UNIVERSITÀ DEGLI STUDI DI NAPOLI
"FEDERICO II"

Scuola Politecnica e delle Scienze di Base

Area Didattica di Scienze Matematiche Fisiche e Naturali

Dipartimento di Fisica "Ettore Pancini"

Laurea Triennale in Fisica

A quantum implementation of Support Vector Machine algorithm on IBM QX processors

Relatori:
Acampora Giovanni
Vitiello Autilia

Candidato:
Mendozza Raffaele
Matr. $N85000978$

Anno Accademico 2018/2019

Introduction

Since the introduction of computers, the way to do research and manage data has radically changed. Complex software can be executed with programmable hardwares, giving us real time responses to hard problem. Of course, not all problems are solvable by a classical computer, or at least not in polynomial time. On the other hand the Von Neumann model for computers is based on a sequential logic, so it necessarly exhibits structural limits when managing big amounts of data. This seems to be an insurmontable limit : even if according to *Moore's first law* the complexity of a circuit (measured, for example, as the number of transistors) is going to double every eighteen months, there will be structural limits imposed by microscopic quantum effects, poisoning our circuit. But if the real nature of matter is quantum, why don't use these effects as the building blocks of our computer, rather than as flaws? This is what happens with a quantum computer. As the research about semiconductors led to the invenction of the transistor and then to the digital logic at the base of nowadays computers, the blossoming research about superconductive materials and atomic physics has brought to the birth of the first quantum computer. EDVAC and ENIAC extended for several m^2 of surface and required a continous monitoring by human operators : no one thought they could evolve in a laptop or a smartphone. At the same manner, quantum computers developed by IBM are very big and have small processors. They can be programmed by remote via cloud thanks to the IBM Q EXPERIENCE platform. This could be a turning point in the history cause it could overturn our everiday life, as the classical computer did. Actually the idea of a quantum computer was born in the early 1980s thanks to Richard Feynman. Then numerous computer scientists and physicists developed new theories about quantum information and showed that (at least theorically) a quantum computer can perform better than a classical one. This is because a quantum computer can take advantage of phenomena unknown to its classical counterpart. On the other hand, modern works and experiments require to manage with big amounts of data, constantly monitored and analyzed. This led to the birth of new tecniques to *learn from data*, i.e. to the birth of *machine learning*. Among the most important algorithms widley used by researchers, there is the *Support Vector Machine* (SVM), a linear classifier used

to predict some properties from new data, according to the *previous experience* of the algorithm. Then one question naturally emerges : is it possible to improve a symilar algorithm with quantum effects? Is it possible to develop a *quantum machine learning* theory?
In this work we will try to lay the foundations to apply all benefits of quantum mechanics to machine learning, hoping this will considerably improve performances. We will adopt the following scheme : in the first chapter basic ideas about machine learning will be showed, then we will shift the focus to supervised learning and classification problem, presenting the SVM. The basic concepts about quantum computation will be presented in the second chapter, with particular emphasis about the "quantum speed-up". In the same chapter a brief description of the processors used in the IBM quantum computers will be provided, as well as the description of the IBMQ platform. The third chapter presents a brief introduction to quantum machine learning and a detailed description of the quantum support vector machine implemented by IBM. Eventually, in the last chapter, a comparison is made between the classical and quantum implementation of the SVM on the same data.

Contents

Chapter 1

Machine learning : the SVM algorithm

Talking about *machine learning*, we simply refer to a class of algorithms able to *infer* unknown properties from data. There are three main different ways to perform this task, so it is possible to identify three subsections of the machine learning domain :

- Supervised learning: the algorithm tries to "learn" a model from a limited set of known data , e.g. classification and regression algorithms.
- Unsupervised learning: the algorithm tries to extract information from unstructured data, e.g. clustering algorithms.
- Reinforcement learning: the algorithm tries to maximize some notion of rewards throug its interaction with the environment.

In this chapter we will describe how supervised learnig algorithms work, then the SVM will be presented and discussed in detail. It is a very important implementation of linear (and not only) classifier, cause differently from other algorithms it provides for one and one only analytical solution.

1.1 A formal definition of the classification problem

Let us say we are studying a phenomenon $\boldsymbol{x}$. We can think about it as a numerical vector of d components, each representing the measure of one characteristic of $\boldsymbol{x}$: they are called *features* and than $\boldsymbol{x} \in \mathbb{R}^d$. Repeating these measurement t times, their collection $\chi' := \{\boldsymbol{x^i}\}_{i=1}^{i=t}$ is called *dataset*, where each $\boldsymbol{x_i}$ is an $instance$ and $\chi' \in \mathbb{R}^d \times \mathbb{R}^t$. The i^{th} feature of the j^{th} instance from the dataset will be referred to as x_i^j. Given $\boldsymbol{x}^j$ we may be interested to know a boolean value (yes/not, for

example) or a numeric function related to its features. Then collecting a sufficient number N of instances $\boldsymbol{x}^i$ and relative output r^i it is possible to define the set $\chi := \{\boldsymbol{x}^t, r^t\}_{t=1}^{t=N}$.
Just to fix our ideas, let us think about a feature space of dimension d=2 with just two classes ($r \in \{0,1\}$) so we can represent the training set as in figure 1.1

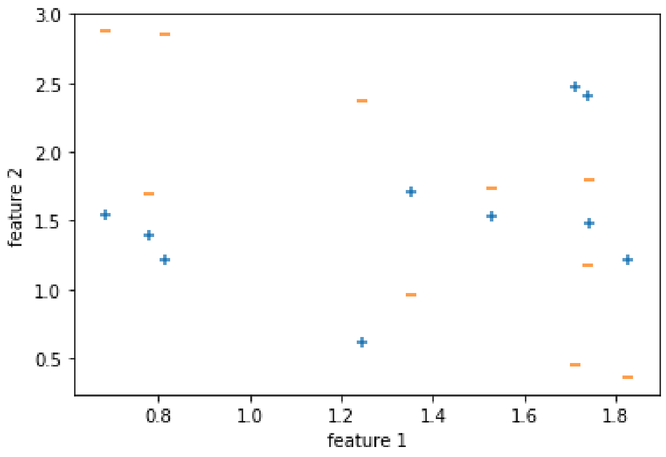


Figure 1.1: Representation of instances with two features, belonging to two different classes. In this example there are 10 instances per class

If we want to predict the class C_i $\boldsymbol{x}^i$ belongs to (for any $\boldsymbol{x}^i \notin \chi$), we can think that for suitable values of x_1 and x_2, $\boldsymbol{x}$ belongs to a specific class C, so we can draw some sort of *decision margins* to geometrically classify the instances; obiviously this can be generalized to any dimension of the feature space.

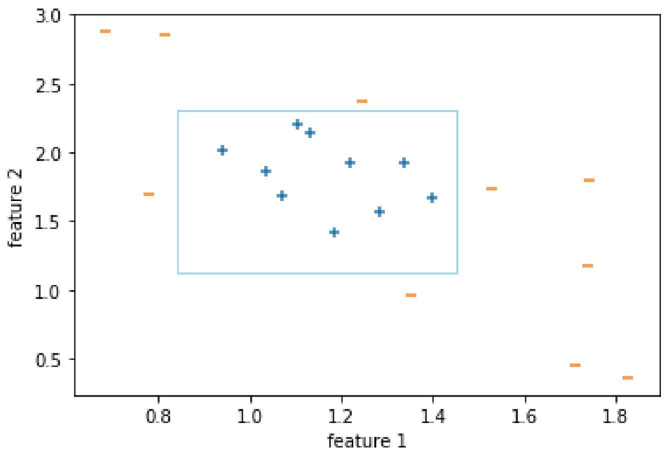


Figure 1.2: In this example all instances with feature values belonging to a determined range share the same label

Fixing the geometry of the boundaries is fixing the *hypotesis class H* from which we believe *C* is determined , while running the algorithm on the training set and choosing its "optimal configuration" from the validation set, we are fixing a particular $h \in H$. The aim is to find an h as close as possible to *C* (unknown) such that :

$$h(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \text{ is classified as belonging to } C_1 \\ 0 & \text{if } \boldsymbol{x} \text{ is classified as belonging to } C_2 \end{cases}$$

So it is possible to obtain a first (rough) evaluation of the precision of the algorithm evaluating the error E (assuming the validation set composed of N instances) as

$$E(h \mid \chi) = \frac{N - \sum_{t=1}^{N} \delta_{h(\boldsymbol{x}^t),r^t}}{N} \tag{1.1}$$

We can immediatly notice that , cause of the limitate number of instances in the dataset, there could be different hipoteses $h_i \in H$ in agreement with collected data and they could reproduce the same error $E(h \mid \chi)$, but they have different properties of *generalization*. We define S the minimal hypotesis and G the maximal one, with $S, G \in H$. The best hypotesis h, the one which will generalize the best, is expected to lie somewhere between these two classes, i.e. $h \in [S, G]$: this is to increase the *margin*, which is the distance between the boundary and the instances closest to it. To choose between these algorithms the definition of another type of "error" is required: it is called *loss function* and penalizes not only misclassifications but also the distance from the margins of the hypotesis h.

