Mathematical Notes

Contents

Part I

Machine Learning

Chapter 1

Introduction

Definition 1.1 (Machine Learning)**.** ***Machine learning*** *is the field of study that gives computers the ability to learn without being explicitly programmed. (Arthur Samuel, 1969)*

Definition 1.2 (Machine Learning)**.** *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E. (Tom Mitchell)*

The essence of machine learning is that a pattern exists and it can not be pined down mathematically, however we have data on it and we can treat it in a probabilistic way.

Thus, machine learning is great for:

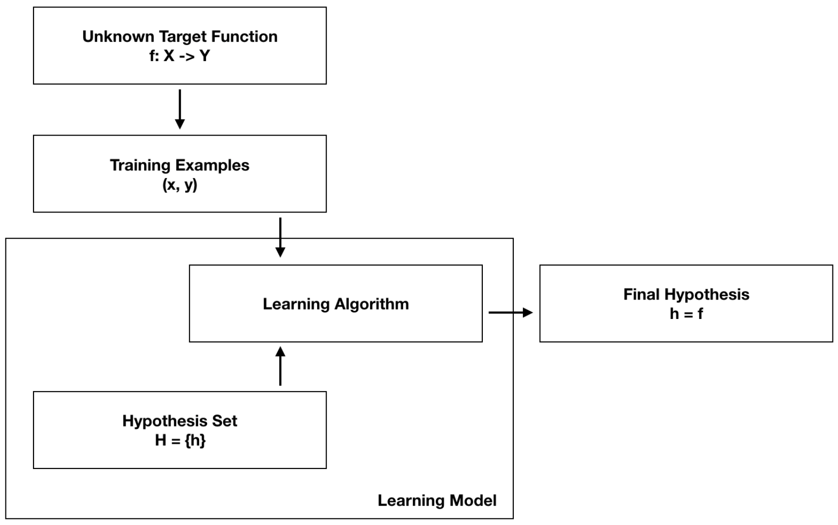
- Problems for which existing solutions require a lot of fine-tuning or long lists of rules: one machine learning algorithm can often simplify code and perform better than the traditional approach.
- Complex problems for which using a traditional approach yields no good solution: the best machine learning techniques can perhaps find a solution.
- Fluctuating environments since a machine learning system can adapt to new data.

Some usual examples of machine learning algorithms are: analyzing images to automatically classify them, detecting tumors in brain scans, automatically classifying news articles, automatically flagging offensive comments on discussion forums, creating a chat-bot or a personal assistant, forecasting company revenue, making an application reacting to voice commands, detecting credit card fraud, segmenting clients based on their purchases, representing a complex high-dimensional dataset in a clear and insightful way, recommending a product that a client may be interested in, building an intelligent bot for a game, and many, many more...

There are some common conventions in the machine learning community around the notation used to describe various notions. We will of course follow the same conventions. In order to briefly formalize the essence of machine learning we will introduce some of the very basic notation that we will be using throughout the notes now, although we will introduce more notation in the later chapters. Here are some very basic concepts with their usual notation:

- Input: $x \in X$.
- Output: $y \in Y$.
- Data: $\{x_i, y_i\}, \quad i = 1, 2, 3, \ldots, m$.
- Target Function: $f : X \to Y$.
- Hypothesis Function: $h : X \to Y$ with $h \approx f$.
- Hypothesis Set: $H = \{h\}$.

Informally, the goal of machine learning is, based on the data $\{x_i, y_i\}$, to discover a hypothesis function h, out of a set of possible hypothesis functions H, that behaves in a similar way with the target function f which is, and always will be, unknown to us.



The question is how can we learn an unknown function f just based on the data we already have, when the unknown function f in general can take any value outside the known data. The short answer is that we can not however, without proving it, the following relation holds:

$$P\Big[|E_{\text{in}}(h) - E_{\text{out}}(h)| > \epsilon\Big] \leq 2 \cdot M \cdot e^{2\epsilon^2 m}$$

where $E_{\text{in}}(h)$ is the error that we get for h in the known data, $E_{\text{out}}(h)$ is the error that we will get when we use h for new data, M is the number of possible hypothesis function h (i.e the cardinality of the hypothesis set $H = \{h\}$, ϵ is the tolerance that we have for errors, and m is the number of data points. This equation tells us that no matter what, learning is possible only in a probabilist sense. We will always have an error, since the whole process carries a stochastic nature.

So we can informally summarize what we are trying to do with machine learning as:

- From aforementioned relation: $E_{\text{in}} \approx E_{\text{out}}$.
- From learning algorithm: $E_{\text{in}} \approx 0$.
- From the combination of these 2: $E_{\text{out}} \approx 0$.

By having $E_{\text{out}} \approx 0$, that means that our hypothesis function h generalizes well for out of sample data, so we can use it for predictions. That in a nutshell is how machine learning works.

There are so many different types of machine learning systems that it is useful to classify them in broad categories, based on the following criteria:

1. Whether or not they are trained with human supervision. Based on this category we have the following subcategories:
   - **Supervised Learning**
     In supervised learning the training set you feed to the algorithm includes the desired solutions, called labels. Some of the most important supervised learning algorithms are: Linear Regression, Logistic Regression, Support Vector Machines (SVMs), k-Nearest Neighbors, Decision Trees, Random Forests, Neural Networks.
   - **Unsupervised Learning**
     In unsupervised learning the training data is unlabeled so the system tries to learn without a teacher. Some of the most important unsupervised learning algorithms are:
     - Clustering algorithms like K-Means, DBSCAN and Hierarchical Cluster Analysis (HCA).
     - Anomaly detection algorithms such as One-Class SVM and Isolation Forest.
     - Dimensionality reduction algorithms such as Principal Component Analysis (PCA), Kernel PCA, Locally Linear Embedding (LLE), and t-Distributed Stochastic Neighbor Embedding (t-SNE).

- **Semisupervised Learning**
  In semisupervised learning one has plenty of unlabeled instances, and few labeled instances. Most semisupervised learning algorithms are combinations of unsupervised and supervised algorithms.
- **Reinforcement Learning**
  In reinforcement learning the learning system, called an agent, can observe the environment, select and perform actions, and get rewards in return. It must then learn by itself what is the best strategy, called a policy, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

2. Whether or not they can learn incrementally on the fly. Based on this category we have the following subcategories:

   - **Offline or Batch Learning**
     In offline, or batch, leaning the system is incapable of learning incrementally and it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore, it just applies what it has learned.
   - **Online Learning**:
     In online learning one trains the system incrementally by feeding it data instances sequentially, either individually or in small groups called “mini-batches”. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives.

3. Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model. Based on this category we have the following subcategories:

   - **Instance-Based Learning**
     In instance-based learning the system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them).
   - **Model-Based Learning**
     In model-based learning one in order to generalize from a set of examples they build a model of these examples and then use that model to make predictions.

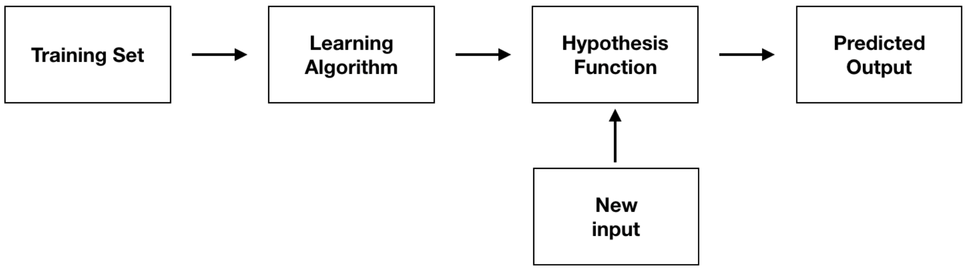
These criteria are not exclusive. You can combine them in any way you like. This fact makes machine learning is a very broad topic with many different branches and applications. In these notes we will cover the vast majority of them.

Chapter 2

Supervised Learning

Supervised learning is one of the four basic categories of machine learning, and it consists of a family of models and techniques that we will introduce in this chapter. First let's start with a formal definition of supervised learning.

Definition 2.1 (Supervised Learning)**.** ***Supervised learning*** *is the machine learning task of learning a function that maps an input to an output based on examples of "input - output" pairs called a "training set".*



Some more specific notation that we will be using throughout supervised learning:

- Input variables, or attributes, or features: x.
- The i^{th} feature: x_i.
- The j^{th} training example: $x^{(j)}$.
- The i^{th} feature of the j^{th} training example: $x_i^{(j)}$.
- Output variables, or targets, or classes, or labels: y.
- The i^{th} target: $y^{(i)}$.
- Total number of training examples: m.
- Total number of features: n.

Now let's dive in the models and techniques of supervised learning, starting with one of the most basic ones called "linear regression".

2.1 Linear Regression

Definition 2.2 (Linear Regression)**.** ***Linear regression*** *is a linear approach to modelling the relationship between a dependent variable (target) and one or more independent variables (features).*

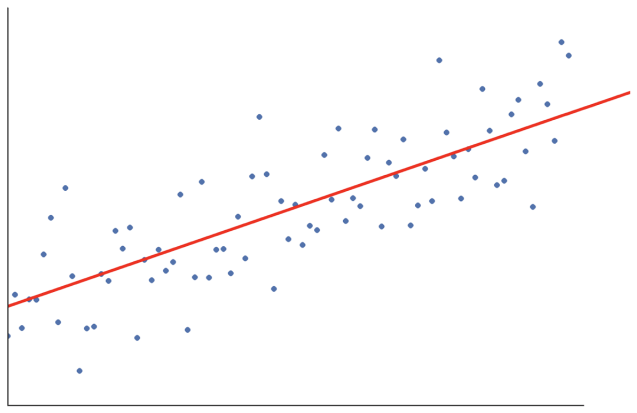
Definition 2.3 (Simple / Multiple Linear Regression)**.** *When there is only one independent variable (feature) then the model is called **simple linear regression**. For more than one independent variables (features) the process is called **multiple linear regression**.*

Definition 2.4 (Univariate / Multivariate Linear Regression)**.** *When only one dependent variable (target) is predicted then the model is called **univariate linear regression**. For more than one correlated, dependent variables (targets) being predicted, the process is called **multivariate linear regression**.*

In linear regression the hypothesis function h is a linear combinations of the features:

$$h(x) = w_0 + w_1x_1 + \ldots + w_nx_n$$

where w_0 is called "bias" or "intercept" and the rest w_i's are called "weights". We usually refer to weights and bias as the "parameters" of the regression (or the model) and they are the ones that we try to determine through the training examples by using a learning algorithm. Once we find them then h is ready to predict new inputs with unknown outcomes.



It is a usual procedure to define $x_0 = 1$, so the linear regression hypothesis function can be rewritten as:

$$h(x) = w_0x_0 + w_1x_1 + \ldots + w_nx_n = \sum_{i=0}^{n} w_ix_i$$

Moreover by defining the feature vector $\boldsymbol{x}$ and parameter vector $\boldsymbol{w}$ as:

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}, \qquad \boldsymbol{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$$

we can rewrite the linear regression the hypothesis function in the very simple form of:

$$h(x) = \boldsymbol{w}^\intercal \boldsymbol{x}$$

(Notice that linear regression does not "allow" any polynomial terms of second or higher degree, thus the naming. However, what if our data is more complex than a straight line? Surprisingly, we can use a linear model to fit nonlinear data. The way to do this is to add powers of each feature as new features, and then move on with linear regression. This technique is called "polynomial regression").

Now that we have a hypothesis function, we need a rule in order to be able to find the parameters $\boldsymbol{w}$. This rule can be obtained through the probabilistic interpretation of linear regression.

More precisely, after having obtained the parameters $\boldsymbol{w}$, the hypothesis will fit the data in the best possible way but, since as we said we are dealing with probabilistic systems, we will still have some errors ϵ. In other words, for each training example the following formula will apply:

$$y^{(i)} = h(x^{(i)}) + \epsilon^{(i)} = \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + \epsilon^{(i)}$$

where $\boldsymbol{x}^{(i)}$ is the corresponding training example feature vector:

$$\boldsymbol{x}^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix}$$

At this point we will make one assumption which needs to be valid in order for the linear regression to be valid. Namely, we assume that **the errors $\epsilon^{(i)}$ are independent and identically distributed following a normal distribution with mean 0 and variance σ^2**:

$$\epsilon^{(i)} \sim N(0, \sigma^2)$$

which means that the probability distribution of the errors is given by:

$$P(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\Big(- \frac{(\epsilon^{(i)})^2}{2\sigma^2} \Big)$$

From the assumption of the errors follows:

$$y^{(i)} = \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + \epsilon^{(i)} \Rightarrow$$
$$\epsilon^{(i)} = y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)}$$

By substituting the error back to the probability:

$$P(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot exp\Big(- \frac{(\epsilon^{(i)})^2}{2\sigma^2} \Big) \Rightarrow$$

$$P(y^{(i)} | \boldsymbol{x}^{(i)}; \boldsymbol{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot exp\Big(- \frac{(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2}{2\sigma^2} \Big)$$

In other words we get that the conditional distribution of $y^{(i)}$ given $\boldsymbol{x}^{(i)}$ and $\boldsymbol{w}$ is a normal distribution with mean $\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)}$ and variance σ^2:

$$y^{(i)} \sim N(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)}, \sigma^2)$$

Given the probability distribution of $y^{(i)}$ as a function of the parameters, we can now use the principle of maximum likelihood that we developed in parametric inference chapter, in order to find the rule that will give us the best parameters $\boldsymbol{w}$.

For the likelihood we get:

$$\mathcal{L}(\boldsymbol{w}|y) = P(y^{(1)}, y^{(2)}, \ldots, y^{(m)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}; \boldsymbol{w}) = \prod_{i=1}^{m} P(y^{(i)} | \boldsymbol{x}^{(i)}; \boldsymbol{w})$$

where we used the fact that $\epsilon^{(i)}$ are independent.

By substituting the probability:

$$\mathcal{L}(\boldsymbol{w}|y) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot exp\Big(- \frac{(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2}{2\sigma^2} \Big)$$

Subsequently for the log-likelihood:

$$
\begin{aligned}
l(\boldsymbol{w}|y) &= \ln \mathcal{L}(\boldsymbol{w}|y) \\
&= \ln \Big[\prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot exp\Big(- \frac{(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2}{2\sigma^2} \Big) \Big] \\
&= \sum_{i=1}^{m} \ln \Big[\frac{1}{\sqrt{2\pi\sigma^2}} \cdot exp\Big(- \frac{(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2}{2\sigma^2} \Big) \Big] \\
&= \sum_{i=1}^{m} \ln \Big[\frac{1}{\sqrt{2\pi\sigma^2}} \Big] + \sum_{i=1}^{m} \ln \Big[exp\Big(- \frac{(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2}{2\sigma^2} \Big] \Big) \\
&= \sum_{i=1}^{m} \ln \Big[\frac{1}{\sqrt{2\pi\sigma^2}} \Big] + \sum_{i=1}^{m} \Big[- \frac{(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2}{2\sigma^2} \Big]
\end{aligned}
$$

According to the principle of maximum likelihood, the best parameters can be found by maximizing the log-likelihood. The first term of the log-likelihood is just a constant term so it does not contribute at all to the maximization, and the same holds for the denominator of the second term. Hence:

$$
\begin{aligned}
\boldsymbol{w} &= \arg\max_{\boldsymbol{w}} [l(\boldsymbol{w}|y)] \\
&= \arg\max_{\boldsymbol{w}} \Big[\sum_{i=1}^{m} (-(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2) \Big] \\
&= \arg\max_{\boldsymbol{w}} \Big[- \sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2) \Big] \\
&= \arg\min_{\boldsymbol{w}} \Big[\sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2 \Big]
\end{aligned}
$$

At this point we can formally define the following function.

Definition 2.5 (Mean Squared Error Loss Function)**.** ***Mean squared error loss function*** *(MSE) $J(\boldsymbol{w})$ is defined as:*

$$
J(\boldsymbol{w}) = \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2
$$

Hence, the principle of maximum likelihood translates to finding the parameters $\boldsymbol{w}$ that minimize the MSE loss function. The intuition behind the minimization of the MSE loss function is straight forward since what we are actually doing is minimizing the square of the errors between the prediction and the actual outcome (square because only the magnitude of the error is important and not the sign). By minimizing as much as possible the errors we will eventually get the best line that fits the data.

As a final note, we can group together all training examples in one matrix and all training labels in one vector as follows:

$$
X = \begin{bmatrix} \big(\boldsymbol{x}^{(1)}\big)^\intercal \\ \big(\boldsymbol{x}^{(2)}\big)^\intercal \\ \vdots \\ \big(\boldsymbol{x}^{(m)}\big)^\intercal \end{bmatrix} = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & \dots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \dots \\ x_0^{(m)} & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix}, \qquad \boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}
$$

By doing so then we can write the MSE loss function in the simple form of:

$$J(\boldsymbol{w}) = \frac{1}{2m}(X\boldsymbol{w} - \boldsymbol{y})^2 = \frac{1}{2m}(X\boldsymbol{w} - \boldsymbol{y})^\intercal(X\boldsymbol{w} - \boldsymbol{y})$$

Beside the MSE loss function that is derived directly through the principal of maximum likelihood, in machine learning is quite common to use some variations of MSE depending on the problem. Here we will introduce the most basic of them.

Definition 2.6 (Root Mean Squared Error Loss Function)**.** ***Root mean squared error loss function*** *(RMSE) $J(\boldsymbol{w})$ is defined as:*

$$J(\boldsymbol{w}) = \sqrt{MSE} = \sqrt{\frac{1}{2m}\sum_{i=1}^{m}(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2}$$

RMSE is a frequently used measure of the differences between values predicted by a model and the values observed, and it is probably the most easily interpreted statistic, since it has the same units as the data so it is a better measure of goodness of fit than a correlation coefficient. RMSE is simply the square root of the average of squared errors, or in other words it is the average distance of a data point from the fitted line, measured along a vertical line. The effect of each error on RMSE is proportional to the size of the squared error, thus larger errors have a disproportionately large effect on RMSE. Consequently, RMSE is sensitive to outliers.

RMSE is always non-negative, and a value of 0 (almost never achieved in practice) would indicate a perfect fit to the data. In general, a lower RMSE is better than a higher one. However, comparisons across different types of data would be invalid because the measure is dependent on the scale of the numbers used.

Definition 2.7 (Mean Bias Error)**.** ***Mean bias error loss function*** *(MBE) $J(\boldsymbol{w})$ is defined as:*

$$J(\boldsymbol{w}) = \frac{1}{2m}\sum_{i=1}^{m}(y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})$$

MBE captures the average bias in the prediction and is usually not used as a measure of the model error as high individual errors in prediction can also produce a low MBE. MBE is primarily used to estimate the average bias in the model and to decide if any steps need to be taken to correct the model bias. MBE can convey useful information, but should be interpreted cautiously because positive and negative errors will cancel out.

