Introduction to Machine Learning

A self study guide to the main algorithms

Andrés Méndez L.

April 2019

Contents

Chapter 1

Preface

These notes were created as a brief introduction and summary of the main concepts behind Machine Learning. Although I did not go very deep into the mathematics of it, it contains a considerable amount of definitions explained as clearly as I could in order to get a basic understanding of the concepts behind the algorithms used in ML. I do not intend to claim these notes as an original work, but as a review of the contents of many books, lectures and other literature. To create these notes, I used Shalev-Shwartz book *Understanding Machine Learning, from Theory to Algorithms*, and followed very closely the first, more theoretical chapters. I also include material from lectures and YouTube lessons on the topic that will be listed at the end.

Chapter 2

Basics of Learning Theory

2.1 What do we understand by Machine Learning?

Machine Learning (ML), or automated learning, can be define in many ways; as a paradigm, an subfield of A.I., a set of algorithms, a data analysis methodology, etc. However, the main concept behind ML is the idea of programming computers so they can *learn* from a set of input variables. In more technical lingo, the basic premise of machine learning is to build algorithms that, given a set of input variables, can fit a suitable model to predict a relative accurate output for new unseen data.

2.1.1 Types of Learning

Over the last decade, ML has become a widely studied subject, branching into several subfields. In this regard, different types of learning has been proposed, each one dealing with different learning tasks. However, this guide only deals with two types of learning, which enclose most of the tasks we are interested in.

(a) **Supervise Learning:** Input and desired output data are provided as training examples. The acquired expertise is then used to predict outputs for unseen data (test data). Supervised learning problems can be subclassified also as:

(a.1) Classification: predict to which class of objects a certain observation belongs given its different features. Its output is a categorical variable.

(a.2) Regression: looks for a functional relationship between the input set and the output set of the data. Its output is a continuous variable.

(b) **Unsupervised Learning:** There is no distinction between training and test data. The learner processes input data with the goal of coming up with some summary, or compressed version of that data.

(b.1) Clustering: looks for inherent groupings in the data and then it divides it into subsets of similar objects.

(b.2) Association: analyzes data for patterns, or co-occurrence, in a database. It identifies frequent if-then associations, which are called association rules.

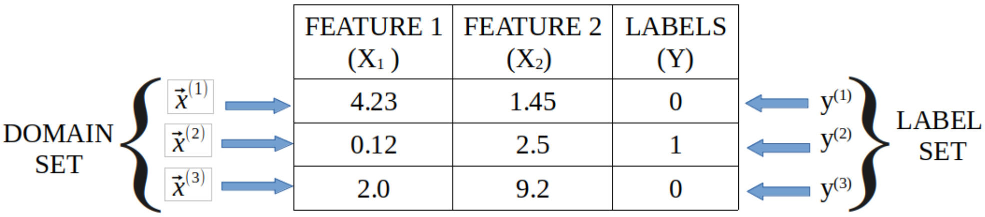
2.1.2 The Statistical Learning Framework

In order to understand what *learning* means in this context, we are going to describe the statistical learning framework, an attempt to model the learning process by which computers can classify or predict new data base on 'previous experience'. In order to simplify the description of the model, let's suppose we are dealing with a classification problem.

The input: In the SLF, this is all the information the learner has access to, which it will use to perform the corresponding task.

- Domain set: Also called *instance space*, is the set $\mathbb{X}$ of objects -or *instances*- we want to classify. The points or elements in this set are represented by a **vector of features** $\{\vec{x} \in \mathbb{X}\}$.
- Label set: Is the set $\mathbb{Y}$ of all possible labels for the elements of the domain set. If the classification is binary, then $\mathbb{Y} = \{1, -1\}$
- Training data: Sequence of instances with their corresponding labels, ordered as a set of tuples $\{(x_i, y_i) \in \mathbb{X} \times \mathbb{Y}\}$.

The output: after 'seeing' the input data, the learner should be able to output a *prediction rule*, a function h (predictor, hypothesis, classifier, etc), that maps a point of the domain set to a label in the label set $h : \mathbb{X} \to \mathbb{Y}$



FEATURE 1 (X_1)	FEATURE 2 (X_2)	LABELS (Y)
4.23	1.45	0
0.12	2.5	1
2.0	9.2	0

We also need to define some concepts to understand how our training data is generated in the first place. We assume that all instances or points in our domain set are generated or sampled from a underlying probability distribution unknown to the learner - exists a **probability distribution** $\mathcal{D}$ **over** $\mathbb{X}$-, and that exists a **correct labeling function** $\{f : \mathbb{X} \to \mathbb{Y} | f(x_i) = y_i\}$, which is unknown to the learner and correspond to the function the learner is trying to figure out by creating the predictor h.

2.1.3 Measure of Success

To measure how good the process of learning was, we define the **error of a classifier** or the error of h, as the probability that, for a random point of the data sampled from $\mathcal{D}$, the prediction rule does not predict the correct label. Mathematically, the error of the prediction rule is given by:

$$L_{(D,f)} \equiv \mathbb{P}_{x \sim \mathcal{D}} \Big[h(x) \neq f(x) \Big] \equiv \mathcal{D} \Big(\{ x : h(x) \neq f(x) \} \Big) \tag{2.1}$$

The subscript $(\mathcal{D}, f)$ indicates that it is measure with respect to the underlying distribution $\mathcal{D}$ and the correct target function f. This measure is also called **the true risk** or **generalization error**.

However, the learning algorithm does not have a way to calculate this error, as the probability distribution, as well as the correct target function, are unknown to the learner. What the learner has it just a random set of instances and their correct labels. Using this input, we define the **training error** as the error of a classifier when predicting labels over training samples:

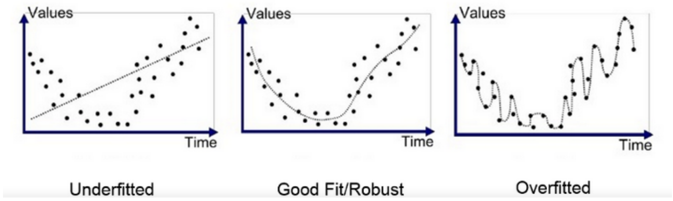
$$L_S(h) \equiv \frac{|\{ i \in [0, m] : h(x_i) \neq y_i \}|}{m} \tag{2.2}$$

where m is the number of instances. This measure is also called the **empirical error** or **empirical risk**, and so we define **Empirical Risk Minimization** (ERM) as the process of choosing a predictor h that minimize $L_S(h)$.

2.2 Overfitting vs Underfitting

Although it might seems a good idea to use ERM the learning process, this could lead to some problems related to the complexity and specificity of our predictive model when we want to generalize to new, unseen data.

If we allow the learner just to minimize the ER without any restriction on our possible predictors, then we might fall into the case where the model predicted fits the training set too well, and it would fail to learn the general tendency of the data. This is called **overfitting**, which leads to a very specific and complex model that would also learn the noise and random fluctuations in the data, giving a very low training error, but performing very poorly in the testing data (high testing error). The other possible scenario is called **underfitting**, and it is the opposite of what we just explained, that is, it correspond to the case in which the prediction rule fails not only to generalize to new data, but it also fails to model the training set, giving a big training error and a big testing error. This results in a very simplistic model with low complexity.



Underfitted

Good Fit/Robust

Overfitted

2.2.1 Avoiding Overfitting - Inductive Bias

We mentioned in the last section that if we use the ERM rule, this might lead to overfitting. To avoid this problem, we can apply some conditions on the ERM paradigm under which the ERM predictor performs well on both the training data and the testing data. The first and most common condition is to apply the ERM learning rule over a restricted search space. This means that the learner should choose before seeing the data, an specific set of predictors from which the final hypothesis will be selected. This set of predictors is also called the **hypothesis class**, denoted by $\mathcal{H}$, such that $\{h \in \mathcal{H} : \mathbb{X} \to \mathbb{Y}\}$. Formally, for a given chosen class $\mathcal{H}$ and training sample S, we will update the rule, so now, in ERM$_{\mathcal{H}}$ the learner uses the ERM rule to choose a predictor h from a previously selected class, with the lowest possible error:

$$ERM_{\mathcal{H}}(S) = \operatorname*{argmin}_{h \in \mathcal{H}} L_S(s) \tag{2.3}$$

When we choose *a priori* a set of classifiers, or an hypothesis class, we are imposing a restriction 'by hand' on the learner. This is what is called **inductive bias**, as we *bias* the learner towards a particular set of predictors beforehand.

We can also impose further restrictions like the inductive bias to avoid overfitting. We can, for example, impose an upper bound on the size or number of predictors h in $\mathcal{H}$. We call these classes **Finite Hypothesis Classes** and we will refer to these from now on, unless we say otherwise.

2.3 A Window to the Real World

In the previous sections we assumed that the points on the training set are independently and identically distributed (i.i.d.) according to the underlying distribution. Although we didn't metion it before, this is a very common assumption in statistical ML and gets the name of **the i.i.d. assumption** (denoted by $S \sim D^m$, where m is the size of the training set). In other words, every $x_i \in S$ is freshly sampled according to $\mathcal{D}$ and labeled according to a correct labeling function f. The training set is then said to be like a windows through which the learner could get information about the 'real world', represented by the unknown distribution $\mathcal{D}$ and f.

We now define **The Realizability Assumption** (condition that eventually will be relaxed). It states that for every hypothesis class, exists a predictor $h^* \in \mathcal{H}$ (a 'perfect' predictor), that will fully minimize the true error $L_{(\mathcal{D},f)}(h^*) = 0$. Therefore, if we consider a random set of points S sampled from D and labeled by f, we also expect that $L_S(h^*) = 0$.

This means that it might be possible to get a perfect classifier h^* if we are lucky enough to get a training set extremely representative of the real distribution from which we sample our data. However, our training set S, which is randomly picked, could be very nonrepresentative of the underlying distribution, and so there is a degree of randomness also in the true error $L_(D,f)(h_S)$. We cannot expect our training set will guide the learner towards a good classifier all the time; it might happen that the training set is such that the predictive model does not look any similar to the real distribution.

To address this randomness in the true risk, we define **the confidence parameter** $(1-\delta)$, as the probability to sample a training set for which $L_{(\mathcal{D},f)}$ is not too large, that is, the probability of getting a representative enough sample. We also define the **accuracy parameter** ϵ, which will parametrize the quality of our predictor. That is, we will interpret the event $L_(D,f)(h_S) > \epsilon$ as a failure of the learner, but if $L_(D,f)(h_S) \leq \epsilon$, then we will consider the predictor as an approximately correct rule.

2.3.1 PAC Learning

In the previous part we study the statistical learning framework, in which the $\text{ERM}_\mathcal{H}$ was the main rule for the learner to produce a prediction rule. Here we will go a little bit further and define a more formal model: the PAC learning model.

