* Big Data Scenarios are usually classified by:
  + Volume: Terabytes of data
  + Velocity: Continuous stream of real-time data
  + Variety
  + Veracity: Unreliable, noisy, biased or incorrect data
* Data Science: Involves principles, processes and methods for identifying and understanding phenomena via automated/semi-automated analysis of data
* Data Mining: Extracting knowledge from data via algorithms which incorporate these principles
* Data Science workflow:  
  1. Data Collection: Retrieve/import Raw Data from sources  
  2. Data Preparation: Clean Manipulate and aggregate data into analyzable form  
  3. Analysis and Modelling  
  4. Report & Visualization
* Knowledge Discovery in Databases (KDD): Detailed timeline for data science projects
  + Exploration🡪Preprocessing🡪transformation🡪Data Mining🡪Interpretation and Evaluation
  + Human Interaction is important
* Analytics Base Table (ABT): Structure we use to represent a new dataset
  + Row = Case/Example which are made up of
    - Descriptive features: Characteristics and properties of cases
    - Target Feature: Feature we aim to predict/analyse
  + Features depend on domain
  + Descriptive features have types
* Important considerations to think about with features:
  + Data availability
  + Timing
  + Longevity
* Cost-sensitive features: Think about how financial cost of getting features and new data
* Privacy-sensitive features: Some might be good in an algorithm but very bad to include from legal/ethical standpoint
* When working on a new dataset, we characterize the data to understand nature of data in ways like:
  + Understand meaning of each feature in data
  + Identify type of value taken by each feature
  + Identify any data quality issues
* CSV are comma separated values
  + File format used in tabular data
  + Like plaintext but values are split by commas or a different separator
  + Has a header line with column names
  + Python has a built in module to assist with reading and writing
    - import csv  
      fin = open("countries10.csv", "r")  
      reader = csv.DictReader(fin)  
      for row in reader:  
      print(row)  
      fin.close()
    - fout = open("output.csv", "w")  
      fields = ["code", "name", "population", "life\_expect"]  
      writer = csv.DictWriter(fout, fieldnames=fields)  
      writer.writeheader()  
      for row in data:  
      writer.writerow(row)  
      fout.close()
* JSON (JavaScript Object Notation)
  + Lightweight format
  + Based on Javascript
  + Built from 2 structures:
    - Object which is a collection of name/value pairs.
      * Begins and ends with {}
      * Each name is followed by :
      * Name/Val pairs separated by commas
    - Arrays:
      * Separated by commas and like python begins and ends with []
  + Can mix both
  + Outmost value can be either
  + Whitespace isn’t needed
  + Language agnostic
  + json.dumps(): python struct🡪JSON string
  + json.loads(): JSON strong🡪Python struct
* XML
  + Markup language
  + Describes data using tags which are user define
  + Tags opened and closed using <tag> and </tag> respectively
  + Can have 0+ name/val attribute paors
  + <collection>  
    <note>  
    <to>Alice</to>  
    <from>John</from>  
    <subject>Reminder</subject>  
    <body>Remember to buy milk!</body>  
    </note>  
    <note>  
    <to>John</to>  
    <from>Alice</from>  
    <subject>Shopping</subject>  
    <body>I forgot the milk!</body>  
    </note>  
    </collection>
  + Hierarchal data format
  + Use trees with nodes to represent
    - Document = Root node
    - Lowest level has leaf node
  + Parse the xml amd finds tags interested in
    - import xml.etree.ElementTree  
      tree = xml.etree.ElementTree.parse("products.xml")

for book in tree.iterfind("book"):  
n = book.findtext("name")  
a = book.findtext("author")  
print("%s by %s" % (n,a) )

* HTML
  + Like XML but is also describes presentation
  + Usually badly-written and invalid
* Web scraping: Extracting data from website using tolls which act like a browser
  + Follows the steps identification🡪collection🡪extraction🡪cleaning
  + Check a sites terms if it allows it
  + Don’t hammer site with many automated requests
  + Re-write often due to layout changes in site
  + Collection:
    - urllib.request.urlopen(link) gets the html code from website after which we do .read().decode() on it
  + Extraction:
    - BeautifulSoup allows one to parse HTMLS even bad ones
    - Example:  
