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)*

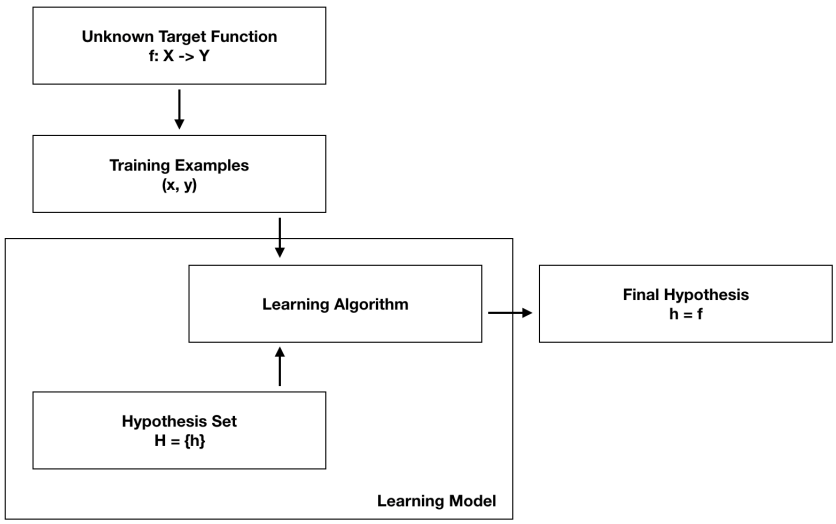
Definition 1.2 (Machine Learning)**.** *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E. (Tom Mitchell)*

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.

In order to formalize it we will be using the following notation throughout the notes:

- 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 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.

Machine learning is a very broad topic with many different branches and applications. In these notes we will cover the most basic branches which are:

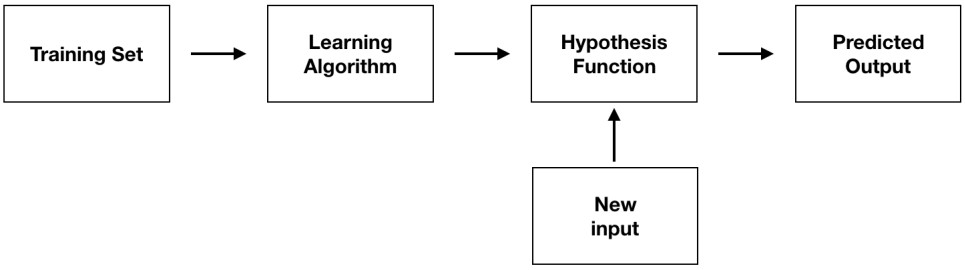
1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning
4. Deep Learning

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 (in case of many features): x_i
- The i'th training example (in case of many training examples): $x^{(i)}$
- The i'th feature of the j'th training example: $x_i^{(j)}$
- Output variables or targets or classes: y
- The i'th output 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).*

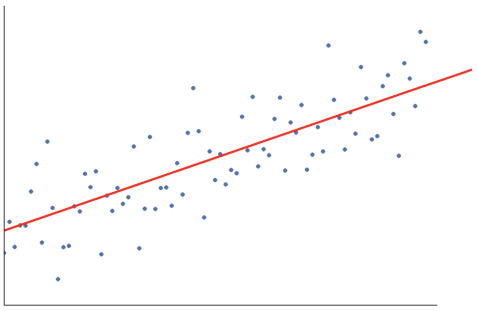
Definition 2.3 (Simple & Multiple Linear Regression)**.** *When there is only one independent variable then the model is called* ***simple linear regression****. For more than one independent variable, the process is called* ***multiple linear regression****.*

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

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

$$h(x) = w_0 + w_1 x_1 + \ldots + w_n x_n$$

where w_0 is is called "bias" and w_i's are called "weights". We usually refer to weights and bias as the "parameters" of the regression 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 linear regression the hypothesis function can be rewritten as

$$h(x) = w_0 x_0 + w_1 x_1 + \ldots + w_n x_n = \sum_{i=0}^{n} w_i x_i$$

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

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

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

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

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:

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

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

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

By substituting the error back to the probability

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

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

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 $\boldsymbol{w}^T \boldsymbol{x}^{(i)}$ and variance σ^2

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

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 it is

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

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

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

Subsequently for the log-likelihood

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

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:

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

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:*

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

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:

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

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

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

2.2 Optimization Techniques

Given the MSE loss function (or any other loss function that we will introduce later on), 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 2.6 (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 loss function, so for the derivative we get:

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

By setting the derivative to 0 we obtain:

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

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 $(m \times (n+1)) \times ((n+1) \times m)$ matrix is, with today's most efficient computer science algorithm, of cubic time complexity! This means that as the dimensions of X increase, 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.

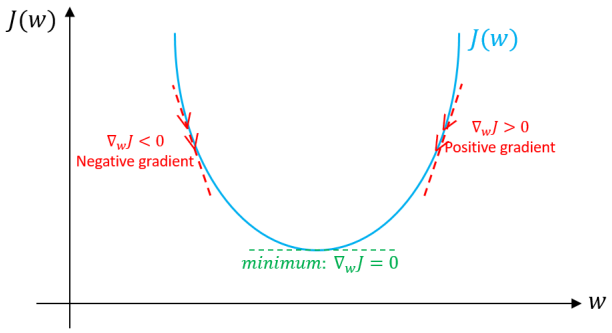
In practise, for the vast majority of any industrial application with large datasets, the normal equation would take extremely, sometimes nonsensically, long. This is the reason why normal equation is almost never used. Now let's move on the the most standard optimization technique used today called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

2.2.2 Gradient Descent

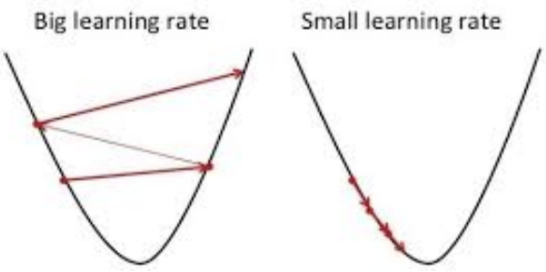
Gradient descent, and all its improvements and alternatives, is the most used optimization technique in machine learning and deep learning. In gradient descent we begin with some random initial values for the parameters and we calculate the value the loss function based on them. Then we update them based on the following relation:

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

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 have researched today!



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

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

Hence the update rule reads:

$$\boldsymbol{w} \coloneqq \boldsymbol{w} - \frac{\alpha}{m}X^T\Big(X\boldsymbol{w} - \boldsymbol{y}\Big)$$

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.

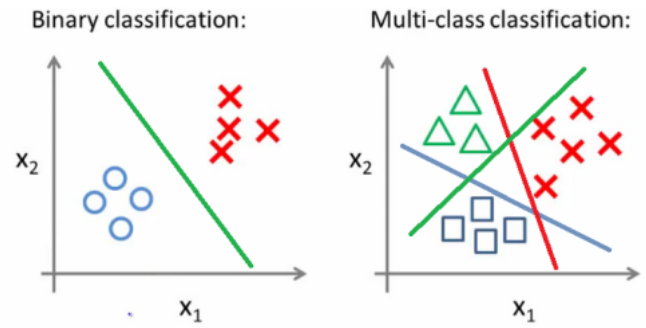
One of the main problems of batch gradient descent is that as the number of training examples grows the dimensions of X grows and using the whole training set for every iteration becomes computationally expensive. Both mini-batch and stochastic gradient descent deal with this problem.

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 2.7 (Logistic Regression)**.** ***Logistic regression*** *is a statistical model that is used to model a binary dependent variable (usually 0 or 1).*

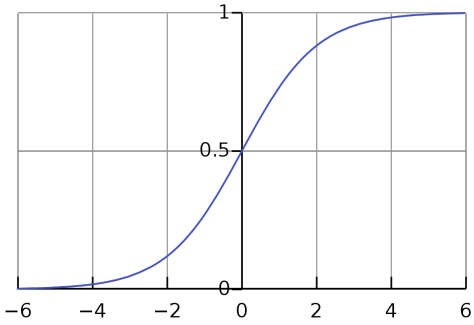
Sometimes logistic regression is called "binary classification" since we try to decide if an input belongs to class 0 or class 1 of the binary output. Wecan also generalize to more than 2 discrete classes where then we have "multi-class classification". For now we will focus on binary outputs.



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:

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

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



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 try to give a different meaning to the hypothesis function.

Namely the intuition is that the sigmoid function will act as a probability measure of the target to belong to class 1. The closest to 0 the sigmoid 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.

As we said, based on this intuition, we can interpret the hypothesis function as the probability of the input to be 1 hence:

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

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 it is

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

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:

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

At this point we can formally define the following function:

Definition 2.8 (Cross Entropy Loss Function)**.** ***Cross entropy loss function*** *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 gonna 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. First, let's calculate the derivative of $J(\boldsymbol{w})$ for logistic regression.

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

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

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

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

$$= -\frac{1}{m}\sum_{i=1}^{m}\Big(\frac{y^{(i)}\cdot(1-h(\boldsymbol{x}^{(i)}) - (1-y^{(i)})\cdot h(\boldsymbol{x}^{(i)})}{h(\boldsymbol{x}^{(i)})\cdot(1-h(\boldsymbol{x}^{(i)}))}\Big)\cdot \nabla_{\boldsymbol{w}}\Big[\frac{1}{1+exp(-\boldsymbol{w}^T\boldsymbol{x}^{(i)})}\Big]$$

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

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

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

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

Hence the update rule reads:

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

Or in vectorized form:

$$\boldsymbol{w} \coloneqq \boldsymbol{w} - \frac{\alpha}{m} X^T \Big(\frac{1}{1 + exp(-X\boldsymbol{w})} - \boldsymbol{y} \Big)$$

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:

$$\boldsymbol{w} \coloneqq \boldsymbol{w} - \frac{\alpha}{m} X^T \Big(h(X) - \boldsymbol{y} \Big)$$

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

2.4 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 2.9 (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:*

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

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, sampling deviation, bias and variance.

Out of all possible quantities that can be defined, MSE, bias and variance are of crucial importance in machine learning, since we use them in order to evaluate how well a mahcine 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 in machine learning case we define the following quantities:

Definition 2.10 (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 2.11 (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 2.12 (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 2.13 (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 $E_{out} \approx 0$.

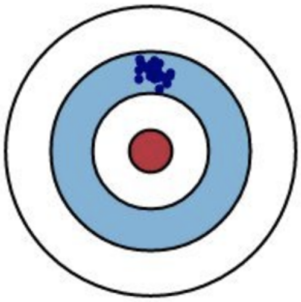
2.5.2 Bias & Variance

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:

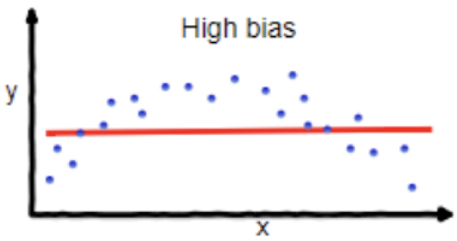
Definition 2.14 (Bias)**.** *We define the* ***bias*** B *of the hypothesis function* h *as the quantity:*

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

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.



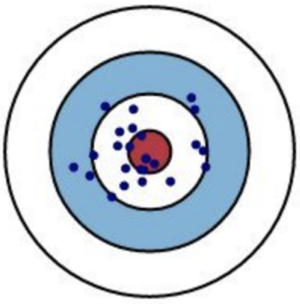
Underfitting is one of the two main problems that a machine learning model can have and it leads to a high in-sample error E_{in} which subsequently leads to a high out-of-sample error E_{out}. Hence even that the problem is coming from the in-sample error it leads to not being able to generalize for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

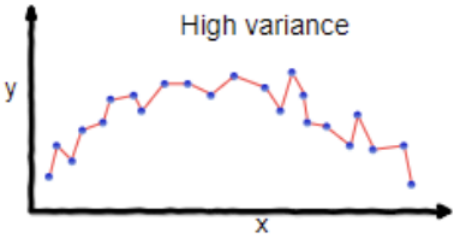
Definition 2.15 (Variance)**.** *We define the* ***variance*** *B of the hypothesis function h as the quantity:*

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

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.

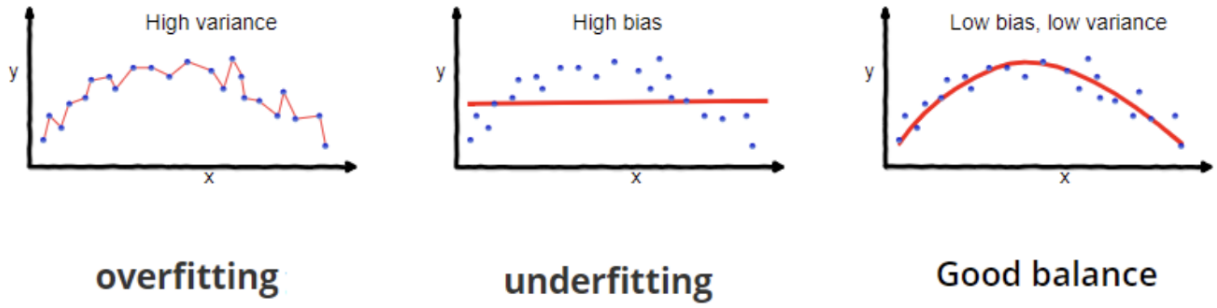


Overfitting leads to a very low in-sample $E_{in} \approx 0$, since it does a very good job on fitting the given data. However it fails to generalize, hence to predict new data, which leads to a very high out-of-sample error E_{out}.

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:

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

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.



In this graph 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!

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 as:

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

In general, for the error function e we use the corresponding loss function J that we used to train the model, 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 evaluation 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 evaluation 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 intro training set and evaluation set. This again is an area of heavy research, but in general in machine learning we usually split them either in a proportion of "70% - 30%" or "80% - 20%" depending on the amount of data. (In all cases the largest proportion goes for training the model). 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: $\{x^{(i)}_{\text{train}}, y^{(i)}_{\text{train}}\}, \quad i = 1, 2, \ldots, m_{\text{train}}$ (70% or 80% of initial dataset)

- Test Dataset: $\{x^{(i)}_{\text{eval}}, y^{(i)}_{\text{eval}}\}, \quad i = 1, 2, \ldots, m_{\text{eval}}$ (30% or 20% of initial dataset)

And subsequently the errors become:

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

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

So for example for linear regression we would have:

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

While for logistic regression we would have

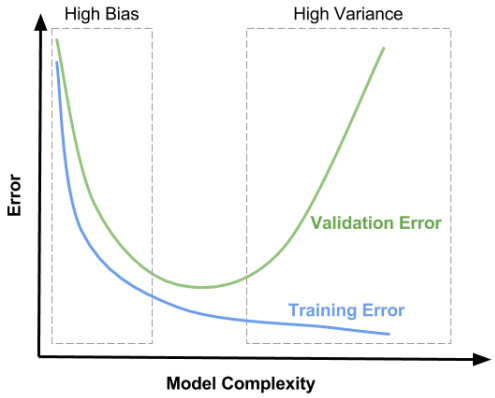
$$J_{\text{train}} = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left(y^{(i)} \cdot \ln h(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \cdot \ln(1 - h(\boldsymbol{x}^{(i)})) \right)$$

and

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

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:



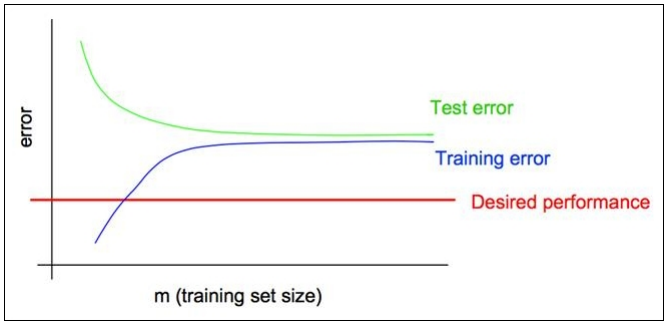
In the high bias area both J_{train} and J_{eval} are high. This means that the model does not fit the training data well hence it fails to generalize. This is the case of underfitting. In the high variance area, J_{train} is low but J_{eval} is high. This means that the model fits the training data 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 2.16 (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.



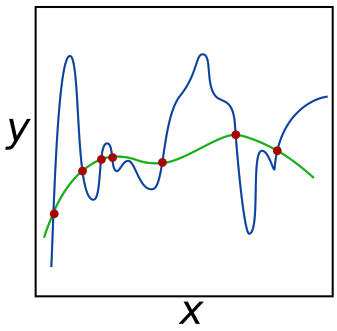
On the other hand when there is a large gap between the two curves we are dealing with the case of high variance and overfitting.

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 (polynomial terms)
  - Increase number of features
- For high variance (overfitting):
  - Decrease model complexity (polynomial terms)
  - 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 2.17 (Regularization)**.** *Regularization is the process of adding information in order to solve an ill-posed problem and to prevent overfitting.*

We will see many different regularization techniques throughout the notes. For now we will start with "Ridge Regression" or "L2 Regularization" and "Lasso Regression " or "L1 Regularization".

2.7.1 Ridge Regression - 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:

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

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

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

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:

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

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 given by (**??**) the Lagrangian reads:

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

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

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

and then the problem is to minimize this loss function which is actually a regression problem. The corresponding regression is called "Ridge regression", 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 $\boldsymbol{w}^T\boldsymbol{x} = 0$. 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:

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

By setting the derivative to 0 we obtain:

$$\begin{aligned}
&\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = 0 \Rightarrow \\
&\frac{1}{2m}\nabla_{\boldsymbol{w}}\Big[\boldsymbol{w}^TX^TX\boldsymbol{w} - 2\boldsymbol{w}^TX^T\boldsymbol{y} + \boldsymbol{y}^T\boldsymbol{y} + \lambda \boldsymbol{w}^T\boldsymbol{w}\Big] \\
&\frac{1}{2m}\Big[2X^TX\boldsymbol{w} - 2X^T\boldsymbol{y} + 2\lambda\boldsymbol{w}\Big] = 0 \Rightarrow \\
&\frac{1}{m}\Big[X^TX\boldsymbol{w} - X^T\boldsymbol{y} + \lambda\boldsymbol{w}\Big] = 0 \Rightarrow \\
&X^TX\boldsymbol{w} - X^T\boldsymbol{y} + \lambda\boldsymbol{w} = 0 \Rightarrow \\
&(X^TX + \lambda I)\boldsymbol{w} = X^T\boldsymbol{y} \Rightarrow \\
&\underbrace{(X^TX + \lambda I)^{-1}(X^TX + \lambda I)}_{I}\boldsymbol{w} = (X^TX + \lambda I)^{-1}X^T\boldsymbol{y} \Rightarrow \\
&\boldsymbol{w} = (X^TX + \lambda I)^{-1}X^T\boldsymbol{y}
\end{aligned}$$

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:

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

Hence the update rule reads:

$$\boldsymbol{w} \coloneqq \boldsymbol{w} - \frac{\alpha}{m}X^T\Big((X + \lambda I)\boldsymbol{w} - \boldsymbol{y}\Big)$$

Of course, L2 regularization can be applied also for the case of logistic regression. More specifically, for cross entropy loss function of logistic regression $J(\boldsymbol{w})$ the Lagrangian reads:

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

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

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

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:

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

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, or L2 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:

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

and for logistic regression:

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

The corresponding regression is called "Lasso regression". As before, we can use normal equation and gradient descent to solve Lasso regression.

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 2.18 (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.*

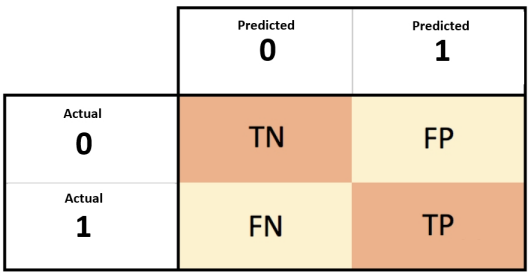
Definition 2.19 (True Negative)**.** ***True negative*** *(TN) also called correct rejection, is the case where the model predicts a negative result when the actual outcome is indeed negative.*

Definition 2.20 (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 2.21 (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 2.22 (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.*



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

Definition 2.23 (Accuracy)**.** ***Accuracy*** *(ACC) is the rate that shows overall how often the model was correct:*

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

Definition 2.24 (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 2.25 (True Positive Rate)**.** ***True positive rate*** *(TPR) also called sensitivity, 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 2.26 (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 2.27 (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 2.28 (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 2.29 (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 2.30 (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 2.31 (Negative Predicted Value)**.** ***Negative predicted value*** *(NPV) also called precision, is the rate that shows how often the model is correct when it predicts negative:*

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

Definition 2.32 (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 2.33 (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:*

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

The coefficient β is chosen such that recall is considered β times as important as precision. The most commonly used value for β is 1, corresponding to the F_1 where precision and recall are weighted equally:

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

Two other 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).

Definition 2.34 (Null Error Rate)**.** ***Null error rate*** *is the rate that shows how often a model would be wrong if it always predicts the most frequent type of outcome (either positive or negative depending on the dataset).*

Definition 2.35 (Cohen's Kappa)**.** ***Cohen's kappa*** *is the rate that shows how much better a model performs compared to a hypothetical model that would pick a category completely randomly.*

2.9 Support Vector Machine

Support vector machine (SVM) is another, more advanced 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.

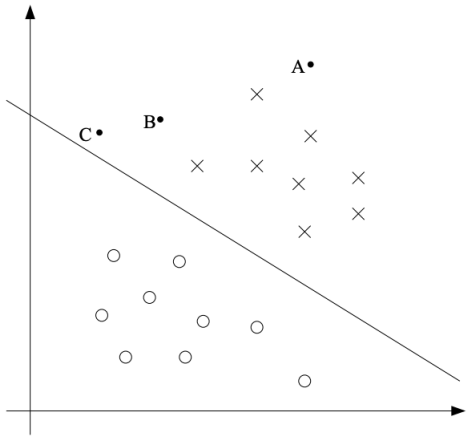
To tell the SVM story, we'll need to first talk about margins and the idea of separating data with a large "gap". Next, we'll talk about the optimal margin classifier, which will lead us into a digression on Lagrange duality. We'll also see kernels, which give a way to apply SVM's efficiently in very 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

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

We would then predict 1 on an input $\boldsymbol{x}$ if and only if $h(\boldsymbol{w}^T\boldsymbol{x}) \geq 0.5$ or equivalently, if and only if $\boldsymbol{w}^T\boldsymbol{x} \geq 0$. Consider a positive training example ($y = 1$). The larger $\boldsymbol{w}^T\boldsymbol{x}$ is, the larger also is $h(\boldsymbol{w}^T\boldsymbol{x})$ 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 $\boldsymbol{w}^T\boldsymbol{x} \gg 0$. Similarly, we think of logistic regression as making a very confident prediction of $y = 0$, if $\boldsymbol{w}^T\boldsymbol{x} \ll 0$. 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 $\boldsymbol{w}^T\boldsymbol{x}^{(i)} \gg 0$ whenever $y^(i) = 1$ and $\boldsymbol{w}^T\boldsymbol{x}^{(i)} \ll 0$ 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 $\boldsymbol{w}^T\boldsymbol{x} = 0$ 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

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

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 2.36 (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*

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

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 $\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b$ 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 $\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b$ to be a large negative number. Moreover, if $\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b \neq 0$, then our prediction on this example is correct. Hence, a large functional margin represents a confident and a correct prediction.

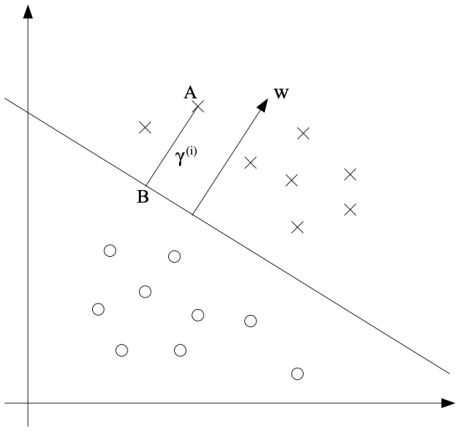
For a linear classifier with the choice of g given above, there's one property of the functional margin that makes it not a very good measure of confidence, however. Given our choice of g, we note that if we replace $\boldsymbol{w}$ with $2\boldsymbol{w}$ and b with $2b$, then since $g(\boldsymbol{w}^T \boldsymbol{x} + b) = g(2\boldsymbol{w}^T \boldsymbol{x} + 2b)$ 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 $\boldsymbol{w}^T \boldsymbol{x} + b$. 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

Definition 2.37 (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 $\boldsymbol{w}^T \boldsymbol{x} + b = 0$. Hence

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

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

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

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. More generally, we define the geometric margin of ($\boldsymbol{w}$, b) with respect to a training example $(x^{(i)}, y^{(i)})$ to be

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

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:

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

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:

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

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 $\mathcal{L}$ turns 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 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.*

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