Definition 2.8 (Mean Absolute Error)**.** ***Mean absolute error loss function*** *(MAE) $J(\boldsymbol{w})$ is defined as:*

$$J(\boldsymbol{w}) = \frac{1}{2m}\sum_{i=1}^{m}|y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)}|$$

MAE measures the average magnitude of the errors in a set of predictions, without considering their direction. It's the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight. If the absolute value is not taken (the signs of the errors are not removed), the average error becomes the MBE.

Definition 2.9 (Mean Absolute Percentage Error)**.** ***Mean absolute percentage error loss function*** *(MAPE) $J(\boldsymbol{w})$ is defined as:*

$$J(\boldsymbol{w}) = \frac{1}{2m}\sum_{i=1}^{m}\left|\frac{y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)}}{y^{(i)}}\right|$$

MAPE (also known as mean absolute percentage deviation (MAPD)) is the mean of the absolute percentage errors of forecasts. Error is defined as actual or observed value minus the forecasted value. Percentage errors are summed without regard to sign to compute MAPE. This measure is easy to understand because it provides the error in terms of percentages. Also, because absolute percentage errors are used, the problem of positive and negative errors canceling each other out is avoided. Consequently, MAPE has managerial appeal and is a measure commonly used in forecasting. The smaller the MAPE the better the forecast.

2.2 Optimization Techniques

Given the MSE loss function (or any other loss function), the goal of machine learning is to optimize it (usually minimize it) in order to obtain the best parameters that fit the data. Optimizing loss functions is one of the biggest parts of machine learning and we can do so with the so called "optimization techniques".

Definition 2.10 (Optimization Techniques)**.** ***Optimization techniques*** *are techniques used for finding the optimum solution or unconstrained maxima or minima of continuous and differentiable functions. These are analytical methods and make use of differential calculus in locating the optimal solution.*

2.2.1 Normal Equation

Probably the most straight forward optimization technique is the so called "normal equation". Since we are looking a minimum for $J(\boldsymbol{w})$ the natural thing to do, is to simply calculate the derivative with respect to the parameter vector and then set it to zero (as we did when we introduced the principle of maximum likelihood).

It is more handy to use the vector form of MSE loss function, so for the derivative we get:

$$\begin{aligned}
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &= \frac{1}{2m}\nabla_{\boldsymbol{w}}\Big[(X\boldsymbol{w}-\boldsymbol{y})^\intercal(X\boldsymbol{w}-\boldsymbol{y})\Big] \\
&= \frac{1}{2m}\nabla_{\boldsymbol{w}}\Big[\Big((X\boldsymbol{w})^\intercal-\boldsymbol{y}^\intercal\Big)\Big(X\boldsymbol{w}-\boldsymbol{y}\Big)\Big] \\
&= \frac{1}{2m}\nabla_{\boldsymbol{w}}\Big[(X\boldsymbol{w})^\intercal(X\boldsymbol{w})-(X\boldsymbol{w})^\intercal\boldsymbol{y}-\boldsymbol{y}^\intercal(X\boldsymbol{w})+\boldsymbol{y}^\intercal\boldsymbol{y}\Big] \\
&= \frac{1}{2m}\nabla_{\boldsymbol{w}}\Big[(X\boldsymbol{w})^\intercal(X\boldsymbol{w})-2(X\boldsymbol{w})^\intercal\boldsymbol{y}+\boldsymbol{y}^\intercal\boldsymbol{y}\Big] \\
&= \frac{1}{2m}\nabla_{\boldsymbol{w}}\Big[\boldsymbol{w}^\intercal X^\intercal X\boldsymbol{w}-2\boldsymbol{w}^\intercal X^\intercal\boldsymbol{y}+\boldsymbol{y}^\intercal\boldsymbol{y}\Big] \\
&= \frac{1}{2m}\Big[2X^\intercal X\boldsymbol{w}-2X^\intercal\boldsymbol{y}\Big] \\
&= \frac{1}{m}\Big[X^\intercal X\boldsymbol{w}-X^\intercal\boldsymbol{y}\Big]
\end{aligned}$$

By setting the derivative to 0 we obtain:

$$\begin{aligned}
&\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = 0 \Rightarrow \\
&\frac{1}{m}\Big[X^\intercal X\boldsymbol{w}-X^\intercal\boldsymbol{y}\Big] = 0 \Rightarrow \\
&X^\intercal X\boldsymbol{w}-X^\intercal\boldsymbol{y} = 0 \Rightarrow \\
&X^\intercal X\boldsymbol{w} = X^\intercal\boldsymbol{y} \Rightarrow \\
&\underbrace{(X^\intercal X)^{-1}(X^\intercal X)}_{I}\boldsymbol{w} = (X^\intercal X)^{-1}X^\intercal\boldsymbol{y} \Rightarrow \\
&\boldsymbol{w} = (X^\intercal X)^{-1}X^\intercal\boldsymbol{y}
\end{aligned}$$

This final expression is called "Normal Equation", and it is an exact analytical solution that gives the parameter vector.

Despite the fact that normal equation gives an exact analytical result, computing the seemingly harmless inverse of an $((n+1)\times m)\times(m\times(n+1)) = (n+1)\times(n+1)$ matrix is, with today's most efficient computer science algorithm, of cubic time complexity (in other words, if you double the number of features, you multiply the computation time by roughly 2^3). This means that as the dimensions of X increase (mainly the number of features), the amount of operations required to compute the final result increases in a cubic trend. If X was rather small, then using the normal equation would be feasible.

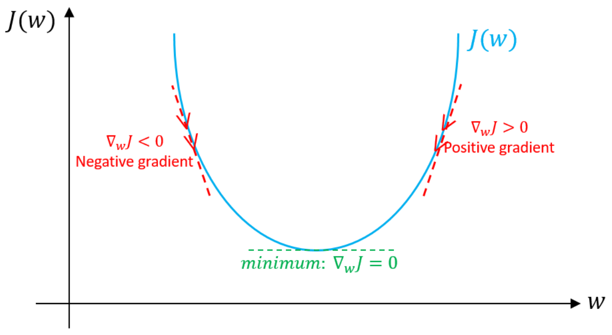
In practise, for the vast majority of any industrial application with large datasets, the normal equation would take extremely, sometimes nonsensically, long. This is the reason why normal equation is almost never used. Now let's move on the the most standard optimization technique used today called "gradient descent".

2.2.2 Gradient Descent

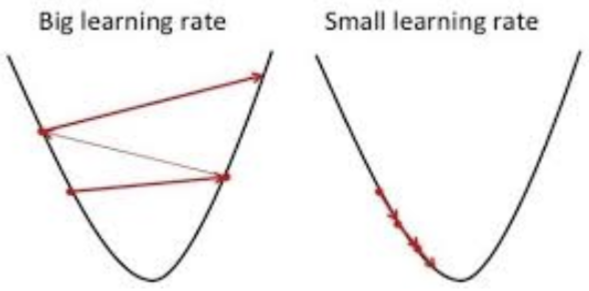
Gradient descent, and all its improvements and alternatives, is the most used optimization technique in machine learning and deep learning. It is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of gradient descent is to tweak parameters iteratively in order to minimize a cost function by starting with some random initial values for the parameters and calculating the value of the loss function based on them, and finally updating them based on the following relation:

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w})$$

Since the derivative is positive when J is upwards slopping and negative when it is downwards slopping, the minus sign makes sure that we always update the parameters towards the direction that minimizes J. Once the minimum is reached then J is at a global optimum so the derivative is 0 and further updates are not possible. At this stage the gradient descent is over and the best parameters have been found.



The hyperparameter α is called "learning rate" and defines how big or small steps we take after each iteration of gradient descent. If α is too large, we might fail to find the minimum due to oscillations around it. If α is too small then gradient descent might take too much time to reach the minimum of J. Tuning learning rate in a "right" value is a topic by itself and it is quite heavily researched today.



Of course, not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult. In general gradient descent works best with convex functions, where if you pick any two points on the curve of a convex function, the line segment joining them never crosses the curve. This implies that there are no local minima, just one global minimum. They are also continuous functions with slopes that never changes abruptly. Hence in convex functions gradient descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

Coming back to our case, let's find the update rule specifically for linear regression. Fortunately, the MSE cost function of linear regression happens to be a convex function. The only thing missing is the derivative of MSE loss function J. However in the previous chapter with normal equation we showed that:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = \frac{1}{m}\Big(X^\intercal X\boldsymbol{w} - X^\intercal \boldsymbol{y}\Big) = \frac{1}{m}X^\intercal\Big(X\boldsymbol{w} - \boldsymbol{y}\Big)$$

Hence the update rule reads:

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{m}X^\intercal\Big(X\boldsymbol{w} - \boldsymbol{y}\Big)$$

As we already mentioned there are many improvements and modified algorithms based on gradient descent philosophy. We are going to cover a lot of them in these notes. For now, let's start with 3 basics versions of gradient descent.

- **Batch Gradient Descent**

  Batch gradient descent is actually the one we just saw. As we see in gradient descent, the whole training set X is used in order to make just one update of the parameters. We usually refer to the whole training set as the "batch". For that reason, the usual terminology for what we have seen so far is batch gradient descent, meaning that the whole batch is used to update the parameter.

- **Mini-Batch Gradient Descent**

  In mini-batch gradient descent we divide the whole dataset to b subsets of $\frac{m}{b}$ training examples each, called "mini-batches", and we update the parameters using each of the mini-batches in each iteration.

- **Stochastic Gradient Descent**

  In stochastic gradient descent we only use one training example per time to update the parameters. In every iteration we pick the training example randomly hence the naming.

One of the main problems of batch gradient descent is that as the number of training examples grows the dimensions of X grows and using the whole training set for every iteration becomes computationally expensive. Both mini-batch and stochastic gradient descent deal with this problem.

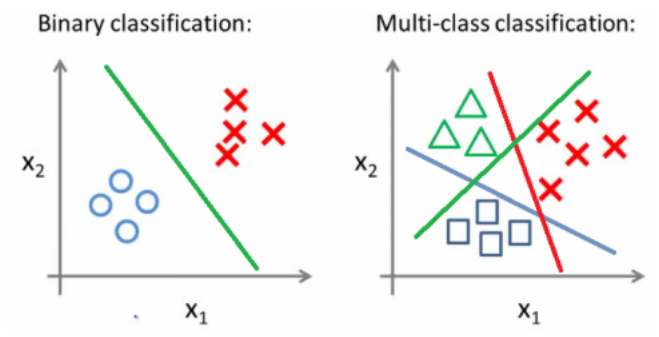
In a way, mini-batch gradient descent tries to strike a balance between the goodness of batch gradient descent and speed of stochastic gradient descent. In general, batch gradient descent works just fine so we don't need the alternative techniques we just introduced. However in more complicated models, such as deep learning models, these techniques can be really useful. For this reason we will examine again these techniques in more details in the chapter of deep learning.

2.3 Logistic Regression

Definition 2.11 (Logistic Regression)**.** ***Logistic regression*** *(or classification) is a statistical model that is used for the classification of a discrete dependent variable (target) to a specific label (class) from a set of labels (classes).*

Definition 2.12 (Binary / Multinomial Classification)**.** ***Binary*** *classification is the problem of classifying instances into one of two classes. Classifying instances into three or more classes is called* ***multinomial*** *(or multiclass) classification.*

Definition 2.13 (Multilabel Classification)**.** ***Multilabel*** *classification is a variant of the classification problem where multiple labels may be assigned to each instance. Multilabel classification is a generalization of multinomial classification, since there is no constraint on how many of the classes the instance can be assigned to.*

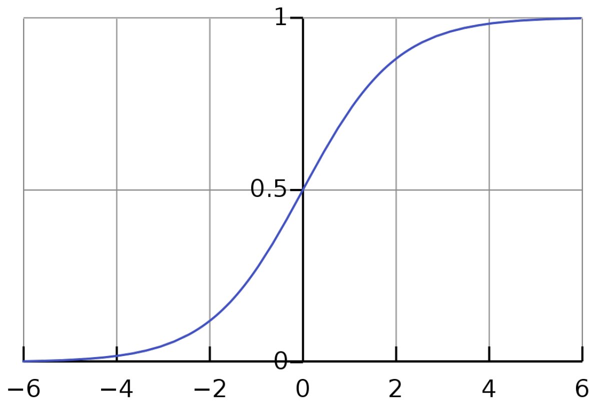


For now we will focus on binary classification, and later we will make a comment on multinomial classification.

In binary classification, since the output is binary and can take only the values 0 or 1, the hypothesis function of the linear regression is not a valid approximator for logistic regression since it produces a continuous set of outputs. So our first step is to find a suitable hypothesis function h for logistic regression. The hypothesis function that we actually use in logistic regression is the sigmoid function and it is given by:

$$h(\boldsymbol{x}) = \frac{1}{1 + exp(-\boldsymbol{w}^{\intercal}\boldsymbol{x})}$$

(From now on, in order to save space, we will write $h(\boldsymbol{x})$ instead of the actual expression for the sigmoid function).



Hence, in logistic regression, the hypothesis function computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the linear regression hypothesis function does, it outputs the logistic (sigmoid function) of this result. Notice that $h(\boldsymbol{x}) < 0.5$ when $\boldsymbol{w}^{\intercal}\boldsymbol{x} < 0$, and $h(\boldsymbol{x}) \geq 0.5$ when $\boldsymbol{w}^{\intercal}\boldsymbol{x} \geq 0$, so a logistic regression model predicts 1 if $\boldsymbol{w}^{\intercal}\boldsymbol{x}$ is positive and 0 if it is negative. (The argument of the sigmoid function is often called "logit").

It is worth mentioning that one can alter the discrimination threshold of 0.5 to any value between 0 and 1. This is quite usual in logistic regression models when obtaining the correct class is more important than a possible misclassification.

Now let's try to give a meaning to the hypothesis function. As we can see the sigmoid function produces results in the interval $[0, 1]$. It is quite close to what we need, but not exactly so, since we do not need all the values between 0 and 1. For this reason we will interpret the hypothesis function of logistic regression as a probability measure of the target to belong to class 1. The closest to 0 the hypothesis function, the more unlikely for the target to belong in class 1 (hence it belongs to class 0) and the closest to 1 the more likely to belong to the class 1. Hence, by defining a threshold (say at 0.5) the idea is that for $h(\boldsymbol{x}) < 0.5$ the algorithm will predict 0 and for $h(\boldsymbol{x}) \geq 0.5$ the algorithm will predict 1. Based on this intuition, we can write:

$$h(\boldsymbol{x}) = P(y = 1|\boldsymbol{x}; \boldsymbol{w})$$

Of course since the output must be either 0 or 1 we get:

$$
\begin{aligned}
&P(y=0|\boldsymbol{x};\boldsymbol{w}) + P(y=1|\boldsymbol{x};\boldsymbol{w}) = 1 \Rightarrow \\
&P(y=0|\boldsymbol{x};\boldsymbol{w}) = 1 - P(y=1|\boldsymbol{x};\boldsymbol{w}) \Rightarrow \\
&P(y=0|\boldsymbol{x};\boldsymbol{w}) = 1 - h(\boldsymbol{x})
\end{aligned}
$$

We can combine these two probabilities in one in the following way:

$$P(y|\boldsymbol{x};\boldsymbol{w}) = h(\boldsymbol{x})^y \cdot (1 - h(\boldsymbol{x}))^{(1-y)}$$

So in logistic regression the output follows a Bernoulli distribution with parameter $h(\boldsymbol{x})$. Now that we have a probability distribution, similarly to the linear regression, we can use the principle of the maximum likelihood in order to obtain the best parameters that maximize the likelihood. Thus, we will obtain the loss function for the logistic regression case.

By making again the assumption that we are dealing with independent and identically distributed random variables, for the likelihood is:

$$\mathcal{L}(\boldsymbol{w}|y) = P(y^{(1)}, y^{(2)}, \dots, y^{(m)}|\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots, \boldsymbol{x}^{(m)}; \boldsymbol{w}) = \prod_{i=1}^{m} P(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{w})$$

By substituting the probability:

$$\mathcal{L}(\boldsymbol{w}|y) = \prod_{i=1}^{m} h(\boldsymbol{x}^{(i)})^{y^{(i)}} \cdot (1 - h(\boldsymbol{x}^{(i)}))^{(1-y^{(i)})}$$

Subsequently for the log-likelihood:

$$
\begin{aligned}
l(\boldsymbol{w}|y) &= \ln \mathcal{L}(\boldsymbol{w}|y) \\
&= \ln \Big[\prod_{i=1}^{m} h(\boldsymbol{x}^{(i)})^{y^{(i)}} \cdot (1 - h(\boldsymbol{x}^{(i)}))^{(1-y^{(i)})} \Big] \\
&= \sum_{i=1}^{m} \ln \Big[h(\boldsymbol{x}^{(i)})^{y^{(i)}} \cdot (1 - h(\boldsymbol{x}^{(i)}))^{(1-y^{(i)})} \Big] \\
&= \sum_{i=1}^{m} \Big[\ln \Big(h(\boldsymbol{x}^{(i)})^{y^{(i)}} \Big) + \ln \Big((1 - h(\boldsymbol{x}^{(i)}))^{(1-y^{(i)})} \Big) \Big] \\
&= \sum_{i=1}^{m} \Big[y^{(i)} \cdot \ln h(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \cdot \ln(1 - h(\boldsymbol{x}^{(i)})) \Big]
\end{aligned}
$$

Once again, according to the principle of maximum likelihood, the best parameters can be found by maximizing the log-likelihood. Hence:

$$
\begin{aligned}
\boldsymbol{w} &= \arg\max_{\boldsymbol{w}} [l(\boldsymbol{w}|y)] \\
&= \arg\max_{\boldsymbol{w}} \Big[\sum_{i=1}^{m} \Big((y^{(i)} \cdot \ln h(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \cdot \ln(1 - h(\boldsymbol{x}^{(i)})) \Big) \Big] \\
&= \arg\min_{\boldsymbol{w}} \Big[- \sum_{i=1}^{m} \Big(y^{(i)} \cdot \ln h(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \cdot \ln(1 - h(\boldsymbol{x}^{(i)})) \Big) \Big]
\end{aligned}
$$

where in the last step we did the usual trick of multiplying the whole expression with a minus sign and switching the optimization problem from maximizing the quantity to minimizing it.

At this point we can formally define the following logistic regression loss function.

