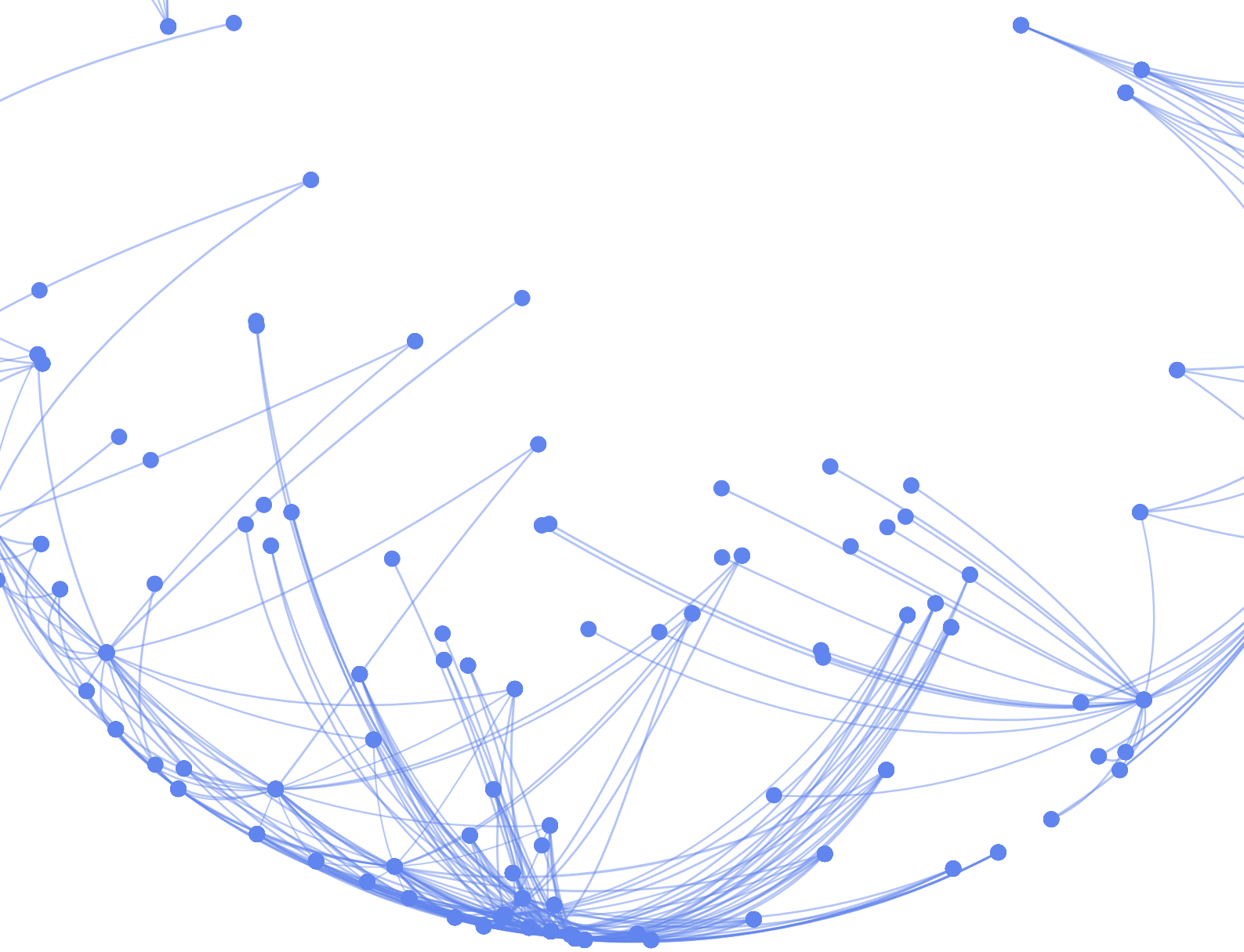
PerceptiLabs

Machine Learning
Handbook



Preface

The aim of this handbook is to make it easier to understand and to start using machine learning. In this handbook, one can find everything from classical machine learning algorithms to deep neural networks. Every algorithm has an explanation of when to use it and how to get the structure of the data right by preprocessing it appropriately.

The algorithms are also written in pseudocode, so it is easy to understand how to implement them and how they work.

The input data may look different for different algorithms; sometimes the data will be *univariate* (one independent variable) and sometimes *multivariate* (several independent variables). This is only for ease of describing the algorithm, and not necessarily a requirement.

Depending on the limitations of the dataset, we may partition the dataset into training/validation/test sets (e.g. 70%/20%/10%) or use *k-fold* cross-validation.

The preprocessing and postprocessing methods are described in detail in the *Appendix*.

Table of contents

1.0 Linear Regression

Linear regression is the most basic type of regression. It is often used in predictive analysis. The regression estimates are used to explain the relation between one dependent variable (Y) and one (or more) independent variable(s) (X). The goal is to minimize the sum of the squared errors to fit a straight line to a set of data points (see *Figure 1*).

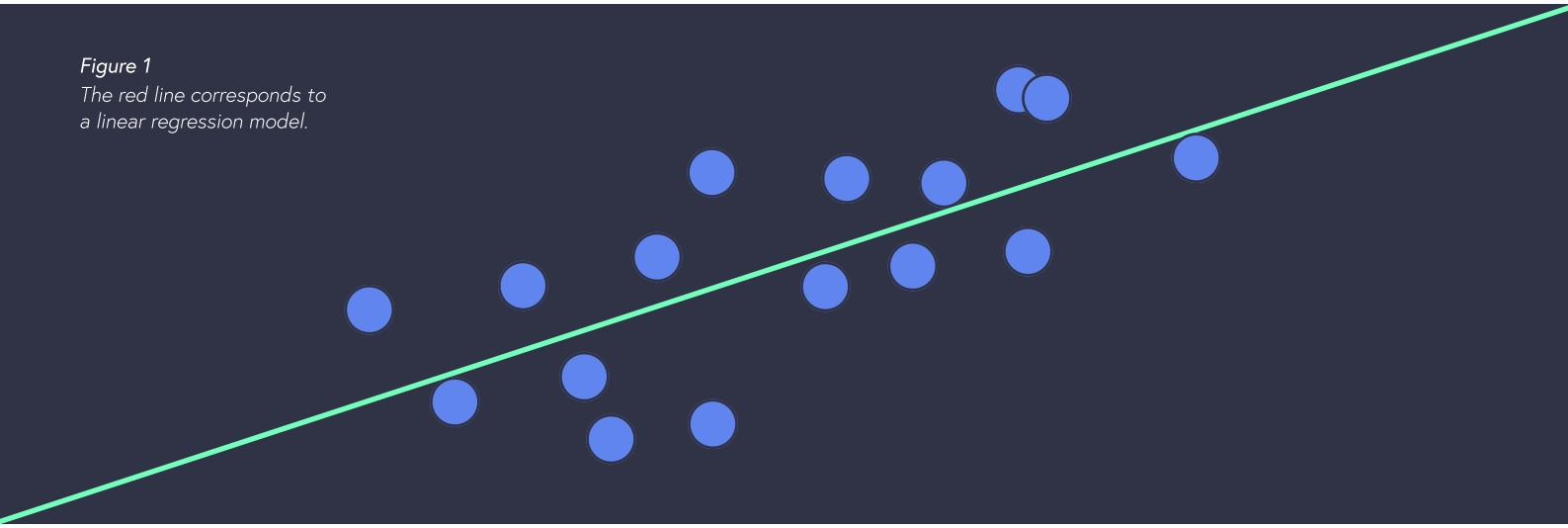


Figure 1
The red line corresponds to a linear regression model.

The simplest form of a linear regression model is given in Eq. 1. This is the *simple linear regression*, which only uses one independent variable.

$$Y(X) = c + bX \quad (1)$$

where b is a regression coefficient and c is a constant. *Multiple linear regression* is shown in Eq. 2.

$$Y = b_0 + b_1 X_1 + \ldots + b_N X_N \quad (2)$$

The main areas where linear regression is useful are:

- Forecasting an effect
- Trend forecasting

Linear regression assumes a link or relationship between one (or more) independent variables and a dependent variable.

