# Definitions

* Supervised Learning: Using data labels, we predict outputs using inputted data and labels
* Classification: Decide target class using set of features
  + Binary Classification: Only two target classes
  + Multiclass classification: More than two target classes
  + Eager Learning Classification strategy: Build a model during training phase. Less work at run-time but more offline setup work
  + Lazy Learning Classification Strategy: The classifier keeps all the training data to use later. Less offline work but waits for new query examples
* Regression: Using set of features, determine value of a continuous output variable
* Distance-based models: Via exploitation of distance between two points (generalised training and unseen data), create a model to classify points
* Unsupervised Learning: Unlabelled data inputs are mapped to outputs using data exploration and knowledge discovery algorithms
* Target class label: Value in which our model will output. i.e. what we aim to find
* Feature space: D-dimensional coordinate space that represents inputs with each co-ordinate representing a feature
* Distance function: Measures how close two points in a feature space are
  + Local distance function: Based off only a single feature
  + Global distance function: Based off all features in a feature space
  + Overlap function: Function which returns 0 if two feature values are equals, else 1 (Categorical)
  + Hamming Distance: Sum of overlap function across all features
  + Absolute Difference: Distance between two points then absolute it. For ordinal values, use positions in list as distance
  + Euclidean distance: Square root of sum of squared differences across all featuresA math equation with square root and square root

    AI-generated content may be incorrect.
  + If using Ordinal values: assign each rank a number and go from there or normalise it.
* Normalisation: The process of making data in a similar range
  + Min-max Normalisation: Rescale values to be in range [0.1] by doing

# KNN

* Nearest neighbour rule (1NN): find a single labelled example close to the input and assign it the same label as x
* K-Nearest Neighbours: Using k-nearest neighbours from training set, assign a label to an input
* Majority voting: Decide on a label using votes of its k nearest neighbours, the majority it the new label
* Here we mainly use distance to determine what is the nearest neighbour and then assign the label
* We can add weights to some of the training data to influence the votes
* Inverse distance weighting: Sum of the inverse distance between point and neighbour
* We use odd k values to prevent ties however if a class has no clear majority or if the value is equally close to all odd neighbours, it becomes hard.
* 1NN susceptible to noise
* Hyperparameter tuning is adjusting settings of an algorithm to better suit data
* In this case, setting k too low leads to easier misclassification but if k is too high, classes become unbalanced
* Process in python:
  + Load csv
  + Set the features and targets (can normalise or add weights)
  + Set up classifier and then train it
  + Make predictions

# Decision Trees

* Idea of a decision tree is to split data into subsets using rules on features which are inferred from training set
* Eager Learning Strategy
* Python implementation:
  + from sklearn.tree import DecisionTreeClassifier,  
    df = pd.read\_csv(df.csv')  
    y = df.pop(target\_feature).values  
    df.pop(Categorical Features)   
    df\_features = df.columns  
    X = df.values  
    tree = DecisionTreeClassifier(criterion='entropy') #Different criterion exist  
    df\_tree = tree.fit(X, y)
* How to create a decision tree:
  + Place all training set into a root node
  + Use one of the features to split data into child nodes, ideally with as many child nodes which don’t split up further
  + Repeat the above but using a different feature on split data and is done until all the initial dataset is split into subsets in which all datapoints are of the same class (i.e. can’t be split further)
* Pure Node: All nodes have same class label (aim is to get this)
* A decision tree in which all the leaf nodes are pure can always be constructed provided there are no clashes in the data
* Aim for simpler trees rather than complex ones and aim for fewer assumptions
  + Good trees classify data using as few tree nodes as possible (minimise depth)
  + Aim for high purity when splitting (go for splitting that leads to highest purity)
  + Use a good feature to divide examples into single category classes
* Entropy: Measure of uncertainty around information source. Low for predictable, high for random
  + Low entropy= high purity
  + High entropy= high impurity
  + Formula: where pj is probability of Class Cj
    - Example: A math equations on a white background

