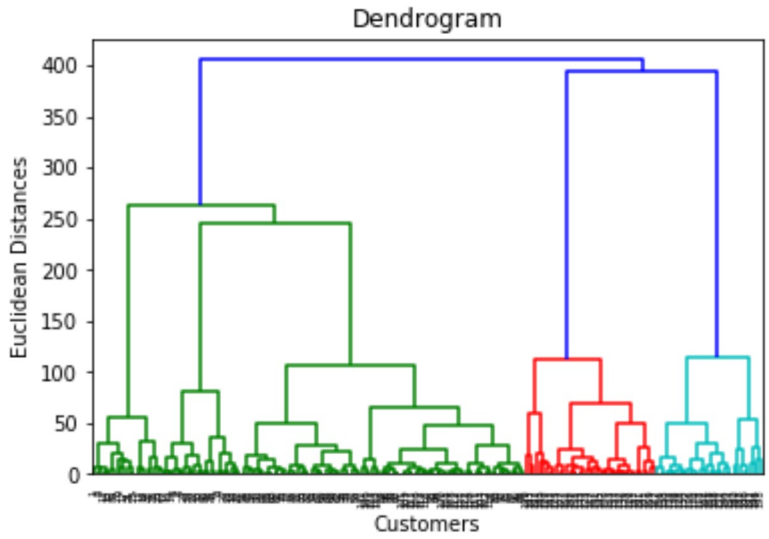
One method people use is to extend all horizontal lines, then choose any vertical line that has no horizontal lines crossing it. Choose the longest one. It means it is the largest distance, and so the resulting cluster would be closer to each other.

HC performs poorer than KM on large datasets. Furthermore, we use **within-cluster variance** to get clusters and create dendrograms.

Clustering Model	Pros	Cons
K-Means	Simple to understand, easily adaptable, works well on small or large datasets, fast, efficient and performant	Need to choose the number of clusters
Hierarchical Clustering	The optimal number of clusters can be obtained by the model itself, practical visualisation with the dendrogram	Not appropriate for large datasets

In [16]:
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,1:].values # correct version actually
X = data.iloc[:,[3,4]].values # for vosualisation so only 2D 2 featur

### 3. Modelling

# Dendrogram
import scipy.cluster.hierarchy as sch
dendrogram = sch.dendrogram(sch.linkage(X, method='ward')) # linkage(
plt.title('Dendrogram')
plt.xlabel('Customers') # becos each point on the xaxis is each data
plt.ylabel('Euclidean Distances')
```

```
plt.show()

# Actual
from sklearn.cluster import AgglomerativeClustering
hc = AgglomerativeClustering(n_clusters=5, affinity='euclidean', link
y_hc = hc.fit_predict(X)

### 4. Visualisation

plt.scatter(X[y_hc==0,0], X[y_hc==0,1], s=100, c='red', label='Cluste
plt.scatter(X[y_hc==1,0], X[y_hc==1,1], s=100, c='blue', label='Clust
plt.scatter(X[y_hc==2,0], X[y_hc==2,1], s=100, c='green', label='Clus
plt.scatter(X[y_hc==3,0], X[y_hc==3,1], s=100, c='cyan', label='Clust
plt.scatter(X[y_hc==4,0], X[y_hc==4,1], s=100, c='magenta', label='Cl
plt.title('Clusters of Customers')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.show()

### 5. Evaluation

# Cluster 1: high income yet low spending. target them and incentivis
# Cluster 2: up to you
# Cluster 3: target the most since high income and high purchase
# Cluster 4: if you wanna be socially responsible since poor but spen
# Cluster 5: low salary and low spending. don't target.
```



Association Rule Learning

Chapter 16 - Apriori

An analytics team found that between 6-9pm, people who buy diapers also buy beer.

There are 3 parts of **apriori**: **support, confidence, and lift**. Imagine if total is 100 people.

Movie Recommendation: $$\text{support}(M) = \frac{\#\text{ user watchlists containing } M}{\#\text{ user watchlists}}$$

Market Basket Optimisation: $$\text{support}(I) = \frac{\#\text{ transactions containing } I}{\#\text{ transactions}}$$

Movie Recommendation: $$\text{confidence}(M_1 \rightarrow M_2) = \frac{\#\text{ user watchlists containing } M_1 \text{ and } M_2}{\#\text{ user watchlists containing } M_1}$$

Market Basket Optimisation: $$\text{confidence}(I_1 \rightarrow I_2) = \frac{\#\text{ transactions containing } I_1 \text{ and } I_2}{\#\text{ transactions containing } I_1}$$

Movie Recommendation: $$\text{lift}(M_1 \rightarrow M_2) = \frac{\text{confidence}(M_1 \rightarrow M_2)}{\text{support}(M_2)}$$

Market Basket Optimisation: $$\text{lift}(I_1 \rightarrow I_2) = \frac{\text{confidence}(I_1 \rightarrow I_2)}{\text{support}(I_2)}$$

Support means if 10 people out of 100 watched Movie B, then it is 10% (support can apply to multiple products also and the number tells you how many transactions contain all these pdts). **Confidence** is given that 40 people have watched Movie A and 7 out of those 40 have watched BOTH Movie A and Movie B, then it is 7/40 = 17.5%.

Lift is confidence divided by support = 17.5% / 10% = 1.75. Lift means if we randomly ask a different population, what is the likelihood that these new people would like Product B given that they liked Product A. So in a sense, lift is like an improvement - earlier it was just 10%, but now since you know they liked Product A, you think they will like Product B too, and yes your chances

of success are now at 17.5%. In essence, **lift measures the strength of your rule**.

- Step 1: Set a minimum support and confidence (eg there are just too many recommendations, so set a yardstick, and do not look at those with support less than 20% etc.)
- Step 2: Take all the subsets in transactions having higher support than minimum support
- Step 3: Take all the rules of these subsets having higher confidence than minimum confidence
- Step 4: Sort the rules by decreasing lift (aka strongest rule on top)

Interpretation:

```
[RelationRecord(items=frozenset({'chicken', 'light cream'}), supp
ort=0.004532728969470737, ordered_statistics=[OrderedStatistic(it
ems_base=frozenset({'light cream'}), items_add=frozenset({'chicke
n'}), confidence=0.29059829059829057, lift=4.84395061728395)]

This means that people who buy light cream -> how likely will the
y buy chicken. Support of 0.45% means that the rule containing th
ese two products appear in 0.45% of all transactions. The confide
nce is 29%; a customer who bought light cream will have 29% chanc
e of buying chicken.
```

Problems:

- Products with high support (aka bought a lot) can appear in many rules but not because they are very related to the other products, but because they are abuden will definitely buy one. For eg, everyone who go NTUC will definitely buy Pepsi. So no matter what, all the rules will have Pepsi in them as it seems that people who buy Pepsi will buy other things, when in fact they are not related. To fix this, you can change either support or confidence. But since you don't want to touch your business rules, you will lower confidence instead so you will see more transactions, not only those that appear because Pepsi was in them.

In [53]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
transactions = []
for i in range(0,7501):
    transactions.append([str(data.values[i,j]) for j in range(0,20)])

### 3. Modelling

# for our min_support, we want to see products that bought min 3 time
# R's default min_confidence is 0.8, but that may be too high. 0.2 is
# generally, lift should be min 3, based on experience
```

```
# minlength and maxlength is min/max of # of pdts in both left and ri
from apyori import apriori
rules = apriori(transactions=transactions, min_support=0.003, min_con

### 4. Interpretation

# display the results
results = list(rules)

# using a very customised method only for apriori
def inspect(results):
    lhs         = [tuple(result[2][0][0])[0] for result in results]
    rhs         = [tuple(result[2][0][1])[0] for result in results]
    supports    = [result[1] for result in results]
    confidences = [result[2][0][2] for result in results]
    lifts       = [result[2][0][3] for result in results]
    return list(zip(lhs, rhs, supports, confidences, lifts))
resultsinDataFrame = pd.DataFrame(inspect(results), columns = ['Left

resultsinDataFrame # unsorted method

resultsinDataFrame.nlargest(n=10, columns = 'Lift') # 10 rows, sort b
```

Out[53]:

	Left Hand Side	Right Hand Side	Support	Confidence	Lift
3	fromage blanc	honey	0.003333	0.245098	5.164271
0	light cream	chicken	0.004533	0.290598	4.843951
2	pasta	escalope	0.005866	0.372881	4.700812
8	pasta	shrimp	0.005066	0.322034	4.506672
7	whole wheat pasta	olive oil	0.007999	0.271493	4.122410
5	tomato sauce	ground beef	0.005333	0.377358	3.840659
1	mushroom cream sauce	escalope	0.005733	0.300699	3.790833
4	herb & pepper	ground beef	0.015998	0.323450	3.291994
6	light cream	olive oil	0.003200	0.205128	3.114710

Chapter 17 - Eclat

Also another associative rule learning model. It is like a simpler version of apriori. In apriori, we use the model to output rules and judge them based on their lift.

For eclat, we are talking about **sets of products**. They use the concept of **support** only (calculated the same way). In essence, we use support to calculate (assume 2 movies) how many people watch both Movies A and B out of everyone. Hence if there is high support, then you can tell those who watched Movie A probably watch / will want to watch Movie B as well.

We use eclat more as a supplementary model. Apriori is the main one.

In [81]:

```
### 1. Libraries

import numpy as np
```

```
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
transactions = []
for i in range(0,7501):
    transactions.append([str(data.values[i,j]) for j in range(0,20)])

### 3. Modelling

from apyori import apriori
rules = apriori(transactions=transactions, min_support=0.003, min_con

### 4. Interpretation

results = list(rules)

def inspect(results):
    lhs         = [tuple(result[2][0][0]) for result in results]
    rhs         = [tuple(result[2][0][1]) for result in results]
    supports    = [result[1] for result in results]
    return list(zip(lhs, rhs, supports))
resultsinDataFrame = pd.DataFrame(inspect(results), columns = ['Produ

resultsinDataFrame.nlargest(n=10, columns = 'Support')
```

Out[81]:

	Product 1	Product 2	Support
4	(herb & pepper,)	(ground beef,)	0.015998
43	(herb & pepper,)	(ground beef, nan)	0.015998
30	(spaghetti, frozen vegetables)	(ground beef,)	0.008666
95	(spaghetti, frozen vegetables)	(ground beef, nan)	0.008666
7	(whole wheat pasta,)	(olive oil,)	0.007999
60	(whole wheat pasta,)	(olive oil, nan)	0.007999
34	(mineral water, shrimp)	(frozen vegetables,)	0.007199
55	(milk, spaghetti)	(olive oil,)	0.007199
102	(mineral water, shrimp)	(frozen vegetables, nan)	0.007199
128	(milk, spaghetti)	(olive oil, nan)	0.007199

Reinforcement Learning

Chapter 18 - Upper Confidence Bound

The Multi-Armed Bandit Problem

The name refers to the casino machines that have the lever. Bandit as it takes peoples' money

away.

We use the concept of **regret**, where the longer you spend on non-optimal choices, the more opportunity cost you would have incurred. OTOH if you don't explore enough, your non-optimal choice might appear as an optimal choice.

In practical terms, think of advertisement campaigns where there are multiple ads and the company will see their success rate / distribution only after thousands of people click the ads. The challenge here is to do the A/B testing and using the best one (exploration and exploitation) in the same phase. It can be simplified to this:

- Basically you are counting the total points each advertisement makes. Our goal is to maximise the total reward we get over many rounds.
- We have *d* arms. For eg, arms are ads that we display to users each time they connect to a webpage.
- Each time a user connects to this webpage, that makes a **round**.
- At each round *n*, we choose one ad to display to the user. This ad *i* gives a reward $r_i(n) \in \{0,1\}$: $r_i(n) = 1$ if the user clicked on the ad *i*, 0 if never.
  - $N_i(n)$ = cumulative no. of times ad *i* was selected up to round *n*.
  - $R_i(n)$ = cumulative sum of rewards of ad *i* up to round *n*.
- From these we compute 2 things:
  - the average reward of ad i up to round n

$$\bar{r}_i(n) = \frac{R_i(n)}{N_i(n)}$$

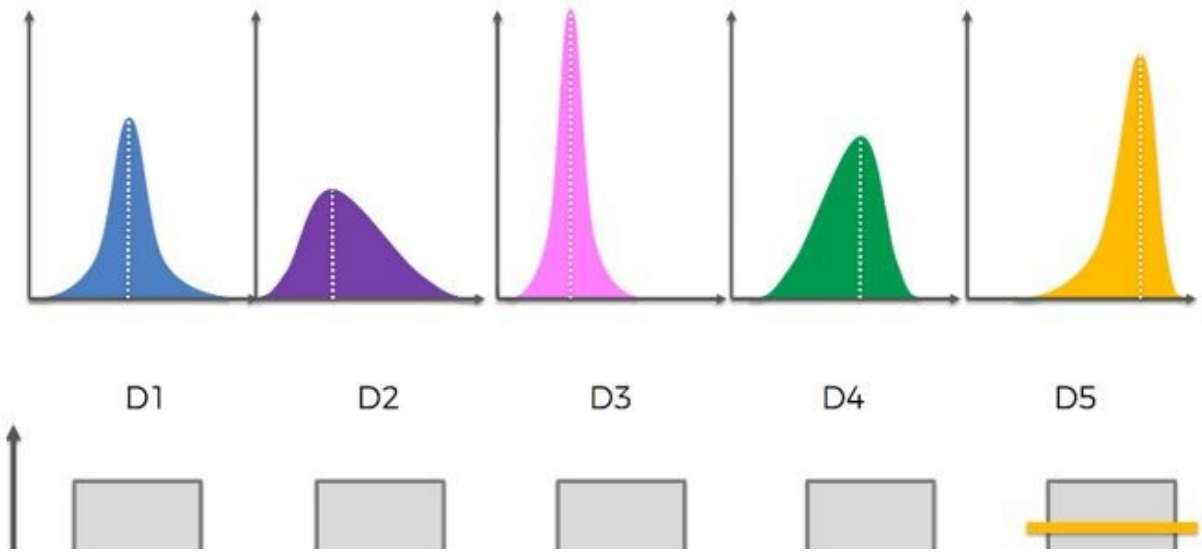
  - the confidence interval $[\bar{r}_i(n) - \Delta_i(n), \bar{r}_i(n) + \Delta_i(n)]$ at round n with

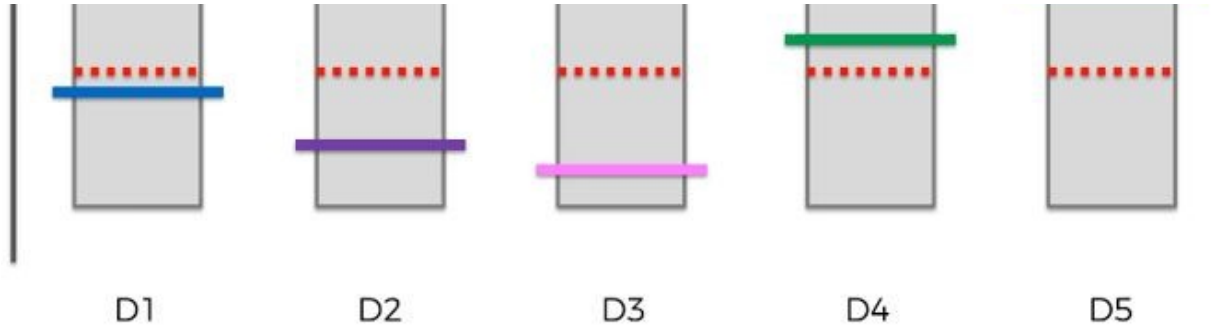
$$\Delta_i(n) = \sqrt{\frac{3}{2}\frac{\log(n)}{N_i(n)}}$$

- Choose the one with the highest UCB (mean+change).

Pictorial Explanation

Do one round per ad to get the average. This is the red dotted line. Everyone will start from here.

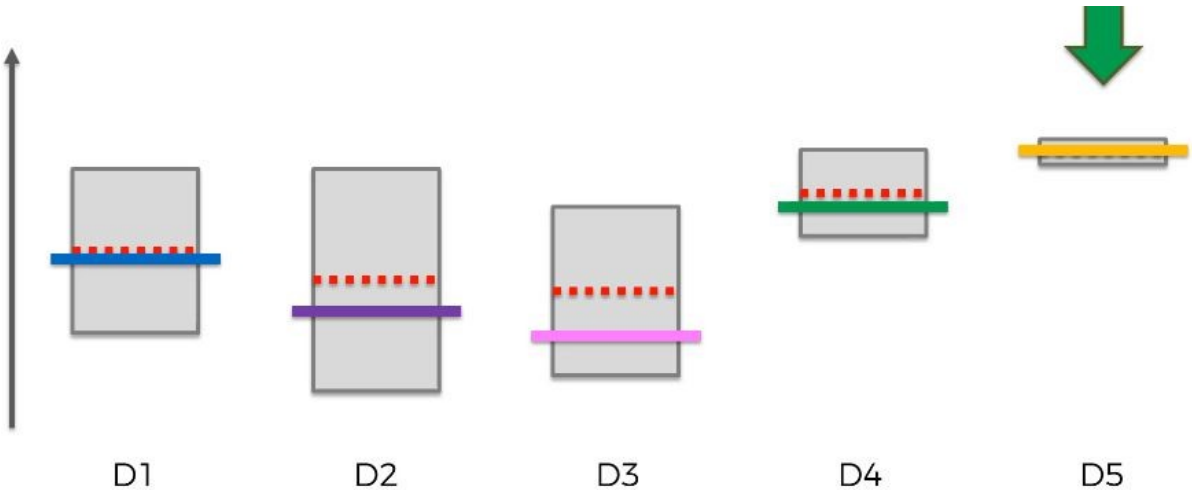




Test one of the ads. User didn't click. Hence shift the confidence interval down. We are more confident that it is not that good, so shorten it as well. Its UCB is now lower than the rest (gives chance to other options as we pick the higher one).

Test another ad. User clicks. Shift up the CI. We are more confident that it is good, so shorten the CI. Its UCB is now lower than the rest.

Keep repeating with the higher UCB option. May switch to a lower one just that you are not stuck with a seemingly good but actually suboptimal solution.



Big Notes

- In reality, this is a dynamic process and not using static datasets. Everything happens in real time. Hence the dataset is not reliable and only serves as an introduction.
- Each advertisement in the dataset is assumed to have a **fixed conversion rate**. This is an important assumption and is close to reality.

In [4]:
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup

### 3. Modelling
```

```
12
13 N = 10000 # no of times you wanna run. can be higher or lower
14 d = 10 # no of ads
15 ads_selected = [] # to store per round the ads selected
16 numbers_of_selections = [0]*d # this is Ni(n). if ad 3 was chosen the
17 sums_of_rewards = [0]*d # this is Ri(n)
18 total_reward = 0
19
20 import math
21 for n in range(0,N):
22     ad = 0
23     max_upperbound = 0
24
25     for i in range(0,d):
26         if numbers_of_selections[i] > 0: # if ad was chosen before
27             average_reward = sums_of_rewards[i] / numbers_of_selectio
28             delta_i = math.sqrt(3/2 * math.log(n+1) / numbers_of_sele
29             upper_bound = average_reward + delta_i
30             print("was ran:",average_reward, delta_i)
31
32         else: # this is actually what happens for the 10 first rounds
33             upper_bound = 1e400
34         if upper_bound > max_upperbound:
35             max_upperbound = upper_bound
36             ad = i # choose the one with the highest upperbound
37         print(i,numbers_of_selections[i],ad, max_upperbound, upper_bo
38
39     ads_selected.append(ad)
40     numbers_of_selections[ad] += 1
41     reward = data.values[n,ad]
42     sums_of_rewards[ad] += reward
43     total_reward += reward
44
45     print(n,ads_selected,numbers_of_selections, sums_of_rewards, tota
46
47 ### 4. Visualisation
48
49 plt.hist(ads_selected)
50 plt.title('Histogram of ads selections')
51 plt.xlabel('Ads')
52 plt.ylabel('No of times each ad was selected')
53 plt.show()
```

```
0 0 0 inf inf
1 0 0 inf inf
2 0 0 inf inf
3 0 0 inf inf
4 0 0 inf inf
5 0 0 inf inf
6 0 0 inf inf
7 0 0 inf inf
8 0 0 inf inf
9 0 0 inf inf
0 [0] [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] 1
was ran: 1.0 1.019666990168809
0 1 0 2.019666990168809 2.019666990168809
1 0 1 inf inf
```

```
2 0 1 inf inf
3 0 1 inf inf
4 0 1 inf inf
5 0 1 inf inf
6 0 1 inf inf
```

In [2]:

Out[2]:

	Ad 1	Ad 2	Ad 3	Ad 4	Ad 5	Ad 6	Ad 7	Ad 8	Ad 9	Ad 10
0	1	0	0	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...
9995	0	0	1	0	0	0	0	1	0	0
9996	0	0	0	0	0	0	0	0	0	0
9997	0	0	0	0	0	0	0	0	0	0
9998	1	0	0	0	0	0	0	1	0	0
9999	0	1	0	0	0	0	0	0	0	0

10000 rows × 10 columns

In [5]:

Out[5]: `[705, 387, 186, 345, 6323, 150, 292, 1170, 256, 186]`

In [14]:

Out[14]: `[120, 47, 7, 38, 1675, 1, 27, 236, 20, 7]`

In [16]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 1,
2, 3, 4, 5, 6, 7, 8, 8, 9, 0, 8, 1, 2, 3, 4, 5, 6, 6, 7, 7, 7, 9, 7,
0, 6, 8, 1, 2, 3, 4, 4, 4, 4, 4, 5, 9, 7, 0, 0, 6, 8, 4, 0, 7, 1, 1,
```

Chapter 19 - Thompson Sampling

Step 1. At each round n, we consider two numbers for each ad i:

- $N_i^1(n)$ - the number of times the ad i got reward 1 up to round n,
- $N_i^0(n)$ - the number of times the ad i got reward 0 up to round n.