Different types of linear regression are:

- Simple linear regression
  - 1 dependent variable, 1 independent variable
- Multiple linear regression
  - 1 dependent variable, 2+ independent variables
- Logistic regression **(for classification problems)**
  - 1 binary dependent variable, 2+ independent variables

It is important to consider the model fit when choosing which model to use, i.e., choosing the regression coefficient *b* and the constant *c*. Adding more independent variables to a linear regression model will always increase the variance of the model (often expressed as R^2). Adding more variables makes the model more inefficient, and *overfitting* can occur. *Occam's razor* applies here: always choose the simplest model possible, i.e., as few independent variables as possible.

Underfitting can also occur. This happens when the model estimates are biased. It often occurs when linear regression is used to attempt to prove a relationship that does not exist.

The standard way to evaluate the model is to use a *cost function*. Then the cost for each model is calculated as the root mean squared error (RMSE) or the R^2. The chosen model is the one that has the minimum RMSE or R^2. But a model (the variables *b* and *c*) can be chosen in different ways. The most common ways are shown in Eq. 3 and Eq. 4. This approach should give the optimum solution. Another way is to use the stochastic gradient descent (SGD), which is the most popular way to calculate the parameters in *neural networks*. By using SGD, one must set hyperparameters such as the learning rate and the number of training iterations. To choose the perfect hyperparameters, trial and error is necessary. Also, more preprocessing of the data, like scaling features, is often necessary.

$$b = \frac{covariance(X,Y)}{variance(X)} \quad (3)$$

$$c = mean(Y) - b \cdot mean(X) \quad (4)$$

Linear regression is a *global model*, in which a single predictive formula holds over the entire data space. If the data have lots of features that interact in nonlinear ways, using a single global model is not the way to go.

1.1 Preprocessing

Remove outliers from the dataset. Outliers will affect the regression negatively by giving a large mean square error (MSE). The red dot in *Figure 2* is an outlier.

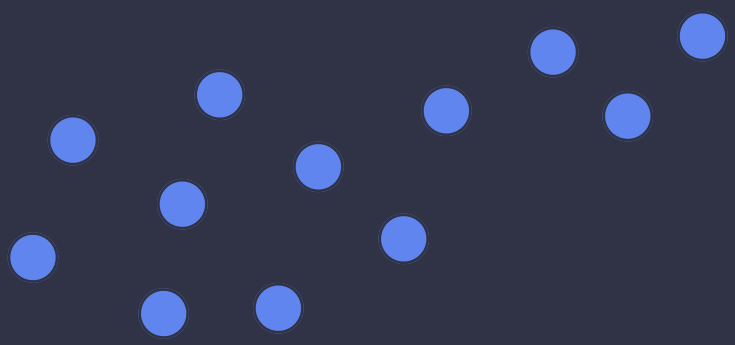


Figure 2
The red data sample is an outlier.

Be aware of missing values. A missing value is when there is a Nan or zero instead of a sample in the dataset.

The following methods can be used to create artificial values:

- Global constant
- Mean/mode of the attribute
- Mean/mode of the attribute, but only the *k-nearest* examples
- Learn a model to predict the missing data (regression, Bayesian)

If using SGD and multiple linear regression, perform feature normalization so that the mean value of each feature in the data is zero and the standard deviation is one.

1.2 Algorithms

Simple Linear Regression

Input: Set of datapoints $X=\{x_1,x_2,\ldots,x_N\}$ and corresponding targets $Y=\{y_1,y_2,\ldots,y_N\}$

Output: The constant c and the regression coefficient b

Method:

1. Calculate the mean of set X ($\bar{X}$) and set Y ($\bar{Y}$)
2. Calculate the covariance by
$$cov = \sum_i^N (x_i - \bar{X})(y_i - \bar{Y})$$
3. Calculate the variance of X by
$$var = \sum_i^N (x_i - \bar{X})^2$$
4. Calculate b as
$$b = \frac{cov}{var}$$
5. Calculate c as
$$c = \bar{Y} - b\bar{X}$$

Multiple Linear Regression

Input: Set of data $X=\{x_i^0, x_i^1, \ldots, x_i^M\}$, $i= 1, \ldots, N$ and corresponding targets $Y=\{y_1,y_2,\ldots,y_N\}$

Output: The regression coefficient vector B

Method:

1. $B = (X^T X)^{-1} X^T Y$, where X^T is the transpose of X

SGD Simple Linear Regression

Input: Set of datapoints $X=\{x_1,x_2,\ldots,x_N\}$ and corresponding targets $Y=\{y_1,y_2,\ldots,y_N\}$ and learning rate η and number of iterations

Output: The constant θ_0 and the regression coefficient θ_1

Method:

1. Calculate the cost function
$$J(\theta)=\frac{1}{2N}\sum_i^N(\hat{y}_i-y_i)^2=\frac{1}{2N}\sum_i^N(\theta_1x_i+\theta_0-y_i)^2$$
2. Calculate gradients of θ_0 and θ_1
$$\frac{\partial}{\partial\theta_0}=\frac{1}{N}\sum_i^N(\theta_1x_i+\theta_0-y_i),\ \frac{\partial}{\partial\theta_1}=\frac{1}{N}\sum_i^N x_i(\theta_1x_i+\theta_0-y_i)$$
3. Calculate the new updated θ_0 and θ_1
$$\hat{\theta}_0=\theta_0-\eta\frac{\partial}{\partial\theta_0},\hat{\theta}_1=\theta_1-\eta\frac{\partial}{\partial\theta_1}$$
4. Calculate $J(\hat{\theta})$ and make sure it decreases for each iteration
5. Repeat from 1. until convergence

SGD Multiple Linear Regression

Input: Set of datapoints $X=\{x_i^0,x_i^1,\ldots,x_i^M\}$, $i=1,\ldots,N$ and corresponding targets $Y=\{y_1,y_2,\ldots,y_N\}$ and learning rate η and number of iterations

Output: The regression coefficient θ

Method:

1. Calculate the cost function
$$J(\theta)=\frac{1}{2N}(X\theta-Y)^T(X\theta-Y)$$
2. Calculate gradient of θ
$$\frac{\partial}{\partial\theta}=\frac{1}{N}X^T(X\theta^T-Y)$$
3. Calculate the new updated θ
$$\hat{\theta}=\theta-\eta\frac{\partial}{\partial\theta}$$
4. Calculate $J(\hat{\theta})$ and make sure it decreases for each iteration
5. Repeat from 1. until convergence

2.0 Decision Tree

Decision trees are very popular, since they are so easy to understand. The final decision tree can explain exactly why a certain prediction was made. A decision tree identifies the most significant variable/attribute and the value that gives the most homogeneous set of data.

In a decision tree, each internal node (non-leaf) is labeled with an input feature. The arcs from a node labeled with an input feature are labeled with the possible values of the target, the output feature, or the arc that leads to a subordinate decision node on another input feature.

To create a tree as a machine learning model, one must select input variables and split the points of those variables until a suitable tree is created (see *Figure 3* for a comparison between an ordinary rule-based decision tree and a decision tree used in machine learning). A tree is created from the top down.