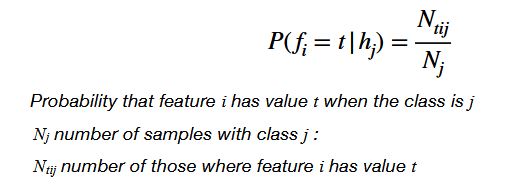
      AI-generated content may be incorrect.
* ID3:
  + Algorithm which builds a tree top down
  + Steps:
    - Start with an empty tree and a set of all training sets
    - If all examples in S are of the same class, then return a new leaf node and give it a label.
    - Else: Select a feature then generate leaf nodes where the selected feature is test feature. Repeat the if else conditions with create subtree
* Information Gain(IG): An approach to select features in decision trees using entropy.
  + Formula: original entropy – entropy after split (note entropy after split will be weighted in proportion to size) 🡪IG(S,A) = where S is original set and Si is subset
  + Higher IG, the better the selected feature is
* Once tree is constructed, we can use it classify objects
* If we have two identical data (i.e. same features) but different target labels, we can either take the majority vote at a leaf node or randomly choose 1 if a tie
* Overfitting can occur and to avoid this we reduce the models capacity through various means such as setting max depth
* Note, we must convert any categorical variables into numbers, one hot encoding helps with this or we can use get\_dummies from pandas
  + One hot encoding:   
    from sklearn.preprocessing import OneHotEncoder  
    onehot\_encoder = OneHotEncoder(sparse=False)  
    dfOH = onehot\_encoder.fit\_transform(df)
  + Dummies:  
    df = pd.get\_dummies(df,drop\_first=True)

# Naïve Bayes

* Eager learner
* A mathematical equation with black text

  AI-generated content may be incorrect.
  + P(A|B) is conditional probability of A given B
  + P(B|A) is likelihood
  + P(A|B) is known as posterior probability
  + P(A) is usually the probability of hypothesis but its also the probability of encountering a class
* The below table tells us where to find the value for probability calculations

|  |  |  |  |
| --- | --- | --- | --- |
|  | C | D | Total |
| A | P(A|C) | P(A|D) | P(A) |
| B | P(B|C) | P(B|D) | P(B) |
| Total | P(C) | P(D) | P(C) + P(D) = P(A)+P(D) |

* Idea is to use estimates of likelihoods to determine most likely prediction for classification
* Can be used for classification of texts
* As we aim to find the most likely hypothesis for data, we usually use Maximum Aposteriori Hypnothesis (MAP): hMAP  = arg max h∈H P(h|D)
* As it is tough to estimate the likelihood, we use a naïve assumption in that given all the features, our likelihood is where fi is a given label and hj is given feature then take the hj which has the highest probability and multiply it by the likelihood
* Steps:
  + Calculate class prior probabilities
  + Get conditional probabilities for each feature via:  
    
  + Construct a full contingency table for all features on both classes
  + Test a new example by multiplying class feature probabilities by the target class probability. In this case, look at the test input features and multiply the probabilities which has the same features
  + Normalise data
* If we have numerical values as a feature, we can either convert it to take a fixed value or we can use some distribution if it fits some distribution such as storing mean and standard deviation if it fits some norma distribution
  + Normal PDF: A mathematical equation with numbers and symbols

    AI-generated content may be incorrect.
* IN python:
  + CategoricalNB will work with categorical data once it is processed  
    using an OrdinalEncoder
  + GaussianNB assumes the numeric features have a Gaussian  
    distribution
  + BernoulliNB binary data (use with one hot encoding)
  + MultinomialNB count data such as word counts
  + We use OrdinalEncoder to convert values to numbers

# Evaluation

* Null Hypothesis(H0): Assumption being tests. We either reject or fail to reject it.
* Alternative Hypothesis(Ha): Hypothesis we accept if we have enough proof to reject null hypothesis
* Type I error: Reject H0 when it is true. False Positive
* Type II error: Fail to reject null hypothesis when it should be rejected. False negative
* Aim is minimizing False negatives as much as we can. While we want to reduce false positives, a false negative is more serious (think pandemic)
* Misclassification rate = (# incorrect predictions)/total predictions
* Accuracy = (#correct predictions)/total predictions
* Usually, split data into training and test (split is your decision) and train on the training data then use held out data as test data to test accuracy
* Confusion matrix: Summarise performance of model

|  |  |
| --- | --- |
| True Negative(TN) | False Positive(FP) |
| False Negative(FN) | True Positive(TP) |

* Isnt necessarily a 2x2 table, can be multiclass
* Recall(TP rate)=TP/(TP+FN)
* Precision=TP/(TP+FP)
* Specificity(TN Rate)= TN/(TN+FP)
* False Positive Rate)=FP/(FP+TN)
* Can plot precision and recall on a curve to study output
* Imbalanced data: Refers to a problem in classification where the  
  classes in the data are not represented equally
* To evaluate skewed classes:
  + BAR(Balance Accuracy Rate): Mean of TP Rate and TN rate
  + BER(Balance error rate): Mean of FP rate and FN rate
* F-Measure: Measure which trades off precision against recall for given balance
  + Formula: A close-up of a sign