Step 2. For each ad i, we take a random draw from the distribution below:

$$\theta_i(n) = \beta(N_i^1(n) + 1, N_i^0(n) + 1)$$

Step 3. We select the ad that has the highest $\theta_i(n)$.

It uses this:

Bayesian Inference

- Ad i gets rewards $\mathbf{y}$ from Bernoulli distribution $p(\mathbf{y}|\theta_i) \sim \mathcal{B}(\theta_i)$.
- θ_i is unknown but we set its uncertainty by assuming it has a uniform distribution $p(\theta_i) \sim \mathcal{U}([0, 1])$, which is the prior distribution.
- Bayes Rule: we approach θ_i by the posterior distribution

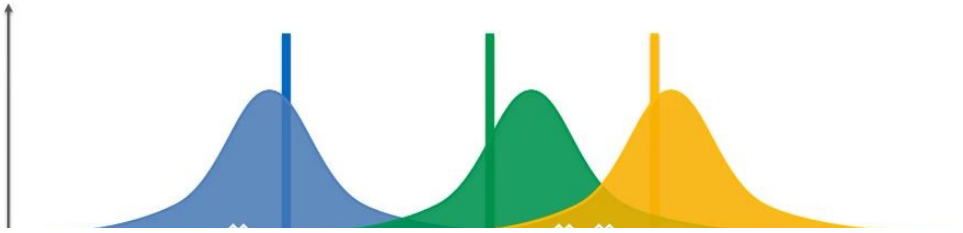
$$\underbrace{p(\theta_i|\mathbf{y})}_{\text{posterior distribution}} = \frac{p(\mathbf{y}|\theta_i)p(\theta_i)}{\int p(\mathbf{y}|\theta_i)p(\theta_i)d\theta_i} \propto \underbrace{p(\mathbf{y}|\theta_i)}_{\text{likelihood function}} \times \underbrace{p(\theta_i)}_{\text{prior distribution}}$$

- We get $p(\theta_i|\mathbf{y}) \sim \beta(\text{number of successes} + 1, \text{number of failures} + 1)$
- At each round n we take a random draw $\theta_i(n)$ from this posterior distribution $p(\theta_i|\mathbf{y})$, for each ad i.
- At each round n we select the ad i that has the highest $\theta_i(n)$.

The pictorial description:

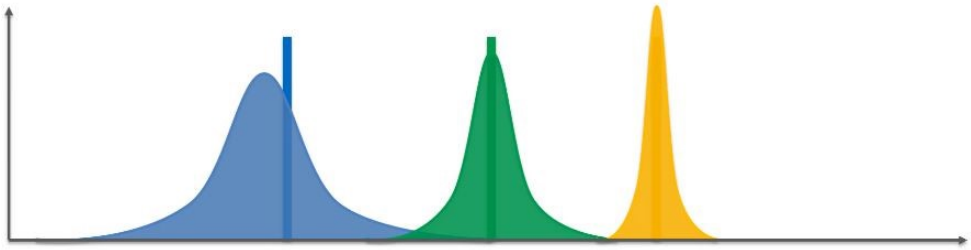
The coloured vertical axis show us the expected returns of 3 machines. You cannot see it in actuality. The yellow one is the best.

As above: You then do several rounds for each machine to construct its respective initial distribution, to guess where we THINK the estimated value might lie.



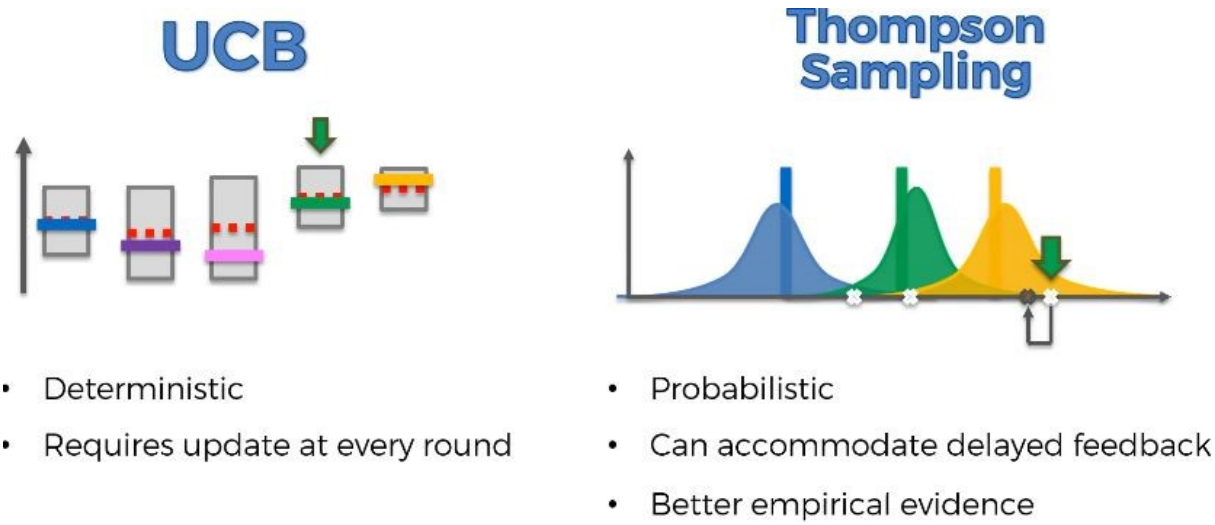
As above: You then choose a random spot in each distribution. These are acted as if they were the expected values. We then choose the one with the highest value (green in picture) and pull the green machine. A new value is generated. Its curve will then shift towards it and become thinner.

Hence the earlier probabilites are the prior probability.



Comparison

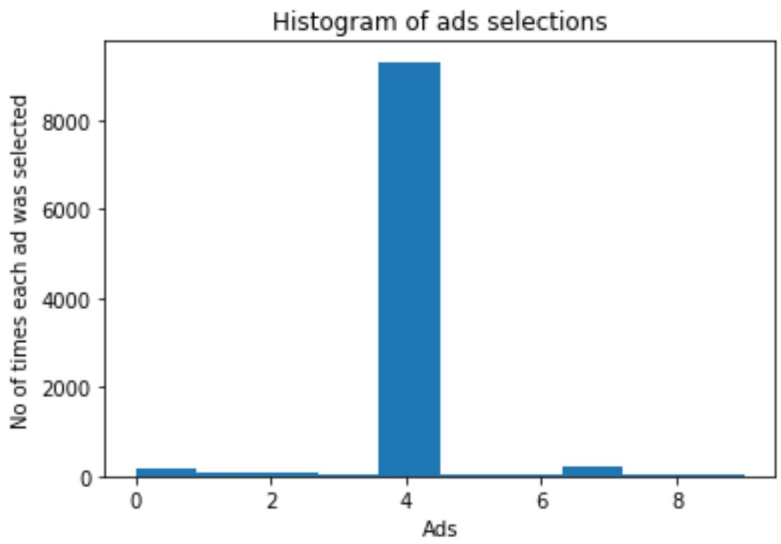
The key difference is that TS can update in batches while UCB must update every round, which may be computationally expensive.



In [20]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup

### 3. Modelling

import random # since we need beta distributions
```

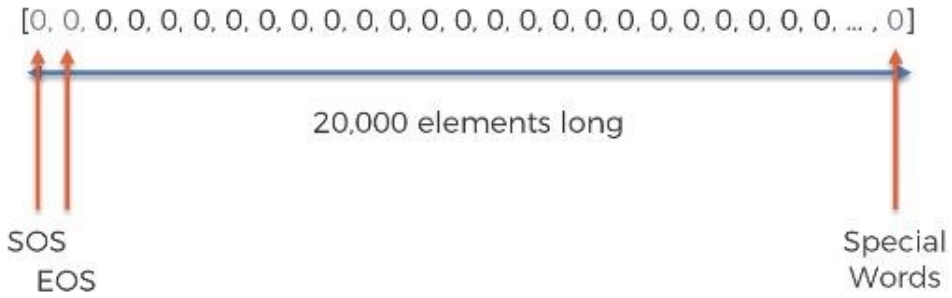
```
N = 10000
d = 10
ads_selected = []
numbers_of_rewards_1 = [0] * d
numbers_of_rewards_0 = [0] * d
total_reward = 0

for n in range(0, N):
    ad = 0 # index for ads
    max_random = 0 # highest beta distribution

    for i in range(0,d):
        random_beta = random.betavariate(numbers_of_rewards_1[i] + 1,
        if random_beta > max_random: # keep track of largest return
            max_random = random_beta
            ad = i

    ads_selected.append(ad)
    reward = data.values[n,ad]
    if reward == 1:
        numbers_of_rewards_1[ad] += 1
    elif reward == 0:
        numbers_of_rewards_0[ad] += 1
    total_reward += reward

### 4. Visualisation

plt.hist(ads_selected)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('No of times each ad was selected')
plt.show()
```



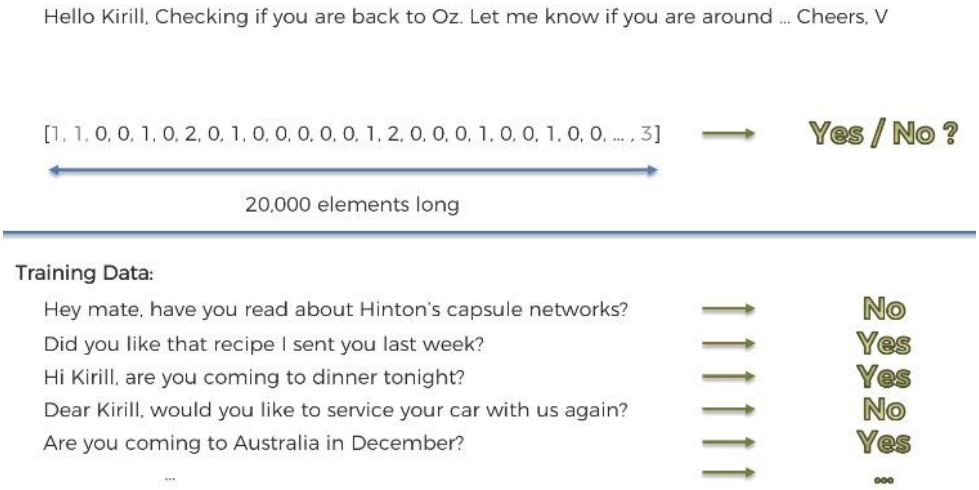
Chapter 20 - Natural Language Processing (NLP)

Bag of Words Model

Assume 20,000 words are commonly used (actually just 3,000!). Map each word to somewhere in the array. The first two are reserved for SOS (Start of Sentence) and EOS (End of Sentence), and the last one is Special Words (any words not in the 20K).



You will then use past data and categorise them into Yes 1 or No 0. Using that, you predict what your current data would be categorised into.



In [15]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

# Get the Dataset
data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
# Preprocessing Preparation (incl. Stopwords + Stemming)
import re
import nltk # for stopwords, ie irrelevant words like 'the' 'and'
# nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer # to apply stemming, ie tr
corpus = [] # to store all the reviews but cleaned up
for i in range(0,1000):
    review = re.sub('[^a-zA-Z]',' ',data['Review'][i]) # re.sub() sub
    review = review.lower()
    review = review.split() # split into individual words
```

```
23
24      ps = PorterStemmer()
25      all_stopwords = stopwords.words('english')
26      all_stopwords.remove('not') # remove the word 'not' from stopword
27      review = [ps.stem(word) for word in review if not word in set(all
28      review = ' '.join(review) # link up and separate word with space
29      corpus.append(review)
30
31  # Tokenisation = Bag of Words model
32
33  from sklearn.feature_extraction.text import CountVectorizer
34  cv = CountVectorizer(max_features=1500) # no. if max size of matrix y
35  X = cv.fit_transform(corpus).toarray() # fittransform puts the words
36  Y = data.iloc[:,-1].values
37
38  # Train Test Split
39
40  from sklearn.model_selection import train_test_split
41  X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0
42
43  ### 3. Modelling
44
45  from sklearn.naive_bayes import GaussianNB
46  classifier = GaussianNB()
47  classifier.fit(X_train,Y_train)
48
49  ### 4. Predictions
50
51  y_pred = classifier.predict(X_test)
52  print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l
53
54  ### 5. Confusion Matrix
55
56  from sklearn.metrics import confusion_matrix
57  cm = confusion_matrix(Y_test, y_pred)
58
59  from sklearn.metrics import accuracy_score
60  ac = accuracy_score(Y_test, y_pred)
61  print(cm,ac)
62
63  ## use these other metrics other than just CM
64  # Accuracy = (TP + TN) / (TP + TN + FP + FN)
65  # Precision = TP / (TP + FP)
66  # Recall = TP / (TP + FN)
67  # F1 Score = 2 * Precision * Recall / (Precision + Recall)
68
69  ### 6. Evaluations
70
71  # Predict if a Review is Positive or Negative
72
73  new_review = 'I love this restaurant so much'
74  new_review = re.sub('[^a-zA-Z]', ' ', new_review)
75  new_review = new_review.lower()
76  new_review = new_review.split()
77  ps = PorterStemmer()
78  all_stopwords = stopwords.words('english')
```

```
all_stopwords.remove('not')
new_review = [ps.stem(word) for word in new_review if not word in set
new_review = ' '.join(new_review)
new_corpus = [new_review]
new_X_test = cv.transform(new_corpus).toarray()
new_y_pred = classifier.predict(new_X_test)
print(new_y_pred) # positive

new_review = 'I hate this restaurant so much'
new_review = re.sub('[^a-zA-Z]', ' ', new_review)
new_review = new_review.lower()
new_review = new_review.split()
ps = PorterStemmer()
all_stopwords = stopwords.words('english')
all_stopwords.remove('not')
new_review = [ps.stem(word) for word in new_review if not word in set
new_review = ' '.join(new_review)
new_corpus = [new_review]
new_X_test = cv.transform(new_corpus).toarray()
new_y_pred = classifier.predict(new_X_test)
print(new_y_pred) # negative
```

```
[[1 0]
 [1 0]
 [1 0]
 [0 0]
 [0 0]
 [1 0]
 [1 1]
 [1 0]]
[[ 67  50]
 [ 20 113]] 0.72
[1]
[0]
```

In [22]:

Out[22]:

	Review	Liked
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday of...	1
4	The selection on the menu was great and so wer...	1
...	...	...
995	I think food should have flavor and texture an...	0
996	Appetite instantly gone.	0
997	Overall I was not impressed and would not go b...	0
998	The whole experience was underwhelming, and I ...	0
999	Then, as if I hadn't wasted enough of my life ...	0

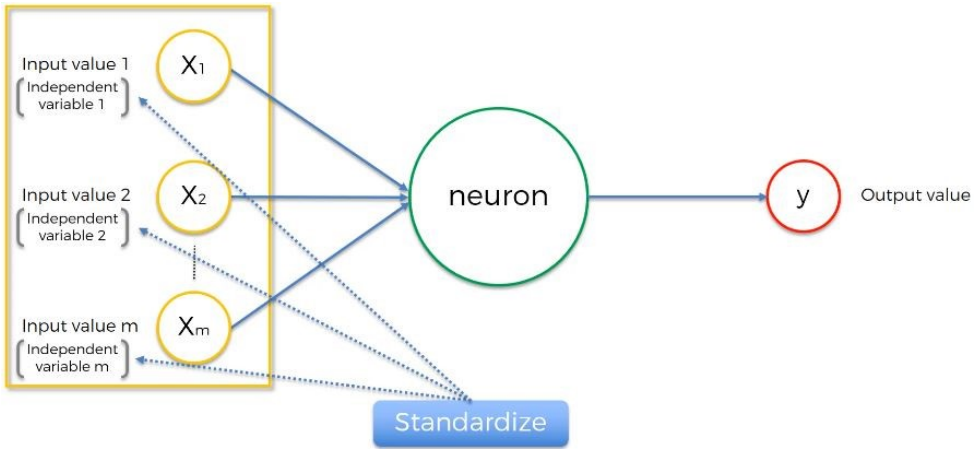
In [7]:

Out[7]:
```
['wow love place',
 'crust not good',
 'not tasti textur nasti',
 'stop late may bank holiday rick steve recommend love',
 'select menu great price']
```

Deep Learning

Chapter 21 - Artificial Neural Networks

Everything here can be found here (https://www.superdatascience.com/blogs/the-ultimate-guide-to-artificial-neural-networks-ann) as well.



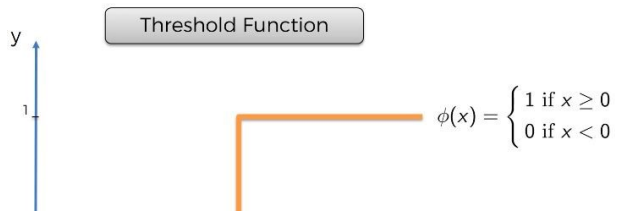
Each input layer is for each feature. We should standardise or normalise each input as well. Furthermore, all of this is for ONE ROW of our data. You also see one output node there which can be continuous, binary, or categorical. If categorical, then there will be many output nodes; the rest are just one node. Each node is called a **perceptron**.

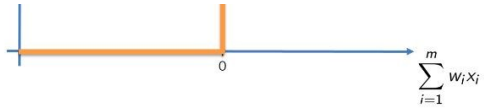
Furthermore each line has a weight, which says which signal is important and which is not for that neuron.

Inside each node, it (a) sums up all the weights, (b) applies activation function on the weighted sum, and (c) pass or not pass the signal.

Activation Function (4 Types)

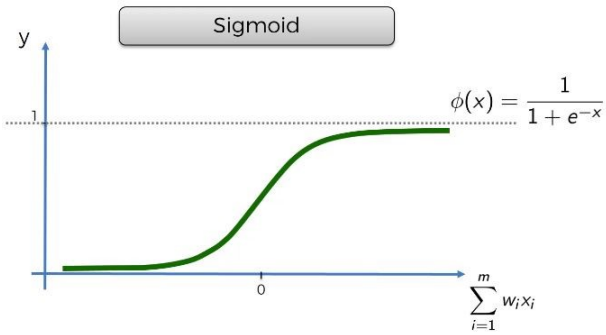
1) Threshold Function





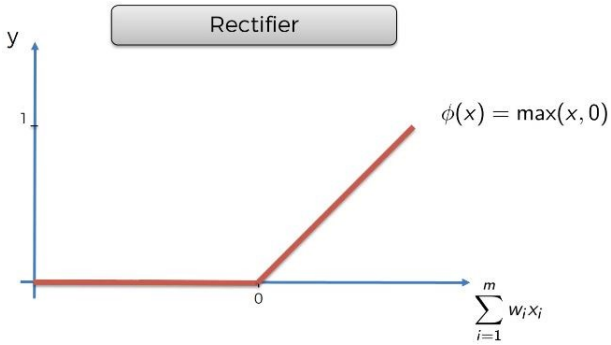
If weighted sum is negative = 0; if positive = 1.

2) Sigmoid



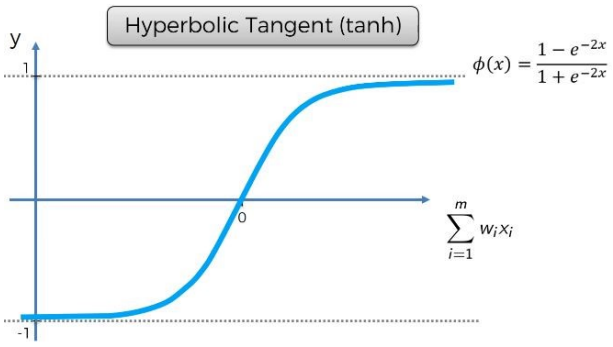
If weighted sum is negative then tends towards 0; if positive tends towards 1.