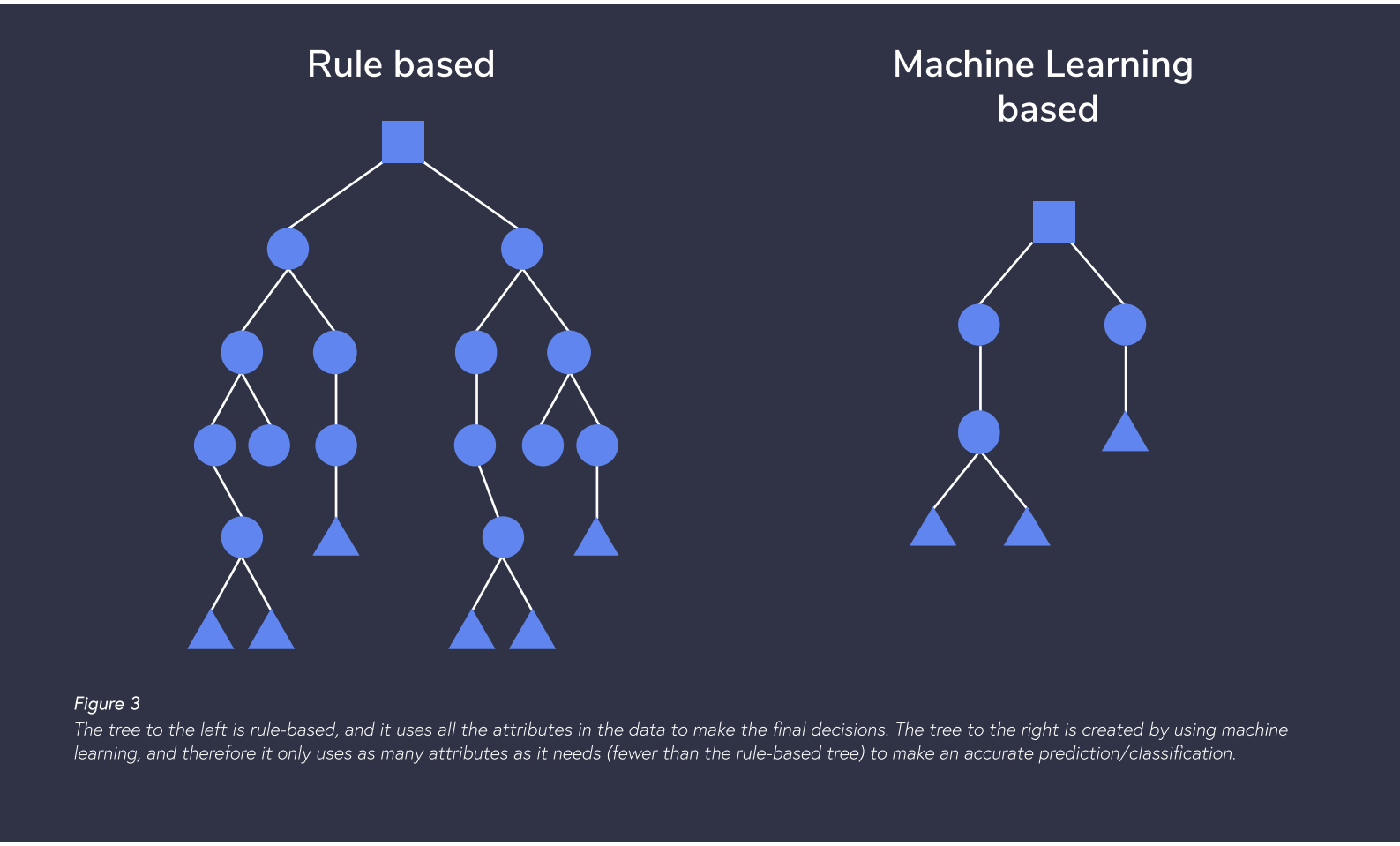


Figure 3
The tree to the left is rule-based, and it uses all the attributes in the data to make the final decisions. The tree to the right is created by using machine learning, and therefore it only uses as many attributes as it needs (fewer than the rule-based tree) to make an accurate prediction/classification.

To select these variables and to get each splitting point, a greedy algorithm is used. That is, the solution is based on the benefit of the next step without considering the larger problem as a whole. The greedy algorithm is used to minimize a cost function. The tree construction ends by using a previously defined stopping criterion.

To perform the greedy splitting (also known as *recursive binary splitting*), i.e., to choose which attribute to use in the root/node, all possible splits at that point in the training are tried and tested using a cost function. The split with the lowest cost is selected.

Tree models in which the target variable can take a discrete set of values are called *classification trees*, and tree models in which the target value can take continuous values (often real numbers) are called *regression trees*.

The cost function for a classification problem will look different than the function for a regression problem.

2.1 Preprocessing

When training a decision tree, it is necessary to know when to stop training. This is best done by splitting the dataset so that *cross-validation* can be used. This is a way to split the dataset when the most common way of splitting a dataset ([train-set 70%, validation-set 20%, test-set 10%] or [train-set 70%, validation-set 30%]) is not viable because the dataset is too small.

The training data are used to train the model, which should always increase the accuracy, while the validation set is used to validate the model after a training iteration to ensure that its accuracy is also increasing. If the accuracy of the validation set starts to decrease, the model is overfitting, and the training should stop. The test set is usually used for a final prediction.

Cross-validation is a technique that averages the prediction errors over a partitioned dataset to get a more accurate estimate of a models performance. The most common cross-validation method is called *k-fold*.

2.2 Postprocessing

To avoid overfitting, i.e., the model learning the training data too well, we prune the tree. This can be done in several ways. One common, simple, and fast way is to replace the node by a leaf with the label instead and then to see if this change significantly affects the accuracy of the trained model. If it does not, the change is kept. The procedure is repeated until a pruning changes the accuracy of the model significantly.

2.3 Classification Tree

In these tree structures, the leaves represent class labels and the branches of the tree represent conjunctions of features that lead to these class labels.

The first thing to do is to choose the best attribute to test first. To decide which attribute should be the root, one must look at the information on every attribute and then do a comparison. There are different ways to do this; the most common method is to compare the entropy (Eq. 5) of the data. Another way is to compare the Gini index (calculate the impurity).

When using the entropy as the method to select important features to use as nodes, we want to consider the information gain (we want to maximize the information gain; Eq. 7), which is calculated from the entropy and the conditional entropy (Eq. 6).

$$H(X) = I_{Entropy}(p_1, p_2, \ldots, p_J) = -\sum_{i}^{J} p_i p_i \quad (5)$$

$$H(X|A) = -\sum_{i,j} p(x_i, a_j) \log log \frac{p(x_i, a_j)}{p(a_j)} \quad (6)$$

$$I_{Gain}(X, A) = H(X) - H(X|A) \quad (7)$$

Where *J* is the number of classes and $p_1, p_2, \ldots, p_J$ are fractions that add up to 1 and represent the percentage of each class in the child node that results in a split in the tree. X is the data and A is the attribute.

It is also possible to consider the number of splits (we want as few splits as possible; Eq. 8), so that the attribute with the highest ratio of Eq. 7 and Eq. 8 is chosen.

$$I_{Split}(X, A) = -\sum_{j}^{J} \frac{|X_j|}{|X|} \log log \frac{|X_j|}{|X|} \quad (8)$$

Where $|X_j|$ is the number of examples in branch *j*.

The Gini index or impurity explains how often a randomly chosen element from the set would be incorrectly labeled if it was labeled randomly according to the distribution of labels in the subset. Eq. 9 shows how it is calculated.

$$I_{Gini}(p) = \sum_{i}^{J} p_i(1 - p_i) \quad (9)$$

Where *J* is the number of classes and $p_1, p_2, \ldots, p_J$ are fractions that add up to 1 and represent the percentage of each class in the child node that results in a split in the tree.

Keep splitting until:

- There are no more examples left
- A node contains a minimum number of data points
- All examples have the same class label (impurity is zero)
- There are no more attributes to split

2.3.1 Algorithm

Input: Set of datapoints $X=\{x_i^0, x_i^1, \ldots, x_i^M, c_i\}$, $i= 1, \ldots, N$ and c is the label

Output: A decision tree

Method:

1. A. Calculate the Root of the tree, by calculating the Gini index of X (except the last element). The attribute with the lowest Gini index will be the Root

   B. Store the split index, the split attribute and the split groups (2 groups)

2. Repeat 1. (instead of Root of the tree, the parent node) using the split groups as datasets

3. Stop when predefined stopping criteria is reached

4. The label with maximum count in the leaf is the predicted class

Decision Tree for Binary Classification

2.4 Regression Tree

What distinguishes a regression tree from a classification tree is that the leaf does not contain a label. There are a certain number of data points (learned by the model/tree) at each leaf, and each leaf has its own model that predicts the data points at that leaf. There are different ways to predict the data points, but two common ways are to use an average or to do a linear regression.

To handle the splits in the regression tree, look at the standard deviation (Eq. 10), rather than calculating the entropy or the Gini index.