    AI-generated content may be incorrect.
  + Beta parameter is what controls the trade-off. If one, trats both equally, more than one focuses on recall while less than one focuses on precision
  + F1-measure is when beta is one and most widely used
  + Python example:   
    from sklearn.metrics import f1\_score  
    kNN = KNeighborsClassifier(n\_neighbors=3)  
    dtree = DecisionTreeClassifier(criterion='entropy')  
    mnb = GaussianNB()  
    models = [mnb,kNN,dtree]  
    X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,  
    random\_state=0,  
    test\_size=1/3)  
    print('F1 Measure')  
    for m in models:  
     mm = m.fit(X\_train, y\_train)  
     y\_pred = mm.predict(X\_test)  
     f1s = f1\_score(y\_test, y\_pred, pos\_label='good')  
     print("F1 Score Test set {:22} {:.2f}".format(type(m).\_\_name\_\_, f1s))
* F1 usually rewards balance between TP rate and TN rate but punishes extreme values
* Good BAR score needs good TP rates
* Use sklearns classification report for reports
* Decision Threshold: Value of theta which discriminates the selection of +ve and -ve outcomes. Usually 0.5
* ROC curve: plot of how TP rates and FP rates change over different thresholds
* The better the ROC curve, the closer it is to the top left corner, meaning a larger area under the graph
* Overfitting: When the model is fitted too close to training the data, it no longer is generalised and gives less accurate outputs
* To prevent overfitting, we can use the training-test split or use cross-fold validation
* K-fold cross validation:
  + Divide data into k disjointed sets
  + Select one fold for test and the rest for training
  + Repeat for all folds then average the accuract/error
* Hold out:
  + Split data into test, training and validation set(used to tune classifier)
  + Can use k fold with this
  + Is slower

# Dimension Reduction

* Adding more features (or dimensions) can arguably reduce performance of dataset, reducing accuracy
* To prevent this:
  + Apply pre-processing techniques to reduce number of features then build a model from this reduced dataset
* Feature Selection: Find a minimum subset of the original features which optimises one or more criteria. This includes Information Gain filter.
* Feature Subset selection: Find the best subset of all available features which contains the smallest numbers of features that contribute to the highest accuracy
  + Reasons to do this:
    - Builds a better classifier
    - Identifies useful features
    - Its expensive to work with a lot of features
    - Better interpretability
* Brute force feature selection leads to dCk if k is fixed(k is number of features and d is dimensions ) or 2d if k isn’t fixed
* Filter Feature Selection
  + Here we rank and filter features independently of the choice of classifier that is subsequently applied
  + We use an Evaluation function: How a filter algorithm score a different subset to produce an overall ranking. We aim to score the predictiveness of the features
* Information gain filter:
  + Score all features based on their IG
  + Rank the features
  + Select a subset of top ranked features
    - One can select top k, top 50% or high IG scores
    - Best way is to select top features is to evaluate classification using feature subsets.
      * Do this by Starting with the highest IG feature then add the next one and get its accuracy via Cross Validation. Choose highest accuracy subset   
          