Definition 2.14 (Cross Entropy Loss Function)**.** ***Cross entropy loss function*** *(also called log loss) is defined as:*

$$J(\boldsymbol{w}) = -\frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)} \cdot \ln h(\boldsymbol{x}^{(i)}) + (1-y^{(i)}) \cdot \ln(1-h(\boldsymbol{x}^{(i)}))\Big)$$

The cross entropy is the loss function of the logistic regression in the similar way where MSE loss function is the loss function for linear regression. The name "cross entropy" comes from the definition of cross entropy which is the average amount of information needed to identify an event between two probability distributions p and q over the same underlying set of events

Notice that the only possible values of $y^{(i)}$ is 0 or 1. This means that in any case, one of the terms $y^{(i)}$ or $(1-y^{(i)})$ in J will vanish and the other one will be equal to 1. So in the end the only thing that is actually part of the loss is the logarithm of the hypothesis function, which given that the hypothesis function is a sigmoid function which is always between 0 and 1, the logarithm is always negative. With the overall negative sign the loss turns positive and this is what we want to minimize.

We are not going to write a vectorized form for the cross entropy loss function for two reasons. First of all, not all matrices have a logarithm and those matrices that do have a logarithm may have more than one logarithm. So one has to be careful when uses the vectorized form of logistic regression because it carries logarithms of matrices. Secondly, derivatives of logarithms of non square matrices sometimes are not defined. Since we need to calculate the derivative of J we might get problems. For this reason we will use the non vectorized form for the calculations, however we will express the final result in a vectorized form.

As in the linear case, the principle of maximum likelihood leads to the minimization of the cross entropy loss function in order to obtain the best parameters. We will examine the same techniques that we developed for the gradient descent in the linear case, i.e normal equation and gradient descent.

2.3.1 Normal Equation

As we already mentioned, since we want to minimize a function the straight forward way of doing that is to calculate the derivative and then set it to 0. However, in the case of logistic regression the normal equation does not apply since there is no closed analytical solution of $\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = 0$. The only way for solving the optimization problem is through gradient descent.

2.3.2 Gradient Descent

Gradient descent works fine in logistic regression and, as with the MSE cost function in linear regression, it also happens to be a convex function. First, let's calculate the derivative of $J(\boldsymbol{w})$ for logistic regression:

$$\begin{aligned}
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &= -\frac{1}{m}\nabla_{\boldsymbol{w}}\Big[\sum_{i=1}^{m}\Big(y^{(i)} \cdot \ln h(\boldsymbol{x}^{(i)}) + (1-y^{(i)}) \cdot \ln(1-h(\boldsymbol{x}^{(i)}))\Big)\Big] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)} \cdot \nabla_{\boldsymbol{w}} \ln h(\boldsymbol{x}^{(i)}) + (1-y^{(i)}) \cdot \nabla_{\boldsymbol{w}} \ln(1-h(\boldsymbol{x}^{(i)}))\Big) \\
&= -\frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)} \cdot \frac{\nabla_{\boldsymbol{w}} h(\boldsymbol{x}^{(i)})}{h(\boldsymbol{x}^{(i)})} - (1-y^{(i)}) \cdot \frac{\nabla_{\boldsymbol{w}} h(\boldsymbol{x}^{(i)})}{1-h(\boldsymbol{x}^{(i)})}\Big) \\
&= -\frac{1}{m}\sum_{i=1}^{m}\Big(\frac{y^{(i)}}{h(\boldsymbol{x}^{(i)})} - \frac{1-y^{(i)}}{1-h(\boldsymbol{x}^{(i)})}\Big) \cdot \nabla_{\boldsymbol{w}} h(\boldsymbol{x}^{(i)}) \\
&= -\frac{1}{m}\sum_{i=1}^{m}\Big(\frac{y^{(i)} \cdot (1-h(\boldsymbol{x}^{(i)}) - (1-y^{(i)}) \cdot h(\boldsymbol{x}^{(i)})}{h(\boldsymbol{x}^{(i)}) \cdot (1-h(\boldsymbol{x}^{(i)}))}\Big) \cdot \nabla_{\boldsymbol{w}}\Big[\frac{1}{1+exp(-\boldsymbol{w}^{\intercal}\boldsymbol{x}^{(i)})}\Big]
\end{aligned}$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\Big(\frac{y^{(i)} - y^{(i)}\cdot h(\boldsymbol{x}^{(i)}) - h(\boldsymbol{x}^{(i)}) + y^{(i)}\cdot h(\boldsymbol{x}^{(i)})}{h(\boldsymbol{x}^{(i)})\cdot(1-h(\boldsymbol{x}^{(i)}))}\Big)\cdot\Big(\frac{(-1)\cdot exp(-\boldsymbol{w}^{\intercal}\boldsymbol{x}^{(i)})\cdot(-\boldsymbol{x}^{(i)})}{(1+exp(-\boldsymbol{w}^{\intercal}\boldsymbol{x}^{(i)}))^2}\Big)$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\Big(\frac{y^{(i)} - h(\boldsymbol{x}^{(i)})}{h(\boldsymbol{x}^{(i)})\cdot(1-h(\boldsymbol{x}^{(i)}))}\Big)\cdot\Big(h(\boldsymbol{x}^{(i)})^2\cdot\frac{1-h(\boldsymbol{x}^{(i)})}{h(\boldsymbol{x}^{(i)})}\cdot\boldsymbol{x}^{(i)}\Big)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\frac{h(\boldsymbol{x}^{(i)}) - y^{(i)}}{h(\boldsymbol{x}^{(i)})\cdot(1-h(\boldsymbol{x}^{(i)}))}\cdot h(\boldsymbol{x}^{(i)})\cdot(1-h(\boldsymbol{x}^{(i)}))\cdot\boldsymbol{x}^{(i)}$$

$$= \frac{1}{m}\sum_{i=1}^{m}(h(\boldsymbol{x}^{(i)}) - y^{(i)})\cdot\boldsymbol{x}^{(i)}$$

Hence the update rule reads:

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{m}\sum_{i=1}^{m}\Big(\frac{1}{1+exp(-\boldsymbol{w}^{\intercal}\boldsymbol{x}^{(i)})} - y^{(i)}\Big)\cdot\boldsymbol{x}^{(i)}$$

Or in vectorized form:

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{m}X^{\intercal}\Big(\frac{1}{1+exp(-X\boldsymbol{w})} - \boldsymbol{y}\Big)$$

At this point, notice that gradient descent can be generalized into one model for both linear and logistic regression since the update rule for both of them can be be written in one coherent way as:

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{m}X^{\intercal}\Big(h(X) - \boldsymbol{y}\Big)$$

where one has to use either MSE loss function or cross entropy loss function depending on the regression problem.

2.3.3 Multinomial Classification

Multinomial (or multiclass) classification is the problem of classifying instances into one of three or more classes. (Multinomial classification should not be confused with multilabel classification, where multiple labels are to be predicted for each instance.) There are many techniques to deal with this problem which can be categorized into the following three categories:

- **Transformation To Binary**

  Which can be sub-categorized into the following sub-categories:

  - **One VS Rest (OvR), or One VS All (OvA), or One Against All (OAA)**

    OvR involves training a single classifier per class, with the samples of that class as positive samples and all other samples as negatives. This strategy requires the base classifiers to produce a real-valued confidence score for its decision, rather than just a class label due to the fact that discrete class labels alone can lead to ambiguities, where multiple classes are predicted for a single sample. Although this strategy is popular, it is a heuristic that suffers from several problems. Firstly, the scale of the confidence values may differ between the binary classifiers. Second, even if the class distribution is balanced in the training set, the binary classification learners see unbalanced distributions because typically the set of negatives they see is much larger than the set of positives.

  - **One VS One (OvO)**

    In OvO one trains $\frac{K(K-1)}{2}$ binary classifiers for a K-way multinomial problem, where each receives the samples of a pair of classes from the original training set, and must learn to distinguish these two classes. At prediction time, a voting scheme is applied: all $\frac{K(K-1)}{2}$ classifiers are applied to an unseen sample and the class that got the highest number of "+1" predictions gets predicted by the combined classifier. Like OvR, OvO suffers from ambiguities in that some regions of its input space may receive the same number of votes.

- **Extension From Binary**

  Extension from binary techniques extend the existing binary classifier to solve multinomial classification problems. Several algorithms have been developed based on neural networks, decision trees, k-nearest neighbors, naive Bayes, support vector machines and extreme learning machines to address multi-class classification problems.

- **Hierarchical Classification**

  Hierarchical classification tackles the multinomial classification problem by dividing the output space into a tree. Each parent node is divided into multiple child nodes and the process is continued until each child node represents only one class. Several methods have been proposed based on hierarchical classification.

As a final comment, it is worth mentioning that logistic regression itself can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers. The final result is called "softmax regression". However, since softmax regression is used heavily in deep learning we will develop its corresponding theory in the deep learning chapter.

2.4 Generalized Linear Model

As we showed, in linear regression the target follows a normal distribution while in logistic regression the target follows a Bernoulli distribution. We can generalize both regressions in one coherent model called "generalized linear model" in which the target is allowed to follow a broad family of probability distributions.

Definition 2.15 (Generalized Linear Model)**.** ***Generalized linear model*** *(GLM) is a model that allows the dependent variable to follow an exponential family of probability distributions of the form:*

$$P(y|\eta) = b(y) \cdot exp(\eta^{\intercal} T(y) - a(\eta))$$

By picking specific values for b, η, T and a we end up with different distributions (including linear and logistic regression). Then we simply assume independence and apply the principle of maximum likelihood to obtain a loss function, in order to minimize it and find the best parameters.

2.5 Errors

Since h is an estimator of f, the theory we developed in the chapter of parametric inference for estimators also holds for h. In other words, for the hypothesis function, which acts as an estimator for f, we can define quantities such as MSE (not the loss function but they coincide), sampling deviation, bias and variance. We can then use these quantities in order to evaluate how well a machine learning model performs. Let us see now the definitions of these quantities adjusted for the case of machine learning where the estimator is h.

2.5.1 Point-Wise, Overall, In-Sample & Out-Of-Sample Error

Starting from the corresponding MSE (again not the loss function), in machine learning case we define the following quantities:

Definition 2.16 (Point-Wise Error)**.** *We define the* ***point-wise error*** e *as a function of the real target function* f *and the hypothesis function* h *at point* x*:*

$$e = e(f(x), h(x))$$

For example, for linear regression we could use $e(f(x), h(x)) = (f(x) - h(x))^2$ while for logistic regression $e(f(x), h(x)) = [f(x) \neq h(x)]$. Given point-wise error we can generalize to overall error.

Definition 2.17 (Overall Error)**.** *We define the* ***overall error*** E *as the average over all point-wise errors* $e(f(x), h(x))$ *at every point* x*.*

We distinguish between two kind of overall errors: the in-sample error and the out-of-sample error.

Definition 2.18 (In-Sample Error)**.** *We define the* ***in-sample error*** E_{in} *as the average of point-wise errors of the dataset that the model was trained:*

$$E_{in} = \frac{1}{m}\sum_{i=1}^{m} e(f(x^{(i)}), h(x^{(i)}))$$

In other words in-sample error shows how well the model performs on the data used to build it.

Definition 2.19 (Out-Of-Sample Error)**.** *We define the* ***out-of-sample error*** E_{out} *as a the expected value of point-wise errors of new data:*

$$E_{out} = E_x[e(f(x), h(x))]$$

In other word out-of-sample error show how well the model generalizes to predictions for data it has not seen before. It makes sense that in order for h to work well out of sample, so it can predict, it must be $E_{out} \approx 0$.

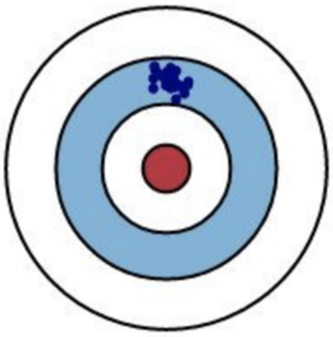
2.5.2 Bias & Variance

Back in parametric inference chapter, we introduced the bias B of an estimator $\hat{\theta}$, as the difference between the expected value of the estimator and the actual true parameter we want to estimate, $B = E[\hat{\theta}_n] - \theta$. Coming to our case where our estimator is $\hat{\theta} = h(x)$ and the true parameter is the target funtion $\theta = f(x)$, for the bias we get:

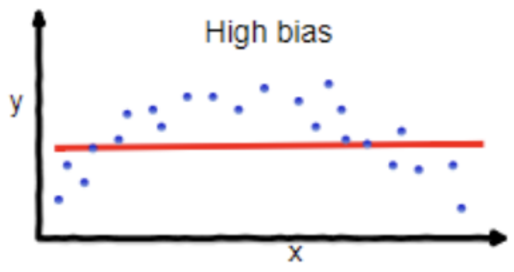
Definition 2.20 (Bias)**.** *We define the* ***bias*** B *of the hypothesis function* h *as the quantity:*

$$B = E[h(x)] - f(x)$$

The bias error is an error from erroneous assumptions in the learning algorithm. When we are dealing with high bias, formally we can say that the hypothesis set $H = \{h\}$ was not big enough in order to contain function that can approximate well the target function f. So our best approximation for f is still a bad one that cannot fit the data well.



High bias can cause an algorithm to miss the relevant relations between features and target outputs and fail to capture the underlying structure of the dataset. We call this a case of **underfitting**, since the model fails to fit the given dataset well.



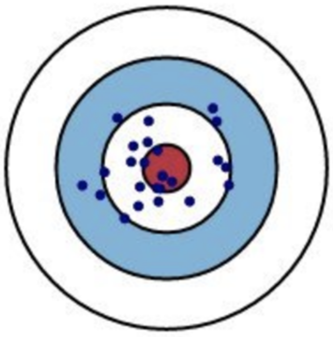
Underfitting is one of the two main problems that a machine learning model can have and it leads to a high in-sample error E_{in} which subsequently leads to a high out-of-sample error E_{out}. Hence even that the problem is coming from the in-sample error it leads to not being able to generalize for out-of-sample data.

In parametric inference chapter, we also defined the variance of an estimator $\hat{\theta}_n$ as the expected value of the square difference of the estimator from the expected value of the estimator: $Var = E[(\hat{\theta}_n - E[\hat{\theta}_n])^2]$. Coming back to machine learning for the variance we get:

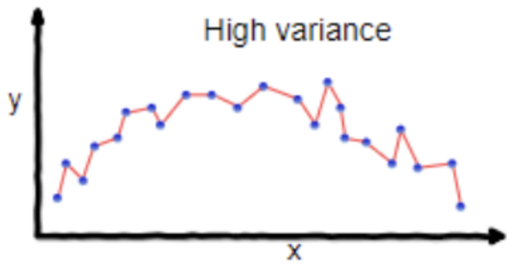
Definition 2.21 (Variance)**.** *We define the* ***variance*** *B of the hypothesis function h as the quantity:*

$$Var = E_x[(h(x) - E_x[h(x)])^2]$$

The variance is an error from sensitivity to small fluctuations in the training set.



When we are dealing with high variance, informally we can say that the hypothesis set $H = \{h\}$ is very big so in order to compensate the spread of dataset the model finds a function that fits the particular data very well but fails to generalize to new data. We call this a case of **overfitting**, since the model fails to generalize to new data.



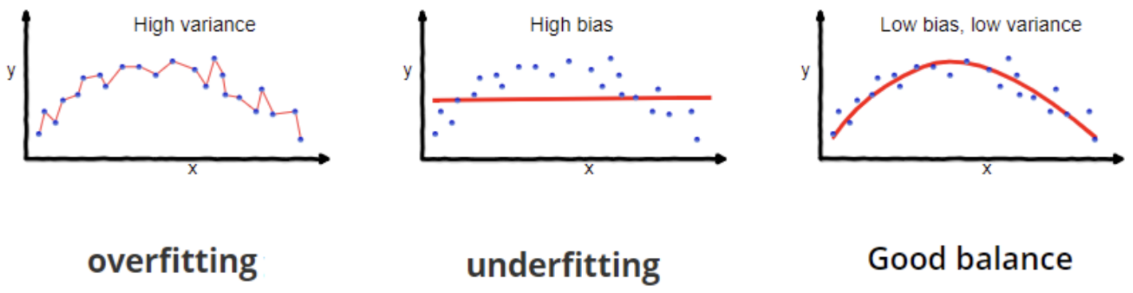
Overfitting leads to a very low in-sample $E_{in} \approx 0$, since it does a very good job on fitting the given data. However it fails to generalize, hence to predict new data, which leads to a very high out-of-sample error E_{out}.

Back in parametric inference we also showed that the mean squared error can be decomposed to bias and variance, and of course the same holds in our case since for the out of sample error of linear regression we can show:

$$\begin{aligned}
E_{out} &= E_x\Big[e(f(x), h(x))\Big] \\
&= E_x\Big[\Big(f(x) - h(x)\Big)^2\Big] \\
&= E_x\Big[\Big(f(x) - h(x) + E_x[h(x)] - E_x[h(x)]\Big)^2\Big] \\
&= E_x\Big[\Big(\Big(f(x) - E_x[h(x)]\Big) + \Big(E_x[h(x)] - h(x)\Big)\Big)^2\Big] \\
&= E_x\Big[\Big(f(x) - E_x[h(x)]\Big)^2 + 2\Big(f(x) - E_x[h(x)]\Big)\Big(E_x[h(x)] - h(x)\Big) + \Big(E_x[h(x)] - h(x)\Big)^2\Big] \\
&= E_x\Big[\Big(f(x) - E_x[h(x)]\Big)^2\Big] + E_x\Big[2\Big(f(x) - E_x[h(x)]\Big)\Big(E_x[h(x)] - h(x)\Big)\Big] + E_x\Big[\Big(E_x[h(x)] - h(x)\Big)^2\Big]
\end{aligned}$$

$$
\begin{aligned}
&= \Big(f(x) - E_x[h(x)]\Big)^2 + 2\Big(f(x) - E_x[h(x)]\Big)\Big(E_x\Big[E_x[h(x)] - h(x)\Big]\Big) + E_x\Big[\Big(E_x[h(x)] - h(x)\Big)^2\Big] \\
&= \Big(f(x) - E_x[h(x)]\Big)^2 + 2\Big(f(x) - E_x[h(x)]\Big)\Big(E_x[h(x)] - E_x[h(x)]\Big) + E_x\Big[\Big(E_x[h(x)] - h(x)\Big)^2\Big] \\
&= \Big(f(x) - E_x[h(x)]\Big)^2 + E_x\Big[\Big(E_x[h(x)] - h(x)\Big)^2\Big] \\
&= B^2 + Var
\end{aligned}
$$

Hence, the out-of-sample error is actually a combination of bias and variance. So in order to have a model that generalizes well, we have to keep both of them low. However, since they are of opposite nature, the more we reduce bias the more variance increases and vice versa. This is the so called **bias-variance trade off**. The goal in machine learning is to balance this trade off so the model fits the data well and doesn't fail to generalize.



Let's sum up. In the graph above we see that in the case of high bias (underfitting) we have restricted our model to linear predictors, however the data do not follow a linear trend, hence the hypothesis set is too small and the model cannot find a good curve to fit the data. On the other hand, in the case of high variance (overfitting) the hypothesis set is so big allowing complex predictors so the model managed to find a high degree polynomial that fit the data really good, however it will fail to generalize since it depends a lot on the initial values of the data and it is sensitive to fluctuations of them. Finally in the last graph we have a good balance of bias and variance and the model found a good curve. One has to keep in mind that increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a "trade-off".

It is worth mentioning that we have neglected the so-called "irreducible error" for the equations. This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g fix the data sources, such as broken sensors, or detect and remove outliers).

2.6 Evaluation

Evaluation is about how good a model generalizes to new data. After applying the learning algorithm to the data and having obtained a hypothesis h, the machine learning model is ready to make new predictions. However before that, we have to evaluate the model by analysing the errors that we just introduced.

The starting part is the in-sample and out-of-sample errors that we defined previously:

$$
E_{in} = \frac{1}{m}\sum_{i=1}^{m} e(f(x^{(i)}), h(x^{(i)})) \qquad \text{and} \qquad E_{out} = E_x[e(f(x), h(x))]
$$

In general, for the error function e we use the corresponding loss function J that we used to train the model (although some times we can use variations of it), since it is a function of the target and hypothesis functions as e, and it is a really good measure of error:

$$E_{in} = \frac{1}{m}\sum_{i=1}^{m} J^{(i)}(f(x^{(i)}), h(x^{(i)})) \qquad \text{and} \qquad E_{out} = E_x[J(f(x), h(x))]$$

where here the notation $J^{(i)}$ means the error coming from the i^{th} training example. Hence, now E_{in} is calculated with the data that we trained the model, so it's a very good measure of how well the model performs in the data that it was trained on. The problem comes from E_{out} since we don't know how to compute this expected value. Unsurprising we will perform the usual trick of substituting the expected value with the average so:

$$E_{in} = \frac{1}{m}\sum_{i=1}^{m} J^{(i)}(f(x^{(i)}), h(x^{(i)})) \qquad \text{and} \qquad E_{out} = \frac{1}{m}\sum_{i=1}^{m} J^{(i)}(f(x^{(i)}), h(x^{(i)}))$$

Of course since we estimate the expected value with an average that brings an error to the estimation of the out-of-sample error. However for our purposes we assume that this error is neglectful, and from now on we will treat the estimated out-of-sample error as the actual out-of-sample error. In general we have to keep in mind though that out-of-sample error carries an error.

The question that arises is what data are we going to use for E_{out}. Using the same data that we trained the model is a really bad idea since, first of all, we will simply get $E_{out} = E_{in}$ and secondly the model already knows the correct answers of the data since we used them to train it, and the evaluation will be biased.

In order to overcome this problem, we split the dataset (before training the model) into two parts: training set and test set. Then we use the first to train the model and obtain E_{in} and the latter to evaluate its performance and obtain E_{out}. Since the model is trained with the train set, it has never seen the test set so the estimation of the out-of-sample error with the evaluation set will be unbiased.

One of the things to consider is the proportions of splitting the dataset into training and test sets. This again is an area of heavy research, but in general in machine learning we usually split them in a proportion of "80% - 20%". In other areas of machine learning like deep learning where we usually have a very large amount of data we use splitting rules of "99% - 1%". But we will address this issue in details in deep learning chapter.

Hence, before training we split the dataset as:

- Training Dataset: $\{x^{(i)}_{\text{train}}, y^{(i)}_{\text{train}}\}, \quad i = 1, 2, \dots, m_{\text{train}}$ (80% of initial dataset).
- Test Dataset: $\{x^{(i)}_{\text{test}}, y^{(i)}_{\text{test}}\}, \quad i = 1, 2, \dots, m_{\text{test}}$ (20% of initial dataset).

And subsequently the errors become:

$$E_{in} = \frac{1}{m_{\text{train}}}\sum_{i=1}^{m_{\text{train}}} J^{(i)}(f(x^{(i)}), h(x^{(i)})) \qquad \text{and} \qquad E_{out} = \frac{1}{m_{\text{test}}}\sum_{i=1}^{m_{\text{test}}} J^{(i)}(f(x^{(i)}), h(x^{(i)}))$$

From now on we will be referring to the first expression as $J_{\text{train}} = E_{in}$ and to the second one as $J_{\text{test}} = E_{out}$.

So for example for linear regression we would have:

$$J_{\text{train}} = \frac{1}{2m_{train}}\sum_{i=1}^{m_{train}} (y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2 \qquad \text{and} \qquad J_{\text{test}} = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}} (y^{(i)} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)})^2$$

While for logistic regression we would have:

$$J_{\text{train}} = -\frac{1}{m_{train}}\sum_{i=1}^{m_{train}} \Big(y^{(i)} \cdot \ln h(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \cdot \ln(1 - h(\boldsymbol{x}^{(i)}))\Big)$$

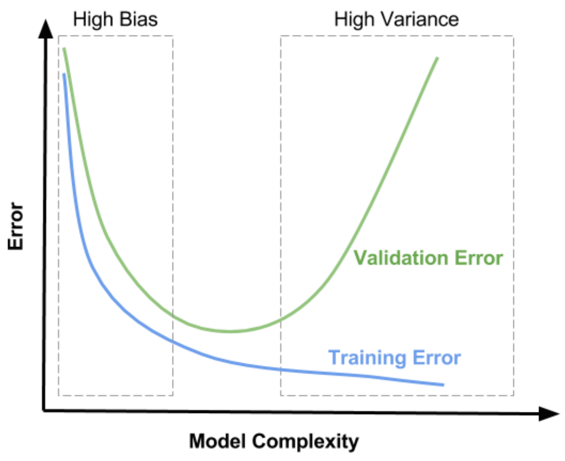
and:

$$J_{\text{test}} = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \left(y^{(i)} \cdot \ln h(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \cdot \ln(1 - h(\boldsymbol{x}^{(i)})) \right)$$

Coming to a more applied side of machine learning, it is worth mentioning that it is very important to not use the test dataset until the very end when the final model is ready. The test set is actually something we use in order to provide it as a performance measure together with the model and not something to use for gaining insights or improving the model. For the later, what people usually do is to further split the train set to a "new" (even smaller) train set, and a so called "evaluation set" (using again a proportion of "80% - 20%") and then the use the new smaller train set to train the model and the evaluation set in order to improve the model (e.g tune the parameters, find best features, etc). There are many best practices on how to use the evaluation set (e.g K-fold cross validation, grid search, etc) however these techniques are of an applied nature and they get a bit out of topic for these notes.

Coming back to the theoretical aspect of machine learning, now that we have J_{train} and J_{eval} we know how well the models performs in and out of sample. However we can also use them in order to find if the model suffers from underfitting or overfitting. There are two ways that we can do so.

The first way, is by gradually increasing the complexity of the model (higher polynomial degrees so bigger hypothesis set), training the model for each complexity level and calculate both J_{train} and J_{eval} for each model. Then by plotting out the different values for different levels of complexity we usually end up with the following graph:



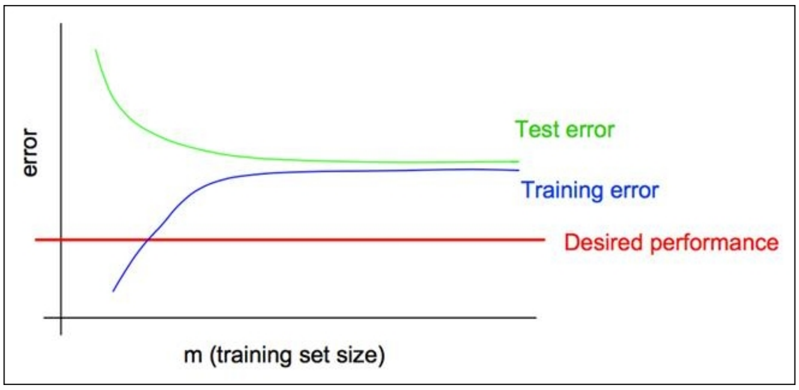
In the high bias area both J_{train} and J_{eval} are high. This means that the model does not fit the training data well hence it fails to generalize. This is the case of underfitting. In the high variance area, J_{train} is low but J_{eval} is high. This means that the model fits the training data well but fails to generalize to unseen data. This is the case of overfitting. Hence by using the graph we can diagnose both cases!

The second way to diagnose the problem of the model, is through the so called "learning curves".

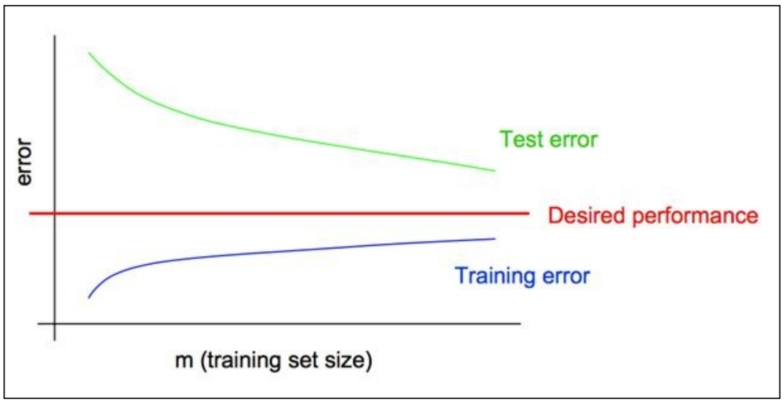
Definition 2.22 (Learning Curve)**.** *A **learning curve** is a graphical representation of how an increase in learning comes from greater experience; or how the more someone performs a task, the better they get at it.*

Informally, a learning curve is the relation between error (as expressed in loss function) and training examples m. By plotting this relation for both J_{train} and J_{eval} we end up with two learning curves and by their relative position we can diagnose if our model suffers from high bias or high variance.

More specifically, when the learning curves of J_{train} and J_{eval} do not have a large gap between them as m increases we are usually dealing we a case of high bias and underfitting.



On the other hand when there is a large gap between the two curves we are dealing with the case of high variance and overfitting.



Once we detect the problem then we have to make some changes in order to fix them! Here are some of the techniques that we follow:

- **For high bias (underfitting)**
  - Increase model complexity.
  - Increase number of features.
- **For high variance (overfitting)**
  - Decrease model complexity.
  - Decrease number of features.
  - Find more training examples.
  - Regularization (next section).

2.7 Regularization

As we saw in the previous section, when we allow a very broad hypothesis set with many higher order terms the model might find a hypothesis function h that gives a 0 in-sample error but fails to generalize. This is due to high variance, i.e large dependence on the very specific dataset used for training, and we call this a case of overfitting. A way to deal with overfitting is a collection of techniques that undergo with the name “regularization”.

Definition 2.23 (Regularization)**.** ***Regularization*** *is the process of adding information in order to solve an ill-posed problem and to prevent overfitting.*

There are many different regularization techniques. We will begin with four main ones called “Ridge regression” (or “Tikhonov regression” or “L2 regularization”), “Lasso regression ” (or “L1 regularization”), “elastic net”, and “early stopping”.

2.7.1 Ridge Regression / Tikhonov Regularization (L2 Regularization)

The reason of overfitting is that the parameters $\boldsymbol{w}$ are free to get any value. With regularization we penalize the parameters by imposing an extra constraint on $\boldsymbol{w}$ of the form:

$$\boldsymbol{w}^\intercal \boldsymbol{w} \leq C$$

where C is a constant defined by us and it controls the effect of regularization. It is called "L2 regularization" because the quantity $\boldsymbol{w}^\intercal \boldsymbol{w}$ is actually the squared L2 norm of the vector $\boldsymbol{w}$:

$$||\boldsymbol{w}||_2^2 = \boldsymbol{w}^\intercal \boldsymbol{w}$$

Hence now the optimization problem becomes to minimize the loss function $J(\boldsymbol{w})$ subject to the above mentioned constraint. According to Appendix A in order to do so we define the Lagrangian:

$$\mathcal{L}(\boldsymbol{w}) = J(\boldsymbol{w}) + \frac{\lambda}{2m} \boldsymbol{w}^\intercal \boldsymbol{w}$$

where λ is the Lagrange multiplier, and then we solve the equation:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = 0$$

For example, for linear regression where $J(\boldsymbol{w})$ is the MSE loss function, the Lagrangian reads:

$$\mathcal{L}(\boldsymbol{w}) = J(\boldsymbol{w}) + \frac{\lambda}{2m} \boldsymbol{w}^\intercal \boldsymbol{w} = \frac{1}{2m}(X\boldsymbol{w} - \boldsymbol{y})^\intercal (X\boldsymbol{w} - \boldsymbol{y}) + \frac{\lambda}{2m} \boldsymbol{w}^\intercal \boldsymbol{w}$$

At this point we can redefine this Lagrangian as a new loss function of the form:

$$J(\boldsymbol{w}) = \frac{1}{2m}\Big[(X\boldsymbol{w} - \boldsymbol{y})^\intercal (X\boldsymbol{w} - \boldsymbol{y}) + \lambda \boldsymbol{w}^\intercal \boldsymbol{w}\Big]$$

and then the problem is to minimize this loss function which is actually a regression problem. The corresponding regression is called "Ridge regression" (or "Tikhonov regularization"), where the only difference with linear regression is that we have to add the extra term in the loss function to reduce overfitting.

The coefficient λ is the one that controls the regularization effect on the regression. In one extreme where $\lambda = 0$ the regularization term vanishes, and the loss function ends up to the mean squared error loss function, hence the ridge regression turns to linear regression. In the other extreme where $\lambda \to \infty$ then the regularization term penaltizes all parameters in an extreme way, so the ridge regression, in order to minimize the loss, is forced to set all the parameters to 0. In the end we end up with $\boldsymbol{w}^\intercal \boldsymbol{x} = 0$. For all intermediate values of λ we get different levels of regularization. It is actually our job to tune the model to the right λ that does the job.

Now that we have a loss function, we treat the problem in the similar way as we did before. For example, in the linear case of ridge regression we can solve the normal equation in the same way we solved it before:

$$\begin{aligned}
J(\boldsymbol{w}) &= \frac{1}{2m}\Big[(X\boldsymbol{w} - \boldsymbol{y})^\intercal (X\boldsymbol{w} - \boldsymbol{y}) + \lambda \boldsymbol{w}^\intercal \boldsymbol{w}\Big] \\
&= \frac{1}{2m}\Big[\Big((X\boldsymbol{w})^\intercal - \boldsymbol{y}^\intercal\Big)\Big(X\boldsymbol{w} - \boldsymbol{y}\Big) + \lambda \boldsymbol{w}^\intercal \boldsymbol{w}\Big] \\
&= \frac{1}{2m}\Big[(X\boldsymbol{w})^\intercal (X\boldsymbol{w}) - (X\boldsymbol{w})^\intercal \boldsymbol{y} - \boldsymbol{y}^\intercal (X\boldsymbol{w}) + \boldsymbol{y}^\intercal \boldsymbol{y} + \lambda \boldsymbol{w}^\intercal \boldsymbol{w}\Big] \\
&= \frac{1}{2m}\Big[(X\boldsymbol{w})^\intercal (X\boldsymbol{w}) - 2(X\boldsymbol{w})^\intercal \boldsymbol{y} + \boldsymbol{y}^\intercal \boldsymbol{y} + \lambda \boldsymbol{w}^\intercal \boldsymbol{w}\Big] \\
&= \frac{1}{2m}\Big[\boldsymbol{w}^\intercal X^\intercal X\boldsymbol{w} - 2\boldsymbol{w}^\intercal X^\intercal \boldsymbol{y} + \boldsymbol{y}^\intercal \boldsymbol{y} + \lambda \boldsymbol{w}^\intercal \boldsymbol{w}\Big]
\end{aligned}$$

By setting the derivative to 0 we obtain:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = 0 \Rightarrow$$

$$\frac{1}{2m}\nabla_{\boldsymbol{w}}\Big[\boldsymbol{w}^\intercal X^\intercal X\boldsymbol{w} - 2\boldsymbol{w}^\intercal X^\intercal \boldsymbol{y} + \boldsymbol{y}^\intercal \boldsymbol{y} + \lambda \boldsymbol{w}^\intercal \boldsymbol{w}\Big]$$

$$\frac{1}{2m}\Big[2X^\intercal X\boldsymbol{w} - 2X^\intercal \boldsymbol{y} + 2\lambda\boldsymbol{w}\Big] = 0 \Rightarrow$$

$$\frac{1}{m}\Big[X^\intercal X\boldsymbol{w} - X^\intercal \boldsymbol{y} + \lambda\boldsymbol{w}\Big] = 0 \Rightarrow$$

$$X^\intercal X\boldsymbol{w} - X^\intercal \boldsymbol{y} + \lambda\boldsymbol{w} = 0 \Rightarrow$$

$$(X^\intercal X + \lambda I)\boldsymbol{w} = X^\intercal \boldsymbol{y} \Rightarrow$$

$$\underbrace{(X^\intercal X + \lambda I)^{-1}(X^\intercal X + \lambda I)}_{I}\boldsymbol{w} = (X^\intercal X + \lambda I)^{-1}X^\intercal \boldsymbol{y} \Rightarrow$$

$$\boldsymbol{w} = (X^\intercal X + \lambda I)^{-1}X^\intercal \boldsymbol{y}$$

The only difference with the normal equation of linear regression is the extra term λI coming from regularization.

Gradient descent also works for ridge regression. For the derivative of J:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = \frac{1}{m}\Big(X^\intercal X\boldsymbol{w} - X^\intercal \boldsymbol{y} + \lambda\boldsymbol{w}\Big) = \frac{1}{m}X^\intercal\Big((X + \lambda I)\boldsymbol{w} - \boldsymbol{y}\Big)$$

Hence the update rule reads:

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{m}X^\intercal\Big((X + \lambda I)\boldsymbol{w} - \boldsymbol{y}\Big)$$

Of course, L2 regularization can be applied also for the case of logistic regression. More specifically, for cross entropy loss function of logistic regression the Lagrangian reads:

$$\mathcal{L}(\boldsymbol{w}) = J(\boldsymbol{w}) + \frac{\lambda}{2m}\boldsymbol{w}^\intercal\boldsymbol{w} = -\frac{1}{m}\Big(\boldsymbol{y}^\intercal \cdot \ln h(X) + (I - \boldsymbol{y})^\intercal \cdot \ln(I - h(X))\Big) + \frac{\lambda}{2m}\boldsymbol{w}^\intercal\boldsymbol{w}$$

Similarly to the linear case, we redefine this Lagrangian as a new loss function of the form:

$$J(\boldsymbol{w}) = -\frac{1}{m}\Big[\boldsymbol{y}^\intercal \cdot \ln h(X) + (I - \boldsymbol{y})^\intercal \cdot \ln(I - h(X)) - \frac{\lambda}{2}\boldsymbol{w}^\intercal\boldsymbol{w}\Big]$$

As we said back in logistic regression, normal equation does not apply here since there is no closed analytical solution, however gradient descent still applies where the rule simply reads:

$$\boldsymbol{w} := \Big(1 - \frac{\alpha\lambda}{m}\Big)\boldsymbol{w} - \frac{\alpha}{m}X^\intercal\Big(\frac{1}{1 + exp(-X\boldsymbol{w})} - \boldsymbol{y}\Big)$$

In both cases solving ridge regression will give us as a result a solution that slightly underfits the data, compared to linear or logistic regression. This underfitting will produce higher bias hence, due to bias-variance trade off, a reduced variance which will lead to the reduction of overfitting.

2.7.2 Lasso Regression (L1 Regularization)

In ridge regression we used the L2 norm of the vector $\boldsymbol{w}$. Another way of regularization is to use L1 norm which is:

$$||\boldsymbol{w}||_1 \leq C$$

By repeating the same way of analysis as in ridge regression, we can define the following loss function for linear regression:

$$J(\boldsymbol{w}) = \frac{1}{2m}\Big[(X\boldsymbol{w}-\boldsymbol{y})^\intercal(X\boldsymbol{w}-\boldsymbol{y}) + \lambda||\boldsymbol{w}||_1\Big]$$

and for logistic regression:

$$J(\boldsymbol{w}) = -\frac{1}{m}\Big[\boldsymbol{y}^\intercal \cdot \ln h(X) + (I-\boldsymbol{y})^\intercal \cdot \ln(I-h(X)) - \frac{\lambda}{2}||\boldsymbol{w}||_1\Big]$$

The corresponding regression is called "Least Absolute Shrinkage and Selection Operator Regression", usually simply called "Lasso regression". An important characteristic of Lasso regression is that it tends to eliminate the weights of the least important features (i.e set them to zero). In other words, Lasso regression automatically performs feature selection and outputs a sparse model (i.e with few nonzero feature weights).

As before, we can use normal equation and gradient descent to solve Lasso regression.

2.7.3 Elastic Net

Elastic net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio between them with a parameter r.

The elastic net loss function for linear regression reads:

$$J(\boldsymbol{w}) = \frac{1}{2m}\Big[(X\boldsymbol{w}-\boldsymbol{y})^\intercal(X\boldsymbol{w}-\boldsymbol{y}) + r\lambda||\boldsymbol{w}||_1 + (1-r)\lambda\boldsymbol{w}^\intercal\boldsymbol{w}\Big]$$

and for logistic regression:

$$J(\boldsymbol{w}) = -\frac{1}{m}\Big[\boldsymbol{y}^\intercal \cdot \ln h(X) + (I-\boldsymbol{y})^\intercal \cdot \ln(I-h(X)) - r\frac{\lambda}{2}||\boldsymbol{w}||_1 - (1-r)\frac{\lambda}{2}\boldsymbol{w}^\intercal\boldsymbol{w}\Big]$$

When r = 0, Elastic Net is equivalent to Ridge Regression, and when r = 1, it is equivalent to Lasso Regression.

Since elastic net summarizes both ridge and lasso regression, it is a good point to analyse, when one should use plain linear (or logistic) regression (without any regularization), ridge, lasso, or elastic net? It is almost always preferable to have at least a little bit of regularization, so generally plain linear regression should be avoided. Ridge is a good default, but if one suspects that only a few features are useful, they should prefer lasso or elastic net because they tend to reduce the useless features' weights down to zero, as we have discussed. In general, elastic net is preferred over lasso because lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

2.7.4 Early Stopping

A very different way to regularize iterative learning algorithms such as gradient descent is to stop training as soon as the validation error reaches a minimum. This is called "early stopping". In general as the algorithm learns, its prediction error on the training set goes down, along with its prediction error on the validation set. After a while though, the validation error stops decreasing and starts to go back up. This indicates that the model has started to overfit the training data. Early stopping just stops training as soon as the validation error reaches the minimum. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a "beautiful free lunch".

2.8 Classification Error Metrics

In classification problems where both input and output can be either 0 or 1, we can follow a different approach of error evaluation based on exact matches and mismatches between prediction and actual result. The usual case, since we are dealing with a binary output, is to define either 0 or 1 as the positive class and the remaining as the negative one. Which one is which depends on the nature of the problem. For now we will stick with the case were 0 represents the negative class and 1 the positive one.

Given that both the actual class and the predicted class can be either positive or negative we end up with 4 different, distinct situations. Let us define them formally.

Definition 2.24 (True Positive)**.** ***True positive*** *(TP) also called* ***hit****, is the case where the model predicts a positive result when the actual outcome is indeed positive.*

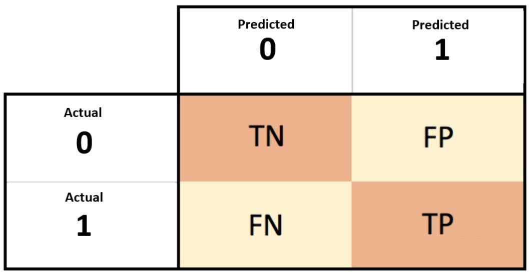
Definition 2.25 (True Negative)**.** ***True negative*** *(TN) also called* ***correct rejection****, is the case where the model predicts a negative result when the actual outcome is indeed negative.*

Definition 2.26 (False Positive)**.** ***False positive*** *(FP) also called* ***false alarm*** *or* ***type I error****, is the case where the model predicts a positive result when the actual outcome is negative.*

Definition 2.27 (False Negative)**.** ***False negative*** *(FN) also called* ***miss*** *or* ***type II error****, is the case where the model predicts a negative result when the actual outcome is positive.*

Once the model is trained, we test it on the evaluation set and we measure the number of occurrences of each category. Then we gather them all together to the so called "confusion matrix".

Definition 2.28 (Confusion Matrix)**.** ***Confusion matrix*** *is a table that reports the number of true positives TP, true negatives TN, false positives FP and false negatives FN of a model.*



	Predicted **0**	Predicted **1**
Actual **0**	TN	FP
Actual **1**	FN	TP

Once we have constructed the confusion matrix we can define the following error metrics:

Definition 2.29 (Accuracy)**.** ***Accuracy*** *(ACC) is the rate that shows overall how often the model was correct:*

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

Definition 2.30 (Error Rate)**.** ***Error rate*** *(ERR) also called* ***misclassification****, is the rate that shows overall how often the model was incorrect:*

$$ERR = \frac{FP + FN}{TP + TN + FP + FN}$$

It is of course: $ACC + ERR = 1$.

Definition 2.31 (True Positive Rate)**.** ***True positive rate*** *(TPR) also called* ***sensitivity*** *or* ***recall*** *or* ***hit rate****, is the rate that shows how often the model predicts positive when the actual outcome is indeed positive:*

$$TPR = \frac{TP}{TP + FN}$$

Definition 2.32 (False Negative Rate)**.** ***False negative rate*** *(FNR) also called* ***miss rate****, is the rate that shows how often the model predicts negative when the actual outcome is positive:*

$$FNR = \frac{FN}{TP + FN}$$

It is of course: $TPR + FNR = 1$.

Definition 2.33 (True Negative Rate)**.** ***True negative rate*** *(TNR) also called* ***specificity*** *or* ***selectivity,*** *is the rate that shows how often the model predicts negative when the actual outcome is indeed negative:*

$$TNR = \frac{TN}{TN + FP}$$

Definition 2.34 (False Positive Rate)**.** ***False positive rate*** *(FPR) also called* ***fall-out rate****, is the rate that shows how often the model predicts positive when the actual outcome is negative:*

$$FPR = \frac{FP}{TN + FP}$$

It is of course: $TNR + FPR = 1$.

Definition 2.35 (Positive Predicted Value)**.** ***Positive predicted value*** *(PPV) also called* ***precision****, is the rate that shows how often the model is correct when it predicts positive:*

$$PPV = \frac{TP}{TP + FP}$$

Definition 2.36 (False Discovery Rate)**.** ***False discovery rate*** *(FDR) is the rate that shows how often the model is wrong when it predicts positive:*

$$FDR = \frac{FP}{TP + FP}$$

It is of course: $PPV + FDR = 1$.

Definition 2.37 (Negative Predicted Value)**.** ***Negative predicted value*** *(NPV) is the rate that shows how often the model is correct when it predicts negative:*

$$NPV = \frac{TN}{TN + FN}$$

Definition 2.38 (False Omission Rate)**.** ***False omission rate*** *(FOR) is the rate that shows how often the model is wrong when it predicts negative:*

$$FOR = \frac{FN}{TN + FN}$$

It is of course: $NPV + FOR = 1$.

Definition 2.39 (F_β Score)**.** *F_β* ***score*** *(FOR) is defined as the harmonic mean of positive predicted value PPV (aka precision) and true positive rate (aka recall) each weighted based on value of β:*

$$F_\beta = \frac{(1+\beta^2) \cdot PPV \cdot TPR}{\beta^2 \cdot PPV + TPR} = \frac{(1+\beta^2) \cdot \mathit{precission} \cdot \mathit{recall}}{\beta^2 \cdot \mathit{precission} + \mathit{recall}} = \frac{(1+\beta^2) \cdot TP}{(1+\beta^2) \cdot TP + \beta^2 \cdot FN + FP}$$

The coefficient β is chosen such that recall is considered β times as important as precision. Two commonly used values for β are 2 and 0.5, corresponding to the F_2 where weighs recall higher than precision (by placing more emphasis on false negatives) and the $F_{0.5}$ measure, which weighs recall lower than precision (by attenuating the influence of false negatives). However, the most commonly used value for β is 1, corresponding to the F_1 where precision and recall are weighted equally:

$$F_1 = \frac{2 \cdot PPV \cdot TPR}{PPV + TPR} = \frac{2 \cdot \text{precission} \cdot \text{recall}}{\text{precission} + \text{recall}} = \frac{2 \cdot TP}{2 \cdot TP + FN + FP}$$

It is clear now that there are a lot of different error metrics for classification. One good question is “which ones should I use?”. What follows is a practical guide on which ones are the most important and how to use them.

2.8.1 Accuracy & Precision/Recall Trade-Off

Accuracy is quire straight forward and easy to understand evaluation metric. It just says how accurate the classifier is. However, one has to be extra careful with using accuracy as an evaluation metric. More often than not the dataset is skewed which results to the positive label (the one we want to predict) appearing in just a small portion of the dataset (say 5%). This means that an algorithm predicting always negative would be 95% accurate. What accuracy lacks is capturing the importance of not finding the positive ones.

What people usually pay attention in classification errors is the accuracy of the positive predictions, i.e the **precision** of the classifier. In simple words precision tells us how many times the algorithm was correct when it predicted positive. A trivial way to have perfect precision is to make one single positive prediction and ensure it is correct. But this would not be very useful, since the classifier would ignore all but one positive instance. So precision is typically used along **recall**. Recall simply states what percentage of the positive labels in the dataset was predicted correctly from the algorithm. Precision and recall are usually the most useful metrics to evaluate a classifier.

Depending on the nature of the problem sometimes one mostly cares about precision, and in other contexts really care about recall. For example, if one trains a classifier to detect videos that are safe for kids, they would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up. On the other hand, suppose one trains a classifier to detect shoplifters in surveillance images: it is probably fine if their classifier has only 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught). Unfortunately, one can’t have it both ways: increasing precision reduces recall, and vice versa. This is called the “precision/recall trade-off”.

Subsequently F_1 **score** that combines precision and recall is also quite useful, in particular if one needs a simple way to compare two classifiers. The F_1 score is the harmonic mean of precision and recall and, unfortunately, it does not have a straight forward intuitive meaning. The way to interpret it is to keep in mind that the harmonic mean gives much more weight to low values. and as a result, a classifier will only get a high F_1 score if both recall and precision are high.

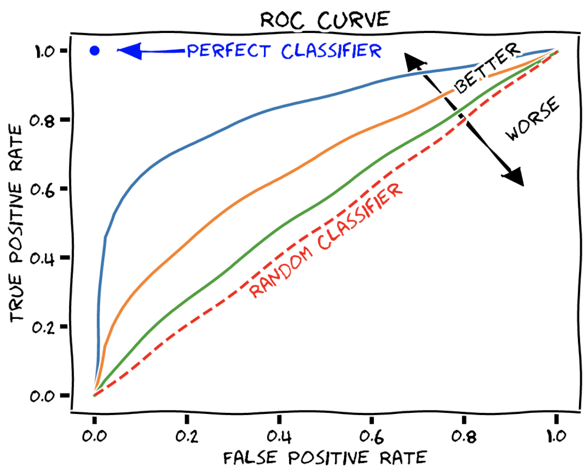
2.8.2 Receiver Operating Characteristic (ROC) Curve

Another very widely used technique in classification evaluation is the so called “receiver operating characteristic curve” (ROC) which is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. In other words, each point in the curve represents a different value of the discrimination threshold (from 0 to 1), hence a completely different behaviour of the model which leads to a different confusion matrix and subsequently different error metrics for each value. The method was originally developed for operators of military radar receivers, which is why it is so named.

The ROC curve is created by plotting the true positive rate (TPR or sensitivity or recall) against the false positive rate (FPR or 1 - specificity). It can also be thought of as a plot of the power as a function of the Type I Error of the decision rule (when the performance is calculated from just a sample of the population, it can be thought of as estimators of these quantities). The ROC curve is thus the sensitivity or recall as a function of fall-out.

ROC analysis provides tools to select possibly optimal models and to discard suboptimal ones independently from (and prior to specifying) the cost context or the class distribution. ROC analysis is related in a direct and natural way to cost/benefit analysis of diagnostic decision making.

When using normalized units, the area under the curve (often referred to as simply the AUC) is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one (assuming ’positive’ ranks higher than ’negative’).



The machine learning community most often uses the ROC AUC statistic for model comparison. In the figure for example, the blue line is a better classifier than the orange and the green, because it has a larger are under the curve. Subsequently the orange is better than the green. This practice however has been questioned because AUC estimates are quite noisy and suffer from other problems. Nonetheless, the coherence of AUC as a measure of aggregated classification performance has been vindicated, in terms of a uniform rate distribution, and AUC has been linked to a number of other performance metrics such as the Brier score. Another problem with ROC AUC is that reducing the ROC curve to a single number ignores the fact that it is about the tradeoffs between the different systems or performance points plotted and not the performance of an individual system, as well as ignoring the possibility of concavity repair.

2.9 Support Vector Machine

Support vector machine (SVM) is another, more advanced supervised learning algorithm. It applies mainly in classification problems however there is also another model called support vector regression that applies the same ideas in regression problems. Here we will explore only SVM.

To tell the SVM story, we'll need to first talk about margins and the idea of separating data with a large "gap". Next, we'll talk about the optimal margin classifier, which will lead us into a digression on Lagrange duality. We'll also see kernels, which give a way to apply SVM's efficiently in very high dimensional (such as infinite dimensional) feature spaces, and finally, we'll close off the story with the SMO algorithm, which gives an efficient implementation of SVM's.

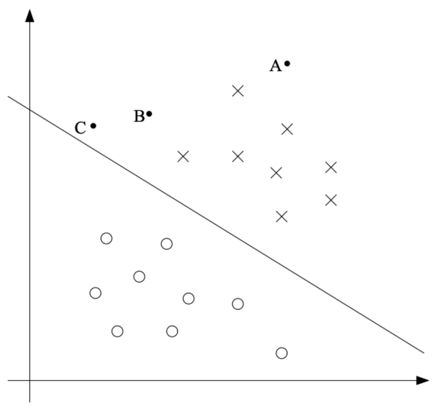
Consider logistic regression, where the probability $P(y = 1|\boldsymbol{x}; \boldsymbol{w})$ is modelled by:

$$h(\boldsymbol{w}^\intercal \boldsymbol{x}) = \frac{1}{1 + exp(-\boldsymbol{w}^\intercal \boldsymbol{x})}$$

We would then predict 1 on an input $\boldsymbol{x}$ if and only if $h(\boldsymbol{w}^\intercal \boldsymbol{x}) \geq 0.5$ or equivalently, if and only if $\boldsymbol{w}^\intercal \boldsymbol{x} \geq 0$. Consider a positive training example ($y = 1$). The larger $\boldsymbol{w}^\intercal \boldsymbol{x}$ is, the larger also is $h(\boldsymbol{w}^\intercal \boldsymbol{x})$ a.k.a the larger $P(y = 1|\boldsymbol{x}; \boldsymbol{w})$ is, and thus also the higher our degree of confidence that the label is 1. Thus, informally we can think of our prediction as being a very confident one that $y = 1$ if $\boldsymbol{w}^\intercal \boldsymbol{x} \gg 0$.

Similarly, we think of logistic regression as making a very confident prediction of $y = 0$, if $\boldsymbol{w}^\intercal \boldsymbol{x} \ll 0$. Given a training set, again informally it seems that we'd have found a good fit to the training data if we can find $\boldsymbol{w}$ so that $\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} \gg 0$ whenever $y^(i) = 1$ and $\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} \ll 0$ whenever $y^(i) = 0$, since this would reflect a very confident (and correct) set of classifications for all the training examples. This seems to be a nice goal to aim for, and we'll soon formalize this idea using the notion of functional margins.

For a different type of intuition, consider the following figure, in which the symbol "X" represent positive training examples, the symbol "O" denote negative training examples, a decision boundary (this is the line given by the equation $\boldsymbol{w}^\intercal \boldsymbol{x} = 0$ is also called the separating hyperplane) is also shown, and three points have also been labelled "A", "B" and "C".



Notice that the point "A" is very far from the decision boundary. If we are asked to make a prediction for the value of y at "A", it seems we should be quite confident that $y = 1$ there. Conversely, the point "C" is very close to the decision boundary, and while it's on the side of the decision boundary on which we would predict $y = 1$, it seems likely that just a small change to the decision boundary could easily have caused out prediction to be $y = 0$. Hence, we're much more confident about our prediction at "A" than at "C". The point "B" lies in-between these two cases, and more broadly, we see that if a point is far from the separating hyperplane, then we may be significantly more confident in our predictions. Again, informally we think it'd be nice if, given a training set, we manage to find a decision boundary that allows us to make all correct and confident (meaning far from the decision boundary) predictions on the training examples. We'll formalize this later using the notion of geometric margins.

To make our discussion of SVM's easier, we'll first need to introduce a new notation for talking about classification. We will be considering a linear classifier for a binary classification problem with labels y and features x. From now, we'll use $y \in \{-1, 1\}$ (instead of $\{0, 1\}$) to denote the class labels. Also, we will separate the w_0 component from $\boldsymbol{w}$ and from now on we will be denoting it b, and we will write our classifier as:

$$h_{\boldsymbol{w},b}(\boldsymbol{x}) = g(\boldsymbol{w}^\intercal \boldsymbol{x} + b)$$

This $\boldsymbol{w}$, b notation allows us to explicitly treat the intercept term b separately from the other parameters. (We also drop the convention we had previously of letting $x_0 = 1$ be an extra coordinate in the input feature vector.) Note also that, from our definition of g above, our classifier will directly predict either 1 or -1 without first going through the intermediate step of estimating the probability of y being 1 (which was what logistic regression did).

Definition 2.40 (Functional Margin Of A Training Example)**.** *Given a training example* $(x^{(i)}, y^{(i)})$ *we define the* ***functional margin of*** $(\boldsymbol{w}, b)$ ***with respect to the training example*** *as:*

$$\hat{\gamma}^{(i)} = y^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b)$$

Note that if $y^{(i)} = 1$, then for the functional margin to be large (i.e., for our prediction to be confident and correct), we need $\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b$ to be a large positive number. Conversely, if $y^{(i)} = -1$, then for the functional margin to be large (i.e., for our prediction to be confident and correct), we need $\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b$ to be a large negative number. Moreover, if $\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b \neq 0$, then our prediction on this example is correct. Hence, a large functional margin represents a confident and a correct prediction.

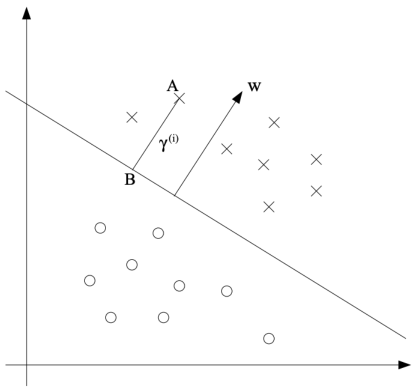
For a linear classifier with the choice of g given above, there's one property of the functional margin that makes it not a very good measure of confidence, however. Given our choice of g, we note that if we replace $\boldsymbol{w}$ with $2\boldsymbol{w}$ and b with $2b$, then since $g(\boldsymbol{w}^\intercal \boldsymbol{x} + b) = g(2\boldsymbol{w}^\intercal \boldsymbol{x} + 2b)$ this would not change $h_{\boldsymbol{w},b}(\boldsymbol{x})$ at all meaning that g, and hence also $h_{\boldsymbol{w},b}(\boldsymbol{x})$, depends only on the sign, but not on the magnitude, of $\boldsymbol{w}^\intercal \boldsymbol{x} + b$. However, replacing the scaling by a factor also results in multiplying our functional margin by the same factor. Thus, it seems that by exploiting our freedom to scale $\boldsymbol{w}$ and b, we can make the functional margin arbitrarily large without really changing anything meaningful. Intuitively, it might therefore make sense to impose some sort of normalization condition.

Given a training set we also define the functional margin of $(\boldsymbol{w}, b)$ with respect to the set as follows.

Definition 2.41 (Functional Margin Of A Set)**.** *Given a training set $\{x^{(i)}, y^{(i)}\}$ we define the* ***functional margin of*** $(\boldsymbol{w},\, b)$ ***with respect to the training set*** *as:*

$$\hat{\gamma} = \min \hat{\gamma}^{(i)}$$

Next, let's talk about geometric margins. Consider the picture below:



The decision boundary corresponding to $(\boldsymbol{w},\, b)$ is shown, along with the vector $\boldsymbol{w}$. Note that $\boldsymbol{w}$ is orthogonal to the separating hyperplane. Consider the point at A, which represents the input $\boldsymbol{x}^{(i)}$ label $y^{(i)} = 1$. Its distance to the decision boundary, $\gamma^{(i)}$ is given by the line segment AB.

How can we find the value of $y^{(i)}$? Well, $\frac{\boldsymbol{w}}{||\boldsymbol{w}||}$ is a unit-length vector pointing in the same direction as $\boldsymbol{w}$. Since A represents $\boldsymbol{x}^{(i)}$ we therefore find that the point B is given by $\boldsymbol{x}^{(i)} - \gamma^{(i)} \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$. But this point lies on the decision boundary, and all points on the decision boundary satisfy the equation $\boldsymbol{w}^\intercal \boldsymbol{x} + b = 0$. Hence:

$$\boldsymbol{w}^\intercal \left(\boldsymbol{x}^{(i)} - \gamma^{(i)} \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \right) + b = 0$$

Solving for $\gamma^{(i)}$ yields:

$$\gamma^{(i)} = \frac{\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b}{||\boldsymbol{w}||} = \left(\frac{\boldsymbol{w}}{||\boldsymbol{w}||} \right)^\intercal \boldsymbol{x}^{(i)} + \frac{b}{||\boldsymbol{w}||}$$

This was worked out for the case of a positive training example at A in the figure, where being on the positive side of the decision boundary is good.

We are now ready to properly define the geometrical margin.

Definition 2.42 (Geometrical Margin Of A Training Example)**.** *Given a training example $(x^{(i)}, y^{(i)})$ we define the* ***geometrical margin of*** $(\boldsymbol{w},\, b)$ ***with respect to the training example*** *as:*

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{\boldsymbol{w}}{||\boldsymbol{w}||} \right)^\intercal \boldsymbol{x}^{(i)} + \frac{b}{||\boldsymbol{w}||} \right)$$

Note that if $||\boldsymbol{w}|| = 1$, then the functional margin equals the geometric margin. This thus gives us a way of relating these two different notions of margin. Also, the geometric margin is invariant to rescaling of the parameters. This will in fact come in handy later. Specifically, because of this invariance to the scaling of the parameters, when trying to fit $\boldsymbol{w}$ and b to training data, we can impose an arbitrary scaling constraint on $\boldsymbol{w}$ without changing anything important.

Finally, given a training set we also define the geometric margin of (w, b) with respect to the set to be the smallest of the geometric margins on the individual training examples:

Definition 2.43 (Geometrical Margin Of A Set)**.** *Given a training set $\{x^{(i)}, y^{(i)}\}$ we define the* ***geometrical margin of*** $(\boldsymbol{w},\, b)$ ***with respect to the training set*** *as:*

$$\gamma = \min \gamma^{(i)}$$

Given a training set, it seems from our previous discussion that a natural desideratum is to try to find a decision boundary that maximizes the (geometric) margin, since this would reflect a very confident set of predictions on the training set and a good "fit" to the training data. Specifically, this will result in a classifier that separates the positive and the negative training examples with a "gap" (geometric margin).

For now, we will assume that we are given a training set that is linearly separable, i.e that it is possible to separate the positive and negative examples using some separating hyperplane. How we we find the one that achieves the maximum geometric margin? We can pose the following optimization problem:

$$\max_{\gamma,\boldsymbol{w},b} \gamma$$

subject to:

$$y^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b) \geq \gamma, \quad i = 1, 2, \ldots, m \qquad \textit{and} \qquad ||\boldsymbol{w}|| = 1$$

In other words, we want to maximize γ, subject to each training example having functional margin at least γ. The $||\boldsymbol{w}|| = 1$ constraint moreover ensures that the functional margin equals to the geometric margin, so we are also guaranteed that all the geometric margins are at least γ,. Thus, solving this problem will result in parameters with the largest possible geometric margin with respect to the training set.

If we could solve the optimization problem above, we'd be done. But the $||\boldsymbol{w}|| = 1$ constraint is a nasty (non-convex) one, and this problem certainly isn't in any format that we can plug into standard optimization software to solve. So, lets try transforming the problem into a nicer one. Consider:

$$\max_{\gamma,\boldsymbol{w},b} \frac{\hat{\gamma}}{||\boldsymbol{w}||}$$

subject to:

$$y^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b) \geq \hat{\gamma}, \quad i = 1, 2, \ldots, m$$

Here, we're going to maximize $\frac{\hat{\gamma}}{||\boldsymbol{w}||}$ subject to the functional margins all being at least $\hat{\gamma}$. Since the geometric and functional margins are related by $\gamma = \frac{\hat{\gamma}}{||\boldsymbol{w}||}$ this will give us the answer we want. Moreover, we've gotten rid of the constraint $||\boldsymbol{w}|| = 1$ that we didn't like. The downside is that we now have a nasty (again, non-convex) objective $\frac{\hat{\gamma}}{||\boldsymbol{w}||}$ function, and, we still don't have any off-the-shelf software that can solve this form of an optimization problem.

Lets keep going. Recall our earlier discussion that we can add an arbitrary scaling constraint on $\boldsymbol{w}$ and b without changing anything. This is the key idea we'll use now. We will introduce the scaling constraint that the functional margin of $\boldsymbol{w}$ and b with respect to the training set must be 1:

$$\hat{\gamma} = 1$$

Since multiplying $\boldsymbol{w}$ and b by some constant results in the functional margin being multiplied by that same constant, this is indeed a scaling constraint, and can be satisfied by rescaling $\boldsymbol{w}$ and b. Plugging this into our problem above, and noting that maximizing $\frac{\hat{\gamma}}{||\boldsymbol{w}||} = \frac{1}{||\boldsymbol{w}||}$ is the same thing as minimizing $||\boldsymbol{w}||^2$, we now have the following optimization problem:

$$\min_{\gamma,\boldsymbol{w},b} \frac{1}{2}||\boldsymbol{w}||^2$$

subject to:

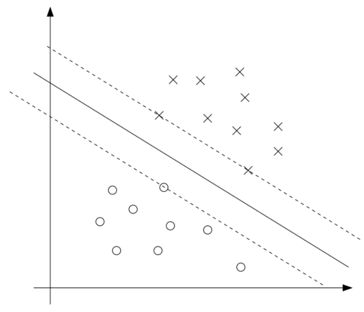
$$y^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b) \geq 1, \quad i = 1, 2, \ldots, m$$

We can rewrite the constraints as:

$$g_i(\boldsymbol{w}) = -y^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b) + 1 \leq 0, \quad i = 1, 2, \ldots, m$$

According to Appendix A we are dealing with an inequality constrained optimization problem (with no equality part) so we will use the KKT multipliers in order to solve it.

Notice that that all KKT conditions are satisfied, since we do not have any equality constraints (hence $\mu_i = 0, \ \ \forall i$), and we have $\lambda_i > 0$ only for the training examples that have functional margin exactly equal to one, i.e the ones corresponding to constraints that hold with equality $g_i(\boldsymbol{w}) = 0$. Consider the figure below, in which a maximum margin separating hyperplane is shown by the solid line.



The points with the smallest margins are exactly the ones closest to the decision boundary. Here these are the three points: one negative and two positive examples, that lie on the dashed lines parallel to the decision boundary. Thus, only three of the λ_i (the ones corresponding to these three training examples) will be non-zero at the optimal solution to our optimization problem. These three points are called "support vectors". The fact that the number of support vectors can be much smaller than the size the training set will be useful later.

Moving on by applying what we developed in Appendix A the Lagrangian of the inequality constrained optimization problem reads:

$$\mathcal{L}(\boldsymbol{w}, b, \lambda_i) = \frac{1}{2}||\boldsymbol{w}||^2 - \sum_i \lambda_i \left(y^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b) - 1 \right)$$

Note that there are only λ_i but no μ_i Lagrange multipliers, since the problem has only inequality constraints.

Now we move on trying to solve this optimization problem. Once again, according to Appendix A the necessary conditions for optimization of $\mathcal{L}$ turn to:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, b, \lambda_i) = 0 \ \text{ and } \ \frac{\partial \mathcal{L}(\boldsymbol{w}, b, \lambda_i)}{\partial b} = 0 \ \text{ and } \ \frac{\partial \mathcal{L}(\boldsymbol{w}, b, \lambda_i)}{\partial \lambda_i} = 0$$

Starting with the first one, quite straightforward we obtain:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, b, \lambda_i) = \boldsymbol{w} - \sum_i \lambda_i y^{(i)} \boldsymbol{x}^{(i)} = 0$$

which implies:

$$\boldsymbol{w} = \sum_i \lambda_i y^{(i)} \boldsymbol{x}^{(i)}$$

As for the derivative with respect to b, we obtain:

$$\frac{\partial \mathcal{L}(\boldsymbol{w}, b, \lambda_i)}{\partial b} = \sum_i \lambda_i y^{(i)} = 0$$

By manipulating the initial Lagrangian a bit we get:

$$\begin{aligned}\mathcal{L}(\boldsymbol{w}, b, \lambda_i) &= \frac{1}{2}||\boldsymbol{w}||^2 - \sum_i \lambda_i \left(y^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b) - 1 \right) \\ &= \frac{1}{2}||\boldsymbol{w}||^2 - \sum_i \lambda_i y^{(i)} \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} - b \sum_i \lambda_i y^{(i)} + \sum_i \lambda_i\end{aligned}$$

By substituting the results from the two first derivatives in the previous expression of the Lagrangian we end up to:

$$
\begin{aligned}
\mathcal{L}(\boldsymbol{w}, b, \lambda_i) &= \frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}\left(\boldsymbol{x}^{(i)}\right)^{\mathsf{T}}\boldsymbol{x}^{(j)} - \sum_i \lambda_i y^{(i)}\left(\sum_j \lambda_j y^{(j)}\left(\boldsymbol{x}^{(j)}\right)^{\mathsf{T}}\right)\boldsymbol{x}^{(i)} + \sum_i \lambda_i \\
&= \frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}\left(\boldsymbol{x}^{(i)}\right)^{\mathsf{T}}\boldsymbol{x}^{(j)} - \sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}\left(\boldsymbol{x}^{(i)}\right)^{\mathsf{T}}\boldsymbol{x}^{(j)} + \sum_i \lambda_i \\
&= \sum_i \lambda_i - \frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}\left(\boldsymbol{x}^{(i)}\right)^{\mathsf{T}}\boldsymbol{x}^{(j)}
\end{aligned}
$$

Hence now the Lagrangian reads:

$$
\mathcal{L}(\lambda_i) = \sum_i \lambda_i - \frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}\left(\boldsymbol{x}^{(i)}\right)^{\mathsf{T}}\boldsymbol{x}^{(j)}
$$

where the dependencies on the parameters $\boldsymbol{w}$ and b are gone and the Lagrangian is just a function of the KKT multipliers. So we ended up with the so called "dual optimization problem" which is:

$$
\max_{\lambda_i}\left(\sum_i \lambda_i - \frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}\left(\boldsymbol{x}^{(i)}\right)^{\mathsf{T}}\boldsymbol{x}^{(j)}\right)
$$

subject to:

$$
\lambda_i \geq 0, \quad i = 1, 2, \ldots, m \qquad \textit{and} \qquad \sum_i \lambda_i y^{(i)} = 0
$$

where as per usual we can turn the maximization to minimization by multiplying the Lagrangian with a minus sign so we end up with:

$$
\min_{\lambda_i}\left(\frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}\left(\boldsymbol{x}^{(i)}\right)^{\mathsf{T}}\boldsymbol{x}^{(j)} - \sum_i \lambda_i\right)
$$

subject to:

$$
\lambda_i \geq 0, \quad i = 1, 2, \ldots, m \qquad \textit{and} \qquad \sum_i \lambda_i y^{(i)} = 0
$$

As usual we can rewrite everything in form of matrices. By introducing the matrix Q and the KKT multipliers vector $\boldsymbol{\lambda}$:

$$
Q = \begin{bmatrix}
y^{(1)}y^{(1)}\left(\boldsymbol{x}^{(1)}\right)^{\mathsf{T}}\boldsymbol{x}^{(1)} & \ldots & y^{(1)}y^{(m)}\left(\boldsymbol{x}^{(1)}\right)^{\mathsf{T}}\boldsymbol{x}^{(m)} & \\
y^{(2)}y^{(2)}\left(\boldsymbol{x}^{(2)}\right)^{\mathsf{T}}\boldsymbol{x}^{(2)} & \ldots & y^{(2)}y^{(m)}\left(\boldsymbol{x}^{(2)}\right)^{\mathsf{T}}\boldsymbol{x}^{(m)} & \\
\vdots & \vdots & \ddots & \ldots \\
y^{(m)}y^{(1)}\left(\boldsymbol{x}^{(m)}\right)^{\mathsf{T}}\boldsymbol{x}^{(1)} & \ldots & y^{m)}y^{(m)}\left(\boldsymbol{x}^{(m)}\right)^{\mathsf{T}}\boldsymbol{x}^{(m)} &
\end{bmatrix}, \qquad
\boldsymbol{\lambda} = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_m \end{bmatrix}
$$

we can rewrite the dual optimization problem as:

$$
\min_{\boldsymbol{\lambda}}\left(\frac{1}{2}\boldsymbol{\lambda}^{\mathsf{T}}Q\boldsymbol{\lambda} - \boldsymbol{\lambda}\right)
$$

subject to:

$$
\boldsymbol{\lambda} \geq 0 \qquad \textit{and} \qquad \boldsymbol{y}^{\mathsf{T}}\boldsymbol{\lambda} = 0
$$

This final dual optimization problem is usually solved numerically through quadratic programming (as for example we use gradient descent for loss functions optimization problems). Once we solve it we obtain the KKT multipliers vector $\boldsymbol{\lambda}$ and subsequently all the individual KKT multipliers λ_i from its components. We also notice that almost all λ_i's are 0 except from a few ones where $\lambda_i > 0$ coming from the support vectors. By making use of the KKT multipliers we find the parameter $\boldsymbol{w}$ as:

$$\boldsymbol{w} = \sum_i \lambda_i y^{(i)} \boldsymbol{x}^{(i)}$$

We once again notice that the only terms that survive are the ones with $\lambda_i > 0$ coming from the support vectors. All the other points do not contribute at all to the model.

Once we have $\boldsymbol{w}$ we can compute b from the fact that for support vectors their distance from the plane is equal to 1, i.e:

$$y^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x}^{(i)} + b) = 1$$

hence:

$$b = \frac{1}{y^{(i)}} - \boldsymbol{w}^\intercal \boldsymbol{x}^{(i)}$$

for any i that is a support vector. This last equation should produce the same parameter b for all support vectors. This is a good way to check that the models works fine.

By having found both $\boldsymbol{w}$ and b, we have found the separator $\boldsymbol{w}^\intercal \boldsymbol{x} + b$ and we are in the position to predict to which class a new point belongs. Since $\boldsymbol{w}^\intercal \boldsymbol{x} + b$ is going to be either negative (if the point lies below the plane) or positive (if the point lies abbove the plane) we finally have for the prediction:

$$h(\boldsymbol{x}) = sign(\boldsymbol{w}^\intercal \boldsymbol{x} + b)$$

This model of classification is called "hard margin support vector machine". A careful reader would notice that in hard margin support vector machine we strictly impose that all instances must be off the street and on the right side, i.e no violations are allowed (hence the naming). There are two main issues with hard margin classification. First, it only works if the data is linearly separable. Second, it is sensitive to outliers.

However in real life the data are almost never linearly separable and the hard margin support vector machine would not be able to find a solution. To avoid these issues, one should use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the margin violations (i.e., instances that end up in the middle of the street or even on the wrong side).

Fortunately the generalization of hard margin support vector machine to non-linear separable is easy. Without getting into details two of the ways we can achieve this are the following.

- **Non-Linear Transformations & Kernels**

  By switching from a space X where the data are not linear separable to a space Z where the data are linear separable through a transformation Φ:

  $$(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}) \xrightarrow{\Phi} (\boldsymbol{z_1}, \boldsymbol{z_2}, \ldots, \boldsymbol{z_n})$$

  Accordingly the dual optimization problem would change to:

  $$\min_{\lambda_i} \left(\frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y^{(i)} y^{(j)} \left(\boldsymbol{z}^{(i)}\right)^\intercal \boldsymbol{z}^{(j)} - \sum_i \lambda_i \right)$$

  subject to:

  $$\lambda_i \geq 0, \quad i = 1, 2, \ldots, m \qquad and \qquad \sum_i \lambda_i y^{(i)} = 0$$

  and then we would follow the same procedure as before in order to obtain a solution.

  The biggest caveat of this method is the inner product $\left(\boldsymbol{z}^{(i)}\right)^\intercal \boldsymbol{z}^{(j)}$ that needs to be computed in Z space and it's computationally expensive. One way to overcome this problem is by using the trick of kernels (Appendix B) in which we substitute the inner product with a kernel:

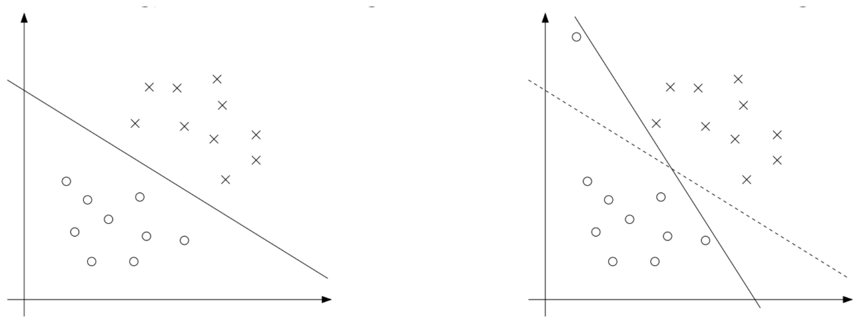
$$\left(\boldsymbol{z}^{(i)}\right)^{\intercal} \boldsymbol{z}^{(j)} = K(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$$

Given the freedom of choosing a kernel the model turns quite flexible.

- **L1 Regularization & Soft Margin SVM**

  While mapping data to a high dimensional feature space via Φ does generally increase the likelihood that the data is separable, we can't guarantee that it always will be so. Also, in some cases it is not clear that finding a separating hyperplane is exactly what we'd want to do, since that might be susceptible to outliers.

  To make the algorithm work for non-linearly separable datasets as well as be less sensitive to outliers, we can simply impose the technique of L1 Regularization that we mentioned in the previous chapter. That way our model is more open to errors and it allows some wrongly labeled data.



  In this case the final model is called "soft margin support vector machine" (in contrast with the SVM we developed in the beginning called "hard margin support vector machine"), due to the fact that it allows the margin to be violated by outliers.

2.10 k-Nearest Neighbors

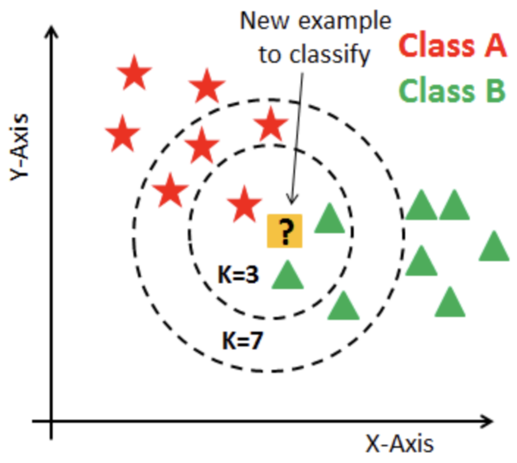
In this section we will introduce yet another famous supervised learning algorithm called "k-nearest neighbors algorithm" (k-NN), which is a non-parametric[1] method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space, since the algorithm assumes that similar things exist in close proximity (in other words, similar things are near to each other). The output depends on whether k-NN is used for classification or regression:
Excerpt From: Aurélien Géron. "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow". Apple Books.

- In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.
- In k-NN regression, the output is the property value for the object. This value is the average of the values of k nearest neighbors.

One good question is "how to measure the distance between two points"? There are many ways of calculating distance, and one way might be preferable depending on the problem we are solving. However, the straight-line distance (i.e the Euclidean distance) is a popular and familiar choice.

[1]A model is called "nonparametric" not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a parametric model, such as a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

Another good questions is “how many neighbors shoud I choose for the model”? In other words what should be the value of k? The best choice of k depends upon the data. Generally, larger values of k reduces effect of the noise on the classification, but make boundaries between classes less distinct. A good k can be selected by various heuristic techniques. In binary classification problems, it is helpful to choose k to be an odd number as this avoids tied votes. One popular way of choosing the empirically optimal k in this setting is via bootstrap method. The special case where the class is predicted to be the class of the closest training sample (i.e when $k = 1$) is called the “nearest neighbor” algorithm.



k-NN has many advantages. First of all the algorithm is very simple and easy to implement. Also, there is no need to build a model, tune several parameters, or make additional assumptions. and finaly the algorithm is versatile., i.e it can be used for classification, regression. However of course, it carries and some disadvantages. One has to keep in mind that the algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase and that the accuracy of k-NN algorithm can be severely degraded by the presence of noisy or irrelevant features, or if the feature scales are not consistent with their importance.

A drawback of the basic “majority voting” classification occurs when the class distribution is skewed. That is, examples of a more frequent class tend to dominate the prediction of the new example, because they tend to be common among the k nearest neighbors due to their large number. In order to overcome this problem (both for classification and regression), a useful technique can be to assign weights to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbor a weight of $\frac{1}{d}$, where d is the distance to the neighbor.

One final, general comment is in order. Up to this point, all the algorithms we developed were model-based algorithms since in each of them we built a model based on the data. k-NN is the first instance-based algorithm we introduce, where the function is only approximated locally and all computation is deferred until function evaluation. In other words, the neighbors are taken from a set of objects for which the class is known. This can be thought of as the training set for the algorithm, though no explicit training step is required. This has as a consequence for the k-NN algorithm to be sensitive to the local structure of the data. Since this algorithm relies on distance for classification, normalizing the training data can improve its accuracy dramatically.

2.11 Decision Trees & Random Forests

Like SVM and k-NN, decision trees are versatile, non-parametric machine learning algorithms that can perform both classification and regression tasks. They go from observations about an item (represented in the branches) to conclusions about the item’s target value (represented in the leaves). They are powerful algorithms, capable of fitting complex datasets. Decision trees are also the fundamental components of random forests (that we will see right after), which are among the most powerful machine learning algorithms available today.

2.11.1 Decision Trees

Definition 2.44 (Decision Tree)**.** *A **decision tree** is a decision support tool that uses a flowchart-like tree-like structure of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.*

The flowchart-like structure of a decision tree helps in decision making. It's a visualization like a flowchart diagram which easily mimics the human-level thinking. That is why decision trees are easy to understand and interpret.

Now let's give some basic definitions.

Definition 2.45 (Node)**.** *A **node** of a decision tree represents a feature.*

Definition 2.46 (Root Node)**.** *The topmost node in a decision tree is known as the **root node**, or top decision node, of the decision tree. It represents the entire population or sample and this further gets divided into two or more homogeneous sets.*

The root node learns to partition based on the attribute value. It partitions the tree in a recursive manner called "recursive partitioning".

Definition 2.47 (Splitting)**.** ***Splitting** is a process of dividing a node into two or more sub-nodes.*

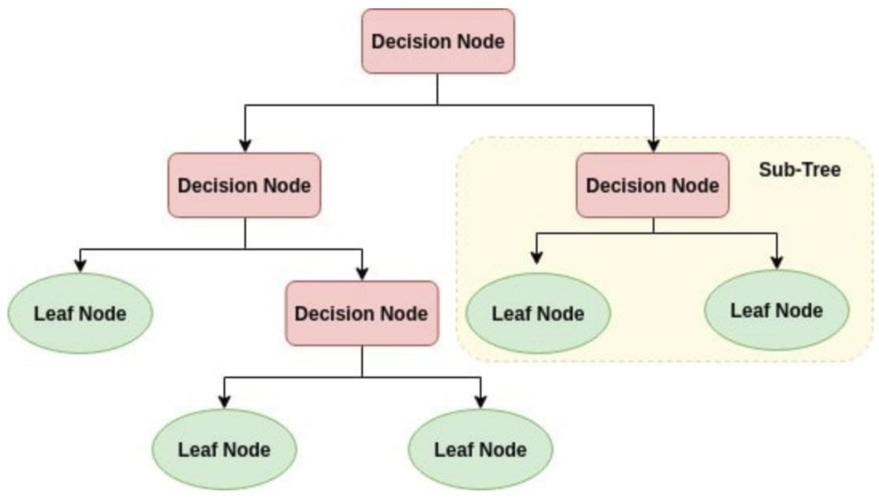
Definition 2.48 (Pruning)**.** ***Pruning** is the process when one reduces the size of decision trees by removing nodes (opposite of splitting).*

Definition 2.49 (Decision Node)**.** *When a subnode splits into further subnodes, then it is called a **decision node**.*

Definition 2.50 (Parent / Child Node)**.** *A node which is divided into subnodes is called a **parent node** of subnodes whereas subnodes are the **child** of a parent node.*

Definition 2.51 (Leaf)**.** *A **leaf**, or terminal node, is a node with no children (no further split) and it represents the outcome of a decision rule.*

Definition 2.52 (Branch)**.** *A **branch** , or subtree, is a subsection of the decision tree and it represents a decision rule.*



Decision trees classify (or regress) the examples by sorting them down the tree, from the root to some leaf node, with the leaf node providing the classification (or regression) to the example. This approach is called a "top-down" approach. Each node in the tree acts as a test case for some attribute, and each edge descending from that node corresponds to one of the possible answers to the test case.

The algorithm begins by selecting the best attribute using something called "attribute selection measures" to split the records. Then it makes that attribute a decision node and breaks the dataset into smaller subsets. Is starts tree building by repeating this process recursively for each child until one of the condition will match:

- All the tuples belong to the same feature value.
- There are no more remaining feature.
- There are no more instances.

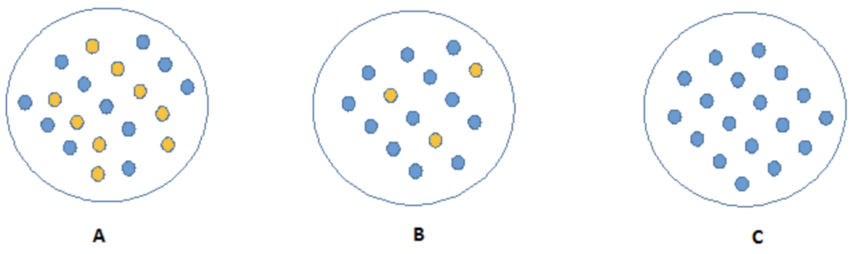
Attribute selection measure is a heuristic for selecting the splitting criterion that partition data into the best possible manner. It is also known as "splitting rules" because it helps us to determine breakpoints for tuples on a given node. Attribute selection measure provides a rank to each feature by explaining the given dataset. Best score attribute will be selected as a splitting attribute. In the case of a continuous-valued attribute, split points for branches also need to define.

Two of the most popular selection measures are: "information gain", and "Gini impurity" and we will now explain what they are. Both information gain and Gini impurity measure the impurity of a node. So let's start by defining what impurity is.

Definition 2.53 (Node Impurity)**.** ***Node impurity*** *is a measure of the homogeneity of the labels at the node.*

Definition 2.54 (Pure / Impure Node)**.** *A node is called* **pure** *when all of its data belongs to a single class. Subsequently a node is called* **impure** *when not all of its data belongs to a single class. A node is 100% impure when a node is split evenly 50/50.*

As an example, in the figure below C requires less information to describe as all values are similar. On the other hand, B requires more information to describe it and A requires the maximum information. In other words, we can say that C is a pure node, B is less impure and A is more impure.



As a final step we need to introduce the concept of information entropy.

Definition 2.55 (Information Entropy)**.** *In information theory, the* ***information entropy****, or simply entropy, of a random variable is the average level of information inherent in the variable's possible outcomes. Given a discrete random variable X, with possible outcomes $\{x_1, x_2, \dots, x_n\}$ which occur with probability $P(x_1), P(x_2), \dots, P(x_n)$, the entropy of X is formally defined as:*

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log P(x_i)$$

where the summation is over the variable's possible values.

Having defined entropy we can now define information gain.

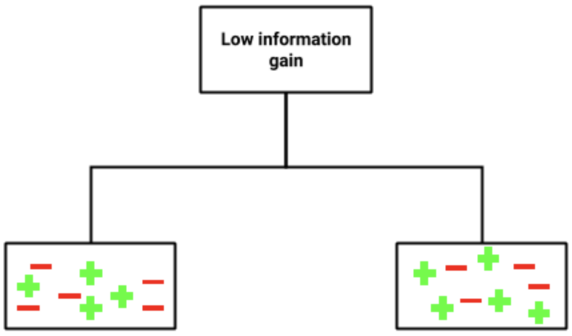
Definition 2.56 (Information Gain)**.** ***Information gain*** *IG is the change in information entropy H from a prior state T to a state that takes some information a as given:*

$$IG(T, a) = H(T) - H(T|a)$$

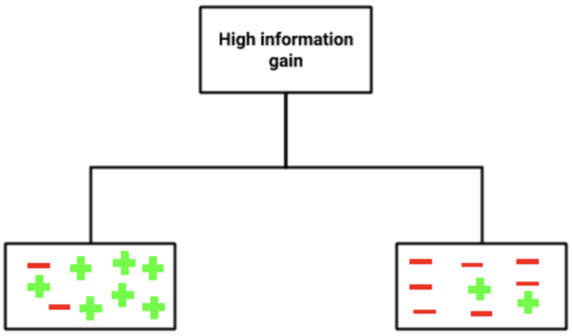
where $H(T)$ is the entropy of the parent node and $H(T|a)$ is the sum of the entropies of all children nodes.

Informally speaking, information gain is a statistical property that measures how well a given attribute separates the training examples according to their target classification. It is a synonym for "Kullback-Leibler divergence", that we introduced in the parametric inference chapter,, which shows the amount of information gained about a random variable or signal from observing another random variable. However, in the context of decision trees, the term is sometimes used synonymously with mutual information, which is the conditional expected value of the Kullback-Leibler divergence of the univariate probability distribution of one variable from the conditional distribution of this variable given the other one.

In the figure below, we can see that a feature with low information gain splits the data relatively evenly and as a result doesn't bring us any closer to a decision.



Whereas, an attribute with high information gain splits the data into groups with an uneven number of positives and negatives and as a result, helps in separating the two from each other.



In other words we are always looking to maximize the information gain after any splitting. Having defined information gain we can now define Gini impurity.

Definition 2.57 (Gini Impurity)**.** ***Gini impurity**, or Gini index, I_G measures the degree or probability of a particular variable being wrongly classified when it is randomly chosen. Given a discrete random variable X, with possible outcomes $\{x_1, x_2, \ldots, x_n\}$ which occur with probability $P(x_1), P(x_2), \ldots, P(x_n)$, the Gini impurity of X is formally defined as:*

$$I_G(X) = 1 - \sum_{i=1}^{n} P^2(x_i)$$

where the summation is over the variable's possible values.

Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset.

Now that we have defined both information gain and Gini impurity, a good question is "which one should one use"? In general keep in mind that it only matters in 2% of the cases whether you use Gini impurity or information gain. And given that information gain uses information entropy (which uses logarithms), it might be a little slower to compute. Hence most of the algorithms use Gini impurity as a selection measure.

In nay case, by using either information gain or Gini impurity one can start building (training) the tree. The building (training) is usually done through a procedure called "Classification and Regression Tree" (CART) algorithm. The algorithm works by first splitting the training set into two subsets using a single feature k and a threshold t_k, by searching for the pair $(k,\ t_k)$ that produces the purest subsets (weighted by their size). The loss function that the algorithm tries to minimize is the so called "CART loss function". There is one CART loss function for classification and one for regression. Let's start with the classification one.

Definition 2.58 (CART Loss Function (For Classification))**.** *We define the* ***CART loss function*** *for classification as:*

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$$

where m is the total number of instances in both left and right subsets, m_{left} and m_{right} are the number of instances in the left and right subset, and G_{left} and G_{right} measure the impurity of the left and the right subset (G could be either information gain or Gini impurity).

For regression, CART algorithm instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. Hence the corresponding CART loss function for regression looks like the following.

Definition 2.59 (CART Loss Function (For Regression))**.** *We define the* ***CART loss function*** *for regression as:*

$$J(k, t_k) = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right}$$

where m is the total number of instances in both left and right subsets, m_{left} and m_{right} are the number of instances in the left and right subset, and MSE_{left} and MSE_{right} are the mean squared errors of the left and the right subsets given by:

$$MSE_{node} = \sum_{i \in node} (\hat{y}_{node} - y^{(i)})^2$$

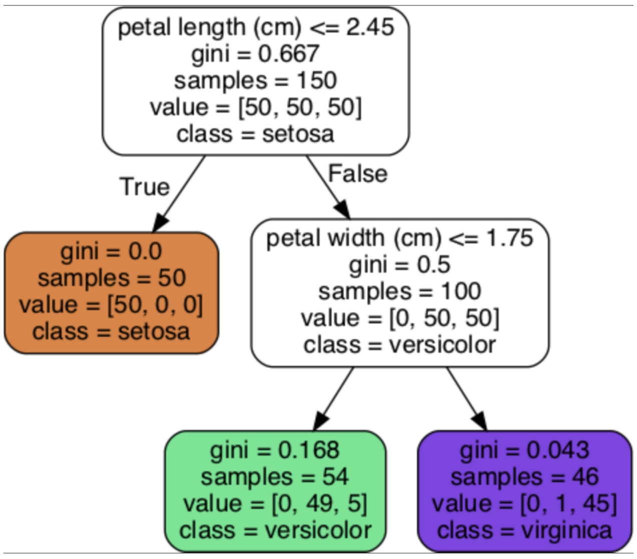
where:

$$\hat{y}_{node} = \frac{1}{m_{node}} \sum_{i \in node} y^{(i)}$$

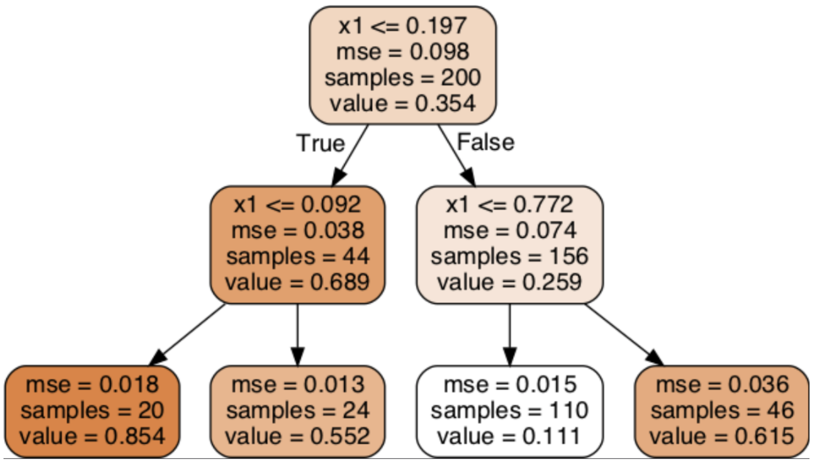
In both cases (classification and regression), once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the subsubsets, and so on, recursively. As we can see, the CART algorithm is a greedy algorithm. It greedily searches for an optimum split at the top level, then repeats the process at each subsequent level. It does not check whether or not the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a solution that's reasonably good but not guaranteed to be optimal. Unfortunately, finding the optimal tree requires $O(exp(m))$ time, making the problem intractable even for small training sets. This is why we must settle for a "reasonably good" solution.