      import bs4  
      parser = bs4.BeautifulSoup(html,"html.parser")  
      for match in parser.find\_all("h3"):  
      text = match.get\_text()  
      print(text)
* API (Application programming interface)
  + Web service which contain a set of HTTP request messages with a definition of expected structure of response
  + Accept a query/call for particular data
  + Able to respond dynamically
  + 1+ endpoint which are URLs which can be used to retrieve data
* Data Serialisation
  + Flattening data structures into a format which can be stored/transferred or shared with another program
  + Use eother JSON or Pickle
  + Reverse process is deserialization
  + json.dump and json.load serialises/deserialises value into a JSON value
  + JSON can be slow and doesn’t necessarily support all Python data types but they are language agnostic and always has a text-based output
  + Pickle turns python data into bytestream representation which is easily stores and then can later reconstruct the data
  + Can serialize:
    - Python native types
    - Lists,tuples, dictionaries
    - Functions
    - Objects
  + Pickle.dump() serialises pythn object into an open file and pickle.load does the opposite
  + Note if using pickle, use a binary file
  + import pickle  
    ages = {"Steve":25,"Linda":40,"John":33}  
    fout = open("example.pkl", "wb")  
    pickle.dump(ages, fout)  
    fout.close()
  + fin = open("example.pkl", "rb")  
    x = pickle.load(fin)  
    fin.close()
* Pandas is an open source package which allows one to analyse data and use data structures
* Pandas offers two data structures: Dataframes and Series
* DataFrame: flexible 2D labeled data structure
  + Fastest way to create a DataFrame is to pass a dictionary of lists
  + d = {"Country":countries, "Region":regions, "Population":pops, "Life Exp":life\_exp}  
    df = pd.DataFrame(d)
  + df.shape gives us how many rows, columns a df has
  + pd.read\_csv(CSV\_Name) reads in a CSV file file as df, assumes values are comma separated
  + df.columns gives us all the columns while df.index gives us the indexes
  + df.describe() here describes all numerical columns
  + To access a column, we use the index name df[Index\_Name]
  + To access row:
    - df.iloc[x] where x is a numerical position
    - df.loc[Index\_name]
    - Can use splicing or a lost of index names to access multiple rows
  + To access individual values:
    - df.iloc[x][y]
    - df.loc[Index\_name] [“Column Name”]
  + To filter: df[df[“Column Name”] bool expression] . can use & or | here
  + To check which index values satisfy bool: df[Column Name] Bool Expression
  + To sort:
    - df.sort\_values(by=”column”) or can use multiple columns
* Series is a data structure for a single column
  + Think of it as a 1D array holding any data type, but all values must be of the same type and the elements can have a custom index
  + pd.series(y,labels) converts Y (usually an array) into a series
  + Axis labels = index. Note index and position aren’t necessarily the same
  + Can create a pd series from a dictionary
  + .describe() allows us to get a statistical summary of a Series
  + To access/modify a series, one can either use position or the index
  + We can filter/ check values in a series
    - Filter we do: Series[bool expression]
    - Check: Series bool expression
  + Can use sort\_values() or sort\_index() to sort the series
* Cleaning a dataframe:
  + Df[“column’].replace() erplaces values with another
  + All numerical values must share a common scale (Mean, Max, Z Score normalisation, Min-Max)
    - For min max: get the min and max vaues then do :   
      df["column’"] = (df["column’"]-lmin)/(max-min)
  + To Bin Data: pd.cut(df[“column”], bins = The bins you wish to use)
  + To deal with missing values, we can either filer missing data by dropping values or we can fill it in with either mean/median, a constant or use prediction
  + .drop() will remove one or two rows/columns depending on axis
  + .dropna() drops missing values
  + .fillna() fills all missing values with specified value
  + .drop\_duplicates() drops all duplicate rows
* Sampling
  + Work with a smaller subset of he data
  + Must still be representative of OG dataset while having no unintended biases
  + Random Sampling: Choosing a fraction % of cases from dataset
    - Use df.sample() with parameter either being number of samples of the fraction of data
* Exporting df:
  + to.csv() changes it to a CSV