        acc\_scores = []  
        for kk in range(1, X.shape[1]+1):  
         FS\_trans = SelectKBest(mutual\_info\_classif,k=kk).fit(X\_train, y\_train)#Sets up the data transformer  
         X\_tR\_new = FS\_trans.transform(X\_train)  
         X\_tS\_new = FS\_trans.transform(X\_test)  
         seg\_NB = mnb.fit(X\_tR\_new, y\_train)#Tests the feature subset  
         y\_dash = seg\_NB.predict(X\_tS\_new)  
         acc = accuracy\_score(y\_test, y\_dash)  
         acc\_scores.append(acc)  
        df['Accuracy'] = acc\_scores  
        df.head(10)
  + To do this is python:   
    from sklearn.feature\_selection import SelectKBest, mutual\_info\_classif  
    mi = dict()  
    X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,  
    random\_state=2,  
    test\_size=1/2)  
    i\_scores = mutual\_info\_classif(X\_train, y\_train)  
    for i,j in zip(seg\_data.columns,i\_scores):  
     mi[i]=j  
    df = pd.DataFrame.from\_dict(mi,orient='index',columns=['I-Gain'])  
    df.sort\_values(by=['I-Gain'],ascending=False,inplace=True)  
    df.head(10)
  + Problems:
    - No model bias as some features are better for different algorithms
    - No feature dependencies in that we consider them isolated
* Wrapper Feature Selection:
  + Features are directly evaluated based on performance when used with the specific classifier
  + Pros:
    - Takes bias into account
    - Considers features in context
  + Search Strategies:
    - Exhaustive Search: Evaluate all possible 2d subsets
    - Exponential search: Return optimal subset by using heuristics to not iterate over all 2d subsets
    - Sequential search: Add/Remove one feature at a time. Not optimal feature
      * Forward: Start with an empty subset and add most informative feature till no improvement
      * Backwards: Start with all features and keep removing one (least informative) till no improvement.
      * Backwards produces the better model than forward but slower as forward starts with small subsets and neds less running time
  + Python implementation:   
    from sklearn.neighbors import KNeighborsClassifier  
    from mlxtend.feature\_selection import SequentialFeatureSelector as SFS  
    sfs\_forward = SFS(knn,k\_features=7,forward=True, floating=False, verbose=1, scoring='accuracy', cv=10, n\_jobs = -1)  
    sfs\_forward = sfs\_forward.fit(X, y)
  + Disadvantages:
    - High computational cost, especially if large number of features
    - High risk of overfitting, especially if we don’t have a large amount of training data
* Permutation importance: Randomly permute values of variables and reclassify them. If error significantly increases, variable is important, otherwise it isn’t.
  + knn\_perm = permutation\_importance(knn, X\_train, y\_train, n\_repeats=10, random\_state=0)  
    knn\_perm.importances\_mean # returns the score array
  + General flow is: fir classifier, get a baseline accuracy, shuffle feature values in test set then get accuracy, examine error against non-shuffled error
* Feature Transformation: Takes the original features of a dataset and make it smaller, more compact. This includes Principal Components Analysis
  + Mainly uses linear transformation (Mapping of data to new variables using a linear function)
* Matric multiplication in python uses numpy and the .dot() function
* Projection method: Mapping original d-dimensional space to a new dimensional space with minimum info loss. Important part is to preserve information in data by maximising variation.
* Principal Component Analysis(PCA): A dimensional reduction method which aims to keep as much variance in data as possible
* Eigenvector X is a nonzero matrix that satisfies the equation Xv = λv where λ is a number known as the eigenvalue(how much variance is in data)
* Eigendecomposition is when we factor the matrix into eigenvalues and eigenvectors
  + Python does this using np.linalg.eig(X)
* Eigenvectors of symmetric matrices are orthogonal (point in different direction) to each other
* Principal Component (PC): New dimensions made from the linear combinations of original features. Made from eigenvectors
* Covariance: Tendency of features to vary in the same direction
  + Measured using a covariance matrix
  + Covariance Matrix is XTX/(row-1)
* Steps(Given matrix x with n rows):
  + Get mean and standard deviation of the columns of X
  + Subtract the column means from each row of X and divide by the std  
    deviation to create the normalised centred matrix Y
  + C=YTY/(row-1)
  + Get eigenvalues of C
  + PC’s are given by the eigenvectors of C. i-th PC is given by eigenvector corresponding to i-th largest eigenvalue of C
  + Select an appropriate number of PCs(let that be k) and use them to produce a new reduced n by k representation of the dataset.
* For Python:  
  pca = PCA(n\_components=4)  
  X\_r = pca.fit(X\_scal).transform(X\_scal)  
  # Proportion of variance explained for each components  
  pca.explained\_variance\_ratio\_

# Regression

* Linear dependence: constant rate of increase of one variable with respect to another
* Idea is to predict a value in a continuous set given variables
* Linear Regression: Linear relationship between input variables and output variables
  + Simple Linear regression has only one input variable whereas Multiple Linear Regression has more than one input variable
* General form of linear regression is Y=β0 +X1β1 + .. Xnβn
* Notations:
  + X = Independent variables
  + Y= Dependant variables
  + y = observed value of y
  + yi = predicted value of y
  + β0 = intercept
  + β1 = slope
  + ϵi = error
* Python implementation:  
  A close-up of a computer code

  AI-generated content may be incorrect.
* Residual: Difference between data points and regression line
* We adjust slope until residual above line = residual below line using least square method. We use the following calculations to get a good fit:
  + Residual for I value: ϵi = yi - (β0 +xiβi)
  + Sum of all square residual =
  + β1 = Sxy/ Sxx  where
    - =
* Error in regression model:
  + Total Sum of Squares (SST): 🡪 Total sum of squared deviation in Y from its means
  + Variation in Y explained by regression line (SSR): 🡪Total sum of squared deviation of best for from the mean
  + Unexplained Variation in Y(SSE) =
  + SST (Variability) = SSR (Regression Model) +SSE(Error)
* Correlation:
  + Coefficient of determination (R2) = SSR/SST
  + As R2 is closer to one, the better the fit
  + Correlation != Causation
  + Variance explained = 1 -
  + Mean Absolute Error (MAE(y,y\_hat) =
  + Mean Absolute Percentage Error (MAPE of y, y\_hat) =
  + Mean Square Error:
* Gradient Descent:
  + Idea of iteratively adjusting parameters with the aim of finding a local min for our loss function by taking small steps along the gradient.
  + What happens is we either randomly choose the initial parameter values or use a reasonable guess. Afterwards we get the gradient of the loss function wrt to parameter. Afterwards we adjust said parameter by η (this controls step size).
    - Large η (Overshoot minimum)
    - Small η (May not converge to local min)
    - Always in the range [0.0001,10]
  + Update formula:

A mathematical equation with numbers

AI-generated content may be incorrect.