3) Rectifier



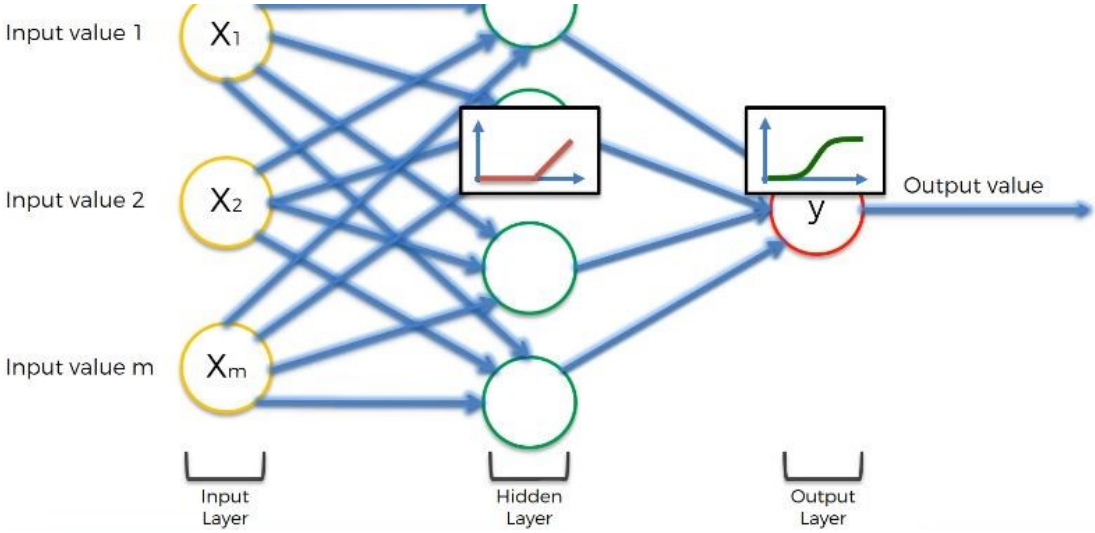
If negative = 0; if positive tends towards 1. Most used.

4) Hyperbolic Tangent / Tanh

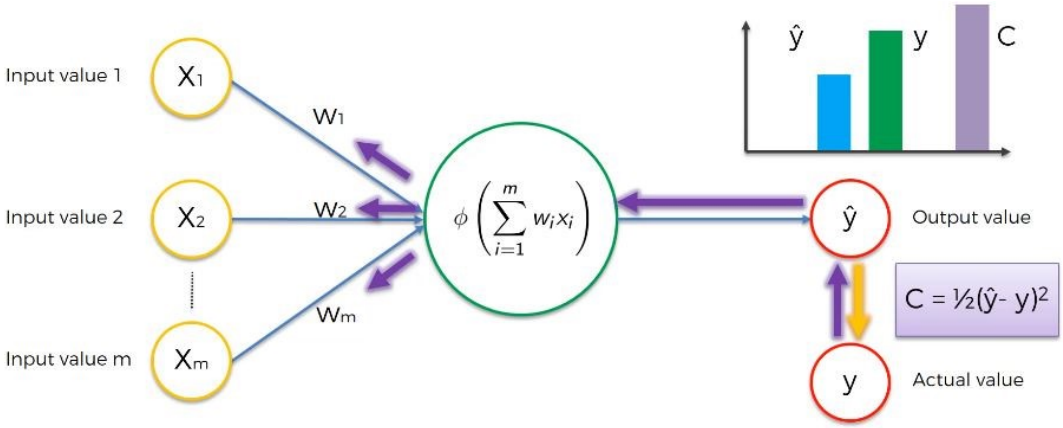


If negative tends towards -1; if positive tends towards 1.

But **practically** for binary outputs, usually use **rectifier function for hidden layers** and use **sigmoid function for the output layer**.



Mechanism

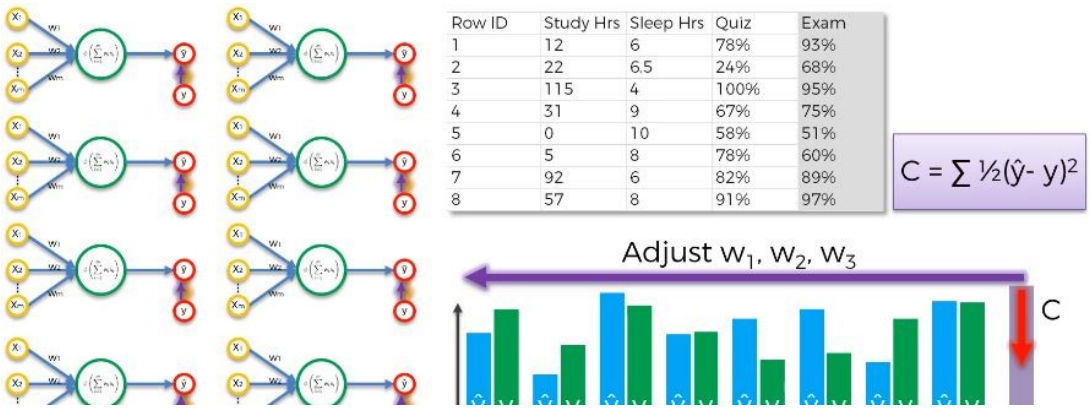


The input values go inside the perceptron, through the activation function outputs a value y-hat. The y-hat is then compared to the actual y. This comparison is called the **cost function** and there are many types - one popular type is 0.5 * (y-hat - y)^2. The cost function tells us what is the error we have in our prediction; the lower it is the less error.

After calculating the cost function, you feed this value back into the NN, and it goes backwards to the weights to update them. The weights are tweaked until the error is minimised.

Then we repeat the entire process again, the error reduces, and y-hat gets closer to y.

Now let us visualise working with many rows.



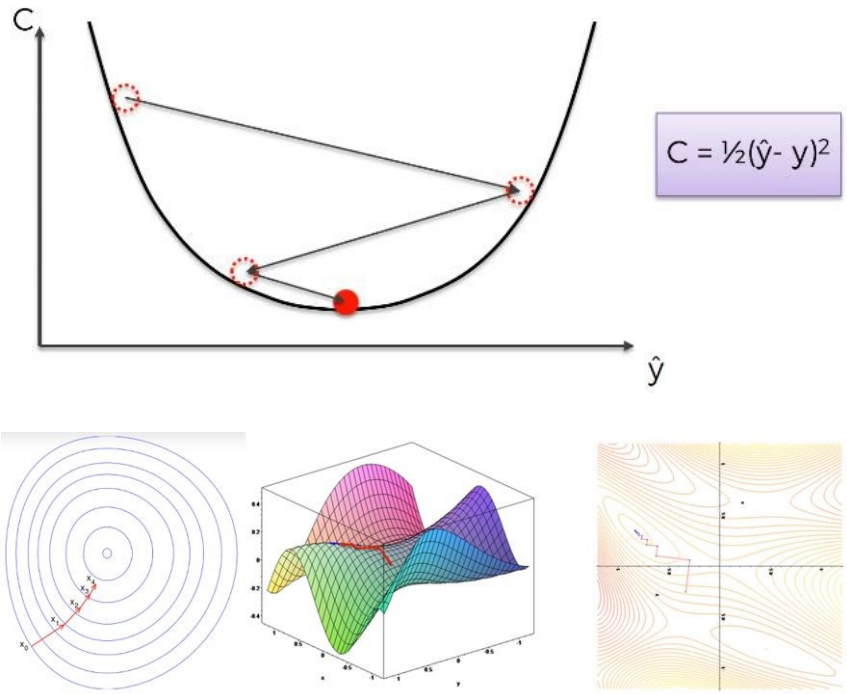
All the rows are fed into the NN to calculate their y-hats. They are then compared to their actual y's. The cost function then calculates the **sum of all cost functions**. We then backpropagate and update the weights. Note that the weights are the same for ALL rows. This is one epoch. Repeat this again many times with a decreasing cost function.

Read more here (https://iamtrask.github.io/2015/07/12/basic-python-network/).

Gradient Descent

This is basically how the weights are adjusted.

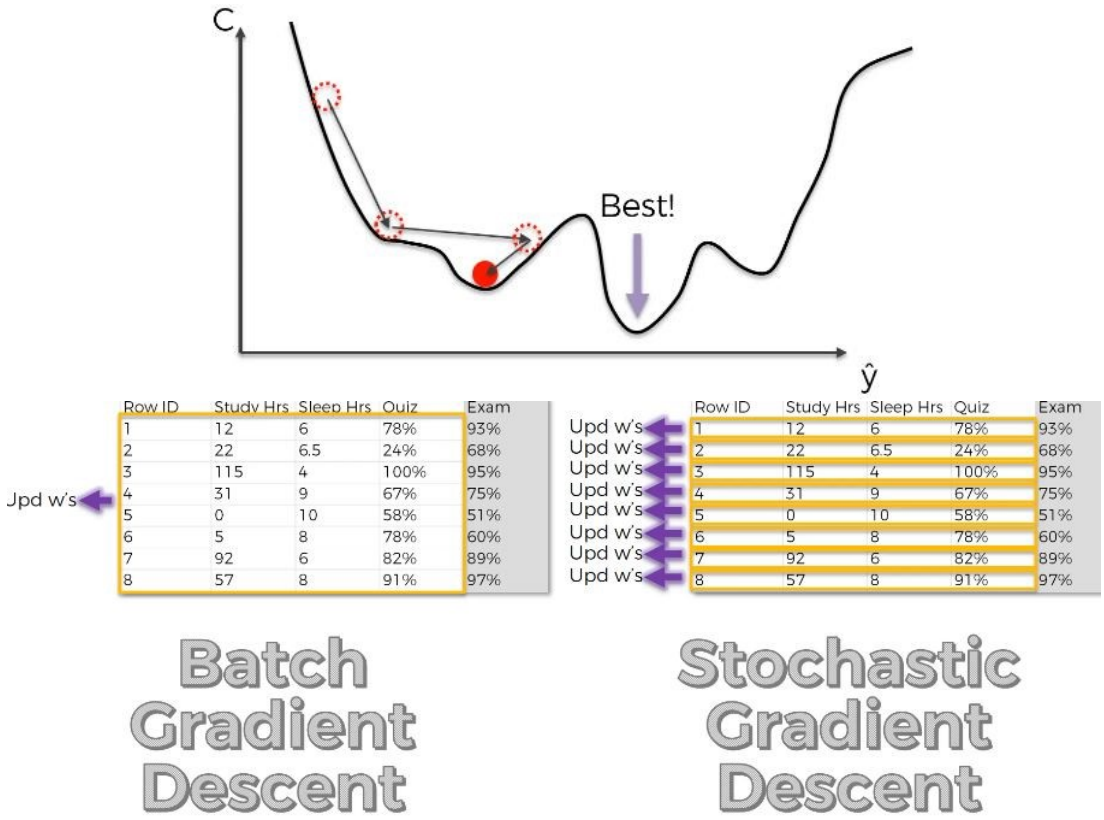
One method would be a brute force method. You try out millions of combinations of weights, measure the cost functions, and choose the minimum. However, as you increase no. of features, you will increase no. of weights, and will lead to the **curse of dimensionality**. Basically, no don't do this. Hence we invited gradient descent.



Basically what happens is we calculate our first cost function, then calculate its tangent by using differentiation. If the slope is negative it means you're downhill, so you need to go right. If the slope is positive you're uphill, you need to go left. So keep repeating until you find the minimum.

Pictured above is the 1D showcase, then the 2D concentric circles, and 3D and mapped to a 2D graph. All show how the gradient descents.

But all that is only possible if the cost function is **convex** and has only **one global minimum**. It is convex because its equation is the squared difference between y-hat and y. But a cost function equation may not be convex, or may be convex but placed in a multidimensional space becomes not convex. In such a way, we would only be finding the **local minimum but not the global one**. This is why we use the **stochastic gradient descent**.



How does SGD work?

While the normal GD takes ALL the rows and feed it into the NN, SGD will take each row one by one, feed into the model to adjust the weights, then take the next row and repeat. In SGD, we are adjusting the weights after every single row rather than doing everything together. The normal GD is called **batch gradient descent** while the per row method is SGD.

SGD does one row at a time so the fluctuations are much higher, and hence it is much likelier to find the global minimum rather than the local minimum. It is also faster than BGD as it doesn't need to load up all the data into memory and run them all together.

BGD is better in one way as it is deterministic - if you start with the same starting weights, it will always give you the same results. But SGD is stochastic and is more random - you are picking your rows at random and updating the NN such that with the different process and iterations you will end with different results.

There is a hybrid version called the **mini batch gradient descent method** which is running 5, 10, 15 etc rows each time only.

Read more about gradient descents here (https://iamtrask.github.io/2015/07/27/python-network-part2/). Or for the heavy math read here (http://neuralnetworksanddeeplearning.com/chap2.html), entire book here (http://neuralnetworksanddeeplearning.com/index.html).

Backpropagation

Backpropagation actually adjusts ALL the weights SIMULTANEOUSLY, not one by one. This means that **it knows which part of the error each of the weights is responsible for**.

Method

- Step 1: Randomly initialise the weights to small numbers close to 0 but not 0.
- Step 2: Input the first observation in the input layer, each feature in one input node.
- Step 3: Forward propagation. Calculate the predicted result y-hat.
- Step 4: Compare with actual result y. Measure the generated error.
- Step 5: Back propagation. The error is backpropagated. Update the weights according to how much they are responsible for the error. The **learning rate** decides by how much we update the weights.
- Step 6: Repeat all and either: (i) update the weights after each observation for *reinforcement learning with SGD*, or (ii) update the weights only after a batch of observations for *batch learning with BGD*.
- Step 7: When the whole training set passes through the ANN, that makes an **epoch**. Redo more epochs.

NOTE: MUST do feature scaling for ALL types of deep learning, even if it's already 0 and 1. Only

In [29]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
print(tf.__version__)

### 2. Dataset

# Actual
data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,3:-1].values # correct version actually
Y = data.iloc[:,-1].values

# Preprocessing
from sklearn.preprocessing import LabelEncoder # 1s & 0s for Gender
le = LabelEncoder()
X[:,2] = le.fit_transform(X[:,2])

from sklearn.compose import ColumnTransformer # one-hot encoding Coun
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder',OneHotEncoder(),[1])]
X = np.array(ct.fit_transform(X)) # fit and transform together. retur

# Train Test Split

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

# Feature Scaling

from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
```

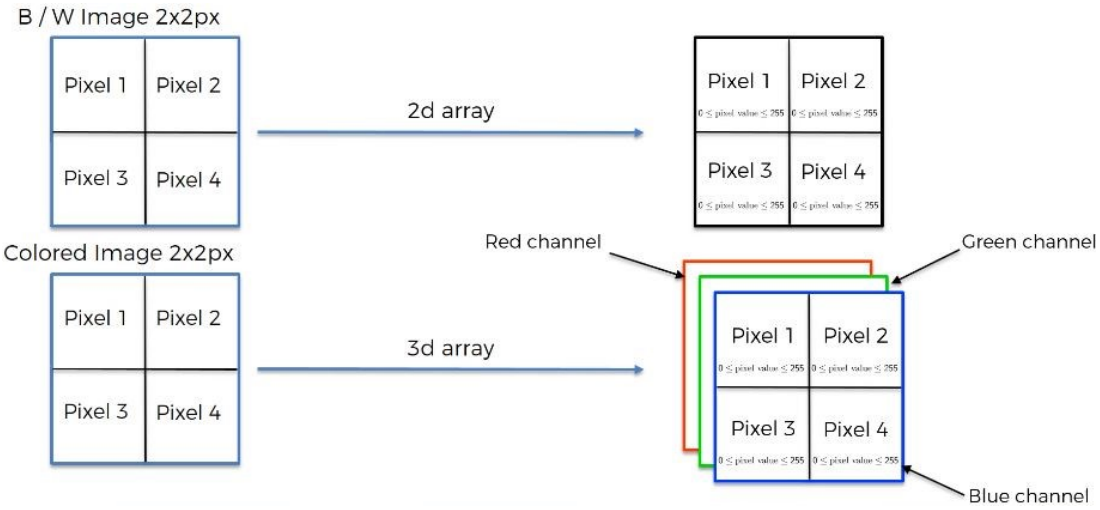
```
X_test = sc_X.transform(X_test)

### 3. Modelling

# Build
ann = tf.keras.models.Sequential() # keras built in TF2
ann.add(tf.keras.layers.Dense(units=6, activation='relu')) # input la
ann.add(tf.keras.layers.Dense(units=6, activation='relu')) # hidden l
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid')) # outpu

# Compile
ann.compile(optimizer='adam', loss='binary_crossentropy', metrics=['a

# Train
ann.fit(X_train, Y_train, batch_size=32, epochs=100) # batchsize give

### 4. Predictions

ann.predict(sc_X.transform([[1,0,0,600,1,40,3,60000,2,1,1,50000]]))

y_pred = ann.predict(X_test)
y_pred = (y_pred > 0.5) # turn it into 0 and 1
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)
```