Once the CART algorithm finished, the building (training) part is over and the decision tree is ready. One can now feed a new instance to the tree and by asking the corresponding question of each node they will end up in one of the leafs, hence to a decision. In order to have a better understanding of the procedure (although it is really simple) we will provide two small examples (one for classification and one for regression).

Let's start with classification. One starts at the root node (depth 0, at the top). This node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then they move down to the root's left child node (depth 1, left). In this case, it is a leaf node (i.e it does not have any child nodes), so it does not ask any questions but simply look at the predicted class for that node, and the decision tree predicts that the flower is a "setosa". Now suppose one finds another flower, and this time the petal length is greater than 2.45 cm. They must move down to the root's right child node (depth 1, right), which is not a leaf node, so the node asks another question: "is the petal width smaller than 1.75 cm"? If it is, then the flower is most likely a "versicolor" (depth 2, left). If not, it is likely a "virginica" (depth 2, right).



The regression decision tree example looks very similar to the classification one with the main difference that instead of predicting a class in each node, it predicts a value. Suppose you want to make a prediction for a new instance with $x_1 = 0.6$. You traverse the tree starting at the root, and you eventually reach the leaf node that predicts $value = 0.111$. This prediction is the average target value of the 110 training instances associated with this leaf node, and it results in a mean squared error equal to 0.015 over these 110 instances.



Decision trees are intuitive, and their decisions are easy to interpret. Such models are often called “white box” models. In contrast, as we will see, random forests or neural networks are generally considered “black box” models. They make great predictions, and you can easily check the calculations that they performed to make these predictions, nevertheless, it is usually hard to explain in simple terms why the predictions were made. Conversely, decision trees provide nice, simple classification rules that can even be applied manually if need be. The main drawback of decision trees is that they are very sensitive to small variations in the training data. Random forests can limit this instability by averaging predictions over many trees, as we will see in a while.

2.11.2 Ensemble Learning

Suppose one pose a complex question to thousands of random people, then aggregate their answers. In many cases they will find that this aggregated answer is better than an expert’s answer. This is called the “wisdom of the crowd”. Similarly, if one aggregates the predictions of a group of predictors, they will often get better predictions than with the best individual predictor. A group of predictors is called an “ensemble”, thus, this technique is called “ensemble learning”, and an ensemble learning algorithm is called an “ensemble method”.

As an example of an ensemble method, one can train a group of decision tree classifiers, each on a different random subset of the training set. To make predictions, one obtain the predictions of all the individual trees, then predict the class that gets the most votes. Such an ensemble of decision trees is called a “random forest”, and despite its simplicity, this is one of the most powerful machine learning algorithms available today. More on random forests in the next section.

An ensemble method is itself a supervised learning algorithm, because it can be trained and then used to make predictions. The trained ensemble method, therefore, represents a single hypothesis. This hypothesis, however, is not necessarily contained within the hypothesis space of the models from which it is built. Thus, ensemble methods can be shown to have more flexibility in the functions they can represent. This flexibility can, in theory, enable them to overfit the training data more than a single model would, but in practice, some ensemble methods tend to reduce problems related to overfitting of the training data.

Empirically, ensembles tend to yield better results when there is a significant diversity among the models. Many ensemble methods, therefore, seek to promote diversity among the models they combine. Although perhaps non-intuitive, more random algorithms can be used to produce a stronger ensemble than very deliberate algorithms. Using a variety of strong learning algorithms, however, has been shown to be more effective than using techniques that attempt to dumb-down the models in order to promote diversity.

While the number of component classifiers of an ensemble has a great impact on the accuracy of prediction, there is a limited number of studies addressing this problem. A priori determining of ensemble size and the volume and velocity of big data streams make this even more crucial for online ensemble classifiers. Mostly statistical tests were used for determining the proper number of components. More recently, a theoretical framework suggested that there is an ideal number of component classifiers for an ensemble such that having more or less than this number of classifiers would deteriorate the accuracy. It is called “the law of diminishing returns in ensemble construction”. Their theoretical framework shows that using the same number of independent component classifiers as class labels gives the highest accuracy.

Now we will discuss the most popular ensemble methods, including bagging, boosting, and stacking. After that we will explore random forests.

2.11.3 Random Forests

Appendices

Appendix A

Constrained Optimization

Constrained optimization is the problem of finding a minimum (or maximum) of a function $f(x)$ called the "objective function", subject to a number of constraints of the following types:

- $h_i(x) = 0, \;\; i = 1, 2, \ldots$ called "equality constraints"
- $g_i(x) \leq 0, \;\; i = 1, 2, \ldots$ called "inequality constraints"

Let's start first with the equality constraints and the we will add inequality constraints.

A.1 Equality Constrained Optimization

The formulation of the optimization problem is to optimize $f(x)$ subject to $h_i(x) = 0, \; i = 1, 2, \ldots$. Let's assume for simplicity only one constraint $h_1(x) = h(x) = 0$. The idea here is that the point that $f(x)$ touches $h(x)$ is the point that $f(x)$ is minimum (or maximum) while the constraint is also valid. At that point $f(x)$ is parallel to $h(x)$ and the tangents $\nabla_{\boldsymbol{w}} f(x)$ and $\nabla_{\boldsymbol{w}} h(x)$ which are perpendicular to $f(x)$ and $h(x)$ respectively, are also parallel to each other. Hence, since $\nabla_{\boldsymbol{w}} f(x)$ and $\nabla_{\boldsymbol{w}} h(x)$ are parallel this translates to:

$$\nabla_{\boldsymbol{w}} f(x) = \mu \nabla_{\boldsymbol{w}} h(x)$$

which is the condition for the $f(x)$ to be minimum (or maximum) while $h(x) = 0$.

Without loss of generality the condition for many constraints reads:

$$\nabla_{\boldsymbol{w}} f(x) = \sum_i \mu_i \nabla_{\boldsymbol{w}} h_i(x)$$

or by bringing everything in one side:

$$\nabla_{\boldsymbol{w}} f(x) - \sum_i \mu_i \nabla_{\boldsymbol{w}} h_i(x) = 0$$

$$\nabla_{\boldsymbol{w}} (f(x) - \sum_i \mu_i h_i(x)) = 0$$

At this point we define the "Lagrangian" as follows:

$$\mathcal{L}(x, \mu_i) = f(x) + \sum_i \mu_i h_i(x)$$

where μ_i are called "Lagrange multipliers". Subsequently the necessary conditions for optimization of $\mathcal{L}$ turns to:

$$\nabla_{\boldsymbol{w}} \mathcal{L} = 0 \;\; \text{and} \;\; \frac{\partial \mathcal{L}}{\partial \mu_i} = 0$$

The solution of this system of equations minimizes (or maximizes) $f(x)$ subject to $h_i(x) = 0, \;\; \forall i$.

A.2 Equality & Inequality Constrained Optimization

Now on top of equality constraints we also have inequality constraints. The formulation of the optimization problem is to optimize $f(x)$ subject to $h_i(x) = 0,\ \ i = 1, 2, \ldots$ and $g_i(x) \leq 0,\ \ i = 1, 2, \ldots$. Following a similar way of thinking as before, although a bit more technical, we can show (but we won't) that if the following four conditions, called "Karush - Kuhn - Talker conditions" (KKT), hold:

- $h_i(x) = 0, \quad \forall i$
- $g_i(x) \leq 0, \quad \forall i$
- $\lambda_i \leq 0, \quad \forall i$
- $\lambda_i g_i(x) = 0, \quad \forall i$

then there exist constants μ_i and λ_i called "KKT multipliers" such that:

$$\nabla_{\boldsymbol{w}} f(x) = \sum_i \mu_i \nabla_{\boldsymbol{w}} h_i(x) + \sum_i \lambda_i \nabla_{\boldsymbol{w}} g_i(x)$$

By following the same philosophy as for the equality constrained optimization we define the "Lagrangian" as follows:

$$\mathcal{L}(x, \mu_i, \lambda_i) = f(x) + \sum_i \mu_i h_i(x) + \sum_i \lambda_i g_i(x)$$

and the necessary conditions for optimization of $\mathcal{L}$ turn to:

$$\nabla_{\boldsymbol{w}} \mathcal{L} = 0 \ \text{ and } \ \frac{\partial \mathcal{L}}{\partial \mu_i} = 0 \ \text{ and } \ \frac{\partial \mathcal{L}}{\partial \lambda_i} = 0$$

This final form of the optimization problem is usually called the "dual optimization problem".

This is the most general case of constrained optimization. If there are no equality constraints $h_i(x)$ then we simply have a theory for inequality constrained optimization. If there are no inequality constraints $g_i(x)$ then the whole theory turns to the equality constrained optimization problem we developed previously and the KKT multipliers turn to Lagrangian multipliers. Finally, if there are no equality constraints $h_i(x)$ neither inequality constraints $g_i(x)$ the theory is just a usual optimization problem where we just find the solution where the derivative of $f(x)$ is zero.

Appendix B

Kernels

Definition B.1 (Kernel)**.** *Let $\bar{x}$ and $\bar{x}'$ be two vectors of space X and Φ a non-linear transformation. We define $\bar{z}$ and $\bar{z}'$ as the transformed vectors $\Phi(\bar{x})$ and $\Phi(\bar{x}')$:*

$$\bar{x} \in X \xrightarrow{\Phi} \bar{z} = \Phi(\bar{x}) \in Z$$
$$\bar{x}' \in X \xrightarrow{\Phi} \bar{z}' = \Phi(\bar{x}') \in Z$$

We define the ***kernel*** *of space Z as the function that is equal to the inner product of the transformation vectors:*

$$K(\bar{x}, \bar{x}') = \bar{z}^T \bar{z}'$$

Let's for example assume the following non-linear transformation:

$$\bar{x} = (x_1, x_2) \xrightarrow{\Phi} \bar{z} = \Phi(\bar{x}) = (1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2)$$
$$\bar{x}' = (x_1', x_2') \xrightarrow{\Phi} \bar{z}' = \Phi(\bar{x}') = (1, x_1^{2'}, x_2^{2'}, \sqrt{2}x_1', \sqrt{2}x_2', \sqrt{2}x_1'x_2')$$

Then for the inner product:

$$\bar{z}^T \bar{z}' = 1 + x_1^2 x_1^{2'} + x_2^2 x_2^{2'} + 2x_1x_1' + 2x_2x_2' + 2x_1x_2x_1'x_2'$$

However we can get to the same result by simply defining a Kernel of the form:

$$\begin{aligned} K(\bar{x}, \bar{x}') &= (1 + \bar{x}\bar{x}')^2 \\ &= (1 + x_1x_1' + x_2x_2')^2 \\ &= 1 + x_1^2 x_1^{2'} + x_2^2 x_2^{2'} + 2x_1x_1' + 2x_2x_2' + 2x_1x_2x_1'x_2' \end{aligned}$$

Hence by knowing the Kernel of a space Z of some non-linear transformation Φ we can compute inner products without the need of transforming vectors from X to Z.

The kernel trick is to use this idea in the opposite direction. Namely, to assume that a function $K(\bar{x}, \bar{x}')$ is the kernel of some space Z for some non-linear transformation Φ and to compute inner products without even knowing the transformation.

The question that arises is how do we know that some function $K(\bar{x}, \bar{x}')$ is actually the kernel of a space Z. There are three approaches to this problem:

1. By construction (as we did in the example above).
2. By Mercer's condition that states that $K(\bar{x}, \bar{x}')$ is a valid kernel for some space Z if

$$\int K(\bar{x}, \bar{x}')g(\bar{x})g(\bar{x}')d\bar{x}d\bar{x}' \geq 0 \;\; \forall \;\text{ square integrable functions }\; g(\bar{x})$$

3. Sometimes we don't care if $K(\bar{x}, \bar{x}')$ is a valid kernel for some space Z as long as it does the job.

Appendix C

Convolution

Definition C.1 (Convolution)**.** *Convolution is a mathematical operation on two functions f and g that produces a third function expressing how the shape of one is modified by the other. The term convolution refers to both the result function and to the process of computing it. It is defined as the integral of the product of the two functions after one is reversed and shifted.*

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

Appendix D

Applied Machine Learning

In this section we will cover the applied side of machine learning. This checklist can guide you through your machine learning projects. There are eight main steps:

1. **Frame The Problem**
    - Define the objective in business terms.
    - How will your solution be used?
    - What are the current solutions/workarounds (if any)?
    - How should you frame this problem (supervised/unsupervised, online/offline, etc)?
    - How should performance be measured?
    - Is the performance measure aligned with the business objective?
    - What would be the minimum performance needed to reach the business objective?
    - What are comparable problems? Can you reuse experience or tools?
    - Is human expertise available? How would you solve the problem manually?
    - List the assumptions you (or others) have made so far and verify them if possible.
2. **Get The Data**
    - List the data you need and how much you need.
    - Find and document where you can get that data.
    - Check how much space it will take.
    - Check legal obligations, and get access authorization if necessary.
    - Create a workspace (with enough storage space).
    - Get the data.
    - Make sure you data are not of insufficient quantity, poor quality, or non representative of the population. Especially validation and test set must be as representative as possible..
    - Convert the data to a format you can easily manipulate (without changing the data itself).
    - Ensure sensitive information is deleted or protected (e.g anonymized).
    - Make a copy of the original dataset and work with this one.
    - Automate as much as possible so you can easily get fresh data.
3. **Explore The Data**
    - Create a Jupyter notebook to keep a record of your data exploration.
    - Study each attribute and its characteristics: name, type, percentage of missing values, noisiness and type of noise, outliers, usefulness for the task basic statistics (mean, standard deviation, type of distribution, etc).
    - For supervised learning tasks, identify the target attribute(s).
    - Sample a test set, put it aside, and never look at it (stratify if needed).

- Visualize the data.
- Study the correlations between attributes.
- Study how you would solve the problem manually.
- Identify extra data that would be useful.
- Document what you have learned.
- Try to get insights from a field expert for these steps.

4. **Prepare The Data**
   - Write functions for all data transformations you apply, for the following reasons:
     - So you can easily prepare the data the next time you get a fresh dataset.
     - So you can apply these transformations in future projects.
     - To clean and prepare the test set.
     - To clean and prepare new data instances once your solution is live.
     - To make it easy to treat your preparation choices as hyperparameters.
   - Clean the data: fix or remove outliers, fill in missing values (e.g with zero, mean, median,etc) or drop their rows (or columns).
   - Feature selection/engineering:
     - Drop the attributes that provide no useful information for the task.
     - Drop variables that have a very high percentage of missing values.
     - Drop variables that have a very low variation (i.e not too much information).
     - Drop variables that have very low correlation with the target.
     - Find variables that are highly correlated with each other (i.e same behaviour), and keep the ones that have higher correlation with the target (drop the other ones).
     - Identify the promising transformations you may want to apply (e.g combine existing features to produce more useful ones or add promising transformations of features).
     - Select best features based on a metric. Either start with one variable and add more (forward selection), or start with all variables and eliminate (backward elimination or recursive feature elimination).
     - Discretize continuous features.
     - Decompose features (e.g categorical, date/time, etc).
     - Standardize or normalize features (feature scaling). Machine learning algorithms don't perform well when the input numerical attributes have very different scales. There are two common ways to get all attributes to have the same scale: normalization and standardization. In normalization the values are shifted and rescaled so that they end up ranging from 0 to 1. We do this by subtracting the minimum value and dividing by the maximum minus the minimum. Standardization first subtracts the mean value (so standardized values always have a zero mean), and then divides by the standard deviation so that the resulting distribution has unit variance. Unlike normalization, standardization does not bound values to a specific range, which may be a problem for some algorithms. However, standardization is much less affected by outliers.

5. **Shortlist Promising Models**
   - If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or Random Forests).
   - Train many quick-and-dirty models from different categories using standard parameters.
   - Measure and compare their performance.
   - For each model, use K-fold cross-validation and compute the mean and standard deviation of the performance measure on the K folds.
   - Analyze the most significant variables for each algorithm.
   - Analyze the types of errors the models make. What data would a human have used to avoid these errors?

- Shortlist the top three to five most promising models, preferring models that make different types of errors.

6. **Fine-Tune The System**
    - You will want to use as much data as possible for this step, especially as you move toward the end of fine-tuning.
    - Fine-tune the hyperparameters using cross-validation.
    - Try ensemble methods. Combining your best models will often produce better performance than running them individually.
    - Once you are confident about your final model, measure its performance on the test set to estimate the generalization error.
    - Don't tweak your model after measuring the generalization error since you would just start overfitting the test set.

7. **Present Your Solution**
    - Document what you have done.
    - Create a nice presentation.
    - Make sure you highlight the big picture first.
    - Explain why your solution achieves the business objective.
    - Don't forget to present interesting points you noticed along the way.
    - Describe what worked and what did not.
    - List your assumptions and your system's limitations.
    - Ensure your key findings are communicated through beautiful visualizations or easy-to-remember statements.

8. **Launch**
    - Get your solution ready for production.
    - Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
    - Beware of slow degradation: models tend to "rot" as data evolves.
    - Measuring performance may require a human pipeline.
    - Also monitor your inputs' quality.
    - Retrain your models on a regular basis on fresh data (automate as much as possible).
    - Keep backups of every model.