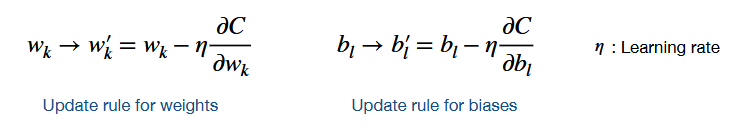
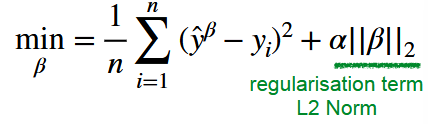
* + - J(βt ) is evaluation of loss function (i.e.) =
    - We usually aim to optimize J(β) by finding the parameter that gives us the minimum J value (i.e.
  + With regards to linear regression:
    - Recall parameters β0 and β1
    - β0 = and β1 =
  + We always stop after either no improvements or after a certain amount of iterations
* Batch Gradient decent is quite slow as we sum overall training examples before making any updates which becomes an issue for large datasets
* Stochastic Gradient Descent is when we randomly choose a data point and do the update from there
* Mini batch gradient descent is when we group together training examples and update that group
* Norm: Quantity which describes size of a vector often defined as ||β||
  + L1 Norm: ||β||1 = | β1 | + | β2 | + . . . + | βn | 🡪 Manhattan Distance
    - Makes some weights 0
    - Good for feature selection
    - Also known as Lasso
  + L2 norm: ||β||2 = sqrt( β12 + β2 2 + . . . + βn2)
    - Good for preventing overfitting
    - Weights scale in proportion
    - Also known as Ridge Regression
  + Elastic net is a combination of the two
* We use regularisation to prevent overfitting by constraining models capacity
* In logistic regression the dependent variable is the log odds that an outcome variable is 1:
  + P is always between 0 and 1
  + The log odds is the dependant variable in this case

# Selecting Model

* Pipeline: When we group steps which will always be done together
  + Code:  
    Pipeline(steps=  
    [('imputer', KNNImputer(missing\_values = np.nan)),  
    ('scaler', StandardScaler()),  
    ('classifier', Classifier())])
* Usually includes hyperparameter tuning
* Hyperparameters are parameters set by hand
* Imputation is a preprocessing step which we fill in missing values. Access to test data can impact it
  + Pipeline should be X is spit into train/test in which the train data in which the train is put into the Imputer then classified
  + Test data should not be here
* Hold out Testing Pipeline:
  + Two Transformation function, one imputer and a scaler and then the estimator. Here we do the standard fit, predict then accuracy score
* Cross Validation Pipeline:
  + Like above, it uses an imputer, scaler and classifier but it’s a new imputer and scaler for each fold as opposed to overall. So it would look like cross\_val\_score(pipeline, x ,y, cv =n)
* Data Leak: When model training or selection accesses training data
  + Feature selection is prone to this
  + Correct way to select feature: FSpipe = Pipeline(steps=[  
    ('fs', SelectKBest(mutual\_info\_classif, k=7)),  
    ('classifier', KNeighborsClassifier())])
* Grid Search: All hyperparameter combinations
* IN a running grid search, sets are scored based on default classifier
* Code is as follows:  
  param\_grid = {'n\_neighbors':[1,3,5,10],  
  'metric':['manhattan','euclidean'],  
  'weights':['uniform','distance']}  
  knn\_gs = GridSearchCV(classifiers, param\_grid, cv=10,  
  verbose = 1, n\_jobs = -1)
* Note that GridSearchCV is a classifier and we can use.best\_params\_ to get the best parameter
  + Classifier= classifier(\*\*model.best\_params\_) or model.best\_params\_ and manually type it into classifier or even model.predict()
* Can combine pipeline and grid search as follows:
  + kNNpipe = Pipeline(steps=[  
    ('imputer', KNNImputer(missing\_values = np.nan)),  
    ('scaler', StandardScaler()),  
    ('classifier', KNeighborsClassifier())])  
    param\_grid = {'scaler':[StandardScaler(), MinMaxScaler(),'passthrough'],  
    'classifier\_\_n\_neighbors':[1,3,5,10],  
    'classifier\_\_metric':['manhattan','euclidean'],  
    'classifier\_\_weights':['uniform','distance']}  
    pipe\_gs = GridSearchCV(kNNpipe,param\_grid,cv=10,  
    verbose = 1, n\_jobs = -1
  + to access all results, we use pd.DataFrame(pipe\_gs.cv\_results\_)
* If param space is huge, we can use a randomised search as opposed to an exhaustive search.