```
2.2.0
Epoch 1/100
250/250 [==============================] - 0s 495us/step - loss: 0.533
8 - accuracy: 0.7960
Epoch 2/100
250/250 [==============================] - 0s 507us/step - loss: 0.460
5 - accuracy: 0.7960
Epoch 3/100
250/250 [==============================] - 0s 499us/step - loss: 0.437
8 - accuracy: 0.7989
Epoch 4/100
250/250 [==============================] - 0s 507us/step - loss: 0.429
8 - accuracy: 0.8037
Epoch 5/100
250/250 [==============================] - 0s 507us/step - loss: 0.426
2 - accuracy: 0.8085
Epoch 6/100
250/250 [==============================] - 0s 589us/step - loss: 0.423
3 - accuracy: 0.8111
```

Chapter 22 - Convolutional Neural Network

(CNN)

Everything below you can also read here (https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn).



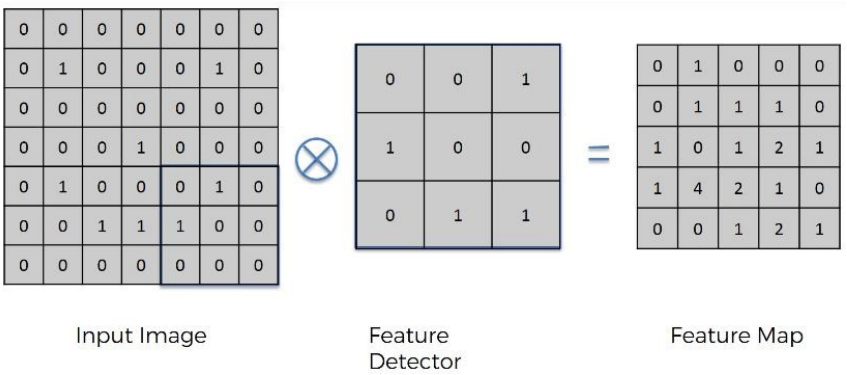
CNN consists of 4 steps: convolution > max pooling > flattening > full connection.

Convolution

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

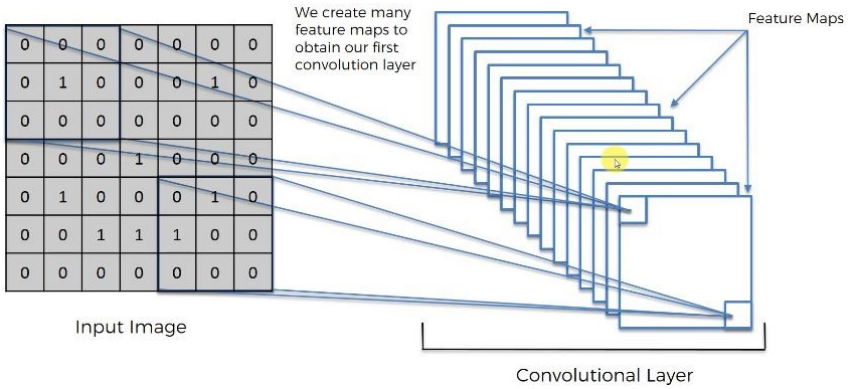
For the math part, can read here (https://cs.nju.edu.cn/wujx/paper/CNN.pdf).

The first thing to note is the use of **feature detectors** which can come in 3x3, 5x5, 7x7 and more. It is usually 3x3. Its other names include **kernel or filter**. Putting this feature detector on the input image, you do some maths like **element-wise multiplication** (input image's square * feature detector's square) and sum all the 9 results here, and the result appears on the **feature map / convolved feature**.

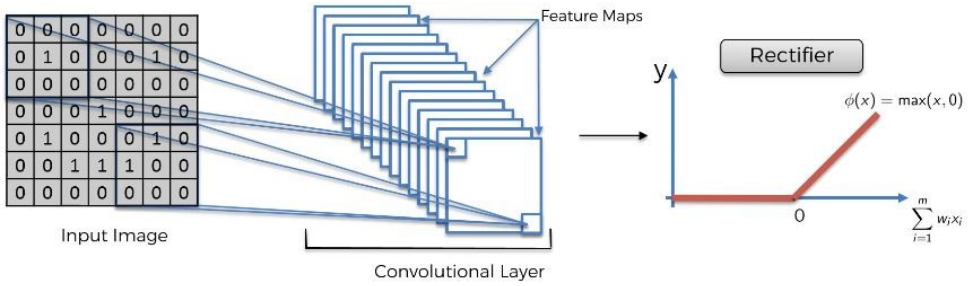


The end result is that the image is reduced. The **stride** makes it even smaller; the bigger the stride, the smaller the image. A smaller image is easier to process and faster. If stride=1 then move filter 1 pixel at a time, if stride=2 move filter 2 pixels at a time.

Doing so also makes you lose information - but when doing CNN you are finding important patterns / features. The highest number in the feature map is where that pattern matches up with the feature detector, ie the "4" is where the input image best matches the feature detector. Hence we do not mind "losing information" because our main aim is to find the features that we are looking for only.



This process is repeated using different feature maps, each finding its own thing. The CNN decides through its training to figure out which features are important and so looks for them.



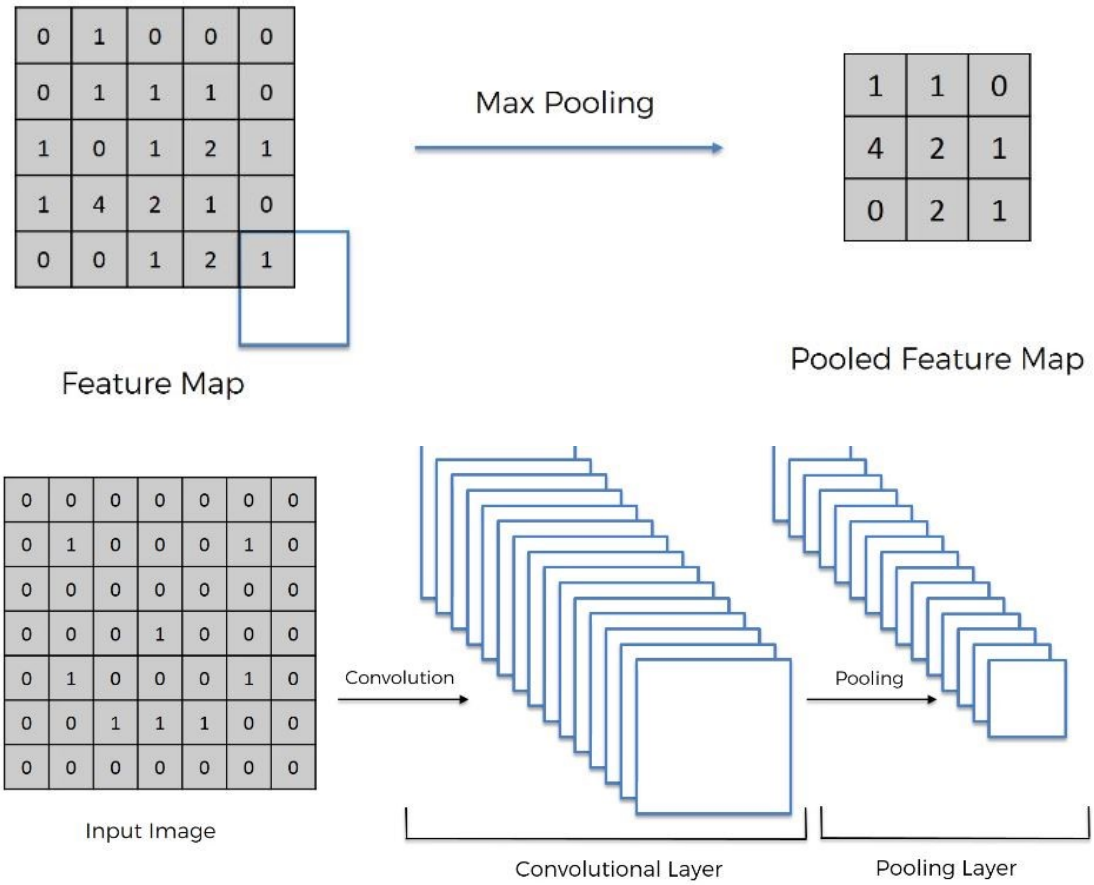
After getting the convolutional layer as above, you apply the rectifier reLU function. This is to increase nonlinearity in our CNN. Pictures normally are nonlinear because each object has its own borders / colours etc and the transition of pixels from each object is nonlinear. Using convolution risks creating something linear so we need to break it up if so.

For example, the middle picture's white-grey-black linearity is broken up by removing the black. There is only white and grey left; there is an abrupt change.

NOTE: You can also use the **parametric ReLU (PReLU)** as well which some people claim is a better ReLU.

Max Pooling

If I flip an image, then by right the pattern would not match and it will be seen as a different image when actually it is the same thing. That is why we must ensure our NN has **spatial invariance** meaning that it doesn't care if the features are slightly tilted, different in texture, closer / further / distorted, our NN must still find that feature. And that is what **max pooling** is about.

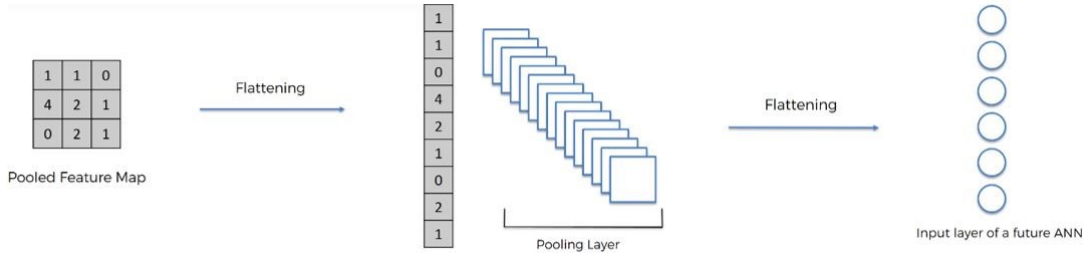


Using the feature map, use a 2x2 (or any size) box, take the max, and place it into the pooled feature map. Then you move the box by a stride (here stride = 2) and do it again. It is okay if the box goes out of borders. All of these will form the **pooling layer**.

By doing this, we further reduce the info and even if there are distortions, the pooled feature map will still be the exact same. In essence, we still preserve the original features and still account for distortions and reduce the size which helps to prevent overfitting(! because the model does not train on everything but just these main big features).

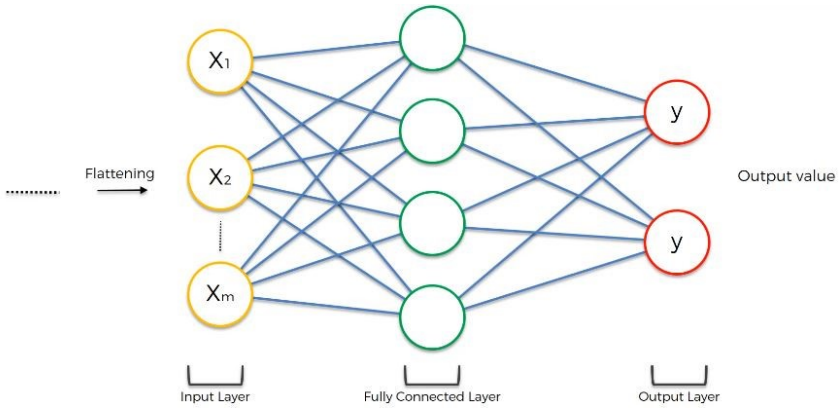
There are also other pooling methods like mean pooling and sum pooling.

Flattening

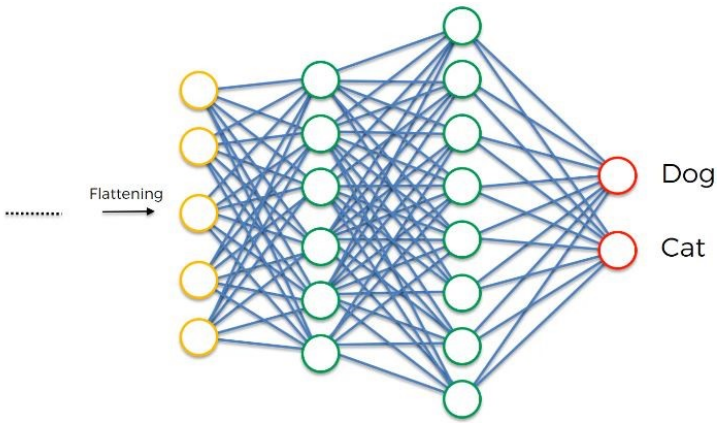


Basically just turning the pooled feature map into a single column.

Full Connection

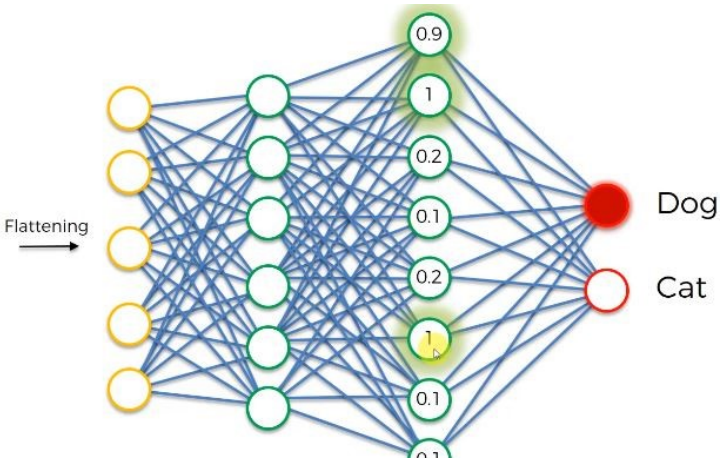


Note: In ANN the hidden layers don't have to be fully connected (no need all the arrows), but for CNN MUST be fully connected.

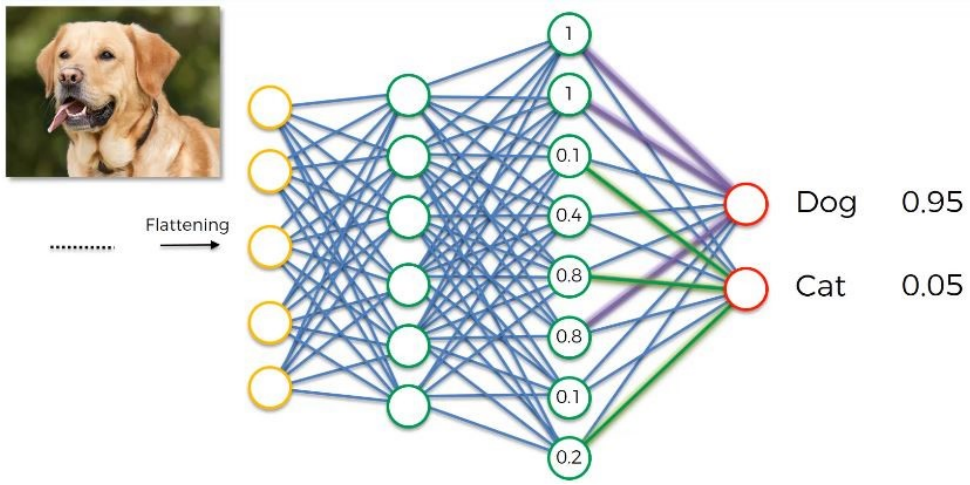


Note 2: Also by right since dog vs cat is 2 categories, the output layer can be just 1. But it can also be 2.

For CNN, the same thing happens except that the cost function is called the **loss function** which is measured by the **cross-entropy function**. We then backpropagate and adjust the weights AND the feature detectors as well to see what happens. Repeat many times.

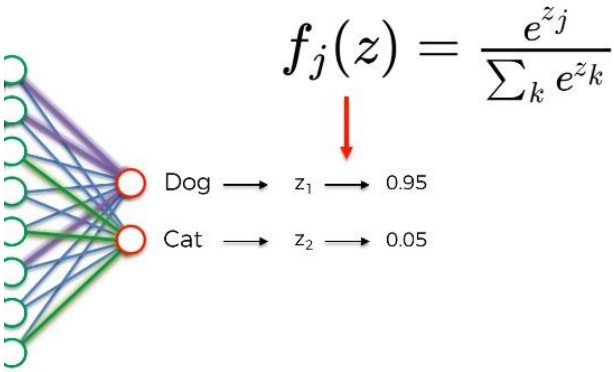


Remember that we are working with FULLY CONNECTED neurons? This means that the number calculated within each neuron means the higher they are, the more confident they have detected a feature. Over many iterations, if the same nodes are firing up often and contributes to saying it is a dog and it indeed matches the actual result, then the model will remember and assign heavier weights for the edges between those nodes and the dog output layer. If not, lower weights.

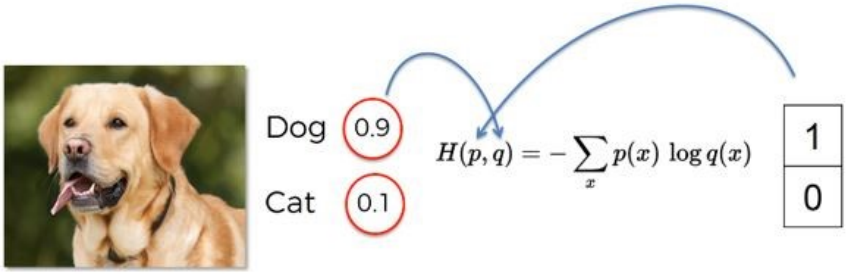


Therefore in the future when a similar picture is fed, the dog or cat nodes will see the few nodes that it listens to and see which lights up. Some will, and some won't. It will then assign a probability based on which nodes light up.

Softmax & Cross-Entropy



In a normal ANN, the output layer(s) all give out a value that may not add up to 1. So it is not useful. Thus we apply **softmax** and it'll transform all these values and make them total to 1 so you know which one has higher votes. It is highly related to the cross-entropy function.



NN1

Row	Dog^	Cat^	Dog	Cat
#1	0.9	0.1	1	0
#2	0.1	0.9	0	1
#3	0.4	0.6	1	0

NN2

Row	Dog^	Cat^	Dog	Cat
#1	0.6	0.4	1	0
#2	0.3	0.7	0	1
#3	0.1	0.9	1	0

Classification Error

NN1	NN2
1/3 = 0.33	1/3 = 0.33

Mean Squared Error

NN1	NN2
0.25	0.71

Cross-Entropy

NN1	NN2
0.38	1.06

Cross-entropy function is highly useful for CNNs. While other ANNs use MSE or cost functions, CNNs use loss functions. Basically we reduce as must cross-entropy to improve the model.

NN1 is always outperforming NN2, even though both wrong for the 3rd entry. If you use classification error, they both seem to be the same. If you use MSE, NN1 is better. If you use cross-entropy which is easier to calculate, NN1 is better also. Cross-entropy is better than MSE

In [79]:

```
### 1. Libraries

import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator #
print(tf.__version__)

### 2. Dataset

# Preprocess the Training Set (to prevent overfitting with test)
train_datagen = ImageDataGenerator( # image augmentation model
    rescale=1./255, # this is feature scaling
    shear_range=0.2,
    zoom_range=0.2, # zooms in
    horizontal_flip=True) # flip
training_set = train_datagen.flow_from_directory( # feed images into
    'zzDataSets/dataset/training_set', # where you keep it
    target_size=(64,64), # final size of images to be fed into CNN
    batch_size=32, # no of images you want in each batch
    class_mode='binary') # binary of categorical

# Preprocess the Test Set (no augmentation other than feature scaling
test_datagen = ImageDataGenerator(rescale=1./255)
test_set = train_datagen.flow_from_directory(
    'zzDataSets/dataset/test_set',
    target_size=(64,64),
    batch_size=32,
    class_mode='binary')

### 3. Modelling

# Initialising
cnn = tf.keras.models.Sequential()

# Convolution
cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation=

# Pooling
```

```
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2)) # poolsize

# Second Convolutional Layer (convolution+pooling together)
cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation=
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))

# Flattening
cnn.add(tf.keras.layers.Flatten())

# Full Connection
cnn.add(tf.keras.layers.Dense(units=128, activation='relu')) # more n

# Output Layer
cnn.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))

# Compile
cnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['a

# Training on Train Set + Evaluating on Test Set
cnn.fit(x=training_set, validation_data=test_set, epochs=25)

### 4. Prediction

import numpy as np
from tensorflow.keras.preprocessing import image
test_image = image.load_img( # load images in PIL format
    'zzDataSets/dataset/single_prediction/cat_or_dog_1.jpg',
    target_size = (64,64)) # need to be same size
test_image = image.img_to_array(test_image) # converts from PIL forma
test_image = np.expand_dims(test_image, axis=0) # since we did batchs
result = cnn.predict(test_image)
training_set.class_indices # to find out what is 0 and 1 (dog=1,cat=0
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'
print(prediction)
```

```
2.2.0
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Epoch 1/25
250/250 [==============================] - 33s 133ms/step - loss: 0.65
39 - accuracy: 0.5990 - val_loss: 0.6116 - val_accuracy: 0.6700
Epoch 2/25
250/250 [==============================] - 33s 132ms/step - loss: 0.58
62 - accuracy: 0.6923 - val_loss: 0.5524 - val_accuracy: 0.7190
Epoch 3/25
250/250 [==============================] - 33s 133ms/step - loss: 0.55
60 - accuracy: 0.7160 - val_loss: 0.5395 - val_accuracy: 0.7225
Epoch 4/25
250/250 [==============================] - 33s 132ms/step - loss: 0.53
20 - accuracy: 0.7340 - val_loss: 0.5371 - val_accuracy: 0.7355
Epoch 5/25
250/250 [==============================] - 33s 132ms/step - loss: 0.51
23 - accuracy: 0.7425 - val_loss: 0.5210 - val_accuracy: 0.7315
```

```
Epoch 6/25
250/250 [==============================] - 33s 134ms/step - loss: 0.49
43 - accuracy: 0.7625 - val_loss: 0.5115 - val_accuracy: 0.7565
Epoch 7/25
250/250 [==============================] - 33s 133ms/step - loss: 0.48
65 - accuracy: 0.7650 - val_loss: 0.5077 - val_accuracy: 0.7510
Epoch 8/25
250/250 [==============================] - 34s 134ms/step - loss: 0.46
91 - accuracy: 0.7734 - val_loss: 0.5260 - val_accuracy: 0.7360
Epoch 9/25
250/250 [==============================] - 34s 136ms/step - loss: 0.45
29 - accuracy: 0.7853 - val_loss: 0.4858 - val_accuracy: 0.7710
Epoch 10/25
250/250 [==============================] - 34s 135ms/step - loss: 0.43
55 - accuracy: 0.7966 - val_loss: 0.4755 - val_accuracy: 0.7765
Epoch 11/25
250/250 [==============================] - 34s 137ms/step - loss: 0.43
13 - accuracy: 0.7996 - val_loss: 0.4783 - val_accuracy: 0.7710
Epoch 12/25
250/250 [==============================] - 34s 135ms/step - loss: 0.42
25 - accuracy: 0.8044 - val_loss: 0.5056 - val_accuracy: 0.7580
Epoch 13/25
250/250 [==============================] - 34s 134ms/step - loss: 0.41
14 - accuracy: 0.8116 - val_loss: 0.4570 - val_accuracy: 0.7815
Epoch 14/25
250/250 [==============================] - 33s 134ms/step - loss: 0.40
11 - accuracy: 0.8183 - val_loss: 0.4962 - val_accuracy: 0.7645
Epoch 15/25
250/250 [==============================] - 33s 133ms/step - loss: 0.40
08 - accuracy: 0.8190 - val_loss: 0.4683 - val_accuracy: 0.7930
Epoch 16/25
250/250 [==============================] - 33s 134ms/step - loss: 0.38
26 - accuracy: 0.8253 - val_loss: 0.4838 - val_accuracy: 0.7800
Epoch 17/25
250/250 [==============================] - 34s 134ms/step - loss: 0.37
45 - accuracy: 0.8271 - val_loss: 0.4628 - val_accuracy: 0.7905
Epoch 18/25
250/250 [==============================] - 34s 135ms/step - loss: 0.36
69 - accuracy: 0.8325 - val_loss: 0.4672 - val_accuracy: 0.7805
Epoch 19/25
250/250 [==============================] - 34s 135ms/step - loss: 0.35
75 - accuracy: 0.8388 - val_loss: 0.4741 - val_accuracy: 0.7885
Epoch 20/25
250/250 [==============================] - 34s 134ms/step - loss: 0.33
99 - accuracy: 0.8484 - val_loss: 0.4710 - val_accuracy: 0.7895
Epoch 21/25
250/250 [==============================] - 33s 133ms/step - loss: 0.33
47 - accuracy: 0.8518 - val_loss: 0.5535 - val_accuracy: 0.7695
Epoch 22/25
250/250 [==============================] - 33s 133ms/step - loss: 0.32
37 - accuracy: 0.8619 - val_loss: 0.4572 - val_accuracy: 0.7930
Epoch 23/25
250/250 [==============================] - 33s 133ms/step - loss: 0.31
59 - accuracy: 0.8590 - val_loss: 0.5086 - val_accuracy: 0.7825
Epoch 24/25
250/250 [==============================] - 34s 135ms/step - loss: 0.29
```