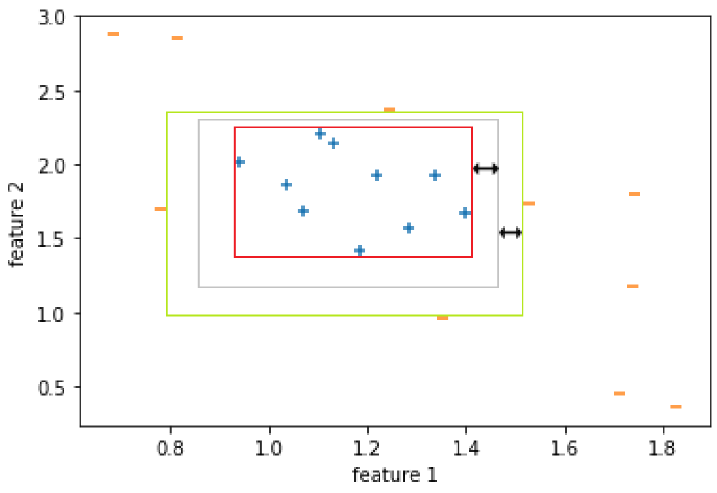


Figure 1.3: Different hypoteses (rectangles) from the minimal,in red, to the maximal, in green, can be choosen. The one in the middle (in gray) is preferred because it maximizes margins, so it is expected to generalize better than the others.

There is still a big question we haven't dealt with jet: fixed *H*, how can we be sure that we are also considering our real (and unknown!) hypotesis C, i.e. how can we be sure that $C \in \boldsymbol{H}$? To answer this question the concept of capacity of an algorithm is introduced. Obviously, according to 1.1, if

$$C \in \boldsymbol{H} \Rightarrow \exists h \in \boldsymbol{H} : E(h \mid \chi) = 0$$

If there is no such h, then the class of hipoteses h is said to have no sufficient *capacity*. The capacity of a class of hipoteses *H* can be measured using the *Vapnik-Chervonenkis (VC) dimension* of *H*. Taken a dataset χ of N instances $\boldsymbol{x}^t$, each labelled by $r^t \in \{-1, 1\}$, 2^N different learning problems can be defined. If for any of these problems an hipotesis $h \in \boldsymbol{H}$ separating positive from negative examples can be found, than it is said that *H* $shatters$ N points and $VC(\boldsymbol{H}) = N$. This means than any learning problem definable by N examples can be learned with no error by a hypotesis from *H*. For example, we can see from 1.4 that fixing *H* as a class of rectangles, we have $VC(\boldsymbol{H}) = 4$.

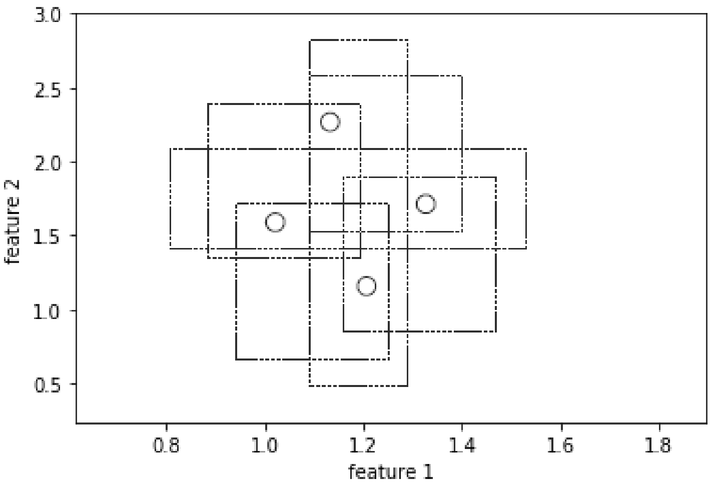


Figure 1.4: [2]The hypotesis class of "rectangles" can shutter four elements. In this picture only rectangles covering two points are shown

Given the usual training set $\chi := \{\boldsymbol{x}^t, r^t\}_{t=1}^{t=N}$, $\boldsymbol{x}^t s$ are always referred to as independent and identical distributed (*iid*) variables. The aim is to obtain $r^t_{predicted} = r^t_{expected}$ for any t, so a *model* $g(\boldsymbol{x}^t \mid \boldsymbol{\theta})$ directly dependent on some parameters is defined to put $r^t_{predicted} = g(\boldsymbol{x}^t \mid \boldsymbol{\theta}) := \tilde{r}^t$. We can immediatly notice that $g(\cdot)$ defines *H* and $\boldsymbol{\theta}$ fixes h, so $g(\cdot)$ is defined from the algorithm designer and h is chosen from the algorithm running on the set χ.To choose the best configuration of the parameters, the *loss function* is defined as:

$$E(\theta \mid \chi) := \sum_{t=1}^{N} L(r^t, g(\boldsymbol{x}^t, \boldsymbol{\theta})) \tag{1.2}$$

where $L(r^t, g(\boldsymbol{x}^t, \boldsymbol{\theta}))$ computes the difference between the desired and predicted label for $\boldsymbol{x}^t$. Then parameters $\boldsymbol{\theta'} = (\theta'_1, \ldots, \theta'_m)$ can be found computing

$$\left.\frac{\partial E(\theta \mid \chi)}{\partial \theta_i}\right|_{\theta'_i} = 0 \tag{1.3}$$

In simple models, this optimization problem can be analitically solved, while for more complex models the optimal $\boldsymbol{\theta}$ is evaluated using gradient-based methods or genetical algorithms.
If for example we have a feature space of dimension $d = 2$ and two possible classes, then explicitly $\boldsymbol{x}^t = (x^t_1, x^t_2), \tilde{r}^t \in \{0, 1\}$, we define

$$\tilde{r}^t = \begin{cases} 1 & \text{if } P(C = 1 \mid x^t_1, x^t_2) > P(C = 0 \mid x^t_1, x^t_2) \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$

so the *posterior probability* $P(C \mid \boldsymbol{x})$, i.e. the probability that a class C is associated to some fixed features x_i, must be known. Using Bayes' rule, the problem is reformulated as:

$$P(C \mid \boldsymbol{x}) = \frac{P(C)p(\boldsymbol{x} \mid C)}{p(\boldsymbol{x})} \tag{1.5}$$

$P(C)$ is called the *prior probability* and is the probability that C takes a fixed value regardless of the $\boldsymbol{x}$ value. $p(\boldsymbol{x} \mid C)$ is the *class likelihood* and is the conditional probability that an event belonging to C has the associated features of $\boldsymbol{x}$. $p(\boldsymbol{x})$ is the *evidence*, the marginal probability that an observation $\boldsymbol{x}$ occurs, regardless the class it belongs to. It now appears clear that the model $g(\cdot)$ must be somehow linked to the *posterior probability* (1.5) , so according to (1.4) we will say that

$$\boldsymbol{x} \in C_i \quad if \quad g_i(\boldsymbol{x}) = \max_k g_k(\boldsymbol{x}) \tag{1.6}$$

$g_i(\boldsymbol{x})$, $i = 1 \ldots k$ are called *discriminant functions* , so the feature space is divided in k *decision regions*.

One approach to face this kind of classification problem (or, in general, of supervised learning problem) is using a *parametric model* : all $\boldsymbol{x}^t s$ from the dataset are assumed to follow a determined distribution, the best evaluation of the moments of theses distributions (before called $\theta_i s$) is computed with the maximum likelihood approach and then the loss function is extimated. In this way, explicit (and strong) hypoteses about $p(\boldsymbol{x} \mid C_i)$ are made and this method is called *likelihood based*. In this paper we will describe another approach to the same problem, called *discriminant based* .

1.2 Linear discrimination

In linear discrimination instances from a class are assumed to be linearly separable from instances from other classes (i.e. we assume we can always find hyperplanes separating decision regions) and explicit hypotesis about the form of boundaries separating classes are made, so a model $g_i(\boldsymbol{x} \mid \boldsymbol{\phi_i})$ is defined for each class C_i (note the direct dependence from parameters) and the algorithm is expected to optimize $\boldsymbol{\phi}$ running on the training set. Let us start from the most simple example: having a feature space of dimension d we can think about a discriminant linear in $\boldsymbol{x}$ as

$$g_i(\boldsymbol{x} \mid \boldsymbol{w}_i, w_{i0}) := \sum_{j=1}^{d} w_{ij}x_j + w_{i0} \tag{1.7}$$

so the output is a weighted sum of the inputs, where some features may be much more determining to choose che class the instance belongs to. Without loss of generality (we will later see why), it can be treated as a *Two classes classification problem*. In two classes problems discriminant functions of the two classes C_1 and C_2 are combined to define one global discriminant function:

$$g(\boldsymbol{x}) = g_1(\boldsymbol{x}) - g_2(\boldsymbol{x}) = (\boldsymbol{w}_1 - \boldsymbol{w}_2)^T\boldsymbol{x} - (w_{10} - w_{20})$$

so

$$g(\boldsymbol{x}) := (\boldsymbol{w}^T\boldsymbol{x} + w_0) \tag{1.8}$$

and according to (1.6) we choose

$$C(\boldsymbol{x}) = \begin{cases} C_1, & \text{if } g(\boldsymbol{x} > 0) \\ C_2, & \text{if } g(\boldsymbol{x} < 0) \end{cases}$$