# Neural Networks

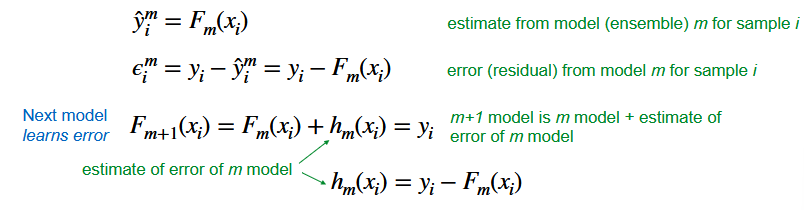
* Based of Neurons
* Receives data from previous nodes and makes use of them
* Most transformer models use these
* Add weights to decisions to indicate importance, higher weight, the higher the importance of that factor
* Neurons are nodes which take in many outputs and produce a single output
* Weight is multiplied by value and totalled for all nodes and depending on final value, a decision is made
* Activation function: When we apply a function to the weighted sum of input sums)
  + Sigmoid
  + ReLu – max(0,x)
  + Hyperbolic tangent=
* Perceptron: Neuron whose output is binary signal based of a threshold function (0 if weighted sum is below a threshold)
  + Single Perceptron only handle linearly separable problems (line divides fires and non-fires such as AND or OR functions
* Bias: Term included on network which we add directly (think of its node value always being 1 with a weight of b).
  + Shifts activation function
* Multilayer Networks are neural networks with multiple layers. They include hidden nodes and can have multiple output nodes
  + Helps solve non-linear problems
  + Decisions made from previous layers
  + Hidden nodes stacked together form a hidden layer
* Feed-forward network: Information flowing in one directions
  + Output is calculated by computing activation of each successive layer
  + No connections in same layer
  + Can add bias in each layer
  + Sum up weights \* value for all inputs in node, then calculate activation of that output to get node output
* Adjusting weights in NN can help reduce error
* Cost/Loss Function: A function that takes in weight and bias in a neural network to quantify inconsistency between predicted and correct value.
  + Task Dependant
  + Lower the cost, better training algorithm
* Mean Squared Error

  + n is number of examples in training set
  + Mainly used in regression
* Cross Entropy
* Gradient Descent is used here and we usually derive the cost function
  + Aim is to choose change in variable so we get closer to a minimum by having the change in Cost be negative
* 
* Stochastic gradient is often used as batch gradient descent is slow on large training sets.
  + Estimates change in cost function for a small batch of random training samples
* Forward propagation: Feed examples through network and calculate outputs then use a cost function to get difference between predicted and actual outputs
* Backpropagation: Propagate errors from output through network allowing us to get gradient of cost function
* Weight update: Gradient is fed through optimisation method which then updates weights to minimise cost function
* Forward propagation🡪 Calculate cost🡪 Backpropagation🡪 Weight updates then restart
* Training of Neural Networks:
  + Choose small random values for weights
  + Repeat till values converge:
    - Set gradients to 0
    - Predict output from model for each training example
    - Calculate resulting cost
    - Update gradients for each weight and bias on training example
    - Update weights and biases via weight update rule
* NN in sklearn:
  + mlp = MLPClassifier(max\_iter=2000, random\_state=2,  
    hidden\_layer\_sizes=[20])
* To avoid overfitting:
  + Use regularisation to restrict weights
  + Restrict complexity of NN
* Regularisation:
  + L2 
  + Scikit-learn’s L2 Norm uses a weight matrix
  + We want to find beta (weights and biases) which will minimise cost function
  + Higher alpha, higher regularisation
* NN’s are sensitive to model parameter selection and weight initialisation
* We can use grid search to get optimal weights
  + RandomisedSearchCV searches over parameters randomly as opposed to GridSearchCV’s cross validation over grid of parameters
* Fast performance once trained, Learn linear and non-linear models and complex problems. Can also be used when data is noisy
* Needs a large amount of training examples to perform well and can take a long time. Acts like a black box