```
76 - accuracy: 0.8701 - val_loss: 0.4753 - val_accuracy: 0.7925
Epoch 25/25
250/250 [==============================] - 33s 133ms/step - loss: 0.30
50 - accuracy: 0.8677 - val_loss: 0.4748 - val_accuracy: 0.7870
```

Dimensionality Reduction

We can choose features either by (i) feature selection using things like backward elimination / forward selection / bidirectional elimination / score comparison taught in the regression chapter, or (ii) **feature extraction** with PCA / LDA / kernel PCA / quadratic discriminant analysis QDA.

Chapter 23 - Principal Component Analysis (PCA)

This is an unsupervised learning algorithm which is popular for: noise filtering, visualisation, stock market predictions, gene data analysis, and feature extraction.

The goal is to identify patterns in data and **detect correlation between variables**. If there is strong correlation, can eliminate it. Hence it reduces the dimensions of a d-dimensional dataset by projecting it onto a k-dimensional subspace where k < d.

Steps:

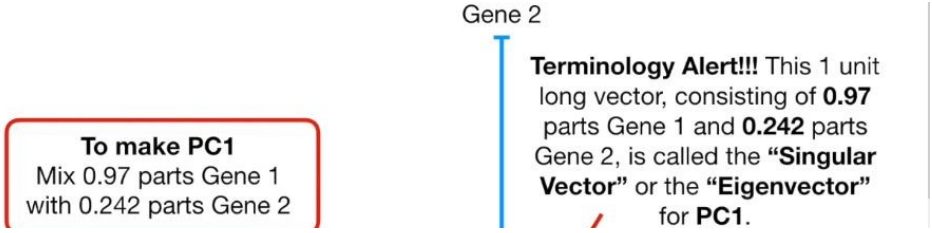
- Standardise the data.
- Obtain the **Eigenvectors and Eigenvalues** from the covariance matrix / correlation matrix, or perform Singular Vector Decomposition.
- Sort eigenvalues in descending order and choose the *k* eigenvectors that correspond to the *k* largest eigenvalues where *k* is the number of dimensions of the new feature subspace (k <= d).
- Construct the projection matrix **W** from the selected *k* eigenvectors.
- Transform the original dataset **X** via **W** to obtain a k-dimensional feature subspace **Y**.

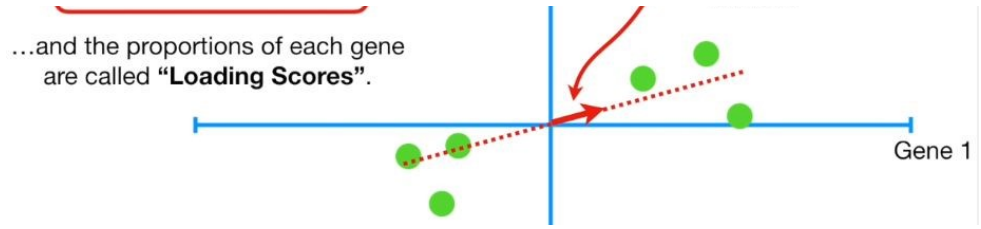
The pros are that it can find relationship between X and Y values, and find the list of principal axes. The cons are that it is very affected by outliers. But considered very popular.

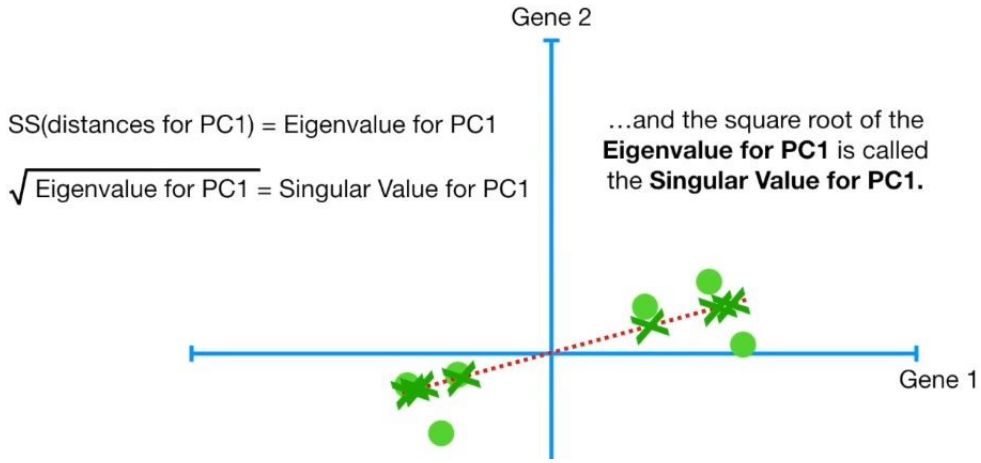
More here (https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c) and especially here (https://www.youtube.com/watch?v=FgakZw6K1QQ).

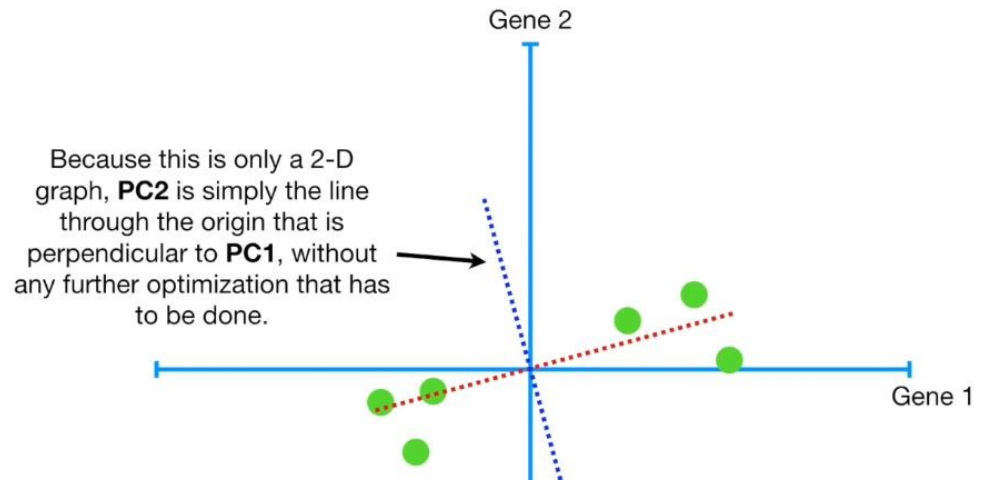
Pictorial Explanation

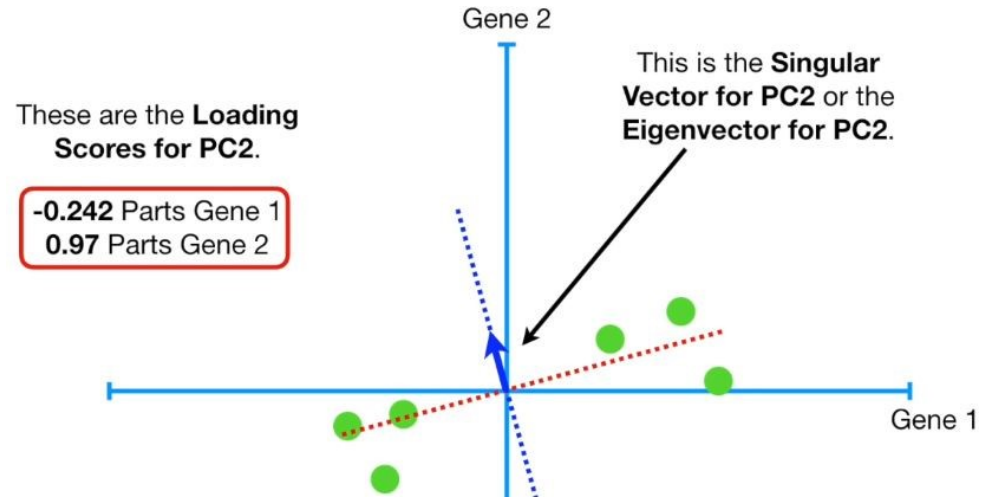
That arrow is **eigenvector**. The value of SS(distance) is **eigenvalue**. The coefficients of features are **loading scores**. The **variation** for each PC can be calculated as %. This can be represented on a **scree plot** as well.




...and the proportions of each gene are called **"Loading Scores"**.
Gene 1


Gene 2
SS(distances for PC1) = Eigenvalue for PC1
$\sqrt{\text{Eigenvalue for PC1}}$ = Singular Value for PC1
...and the square root of the **Eigenvalue for PC1** is called the **Singular Value for PC1.**
Gene 1


Gene 2
Because this is only a 2-D graph, **PC2** is simply the line through the origin that is perpendicular to **PC1**, without any further optimization that has to be done.
Gene 1

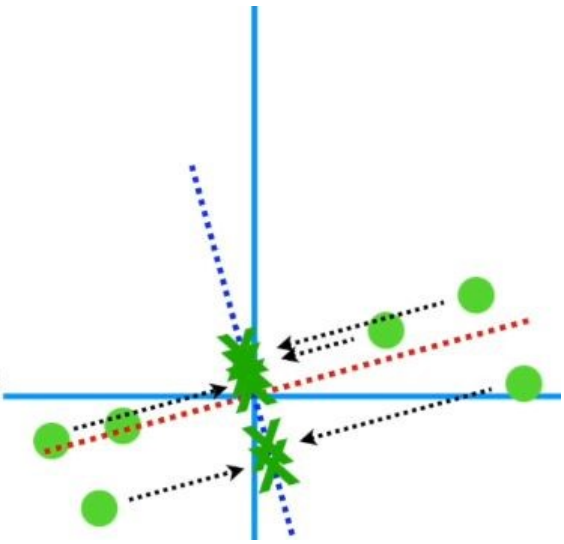

Gene 2
These are the **Loading Scores for PC2**.
-0.242 Parts Gene 1
0.97 Parts Gene 2
This is the **Singular Vector for PC2** or the **Eigenvector for PC2**.
Gene 1


Gene 2

These are the **Loading Scores for PC2**.

-0.242 Parts Gene 1
0.97 Parts Gene 2

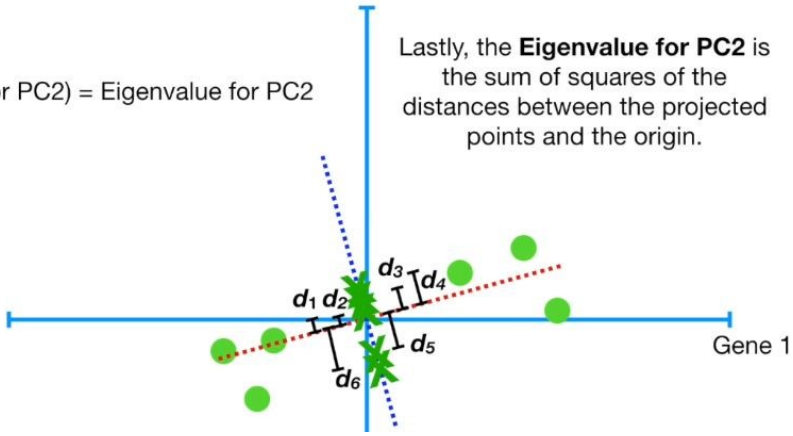
They tell us that, in terms of how the values are projected onto PC2, Gene 2 is 4 times as important as Gene 1.



$d_1^2 + d_2^2 + d_3^2 + d_4^2 + d_5^2 + d_6^2$ = sum of squared distances = SS(distances)

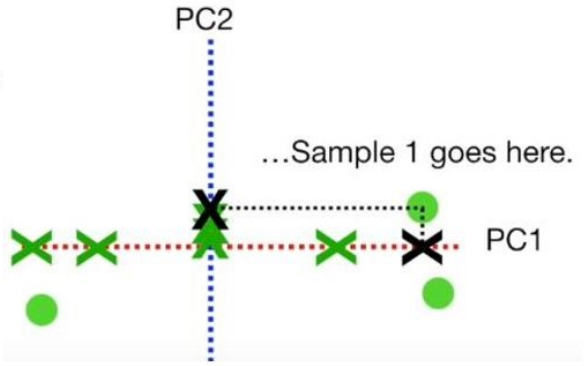
SS(distances for PC2) = Eigenvalue for PC2

Lastly, the **Eigenvalue for PC2** is the sum of squares of the distances between the projected points and the origin.



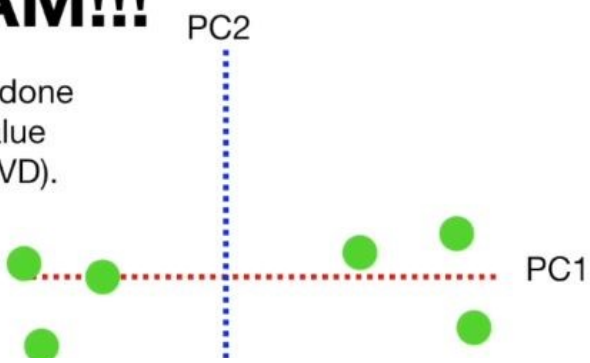
We simply rotate everything so that PC1 is horizontal...

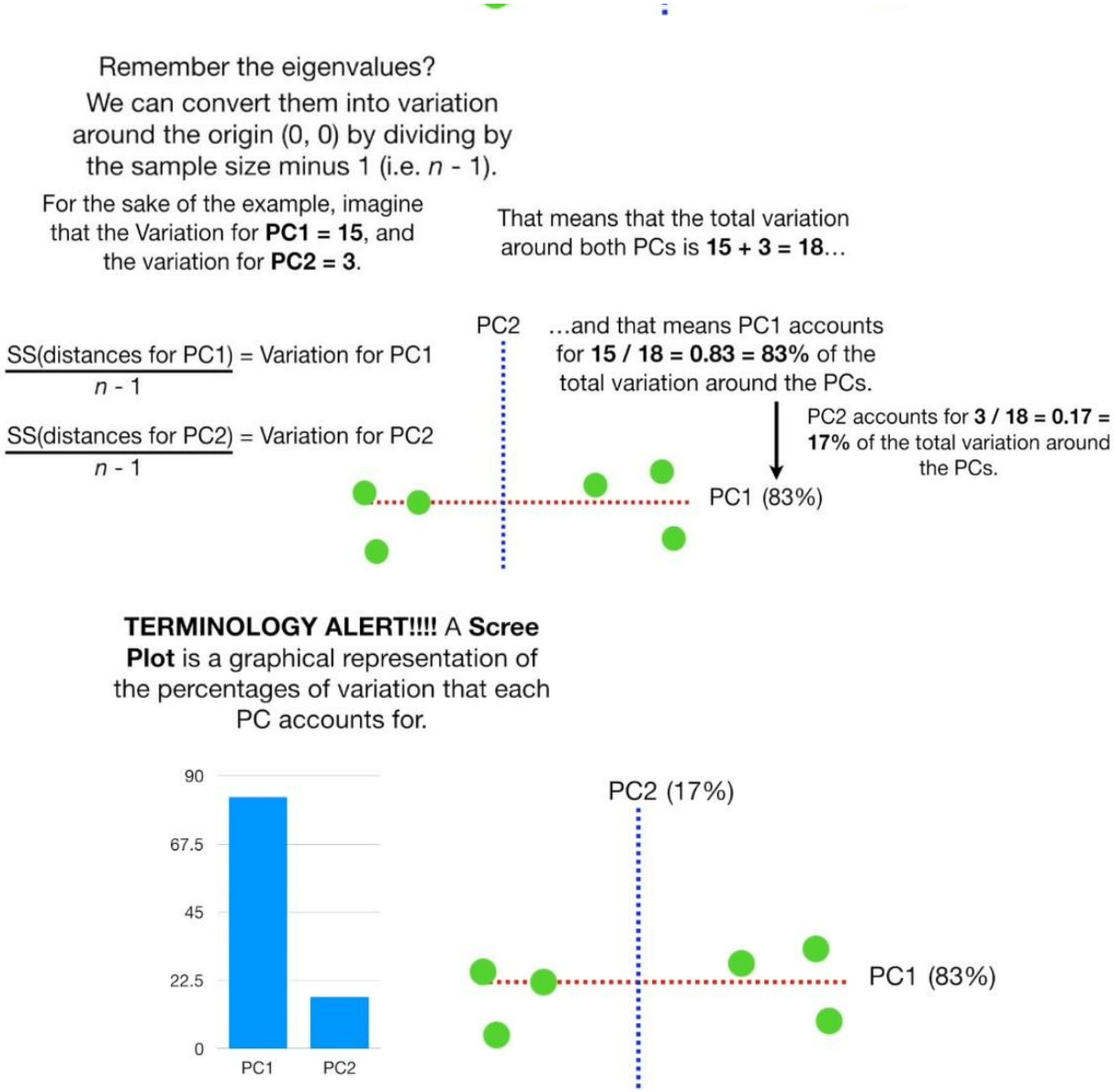
...then we use the projected points to find where the samples go in the PCA plot.



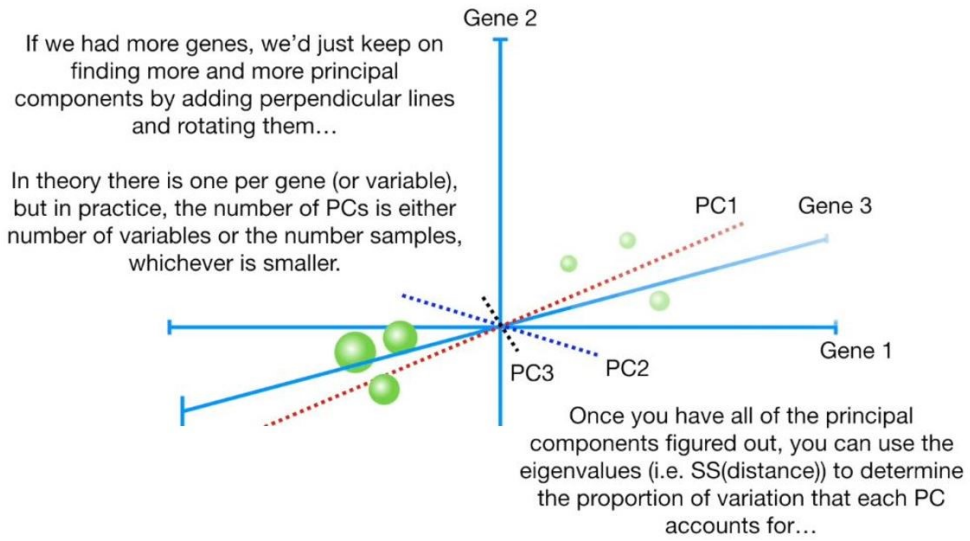
Double BAM!!!

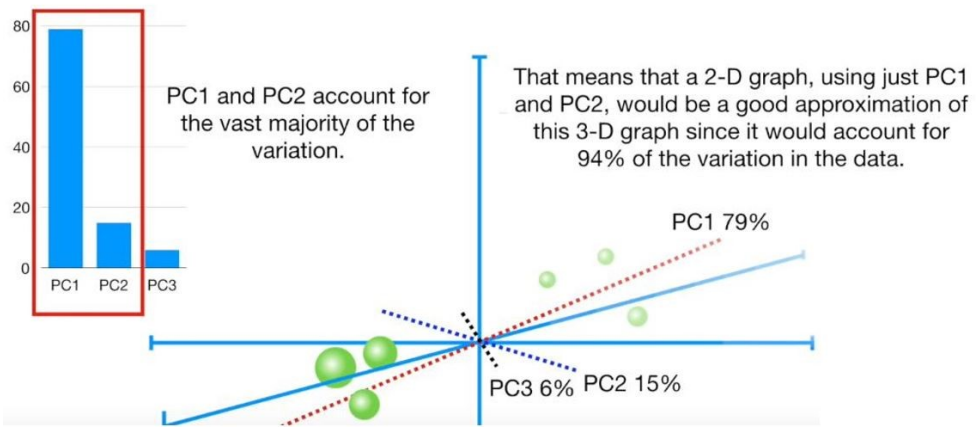
That's how PCA is done using Singular Value Decomposition (SVD).



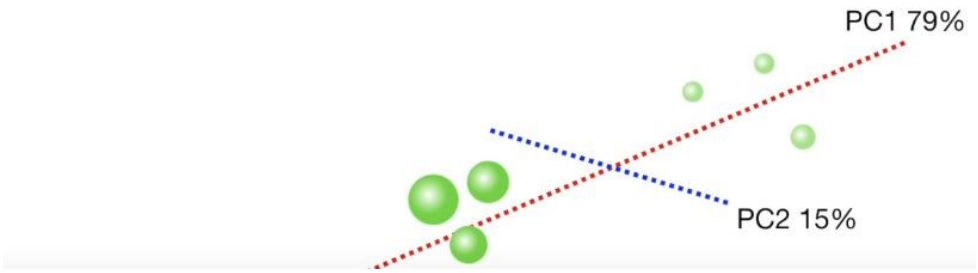


But when you go into 3 or more features, even though you may not be able to visualise, you can just plot the scree graph and choose those PC that account for most of the total variation. These are the important ones.

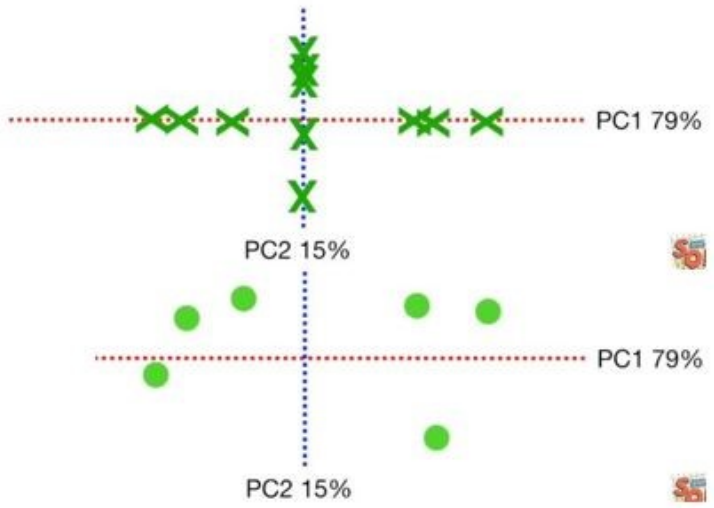




To convert the 3-D graph into a 2-D PCA graph, we just strip away everything but the data and PC1 and PC2...



Then we rotate so that PC1 is horizontal and PC2 is vertical (this just makes it easier to look at).



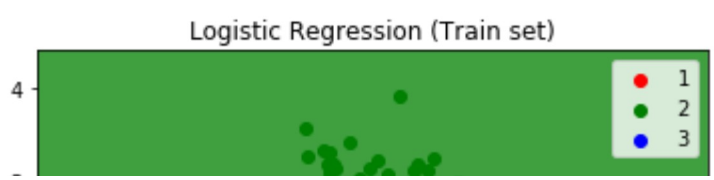
NOTE: We are NOT reducing the number of features to select. Rather we will use all the features to create 2 NEW FEATURES called PC1 and PC2, and then plot from there.

In [7]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("zzDatasets/Wine.csv")
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Apply PCA

from sklearn.decomposition import PCA
pca = PCA(n_components=2) # no. of final extracted features
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)

### 4. Modelling (we use LogReg here but can use any classifier)

from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state=0)
classifier.fit(X_train, Y_train) # smaller the C, stronger the regula

### 5. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 6. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 7. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = X_train, Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = 
                     np.arange(start = X_set[:, 1].min() - 1, stop = 
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.rave
             alpha = 0.75, cmap = ListedColormap(('red', 'green','blu
```