In Eq. 10 - Eq. 12, $\overline{X}$ is the mean, P is the probability, and N is the number of values. The split with the highest standard deviation reduction (SDR), calculated as in Eq. 12, is the criterion to split.
A common stopping criterion in this case is when a node contains a minimum number of data points, e.g., when a node contains less than 5% of the data.

$$I_{SD}(X) = \sqrt{\frac{\sum (X-\overline{X})^2}{N}} \quad (10)$$

$$I_{SD}(X, A) = \sum_{c \in X} P(c) I_{SD}(c) \quad (11)$$

$$I_{SDR}(X, A) = I_{SD}(X) - I_{SD}(X, A) \quad (12)$$

2.4.1 Algorithm

Input: Set of datapoints $X=\{x_i^0, x_i^1, \ldots, x_i^M, r_i\}$, $i= 1, \ldots, N$ and r_i is the target

Output: A decision tree

Method:

1. A. Calculate the Root of the tree, by first calculating the standard deviation of the last column of X

   B. Calculate the standard deviation for two attributes, where one of the attributes is always the last column of X

   C. Calculate the SDR by subracting B. from A.

   D. Choose the split that gives the highest SDR and store the split index, the split attribute and the split groups

2. Repeat 1. (instead of Root of the tree, the parent node) by using the split groups as datasets

3. Stop when predefined stopping criteria is reached

4. The prediction is the mean of the data points in the leaf

Decision Tree for Regression

3.0 k-Nearest Neighbor

k-NN is a type of *lazy learning*, which means that generalization of the prediction is not beyond the training data until a query is made to the system. This is one of the simplest machine learning algorithms. It is most useful for large datasets with few attributes. As the number of attributes (the dimensions) increases, the input space increases exponentially. This will lead to data points with similar attributes that may be separated by large distances, which is a problem for the k-NN algorithm. This problem is more generally known as the *curse of dimensionality*.

The k-NN algorithm takes the *k* nearest neighbors (see *Figure 4*) and either takes the means or the medians of those values (regression) to predict the next sample or takes the modes of those labels (classification) to predict the label of the next sample. In the classification case, it is wise to make *k* an odd number if the number of classes is even and vice versa to avoid a tie.

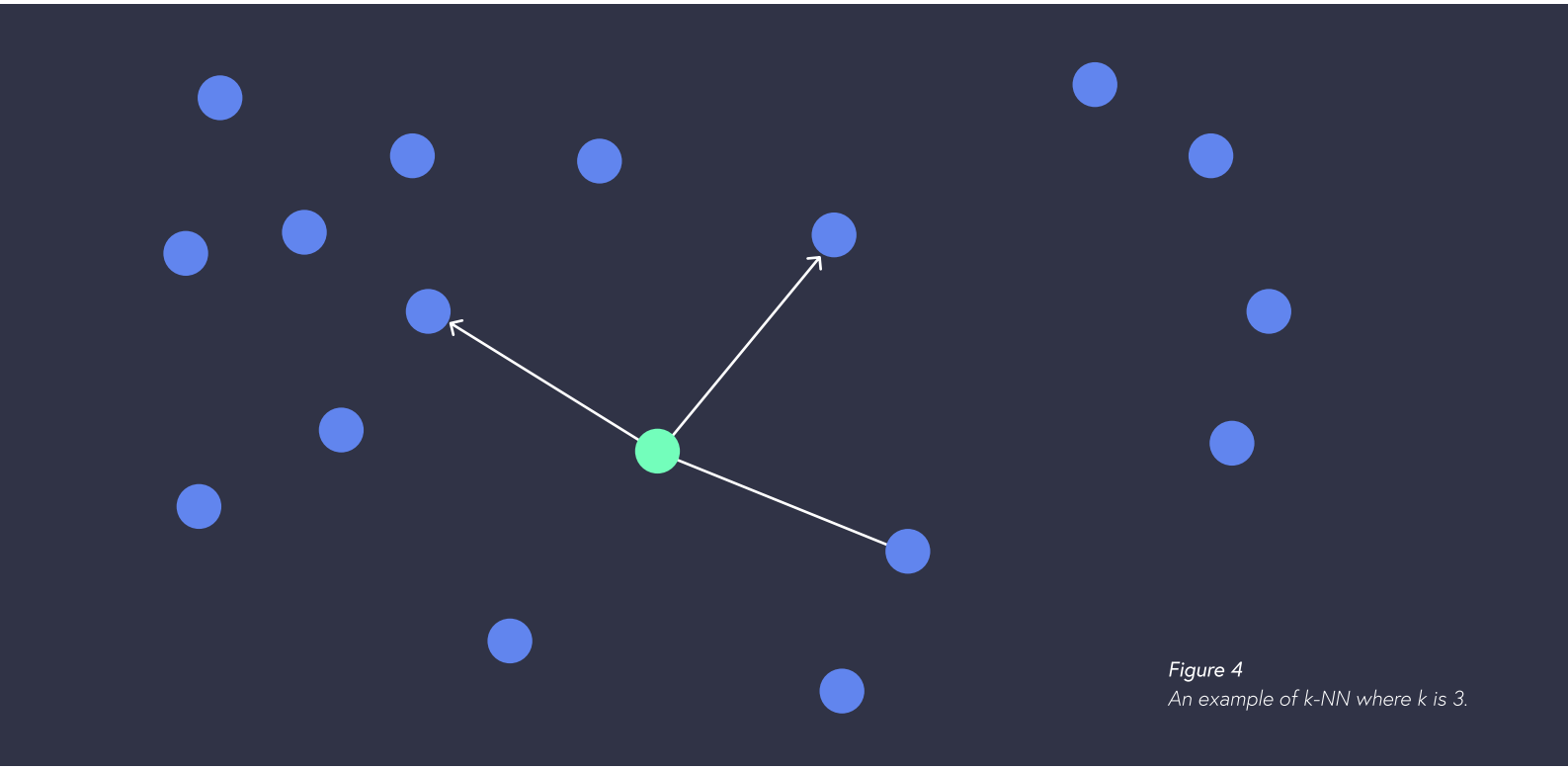


Figure 4
An example of k-NN where k is 3.

For a more accurate prediction, in some cases it is possible to weight each neighbor by $\frac{d}{D}$, where *d* is the distance to the actual neighbor and *D* is the total distance to all neighbors.

Some popular distance measures are:

- Euclidean
- Manhattan
- Minkowski
- Jaccard
- Mahalanobis
- Cosine

where the most common one is *Euclidean distance*. However, it is important to use the distance measure that suits the data/problem to solve, e.g., if the input variables are similar in type, Euclidean distance might be the way to go, and if the input variables differ in type, Manhattan distance is preferred.

3.1 Preprocessing

The following preprocessing methods are not necessary for the use of the k-NN algorithm, but some of them might give better results. This depends on the task and the data.

- Scale the data
- Handle missing values
- Perform dimensional reduction, if there are too many dimensions
  - PCA
  - ICA
  - LDA

3.2 Algorithm

Input: Set of datapoints $X=\{x_1,x_2,\ldots,x_N\}$ and corresponding targets $Y=\{y_1,y_2,\ldots,y_N\}$ and k

Output: The sample value or class

Method:

1. Find the k nearest neighbors by using distance measure

$$d(x_i, x_j) = \sqrt{(x_i - x_j)^2}$$

k-NN

2. Calculate the mean or median of those sample values for *regression*
3. Calculate the mode of those sample labels for *classification*
4. Return that value or label

4.0 Support Vector Machine

A support vector machine (SVM) is a supervised machine learning algorithm that is commonly used for classification, regression, and outlier detection, and it is considered as one of the best off-the-shelf algorithms. The fact that it is supervised means that you need to have labeled data for it to learn. As with many other machine learning algorithms, some hyperparameters need to be set.

The goal of an SVM is to maximize the margin between the decision boundary and the closest data point. The decision boundary is the line that separates the data that belong to different clusters (see *Figure 5*).