It appears clear its geometrical interpretation

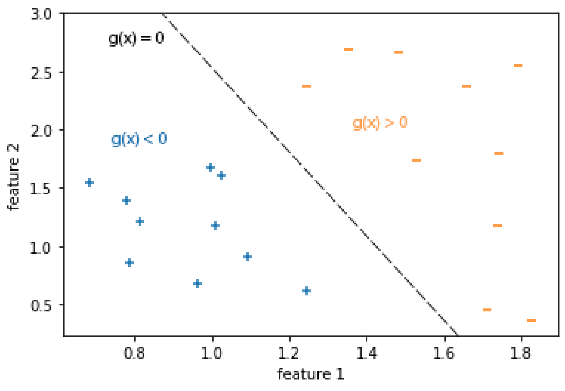


Figure 1.5: In a two classes problem, $g(\boldsymbol{x}) = 0$ represents the separating hyperplane. Instances are classified according to $g(\boldsymbol{x})$ values

Considering $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ lying on that hyperplane we have $\boldsymbol{w}^T(\boldsymbol{x}_1 - \boldsymbol{x}_2) = 0$ and so $\boldsymbol{w}$, as expected, defines the orientation of the decision surface. If we now proceed decomposing $\boldsymbol{x}$ in its parallel and normal components (referred to the hyperplane), i.e. $\boldsymbol{x} = \boldsymbol{x}_{\parallel} + \boldsymbol{x}_{\perp} := \boldsymbol{x}_{\parallel} + r\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}$ and since $g(\boldsymbol{x}_{\parallel}) = 0$, we immediatly obtain $r = \frac{g(\boldsymbol{x})}{\|\boldsymbol{w}\|}$, indicating distance from hyperplane. Its distance from the origin ($\boldsymbol{x} = \underline{0}$) is then $r_0 = \frac{w_0}{\|\boldsymbol{w}\|}$.
The model developed in this paragraph is designed only for two classes problems, so why did we say it is a general approach to a classification problem? According to the so called *one vs all* approach, if all classes are mutually separable, it is always possible to reformulate the problem of k classes as k problems of two classes. Infact in this situation k discriminant functions $g_i(\boldsymbol{x}), i = 1 \dots k$ can be defined and (assuming parameters $\boldsymbol{w_i}, w_{i0}$ known) they will reproduce as results:

$$g_i(\boldsymbol{x} \mid \boldsymbol{w_i}, w_{i0}) = \begin{cases} > 0, & \text{if } \boldsymbol{x} \in C_i \\ < 0, & \text{otherwise .} \end{cases} \tag{1.9}$$

Here is evident the *linear separable* hypotesis: for each class C_i there exists an hyperplane H_i such that all $\boldsymbol{x} \in C_i$ lie on its positive side and all the other $x_j \notin C_i$ lie on its negative side. So ideally, for each $\boldsymbol{x}_j$ there should exist one only i such that $g_i(\boldsymbol{x_j}) > 0$.Cause of noise and other factors (for example the hypotesis choosen is too simple for the facing problem) this doesn't happen and the usual approach is to assume $\boldsymbol{x} \in C_i$ if $g_i(\boldsymbol{x}) = \max_j^k g_j(\boldsymbol{x})$. Remembering the meaning of $g(\boldsymbol{x})$ and assuming $\|\boldsymbol{w}_i\|$ similar for any i, this means we attribute an instance to the class to whose hyperplane the point is most distant.
Analogously the *pairwise* or *one vs one* approach could be used : the k class problem is re-formulated as $\binom{k}{2}$ two classes problems. The main difference between these two approaches regards the dimension of training set used : the *one vs one* approach solves more classification problems, but with smaller training set. This is why the *one vs one* approach is usually preferred.

1.3 Supervised learning: a brief introduction

Studying the phenomenon $\boldsymbol{x}$, a big number N of instances $\boldsymbol{x}^i$ is collected to construct the dataset $\chi = \{\boldsymbol{x}^t, r^t\}_{t=1}^{t=N}$. Then χ is divided in two different sets[1], namely *training set* and *validation set.*

1. Training set: the algorithm predicts $\tilde{r}^t$ for any $\boldsymbol{x^t}$ and adjustes its parameters to have as much $\tilde{r}^t = r^t$ as possible (usually it is better not to have 100%

[1]Actually, χ is almost always divided in three parts : *training* , *validation* and *test* set. This is because the error evaluated on the validation set is part of the algorithm itself, while accuracies and othet metrics are evaluated on the test set. Often these two sets are equal.

of correct predictions because this often brings to *overfitting* problems. We will discuss about them later).

2. Validation set: using the best parameters found running on the training set, the algorithm predicts $\tilde{r}^t$ for any $\boldsymbol{x}^t$ in this set, so the ability of the algorithm to generalize, to "learn from data", is evaluated.The best configuration of the parameters is assumed as the one with the lowest error (notice that we still haven't defined any form of "error" yet).

This is the general structure of a supervised learning algorithm, such as an SVM used as classier, a polinomial regression or a neural network.

1.4 Optimizing parameters: Gradient Descent and hinge loss

To find the best parameters for the algorithm it can used the *gradient descent method*: defined the error $E(\boldsymbol{w} \mid \chi)$ committed by the algorithm with parameters $\boldsymbol{w}$ running on the training set χ, we look for

$$\boldsymbol{w}' : \left.\frac{\partial E(\boldsymbol{w}|\chi)}{\boldsymbol{w}}\right|_{\boldsymbol{w}'} = 0$$

Considering the gradient vector $\nabla_w E = (\frac{\partial E}{\partial w_1} \dots \frac{\partial E}{\partial w_d})^T$ $\boldsymbol{w}$ is randomly initialied, then $\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$ is evaluated and w_i is updated as $w_i^{new} = w_i + \Delta w_i \quad \forall i.$ η is called learning factor and determines how much the algorithm "corrects" $\boldsymbol{w}$ along that direction $\hat{w}_i$. The algorithm stops reaching $\nabla_w(E) \approx \underline{0}$, which means we are in a minimum, but we don't kow whether is it a local or global minimum. Also, the choice of η is determinant because big values could cause great oscillations and even divergence, while small values could cause a too slow convergence.

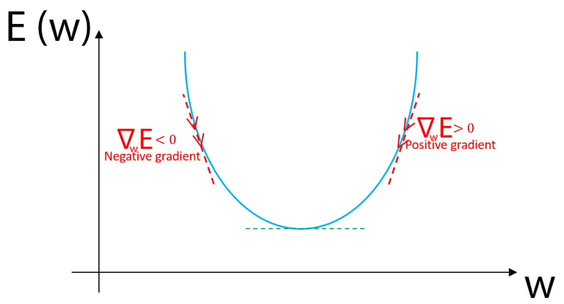


Figure 1.6: Parameters are updated sucht thath they will reach the optimal configuration, minimizing the error (or, in general, the *loss function*)

Then it is possible to find the optimal $\boldsymbol{w}$ for the discriminant using (1.3) through *gradient descent* method. First of all, a *loss function* must be defined. Indicating the rough output of $\boldsymbol{w}^T\boldsymbol{x}^t + w_0$ as $y(\boldsymbol{x}^t)$ and recalling the role of r^t as in (1.11), it is possible to define

$$l_h^t = \begin{cases} 0 & \text{if} \quad y(\boldsymbol{x}^t)r^t \geq 1 \\ 1 - y(\boldsymbol{x}^t)r^t & \text{otherwise} \end{cases}$$

penalizig both misclassified and well classified (but within the margin) instances, each one with its weight. Total loss function can be rewritten as

$$L_h = \sum_t l_h^t = \sum_t \max(0, 1 - y(\boldsymbol{x}^t)r^t) \tag{1.10}$$

Before applying *gradient descent* method, (1.10) should be "smoothed" to make sure it is differentiable.

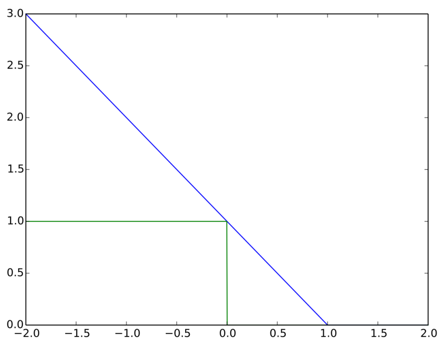


Figure 1.7: Comparison between the *hinge loss* and the "0/1" error function for an instance $\boldsymbol{x}^t$ belonging to C_1. On the abscissa runs $y(\boldsymbol{x}^t)$.*Hinge loss* penalizes $\boldsymbol{x}^t$ even if well classified, but within the margin

1.5 Support Vector Machine

We have now developed all necessary instruments to implement our classificator on a preprocessed dataset (we will see how to preprocess data in section 1.7). To do this, we need to define specific objective and loss functions, making specific assumptions: this defines the *inductive bias* of the algorithm. In this section we will describe the implementation of a supervised learning model for linear classification called SVM : *support vector machine*. The same algorithm can be used even to solve regression problems. It is a method which has become very popular in recent years thanks to different properties:

1. It is a disctriminant based method, so it isn't needed to evaluate the distribution of $\boldsymbol{x}$, while only assumptions about the boundaries of decision regions are made, so, in general, it is a simpler model.
2. The only parameters relevant to evaluate the weight vector $\boldsymbol{w}$ are the ones related to a subset of instances in the dataset, called *support vectors*. This leads to a computational advantage.
3. using the *kernel trick*, a non linearly separable problem can be mapped in an higer dimensional *feature space*, making it linearly separable and allowing to use the same SVM algorithm for a large gamma of classification problem.
4. Thanks to *kernel functions* the same SVM algorith can be used not only to classify events represented as vectors, but also to different formats of data (such as texts or graphs)
5. Kernel algorithms are convex optimization problems, so they have one only analitical solution.