# Ensembles

* Ensemble classification is when we combine individual predictions from different classifiers in order to make more accurate predication
* Can use different classifiers
* Collection of weak learners can lead to a strong decision
* Diversity: Differences in decision made by multiple models
* Average probability of a model being correct is >50%, the chance of ensemble of them reaching the correct decision increases as more members are added only if diversity is in pool.
* As newer members are added in ensemble, the newer members will have voting patterns similar to current existing members
* Its important to ensure diversity among models to generate complementary base classifiers via varying the features/training data or hyperparameters.
* To combine the base classifier outputs, we can use averaging, voting or probability averaging. It is dependent on model
* Simple Random Sampling: Randomly selecting a subset of items from a population. Ever member has an equal chance of being selected
  + Without replacement means we deliberately avoid an item more than once
  + With replacement means we can sample an item more than once
* Bootstrap aggregation/Bagging: Train classifiers using a subset of training data with replacement to encourage diversity.
  + Useful for unstable classifiers
* Random subspace: train classifiers on different subset of features which are sampled without replacement
  + Good for knn models
* Majority voting is done after all classifiers vote for a class, each vote carries equal weight, and the majority is the output of ensemble
* Weighted voting: Give higher weight to better performing classifiers so they contribute more to ensemble
* Balancing accuracy and diversity is important for ensembles to work well
* Ensemble error is the average error of classifiers – Ambiguity of ensembles
* Some classifiers in ensemble are specialised. Meaning they accurately classify related examples from regions of input data space.
* Example bagging classifier  
  tree\_bag = BaggingClassifier(dtree,  
  n\_estimators = 10,  
  max\_samples = 1.0,  
  bootstrap = True) – 10 memebrs with replacement
* random\_SS\_kNN = BaggingClassifier(kNN,  
  n\_estimators = 10,  
  max\_samples=1.0,  
  max\_features=0.5) – random subspace using 50% of features
* Random Trees somewhat combine Bagging and random sampling:
  + Build loads of trees and for each ensemble member:
    - Training set is subsampled with replacement to produce a new training set
    - A subset of features (m) is selected from the set of Features. These are then used in feature selection and when building the tree, m features are selected at random to be the feature which splits the node
* Boosting (Ensembles): Train sequence of classifiers after another so they are trained to work better on labels which was previously poor. They focus on classifying difficult examples
  + AdaBoost, Gradient Boosting are examples of boosting
  + We can assign weights to different examples to focus on the more difficult examples
  + If our data is noisy, a boosting ensemble can actually have a higher error than a component classifier
  + How it works:
    - Assign weight of 1/(number of training samples) to all training examples
    - For 1 to max ensemble size
      * Sample some examples from training set with replacement
      * Train classifier
      * Find misclassified examples and increase their weights while decreasing correctly classified weights
    - Output the final model
  + Code example:   
    from sklearn.ensemble import AdaBoostClassifier  
    adaBoost = AdaBoostClassifier(n\_estimators=100, algorithm = 'SAMME')  
    scores\_adaBoost = cross\_val\_score(adaBoost, X, y, cv=folds, verbose = v, n\_jobs = -1)  
    ab = adaBoost.fit(X,y)  
    ab.estimator\_weights\_[:10]
* Weight update:
  + Alpha is weight of current classifier t while epsilon is error of classifier
* Up-weighting and down-weighting:
  + A mathematical equation with a blue and black text

    AI-generated content may be incorrect.
  + Z\_t is a normalising constant to normalise everything
  + If a sample is correctly classified, the model would be down-weighted and if it was incorrectly classified, it would be up-weighted
* Bias: Systematic error in particular direction
* Variance: Variability in predictability due to changes in training systems
  + 
* High bias but low variance lead to underfitting while low bias but high variance leads to overfitting
* Better to stop at an early epoch
* Gradient boosting is a good ensemble as it uses Gradient Descent to minimise error on next tree



* A math equations with black text

  AI-generated content may be incorrect.

# Clustering

* Unsupervised Learning: Identify patterns solely from data alone, no class information.
  + Can be used in image segmentation, market analysis or in biology
* Topic modelling: Finding underlying thematic structure in text
* Main goal of clustering is to assign similar items into the same clustering while keeping others out
* Partitional clustering: Group data into flat group which consist of disjoint clusters.
  + Specify value k for this, usually chosen at random and use heuristic to find best cluster
* K-means clustering
  + Centroid: Mean vector of all items assigned to a cluster
  + Calculated as an average
  + Aim is to minimize distance between items and nearest centroid by minimizing sum of squard error and using Eucllidean Distance
  + Steps:
    - Calculate initial clusters
    - Get centroid
    - Reassign points to nearest centroid
    - Update the centroids again and keep repeating process till no change
  + K-value is important as too low leads to clusters merging when they shouldn’t whereas too high a k leads too many similar clusters which don’t merge
  + Different initial clusters lead to different solutions
  + Positive:
    - Fast and easy
    - Useful in variety of tasks
  + Negatives:
    - Must pre-specify k
    - Sensitive to initial cluster choice
    - Spherical assumption
    - Empty clusters can happen
  + Code example:
    - penguins\_all = pd.read\_csv('penguins\_af.csv')  
      penguins = penguins\_all[['bill\_length\_mm', 'bill\_depth\_mm',  
      'flipper\_length\_mm', 'body\_mass\_g']]  
        