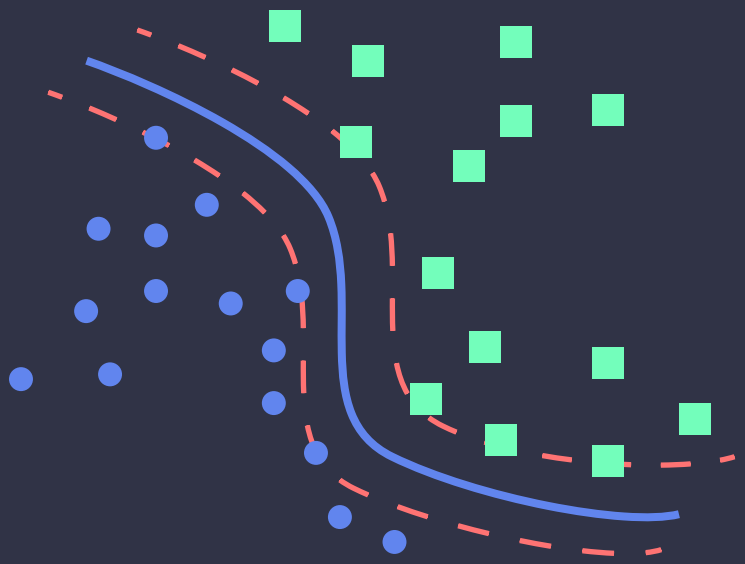


Figure 5
The green dashed line is the margin of the boundary line (blue solid line).

In the ideal case, the data are linearly separable, with no outliers to destroy the classification. This is not always the case, but luckily SVM is flexible and capable of handling both nonlinear cases and cases where a few outliers do not match.

To handle nonlinear cases, SVM uses the *kernel trick*. A common way in machine learning to separate data that are not linearly separable is to "send" the data to a higher dimension and try to separate them there instead. To do this, to every point requires a fair bit of calculation, though, and to avoid that, the kernel trick is used.

To deal with outliers and with margins that are too small, SVM uses *slack variables* to ensure that not every point needs to be taken into account fully.

There are two different ways to approach the SVM and to calculate the decision boundary: solving the *primal problem* (Eq. 13) or the *dual problem* (Eq. 15). They are both optimization problems, which aim to maximize the margin, but they are different approaches to the same problem.

$$J = \frac{1}{N}\sum_{i}^{N}\frac{\lambda}{2}||w||^2 + max(0,\ 1 - y_i f(x_i))$$ *(13)*

Where *N* is the number of samples, *y* is the label, *x* is the input, *w* is the weight, and *f* is Eq. 14. $\frac{1}{\lambda} = \frac{NC}{2}$, where *C* is the regularization parameter, and it decides how much slack we will give our decision boundary.

A small *C* allows constraints to be easily ignored, which leads to large margins. A large *C* has the opposite effect. Setting *C* to infinity to enforces all constraints.

The decision boundary, i.e., the final model (Eq. 14), is much like the linear regression model, only now with a built-in margin.

$$f(z) = w^T z + b$$ *(14)*

The benefit of the dual problem is that it is often much more efficient in to use in high dimensions, and it also allows for the kernel trick, which the primal problem does not. In other words, use the dual problem solution if you want nonlinear decision boundaries.

$$\sum_{i}^{N}\alpha_i - \frac{1}{2}\sum_{i}^{N}\sum_{j}^{N} y_i y_j \alpha_i \alpha_j K\left(x_i, x_j\right) \qquad 0 \le \alpha_i \le C,\ i = 0, \ldots, N \qquad \sum_{i}^{N}\alpha_i y_i = 0$$ *(15)*

We can also see the kernel trick being used in calculating the decision boundary (the final model, Eq. 16). Since there only are a few support vectors (a is often 0), the boundary is fast and efficient to calculate.

$$f(z) = \sum_{i}\alpha_i y_i K(x_i, z) + b$$ *(16)*

Common kernels to use and their equations are:

- R Linear kernel (Eq. 17)
- Polynomial kernel (Eq. 18)
- Radial basis function (Gaussian) kernel (Eq. 19)
- Sigmoid kernel (Eq. 20)

$$K(x, y) = x * y + 1$$ *(17)*

$$K(x, y) = (x * y + 1)^p$$ *(18)*

$$K(x,y) = e^{-\frac{(x-y)^2}{2\sigma^2}} \quad (19)$$

$$K(x,y) = tahn(kx * y - \delta) \quad (20)$$

To apply SVM on a regression problem, simply use the normal regression algorithm and replace the loss and regularization in the cost function with *hinge loss* and *squared regularization* as in SVM.

4.1 Preprocessing

It can be a good idea to scale the inputs.

4.2 Algorithm

Input: Set of datapoints $X=\{x_i^0, x_i^1, \ldots, x_i^M\}$, $i= 1, \ldots, N$ and corresponding targets $Y =\{y_1, y_2, \ldots, y_N\}$ and learning rate η and number of iterations

Output: The regression coefficient θ

Method:

1. Calculate the cost function

$$J(\theta) = \frac{1}{N}\sum_{i}^{N}\frac{\lambda}{2}||w||^2 + \max(0, 1 - y_i f(x_i))$$

2. Calculate the new updated θ

$$\hat{\theta} = \begin{cases} \theta - \eta(\lambda\theta - YX) \; if \; Yf(X) < 1 \\ \theta - \eta\lambda\theta \qquad otherwise \end{cases}$$

3. Calculate $J(\hat{\theta})$ and make sure it decreases for each iteration

4. Repeat from 2. until convergence

SVM Primal

5.0 Neural Networks

Neural networks (see *Figure 6*) are mathematical models inspired by the biological neural networks, i.e., our brains.

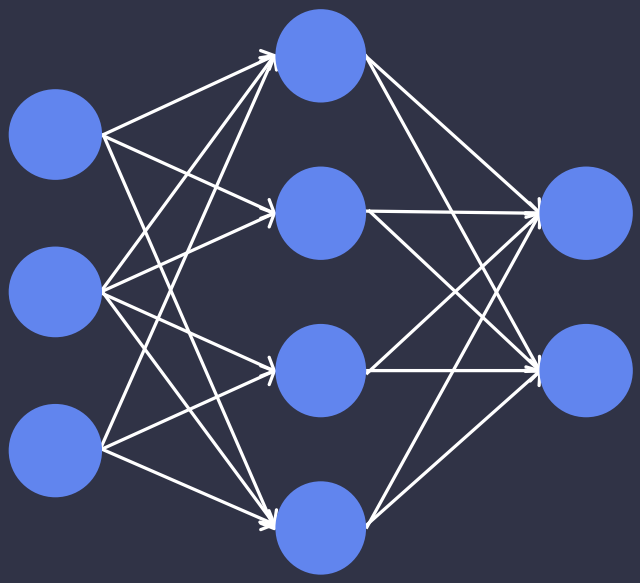


Figure 6
An example of a neural network. The circles to the left corresponds to the input neurons, the circles in the middle correspond to the neurons in the hidden layer, and the circles to the right correspond to the output neurons.

Neural networks consist of multiple layers of neurons (also called units). There is an input layer, there may be one or more hidden layers, and there is an output layer. Normally a neural network is defined as a multi-layer perceptron (MLP) if there are two or more hidden layers, and it is also known as *deep neural network*.

Perceptrons are one sort of neurons. Very briefly, the perceptron takes several binary inputs and produces a single binary output (see *Figure 7*).

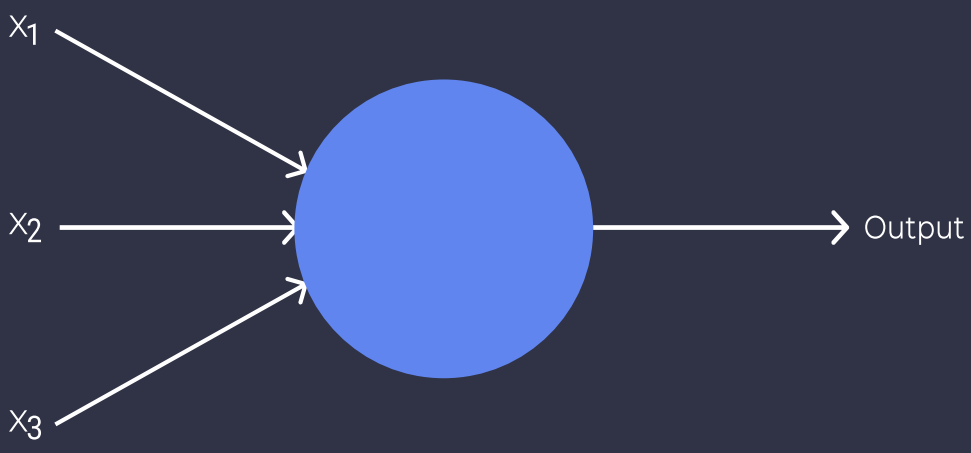


Figure 7
The perceptron.