We can start considering the usual *two classes classification problem*: given the training set $\chi = \{\boldsymbol{x}^t, r^t\}$, it is defined :

$$r^t = \begin{cases} 1 & \text{if } \boldsymbol{x}^t \in C_1 \\ -1 & \text{if } \boldsymbol{x}^t \in C_2. \end{cases} \tag{1.11}$$

and

$$g(\boldsymbol{x}^t) = \boldsymbol{w}^T \boldsymbol{x}^t + w_0 = \begin{cases} \geq 1 & \text{if } r^t = +1 \\ \leq -1 & \text{if } r^t = -1. \end{cases}$$

resumed as

$$r^t(\boldsymbol{w}^T \boldsymbol{x}^t + w_0) \geq 1 \quad \forall t \tag{1.12}$$

Note the difference from (1.9): here we want the instance distant from the separating hyperplane to obtain a better generalization.The distance between the hyperplane and the instance is called *margin* and the aim in the SVM algorithm is to maximize it.

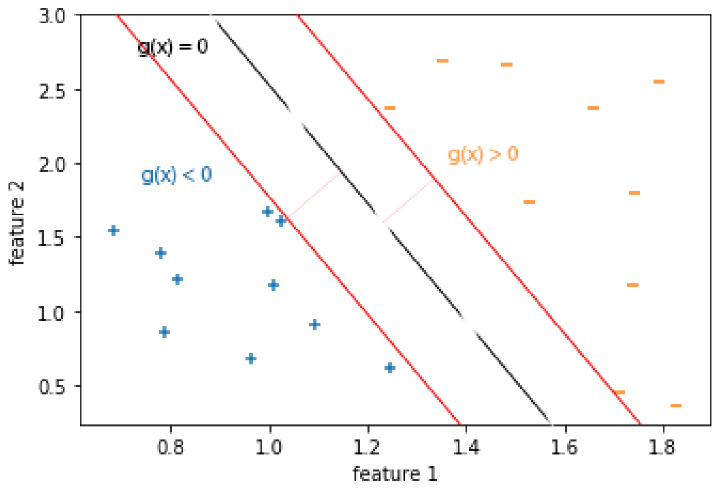


Figure 1.8: Among all possible discriminants, the one that maximizes margins is choosen for a better generalization

It recalls what we introduced in section 1.1. The hyperplane $g(\boldsymbol{x} \mid \boldsymbol{w}^T, w_0) = 0$ satisfying these conditions is called *optimal separating hyperplane*. Let's find this hyperplane. Remembering that the distance between a point and a plane can be evaluated as $d(\boldsymbol{x}^t, g(\boldsymbol{x} \mid \boldsymbol{w}^T, w_0) = 0) = \frac{|\boldsymbol{w}^T\boldsymbol{x}^t + w_0|}{\|w\|}$ that according to 1.12 becomes

$$d(\boldsymbol{x}^t, g(\boldsymbol{x} \mid \boldsymbol{w}^T, w_0) = 0) = r^t \frac{\boldsymbol{w}^T\boldsymbol{x}^t + w_0}{\|w\|} \tag{1.13}$$

with the condition $d(\boldsymbol{x}^t, g(\boldsymbol{x} \mid \boldsymbol{w}^T, w_0)) \geq \rho \quad \forall t$. The aim is to maximize ρ, since it is now clear that the *margin* is 2ρ. Considering the infinite number of $\boldsymbol{w}$ defining the same plane, of course there would be infinite solutions. $\rho\|w\| = 1$ is fixed, so maximizig ρ implies minimizing $\|\boldsymbol{w}\|$. Then the problem has now become

$$\min \frac{1}{2}\|\boldsymbol{w}\|^2 \text{ subject to } r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) \geq 1 \quad \forall t \tag{1.14}$$

and the margin has now become 2ρ. This linear programming optimization problem can be solved using Karush-Kuhn-Tucker (KKT) conditions . The *primal problem* is written as a Lagrange problem (subject to equalities constraint), leading to:

$$\mathcal{L}_p = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{t=1}^{N} \alpha^t [r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) - 1]$$

so

$$\mathcal{L}_p = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{t=1}^{N} \alpha^t r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) + \sum_{t=1}^{N} \alpha^t$$

and explicitly

$$\mathcal{L}_p = \frac{1}{2}(\boldsymbol{w}^T)\boldsymbol{w} - \boldsymbol{w}^T \sum_{t=1}^{N} \alpha^t r^t \boldsymbol{x}^t - w_0 \sum_{t=1}^{N} \alpha^t r^t + \sum_{t=1}^{N} \alpha^t \tag{1.15}$$

where $\alpha^t \geq 0$ are Lagrange multipliers. Now the dual problem must be solved, i.e. $\mathcal{L}_p$ is maximized with respect to α^t with constraints imposed by the resolution of the *primal* problem:

$$\begin{cases} \frac{\partial \mathcal{L}_p}{\partial \boldsymbol{w}} = 0, & \Rightarrow \boldsymbol{w} = \sum_{t=1}^{N} \alpha^t r^t \boldsymbol{x}^t \\ \frac{\partial \mathcal{L}_p}{\partial w} = 0 & \Rightarrow \sum_{t=1}^{N} \alpha^t r^t = 0 \\ \alpha^t \geq 0 & \forall \quad t \end{cases} \tag{1.16}$$

Then substituting (1.16) in (1.15) the *dual problem* is obtained:

$$\mathcal{L}_d = -\frac{1}{2} \sum_{t=1}^{N} \sum_{s=1}^{N} \alpha^t \alpha^s r^t r^s (\boldsymbol{x}^t)^T \boldsymbol{x}^s + \sum_{t=1}^{N} \alpha^t \tag{1.17}$$

subject to the same constraints. Solving in α^t we will find that only a small number of instances $\boldsymbol{x}^t$ have associated lagrange multipliers $\alpha^t \neq 0$: such vectors are called *support vectors*. They are very important because we can clearly see that they are the only instances which define the decision boundary.

Found the *optimal hyperplane*, $g(\boldsymbol{x})$ is evaluated for any $\boldsymbol{x}$ in the dataset and then they are labelled with $r^t = +1$ if $g(\boldsymbol{x}) > 0$ or $r^t = -1$ if $g(\boldsymbol{x}) < 0$, as discussed before.
What about the case of a non separable dataset? How do we choose between a wide margin with numerous misclassifications or a narrow margin with a minimum number of misclassifications? How good will our algorithm generalize what it learnt from the training set?
In this case we prefer to misclassify a restricted number of instances to mantain a sufficiently wide margin. We define *slack variables* $\xi^t \geq 0$: they are used to store information about the deviation of an instance $\boldsymbol{x}^t$ from the margin. So condition (1.12) is relaxed in

$$\begin{cases} r^t(\boldsymbol{w}^T \boldsymbol{x}^t + w_0) \geq 1 - \xi^t & \forall t \\ \xi^t \geq 0 & \forall t \end{cases} \tag{1.18}$$

where

1. $\xi^t = 0$ if $\boldsymbol{x}^t$ is correctly classified and lies beyond the margin

2. $\xi^t \in (0, 1)$ if $\boldsymbol{x}^t$ is correctly classified but lies within the margin

3. $\xi^t > 1$ if $\boldsymbol{x}^t$ is misclassified

This allows to define the *soft error* $\sum_{t=1}^{N} \xi^t$: the aim is now to contemporary maximize the *magin* and to minimize the *soft error*, so the following convex function must be minimized:

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{t=1}^{N} \xi^t \qquad C \in \mathbb{R}, \quad \text{subject to} \tag{1.18}$$

and to do this the Lagrangian problem is solved again using the KKT approach:

$$\mathcal{L}_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{t=1}^{N} \xi^t - \sum_{t=1}^{N} \alpha^t[r^t(\boldsymbol{w}^T\boldsymbol{x}^t + w_0) - 1 + \xi^t] - \sum_{t=1}^{N} \mu^t\xi^t \tag{1.19}$$

with the obvious introduction of the Lagrangian multipliers. Going on as in previous section leads to:

$$\begin{cases} \frac{\partial \mathcal{L}_p}{\partial \boldsymbol{w}} = 0 & \Rightarrow \boldsymbol{w} = \sum_{t=1}^{N} \alpha^t r^t \boldsymbol{x}^t \\ \frac{\mathcal{L}_p}{\partial w_0} = 0 & \Rightarrow \sum_{t=1}^{N} \alpha^t r^t = 0 \\ \frac{\mathcal{L}_p}{\partial \xi^t} = 0 & \Rightarrow C - \alpha^t - \mu^t = 0 \end{cases} \tag{1.20}$$

and substituting in equation (1.19):

$$\mathcal{L}_d = \sum_{t=1}^{N} \alpha^t - \frac{1}{2}\sum_{s=1}^{N}\sum_{t=1}^{N} \alpha^t\alpha^s r^t r^s (\boldsymbol{x}^t)^T \boldsymbol{x}^s \tag{1.21}$$

Solving in α^t using the last two constraints from (1.20) we obtain the so called *soft margin hyperplane*. As in the separable case, all instances $\boldsymbol{x}^t$ associated to $\alpha^t > 0$ are defined *support vectors* . We also have to notice that the form of the discriminants depends now on the *hyperparameter* C, given "by hand" by the algorithm designer. It is now possible to answer the questions asked before: *tuning* C, we can decide the weight of misclassifications in determining the hyperplane, but we have to be very careful: a high value of C leads to a discriminant which well fits the dataset used, but doesn't generalize well (*overfitting*); on the other hand a small value of C leads to a more general model, but it could be too simple (*underfitting*). C can be fine tuned using *cross-valitation* techniques.