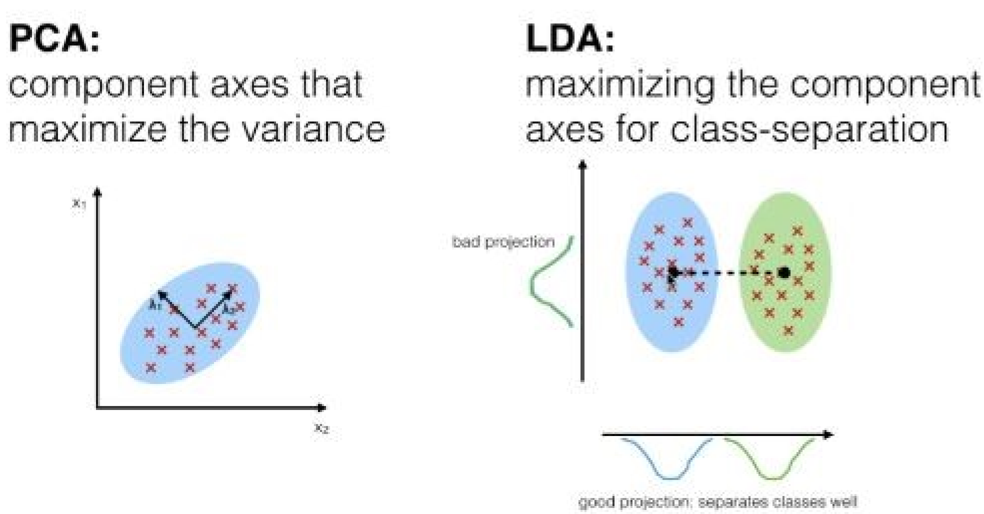
```python
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Logistic Regression (Train set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = X_test, Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = 
                     np.arange(start = X_set[:, 1].min() - 1, stop = 
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.rave
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'bl
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Logistic Regression (Test set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.

[[1 1]
 [3 3]
 [2 2]
 [1 1]
 [2 2]
 [1 2]
 [1 1]
 [3 3]]
[[14  0  0]
 [ 1 15  0]
 [ 0  0  6]] 0.9722222222222222
```



```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```

Chapter 24 - Linear Discriminant Analysis (LDA)



LDA is similar to PCA, but it differs because in addition to finding the component axes with LDA,

we are also interested in the axes that **maximise the separation between multiple classes**. The goal of LDA is to project a feature space of n-dimensions onto a smaller subspace of k-dimensions while maintaining the class-discriminatory information.

PCA is unsupervised by LDA is supervised because of the relation to the dependent variable.

1. Compute the d-dimensional mean vectors for the different classes from the dataset.
2. Compute the scatter matrices (in-between-class and within-class scatter matrix).
3. Compute the eigenvectors ($\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_d$) and corresponding eigenvalues ($\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2, \ldots, \boldsymbol{\lambda}_d$) for the scatter matrices.
4. Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a $d \times k$ dimensional matrix $\boldsymbol{W}$ (where every column represents an eigenvector).
5. Use this $d \times k$ eigenvector matrix to transform the samples onto the new subspace. This can be summarized by the matrix multiplication: $\boldsymbol{Y} = \boldsymbol{X} \times \boldsymbol{W}$ (where $\boldsymbol{X}$ is a $n \times d$-dimensional matrix representing the n samples, and $\boldsymbol{y}$ are the transformed $n \times k$-dimensional samples in the new subspace).

In [9]:

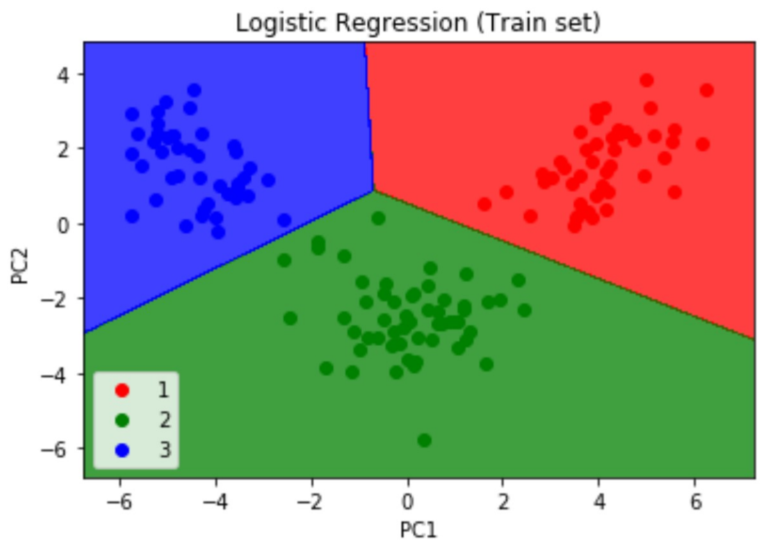
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("zzDatasets/Wine.csv")
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Apply LDA

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis(n_components=2)
X_train = lda.fit_transform(X_train, Y_train) # remember need the Y!
X_test = lda.transform(X_test)

### 4. Modelling (we use LogReg here but can use any classifier)

from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state=0)
classifier.fit(X_train, Y_train) # smaller the C, stronger the regula

### 5. Predictions

```

```
y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 6. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 7. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = X_train, Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = 
                     np.arange(start = X_set[:, 1].min() - 1, stop = 
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.rave
             alpha = 0.75, cmap = ListedColormap(('red', 'green','blu
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Logistic Regression (Train set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = X_test, Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = 
                     np.arange(start = X_set[:, 1].min() - 1, stop = 
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.rave
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'bl
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Logistic Regression (Test set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
```

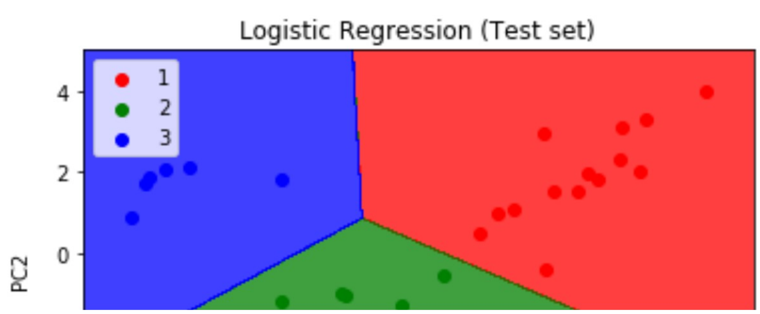
```
[[1 1]
 [3 3]
 [2 2]
 [1 1]
 [2 2]
 [2 2]
 [1 1]
 [3 3]]
[[14  0  0]
 [ 0 16  0]
 [ 0  0  6]] 1.0
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
```

```
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```




```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
```



Chapter 25 - Kernel PCA

Basically PCA but with kernel trick: we map the data to a higher dimension to extract new principal components. Remember that at this higher dimension, our dataset is now linearly separable (previously was not - imagine green data surrounded by red data in a circle). Better at dealing with nonlinear datasets.

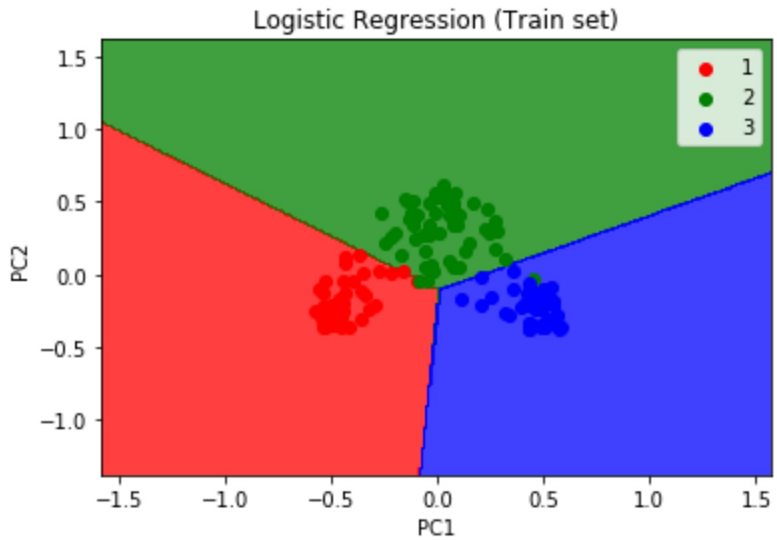
In [11]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("zzDatasets/Wine.csv")
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Apply PCA

from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=2, kernel='rbf') # no. of final extract
X_train = kpca.fit_transform(X_train)
X_test = kpca.transform(X_test)

### 4. Modelling (we use LogReg here but can use any classifier)

from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state=0)
classifier.fit(X_train, Y_train) # smaller the C, stronger the regula

### 5. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 6. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
```

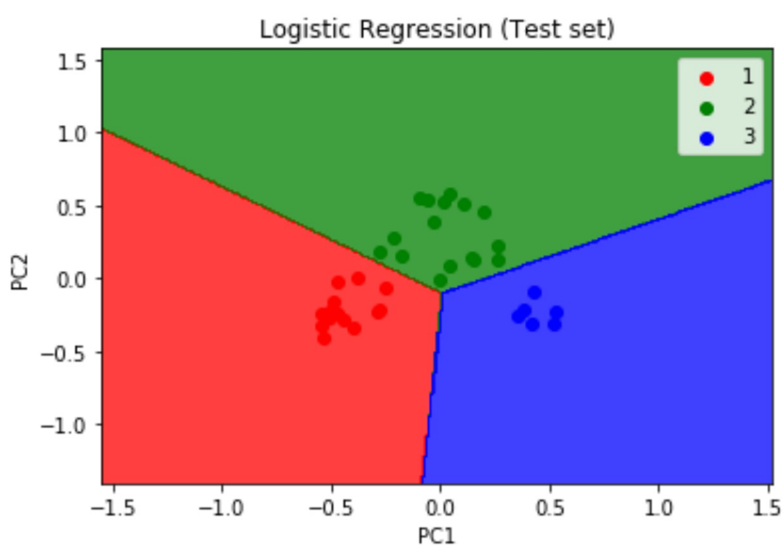
```
print(cm,ac)

### 7. Visualisations

from matplotlib.colors import ListedColormap
X_set, y_set = X_train, Y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = 
                     np.arange(start = X_set[:, 1].min() - 1, stop = 
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.rave
             alpha = 0.75, cmap = ListedColormap(('red', 'green','blu
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Logistic Regression (Train set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()

from matplotlib.colors import ListedColormap
X_set, y_set = X_test, Y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = 
                     np.arange(start = X_set[:, 1].min() - 1, stop = 
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.rave
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'bl
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = Liste
plt.title('Logistic Regression (Test set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth 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 point
s.

[[1 1]
 [3 3]
 [2 2]
 [1 1]
 [2 2]
 [2 2]
```

```
 [1 1]
 [3 3]]
[[14  0  0]
 [ 0 16  0]
 [ 0  0  6]] 1.0
```

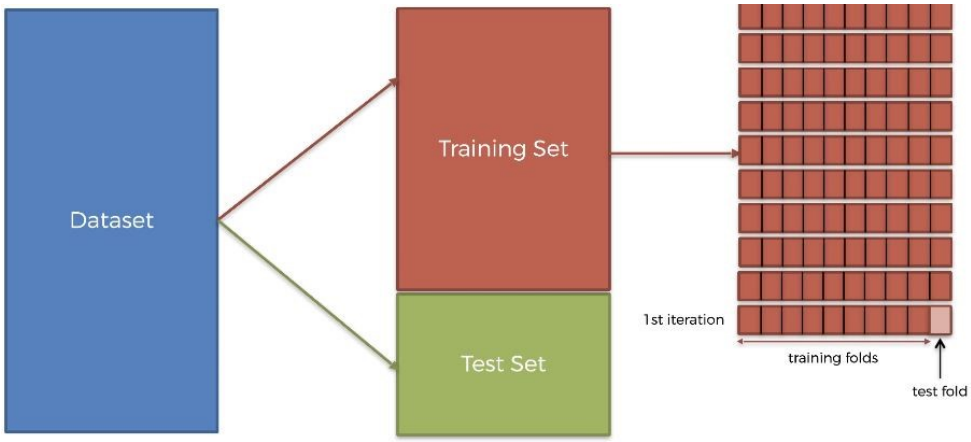



```
'c' argument looks like a single numeric RGB or RGBA sequence, which s
hould be avoided as value-mapping will have precedence in case its len
gth matches with 'x' & 'y'.  Please use a 2-D array with a single row
if you really want to specify the same RGB or RGBA value for all point
s.
'c' argument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



Chapter 26 - Model Selection

K-Fold Cross Validation



The best way to measure the performance of your model either via accuracy (classification) or RMSE (regression).

It creates a certain number of trained test folds (default 10) on which you will train your ML model on each of the trained fold and at the same time test it separately on the test fold.

Since you have 10 trained test folds, you will end up with 10 different accuracies on different test sets. The end result will be the average of these tests. This is better as it ensures you don't get lucky on one test set.

Grid Search

To find the best hyperparameters.

Note that for EACH run of the hyperparameters, we will apply K-fold CV each time.

In [18]:
```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("C:/Users/jrado/Desktop/Hackwagon/Udemy Course/Sup
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
```

```
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

### 3. Modelling

from sklearn.svm import SVC
classifier = SVC(kernel='rbf', random_state=0)
classifier.fit(X_train,Y_train)

### 4. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Applying K-Fold Cross Validation

from sklearn.model_selection import cross_val_score
accuracies = cross_val_score(estimator=classifier, X=X_train, y=Y_tra
print(accuracies)
print("Accuracy: {:.2f}%".format(accuracies.mean()*100))
print("Standard Deviation: {:.2f}%".format(accuracies.std()*100)) # g

### 7. Grid Search
# go to sklearn website to see what parameters even exist

from sklearn.model_selection import GridSearchCV
parameters = [
    {'C': [0.25, 0.5, 0.75, 1], 'kernel': ['linear']},
    {'C': [0.25, 0.5, 0.75, 1], 'kernel': ['rbf'], 'gamma': [0.1, 0.2
               ] # the ones you want to test. 2 dicts becos the linear
grid_search = GridSearchCV(estimator = classifier, # estimator is you
                           param_grid = parameters,
                           scoring = 'accuracy', # scoring is the metr
                           cv = 10,
                           n_jobs = -1) # use all your processors sinc
grid_search.fit(X_train, Y_train)
best_accuracy = grid_search.best_score_ # to see which is best accura
best_parameters = grid_search.best_params_ # and best parameters
print("Accuracy: {:.2f}%".format(best_accuracy*100))
print("Best Parameters:", best_parameters)
```

```
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
```

```
 [0 0]
 [1 1]]
[[64  4]
 [ 3 29]] 0.93
[0.8        0.96666667 0.8        0.96666667 0.86666667 0.86666667
 0.9        0.93333333 1.         0.93333333]
Accuracy: 90.33%
Standard Deviation: 6.57%
Accuracy: 90.67%
Best Parameters: {'C': 0.5, 'gamma': 0.6, 'kernel': 'rbf'}
```

Chapter 27 - XGBoost

Can be used for both regression and classification.

In [25]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv("zzDatasets/Data.csv")
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0

### 3. Modelling

from xgboost import XGBClassifier # if regression use XGBRegression
classifier = XGBClassifier() # actually not much to tune since the de
classifier.fit(X_train, Y_train)

### 4. Predictions

y_pred = classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), Y_test.reshape(l

### 5. Confusion Matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, y_pred)

from sklearn.metrics import accuracy_score
ac = accuracy_score(Y_test, y_pred)
print(cm,ac)

### 6. Applying K-Fold Cross Validation

from sklearn.model_selection import cross_val_score
accuracies = cross_val_score(estimator=classifier, X=X_train, y=Y_tra
print(accuracies)
```

```
41 print("Accuracy: {:.2f}%".format(accuracies.mean()*100))
```

```
[[2 2]
 [2 2]
 [4 4]
 [4 4]
 [2 2]
 [2 2]
 [2 2]
 [4 4]]
[[84  3]
 [ 0 50]] 0.9781021897810219
[0.92727273 0.96363636 0.96363636 0.98181818 0.94545455 0.98181818
 0.94444444 0.98148148 1.         0.96296296]
Accuracy: 96.53%
Standard Deviation: 2.07%
```

Back to Deep Learning

Supervised	Artificial Neural Networks	Used for Regression & Classification
	Convolutional Neural Networks	Used for Computer Vision
	Recurrent Neural Networks	Used for Time Series Analysis
Unsupervised	Self-Organizing Maps	Used for Feature Detection
	Deep Boltzmann Machines	Used for Recommendation Systems
	AutoEncoders	Used for Recommendation Systems

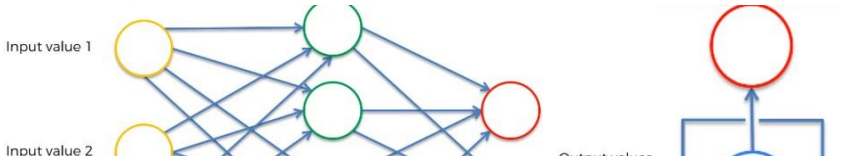
Chapter 28 - Recurrent Neural Network (RNN)

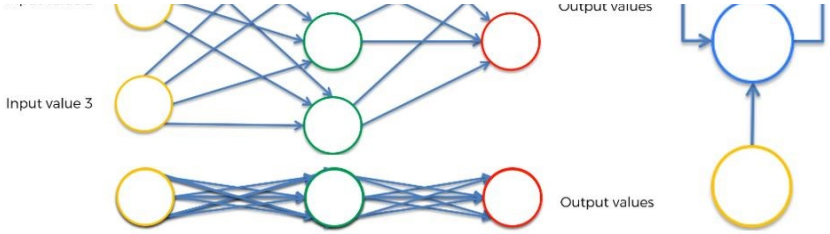
Probably can read here (https://www.superdatascience.com/blogs/the-ultimate-guide-to-recurrent-neural-networks-rnn).

The key thing about ANN are the weights that represent long-term memory, where the weights are memorised forever and no matter when you run it, it'll always be the same. OTOH, **RNNs are about short-term memory**.

For general ANNs, it is assumed that successive inputs are independent of each other. This is not true for RNN's sequential data (https://towardsdatascience.com/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90).

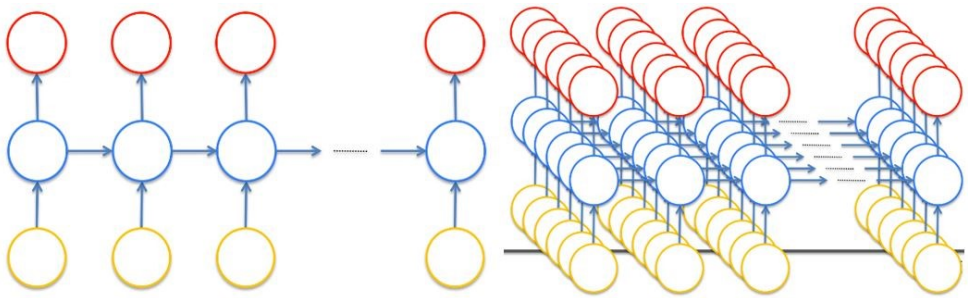
So as below, we take the original ANN, squash it into a single line, flip it to vertical, and then change the middle part to feedback into itself.