Each binary input is multiplied by a *weight*, and if the sum of all these input multiplications is greater than some *threshold value*, the output will be 1; else it will be 0. A common approach is to move the threshold value to the same side as the weight and input, and to call it the *bias*.

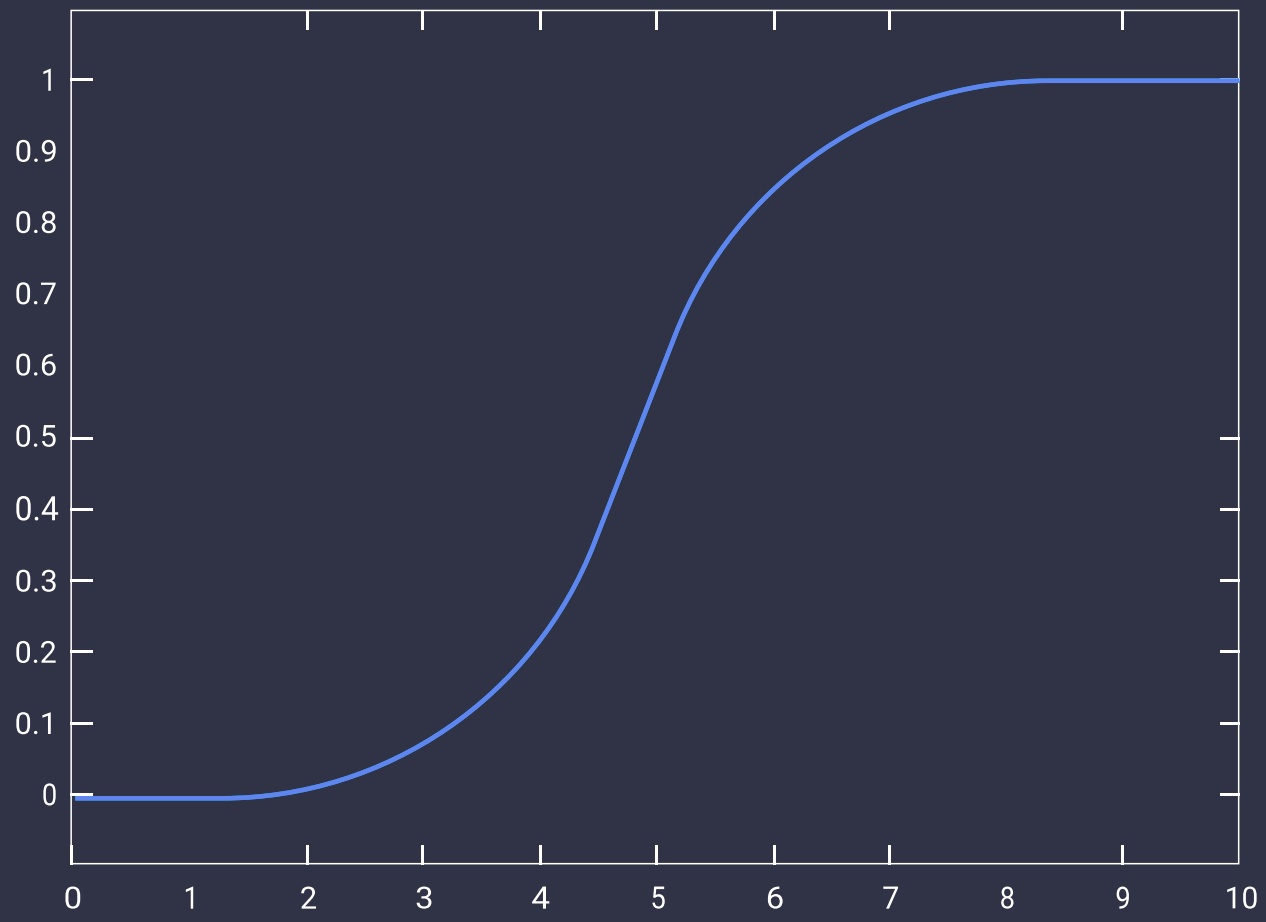
$$output = \begin{cases} 1\,, if\ \sum_i w_i x_i + b > 0 \\ 0\,, if\ \sum_i w_i x_i + b \leq 0 \end{cases} \quad (21)$$

Today, it is more common to have neurons that can take inputs other than binary values, and that can also give an output that is not binary.

Each neuron has several inputs, each multiplied by a weight. These multiplications are added to the threshold value, also called the *bias*, as mentioned above. The final value becomes an input to an *activation function* (the *sigmoid function* is often used as an activation function, see *Figure 8* and Eq. 22), which gives an output that goes to the next layer of neurons. This output differs from the perceptrons, since it is not binary. The great difference is that here a small change in the weights and bias will only cause a small change in the output, which is beneficial for the learning process.

$$f(x) = \frac{1}{1+e^{-x}} \quad (22)$$

Figure 8
The sigmoid function.



The weights and biases must be initialized before training starts, usually with a bias of zero and with random weights. The neural network is evaluated by a cost function at the output of the network. This function tells us if the weights and biases are well chosen, or if they must change. The goal is to find the global minimum of the cost function (though it is easy to get stuck in a local minimum).

To minimize the cost function, a method called backpropagation (see *Appendix*) is used combined with SGD. As the name of the method suggests, the error is propagated backwards in the network, and it simultaneously updates the biases and weights with the help of the backpropagated cost function (Eq. 23 and Eq. 24).

During the forward pass, i.e., using the data as input, the input is sent through the network, where each neuron, in each layer, either activates (send an output) or does not (no output). This will lead to different outputs from the network, depending on the values of the weights and biases. So, for a specific input and specific weights and biases, the network gives an output that is compared with the truth, i.e., the correct classification (for example) for that input.

It is common to use mini-batches when training, instead of the whole input. This leads to a higher variable update frequency and leads to robustness.

A neural network *learns* by updating/changing its weights and biases. This is an iterative process, which takes a lot of time and needs computational resources. In general, the network learns by making a forward pass of the input, calculating the error of the prediction, and then passing back the error to change the weights and biases depending on how wrong the prediction was (as Eq. 23 and Eq. 24 show). The method of the backward pass is the already mentioned backpropagation.

$$w_i \rightarrow w_i' = w_i - \eta \frac{\partial C}{\partial w_i} \qquad (23)$$

$$b_i \rightarrow b_i' = b_i - \eta \frac{\partial C}{\partial b_i} \qquad (24)$$

The most common cost function is the cross-entropy function (Eq. 25). The cross-entropy function considers two probability distributions: the true probability (the true label) and the given distribution (the prediction). The cross-entropy function gives a measure of similarity between these two probability distributions. The cross-entropy function is defined as:

$$C = -\frac{1}{n}\sum_x y_{truth} \ln ln\, y_{pred} + (1 - y_{truth}) \ln ln\, (1 - y_{pred}) \qquad (25)$$

There are several important parameters that control other parameters in the network. These *hyperparameters* must be selected. It is here that the validation data are used. After each epoch (one run over the network with all training data), when new weights and biases have been calculated, the network is tested with the validation data. To do this iteratively after each epoch, one can plot the cost function of the training data and the validation data. From this plot, it is possible to extract information that tells us whether a certain hyperparameter should be changed or not. This is an iterative process, and training neural networks consists of lots of trial and error to find the optimal hyperparameters, if it is even possible.