1.6 The "Kernel trick"

Until now we always used the SVM on linearly separable datasets, the ones the algorithm is designed for. Is this really the most general approach to a classification problem? Let us consider the case of a *feature space* of dimension $d = 2$ and a dataset $\chi = \{\boldsymbol{x}, \boldsymbol{y} \dots\}$ whose instances are not linearly separable. For example, we could define a map $\boldsymbol{\Phi} : \boldsymbol{x} \in \mathbb{R}^2 \rightarrow \boldsymbol{\Phi}(\boldsymbol{x}) = (x_1^2 + x_2^2, x_1, x_2,) \in \mathbb{R}^3$ and we can apply the SVM algorithm on $\boldsymbol{\Phi}(\boldsymbol{x})$ rather than on $\boldsymbol{x}$ itself. Looking at (1.17) and considering the form of $\boldsymbol{w}$ from (1.20), such that

$$g(\boldsymbol{x}^s) = \boldsymbol{w} \cdot \boldsymbol{\Phi}(\boldsymbol{x}^s) = \sum_t \alpha^t r^t \boldsymbol{\Phi}(\boldsymbol{x}^t) \cdot \boldsymbol{\Phi}(\boldsymbol{x}^s)$$

we see that the only relevant quantities are the inner products
$\boldsymbol{\Phi}(\boldsymbol{x}^i) \cdot \boldsymbol{\Phi}(\boldsymbol{x}^j) \quad \forall i, j \in \{1, 2\}$. This means that the only piece of information which has to be stored in the memory is the value of inner products between these maps, saving lots of computational resources; these quantities are called *kernel functions*: $K(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\Phi}(\boldsymbol{x}) \cdot \boldsymbol{\Phi}(\boldsymbol{y})$. Notice that in this way the problem has become linearly separable in the trasformed space (see figure 1.9), i.e. it can be defined a linear *discriminant* $g(\boldsymbol{\Phi}(\boldsymbol{x})) = \sum_j w_j \Phi(\boldsymbol{x}_j)$, where $g(\boldsymbol{x})$ is nonlinear. The idea behind the *kernel trick* is now clear: instances are mapped from a non linearly separable problem in a space of different dimension using a suitable map Φ which linearizes the problem. Actually, since there are no guarantees that the problem is linearly separable in this new space, we look for the *soft margin hyperplane* repeating the same passages from (1.19) with $\boldsymbol{x}$ replaced by $\boldsymbol{\Phi}$.

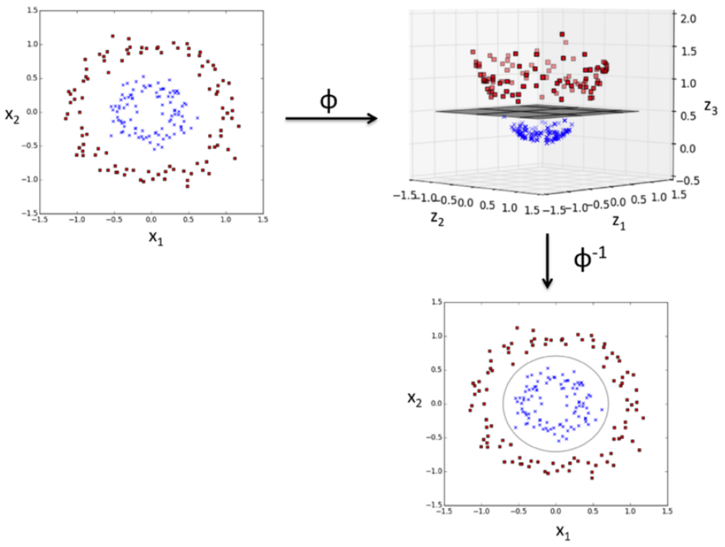


Figure 1.9: [3]A typical example of the *Kernel trick*. Via the *feature map* $\boldsymbol{\Phi}$, the problem becomes linearly separable

Now the natural question is: for which spaces $\mathcal{H}$ there exists a map $\boldsymbol{\Phi} : \mathbb{R}^d \rightarrow \mathbb{R}^m$ such that $K(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\Phi}(\boldsymbol{x}) \cdot \boldsymbol{\Phi}(\boldsymbol{y}) \quad \forall \boldsymbol{x}, \boldsymbol{y} \in \mathcal{H}$? I.e. when we define a kernel function, are we sure we are really mapping our features in another (Hilbert) space? First of all, we want

$$K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$$

to replace an inner product, so it surely must be a *symmetric bilinear positively defined form* . Then representing the associated matrix respect to any basis, it must be symmetric, i.e. defining $(\boldsymbol{K})_{i,j} = K(\boldsymbol{x}^i, \boldsymbol{x}^j)$ then $\boldsymbol{K}_{i,j} = \boldsymbol{K}_{j,i}$. This matrix is called *Gram matrix*. Explicitly:

$$\boldsymbol{K} = \begin{bmatrix} \langle \boldsymbol{\Phi}(\boldsymbol{x}^1)|\boldsymbol{\Phi}(\boldsymbol{x}^1)\rangle & \langle \boldsymbol{\Phi}(\boldsymbol{x}^1)|\boldsymbol{\Phi}(\boldsymbol{x}^2)\rangle & \dots \langle \boldsymbol{\Phi}(\boldsymbol{x}^1)|\boldsymbol{\Phi}(\boldsymbol{x}^N)\rangle \\ \langle \boldsymbol{\Phi}(\boldsymbol{x}^2)|\boldsymbol{\Phi}(\boldsymbol{x}^1)\rangle & \langle \boldsymbol{\Phi}(\boldsymbol{x}^2)|\boldsymbol{\Phi}(\boldsymbol{x}^2)\rangle & \dots \langle \boldsymbol{\Phi}(\boldsymbol{x}^2)|\boldsymbol{\Phi}(\boldsymbol{x}^N)\rangle \\ \vdots & \vdots & \ddots \\ \langle \boldsymbol{\Phi}(\boldsymbol{x}^N)|\boldsymbol{\Phi}(\boldsymbol{x}^1)\rangle & \langle \boldsymbol{\Phi}(\boldsymbol{x}^N)|\boldsymbol{\Phi}(\boldsymbol{x}^2)\rangle & \dots \langle \boldsymbol{\Phi}(\boldsymbol{x}^N)|\boldsymbol{\Phi}(\boldsymbol{x}^N)\rangle \end{bmatrix} \tag{1.22}$$

Gram matrix is invertible *iff* functions defining its elements are independent. Since $\boldsymbol{K}$ must be symmetric, it is always possible to find an orthogonal transformation V such that $\boldsymbol{K} = V\Lambda V^{-1}$, with Λ diagonal matrix of eigenvalues $\lambda_1 \dots \lambda_d$ for $\boldsymbol{K}$ and columns of V composed of eigenvectors $\boldsymbol{v}$ of $\boldsymbol{K}$. Defining a map as[20]

$$\boldsymbol{\Phi} : \boldsymbol{x}^i \in \mathbb{R}^d \to \boldsymbol{\Phi}(\boldsymbol{x}^i) = \{\sqrt{\lambda_1} v_i^1, \dots, \sqrt{\lambda_d} v_i^d\} \in \mathbb{R}^d$$

and computing $\langle \boldsymbol{\Phi}(\boldsymbol{x}^i)|\boldsymbol{\Phi}(\boldsymbol{x}^j)\rangle = \sum_{s=1}^d \lambda_s v_s^i v_s^j$,it is immediatly noticed that $\langle \boldsymbol{\Phi}(\boldsymbol{x}^i)|\boldsymbol{\Phi}(\boldsymbol{x}^j)\rangle = (V\Lambda V^{-1})_{i,j} = K(\boldsymbol{x}^i, \boldsymbol{x}^j)$. We are starting noticing the strong relation between a kernel function and eigenvalues/eigenvectors from Gram matrix. This link is formally expressed by *Mercer's theorem*, here reported in its simple version[20]:

Theorem. *Let χ be a compact set, $\chi \subset \mathbb{R}^n$. Suppose k is a continuous symmetric function such that the integral operator $T_k : L_2(\chi) \to L_2(\chi)$ defined by*

$$(T_k f)(\cdot) = \int_\chi k(\cdot, \boldsymbol{x}) f(\boldsymbol{x})\, d\boldsymbol{x}$$

is positive, which here means

$$\iint_{\chi\times\chi} f(\boldsymbol{z})k(\boldsymbol{z}, \boldsymbol{x})f(\boldsymbol{x})\, d\boldsymbol{x}\, d\boldsymbol{z} \geq 0 \quad \forall f \in L_2(\chi)$$

then $k(\boldsymbol{x}, \boldsymbol{z})$ can be expanded in a uniformly convergent series in terms of T_k 's eigenfunctions $\psi_j \in L_2(\chi)$, normalized so that $\|\psi\|_{L_2} = 1$, and positive associated eigenvalues $\lambda_j \geq 0$,

$$k(\boldsymbol{x}, \boldsymbol{z}) = \sum_{j=1}^{\infty} \lambda_j \psi_j(\boldsymbol{x}) \psi_j(\boldsymbol{z}).$$