  + to\_pickle() serialises it
* df.value\_counts() returns a series with counts of unique values, can normalize it
* Cross tabulation: allows us to quantitively analyse relationship between multiple variables. Done via pd.crosstab(df[“col1”],df[col2])
* Data aggregation: Combines multiple cases/values into a single value. Done by df.groupby(“Label”)
* Figures can contain one or more plots
* General flow of data plotting is:
  + Plt.figure  
    plt.plot(data)  
    plt.xlabel()  
    plt.ylabel()  
    plt.title()  
    plt.show()
* Plt.scatter() plots a scatter graph. Takes in size, shape and colour of points as params
* Plt.pie() plots a pie chart. Takes in datam label, colours and degrees of freedom as params
* Plt.bar()/barh() is for bar graphs
* Plt.pcolour(a) creates a colour plot…include plt.colorbar()
* Df[“column”].plot(kind=”plot\_type”) also plots data then use matplotlib to edit
* In a panda series, we can use .hist() to create a histogram
* Numpy is the main tool for scientific computing imported as np)
* Numpy arrays only can have one type of value
  + Numpy array created like this a=np.array([[a,b,c],[d,e,f]))
  + Can use numerical operators on it
  + Can access the same way as regular python arrays
  + Slicing doesn’t copy the array but provides a view
  + Rank = No of dimensions
  + Shape gives a tuple of length in each dimension
  + Size is the total number of entries in array
  + Np.zeroes(n) or np.zeroes((n,m)) gives us an arra of that size propagated by zeroes
  + Np.arange(min.max,step) generates an array of range min to max using the indicated step size
  + X.reshape(row,column) reshapes a given np array into that shape
  + Used to avoid looping through each value individually and just using vector operations
  + Np.savetext(file,data,delimiter) saves an np array into the text file
  + Np.loadtext(text\_file) loads the text file as an np array
  + Can be comnined with plt
* Predictive modelling (Supervised machine learning): Using statistics to predict outcomes based on historic data
* Classification: Learn from labelled training set to make a prediction to assign test cases to a class
* Regression: Learning from a labelled training set to decide value of a output variable
* Calling np.corrcoef(x,y) calculates the correlation coefficient matrix of x and y
* Df.corr() also accomplishes this
* Causation != Correlation
* Regression analysis: Statistical Process for estimating relationship between variables and make predictions using it
* Simple Linear Regression: X and Y are linearly related (y=mx+c)
  + Y is respence variable
  + X is input feature
  + M and c are coefficients
  + To do this in python, from sklearn.linear\_model import linearRegression

Model=LinearRegression()  
model.fit(x,y)

* + Model.intercept\_ gives us c while model.coef\_[0] gives us m
* Multiple Linear Regression:
  + Like linear regression but each feature has its own coefficient
* Drop outliers in regression as if included leads to poor regression fit
  + Use box plots for this
* Two types of classification:
  + Binary: Assign an input into one of two classes (one is the state of interest and the other is the normal state, commonly referred to as positive and negative classes respectively)
  + Multiclass: Assign input into one of 3+ classes
* Training Set: Dataset of classes for relevant classes
* Target feature: What we aim to predict correctly
* Data normalisation: Changing values of all numeric features in data to a common scale
* One hot encoding: Converts each categorical feature in data into dummy variables in which each category taken by the original feature is transformed into a new binary feature
* Min-max normalisation:  
  from sklearn.preprocessing import MinMaxScaler  
  scaler = MinMaxScaler()  
  X\_norm = scaler.fit\_transform(X)
* For Z-score normalisation:  
  from sklearn.preprocessing import StandardScaler  
  scaler = StandardScaler()  
  X\_norm = scaler.fit\_transform(X)
* To convert a df into one hot encoding🡪df\_new=df.get\_dummies()
* Classification algorithms:
  + K nearest neighbors
    - Find most similar previous example/nearest neighbour from training set
    - Majority voting: Predicted label for a new input example Z is decided based on the votes of its k nearest neighbors where the neighbours are selected to minimise distance
    - Python syntax:  
      from sklearn.neighbors import KNeighborsClassifier  
      model = KNeighborsClassifier(n\_neighbors=3)  