      LE = LabelEncoder()  
      penguins\_all['code'] = LE.fit\_transform(penguins\_all['species'])  
      y = penguins\_all['code']  
        
      X\_raw = penguins.values  
      X = StandardScaler().fit\_transform(X\_raw)  
        
      Xc = X[:, [f1,f2]] # Select the 0th and 3rd columns  
      km = KMeans(n\_clusters = 3, random\_state=1)  
      km.fit(Xc)  
      print(km.cluster\_centers\_)
* Hierarchal Clustering:
  + Dendrogram: Tree diagram used to show how clusters are arranged after a hierarchical clustering algorithm. Generic at the top and more specific at the bottom
    - dendrogram(linked,  
      orientation='left',  
      labels=labelList,  
      distance\_sort='descending',  
      show\_leaf\_counts=True)
  + Cluster Metric:
    - Single Linkage: Cluster distance is smallest distance from item to cluster. Leads to long chains
    - Complete Linkage: Cluster distance is the largest distance from item to cluster. Sensitive to outliers
    - Average Linkage: Cluster distance as average of all pairwise distance between items to cluster
  + Two types:
    - Agglomerative: All items are in their own cluster, then merge most similar clusters.
      * Takes in a distance matrix(specifies distance between each pair of items) and a cluster metric(how we decide which clusters to merge)
      * Assign each value a cluster
      * Merge two closest clusters according to cluster metric
      * Compute the distance between new cluster and remaining clusters ten repeat above two
      * Code: hc = AgglomerativeClustering(n\_clusters=k,  
        linkage = linkage\_type)
    - Divisive; Begin with a single cluster then the cluster is then split into 2 sub-clusters
* Cluster validation: Measures for evaluating a cluster
  + Used for param selection
  + Silhouette measure is an example of this
    - First we get the average distance to all items ins same cluster to centroid (a\_i)
    - We then repeat this with the second closest cluster items to first cluster’s centroid (b\_i)
    - Silhouette width is then calculated as s\_i=(b\_i - a\_i)/max(a\_i,b\_i)
    - Average silhouette width is then calculated as 1/n \* sum of all silhouette widths
    - We then use this to find best number of clusters

# Spectral Clustering

* Graph Partition aims to split a graph into 2+ parts by minimising cut size(number of edges cut)
  + We usually turn this into a spectral problem by computing eigen decomposition of Laplacian matrix of the adjacency matrix
  + To partition the graph in 2 we split it using the egeinvector of 2nd smallest eigen value known as the Fiedler Vector
  + Recall adjacency matrix S means we put a one when two nodes are connected otherwise 0
  + We then sum the row or columns from the adjacency matrix to get the degree matrix D
  + We then do D-S to get the Laplacian matrix and then get its eigenvectors   
    from scipy.sparse.csgraph import Laplacian  
    Lkc = laplacian(kc)
  + If there is a clear divide between the table values (i.e. negative and positive) that is the node we cut that node,
  + (Generally depending on how many clusters we want to get, we aim for that number of eigen vectors and then can use k-means to assign them to cluters
* Code example:   
  from sklearn.cluster import SpectralClustering

sclust = SpectralClustering(

n\_clusters=3,

affinity= ‘precomputed')# data will be passed as an affinity matrix

sclust.fit(n3);

In [28]:

sclust.labels\_

Out[28]:

array([2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0], dtype=int32)

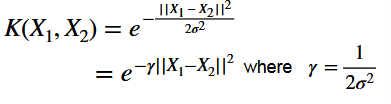
* We can convert a feature vector data to an affinity matrix by creating said affinity matrix and connect it to it’s knn for example:  
  from sklearn.cluster import SpectralClustering

sclust = SpectralClustering(

n\_clusters=2,

affinity='nearest\_neighbors',

n\_neighbors=10)

* One can also use radial basis function to produce an affinity matrix from feature vector data using the following formula  
  
* Spectal clustering always takes in the number of clusters and data in either feature vector or affinity matrix. Although if it’s the former, we need to convert it into the latter.