To sum up, in this way we are sure that defining a kernel functions with the properties discussed below (symmetric and generating a positively semidefinite matrix), this represents an inner product in the space given by the map

$$\boldsymbol{\Phi} : \boldsymbol{x} \in \mathbb{R}^d \rightarrow \boldsymbol{\Phi}(\boldsymbol{x}) = [\sqrt{\lambda_1}\psi_1(\boldsymbol{x}), \dots \sqrt{\lambda_j}\psi_j(\boldsymbol{x}), \dots]$$

as we observed before the theorem with the example. Now that we know what conditions must be satisfied by a kernel function, let' s see the most famous ones used in literature:

1. *polynomial kernel:* $k(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{y} \cdot \boldsymbol{x} + 1)^q$ where q is selected by the user
2. *radial basis function* $(\boldsymbol{x}, \boldsymbol{y}) = exp\left[-\frac{\|\boldsymbol{x} - \boldsymbol{y}\|^2}{2s^2}\right]$
3. *sigmoid kernel* $(\boldsymbol{x}, \boldsymbol{y}) = \frac{1 - e^{-2(\gamma \boldsymbol{x} \cdot \boldsymbol{y} + r)}}{1 + e^{-2(\gamma \boldsymbol{x} \cdot \boldsymbol{y} + r)}}$ with r and γ parameters

When a kernel is coosen and its parameters are fixed for an algorithm, we must keep in mind that it has the same role as an inner product, so it must somehow weigh differences and similarities between the most relevant features for the facing problem.

1.7 Data preprocessing : standardization and PCA

Since the complexity of a classifier scales with the number of inputs, it could require a huge amount of resources to classify big datasets. This is why some methods of *dimensionality reduction* have been developed , trying to reduce the number of features (i.e. reducing the dimensionality d of the *feature space*). To improve algorithm prestations, it is also better to work with *standardized* datasets, substituting each variable $\boldsymbol{x}^i$ with $\boldsymbol{z}_i = \frac{\boldsymbol{x}^i - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$.
In this section we will focus on a particular method of *feature extraction* , called *Principal Component Analysis* (PCA) . Given the usual dataset with samples $x_j^i \in \chi$,with a d dimensional feature space, it can represented as

$$\boldsymbol{X} = \begin{bmatrix} x_1^1 & x_2^1 & \dots & x_k^1 \\ x_1^2 & x_2^2 & \dots & x_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_2^n & \dots & x_k^n \end{bmatrix}$$

and then the mean value for each feature is defined as
$\boldsymbol{\mu} := E[\boldsymbol{x}] = (< \boldsymbol{x}_1 > \cdots < \boldsymbol{x}_k >).$, while $\sigma_{ij} = (\Sigma)_{ij} := E[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T]_{ij}$

is defined as the covariance between two features, obtaining its best extimation as $\sigma_{ij} = \frac{1}{n}\sum_{h=1}^{n}(x_i^h - \mu_i)(x_j^h - \mu_j)$. In a vectorial form it is written as

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \dots & \sigma_{1k} \\ \sigma_{21} & \sigma_2^2 & \dots & \sigma_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{k1} & \sigma_{k2} & \dots & \sigma_k^2 \end{bmatrix} \tag{1.23}$$

Eigenvectors of (1.23) individuate new independent directions in the feature space, while associated eigenvalues are the variances around that axes. Since most important vectors are the ones with the higest variance (a high variance means that different classes of instances have different values of that feature), the dimensionality k of the feature space is reduced to $d < k$, taking as base vectors the ones associated to the d eigenvalues with higer magnitude.

Chapter 2

Quantum computing and IBM Q EXPERIENCE

As said in the previous chapters, machine learning tecniques require to deal with big amounts of data, then to train our algorithms we need huge memory resources. To solve this problem, we would need an harware able to manage multiple information contemporaneously. The best solution to this problem seems to arrive from quantum mechanical systems, since they "live" in high dimensional Hilbert spaces and they can be in multiple status at the same time.In this chapter we want to introduce the main concepts about quantum computing and its improvements compared to classical computers. We first start describing *qubits* (building blocks of quantum computers) and how to perform operations with *quantum gates*. Then the IBM Q EXPERIENCE platform will be presented, talking about how to provide access to a real quantum computer. Eventually, IBM QX processors will be briefly described.

2.1 Qubits and computational basis

We know that classical computers store information by *bits*, each assuming value 0 **or** 1.Then we encode information in sequence of bits stored in a *RAM* and processed by *logic gates* . We remark that with N bits we can represent 2^N different numbers and N operations are needed to extract this sequence from the memory. By analogy, we define *qubit* the foundamental entity in which we store information in a quantum computer. They are two levels physical systems, so we can always find a basis of two eigenvectors $|0\rangle := \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $|1\rangle := \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ for each i^{th} qubit and without loss of generality we can describe its state on the *Bloch sphere* (fig

2.1) as:

$$|\psi_i\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle \qquad \theta \in [0,\pi]\, \phi \in [0, 2\pi] \qquad (2.1)$$

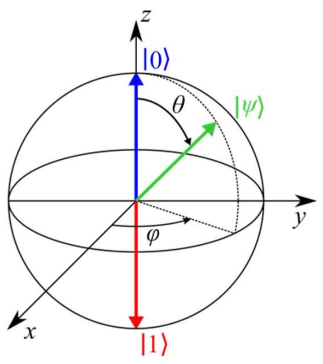


Figure 2.1: We can represent a pure state $|\psi\rangle$ of a two level system as a vector on the surface of the *Bloch sphere*, according to 2.1

Here lies the great advantage of quantum computation : each qubit can be in a superposition of all possible states, so even if measuring it causes the *collapse* of the state on an eigenstate (i.e. we will always have a final result of $|0\rangle$ **or** $|1\rangle$, as a classical bit), our quantum computer will perform on both states simultaneously. Situation becomes much more intrigate (and much more advantageous) dealing with two or more qubits. Let there be N qubits, described by the wave function $|\psi\rangle$, then $|\psi\rangle \in \mathcal{H} = H_1 \otimes H_2 \otimes \cdots \otimes H_N$. Since our new complexive Hilbert space is $n = 2^N$ dimensional, we can span it with a new basis $\{|0\rangle, |1\rangle, \ldots |n\rangle\}$ with

$$|0\rangle = \begin{bmatrix}1\\0\\\vdots\\0\end{bmatrix}, \quad |1\rangle = \begin{bmatrix}0\\1\\\vdots\\0\end{bmatrix} \quad \ldots \quad |n\rangle = \begin{bmatrix}0\\0\\\vdots\\1\end{bmatrix}$$

the so called *computational basis*. This is where real improvements from quantum mechanics come in : the state of our sistem can be an *entangled state*, meaning that we can't find $|\psi_i\rangle \in H_i, i = 1\ldots N$ such that $|\psi\rangle = \otimes_i c_i |\psi_i\rangle \quad c_i \in \mathcal{C}$. This means that our quantum system can represent much more states than its classical counterpart : while for a N qubit system we need $2(2^N - 1)$ parameters to fix our state, for a N bit system we just need N parameters. It now appears clear why quantum computing should lead to an exponential speed up of classical algorithms. Now we need to develop a model to encode information on our qubit; then in the next section we will describe how to implement logic operations on them.
First of all we need to "choose" how to switch from classical representation of information (i.e. a bit string) to a quantum one. We represent the N bit string

$\boldsymbol{x}$ as $\{x_i\}_0^{N-1}$, $x_i \in \{0,1\}$ with a system of N qubit , prepared in the state[25] $|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{i=1}^{N} |\boldsymbol{x_i}\rangle$,where $|\boldsymbol{x_i}\rangle$ encodes for the state with the i^{th} qubit in the state codified by the value of the bit x_i. For example if we have two bit strings $x^1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ and $x^2 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$, we encode this as $|\Psi\rangle = \frac{1}{\sqrt{2}}(|1001\rangle + |0110\rangle)$ with a four qubit system.Then we project this state on the computational basis, obtaining its final representation. This encoding method is called *amplitude encoding* . We will later see how an initial state $|00\ldots0\rangle = (10\ldots0)^T$ can be transformed in such a state.