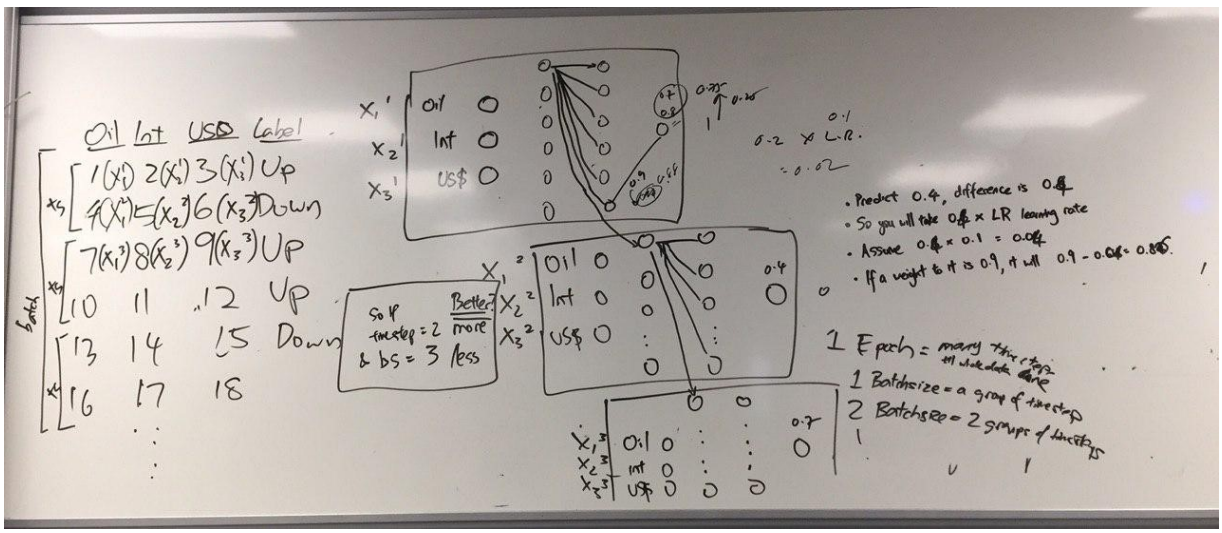


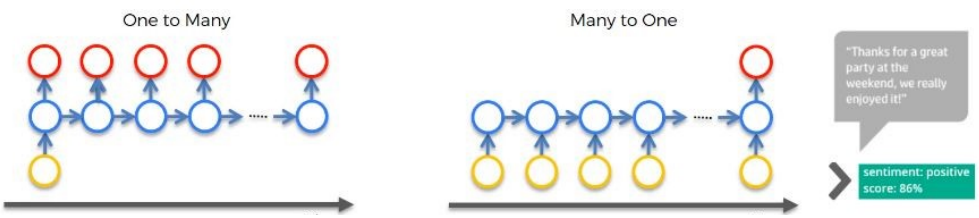


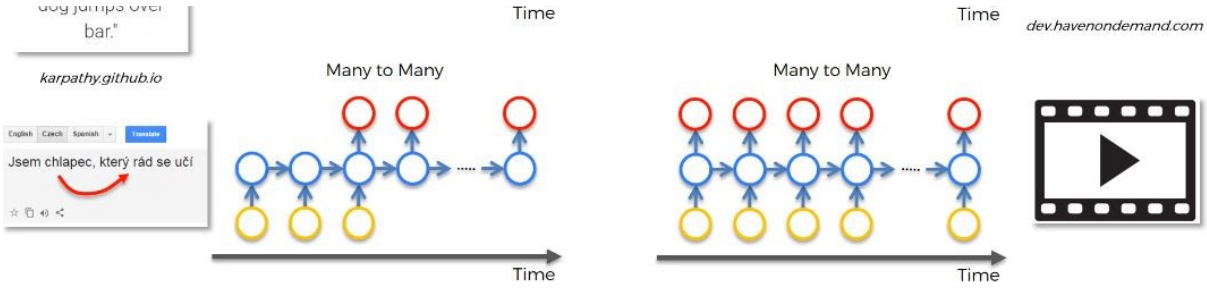
As below, the RNN is normally represented on the left, but actually is the one on the right. Each circle is actually a whole layer of neurons. But we want to visualise simplicity. Hence, the neurons are connecting to themselves through time; they have some sort of memory (short-term memory) that remembers what was in that neuron just previously. This allows them to pass information to themselves in the future and analyse things. See here for more information (https://machinelearningmastery.com/rnn-unrolling/) and also here for beginner version (https://towardsdatascience.com/rnn-simplified-a-beginners-guide-cf3ae1a8895b) or a simplified video version (https://www.youtube.com/watch?v=UNmqTiOnRfg). Also **important to note that the sequential units are the same unit at different points of time and are not cascading units**.

BIG NOTE: Remember prof drew that picture with the squares? Each layer in the Udemy picture is each square / layer in prof's pic.







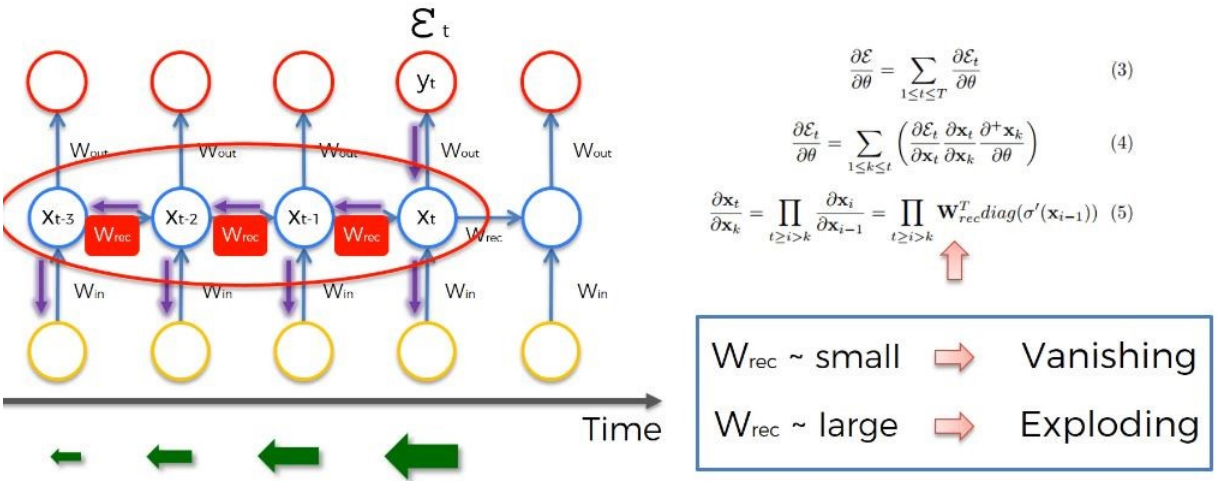


As above, there are one-to-many relationships (one input multiple outputs, ie one picture many words describing it), many-to-one relationships (many inputs to one output, ie many texts to one binary outcome), and many-to-many (many inputs to many outputs, ie translate many words to many words).

Vanishing Gradient

This is a problem inherent in RNNs.

During training, each layer will calculate the error / cost function. How do you backpropagate then? Also the more **time steps** you take the longer.



Ignore the math. But the **Wrec** connects the hidden layers to themselves in the unrolled temporal loop. To get from X(t-3) to X(t-2), we must multiply by Wrec. And because we keep multiplying many times by this small number, the value decreases very quickly. This means the gradient becomes lesser and this vanishing gradient, the harder it is for the network to update the weights, it is slower. Basically the weights in the earlier layers are updated slower, and therefore by the end of 1000 epochs you may not have the final results there and the nodes are left not very trained. But remember when you front propagate back, the untrained layers have to train the future layers so it becomes a vicious cycle. *Not sure what is happening? Me too.*

Some solutions are:

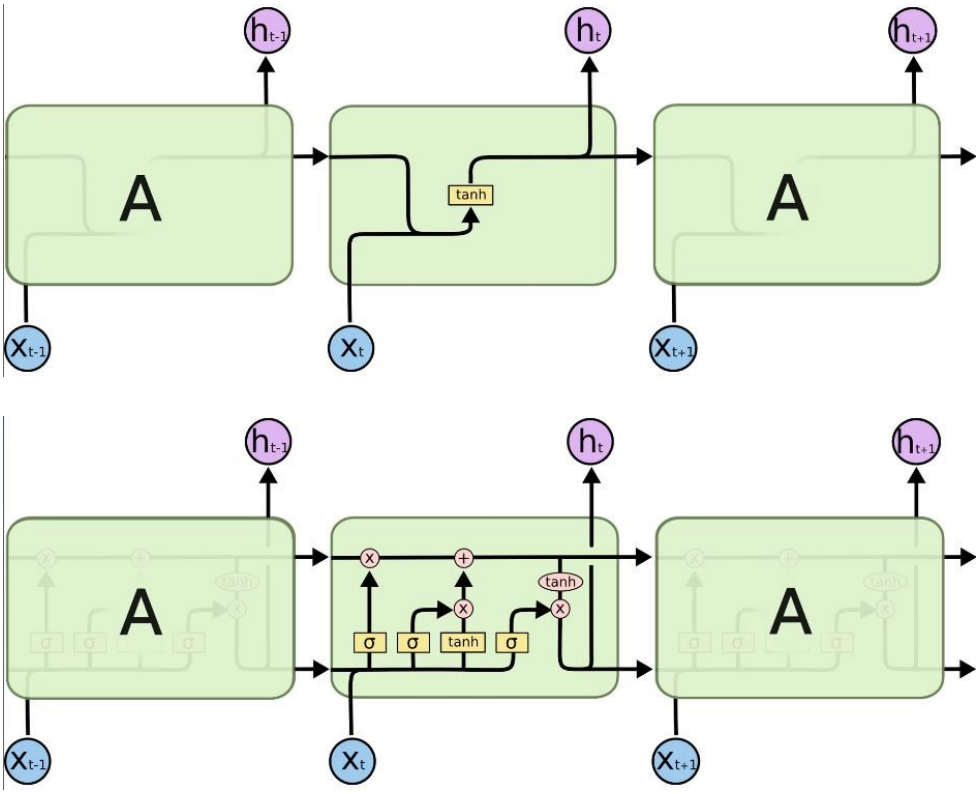
- Exploding Gradient
  - Truncated Backpropagation (stop backpropagating after a while - but not good)
  - Penalties
  - Gradient Clipping (a limit)
- Vanishing Gradient

- Weight Initialisation
- Echo State Networks
- Long Short-Term Memory Networks (LSTMs) <<< this!

LSTMs

Read here (https://colah.github.io/posts/2015-08-Understanding-LSTMs/). Read more here (http://karpathy.github.io/2015/05/21/rnn-effectiveness/).

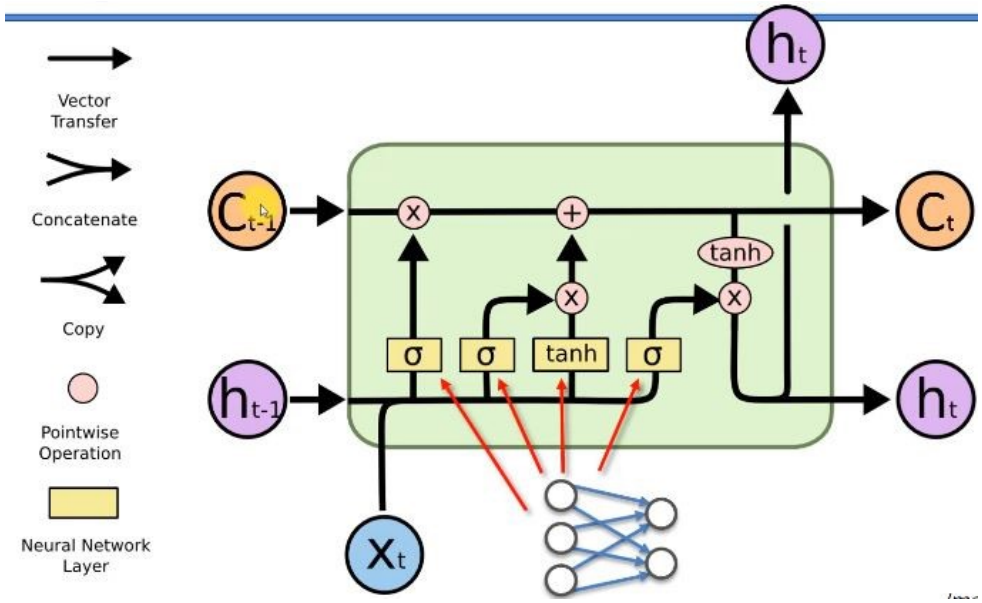
RECAP: For the vanishing gradient problem, as we propagate the error through the network it has to go through the unrolled temporal loop and as it does that, it goes through these hidden layers of neurons connected to themselves with a Wrec (recurrent weight), and because this is a weight that is applied many times on top of itself, that causes the gradient to decline rapidly, leading to weights on the very far left to be updated much slower than the weights on the far right. This creates a domino effect because the weights on the far left are important as they dictate the output of those layers which are the inputs of the far right layers. The whole training suffers.



In simplified information, to solve this we make Wrec = 1 (there is more to this actually). The first picture is a normal RNN. The problem lies with that tanh - as backpropagation, it is where the weights are applied / where Wrec is sitting. The second picture is the LSTM. When we say "Wrec=1", that is the top horizontal line which is also called the **memory cell**. It can go through time (time represented as per node) very freely, where sometimes it can be removed / erased with the X button and sometimes things may be added to it with the + button. That is why when you backpropagate you don't have the vanishing gradient.

Let us rephrase our picture. C is memory cell, h is output, X is input. These are all vectors of many numbers. The pointwise operations are called **valves**, with the top left X as forget valve, the middle X as memory valve, and right X as output valve (in literature will see as F / V / O

respectively).



To take it step by step:

- New value Xt coming in. Previous output's value h(t-1) coming in. Together they decide if the first sigmoid should output a value or not.
- Those two combine again to decide if the second sigmoid and tanh should output a value or not and to what extent.
- The C(t-1) flows on top and interacts with the two new results.
- Finally, the two inputs goes through the last sigmoid, and some fuck happens

In [2]:

```
### 1. Libraries

# Normal
import numpy as np # only can read arrays and not DFs
import matplotlib.pyplot as plt
import pandas as pd

# RNN
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout

### 2. Dataset

# Data
dataset_train = pd.read_csv('zzDatasets/Google_Stock_Price_Train.csv'
training_set = dataset_train.iloc[:,1:2].values

# Feature Scaling
from sklearn.preprocessing import MinMaxScaler # RNN always apply nor
sc = MinMaxScaler(feature_range=(0,1)) # 0,1 because we want it btw 0
training_set_scaled = sc.fit_transform(training_set)

# Making Timesteps (if timestep=60, means at time T, model will look
# we want to make X_train of 60 stock prices, and Y_train the predict
X_train = []
```

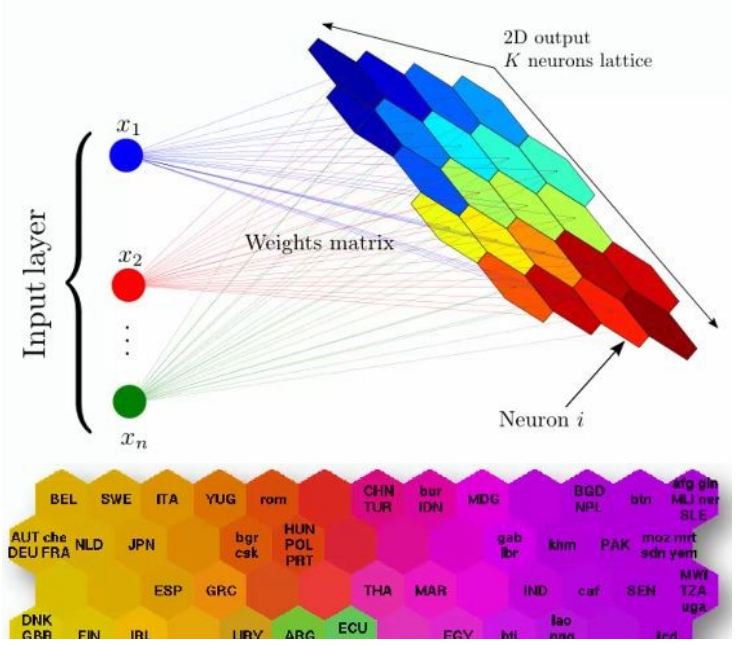
```
Y_train = []
for i in range(60,1258):
    X_train.append(training_set_scaled[i-60:i])
    Y_train.append(training_set_scaled[i,0])
X_train, Y_train = np.array(X_train), np.array(Y_train)

# Reshaping the Data (read RNN documentation - input needs to be 3D w
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1)

### 3. Modelling

# Initialise the RNN
regressor = Sequential()

# Add first LSTM layer + Dropout
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_tr
regressor.add(Dropout(0.2)) # dropout rate. 20% is good rate

# Second LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))

# Third LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))

# Fourth LSTM layer (penultimate before output)
regressor.add(LSTM(units=50)) # return_seq=False is default
regressor.add(Dropout(0.2))

# Output layer
regressor.add(Dense(units=1))

# Compiling
regressor.compile(optimizer='adam', loss='mean_squared_error') # good

# Fitting the RNN to the Training Set
regressor.fit(X_train, Y_train, epochs=100, batch_size=32)

### 4. Predictions

dataset_test = pd.read_csv('zzDatasets/Google_Stock_Price_Test.csv')
real_stock_price = dataset_test.iloc[:,1:2].values # rea; stock price

dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open'
inputs = dataset_total[len(dataset_total) - len(dataset_test) - 60:].
inputs = inputs.reshape(-1,1)
inputs = sc.transform(inputs)

X_test = []
for i in range(60,80):
    X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

predicted_stock_price = regressor.predict(X_test)
```

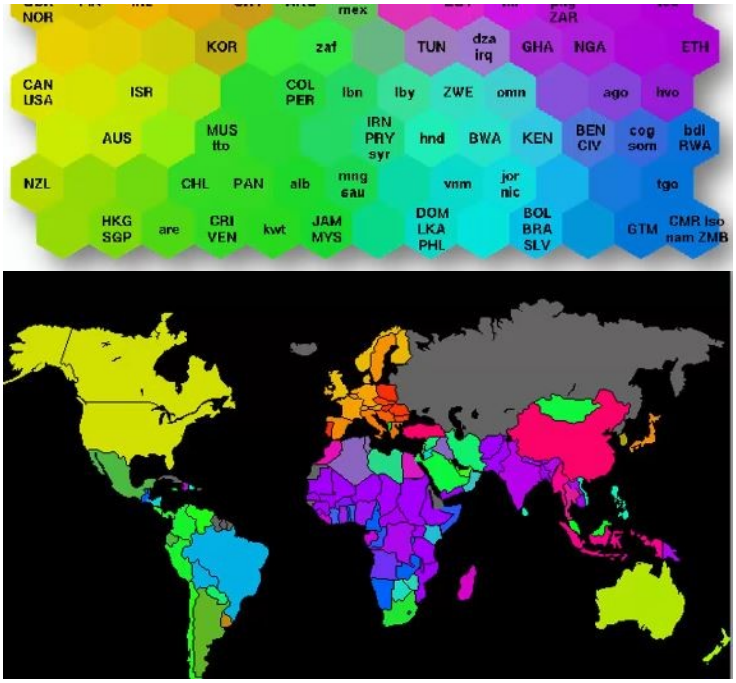
```
predicted_stock_price = sc.inverse_transform(predicted_stock_price)

### 5. Visualisations
plt.plot(real_stock_price, color='red', label='Real Google Stock Pric
plt.plot(predicted_stock_price, color='blue', label='Predicted Google
plt.title('Google Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Google STock Price')
plt.legend()
plt.show()
```

```
Epoch 1/100
38/38 [==============================] - 2s 44ms/step - loss: 0.0330
Epoch 2/100
38/38 [==============================] - 2s 42ms/step - loss: 0.0067
Epoch 3/100
38/38 [==============================] - 2s 43ms/step - loss: 0.0053
Epoch 4/100
38/38 [==============================] - 2s 45ms/step - loss: 0.0051
Epoch 5/100
38/38 [==============================] - 2s 45ms/step - loss: 0.0053
Epoch 6/100
38/38 [==============================] - 2s 45ms/step - loss: 0.0045
Epoch 7/100
38/38 [==============================] - 2s 45ms/step - loss: 0.0052:
0s - loss: 0.
Epoch 8/100
38/38 [==============================] - 2s 47ms/step - loss: 0.0047
Epoch 9/100
38/38 [==============================] - 2s 52ms/step - loss: 0.0046
```

Chapter 29 - Self Organising Maps

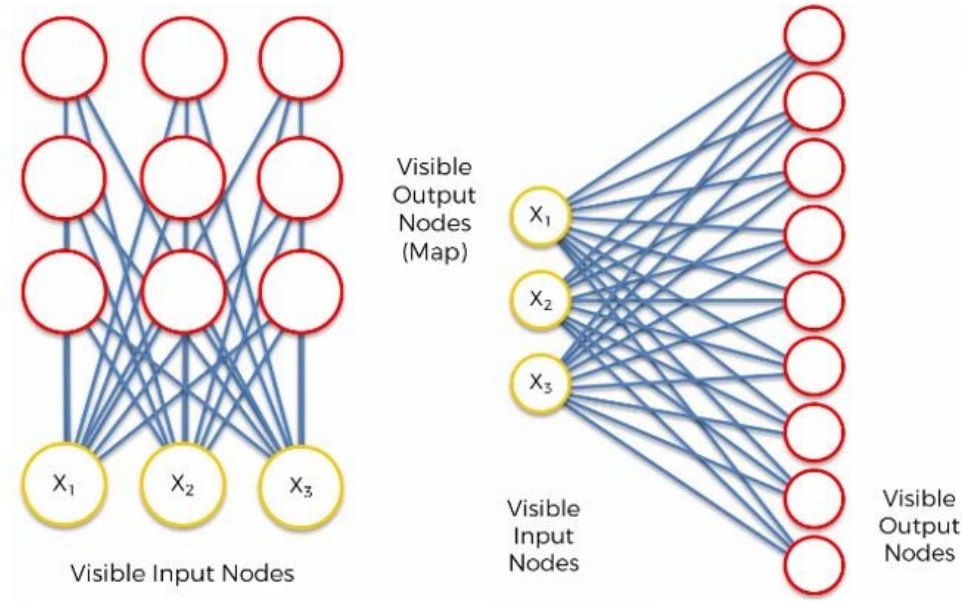
Best for reducing dimensionality. Like the picture below, you transform millions of columns into a 2D representation map. The second picture shows an example where many countries are clustered into groups based on prosperty / poverty indicators (top left best bottom right worst). Third picture uses those colours and put on world map, quite fitting general knowledge.