      model.fit(data, target)  
      model.predict(test\_data)
  + Decision tree
    - Use decision rules to determine output
    - Idea is to build a tree which will split the training set into subsets using rules from training set
    - Use as few trees as possible
    - Syntax:  
      from sklearn.tree import DecisionTreeClassifier  
      model = DecisionTreeClassifier()  
      model.fit(data, target)  
      model.predict(test\_data)
  + Neural network
  + Support vector machine
* Feature space: a n-dimensional co-ordinate space used to represent input examples for give problem with one co-ordinate per descriptive feature
* Similarity measure: Some function to measure how similar 2 input examples are from another are in the n-dimensional co-ordinate space
* Euclidean Distance: Square root of the sum of squared difference between each feature f
* To split data into test/training:   
  from sklearn.model\_selection import train\_test\_split  
  train\_data, test\_data, train\_target, test\_target = train\_test\_split(data, target, test\_size=% split)
* We split data into test and training to prevent overfitting, in which our model is fitted too close to training data and its noise
* Accuracy: # correct predictions/total predictions  
  from sklearn.metrics import accuracy\_score  
  accuracy\_score(actual, predicted)
* Confusion matrix:  
  A screenshot of a computer screen

  Description automatically generated
* To program it:   
  from sklearn.metrics import confusion\_matrix  
  cm = confusion\_matrix(target\_test, predicted, labels=[1,-1])  
  print(cm)
* K-fold cross validation:
  + Divide data into k disjoint subsets
  + For each k subset, use selected as test data and others as training and repeat for all k folds
  + Get average accuracy/error
  + For python:  
    from sklearn.model\_selection import cross\_val\_score  
    scores.mean() = cross\_val\_score(model, data, target, cv=5,  
    scoring="accuracy")
* Text mining tasks:
  + Text classification
  + Topic modelling
  + Sentiment analysis
  + Review mining
  + Authorship attribution
  + Genre classification
  + Moderation
  + Plagiarism detection
* When analysing texts:
  + Split raw text into individual tokens corresponding to a single term  
    from sklearn.feature\_extraction.text import CountVectorizer  
    tokenize = CountVectorizer().build\_tokenizer()
  + Then count the frequency of term occurrence
* Bag-of-words model: Each document is represented by a vector in a m-dimensional coordinate space where m is total number of unique terms across all documents
  + Each document is represented as a term vector with each entry indicating number of times a term appears in a document.
  + Then we can stack them by row to create a document-term matrix.
  + However, we lose position and context of terms within OG document
  + Leads to a high dimensional matrix
  + In python:  
    from sklearn.feature\_extraction.text import CountVectorizer  
    vectorizer = CountVectorizer()  
    X = vectorizer.fit\_transform(documents)
* N-Grams:
  + Used to maintain sequence information
  + We can build sequences of adjacent tokens
  + Term n-grams: Build terms for n adjacent tokens
  + Significantly increases vocabulary size
  + Python:  
    from sklearn.feature\_extraction.text import CountVectorizer  
    vectorizer = CountVectorizer(ngram\_range = (1,2))  
    X = vectorizer.fit\_transform(documents)
* Cosine similarity: Look at cosine angle between term vectors
  + A math equations and formulas

    Description automatically generated
  + If score is 1, they are identical and if 0, they share no terms in common
  + Python implementation:  
    from sklearn.metrics.pairwise import cosine\_similarity  
    print( "cos(D1,D2) = %.2f" % cosine\_similarity( X[0], X[1] ) )
* Issues in text mining:
  + Sparsity problem: Most documens share very few words. N-grams also increase this problem
  + Synonymy: Different words which relate to identical or closely related concept
  + Homonymy: A single word cam have multiple unrelated meaning
  + Polysemy: A single word can have multiple related meanings
* Preprocessing for texts:
  + Case conversion
  + Minimum term length
  + Stemming: remove endings of words
  + Lemmatisation: Reduce term to canonical form
  + Stop-word filtering: Remove terms which appear on a predefined filter list of terms that don’t convey information yet are very frequent like and, the while etc.
  + Low frequency term filtering
* Scikit’s countvectoriser has parameters to help aid us:
  + stop\_words specifies the words which will be filtered
  + min\_df filters terms which appear in less than n documents
* For stemming/lemmatise, we use NLTK library:
  + Individual word stemming:  
    from nltk.stem.porter import PorterStemmer  
    stemmer = PorterStemmer()  
    for w in words:  
     print( stemmer.stem(w) )
  + For lemmatisation:   
    from nltk.stem import WordNetLemmatizer  
    lemmatizer = WordNetLemmatizer()  
    for w in words:  
    print( lemmatizer.lemmatize(w) )
* We can add weights to frequencies via TF-IDF:
  + TF = term frequency
  + IDF is inverse document frequency: this is the total number of distinct documents containing a term with the goal of penalising common terms appearing in almost every document
  + Formula: A math equations with green lines

    Description automatically generated with medium confidence
  + Python:   
    from sklearn.feature\_extraction.text import TfidfVectorizer  
    vectorizer = TfidfVectorizer()  
    X = vectorizer.fit\_transform(documents)
* For text classification
  + Most fundamental task is sentiment polarity classification(classifiying id a document is positive or negative)
* Pipelines allow us to bundle preprocessing transformations and classifiers together
  + Example:  
    pipeline1 = Pipeline([  
    ('vec', CountVectorizer(stop\_words="english")),  
    ('tfidf', TfidfTransformer()),  
    ('clf', KNeighborsClassifier(n\_neighbors=3))  
    ])  
    pipeline1.fit(train\_documents, train\_target)  
    predicted = pipeline1.predict(test\_documents)
* Time series: A dataset containing values of a function which is sampled across different points in time
  + Can be used to track a single or multiple variables over time to see any season changes, trends, outliers, step changes and behaviours
  + Stationary: If a time series’ properties don’t depend on time which series is observed
  + Non-stationary: When their statistical properties change over time
* Important things to note about time series:
  + Sampled at equally spaced intervals
  + How long and or rapidly growing is the series
  + Best resolution/frequency of data
  + Any missing, noisy, or outliers
* Components of a time series:
  + Trends
  + Seasonality: Peaks and troughs which occur in a regular interval
  + Noise: Fluctuations in data which are left when all components are removed
* Pythons datetime module allows to modify and deal with dates and time  
  from datetime import datetime  
  d = datetime.now()  
  print(d.year)  
  print(d.month)  
  print(d.day)  
  print(d.hour)  
  print(d.minute)  
  print(d.second)  
  print(d.microsecond)
* Can use d.date() to get the date component and d.time() to get time component
* Timedelta: Datatype in python which stores temporal difference between values
  + Updates times using days and seconds i.e. timedelta(d,s) + datetime
* Several ways to write a datetime as a string:
  + d.strftime('%d/%m/%y') 🡪 01/03/20  
    d.strftime('%Y-%m-%d')🡪 2020-03-01  
    d.strftime('%H:%M:%S')🡪 09:30:00  
    d.strftime('%Y-%m-%d %H:%M')🡪 2020-03-01 09:30  
    d.strftime('%d %B %Y')🡪 01 March 2020  
    d.strftime('%a %d %Y'🡪 Thu 01 2020
* Code for strftime():  
  A white rectangular box with black text

  Description automatically generated
* The reverse (Str🡪datetime) can be done using strptime()
  + Example is s= some date  
    datetime.strptime(s, Date\_format)
* Most time series data are a pd series indexed by timestamps. Below is an example of how they work:  
  dates = [datetime(2016,4,2), datetime(2016,4,4), datetime(2016,4,6), datetime(2016,4,8), datetime(2016,4,10)]  
  sales = [10,11,12,16,11]   
  ts = pd.Series(sales, index=dates)  
  ax = ts.plot(figsize=(9,6),fontsize=14)  
  ax.set\_xlabel("Date",fontsize=14)  
  ax.set\_ylabel("Count",fontsize=14
* A pd time series is indexed the same way as a regular series
* To read in a pd time series from a csv:  
  df = pd.read\_csv("csv name",index\_col="date",parse\_dates=True)
* We can aggregate dat and time using df.groupby(df.index.Specifed\_group)
* Can use a custom aggregate function to group as follows:  
  def to\_decade(date\_value):  
  return (date\_value.year // 10) \* 10  
  df\_decade = df.groupby(to\_decade).sum()  
  df\_decade.head()
* Sampling frequency: How often an observation of time series occurs
* Resampling: Process of converting time series data from one frequency to another
* Downsample: Aggregate data to lower frequency
* Upsample: Convert lower to higher frequency
* Up and downsample are done via .resample(datetime code)
  + Best used to get end on period of time frequencies
* Moving average/rolling mean: Average across time series
  + Divides series into overlapping regions of fixed size(windows) then the average of all observations in current window is calculated, then moving to next window
  + Done by the rolling(no of windows).mean() method
* To deal with missing data in time series:
  + Drop record
  + Estimate missing information via :
    - Last observation carried forward (LOCF): Use the most recent previous non-missing value to estimate the current missing value.
    - Next observation carried backward (NOCB): Use the next non-missing value to estimate the current missing value.
    - Linear interpolation: Fill in the missing data by assuming that the relationship between the variables follows a line.
* We use forecasting to predict future values in a time series
  + Input is past data
  + Output is future time beyond input data