2.2 Quantum gates and quantum circuits

Given physical systems with the properties of a qubit, we want to perform logical operations on them. Extending the analogy with classical computers, this mean we want to develop *logic gates*. We can simply represent them as unitary operators $\hat{U}$ on the 2^N Hilbert space of our system. First of all, $\hat{U}$ must be unitary to preserve the norm of our state, but this implies that it must be invertible. This is a great difference from classical computation, since according to *Landauer's principle* deleting a bit of information releases an amount of energy $E \geq K_b T \log 2$[5] ,so classical computers are intrinsically disadvantageous under energetical aspects[1]. Let's focus our attention about one qubit gates. They can be represented as 2×2 matrices, allowing us to move the state of our qubit along the surface of the Bloch sphere (the $U(2)$ group). The most important operators are (omitting the^symbol):

- The Hadamard gate $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
- The phase shift gate $R_z(\delta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\delta} \end{bmatrix}$
- Pauli matrices $\sigma_x, \sigma_y, \sigma_z$, forming a basis for our operators (with the identity matrix)

Now we want to underline some important features about these gates. First of all, fixed the measurement axis as z, we immediatly note that $\sigma_x |0\rangle = |1\rangle$ and

[1]Even for classical computers it is possible to define reversible gates, which must have the same number of inputs and outputs. Such a gate can be implemented using control or *ancilla* bits. The universal reversible logic gate is the *Toffoli gate*. Reversible computing is a form of unconventional computing.

$\sigma_x |1\rangle = |0\rangle$, i.e. σ_x acts as a single qubit NOT. Then , fixed the same axis, we interpre the phase shift gate as a rotation around this axis on the Bloch sphere. Infact according to 2.1 we obtain:

$$R_z(\delta) |\psi\rangle = \cos\left(\frac{\theta}{2}\right) + e^{i(\delta+\phi)} \sin\left(\frac{\theta}{2}\right)$$

The Hadamard gate generates superimposed states from eigenstates, i.e.

$$H |0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad H |1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

It is important to note that we can generate every single qubit state from $|\psi\rangle$ just applying a suitable succession of H and R_z gates. Infact given an initial state $|\psi(\theta_0, \phi_0)\rangle$, then

$$R_z(\frac{\pi}{2} + \phi) H R_z(\theta_1 - \theta_0) H R_z(-\frac{\pi}{2} - \phi_0) |\psi(\theta_0, \phi_0)\rangle = |\psi(\theta_1, \phi_1)\rangle \tag{2.2}$$

Now we want to introduce two foundamental two-qubit gates : the *C-NOT* and the *SWAP*. The *controlled-NOT* (C-NOT) gate can generate a two-qubit entangled state starting from a separable one. Its representation on the computational basis is

$$CNOT := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

For example, $CNOT[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)] |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Resuming its truth table we just obtain

$$CNOT |x\rangle |y\rangle = |x\rangle |x \oplus y\rangle$$

where $\oplus$ represents the *mod2 sum*. In this particular implementation we always leave $|x\rangle$ unchanged and, according to its value, we change or not $|y\rangle$ state: that's why we call them *control* and *target* qubits, respectively. The SWAP gate "swaps" the states of the involved qubits; we can represent it in the computational basis as

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can notice that, defined as $CNOT_{ij}$ the gate with control i and target j and $SWAP_{ij}$ the exchange of state between them, then

$$SWAP_{ij} = CNOT_{ij} CNOT_{ji} CNOT_{ij} \tag{2.3}$$

To represent the actual action of these operators on our qubits, we use a circuital model : we represent the qubit state as a line with temporal axes running on the abscissa, then we just draw a "black box" on that line for our operator. In figure 2.2 we can see two qubit in initial states $|0\rangle$ (they are respectiveli q_0 and q_1).The Hadamard gate is applied to the first qubit , then a CNOT gate with q_0 as control and q_1 as target is applied. A set of gates that consists of all one-bit quantum gates ($U(2)$) and the two-qubit CNOT gate is universal in the sense that all unitary operations on arbitrarily many qubits n can be expressed as compositions of these gates. This is why we measure the complexity of an algorithm by the number of elementay gates it needs to perform its task.
From the figure 2.2 it appears clear the circuital model of a quantum computer, even if we still haven' t discussed about how to extract information from it. To obtain the results of our "quantum programm", we have to measure the state of our qubits. We recall that performing measurements, the system will collapse on its eigenstates, so for each qubit we will obtain $|0\rangle$ or $|1\rangle$: this means we can store this string of binary digits on a *classical register*; for analogy we call the first part of our quantum circuit the *quantum register*.

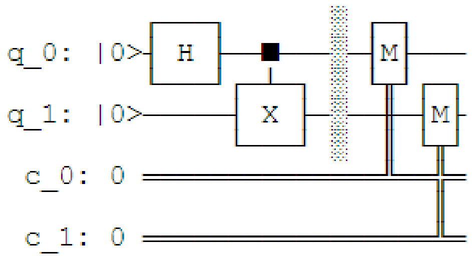


Figure 2.2: The first two line represent the *quantum register* composed of two qubit, q_0 and q_1. The middle zone represents the measumentent operation. Obtained values (M) are stored in c_0 and c_1, the *classical register*.

Now we have all the necessary instruments to understand, with an easy example, the great advantage of working with entangled states : let S be a system of three qubits in the states $|\Phi\rangle = |\Phi_1\Phi_2\rangle$ for the first two and $|0\rangle$ for the last one. Let O be a quantum gate performing on S such that $O\,|\Phi 0\rangle = |\Phi f(\Phi)\rangle$, with a generical function f. If we have $|\Phi\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ (e.g. applying an Hadamard gate on each of them before our quantum circuit) ,then we obtain

$$O\,|\Phi 0\rangle = \frac{1}{2}(|00f(00)\rangle + |01f(01)\rangle + |10f(10)\rangle + |11f(11)\rangle)$$

with one execution of our algorithm we evaluated f in all its possible configurations. Now we have to focus about the second main difference from classical computation : while a classical (deterministic) computer will always give us the same "response" when a program is executed with the same parameters (it works with transistors operating in well defined states), a quantum computer will probably give us different responses due to its intrinsecal probabilistic nature. We know infact that given the system wave function $|\psi\rangle = \sum_i c_i |i\rangle$, then $|c_n|^2$ is the probability to find the system in the $|n\rangle$ state , so if for example $|n\rangle$ encodes for the right solution of our problem, we won't always measure it as our final state (even performing our algorithm in the same conditions). To deal with this "intrinsic imprecision" we perform our algorithm several times and then we assume as the correct answer the one encoded from the state which occurred with the major frequency.

2.3 Qiskit

On 6^{th} March 2017 IBM announced the releasing of *IBM Quantum Experience*, a platform connected to real quantum processors obtained with superconductive materials. Through this platform it is possible to execute programs on these devices via Cloud. After signin up with an account, the user can access public quantum devices and simulators to run his code on. Nowadays there are several software stacks that allow us to access quantum computers, such as *Qiskit*, installable as a Python module. Qiskit enables us to manage with our quantum circuit, real or simulated, with a graphical interface: to build it we just have to drag and drop the gates we want on the line representing the qubit. This is mainly a didactic approach to quantum computing cause it gives us direct access to the state of the system obtained after the measure (repeated several times). In this chapter we will describe another way to manage with quantum processors, using a programming lenguage called QASM (quantum assembly lenguage).
The API (Application Programming Interface) is the set of classes, functions and data structures for interfacing with devices and simulators, and running experiments. In the internal framework of Qiskit we can identify four macro areas; we report their definitions from the official Qiskit "API DOCUMENTATION" [17] :

1. Qiskit Terra: "*Terra provides a bedrock for composing quantum programs at the level of circuits and pulses, to optimize them for the constraints of a particular device, and to manage the execution of batches of experiments on remote-access devices [...]*"

2. Qiskit Aqua : "*Aer provides a high performance simulator framework for quantum circuits using the Qiskit software stack. It contains optimized C++ simulator backends for executing circuits compiled in Terra. Aer also provides tools for constructing highly configurable noise models for performing*

realistic noisy simulations of the errors that occur during execution on real devices."

3. Qiskit Ignis : "*This includes better characterization of errors, improving gates, and computing in the presence of noise [...] Ignis provides code for users to easily generate circuits for specific experiments given a minimal set of user input parameters.*"

4. Qiskit Aer : "*Aqua is where algorithms for quantum computers are built. These algorithms can be used to build applications for quantum computing. [...] To address the needs of the vast population of practitioners who want to use and contribute to quantum computing at various levels of the software stack, we have created Qiskit Aqua.*"

Each of these element comes with internal modules. In this thesis we mainly focused on the "Aqua element" to implement quantum machine learning algorithms. Programs written in qiskit follow a fixed workflow, based on three high level steps:

1. Building circuit

2. Executing the program

3. Analyzing results

To execute this workflow, Quiskit manages three main objects[17] : the `provider`, the `backend` and the `job`. The `provider` gives us access to a group of different `backends` to choose from to lunch our algorithm. By the AER provider we can have access to different simulators, while IBM allows the access to real quantum chips via the IBMQ module. In this work we used both of them. Backends are responsable of running quantum circuits and returning results. They take a `qobj` as input and return a `BaseJob` object. Each execution is identified by an unique address, accessible by the `Job` object; it finds out the execution state at a given point in time (for example, if the job is queued, running, or has failed) and also allow control over the job. We can resume the managing of the workflow as in figure