Learning Mechanic

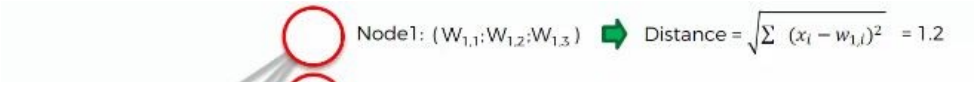
We have 3 input nodes and 9 output nodes. It might look simple but don't let it fool you. The input has 3 columns and millions of rows and is actually 3D. The output is 2D.

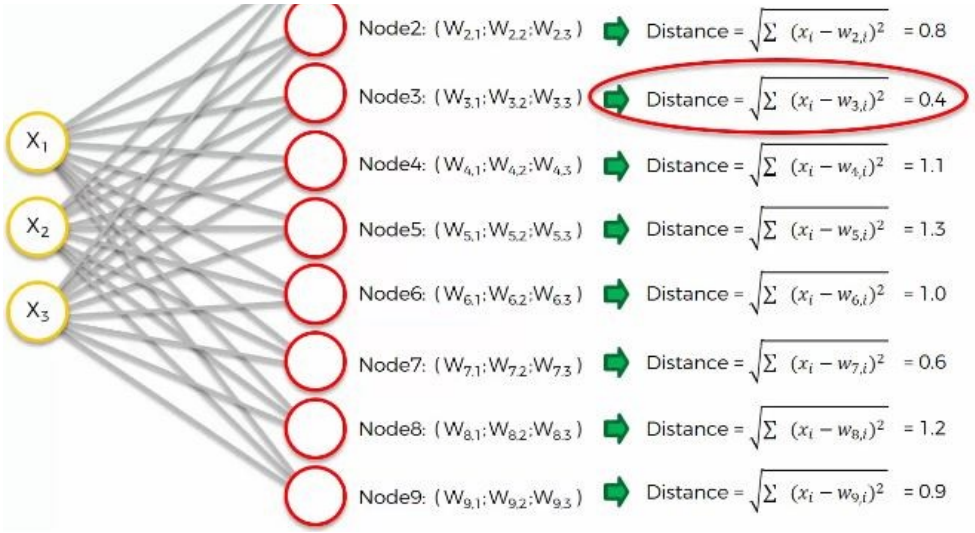


NOTE: The jargon below might have seen before, but actually different meaning in SOMs!

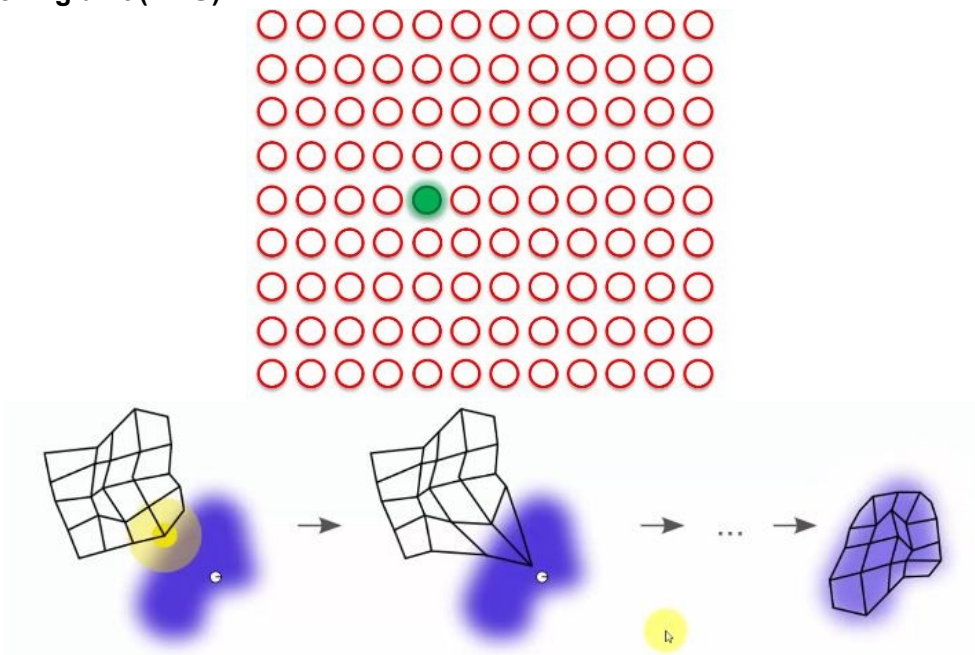
picture

Weights have different meaning. They are actually coordinates for the output node. So if there are 3 columns there are 3 input nodes and if 20 columns there are 20 input nodes, but that output node represents a point in the input space which has 3 or 20 weights respectively. These weights are assigned randomly but close to 0 but not 0 (just like KNNs!).

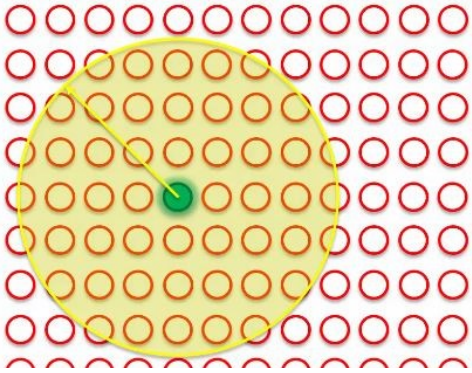




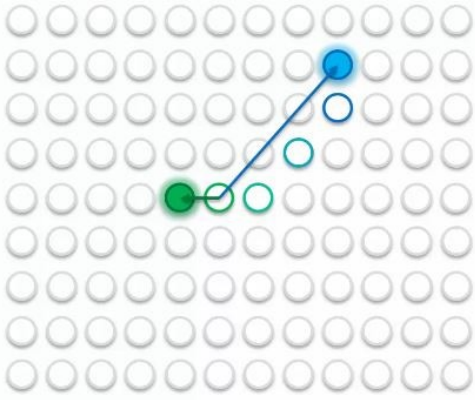
Now we will have a competition: we are going to go through the rows in the dataset and find out which of the output nodes is closest to each of ours in the dataset. Take each row and calculate it against every output node to get the distance, and choose the smallest distance and that is our **best matching unit (BMU)**.



After finding the BMU for the first row, this BMU's weights will get updated to be even closer to the first row in the dataset (like centroid moving towards). In other words, the SOM is moving closer to the data point. The map self-organises to our input data. Note that some of the closer output nodes are also dragged closer as well.



That is not all - there will be a radius around that BMU, and all these nodes will have their weights updated - the closer they are to the BMU, the bigger the change. AKA the closer you are to the BMU, the harder you get pulled to that row.



Repeat. Remember that the closer a node is to a BMU, the harder it gets pulled to that BMU's direction.

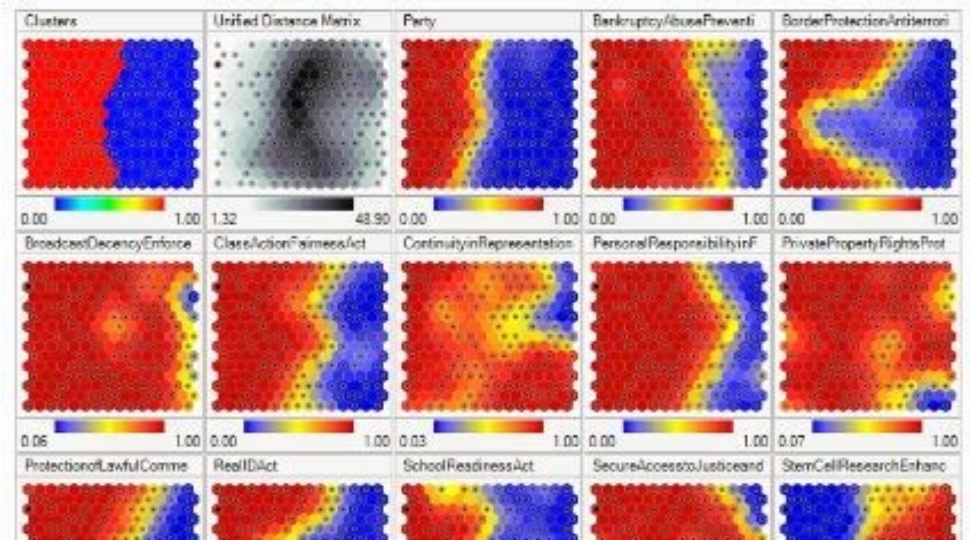
After each epoch, the radius become smaller. Hence only the closer nodes to each BMU get pulled only. Thus the process becomes more accurate per epoch.

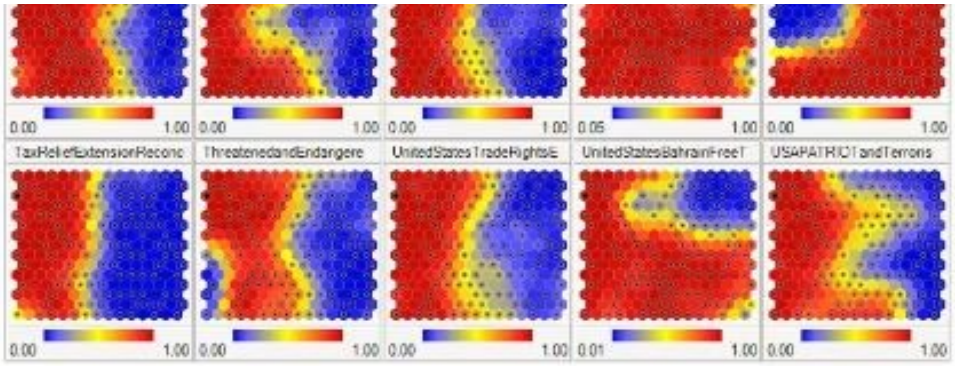
Pointers

- SOMs retain topology of the input set (same shape as inputs)
- SOMs reveal correlations that are not easily identified
- SOMs classify data without supervision
- No target vector = no backpropagation
- No lateral connections between ouput nodes (no connections between output nodes like you see in ANNs, no activation functions etc.)

Interpretation

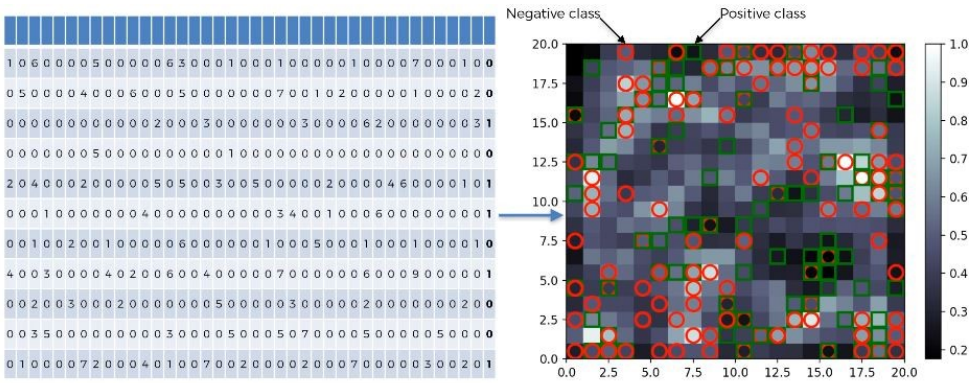
Below are SOM of voting results of US Congress members (over 500+ members). Each picture is a Y/N vote on a question they were voting on. The maps then discovered which members of Congress are close and which are dissimilar in voting patterns.





Watch [video (https://www.udemy.com/course/deeplearning/learn/lecture/6905328#overview)](https://www.udemy.com/course/deeplearning/learn/lecture/6905328#overview).

Others



STEP 1: We start with a dataset composed of n_features independent variables.

STEP 2: We create a grid composed of nodes, each one having a weight vector of n_features elements.

STEP 3: Randomly initialize the values of the weight vectors to small numbers close to 0 (but not 0).

STEP 4: Select one random observation point from the dataset.

STEP 5: Compute the Euclidean distances from this point to the different neurons in the network.

STEP 6: Select the neuron that has the minimum distance to the point. This neuron is called the winning node.

STEP 7: Update the weights of the winning node to move it closer to the point.

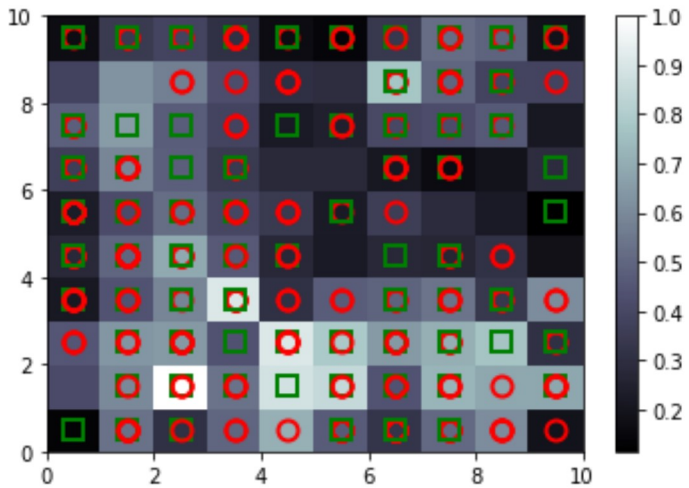
STEP 8: Using a Gaussian neighbourhood function of mean the winning node, also update the weights of the winning node neighbours to move them closer to the point. The neighbourhood radius is the sigma in the Gaussian function.

STEP 9: Repeat Steps 1 to 5 and update the weights after each observation (Reinforcement Learning) or after a batch of observations (Batch Learning), until the network converges to a point where the neighbourhood stops decreasing.

In [16]:

```
### 1. Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset

data = pd.read_csv('zzDatasets/Credit_Card_Applications.csv')
X = data.iloc[:, :-1].values
Y = data.iloc[:,-1].values

from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0,1))
sc.fit_transform(X)
```

```
16
17 ### 3. Modelling
18
19 from minisom import MiniSom
20 som = MiniSom(x=10, y=10, input_len=15, sigma=1.0, learning_rate=0.5)
21 som.random_weights_init(X) # initialise weights
22 som.train_random(data=X, num_iteration=100) # the method for step 4 t
23
24 ### 4. Visualisation
25 # MID = mean interneuron distance -> higher it is the more it is outl
26 from pylab import bone, pcolor, colorbar, plot, show
27 bone()
28 pcolor(som.distance_map().T)
29 colorbar()
30 markers = ['o','s']
31 colors = ['r','g']
32 for i,x in enumerate(X):
33     w = som.winner(x)
34     plot(w[0]+0.5,
35          w[1]+0.5,
36          markers[Y[i]],
37          markeredgecolor=colors[Y[i]],
38          markerfacecolor='None',
39          markersize=10,
40          markeredgewidth=2)
41 show()
42
43 ### 5. Finding the Frauds
44 mappings = som.win_map(X)
45 frauds = np.concatenate((mappings[(8,1)], mappings[(6,8)]), axis=0)
46 frauds = sc.inverse_transform(frauds)
47 frauds
```



Out[16]:
```
array([[3.91985210e+12, 0.00000000e+00, 2.11315500e+03, 4.20000000e+0
1,
        5.00000000e+00, 1.40000000e+01, 9.00000000e+00, 0.00000000e+0
0,
        0.00000000e+00, 1.00000000e+00, 1.34000000e+02, 1.00000000e+0
0,
        5.00000000e+00, 4.00000000e+05, 1.06000010e+07],
       [3.92588755e+12, 1.00000000e+00, 1.41557000e+03, 1.15500000e+0
```

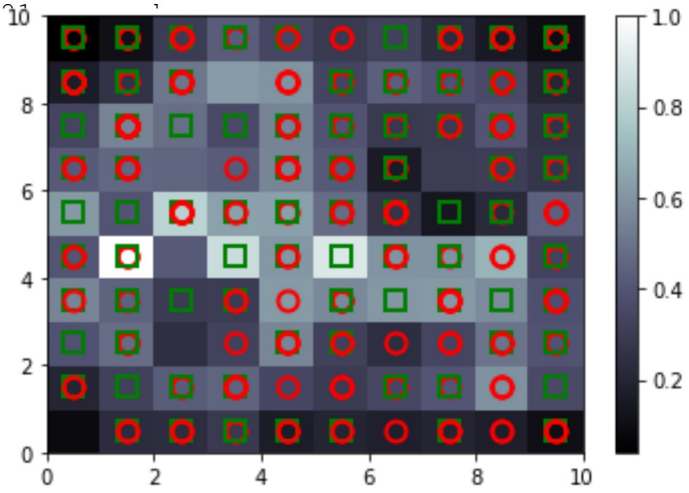
```
2,
        3.00000000e+00, 4.00000000e+01, 6.50000000e+01, 1.14000000e+0
0,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+0
0,
        5.00000000e+00, 2.80000000e+05, 1.01000010e+07],
       [3.92574046e+12, 0.00000000e+00, 2.35255500e+03, 1.05000000e+0
2,
        5.00000000e+00, 1.40000000e+01, 9.00000000e+00, 0.00000000e+0
0,
        0.00000000e+00, 1.00000000e+00, 4.02000000e+02, 0.00000000e+0
0,
        5.00000000e+00, 0.00000000e+00, 2.01000010e+07],
       [3.92584360e+12, 1.00000000e+00, 1.31050000e+03, 8.12000000e+0
0,
        5.00000000e+00, 5.30000000e+01, 3.30000000e+01, 8.26500000e+0
0,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+0
0,
        5.00000000e+00, 5.60000000e+05, 3.65000010e+07],
       [3.92548599e+12, 1.00000000e+00, 1.09969500e+03, 7.70000000e+0
1,
        5.00000000e+00, 7.90000000e+01, 3.30000000e+01, 1.89525000e+0
1,
        0.00000000e+00, 1.00000000e+00, 6.70000000e+01, 0.00000000e+0
0,
        5.00000000e+00, 1.60000000e+05, 2.20000100e+06],
       [3.92591627e+12, 0.00000000e+00, 2.69569500e+03, 2.11120000e+0
2,
        3.00000000e+00, 1.44000000e+02, 6.50000000e+01, 2.28000000e+0
2,
        1.00000000e+00, 1.00000000e+00, 9.38000000e+02, 0.00000000e+0
0,
        5.00000000e+00, 0.00000000e+00, 2.30100001e+08]])
```

In [10]:

```
#### Hybrid Deep Learning Model (ANN + SOM)

#### SOM

### 1. Libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

### 2. Dataset
data = pd.read_csv('zzDatasets/Credit_Card_Applications.csv')
X = data.iloc[:, :-1].values
Y = data.iloc[:,-1].values

from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0,1))
sc.fit_transform(X)

### 3. Modelling
from minisom import MiniSom
som = MiniSom(x=10, y=10, input_len=15, sigma=1.0, learning_rate=0.5)
```

```
som.random_weights_init(X)
som.train_random(data=X, num_iteration=100)

### 4. Visualisation
from pylab import bone, pcolor, colorbar, plot, show
bone()
pcolor(som.distance_map().T)
colorbar()
markers = ['o','s']
colors = ['r','g']
for i,x in enumerate(X):
    w = som.winner(x)
    plot(w[0]+0.5,
         w[1]+0.5,
         markers[Y[i]],
         markeredgecolor=colors[Y[i]],
         markerfacecolor='None',
         markersize=10,
         markeredgewidth=2)
show()

### 5. Finding the Frauds
mappings = som.win_map(X)
frauds = np.concatenate((mappings[(8,1)], mappings[(6,8)]), axis=0)
frauds = sc.inverse_transform(frauds)

#### ANN

### 1. Dataset

customers = data.iloc[:,1:].values # the X

is_fraud = np.zeros(len(data)) # the Y
for i in range(len(data)):
    if data.iloc[i,0] in frauds:
        is_fraud[i] = 1

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
customers = sc.fit_transform(customers)

### 3. Modelling

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

classifier = Sequential()
classifier.add(Dense(units=2, kernel_initializer='uniform', activatio
classifier.add(Dense(units=1, kernel_initializer='uniform', activatio

classifier.compile(optimizer='adam', loss='binary_crossentropy', metr

classifier.fit(customers, is_fraud, batch_size=1, epochs=2)

### 4. Predictions

```

```
y_pred = classifier.predict(customers)
y_pred = np.concatenate((data.iloc[:, 0:1].values, y_pred), axis=1)
y_pred = y_pred[y_pred[:,1].argsort()]
```




```
Epoch 1/2
690/690 [==============================] - 1s 793us/step - loss: 0.431
6 - accuracy: 0.9971
Epoch 2/2
690/690 [==============================] - 0s 715us/step - loss: 0.137
9 - accuracy: 1.0000
```

Out[10]:
```
array([[1.56430560e+07, 5.57094812e-03],
       [1.56009750e+07, 6.38449192e-03],
       [1.57139830e+07, 6.48319721e-03],
       ...,
       [1.56577780e+07, 3.03799868e-01],
       [1.56825760e+07, 3.03799868e-01],
       [1.56214230e+07, 3.03799868e-01]])
```

In [ ]:

In [ ]: