**Linear regression theory**

* Regression analysis is used in supervised machine learning
* A linear regression: linear approximation of a causal relationship between two or more variables
* Machine generated alternative text:
  Y = ßo + ß1X1 + e 
* Process
  + Get sample data
  + Design a model that works for that sample
  + Make predictions for the whole population
* Variables
  + Dependent (y): what you are trying to predict
  + Independent (x) -> predictors
  + Dependent variable Y is a function of the independent variables x1 to xn
  + Y must be dependent on x, change in x must also show a change in y
* Correlation vs regression
  + Correlation measure a relationship, regression measures how one variable affects the other
  + Correlation does not capture causality but the degree of relationship, regression captures cause and effect
  + Correlation between x and y is the same as y and x, in regression it is only one way
  + Correlation is a single point, regression is a line that minimizes the distance between points
* Sum of squares total (SST/TSS): squared differences between the observed and the mean, total variability
* Sum of squares regression (SSR/ESS): sum of the differences between the predicted value and the mean of the dependent variable, explained variability
* Sum of squares error (SSE/RSS): difference between the observed value and predicted value, unexplained variability
* SST = SSR + SSE
* R squared = 0 : regression line explains none of the variability, = 1: explains all of the variability
  + No good R squared: usually between 0.2 and 0.9 in social sciences
  + Depends on a lot of external variables which must be considered
  + The more factors you include, the higher the R squared

**Linear regression in Python**

* Packages required in Python: numpy (multidimentional arrays), pandas (enhances numpy, organise data), scipy (comprises numpy, pandas, marplotlib), matplotlib (visualisation), seaborn(visualisation library based on matplotlib), sklearn (machine learning library), statsmodels.api (built on top of numpy and scipy and provides good summaries)

|  |
| --- |
| import numpy as np  import pandas as pd  import scipy  import statsmodels.api as sm  import matplotlib.pyplot as plt  import seaborn as sns  import sklearn  sns.set() |

sns.set() will override all graphs made in plt as seaborn (like a skin for matplotlib)

* Loading data into python and creating a linear regression:

|  |
| --- |
| data = pd.read\_csv("name.csv") |

* + with pandas this will automatically be converted to a data frame

|  |
| --- |
| data.describe() |

* + To get the summary of the most useful statistics
  + Define variables to be used as y and x:

|  |
| --- |
| y = data ["column1"] |
| x1 = data ["column2"] |
| plt.scatter(x1,y)  plt.xlabel("column2", fontsize = 20)  plt.ylabel("column1", fontsize = 20)  plt.show() | |

* + Add a constant (what the B0 is multiplied by which is \*1)

|  |
| --- |
| x = sm.add\_constant(x1) |

* + Make a variable that will contain the result of the output of the ordinary least squares (OLS) regression

|  |
| --- |
| result = sm.OLS(y,x).fit() |

* + Fit will provide the fit line for the model
  + results.summary()
  + Graph the scatterplot with the regression line

|  |
| --- |
| plt.scatter(x1,y)  yhat = x1\*223.1787+101900  fig = plt.plot(x1,yhat, lw=4, c='orange', label ='regression line')  plt.xlabel('Size', fontsize = 20)  plt.ylabel('Price', fontsize = 20)  plt.show() |

* The 223.1787 is from the summary table under coef size (size is the variable name)
* The 101900 is the coef constant
* Understanding the OLS summary table
  + Coef constant: b0
  + The other coefficient: the predictor (b1)
  + Standard error: accuracy of prediction
  + t: t statistic
  + p: pvalue, if below 0.05 it is a significant predictor
  + In linear regressions: H0: B=0 , does it help explain the variability we have in the data?
  + Dep variable: variable that we are trying to predict
  + Model: the regression model that is being used
  + OLS (ordinary least squares) is the most common method to estimate the linear regression
  + R Squared: shows how much is explained by the variable
    - SSR/SST (variability explained/total variability)
    - How well the model fits the data
  + F statistic: overall significance of the model, check if significant by the below p value, but the lower the f statistic the lower the significance level

**Multiple linear regression**

* Multiple regressions address the higher complexity of problems, more than one independent variable
* Machine generated alternative text:
  = bo + blX1 + b2X2 + + bkXk 
* No way to visually represent all variables because it becomes 3+ dimensional
* It's about the best fitting model, not line of best fit -> least SSE and better fitting model
* Adjusted R squared: smaller than the R squared as it penalises excessive use of variables
* If the variable ruins the model (reduces R squares and is not significant), this variable should be removed
* In Python:
  + Declare the y the same, for x load as a dataframe:

|  |
| --- |
| x1 = data [["var1", "var2"]] |

* + The rest is the same
* Assumptions to consider before performing a regression analysis:
  + Linearity
    - If x and y scatterplot represent a linear regression, then it is suitable
    - If a curved line is seen more than linear, linear regression would not be appropriate
    - Can either run a non-linear regression or run exponential and log transformations
  + No endogeneity
    - Prohibition of the link between the independent variables and the errors
    - Omitted variable bias: happens when you forget to include a relevant variable
  + Normality and homoscedasticity
    - Assume that it is normally distributed
    - Zero mean: if the mean is not expected to be zero, then the line is not the best fitting one but having an intercept solves the problem (unusual to violate)
    - Homoscedasticity: errors should have equal variance between each other (if they are more close to the regression line on the left side than the right side, we cannot be sure that the model is good) -> check for omitted variable bias, look for outliers and try to remove them, try a log transformation
  + No autocorrelation
    - Plot all the residuals on a graph and look at patterns, if there aren't then it's fine
    - Durbin Watson test (included in the summaries in Python): 2: no autocorrelation, <1 and >3 is cause for alarm
    - No remedy, only thing is to avoid using a linear regression in such a setting
  + No multicollinearity
    - Don't want more than two variables to have high correlations between each other
    - If a can be almost fully represented by b, there is no point using variable b
    - Fixes: drop one of the two variables, transform into one variable, can keep them both but must be cautious
* Dummy variables for categorical data:
  + an imitation or copy that stands as a numerical substitute for categorical variables
  + How to include categorical variables into regressions
  + Map yes and no to 1 and 0

|  |
| --- |
| data = raw\_data.copy() |
| data["cat\_variable"] = data["cat\_variable"].map({"Yes":1, "No":0}) |

* + The rest is the same
  + Need to add another yhat variable when plotting with regression lines though
* Make predictions with linear regression
  + There is a statsmodels method which takes a data frame organised in the same way as x and then makes predictions

|  |
| --- |
| new\_data = pd.DataFrame({"const":1, "ind\_var": [score1, score2], "ind\_var2": [score1,score2]}) |
| new\_data = new\_data[["const", "ind\_var", "ind\_var2"]] |

* + The second line overrides the columns so they are in the correct order (ordered alphabetically by default)
  + Check that it looks the same as the model that we fed to the regression
  + predictions = results.predict(new\_data)

**Linear regression with sklearn**

* sklearn: very popular machine learning package
* Fast and efficient but prefers working with arrays
* Have to transform data frames into ndarray
* Very good documentation
* Very good at: regression, classification, clustering, support vector machines, dimensionality reduction
* Not so good at: deeplearning (Tensorflow is better)
* Simple linear regression:
  + Import libraries (+ from sklearn.linear\_model import LinearRegression) , load data, and declare variables same as before
  + Reshape into a matrix from a 1d object to a 2d, don’t actually change anything:

|  |
| --- |
| x\_matrix = x.values.reshape(-1,1) |

* + Reshape only needs to be done for linear regression and will work fine with 2+ independent variables
  + Independent variable: input/feature
  + Dependent variable: output/target
  + Supervised learning: inputs and targets

|  |
| --- |
| reg = LinearRegression() |

* + Reg is now an instance of the linear regression class

|  |
| --- |
| reg.fit(x\_matrix,y) |

* + X is input variable name and y is target variable name, order is important
  + The output of that is a LinearRegression that specifies some parameters:
    - Keep normalise = FALSE
    - Copy\_x = TRUE copies the inputs before fitting them, safety net against normalisation
    - Fit\_intercept = TRUE add a constant which had to be done manually before
    - n\_jobs = 1 uses 1 processor, for bigger data sets can change to more than 1 to increase computational power
  + R squared:

|  |
| --- |
| reg.score(x\_matrix,y) |

* + Coefficients:

|  |
| --- |
| reg.coef\_ |

* + Intercept

|  |
| --- |
| reg.intercept\_ |

* + Predictions

|  |
| --- |
| reg.predict(new\_input) |

* + Plot using the same code as before
* Multiple regression:
  + Sample = observation
  + Sample size of 84 = 84 observations

|  |
| --- |
| reg = LinearRegression() |
| reg.fit(x,y) |

* + reg.coef\_ will now output two or more results, the order will correspond to the order in which the columns were declared in the x variable
  + R squared:

|  |
| --- |
| reg.score(x,y) |

* + Adjusted R squared is not included in this module: can either google someone's code or write your own
* Feature selection through p values and F:
  + Simplifying model by removing variables that are not useful for the model
  + Feature selection simplified models and makes them easier to interpret
  + If a variable has a p value of above 0.05, can be disregarded
  + Not available so easily with sklearn
  + feature\_selection.f\_regression: creates simple linear regressions for each variable with the dependent variable and calculates the F statistic and returns the p values
    - Import: from sklearn.feature\_selection import f\_regression

|  |
| --- |
| f\_regression(x,y) |

* + - Two output arrays: first contains the F statistic for each variable, and second contains the p values

|  |
| --- |
| p\_values = f\_regression(x,y)[1] |
| p\_values = round(3) |

* + - Rounding is easier to see without the scientific notation, if the p value is 0. that means it is .000
    - Note that these are p values for individual variables and do not reflect the interconnection of two variables and their p values
  + Creating a summary tbable from the data:

|  |
| --- |
| reg\_summary = pd.DataFrame(data = x.columns.values, columns = ["Features"] |
| reg\_summary = ["Coefficients"] = reg.coef\_ |
| reg\_summary = ["p-values"] = p\_values.round(3) |
| reg\_summary |

* Feature scaling (normalisation/standartisation)
  + Transforming the data we are working with into a standard scale

|  |
| --- |
| from sklearn.preprocessing import StandardScaler |
| scaler = StandardScaler() |
| scaler.fit(x) |
| x\_scaled = scaler.transform(x) |

* + Scaled inputs allow to compare the coefficients of differently scaled variables together
  + Standardised coefficients = weights, the bigger the weight the bigger the impact on the regression
  + Intercept = bias
  + The closer the weight is to 0, the smaller the impact, the bigger the weight the bigger the impact
* Making predictions with standardised coefficients (weights)
  + Specify new values of the data in a new dataframe
  + The model expects to get values that are standardised, so predicting new values also need to be standardised so apply the scaler that was made earlier
  + reg.predict(data\_scaled)
* Underfitting and overfitting
  + Overfitting: the regression has focused on a particular data set so much that is has missed the point
  + Underfitting: the model has not captures the underlying logic of the data
  + e.g. a linear model for a non-linear regression would be underfitted, it works but does not capture the underlying logic (low accuracy)
  + e.g. an overfitted model would be one that touches every point and is so good at modelling the training data that it misses the point when new data is fed to it (random noise is captured)
  + Solution: split data set into two (training and test) 80/20 (training/testing) ratio is common

|  |
| --- |
| Import numpy as np  From sklearn.model\_selection import train\_test\_split |
| a = np.arange(1,101) |
| b = np.arange(501,601) |
| a\_train, a\_test, b\_train, b\_test = train\_test\_split(a, b, test\_size = 0.2, shuffle = FALSE/TRUE, random\_state=42) |
| a\_train.shape, a\_test.shape |

* + Shuffled data is usually preferred, but has to be shuffled in the same way (when you re-run the train\_test\_split, it will reshuffle every time you run it
  + Random\_state = 42 will give the same shuffled split every time you run (42 is just a joke within the community but any number can be used)
  + If include two arrays in the same argument, they will both be shuffled in the same way

**Practical example**

* Cleaning data
  + Once you have decided which variables to drop after looking at the describe() use drop:

|  |  |
| --- | --- |
| DataFrameName.drop(["col2"], axis=1 | Axis = 1 means you are dropping columns |

* + Check for missing values:

|  |
| --- |
| data.isnull() |
| data.isnull().sum() |

* + - It will return a dataframe with true/false, where true shows missing values
    - With .sum() it will sum all the missing values and present it in a table
    - Can remove the rows with missing values (if you are removing <5% of the observations, you are free to remove all that have missing values

|  |  |
| --- | --- |
| data = data.dropna(axis=0) | Axis=0 because we are dropping rows now |

* + Check for outliers and remove them
    - To check the data with plots:

|  |  |
| --- | --- |
| sns.distplot(data\_no\_mv["column\_name"]) | Check if normally distributed, if any outliers |

* + - Dealing with outliers: remove top 1% of observations

|  |
| --- |
| q = data\_no\_mv["column"].quantile(0.99) |
| data\_1 = data\_no\_mv[data\_no\_mv["column"]<q] |

* + - Can keep building up to data\_2 and data\_3 by removing outliers of other variables
    - In the end the index needs to be reset:

|  |  |
| --- | --- |
| data\_cleaned = data\_4.reset\_index(drop=True) | Drop=True completely forgets the old index |

* OLS assumptions
  + Linearity:
    - Create scatter plots for each of the variables between the predictor to analyse what kind of relationship it is, if does not look linear, one of more will need to be transformed (especially when facing exponential scatterplots)

|  |  |
| --- | --- |
| log\_dep\_var = np.log(data\_cleaned["dep\_var"] | Create a log of the dependent variable |
| data\_cleaned["log\_dep\_var"] = log\_dep\_var  data\_cleaned = data\_cleaned.drop(["dep\_var"], axis =1) | Add the new log to the data  Remove the original (not log) dep var |

* + Multicollinearity:
    - Sklearn does not have a way to check for this, need to use statsmodels

|  |
| --- |
| from statsmodels.stats.outliers\_influence import variance\_inflation\_factor  variables = data\_cleaned(["pred1", "pred2", "pred3"])  vif = pd.DataFrame()  vif["VIF"] = [variance\_inflation\_factor(variable.values, i) for i in range (variables.shape[1])]  vif["features"] = variables.columns |

* + - If VIF is 1, no multicollinearity
    - Between 1 and 5 is perfectly okay
    - Generally above 5-6 is unacceptable
    - But rarely find data that all features have VIF <5, judge on variables and exclude those that definitely seem too high
    - <https://statisticalhorizons.com/multicollinearity>

|  |
| --- |
| data\_no\_mult = data\_cleaned.drop(["col3"], axis = 1) |

* + Others
    - Normality and zero mean already assumed to be true
    - Homoscedasticity: if implemented a log transformation this will already be true
    - No autocorrelation: don’t need to worry if not time series or panel data
* Creating dummies for categorical variables
  + Pandas already has a get\_dummies method that identifies all categorical variables and creates dummies for them automatically
  + For each variable need to create n-1 categories dummies. Each category need to be 0 or 1 and if there are 7 categories, need 6 dummies (don’t need one for the last because it will be assumed that if all others are 0, this one has to be 1), if the 7th dummy is created the multicollinearity will be introduced

|  |
| --- |
| data\_with\_dummies = pd.get\_dummies(data\_no\_mult, drop\_first = True) |

* + This replaced all categorical variables with n-1
* Rearrange the columns in the data frame

|  |  |
| --- | --- |
| data\_with\_dummies.columns.values |  |
| cols = ["dep\_var", "pred1", "pred2", "dummy1", "dummy2"…]  data\_preprocessed = data\_with\_dummies[cols] |  |

* Linear regression model

|  |  |
| --- | --- |
| targets = data\_preprocessed["dep\_var"] | Declaring the dependent variable |
| inputs = data\_preprocessed.drop(["dep\_var"], axis=1) | Declaring the independent variables |
| from sklearn.preprocessing import StandardScaler | Importing scaler |
| scaler = StandardScaler() | Creating scaler |
| scaler.fit(inputs) | Fir the scaler onto the independent variables |
| inputs\_scaled = scaler.transform(inputs) | Transforms scaler |

* + Not usually recommended to standardise dummy variables as they lose their meaning, but scaling has not effect on their predictive power
  + Later will create a custom scaler that will only standardise the non-dummy variables

|  |  |
| --- | --- |
| from sklearn.model\_selection import train\_test\_split | import |
| x\_train, x\_test, y\_train, y\_test = train\_test\_split(inputs\_scaled, targets, test\_size = 0.2, random\_state=42) | Create test/train |

* + Create the regression

|  |
| --- |
| reg = LinearRegression()  reg.fit(x\_train, y\_train) |

|  |
| --- |
| y\_hat = reg.predict(x\_train) |
| plt.scatter(y\_train, y\_hat)  plt.show() |

* + Check the residual plot to check for the differences between the targets and the predictions

|  |  |
| --- | --- |
| sns.distplot(y\_train - y\_hat) |  |

* + Should be normally distributed around 0
  + If there is a longer tail on either end could indicate that the model is overestimating (if negative tail) and underestimating (if positive tail)

|  |
| --- |
| reg.score(x\_train, y\_train) |
| reg.intercept\_ |
| reg.coef\_ |
| reg\_summary = pd.DataFrame(inputs.columns.values, columns = ["Features"])  reg\_summary ["Weights"] = reg\_coef\_  reg\_summary |

* + Continuous var: A positive weight shows that as a feature increases in value, so does the log\_depenedent and dependent variable, negative shows a negative prediction
  + Dummy var: a positive weight shows that the respective category is more expensive than the benchmark (the one that was removed in n-1)
* Testing

|  |  |
| --- | --- |
| y\_hat\_test = reg.predict(x\_test) |  |
| plt.scatter(y\_test, y\_hat\_test, alpha = 0.2) | Check that resemble the testing, alpha shows a heatmap |
| df\_pf = pd.DataFrame(y\_hat\_test, column = ["Prediction"]) | Will output the predictions for the log dependent variable |
| df\_pf = pd.DataFrame(np.exp(y\_hat\_test), column = ["Prediction"]) | Take exponential of the log dependent to reach original |
| df\_pf ["Target"] = np.exp(y\_test) | Place the targets in the same data frame to compare |
| y\_test = y\_test.reset\_index(drop=True) | Too many null values because basing it on index, need to reset the index and run target again |
| df\_pf["Residual"] = df\_pf["Target"] - df\_pf["Prediction"] | Compare the predictions with the targets |
| df\_pf["Difference%"] = np.absolute(df\_pf["Residual"]/df\_pf["Target"]\*100) | To compare with percentages |
| df\_pf.describe() | Print descriptive stats: look at min/max difference and look at percentiles to see how close the predictions were |

**Logistic regression**

* Real life problems often require more sophisticated non-linear models (could be quadratic, exponential, logistic)
* Logistic regression implies that the possible outcomes are not numerical but categorical
* In a business context, decision making often boils down to yes/no
* Logistic regression assumptions are very similar to linear regressions, except it does not need a linearity assumption
* Logistic regression predicts the probability of an event occurring
* Logistic model is represented by the logistic function (bounded by 0 and 1)
* Machine generated alternative text:
  Logistic regression model: 

* Machine generated alternative text:
  Logli regression model: 
* Can use either the logistic or logit regression, but logit is much simpler to use because the log of it represents the linear regression, so this will be used
* In Python
  + Import libraries, load data, view data
  + Convert yes/no entries to 0 and 1 in the same way as did for dummies
  + Define variables
  + Create scatterplot (will be different from a linear regression, and if try to fit a linear regression line on it, it will be wrong and values will fall outside the possible 0 and 1)
  + A logistic regression curve will show the certain score, and anything in the middle will show the probability (on the y axis)
  + Add a constant

|  |
| --- |
| reg\_log = sm.logit(y,x) |
| results\_log = reg\_log.fit() |

* + The output is a message: successful and the function value (the objective function at the nth iteration), iterations will be specified and the model will not run after 35 iterations
  + Get summary, which is very similar to the linear regression summary
    - Method: MLE (maximum likelihood estimation), estimates how likely it is that the model at hand describes the real underlying relationship if variables, tries to maximise the likelihood, when it can no longer improve it it will stop the optimisation
    - Log-likelihood: almost always negative and the bigger it is the better
    - LL-null: log likelihood null, log likelihood of a model which has no independent variables
    - Compare log likelihood with LL null to see if model is significant with the LLR p value
    - Pseudo R squared: McFadden's R squared, a good value is somewhere between 0.2-0.4, useful for comparing variations of the same model not different models
    - Machine generated alternative text:
      Aodds 
    - log(odds) = const + coef\_var1\*var1
  + Binary predictors can be used for logistic regressions
    - Machine generated alternative text:
      Iterations 5 
      Results 
      ) = 2.08 
      out(71: 
      Log it Reg 
      Dep. Variabh: 
      Method: 
      Date; 
      converged: 
      Con St -C 6436 
      2.0786 
      Admnted 
      Logit 
      2018 
      No. Observations: 
      ot R„iduals: 
      Pseudo R.squ.: 
      Log-L ike li : 
      LL*ull: 
      0.1659 
      -96140 
      .11528 
      LLR 6.283+10 
      log (odds) 
      = -0.64 + 2.08 * Gender 
      log (odds) 
      = -0.64+2.08 * Genderz 
      log (oddst) 
      = -0.64+2.08 * Genderl 
      odds 
      log 
      = 2.08 * (Genderz — Genderl) 
      odds 1 
      std err 
      0.222 
      0363 
      -2901 
      5727 
      p>lzl 
      0000 
      [0.075 
      -1.0/8 
      1.367 
      0.9751 
      -ozog 
      2790 
      np . exp(2_ 0786) 
      7 .993270498536412 
      10 g (2-g.iL.zua.L2) 
      odds 
      odds 
      10 g (—LK.zz.u.ue 
      Odds 
      odds 
      female 
      = 2.08* (1 — 0) 
      = 7.99 * oddsmale 
  + Calculating the accuracy of the model
    - results\_log.predict() we will get all the predicted values to format to non-scientific notation:

|  |
| --- |
| np.set\_printoptions (formatter = {"float": lamda x:"{0:0.2f}".format(x)})  resutls\_log.predict() |

* + - These will be the probabilities (0, 1, and number in between), values below 0.5 will be rounded down and values above 0.5 will be rounded up
    - Compare the values predicted by the model to the actual values, if 80% of values coincide we say the model has 80% accuracy

|  |
| --- |
| results\_log.pred\_table() |

* + - Result in a confusion matrix (how confused the model is), represents the type I type II error table
    - Machine generated alternative text:
      For 69 observations the model predicted O and the true value was O 
      For 90 observations the model predicted 1 and the true value was 1 
      For 4 observations the model predicted O while the true value was 1 
      For 5 observations the model predicted 1 and the true value was O 
      In 159 cases 
      the model did 
      its job well 
      In 9 cases 
      the model 
      'got confused' 
      c m_df 
      Predicted o Predicted 1 
      Actual O 
      Actual 1 
      CONFUSION MATRIX 
    - Calculate the percentage accuracy

|  |
| --- |
| cm = np.array(cm\_df) |
| accuracy\_train = (cm[0,0]+cm[1,1])/cm.sum()  accuracy\_train |

* + Testing the model
    - Use model to make predictions based on test data and compare those to the actual outcome and calculate the accuracy

**Cluster analysis**

* Cluster analysis is a multivariate statistical technique that groups observations on the basis of some of their features or variables they are described by
* Observations in a dataset can sometimes be divided into different groups and it can be useful
* Clustering of countries can be done by geographical proximity, or language and it will provide different clusters
* The goal of clustering is to maximise the similarity of observations within a cluster and maximise the dissimilarity between clusters
* Example applications of clustering: market segmentation and image segmentation
  + Useful to determine marketing opportunities by defining groups/clusters and seeing who marketing strategies should be aimed at
  + Useful in image segmentation: creating clusters of colours on an image which helps machine cluster objects and perform image recognition
* Clustering analysis usually precedes other types of analyses (good starting point)
* Cluster analysis is unsupervised learning: cluster the data in different groups but have no labels (must name the output ourselves)
* How does it work?
  + Measure distance between two points (euclidean distance) by creating a right triangle
  + Machine generated alternative text:
    Euclidean distance 
    21) space: 
    3D space: 
    If the coordinates of A are and of B are 
    space: 
  + Centroid: the midpoint of a line which connects two points (for more than 1, it is in the middle of all of them)
  + K means clustering is one of the most popular methods
    - Choose how many clusters you want to have (K stands for the number of clusters)
    - Specify the cluster seeds (either picked at random or specified by data scientist based on previous knowledge about the data)
    - Assign each point on the graph to a seed based on proximity
    - Adjust the centroid of the different clusters to fit the identified clusters better
    - Repeat the last two steps until all the points are centralised to their centroids
  + Kmeans minimizes the distance between points in a cluster and maximises the distance between clusters
    - Distance between points in a cluster = within-cluster sum of squares of WCSS
    - WCSS is a measure within the ANOVA framework (like SST, SSR, SSE)
    - Minimising WCSS means reaching the perfect clustering solution
    - If each point has a cluster it's useless, and if all points belong to one cluster is also useless
    - Want WCSS to be as low as possible but still with a good number of clusters that can be interpreted
    - Elbow method: after n clusters, the solution is not much better than n
    - Need to know the number of clusters and the WCSS
    - WCSS: kmeans.inertia\_ and calculate WCSS for each cluster
* K means clustering in Python
  + Import the relevant libraries

|  |
| --- |
| import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  import seaborn as sns  sns.set()  from sklearn.cluster import KMeans |

* + Load the data
  + Country data can be represented in clusters by their longitude and latitude values
  + DataFrame.iloc(row.indices, column.indices) will slice the data frame so include only the columns that you need to define x
    - x = data.iloc[:,1:3] will slide the second and third columns and leave all the rows
  + Kmeans:

|  |  |
| --- | --- |
| kmeans = KMeans(2) | Value in brackets is how many clusters you want |

* + - kmeans will now be used for the clustering itself:

|  |
| --- |
| kmeans.fit(x) |

* + - Also interested in the clusters themselves

|  |  |
| --- | --- |
| identified\_clusters = kmeans.fit\_predict(x) |  |
| Identified\_clusters | Will return an array of the clusters |
| data\_with\_clusters = data.copy()  data\_with\_clusters ["Cluster"] = identified\_clusters  data\_with\_clusters | Create a data frame to represent the data |

* + - To graph, make the same plt.scatter but add an argument to differentiate base on categories of the clusters
    - c=data\_with\_clusters["Cluster"], cmap = "rainbow"
  + Categorical data
    - See categorical data lecture notes
* Pros and cons of K means clustering
  + Pros: simple to understand and fast to cluster, widely available, easy to implement
  + Cons:
    - we need to pick k --> remedy: elbow method but not perfect)
    - Sensitive to initialisation (your initial centroids matter a lot) --> remedy: kmeans ++ which is a preliminary algorithm that runs to determine the best initial position for the seeds (sklearn deploys this by default)
    - Sensitive to outliers --> remedy: get rid of outliers prior to clustering (or remove one point clusters that appear)
    - Produces spherical solutions on a 2D plane
  + Standardisation
    - Good to standardise between clustering to avoid extremes on scale differences
    - When not to standardise: if we know that one variable is inherently more important than the other
    - When standardising, good to keep the axis of the original values and not the standardised because otherwise it is meaningless
* Market segmentation
  + Full example in Jupyter under "Market segmentation example"
* How is clustering actually useful?
  + Exploratory type of analysis to explore the different clusters
  + Can also be used a confirmatory/explanatory: maybe we already knew the different clusters and just wanted to assign observations to different clusters but is uncommon
  + Can be used for optimisation of marketing, tracking the consumers who are "fans" and targeting others with similar age/interests/etc

**Other types of clustering**

* Two broad types of clustering: flat and hierarchical
* K means is a flat type, but hierarchical was developed first
* An example of a hierarchy of clusters is the animal kingdom: animals are split into mammals, fish, birds, and further split until you reach species and sub species of different animals
* Two types: agglomerative (bottom up) and divisive (top down)
* Agglomerative (easier to solve mathematically)
  + Start with each case being its own cluster (n)
  + Using euclidian distance, group the two closest clusters together (n-1)
  + Repeat this until all observations are in a single cluster
  + Produces a dendrogram
* Dendrogram
  + The bigger the distance between two links, the bigger the difference in terms of features
  + Machine generated alternative text:
    DENDROCRÆM 
    Europe + NA + Australia 
    North America 
    C+USA 
    C+F t 
    e rmany 
    Europe 
    Europe + NA 
    $ C+F+UK 
  + Distance between the links shows similarity between observations
  + Draw a horizontal line and how many links are broken is how many clusters you now have
  + Where to draw the line?: no specific rule, develop an intuition, if the distance is too big then it's probably good to draw a line around there
  + Pros: shows all the possible linkages between clusters, understand the data better, don’t need to preset the number of clusters, many methods to perform hierarchical clustering (Ward)
  + Cons: scalability (1000 observations lose the meaning) and computationally expensive with more observations compared to k means