It can be proven that for a sufficiently large sample of points, the $\text{ERM}_\mathcal{H}$ rule over a finite hypothesis class, could (with probability $1-\delta$) lead to an approximately correct classifier up to an error of ϵ.

$$L_{(\mathcal{D},f)}(h_S) \leq \epsilon \tag{2.4}$$

Therefore, for finite hypothesis classes, if the training sample is large enough, and if the realizability assumption holds, then we say the predictor is **Probably Approximately Correct (PAC)**.

2.3.2 And then, real life happened: Agnostic PAC

So far, our learning model is a little bit too optimistic for the real world, so we need a more realistic model to describe the learning process. Up to this point, we have made use of the realizability assumption, however, for many real practical application, this is a very strong assumption and it might even fail. For example, we cannot assume all the time that the labels are fully determined by the features of the samples. We could have the case where two instances of our domain set share the same features but differ in their labels! This forces us to relax the realizability assumption and replace the notion of target labeling function f, for a **data-labels generating distribution** $\mathcal{D}$, where $\mathcal{D}$ is now a probability distribution over $\mathbb{X} \times \mathbb{Y}$. That is, $\mathcal{D}$ is a *join distribution* over domain points and labels, composed by two parts: the distribution of instances that are possible to characterize by certain features, $\mathcal{D}_x$, and the conditional distribution of classifying certain instance by the label y given the features x of such instance, $\mathcal{D}((x, y)|x)$.

For $\mathbb{X} \times \mathbb{Y} \sim \mathcal{D}$, we now define the **new true error (risk)** as the probability that h missclasifies an instance when labeled points are sampled accordding to $\mathcal{D}$:

$$L_{\mathcal{D}}(h) \equiv \underset{(x,y)\sim\mathcal{D}}{\mathbb{P}}\Big[h(x) \neq y\Big] \tag{2.5}$$

The empirical risk L_S remains the same as Eq.(2). With these new definitions, our goal now is to find a predictor h that minimezes the true risk $L_{\mathcal{D}}$.
Now that the realizability assumption does not longer holds, we cannot expect to get a predictor with a arbitrarily small error. Instead, the learner is required to achieve a small error relative to the best error achievable by the hypothesis class. In this regard, it is possible to prove that for a binary classification taks, the optimal predictor, that is, the predictor whose error is the minimum given the realizability assumption is no longer valid, correspond to the **Bayes Optimal Predictor**. Formally, given any probability distribution $\mathcal{D}$ over $\mathbb{X} \times \{0, 1\}$, the Bayes Optimal Predictor is given by:

$$f(a, b) = \begin{cases} 1 & \text{if } \mathbb{P}[y = 1|x] \geq 1/2 \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

With this definition, we can generalize our learning model and extend the notion of PAC learnability to a more realistic setup. We say a learner is **Agnostic PAC** when the learner declare success if its error is not much larger than the best error achievble by a predictor from the class $\mathcal{H}$.

$$L_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon \tag{2.7}$$

2.4 Generalized Loss Functions

To generalize our models to a wide range of learning tasks, we also need to generalize the formalism we use to measure success. In the case of binary or multiclass classification, we can measure success as in Equation (5). However, for regression tasks, as we want to find some functional relationshio in the data, it is more convenient to define our measure of success by the expected square difference between true labels and their predicted values.

To include all learning tasks, it is necessary to define a **loss function**. Given a set $\mathcal{H}$, and some domain $\mathcal{Z}$, we say a funcction ℓ is a loss function if $\ell : \mathcal{H} \times \mathcal{Z} \to \mathbb{R}_+$. That is, ℓ is a function that maps a classifier and the pairs (x, y) to a real non-negative real number (the loss for each data-label point). The way in which we measure that loss might vary from problem to problem. In linear regression, for example, the Square Loss is used, while for binary classification is the $0-1$ Loss.

Let's now generalize the true error by a **cost function**(or risk function), given by the expected loss of a classifier respect to the probability distribution $\mathcal{D}$ over $\mathcal{Z}$:

$$L_{\mathcal{D}}(h) = \mathop{\mathbb{E}}_{z \sim \mathcal{D}}[\ell(h, z)] \tag{2.8}$$

Similar than before, the distribution $\mathcal{D}$ is still unknown for the learner, so we define the **empirical cost** (or empirical risk) to be the expected loss over a given sample $S = (z_1 .... z_m)$:

$$L_S(h) \equiv \frac{1}{m} \sum_{i=1}^{m} \ell(h, z_i) \tag{2.9}$$

From now on, our learning paradigm will be based completely on the minimization of cost functions given a training set: Upon receiving a training sample S, the learner evaluates the risk of each $h \in \mathcal{H}$ on that given sample and outputs an element of $\mathcal{H}$ that minimizes this empirical risk, hoping that it is also a minimizer of the true risk.

2.5 Uniform Convergence

The idea behind uniform convergence is to ensure that the hypothesis that minimize the empirical risk will also be a minimizer respect to the real distribution. For that, we need that uniformly over all hypotheses in $\mathcal{H}$, the empirical risk will be close to the true risk.

To elaborate a little more on this, we will say a set S is an **ϵ-representative sample** if,

$$\forall h \in, \mathcal{H}, \; |L_S(h) - L_{\mathcal{D}}(h)| \leq \epsilon \tag{2.10}$$

Then we can guarantee that if a training set S is $\epsilon/2$-representative, then any output of $\text{ERM}_{\mathcal{H}}(S)$ satisfies:

$$L_{\mathcal{D}}(h_S) \leq \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h) + \epsilon \tag{2.11}$$

we then say that an hypothesis class $\mathcal{H}$ has the **uniform convergence property** (UCP) if with probability of at least $1 - \delta$, a training set S is $\epsilon-$representative, guaranteeing that the class will be agnostically PAC learnable.

In summary, if the UCP holds for $\mathcal{H}$, then the empirical risk of hypotheses will faithfully represent their true risk and the hypotheses will be agnostic PAC learnable.

2.6 Bias-Complexity Tradeoff

In Section (1.6) we saw that in order to avoid overfitting we could restrict our serach by choosing an hypothesis class (hopefully finite). This selection, however, it is made before the learner could see the data, so how do we choose such hypothesis class? The answer: we must have some prior knowledge about the task. This can be more formally described by the **No-Free-Lunch Theorem**.

2.6.1 No-Free-Lunch Theorem

This theorem establish that **no universal learner exists**. That means, that there is no learner that can challenge any task without having some prior knowledge about the task. In other words, the theorem states that for binarry classification tasks, for every learner there exists a distribution on which it fails: no learner can succeed on all learnable tasks, it will always be a task on which it fails while other learners succeed.

The No-Free-Lunch Theorem then reflects the need to use our prior knowledge about a learning task if we want to succeed on getting a good model. This can be achieved by restricting our hypothesis class. However, this generates a *tradeoff* between the bias and the complexity in our models.

Let's say the error ϵ of an $\text{ERM}_{\mathcal{H}}$ predictor can be written as the sum of two different errors:

$$L_{\mathcal{D}}(h_S) = \epsilon_{app} + \epsilon_{est} \qquad \text{where: } \epsilon_{app} = \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h) \tag{2.12}$$

Then, the **approximation error** correspond to the minimum risk achievable by a predictor in the hypothesis class. This error only depends on the class chosen. For the case of PAC learning, this error is zero. In the Agnostic PAC, however, the error can be large.

The **estimation error** is the difference between the approximation error, and the error achieve by the predictor chosen. This error exists because the empirical risk is only an estimate of the true risk.

With these two definitions, we can clearly see the **bias-complexity tradeoff**: choosing the hypothesis class to be very rich (small bias), decreases the approximation error as we give the model the freedom to become more complex if necessary on the training set, but it will increase the estimation error as it will lead to overfitting. On the other hand, if we choose a very restricted class (big bias), we might reduce the estimation error but the approximation error would increase and we would face underfitting.

Chapter 3

Minimizing the Cost

Before diving into the many algorithms ML uses, it is important to mention how some of these algorithms actually apply the ERM rule. In the previous chapter we mention that our goal here is to minimeze the true cost function. However, this is kind of impossible to do for most applications because it depends on the unknown probability distribution $\mathcal{D}$ from which our instances are sampled. So in order to accomplish our learning task, we need to put our focus on the empirical cost defined by Equation (), so what the ERM really tells us is the hypothesis that minimizes this function. To successfully apply the ERM rule, however, we need a way of calculating such minimum. Here is where **(Stochastic) Gradient Descent** comes in handy as an optimization procedure in which at each step we improve the solution by taking a step along the negative of the gradient of the function to be minimized.

3.1 Gradient Descent

Let's begin describing the standard gradient decent method for minimizing a differentiable function $f(x)$.

We know that in order to find a (local) minimum of a function, we just need to find the point in which its derivative is zero $df/dx = 0$. This approach, although correct, can become extremely hard to solve for some functions, so we need a better and more flexible solution to find the minimum: Gradient Descent. In this approach, we start in any point x along the curve, and from there we figure out which direction we should take in order to make $f(x)$ smaller. Luckly for us, the derivative of the function at that point indicates the slope of the function, so we just need to make a step to the right (left) if the slope is negative (positive), and repeat this proceadure until the slope is zero, a.k.a. reaching the minimum.

For our learning tasks, we usually will have multivariable functions $f(\vec{x})$.

In this case, the derivative is replace with the *gradient* of a function. For a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$. The gradient of f is defined as:

$$\vec{\nabla}_w f = \left(\frac{\partial f}{\partial w_1}, \dots, \frac{\partial f}{\partial w_d} \right) \tag{3.1}$$

Similar to the derivative in the 1d case, it indicates the direction in which the function increases most quickly, and so the negative of the gradient $-\vec{\nabla}_w f$ gives the direction of steepest descent. So just like before, we can start in a random point $\vec{x}$ and move in the direction of the $-\vec{\nabla}_w f$ to reach the lowest value of $f(\vec{x})$.