After this whole procedure, the test data is used to evaluate the network as a final product.

5.1 Preprocessing

Two methods are common for preprocessing the data when using neural networks:

- Transformation, e.g.:
  - 2D image data to 1D array
  - 1D array to 2D image data
  - One-hot encoding
- Normalization

5.2 Algorithm

Input: Set of datapoints $X=\{x_1,x_2,\ldots,x_N\}$ and corresponding targets $Y=\{y_1,y_2,\ldots,y_N\}$ and *hyperparameters* such as η (learning rate), *epochs*, *number of neurons per layer* and *mini-batch size*

Output: The prediction model with trained variables

Method:

1. Initialize the weights randomly and biases with zeros
2. Shuffle the training data and divide it into mini-batches
3. For each weight and bias in neuron i, calculate the output and apply the sigmoid as activation function
$$a_i = \sigma(z_i) = \sigma(w_i x_i + b_i)$$
4. Calculate the *cost function derative* w.r.t the weight and bias by backpropagation
$$\frac{\partial C}{\partial w_i}, \frac{\partial C}{\partial b_i}$$
5. Update each weight and bias, in every layer
$$w_i = w_i - \frac{\eta}{"mini - batch\ size"} \frac{\partial C}{\partial w_i}$$
$$b_i = b_i - \frac{\eta}{"mini - batch\ size"} \frac{\partial C}{\partial b_i}$$
6. Repeat from 3. until convergence

Neural Network

6.0 Clustering

Clustering algorithms are probably the most common unsupervised machine learning algorithms. Unsupervised learning is when there is no truth/label with which to compare the prediction. They divide the data into groups based on the similarity of the data (see *Figure 9*). There are several different types of clustering:

- Connectivity models
- Centroid models
- Distribution models
- Density models
- Etc.

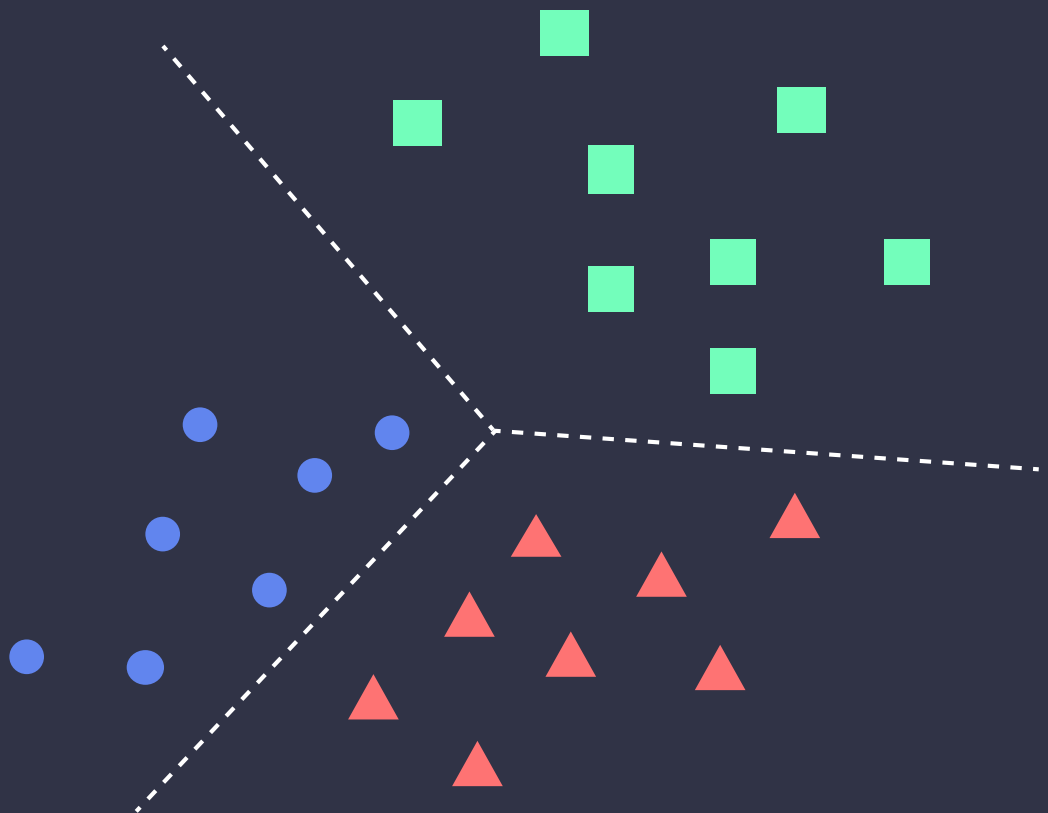


Figure 9
The data divided into three different clusters.

The most famous and popular clustering algorithm is *k-means clustering* (centroid model). The downside with this algorithm is that one must know in advance how many clusters/classes there should be (*k* clusters). There are clustering methods (e.g., density models, like *DBSCAN*) that do not require that knowledge, but they are more computationally heavy.

The k in k-means clustering is the number of cluster centers there will be. The least squared Euclidean distance is calculated between a cluster mean and a data point. The data point gets attached to the cluster with the least squared Euclidean distance. This process is repeated iteratively until the assignments no longer change. There is no guarantee that the optimum will be found using this algorithm.

6.1 Preprocessing

The following preprocessing methods should be considered when using k-means clustering:

- Missing value
- Data/unit normalization
- Dimensionality reduction (curse of dimensionality, see Section 3.0)
    - PCA
    - ICA
    - LDA

6.2 Algorithm

Input: Set of datapoints $X=\{x_1,x_2,\ldots,x_N\}$ and k clusters

Output: The dataset X with each data point x_i classified

Method:

1. Initialize the k cluster centers c_m by random
2. Assign each data point to a cluster by calculating the least square Euclidean distances

$$\sum_{m=1}^{k} \sum_{x_i \in C_m} ||x_i - c_m||^2$$

3. Calculate and update the new cluster centers c_m

$$c_m = \frac{1}{C_m} \sum_{x_i \in C_m} x_i$$

4. Repeat from 2. until the cluster centers c_m stops changing

k-Means Clustering

7.0 Appendix

7.1 Preprocessing Methods

7.1.1 Linear Regression

Input: Set of datapoints $X=\{x_i^0, x_i^1, Nan, \dots, x_i^M\}$, $i= 1, \dots, N$

Output: X with missing values replaced

Method:

1. For each row i in X, if Nan (or unwanted zero)

$$\text{Nan} = \frac{1}{M}\sum_{j=0}^{M} x_i^j$$

2. When all missing variables have been replaced, return X

Missing Values

Input: Set of datapoints $X=\{x_1,x_2,\dots,x_N\}$

Output: X without outliers

Method:

1. Calculate the mean $\bar{X}$ and standard deviation (STD) for each feature
2. For each data point, only keep the data points that

$$\bar{X} - 2STD(X) < x_i < \bar{X} + 2STD(X)$$

3. Return X_{new}

Outliers

Input: Set of datapoints $X=\{x_i^0, x_i^1, \ldots, x_i^M\}$, $i= 1, \ldots, N$

Output: X normalized

Method:

1. Calculate the mean $\bar{x}_i^j$ and standard deviation (STD) for each feature j, for each data point i
2. For each feature j, subtract the mean and divide by the STD

$$x_i^{norm} = \frac{x_i^j - \bar{x}_i^j}{std(x_i^j)}$$

3. Return X_{norm}

Feature Normalization

7.1.2 Decision Tree

Input: Set of datapoints $X=\{x_i^0, x_i^1, \ldots, x_i^M\}$, $i= 1, \ldots, N$

Output: X normalized

Method:

1. Calculate the mean $\bar{x}_i^j$ and standard deviation (STD) for each feature j, for each data point i
2. For each feature j, subtract the mean and divide by the STD

$$x_i^{norm} = \frac{x_i^j - \bar{x}_i^j}{std(x_i^j)}$$

3. Return X_{norm}

Cross-Validation (k-fold)

7.1.3 k-Nearest Neighbor

Input: Set of datapoints $X=\{x_1, x_2, \ldots, x_N\}$

Output: X scaled

Method:

1. For each feature, scale the value to the range [0,1]

$$x_i^{scale} = \frac{x_i}{\max(X)}$$

2. Return X_{scale}

Scale

Input: Set of datapoints $X=\{x_i^0, x_i^1, Nan, \ldots, x_i^M\}$, $i= 1, \ldots, N$

Output: X with missing values replaced

Method:

1. For each row i in X, if Nan (or unwanted zero)

$$\text{Nan} = \frac{1}{M}\sum_{j=0}^{M} x_i^j$$

2. When all missing variables have been replaced, return X

Missing Values

Input: Set of datapoints $X=\{x_i^0, x_i^1, \ldots, x_i^M\}$, $i= 1, \ldots, N$

Output: $X_{reduced}$, i.e., a lower dimensional X

Method:

1. Calculate the mean of X and subtract it from X

$$\tilde{X} = X - \frac{1}{M}\sum_{j=0}^{M} x_i^j$$

2. Calculate the coavriance matrix from $\tilde{X}$

$$\tilde{X}_{cov} = \frac{\tilde{X} \cdot \tilde{X}^T}{N}$$

3. Calculate the *eigenvalues* and *eigenvectors* of $\tilde{X}_{cov}$ and sort the vectors as $\text{E} = [e_{max}, \ldots, e_{min}]$

4. Reduce dimensions by removing *eigenvectors* from the end of the list or array, choose p final dimensions

$$FeatureVector = E[0:p]$$

5. Get the final data by

$$X_{reduced} = X \cdot FeatureVector^T$$

Dimensionality Reduction (PCA)

7.1.4 Support Vector Machine

Input: Set of datapoints $X=\{x_1,x_2,\ldots,x_N\}$

Output: X scaled

Method:

1. For each feature, scale the value to the range [0,1]

$$x_i^{scale} = \frac{x_i}{\max(X)}$$

2. Return X_{scale}

Scale

7.1.5 Neural Networks

Input: Set of datapoints $X=\{x_i^0, x_i^1, \dots, x_i^M\}$, $i= 1, \dots, N$

Output: X normalized

Method:

1. Calculate the mean $\bar{x}_i^j$ and standard deviation (STD) for each feature j, for each data point i
2. For each feature j, subtract the mean and divide by the STD

$$x_i^{norm} = \frac{x_i^j - \bar{x}_i^j}{std(x_i^j)}$$

3. Return X_{norm}

Normalization

7.1.6 Clustering

Input: Set of datapoints $X=\{x_i^0, x_i^1, Nan, \dots, x_i^M\}$, $i= 1, \dots, N$

Output: X with missing values replaced

Method:

1. For each row i in X, if Nan (or unwanted zero)

$$\text{Nan} = \frac{1}{M}\sum_{j=0}^{M} x_i^j$$

2. When all missing variables have been replaced, return X

Missing Values

Input: Set of datapoints $X=\{x_i^0, x_i^1, \dots, x_i^M\}$, $i= 1, \dots, N$

Output: X normalized

Method:

1. Calculate the mean $\bar{x}_i^j$ and standard deviation (STD) for each feature j, for each data point i
2. For each feature j, subtract the mean and divide by the STD

$$x_i^{norm} = \frac{x_i^j - \bar{x}_i^j}{std(x_i^j)}$$

3. Return X_{norm}

Unit Normalization

Input: Set of datapoints $X=\{x_i^0, x_i^1, \ldots, x_i^M\}$, $i= 1, \ldots, N$

Output: $X_{reduced}$, i.e., a lower dimensional X

Method:

1. Calculate the mean of X and subtract it from X

$$\tilde{X} = X - \frac{1}{M}\sum_{j=0}^{M} x_i^j$$

2. Calculate the coavriance matrix from $\tilde{X}$

$$\tilde{X}_{cov} = \frac{\tilde{X} \cdot \tilde{X}^T}{N}$$

3. Calculate the *eigenvalues* and *eigenvectors* of $\tilde{X}_{cov}$ and sort the vectors as $\mathrm{E} = [e_{max}, \ldots, e_{min}]$

4. Reduce dimensions by removing *eigenvectors* from the end of the list or array, choose p final dimensions

$$FeatureVector = E[0:p]$$

5. Get the final data by

$$X_{reduced} = X \cdot FeatureVector^T$$

Dimensionality Reduction (PCA)

7.2 Post-processing Methods

7.2.1 Decision Tree

Input: A decision tree

Output: A pruned decision tree

Method:

1. This prodecure is done from the bottom up. Replace the first node with its most popular class

2. Calculate the accuracy of the new tree

3. If the accuracy is not affected that much, repeat from 1.

Pruning

7.3 Mathematics

7.3.1 Backpropagation

The backpropagation gives an expression of the derivative of the cost function C with respect to the weights and biases. From that we can calculate how fast the cost changes depending on the changes of the weights and biases in the network.

For a cost function to be usable for backpropagation, it needs to fulfill two criteria:

- it can be written as an average over cost functions for individual training examples
- it can be written as a function of the outputs from the network

We calculate the partial derivatives $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$ where w is the weight from neuron *k* in layer (*l* — 1) to neuron *j* in layer *l* and *b* is the bias of the *j* neuron in layer *l*. Then we relate δ_j^l to each partial derivative, which is the error in the *j* neuron in layer *l*.

What we want is for the neural network to perform with as high an accuracy as possible. To increase its accuracy, one must minimize the error. The error comes because the output from an activation function is different than what is optimal, e.g., from activation function output *a* we want the output $\sigma(z_j^l)$, where:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \quad (26)$$

But what we actually get is $\sigma(z_j^l + \Delta z_j^l)$. When this propagates through the network, the final cost will be $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$. Therefore, we choose Δz_j^l so that we minimize the final cost as much as possible, e.g., Δz_j^l can be set to the opposite sign of $\frac{\partial C}{\partial z_j^l}$, to make sure that $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$ is small. This tells us that $\frac{\partial C}{\partial z_j^l}$ acts as the error in each neuron, i.e.,

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \quad (27)$$

The backpropagation algorithm is based on four fundamental equations:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \quad (28)$$

Or, in matrix-form:

$$\delta^L = \nabla_a C \odot \sigma'(z_j^L) \quad (29)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad (30)$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (31)$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \quad (32)$$

where $\odot$ is the Hadamard product, also known as elementwise multiplication, and σ is the activation function.

Eq. 28 is the error in the output layer *L*, Eq. 30 is the error expressed in terms of the error in the next layer (*l*+1), Eq. 31 is the rate of change of the cost with respect to the bias, and Eq. 32 is the rate of change of the cost with respect to the weight.

If a_k^{l-1} in Eq. 32 is small, it is said that the weight *learns slowly*. Depending on the activation function, this is a consequence of the neuron saturating (sigmoid as activation function).

All four equations Eq. 28-Eq. 32 are consequences of the chain rule:

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \quad (33)$$

But the output from the activation function a_k^L of the kth neuron will only depend on the weighted input z_j^L when $k = j$, and it can therefore be rewritten as:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \quad (34)$$

which is the same as:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \quad (35)$$

This is Eq. 29 in component form, i.e., Eq. 28. To prove Eq. 30, one can again apply the chain rule:

(36)

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$$

(37)

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

And, by differentiating, we get:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \quad (38)$$

which, implemented in Eq. 36, gives us Eq. 30 in component form:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \quad (39)$$

For Eq. 32 we can use the chain rule as:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} \quad (40)$$

where the first factor has already been proven to be the error and the second factor is:

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial}{\partial w_{jk}^l} \sum_j w_{jk}^l a_j^{l-1} + b_k^{l+1} = a^{l-1} \quad (41)$$

By implementing Eq. 41 in Eq. 40:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} a^{l-1} = a^{l-1} \delta_j^l \quad (42)$$

For Eq. 31, the first derivative factor is the same as Eq. 40 and the second derivative factor is:

$$\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial}{\partial b_j^l} \sum_j w_{jk}^l a_j^{l-1} + b_k^{l+1} = 1 \quad (43)$$

By implementing Eq. 43, the final result is:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (44)$$