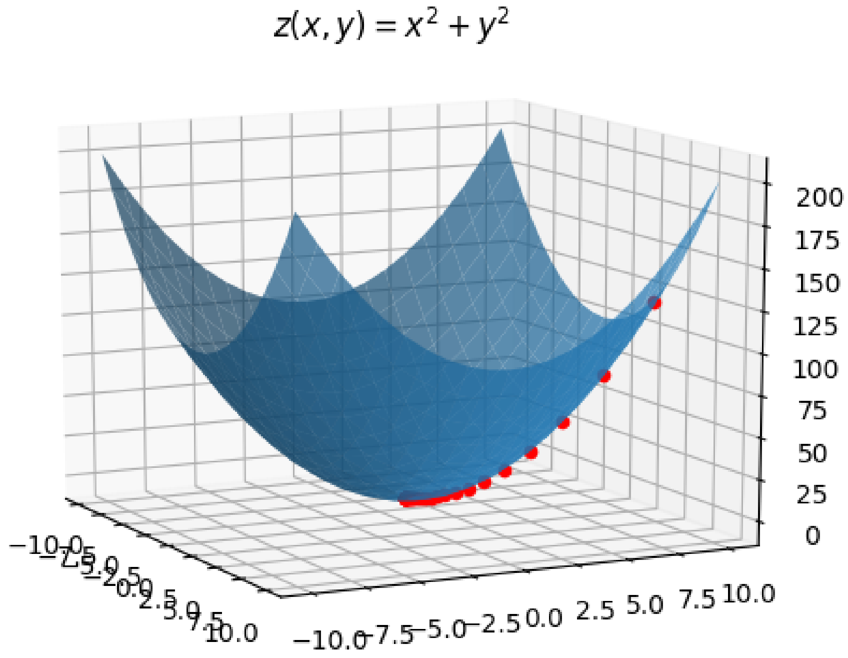


Figure 3.1: Illustration of gradient descent approach for a quadratic function $z = x^2 + y^2$

To implement GD, first we need to start at a random point $\vec{w}^{(0)}$, calculate the negative gradient at such point and then update the weights. This is repeated until reaching the minimum:

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \eta \nabla_w f(\vec{w}^{(t)}) \tag{3.2}$$

where η is the size of the step which is proportional to the gradient, so when we are far away from the minimum the steps are bigger, and when we are closer the steps are smaller. (this prevents overshooting). Eventually, after T iterations, the output of this algorithm can be the last vector $\vec{w}^{(T)}$, the averaged vector, or the best performing vector. This depends on the implementation.

3.2 Stochastic Gradient Descent

Although GD seems a friendly tool, it can be very innefficient once we start working with big sets of data, with hundreds or thousands of features. In those cases, calculating the gradient of the cost of the whole set it is computationally impossible. This is why most of the ML libraries use Stochastic Gradient Descent, SGD.

SGD is used to speed up the process by estimating the gradient $\vec{\nabla} f$ by computing the gradient for a small batch of randomly selected instances (it could be just one!). Recall that our empirical cost function has the form of:

$$L_S(h) = \frac{1}{m}\sum_{i=1}^{m} \ell(h, z_i) \tag{3.3}$$

That is, the cost $L_S(h)$ is a sum over all training instances of the loss $\ell(h, z_i)$ of each instance. So if we were to use the standard form of GD, we would have to compute:

$$\nabla_w L_S(h)) = \frac{1}{m}\sum_{i=1}^{m} \nabla_w \ell(h, z_i) \tag{3.4}$$

Which means that in each iteration we would have to sum over all training instances to compute the gradient. SGD, on the other hand, allow us to approximate this by only selecting small portion of the training set at a time and calculating the step according to this batch . Suppose we have a batch of size $n < m$, then we could approximate the gradient by:

$$\nabla_w L_S(h) \sim \frac{1}{n}\sum_{i=1}^{n} \nabla_w \ell(h, z_i) \tag{3.5}$$

This, of course, it's not be the actual gradient of the cost function, so it is not the most efficient way downhill, but it can be proven that it gives a relatively good approximation, taking less time to compute!

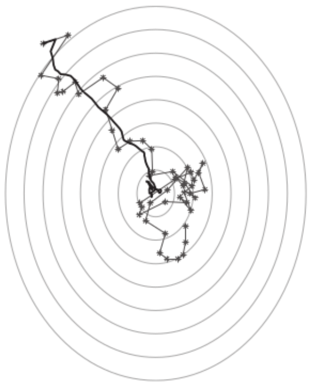


Figure 3.2: Representation of the path to the minimum when using SGD. The solid black line is the average path.

Chapter 4

Classification Algorithms

4.1 Logistic Regression

In this chapter we will study the details behind Logistic Regression, a classifier from the family of linear predictors, one of the most useful and used hypothesis classes. To understand LR, first let's define the class of linear functions as:

$$L_d = \{\vec{x} \rightarrow z = \sum x_i w_i : \vec{w} \in \mathbb{R}^d \text{ and } x_1 = 1, w_1 = b\} \tag{4.1}$$

That is, L_d is a set of functions, where each function is parametrized by a weight vector $\vec{w}$ and a bias b.

In Logistic Regression, the predictor is a function $h : \mathbb{R}^d \rightarrow [0, 1]$. It can be interpreted as the *probability* that a certain instance belong to a class in binary classification tasks. The hypothesis class associated with LR is the composition of a sigmoid function σ over the class of linear functions L_d:

$$H_{sig} = \sigma \circ L_d = \{\vec{x} \rightarrow \sigma(z) | \, \vec{w} \in \mathbb{R}\} \tag{4.2}$$

The sigmoid function or logistic function that it is used as the classifier h is defined as:

$$h_{LR} = \sigma(z) = \frac{1}{1 + \exp{(-z)}} \tag{4.3}$$

So basically, to determine in which class a certain instance should be, we calculate a linear combination of weights $\vec{w}$ and the features $\vec{x}$ associated to that instance and we called that combination $z = \vec{w} \cdot \vec{x}$, and then we use it as the argument of a sigmoid function $\sigma(z)$ which will return the probability of that instance to belong to the class $y = 1$.

However, if we start with random weights, this classifier surely will perform poorly in our training set, so we need to apply ERM rule to find the best classifier. To define a cost function let's start by defining the probability of an instance to be part of one of the classes, given its features and the weights as:

$$\left.\begin{aligned} P(y=1|\vec{x},\vec{w}) &= \sigma(z) \\ P(y=0|\vec{x},\vec{w}) &= 1-\sigma(z) \end{aligned}\right\} \quad P(y|\vec{x},\vec{w}) = \sigma(z)^{y}(1-\sigma(z))^{1-y}$$

A next step to find the cost function is to define the **Likelihood**, that is, the probability of observing the instances of our set S:

$$\mathbb{P}[S=(x_1,...,x_n)] = \prod_i P(y^{(i)}|\,\vec{x}^{(i)},\vec{w}) \tag{4.4}$$

Finally, the cost function for LR will be given by the **Log Likelihood**:

$$L_S = \log\left(\mathbb{P}[S=(x_1,...,x_n)]\right) = \sum_i y^{(i)}\log(\sigma(z^{(i)})) + (1-y^{(i)})\log(1-\sigma(z^{(i)})) \tag{4.5}$$

and the estimator σ of this class will be the **Maximum Likelihood Estimator**, which means that in order to find the best classifier, instead of minimizing our cost, we will maximize it (maximum likelihood):

$$\sigma \in \underset{\mathcal{H}}{\text{argmax}}\ L_S \tag{4.6}$$

4.1.1 Maximizing the Likelihood

In Section () we saw that it is possible to find the minimum of a function (cost function) using SGD. In LR, although we don't want to minimize our cost, we can still use this approach.

To maximize the cost function in LR using the idea of SGD, we can start in a random point over our surface L_S and from there we can move in the direction of the gradient of L_S, that is, in the direction of steepest ascend. This approach is called **Gradient Ascend**, and it works in the same way as GD: in each iteration we must update the weights until we reach the maximum:

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \eta\nabla_w f(\vec{w}^{(t)}) \tag{4.7}$$

Notice that we reversed the sign in the previous equation compared to Equation (). In order to find such gradient we can write:

$$\frac{\partial\sigma(z)}{\partial z} = \frac{\exp(-z)}{(1-\exp(-z)^2} = (1-\sigma(z))\sigma(z) \tag{4.8}$$

Using the previous equation, we can now calculate the gradient of the log likelihood:

$$
\begin{aligned}
(\nabla L_S)_j = \frac{\partial L_S}{\partial w_j} &= \sum_i \frac{y^{(i)}}{\sigma(z^{(i)})} \frac{\partial \sigma(z^{(i)})}{\partial z} \frac{\partial z^{(i)}}{\partial w_j} - \frac{(1-y^{(i)})}{(1-\sigma(z^{(i)})))} \frac{\partial \sigma(z^{(i)})}{\partial z} \frac{\partial z^{(i)}}{\partial w_j} && (4.9) \\
&= \sum_i \frac{\partial \sigma(z^{(i)})}{\partial z} \frac{\partial z^{(i)}}{\partial w_j} \left(\frac{y^{(i)}}{\sigma(z^{(i)})} - \frac{(1-y^{(i)})}{(1-\sigma(z^{(i)})))} \right) && (4.10) \\
&= \sum_i (1-\sigma(z^{(i)}))\sigma(z^{(i)}) x_j^{(i)} \left(\frac{y^{(i)} - \sigma(z^{(i)})}{(1-\sigma(z^{(i)}))\sigma(z^{(i)})} \right) && (4.11) \\
&= \sum_i x_j^{(i)} \left(y^{(i)} - \sigma(z^{(i)}) \right) && (4.12)
\end{aligned}
$$

Therefore, using Equation (), we can write the rule to update the weights and find the maximum likelihood estimator as:

$$
w_j^{(t+1)} = w_j^{(t)} + \eta \sum_i x_j^{(i)} \left(y^{(i)} - \sigma(z^{(i)}) \right) \qquad (4.13)
$$

4.2 K-Nearest Neighbors

The $k-$Nearest Neighbors algorithm for learning is one of the most simplest and easiest to interpret models in ML. The main idea behind this method of classification is to 'memorize' the training set, such it can then predict the label of a new instance based on the labels of the closest $k-$neighbors in the training set. It relies on the assumption that *things that look alike must be alike*, so if an instance have similar features that a point in the training set (and therefore it is close to such point), it isn't too crazy to think that they are very likely to share the same class.

To define how 'close' the different instances are from eachother, we need to define a *metric function* $\rho : \mathbb{X}x\mathbb{X} \rightarrow \mathbb{R}$, that is, a function that takes two vectors or instances and return the distance between those points. If $\mathbb{X} = \mathbb{R}^d$, then the metric function would be the Euclidean distance,

$$\rho(\vec{x}, \vec{x}')) = ||\vec{x} - \vec{x}'|| = \sqrt{\sum_i (x_i - x_i')^2} \tag{4.14}$$

To be able to say which point is closer to a new instance $\vec{x} \in \mathbb{X}$, let's consider a $S = (\vec{x}_1, y_1)...(\vec{x}_m, y_m)$ and reorder the sequence according to the distance $\rho(\vec{x}, \vec{x}_i)$, between the m points in S and the new instance, such that the reordered set is now $S' = \pi_1(\vec{x})...\pi_m(\vec{x})$. If $k = 1$, then we will predict the label of a new instance base on the label of the closest neighbor:

$$h_S(x) = y_{\pi_1(x)}$$

If $k \neq 1$, then the predictor will classify the new point by the majority of labels of the $k-$nearest neighbors. This rule can be generalized by taken the weighted average of the $k-$NN distance from $\vec{x}$:

$$h_S(x) = \sum_i^k \frac{\rho(x, x_{\pi_i})}{\sum_j^k \rho(x, x_{\pi_j})} y_{\pi_i} \tag{4.15}$$

This algorithm requires the entire training data to be stored to be able to compare the distance to a new data point, and so each time we want to predict a label, the algorithm needs to san the entire data set, calculate the distance, and find the neighbors. This leads to expensive computation at test time.

4.3 SVM

Support Vector Machine (SVM) is one of many algorithms for binary or multilabel classification problems. SVM works by drawing a decision boundary that separates the different objects in a set and labels them based on the side the objects are respect to that boundary.

Consider a set of points with positive and negative examples, $\mathbb{Y} = \{+, -\}$, that are linearly separable. What SVM does is to draw a decision boundary given by an optimal straight line between the objects with different labels. The line that is drawn by the algorithm is the one that maximize the width of the 'street' that separates the positive examples from the negative examples. This is sometimes called the *widest street approach.*

To understand how this line is actually chosen and what do we mean by vector support, lets define a vector $\vec{w}$ pointing perpendicular to the line of the street and with a length $\|w\|$ (yet to be found), and an unkown vector $\vec{u}$ as in Figure ().

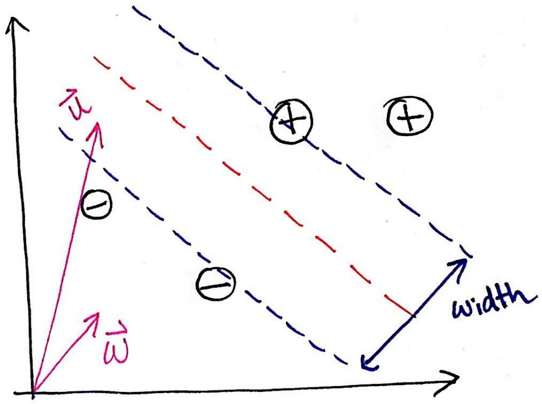


Figure 4.1: SVM decision boundary.

The vector $\vec{u}$ here represents a vector pointing to the new instancec we wish to classify, so our decision rule should return whether the vector is on the left/right side of the street, so our **decision rule** could be:

$$\text{If} \qquad \vec{w} \cdot \vec{u} \geq c \iff \vec{w} \cdot \vec{u} + b \geq 0 \qquad \text{then } y_u = + \tag{4.16}$$

In other words, if the length of the projection of $\vec{u}$ over $\vec{w}$ is greater than a certain constant c, then we can say the point is on the right side of the street. We then say $\vec{w} \cdot \vec{u} + b$ is a measure of distance from the central line of the street. However, we still don't know which b and $\vec{w}$ to used, and so we don't have enough constrains to find these variables.

To calculate b, $\vec{w}$, we will impose the following additional constrains:

$$\vec{w} \cdot \vec{x}_+ + b \geq 1 \tag{4.17}$$

$$\vec{w} \cdot \vec{x}_- + b \leq -1 \tag{4.18}$$

That is, to classify a new point, the constrain function given by Equation () should hold, that is, 'the distance' from the decision boundary should be greater than zero, while for examples already classified it should be equal or greater than 1. For convinience let's define a function y_i:

$$y_i = \begin{cases} +1 & \text{for positive examples} \\ -1 & \text{for negative examples} \end{cases} \tag{4.19}$$

And multiplying the constrains (), we can get a unique equation in terms of y_i:

$$y_i(\vec{x}_i \cdot \vec{w} + b) - 1 \geq 0 \tag{4.20}$$

where the null case its realized when the points are in the extremes of the street (the gutter). This means, the gutter of the street is define by those points for which $y_i(\vec{x}_i \cdot \vec{w} + b) - 1 = 0$. These are the points $\vec{x}_\pm$ of Figure (), also called **Support Vectors**.

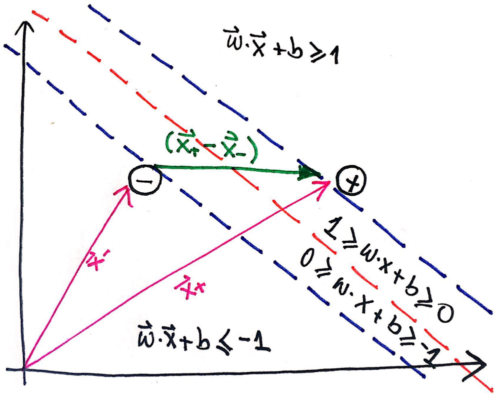

$\vec{w} \cdot \vec{x} + b \geq 1$
$(\vec{x}_+ - \vec{x}_-)$
$-$
$+$
$1 \geq w \cdot x + b \geq 0$
$0 \geq w \cdot x + b \geq -1$
$\vec{x}_-$
$\vec{x}_+$
$\vec{w} \cdot \vec{x} + b \leq -1$

Figure 4.2: SVM decision boundary and support vectors.

We can now define the width of the street. For that, we need the difference of the support vectors projected along a unit vector perpendicular to the decision boundary:

$$\text{width} = (\vec{x}_+ - \vec{x}_-) \cdot \frac{\vec{w}}{\|w\|} \tag{4.21}$$

Using Equation () for the support vectors $\vec{x}_\pm$, we can rewrite this as,

$$\text{width} = \frac{2}{\|w\|} \tag{4.22}$$

So just using the constrains we imposed before we obtained a simple expression for the width of the street, which we would like to maximize:

$$\text{max width} = \max \frac{2}{\|\vec{w}\|} = \max \frac{1}{\|\vec{w}\|} = \min \|\vec{w}\| = \min \frac{\|\vec{w}\|^2}{2} \tag{4.23}$$

To minimize this function under the constrains we imposed before, we can use Lagrangian multipliers. Let's write the Lagrangian for finding the extreme of this function:

$$L = \frac{1}{2}\|\vec{w}\|^2 - \sum_i \lambda_i \left[y_i(\vec{w} \cdot \vec{x}_i + b) - 1 \right] \qquad , \ \lambda_i \geq 0 \tag{4.24}$$

The derivative of the Lagrangian is,

$$\frac{\partial L}{\partial w_j} = w_j - \sum_i \lambda_i y_i x_j = 0 \longrightarrow \vec{w} = \sum_i \lambda_i y_i \vec{x}_i \tag{4.25}$$

$$\frac{\partial L}{\partial b} = -\sum_i \lambda_i y_i = 0 \longrightarrow \sum_i \lambda_i y_i = 0 \tag{4.26}$$

This tells us that the normal vector $\vec{w}$, also called **decision vector**, is a linear combination of the sample vectors! Let's use Equation () back into our Lagrangian,

$$L = \frac{1}{2}\sum_{i,j} \lambda_i \lambda_j y_i y_j \vec{x}_i \cdot \vec{x}_j - \sum_i \lambda_i y_i \vec{x}_i \cdot \left(\sum_j \lambda_j y_j \vec{x}_j \right) - \sum_i \lambda_i y_i b + \sum_i \lambda_i \tag{4.27}$$

$$= \sum_{i,j} \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j \left(\vec{x}_i \cdot \vec{x}_j \right) \tag{4.28}$$

So now the optimization problem has been reduced to find the extremes of the Lagrangian with the form of Equation (), and even though it cannot be solved analytically for large sets, we can see that the whole procedure now depends only on the dot product of pairs of sample vectors.

Finally, let's use the expression we found for the decision vector into the decision rule equation. Now it reads,

$$\text{If} \quad \sum_i \lambda_i y_i \vec{x}_i \cdot \vec{u} + b \geq 0 \quad \text{then } y_u = + \tag{4.29}$$

The class to which the unknown vector $\vec{u}$ belongs, also depends on the dot product with the sample vectors! Using Equation (), the SVM predictor h is given by,

$$h_S = \text{sign} \left(\sum_i \lambda_i y_i \vec{x}_i \cdot \vec{u} + b \right) \tag{4.30}$$

In summary, SVM algorithms deal with an optimization problem of finding the extreme of a function with dependency on the dot product of the sample vectors to find the Lagrange multipliers. This defines the decision boundary and the decision rule applied to new sample vectors as given in Equation ()

4.3.1 Kernel Methods

We already saw that for linearly separable scenarios, SVM draws a decision boundary based on the support vectors that maximizes the width of the street (margin). However, this approach fails when the set of points are not separable. In order to used the SVM algorithm in such cases we need to introduce non-linear transformations.

To work in the non-separable case, instead of working in the $\mathbb{X}$ space, one can go to a higher dimensional space $\mathbb{Z}$ through a non-linear transformation. Let's denote our transformation $\psi(x)$, such that,

$$\psi : \mathbb{X} \longrightarrow \mathbb{Z}$$

Now using this transformation, we can take all vectors $\vec{x} \in \mathbb{X}$ to $\vec{z} \in \mathbb{Z}$ and if the transformation is useful, then the points might be linearly separable in the new space, and we could apply SVM in $\mathbb{Z}$ to get an optimal classifier. This process is show in Figure () below.

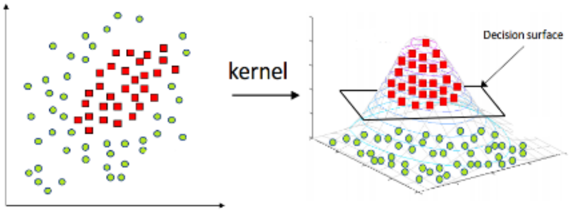


Figure 4.3: Non linear transformation ψ applied to a non-separable set of points.

Using SVM in the new space correspond to apply the same method already discussed but with the transform vectors. Let's call $\vec{z} = \psi(\vec{x})$, so to find a decision boundary in $\mathbb{Z}$ means to solve Equation () and Equation () in the new space:

$$L = \sum_{i,j} \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j \left(\vec{z}_i \cdot \vec{z}_j \right) \tag{4.31}$$

$$\text{If} \quad \sum_i \lambda_i y_i \vec{z}_i \cdot \psi(\vec{u}) + b \geq 0 \quad \text{then } y_u = + \tag{4.32}$$

As we can see, the optimization problem as well as our decision rule don't depend explicitly of the transformation, but on the inner product on the new space. So given two sample vectors $\vec{x}, \vec{x}'$, we only need to worry to find a real number $\vec{z} \cdot \vec{z}'$, and not the actual vetors $\vec{z}, \vec{z}'$.

A **Kernel**, $K(\vec{x}, \vec{x}')$ is then defined as the inner product in the $\mathbb{Z}$ space:

$$\vec{z} \cdot \vec{z}' = K : \psi(\mathbb{X}) \times \psi(\mathbb{X}) \to \mathbb{R}$$

Unfortunally, this procedure requires to transform the vectors in $\mathbb{X}$ into a higher dimensional space to take the inner product, which can be extremely inefficient and difficult if the dimension of $\mathbb{Z}$ is too large, or if $\mathbb{Z}$ is an infinite dimensional space. Luckly, there is a *trick* that allow us to skip some of these steps!

The **Kernel Trick** allow us to compute the inner product in the $\mathbb{Z}$ space, $K(\vec{x}, \vec{x}')$, without transforming the sample vectors $\vec{x}$, provided that $K(\vec{x}, \vec{x}')$ can be map to an inner product in some $\mathbb{Z}$ space. That is, given the kernel K that is the inner product in some existing space $\mathbb{Z}$, we can think of it just as a function of $\vec{x}$ in $\mathbb{X}$, without worrying about what the transformation actually is. This allow us to make all the computations needed to apply SVM only in $\mathbb{X}$.

To get the final predictor that the SVM returns after using the Kernel trick, we only need to express our equations in terms of K,

$$h_S = \operatorname{sign}\left(\vec{w} \cdot \vec{z} + b\right) = \operatorname{sign}\left(\sum_i \lambda_i y_i K(\vec{x}_i, \vec{z}_u) + b\right) \tag{4.33}$$

where for any support vector $\vec{x}_m$,

$$b = y_m - \sum_i \lambda_i y_i K(\vec{x}_i, \vec{x}_m) \tag{4.34}$$

Some examples of valid Kernels are:

- **Polynomial Kernel:** The k-degree polynomial kernel is defined as,

  $$K(\vec{x}, \vec{x}') = \left(1 + \vec{x} \cdot \vec{x}'\right)^k \tag{4.35}$$

  which correspond to the inner product respect to the transformation $\psi : \mathbb{R}^n \to \mathbb{R}^{(n+1)^k}$

- **The Gaussian Kernel:** Given a scalar $\sigma > 0$, the kernel is defined as,

  $$K(\vec{x}, \vec{x}') = e^{-\frac{\left\|\vec{x}-\vec{x}'\right\|}{2\sigma}} \tag{4.36}$$

  which correspond to the inner product in an infinite dimensional space.

4.4 Decision Trees

A Decision Tree (DT) is a kind of predictor h that predicts the label of an instance by traveling along a tree structure that will split data points. Every DT have three main components: a Root node, which have access to the whole training set and it makes the first decision in the tree, to then produce a branch which will go to some other child nodes where other decision will be made to split even more the data, to finally get to the leaf node, the end of the tree where no more splitting happens.

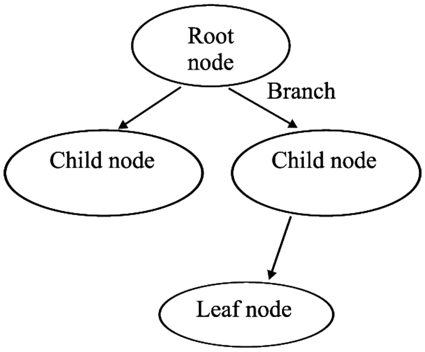


Figure 4.4: Structure of a DT

Depending on the structure we choose for a DT and our data, the output could be the category itself or the probability of being part of certain category at the leaf nodes, which then we can use it as a predictor for the label.

4.4.1 Constructing a DT

To understand how trees are constructed, we need to understand first how the attributes are chosen in each node to split the data. After scanning all attributes, a DT selects the one that splits the examples into subsets that are *ideally* all positive or all negatives.

To measure how 'good' an attribute is and select it as a node, a DT uses the concept of **Entropy**, which is a measure of *information*. For a probability distribution π, the entropy is defined as,

$$H(\pi) = -\sum \pi \log \pi \tag{4.37}$$

In the particular case of a binary classification with a training set containing p positive examples and n negative examples, we have:

$$H\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n} \tag{4.38}$$

Consider now a chosen attribute A, with K possible values that divides the training set in $E = E_1...E_K$ subsets. For each attribute A the **Expected Entropy** (EH) is calculated. This quantity is equivalent to the Expected Utility.

$$EH(A) = \sum_{i}^{k} \frac{p_i + n_i}{p + n} H\left(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}\right) \tag{4.39}$$

So we are weighting the utility or *entropy* of each child node with the probability of the samples to end in such node. Here, the i index refers to the child node, while $(p + n)$ in the denominator is the total number of samples in the parent node.

Additionally, we define the **Information Gain** (I), also known as the reduction in entropy. It tells us how much the entropy gets reduced by making a certain decision. The more we reduce the entropy, the better the decision is.

$$I(A) = H\left(\frac{p}{p + n}, \frac{n}{p + n}\right) - EH(A) \tag{4.40}$$

A DT chooses the attribute with the largest information gain, that is, if the entropy of the child nodes $EA(H)$ is big, then it means $I(A)$ is small, and therefore the entropy hasn't change much (the attribute A didn't reduce the entropy), whereas if $EA(H)$ is small, it means the entropy dropped, and so we get more information (bigger $I(A)$). This procedure is repeated every time a DT split the data and the rest of the structure gets build recursively.

But what if the data is not categorical but continuous? In this case, the features don't have attributes in which we could easily split our data. One way of creating attributes for continuous variables is to project the points on one axis creating a binary splitting point (or decision), S_j. From there, the procedure is exactly the same as before, and for each splitting we can calculate the Information Gain as,

$$I_j = H(S_j) - \sum_{i \in \{L,R\}} \frac{|S_j^i|}{|S_j|} H(S_j^i) \tag{4.41}$$

There are many ways of creating the splits, but the more common ones are to choose randomly different points to project them and create a split, or to create a split line for each point in the data by projecting them into the different axes and then comparing their information gain.

4.4.2 Random Forest

Random forest is an algorithm that creates many decision trees and average their results in order to return a strong predictor. RF works by shuffling and sampling $m \leq n$ points from a training set S and creates a **random tree** based on that data. This process is called *Bootstrapping*, which means that each tree is trained on a slightly different version of the data, which helps to ensure the trees are uncorrelated.

A random tree is no very different of the DTs explained in the last section. The main difference is that a random tree, in order to create a node, instead of comparing all the possible features in terms of their information gain, it selects and compares only two features randomly sampled. After randomly picking such features, the process is repeated in each node until reaching the end of the tree, splitting the points in the same fashion as for a regular DT. A random forest is then just an ensemble of random trees, and the predictor that the algorithm returns is the average of all the decisions made by each random tree. That is, for a new instance, we evaluate its label in each of the random trees created, and from those individual outcomes a decision is made by the majority of votes or the average probability.

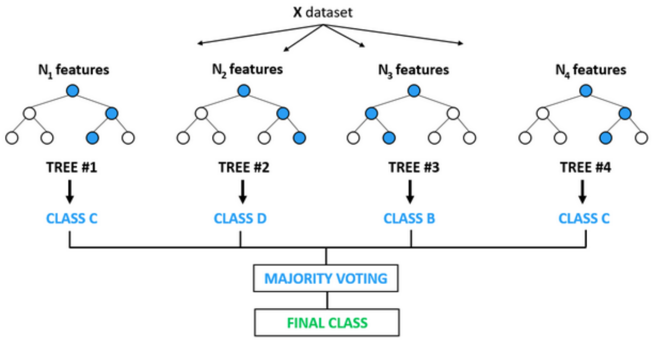


Figure 4.5: Random Forest representation for 4 random trees when depth = 3.

Chapter 5

Regression Algorithms

5.1 Linear Regression

Linnear Regression is commonly used in statistic modeling to find relationships between some *explanatory* variables and some real valued outcome y. If we formulate it as a learning algorithm, we would say that the domain set $\mathbb{X}$ is a subset of $\mathbb{R}$ and the label set $\mathbb{Y}$ is the set of real numbers. We expect the learner to return a linear function $h : \mathbb{R}^d \to \mathbb{R}$ that gives us the best approximation on the relationship between our variables.

The hypothesis class in this case, similarly as what we did in Logistic Regression, is simply the set of linear function,

$$\mathcal{H}_{reg} = L_d = \{\vec{x} \to \vec{w} \cdot \vec{x} + b : \vec{w} \in \mathbb{R}^d, b \in \mathbb{R}\} \tag{5.1}$$

To decide which line from all the ones in $\mathcal{H}$ is the most optimal to solve our problem, we need to define the loss function of linear regression,

$$\ell_{reg} = \big(h(x) - y\big)^2 \tag{5.2}$$

so that the Cost function we would like to minimize to find the optimal solution has the form of:

$$L_S(h) = \frac{1}{m} \sum_i \big(h(\vec{x}_i) - y_i\big)^2 \tag{5.3}$$

For Linear Regression problems, finding the minimum of the cost can be done analytically. This is called **Least Squares Problem**. If we consider $x_1 = 1$ and $w_1 = b$, we can rewrite Equation (), such our prediction function and the cost can written in matrix form:

$$\hat{y}^{(i)} = \vec{x}^{(i)} \cdot \vec{w} = \sum_j x_j^{(i)} w_j \longrightarrow \vec{\hat{y}} = X\vec{w} \tag{5.4}$$

$$L_S(h) = \frac{1}{m} \sum_i \left(\vec{x}^{(i)} \cdot \vec{w} - y^{(i)}\right)^2 \longrightarrow L_S(h) = \big(\vec{y} - X\vec{w}\big)^T \big(\vec{y} - X\vec{w}\big) \tag{5.5}$$

Deriving the previous expression and setting it to zero to find its extreme, we have,

$$\frac{\partial L_S}{\partial w_k} = \frac{2}{m}\sum_i \left(\sum_j X_{ij}w_j - y_i\right) X_{ik} = 0 \tag{5.6}$$

$$\sum_{i,j} X_{ij}w_j X_{ik} = \sum_i y_i X_{ik} \tag{5.7}$$

$$\sum_{ij} X_{ji}(X^T)_{ki}w_k = \sum_i (X^T)_{ki}y_i \tag{5.8}$$

$$\sum_j \left(X^T X\right)_{jk} w_k = \left(X^T \vec{y}\right)_k \tag{5.9}$$

$$X^T X\vec{w} = X^T\vec{y} \tag{5.10}$$

Therefore, the optimal weights that define the line of Linear Regression (aka our predictor), is given by:

$$\vec{w} = \left(X^T X\right)^{-1} X^T\vec{y} \tag{5.11}$$

and our predictor for an observation (i) with $n-1$ features will be then,

$$h(\vec{x}^{(i)}) = \sum_{j=1}^{n} x_j^{(i)} w_j + w_0 \tag{5.12}$$

Although this procedure can be done analytically, it turns out to be very expensive (computationally speaking) when the number of samples and the number of features is too big, because it requires to find the inverse of the matrix $X^T X$. Because of this, the way of finding the minimum could be in these cases SGD, because it only requires evaluating the derivative of the cost for a mini batch of samples and update the weights.

5.1.1 Dealing with non linearity

The previous predictor was only useful in case our data is linearly correlated. However, even when we are dealing with a non linear problem, we can still use all the machinery associated with linear regression. To do so, we introduce **polynomial predictors**. Such predictors have the form

$$h(\vec{x}^{(i)}) = \phi(\vec{x}^{(i)}) \cdot \vec{w} + \epsilon \tag{5.13}$$

Now $\phi(\vec{x}^{(i)})$ doesn't have the same dimension as the number of features as it was the case for linear regression, but its dimension depends on the degree of the polynomial we choose.

For instance, to get the Linear Regression case for a sample of n features, our polynomial transformation should be:

$$\phi(\vec{x}^{(i)}) = [1,\, x_1^{(i)}, ..., x_n^{(i)}] \tag{5.14}$$

However, if we want to fit a quadratic polynomial to our points, we would have to include all the quadratic terms and the cross terms between all the features!. For simplicity, let's take the 2d case, that is, $n = 2$ features:

$$\phi(\vec{x}^{(i)}) = \left[1, x_1^{(i)}, x_2^{(i)}, \left(x_1^{(i)}\right)^2, \left(x_2^{(i)}\right)^2, x_1^{(i)} x_2^{(i)}\right] \tag{5.15}$$

and the set of weights we need to consider now it is also larger than before:

$$\vec{w} = [w_0,\, w_1,\, w_2,\, w_3,\, w_4,\, w_5] \tag{5.16}$$

So the larger the degree of our polynomial, the more weights and parameters we would have to deal with. This makes this approach not very effective, and it is not the way people typically solve non-linear problems. Nevertheless, in some cases it could be useful if the polynomial degree is not very large, and it's implementation is the same as for a regular linear model.

5.1.2 Kernel Regression

Just like we can use polynomials together with the linear regression machinery, we can also introduce a set of any basis functions, these functions are called **kernels** $K(\vec{x}^{(i)})$. If the dimension of the space span by the functions is d, then we have a transformation of the form,

$$\phi(\vec{x}^{(i)}) = \left[K(\vec{x}^{(i)})_1,\, K(\vec{x}^{(i)})_2,\, ...,\, K(\vec{x}^{(i)})_d\right] \tag{5.17}$$

for example, we could use a gaussian kernel $K(\vec{x}) = exp\left(-\left\|\vec{x} - \vec{\mu}\right\|/2\lambda\right)$

Chapter 6

Regularization

6.1 Ridge Regression

In Chapter () we saw that Least Squares could be solved analytically, however, this required the inverse of a matrix which in most cases will be too large to handle computationally, in such cases, SGD could be used to find the minimum. However, there is another way to handle such inversion by means of **Tikhonov regularization**. This regularization technique works by adding a new term to the diagonal that allows us to find an approximation for the inverse of the previously ill define matrix.

$$\vec{w} = \left(X^T X\right)^{-1} X^T \vec{y} \longrightarrow \left(X^T X + \delta^2 \mathbb{I}\right)^{-1} X^T \vec{y} \tag{6.1}$$

where $\delta^2 \mathbb{I}$ is the regulator term. Furthermore, it can be shown that such expression can be obtained from the minimization of the following function:

$$\left(\vec{y} - X\vec{w}\right)^T \left(\vec{y} - X\vec{w}\right) + \delta^2 \|w\|^2 \tag{6.2}$$

That is, instead of doing Least Squares, we can minimize the cost function plus a second term, a **regularization function** f_{reg}, making this approach equivalent to the Tikhonov regularization method. When the regularization function is chosen to be proportional to the norm square as in this case, we call this **Ridge Regression**.

Looking at Equation (), we can see that Ridge Regression is the same as solving an optimization problem with Lagrange multipliers. In other words, what we are doing here is finding the extremes of a function whose parameters are constrained by a certain function $C(\delta) = 1/\delta$. Formally,

$$\min \left(\vec{y} - X\vec{w}\right)^T \left(\vec{y} - X\vec{w}\right) + \delta^2 \|w\|^2 \iff \min_{\|w\|^2 \leq C} \left(\vec{y} - X\vec{w}\right)^T \left(\vec{y} - X\vec{w}\right) \tag{6.3}$$

A **Regularized Cost Function** is then defined as the original Cost Function together with a regularization function which in this case is,

$$J(w) = L_S + f_{reg}(w) = (\vec{y} - X\vec{w})^T (\vec{y} - X\vec{w}) + \delta^2 \|w\|^2 \tag{6.4}$$

If we minimize only f_{reg}, we would have to set $w = 0$, but this could lead to a L_S that could still be large. On the other hand, if we minimize only L_S, then now f_{reg} could be large. Therefore, in order to minimize the whole expression, there is a *trade-off* between these two terms.

Graphically, for a 2d Least Squared regularized case, the cost function L_S is a quadratic function whose level curves are ellipsis, while the regularization function f are circles centered at the origin, with a radius $r = C(\delta)$. We plot both at the same time below:

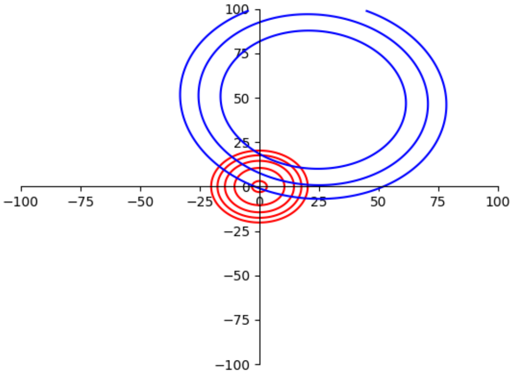


Figure 6.1: Regularization function in red for different values of δ and level curves of the cost function in blue.

In the previous plot, each contour in blue represent a constant cost and each circle in red represent the constrain with different values of δ. When finding the minimum, what we are doing is to find the point where the contours of L_S intersect a red circle of a fixed radius (fixed constrain).

From Figure () we can also see that the constrain forces the algorithm to select smaller values for the weights, which means our model is less complex, and therefore, less likely to have overfitting. This type of function is called a *soft regulator*, because when the features are not important, it tends to make them even smaller, but not identically zero. This, however, depends on the value we choose for δ, and the only way of finding it is through cross validation.

6.2 Lasso Regression

When we choose a L_1-norm as a regularitor function instead of the one we used before, we call it **Lasso Regression**. Formally, the regularized cost function in this case is:

$$J(w) = L_S + f_{reg}(w) = \left(\vec{y} - X\vec{w}\right)^T \left(\vec{y} - X\vec{w}\right) + \delta^2 \|w\|_1 \tag{6.5}$$

The L_1-norm in this case is a *hard regulator*, because for features that are not important, it tends to make the weights identically zero as we can see in Figure (),

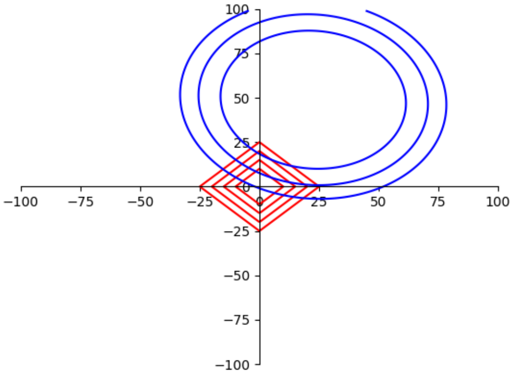


Figure 6.2: Curves representing our constrain in red and level curves of the cost function in blue.

Chapter 7

Cross Validation

Chapter 8

Neural Networks

An artificial Neural Network (NN) is a model of computation inspired by the structure and the way real neural networks in the brain work. A biological brain is composed by millions of different groups of neurons, capable of store information, process it, and learn from it. Biological neurons are connected in many groups, passing information from one neuron to the next through axons and dendrites, structures that channel the potential differences that trigger the neuron making it to *fire* (activating it).

To understand how NNs try to emulate this process, we need to understand what is an **artificial neuron**. The most simple artificial neuron modeled is called **Perceptron**. A perceptron works by taking several binary inputs $\{x_1, ..., x_n\}$, and it returns a single binary output (a yes/no decision). To compute this output, a **weight** has to be assign to each input, denoting the importance of such value in the computation. The output is then determined by a weighted sum of the inputs and a threshold value:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq \text{threshold} \\ 1 & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases} \tag{8.1}$$

This sort of model, where a decision is made based on the weight we assign to the evidence we have (input), is represented graphically as follows,

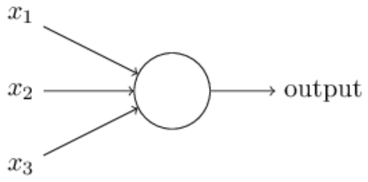


Figure 8.1: Representation of the Perceptron artificial neuron model

To make the connection with the previous chapters and other learning algorithms, we can rewrite our decision rule () to have a more familiar form,

$$\text{output} = \begin{cases} 0 & \text{if } \vec{w} \cdot \vec{x} + b \leq 0 \\ 1 & \text{if } \vec{w} \cdot \vec{x} + b > 0 \end{cases} \tag{8.2}$$

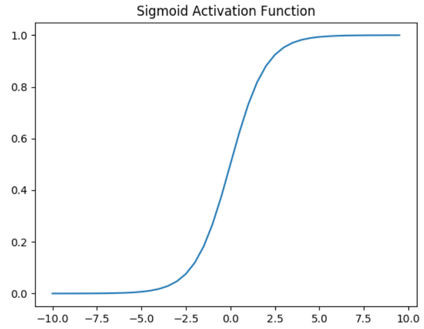
Where b is the *bias*, a measure of how easy it is to get the perceptron to fire. In other words, the bigger the bias, the easiest is to get a positive number and satisfy the second condition.

8.1 Sigmoid Neurons

The perceptron neuron model can be extended in order to create big networks with many perceptrons linked between each other. However, a perceptron network returns a *harsh decision*, that is, a slight change in the weights, could change the output of the whole network and flip from 0 to 1. To learn, however, it is necessary that small changes in the weights and biases only change in a small amount the output of a network so we can modify our links little by little until getting the desired output.

To overcome this problems, lets introduce a new, more realistic type of artificial neuron: **the sigmoid neuron**. This neuron are such that now it is possible to make small changes in the weights and it would be reflected only as a small change in the output. It works similar as the perceptron, that is, it takes some input variables $\{x_1, ..., x_n\}$ and computes a weighted sum. However, instead of returning a binary decision based on $\text{sign}\, z = \text{sign}(\vec{w} \cdot \vec{x} + b)$, it returns a sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \qquad \text{with } z = \vec{w} \cdot \vec{x} + b \tag{8.3}$$



Looking at the shape of the sigmoid function in Figure (), we can clearly see that this type of neurons no longer return a binary output, but instead, a continuous variable between 0 and 1.

8.2 Feedforward Neural Networks

A single artificial neuron, whether is a perceptron or a sigmoid neuron, is not very useful. The idea behing learning requires to join many neurons together by communication links, creating an **Artificial Neural Network**. The structure of a NN can vary, making this approach to learning very versatile. However, it is common to describe such structure as a graph of nodes, correspoding to individuals neurons, and each line representing the output of some neuron to the input of another neuron. When the links on a NN go only in one direction (it doesn't have any cycles), we say that it is a **Feedforward Neural Network**.

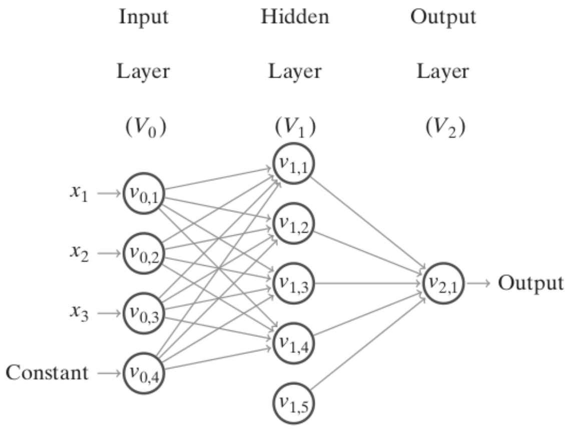


Figure 8.2: Example of a FNN with only one hidden layer

Following Figure (), we will say a network is organized in **layers**, a subset of neurons V_t. The set of all nodes in the network is then the union of T subsets, $V = \cup_{t=0}^{T} V_t$. Furthermore, each neuron in the network is modeled as a simple scalar function $\sigma(z)$, also called **activation function**. Each line in the graph links the output of some neuron to the input of another neuron, which is calculated as the weighted sum of the outputs of all the neurons connected to it. The output of the neuron is then obtained by applying the activation function to such input.

Let's consider a NN with T layers (excluding V_0). This is called the *depth* of the network. The *size* is the total number of neurons $|V|$, and the *width* is $\max_t |V_t|$. We will denote the ith neuron of the tth layer as $v_{t,i}$, and its output as $O_{t,i}$. The basic structure of a NN can be described as follows:

- **Input Layer:** Correspond to the bottom (or first) layer V_0 of a NN. It contains $n+1$ neurons. For every neuron $i \in [n]$ in this layer, the output is simply the input x_i, that is, $O_{0,i} = x_i$. The last neuron in V_0 is the *constant neuron*, which always outputs 1, $O_{0,n+1} = 1$.

- **Hidden Layers:** Layers $V_1, ..., V_{T-1}$ are called hidden layers. They can have an arbitrary number of neurons.
- **Output Layer:** Correspond to the top (or last) layer of the network. It could contain a single neuron in simple prediction problems, or more, depending on the task.

The mechanism by which a NN works is *recursively*. That is, given the outputs of the set of neurons in one layer, we can *feed* it *forward* to the next one until we reach the top layer V_T. Formally, suppose we have the outputs of the neurons at the $t-$th layer, and we would like to calculate the output of the $j-$th neuron of the $(t+1)-$th layer, $v_{t+1,j}$. Let's call the input to such neuron as $a_{t+1,j}$. Then,

$$a_{t+1,j} = \sum_{r:(v_{t,r}, v_{t+1,r})} w_{r,j}^{t+1} O_{t,r} + b_{t+1,j} \tag{8.4}$$

This is equivalent to what we said before: the input of the j-th neuron of the $(t+1)-$layer is no more than the weighted sum of the outputs of the neurons connected to it plus the bias. Here, the sum goes over r, representing the set of all neurons in the $i-$th layer connected to $v_{t+1,j}$, and so $w_{r,k}$ are the weights associated to those links - the weights the r neurons in the previous layer linked to the j-th neuron in the $t+1$-th layer-.

We can now used the input $a_{t+1,j}$ to get the output of the $v_{t+1,j}$:

$$O_{t+1,j} = \sigma\left(a_{t+1,j}\right) \tag{8.5}$$

Figure () represents then a Neural Network of depth 2, size 10 and width 5 with a constant neuron in the hidden layer.

8.3 Learning Neural Networks

To specify a Neural Network, we need its size given by V, the links between the neurons denoted by E, the activation functions of neurons σ and the weights w associated to the links. Once we specify these values, the predictor we get using a neural network is a function h such that,

$$h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \rightarrow \mathbb{R}^{|V_T|}$$

That is, the function h takes an input $\vec{x}$ with dimension $|V_0| - 1$ (because we are not counting the constant neuron input), and we obtain a result that is going to be a vector whose dimension depends on the number of neurons in the output layer $|V_T|$. The triplet of values (V, E, σ) is called the **architecture of the network**, because they correspond to values that we need to specify to define the configuration of the network. These are values represent the kind of *prior knowledge* we need to have about our learning task. Therefore, the weights over the network specify the hypothesis in the hypothesis class, that is, given (V, E, σ), the NN will return a predictor h whose weights are optimal for the task:

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \, \text{is a mapping from E to } \mathbb{R}\} \tag{8.6}$$

8.4 SGD and Backpropagation

As we just saw, the output of a NN is a function or predictor, defined by its weights. Once the weights are optimal for a learning task, we say the network learned successfully. However, to find these weights, we need first a measure of how well the network is doing, in other words, we need to measure the *cost*. The empirical cost for a neural network can be defined in many ways, but for the sake of the explanation, we will use the mean squared erro as we did in the problem of linear regression:

$$L_S(w, b) = \frac{1}{2n} \sum_i \left\| y^{(i)} - O_T^{(i)} \right\|^2 \tag{8.7}$$

Here, $O_T^{(i)}$ is the vector of outputs form the network for the $x^{(i)}$ vector of features (the input) and $y^{(i)}$ is the vector of the desired output of that input. As before, the goal now is to minimize the cost function, which means that the output of the network become similar enough to the desired output. To find the minimum of such cost function we can use GD (SGD) as seen in Chapter (), that is, we can start with some random weights and biases and compute the gradient of the cost in order to move in the opposite direction of the increment of the function by updating the different variables. In this case, for T iterations, the rule is,

$$w_k^{(t+1)} = w_k^{(t)} - \eta \nabla_w L_S(w, b) \tag{8.8}$$

$$b_l^{(t+1)} = b_l^{(t)} - \eta \nabla_b L_S(w, b) \tag{8.9}$$

But computing the gradient of the cost function is not an easy task nor as direct as it was in the previous learning algorithms. The problem is L_S depends on the output of the top layer, which is a function of the outputs of the previous layer, which are also function of the outputs of the previous layer, and so on, until reaching the bottom layer. Therefore, the output vector contains in a implicit way all the weights counting from the input layer to the last one. The algorithm that allows us to understand how the cost function changes if we change a random weight of a random layer is called **Backpropagation**.

To see how the changes in weights affect our final cost function, let's consider Figure (). Here we show the $j-$th neuron in the $t-$th layer. We see on the very left all the weights from the links connected to such neuron and its input

$a_{t,j} = \sum w_r O_{t-1,r}$. Then, the neuron produce an output $O_{t,j} = \sigma(a_{t,j})$ which will be weighted and summed to produce the inputs of the neurons in the next layer. In the right we have the output layer, that recieves the outputs of the previous layer and generates an output vector O_T. Finally, we can use that output to compute the total cost function L_S.

In red, we show what happens when the input of a neuron in a random layer is modified by a small amount Δa. We can see that it will modify the output of such layer and eventually it will propagate to the last layer changing the cost function:

$$a \rightarrow a + \Delta a \qquad \Longrightarrow \qquad L_S \rightarrow L_S + \frac{\partial L_S}{\partial a}\Delta a \tag{8.10}$$

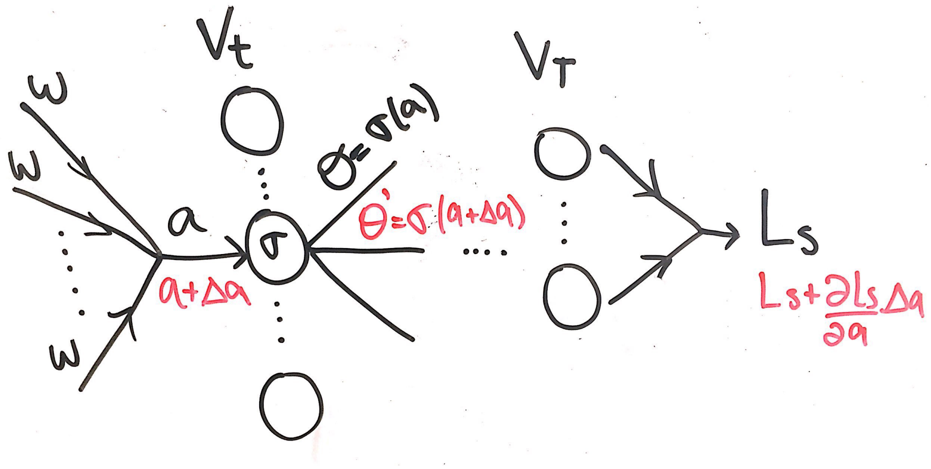


Figure 8.3: Representation of a section of a NN and the propagation of a small change in the input of a neuron.

From Equation () we see that if we want to make the cost smaller, we have to choose Δa according to ∂L_S. That is, if ∂L_S is big, we will have to choose Δa with the opposite sign. If for example ∂L_S is very small, then there is no much we can do with Δa to make the cost smaller. Motivated by this, lets define the *output error* of the $j-$th neuron in the t-th layer as,

$$\delta_j^t = \frac{\partial L_S}{\delta a_{t,j}} \tag{8.11}$$

Using the previous definition we can compute directly the output error of the last layer V_T:

$$\delta_j^T = \frac{\partial L_S}{\delta a_{T,j}} = \frac{\partial L_S}{\partial O_{T,j}} \frac{\partial O_{T,j}}{\partial a_{T,j}} = \frac{\partial L_S}{\partial O_{T,j}} \sigma'(a_{T,j}) \tag{8.12}$$

Now we want to find an expression for the output error of any layer t in terms of the error of the next layer $t+1$:

$$\delta_j^t = \frac{\partial L_S}{\delta a_{t,j}} = \sum_k \frac{\partial L_S}{\partial a_{t+1,k}} \frac{\partial a_{t+1,k}}{\partial a_{t,j}} = \sum_k \delta_k^{t+1} \frac{\partial a_{t+1,k}}{\partial a_{t,j}} \tag{8.13}$$

To calculate this expression, we use the definition (),

$$a_{t+1,k} = \sum_r w_{k,r}^{t+1} O_{t,r} = \sum_r w_{k,r}^{t+1} \sigma(a_{t,r}) \tag{8.14}$$

differentiating,

$$\frac{\partial a_{t+1,k}}{\partial a_{t,j}} = w_{k,r}^{t+1} \sigma^{'}(a_{t,r}) \tag{8.15}$$

So finally we can write Equation () as,

$$\delta_j^t = \sum_k \delta_k^{t+1} w_{k,r}^{t+1} \sigma^{'}(a_{t,r}) \tag{8.16}$$

Using Equation () and () we can now proceed with the backpropagation procedure: we combine these two equations starting by () to compute δ^L, and then apply () to compute δ^{L-1} and then repeat this process all the way back through the network until the first layer. By doing so, we would have the δ^l for all the layers and the only missing step is to relate this error to the gradient of the cost function. This can be done with the following two equation:

$$\frac{\partial L_S}{\partial w_{jk}^l} = O_k^{l-1} \delta_j^l \tag{8.17}$$

$$\frac{\partial L_S}{\partial b_j^l} = \delta_j^l \tag{8.18}$$

Taking the previous equations, we can summarize backpropagation as follows:

1.- Initialize the NN with some weights and biases.

2.- Given an input, feedforward through the network until reaching the output layer.

3.- Compute the error δ^L of the output layer using Equation ().

4.- Backpropagate the error to the previous layers using Equation ().

5.- Compute the gradient of the cost function and update the weights and biases.

8.5 Cross Entropy Cost Function

In Section we described a neural network as a collection of layer, each one containing a fix amount of artificial neurons, whose activation depends on the weighted sum of the outputs of the previous layer and the bias. So far, we have only mentioned sigmoid neurons, however, as we will see, this is not the most convenient activation function for how we defined the cost of the network.

Consider Figure (). We see that a sigmoid neuron is most sensitive when its input is closer to zero, but as soon as the input becomes bigger or smaller than a certain value, the neuron becomes less sensitive: the neuron *saturates*. In terms of learning, this is bad news. The saturation zone of the function $\sigma(z)$ correspond to the points where it flattens, and so its derivative approaches to zero in that area. We also showed that in order to calculate the gradient of the cost, we need to use backpropagation. According to Equation () we see that when we calculate the error of a neuron, we need to take into account the derivative of the activation function, so if the input is such that the output lands in the flat part of the curve, the derivative would be close to zero, and so the gradient of the cost would also be close to zero, making no changes to the total cost. This means our network is not learning optimally, and got stuck. This is called **learning slowdown**, and it is a consequence of the activation function and the cost function we chose.

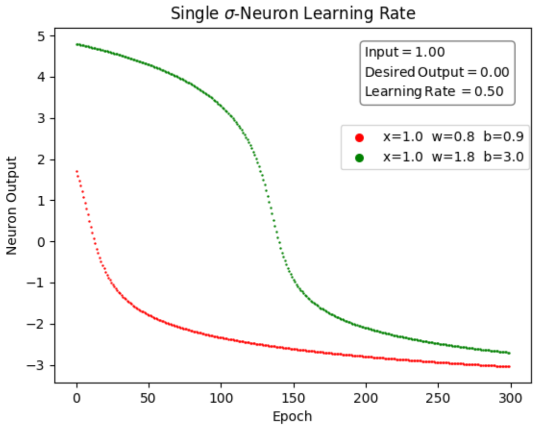


Figure 8.4: Learning rate for a single neuron with MSE cost function, given two different sets of initial conditions.

To visualize this issue, let's consider Figure (). We modeled a single sigmoid neuron with a quadratic cost function, whose input is $x = 1.0$ and we want to take that input to $y = 0$ using GD and backpropagation. The points in red and green reflect two different sets of initial conditions. Given the first set of conditions in red, we see that the neuron starts learning very fast. This happens because its activation start in a point where the sigmoid neuron is very sensitive given the weights and bias we chose. However, if we modify those conditions and force the neuron to start in a point where the sigmoid function saturates, we clearly see that for the first 100 epochs, the weight and biases are not changing much, slowing down the learning process.

Although tt might seems like the problem is the activation function we can solve this issue only by changing the cost function. In doing so, we want a cost function that avoids the learning slowdown caused by the derivative of the activation function in the gradient of the total cost. A widely used function that has this property is the **Cross-Entropy** cost function:

$$L_S = -\frac{1}{N}\sum_i \left[y^{(i)} \log(O_T^{(i)}) + (1 - y^{(i)}) \log(1 - O_T^{(i)}) \right] \tag{8.19}$$

The cross-entropy is a reasonable and very useful cost function. It is non-negative and tends to zero as the network gets better at computing the desired output. Furthermore, from Chapter (), we recognize this expression, as the log likelihood given that O_T is the probability of a sample to be part of a particular class.

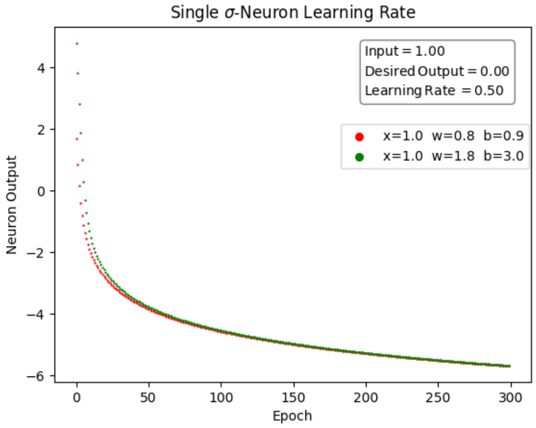


Figure 8.5: Learning rate for a single neuron with CE cost function, given two different sets of initial conditions.

To understand how this cost function solve the problem of slow learning when dealing with sigmoid neurons, let's calculate the gradient. Let's consider the simple case of just one neuron:

$$L_S = -\frac{1}{N}\sum_i \left[y^{(i)} \log(\sigma(z)) + (1 - y^{(i)}) \log(1 - \sigma(z)) \right] \tag{8.20}$$

$$\frac{\partial L_S}{\partial w_j} = \frac{1}{N}\sum_i x_j^{(i)} \left(\sigma(z) - y^{(i)} \right) \tag{8.21}$$

As we can see from Equation (), the learning rate now is being controled by the same error in the output $(\sigma - y)$: the larger the error, the faster the neuron will learn. This implies there won't be any slowdown even if we land in the saturation zone of the sigmoid neuron. Figure () shows a single σ neuron with the same task and initial conditions as before, but this time, we changed the cost function.

We can clearly see now that even with the set of conditions that produced slowdown learning in the MSE case, the neuron doesn't get stuck and there is no issue with its sensitivity now that we are using cross-entropy.

8.6 Activation Functions

Feedforward Neural Networks are defined by the arquitecture of the network, that is, the triplet (V, E, σ). The first two elements can be thought as *geometrical* parameters, concerning the macro structure of the network. However, the activation function σ tells us the way neurons in the different layers respond to certain inputs. This functions are responsable to introduce *non-linearity* into our network, allowing the network to approximate any non-linear function depending on its complexity.

In this section we mention some of the most common activation functions, its properties, advantages and disadvantages, in order to gain a deeper understanding on how to build a neural network.

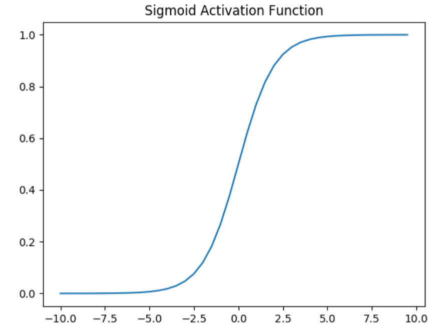
8.6.1 Sigmoid Function

We introduced the Sigmoid activation function, also called *Logistic function* in the first part of this chapter. It was one of the first activation functions used because of the interpretation of its output as a probability.

It takes any real value and squashes it between 0 and 1, introducing non-linearity to the network. However, as we also discussed in previous sections, the sigmoid activation function saturates at certain inputs, *killing* the gradient of the cost. As a result, it becomes more difficult to update the weights and biases, producing slowdown learning.

Another drawback of the sigmoid function is that is non zero-centered. This means the output is always a number between 0 and 1, that is, the output after passing through a sigmoid neuron is always positive. This has certain effects in backpropagation during gradient descent, as it makes the gradient to always be either positive or negative. As a result, the gradient updates go too far in different directions which makes optimization harder.

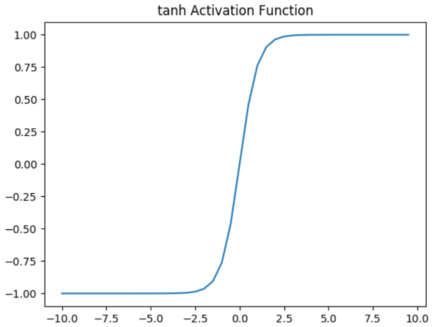
$$\sigma(z) = \frac{1}{1+e^{-z}}$$



8.6.2 Tanh Function

The hyperbolic tangent activation function is very similar to the sigmoid function. in fact, it is a shifted version of it. It takes a real number and it squashes it, this time, between -1 and +1.
Because it shares most of the properties of a sigmoid function, it also saturates at large positive and negatives values. However, this function is zero centered, and so neurons receiving a signal from a tanh neuron will get zero centered inputs. This makes the tanh functions better than a sigmoid in hiddent layers.

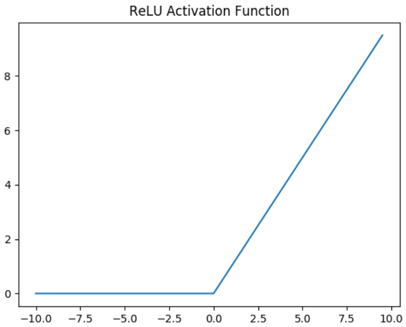
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(z) - 1.$$



8.6.3 ReLU Function

The Rectified Linear Unit (ReLU) activation function has become the most popular and used function nowadays. It takes a real number as input and applies an identity transformation if the number is positive, or outputs zero if the input is negative. In simple words, it thresholds the inputs at zero.

$$f(z) = \max(0, z)$$



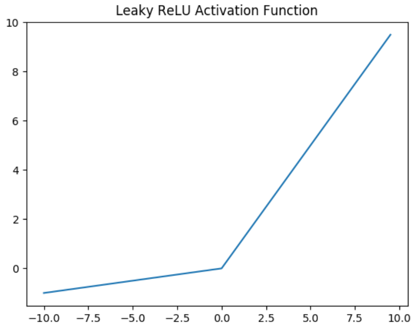
Furthermore, because it is a combination of linear functions, ReLU neurons don't saturate at large positive or large negative values. In fact, the gradient is set to be either 0 or 1 depending on the sign of the input. Another advantage of this activation function is the sparsity of activations. That is, due to its form, using this function for large networks lead to some of the neurons to stop firing (dead neurons), making the network to become lighter and more efficient. However, this can also be problematic if the neurons are not activated initially.

8.6.4 Leaky ReLU Function

As we mentioned before, the form of a ReLU activation function can lead to dead neurons in the initialization of the network. To avoid any issues, instead of killing the neurons, we can just make its output to be small. The Leaky ReLU activation function does exactly that, by modifying the classical ReLU in the negative range.

Leaky ReLU works as a combination of two linear functions. For the positive range of inputs, it is exactly the same as a ReLU, but for the negative range, instead of returning zero, it returns a small value, depending on the slope m we choose for such line (usually very small ~ 0.01). That parameter can be learned by cross validation and it could be considered as another hyperparameter to be setted.

$$f(z) = \begin{cases} mz & \text{if } x < 0 \\ z & \text{if } x \geq 0 \end{cases}$$



8.6.5 SoftMax Function

Another very common activation function is the SoftMax function. It is commonly used fr the neurons in the output layer rather than the hidden layers. Using this function, the output activations are guaranteed to always be positive and sum up to 1, making this function a very good candidate to represent a probability distribution.

$$O_{T,j} = \frac{e^{z_j^T}}{\sum_k e^{z_k^T}} \tag{8.22}$$

We can think the SoftMax function as a way of rescaling the inputs and then squishing them together to form a probability distribution.