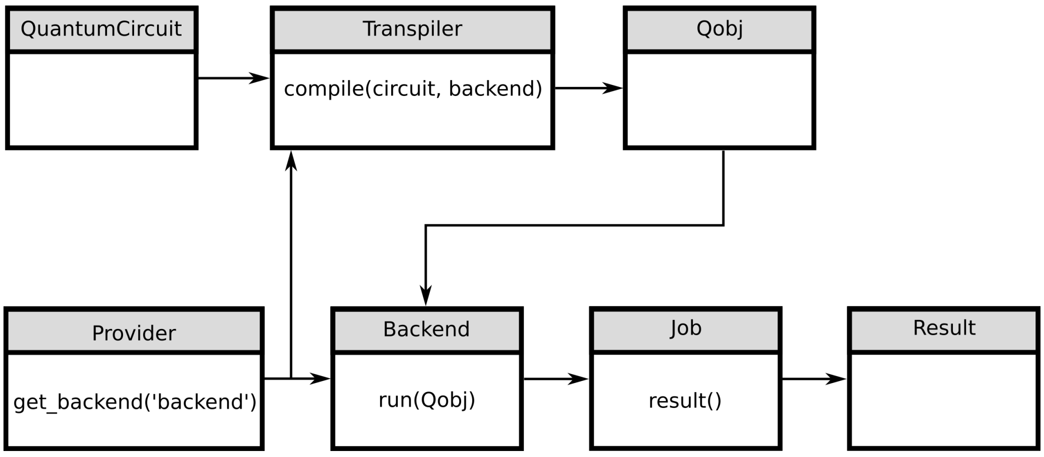


Figure 2.3: Qiskit workflow

2.4 IBM QX processors

2.4.1 Getting access to quantum processors

To perform our experiment on a real backend, we first need to create an Account on the IBM Q Experience platform; we will be provided of a personal API TOKEN, connected to our account, and of 15 credits. Our `circuit` will be divided in `jobs` and any time we perform a `job` on the real `backend`, we get queued and some credits are subtracted from our account. After 24 hours, they will be restored.
At the current time we are allowed to acess four different quantum computers with different characteristics: they distinguish for number of qubits and inner topology.

2.4.2 Transmons : basic concepts

We previously said that a qubit is a system with only two eigenvectors, so we could implement it using any (even approximatively) two levels quantum-mechanical system. Then there could be several ways to implement and control such a system, like spin $\frac{1}{2}$ particles or polarized photons. IBMQ X processors are implemented by superconductive systems. The basic idea is to reduce an LC circuit to a two level systems. Infact writing the Hamiltonian of the system, defining Q and Φ as the charge and the concatenate flux, we obtain

$$\mathcal{H} = \mathcal{U}_C + \mathcal{U}_L = \frac{Q^2}{2C} + \frac{\Phi^2}{2L}$$

where C and L are the capacity and inductance of the circuit. It can be showed that, applying the corrispondence rule, the operators associated to Q and Φ are *conjiugate variables* , i.e $[\Phi, Q] = i\hbar$. We see the Hamiltonian has the same form of one associated to an *harmonic oscillator* , so we can solve it in the same manner, obtaining eigenvalues

$$E_n = \hbar\omega(n + \frac{1}{2}) \quad with \quad \omega = \frac{1}{\sqrt{LC}} \tag{2.4}$$

Then energy levels result equally spaced, so it still isn't a good two level system. To solve this problem, we introduce a *Josephson junction*, a non linear non dissipating inductor with characteristic[7]

$$I = I_c \sin(\mathbf{\Phi}) \qquad V = \frac{\hbar}{2e}\frac{\mathrm{d}\mathbf{\Phi}}{\mathrm{d}dt}$$

with I_c :=*critical current of the junction* , $\mathbf{\Phi} = 2\pi\frac{\Phi}{\Phi_0}$ the *reduced flux* and $\Phi_0 = \frac{h}{2e}$ the *superconductive magnetic flux quantum*. Substituting the Josephson junction to the inductor causes an unevenly separation of energy levels. This means we can address selective transition exciting our system with well defined frequencies, just as happens for electrons in an hydrogenoid atom: that's why this device is often referred to as *artificial atom*. Of course we must keep under control the environment to use it as a two level system, i.e. we must ensure that $k_bT < \hbar\omega$. This model is at the base of the implementation of the *transmon qubit* (transmission line shunted plasma oscillation qubit), the one used to build IBM QX processors.

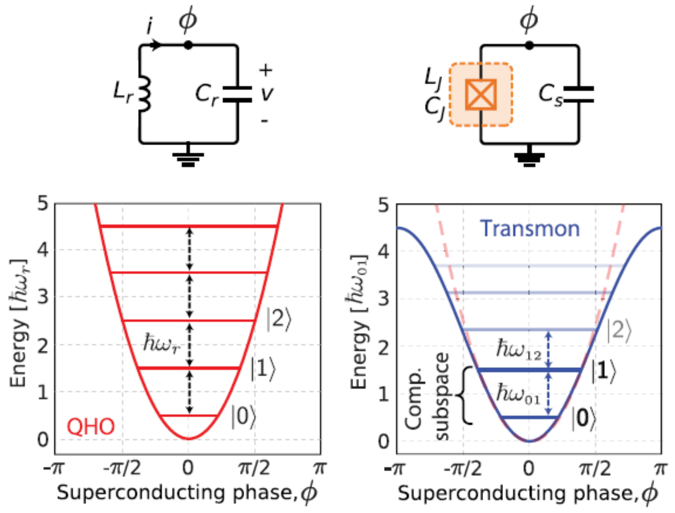


Figure 2.4: [7]The Josephson junction (the cross) introduces anharmonicity in the system. Energetic levels are now unevenly separated

Of course several sources of noise are present in our system, so they must be taken into account to develop a working processor. First of all we know that given the transition rates $\Gamma_{0\to 1}$ and $\Gamma_{1\to 0}$ between the basis states, the thermodynamic equilibrium leads to[7] $\Gamma_{0\to 1} = e^{-\frac{\hbar\omega}{k_b T}}\Gamma_{1\to 0}$. Typical values are $T \approx 20mK$ and $\frac{\omega}{2\pi} \approx 5GHz$, so "exciting" transitions are strongly suppressed. Both processes bring to a depolarization of the qubit, so we can define a *relaxation time* (i.e. *decoherence*) $T_1 = \frac{1}{\Gamma_{0\to 1}+\Gamma_{1\to 0}}$ such that e^{-t/T_1} will account for the relaxation of the qubit[1][7]. Then similarly T_2, the *dephasing time*, is defined to take into account stochastic fluctuations of qubit frequency.

2.4.3 IBM quantum-chips

Accessing to our personal IBM Q account, we can see all public backends provided by IBM and we have access to their characteristics. We resume them in the table 2.1

Table 2.1

$Name$	$Number\,of\,qubit$	$Basic\,gate\,implemented$
ibmq 5 yorktown - ibmqx2 v2.0.0	5	$u1, u2, u3, cx, id$
ibmq vigo v1.0.0	5	$u1, u2, u3, cx, id$
ibmq ourense v1.0.0	5	$u1, u2, u3, cx, id$
ibm 16 melbourne v1.0.0	16	$u1, u2, u3, cx, id$

where the gates $u1,\ u2\,and\,u3$ are represented by the operators[18]

$$U_3(\theta,\phi,\lambda) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\lambda+\phi)}\cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

$$U_2(\phi,\lambda) = U_3(\frac{\pi}{2},\phi,\lambda)$$

$$U_1(\lambda) = U_3(0,0,\lambda)$$

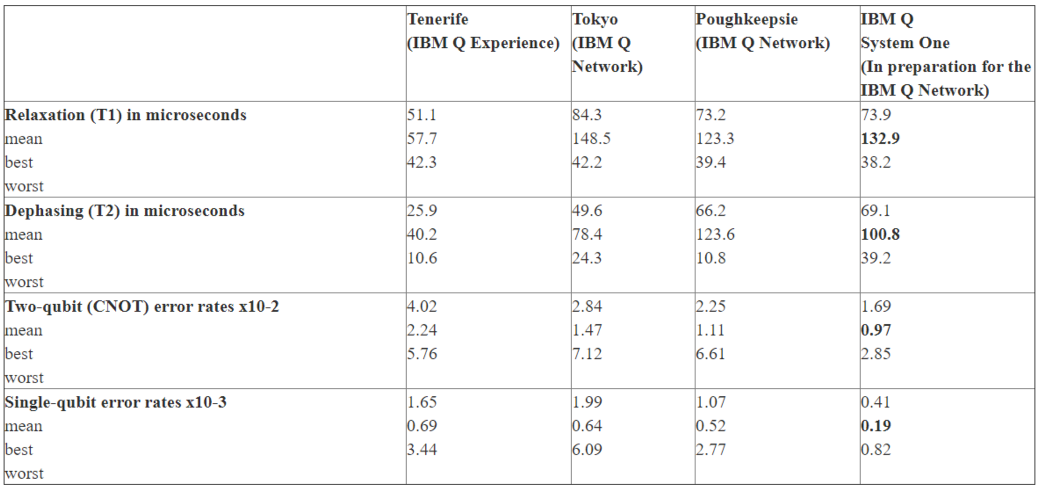
and ic, cx are respectively the *Identity* and the CNOT gates. When building a circuit, these gates can implement the other ones described above. For example :

1. $U_3\left|0\right\rangle = \cos\left(\frac{\theta}{2}\right)\left|0\right\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)$, i.e. we can iniziatilaze any state starting from a $\left|0\right\rangle$ state

2. $U_2(0,\pi) = H$

3. $U_1 = R_z(\lambda)$

But we have to remind about the topology of the quantum chip we are using, since not all qubits are connected each other. The implementation of these gates is obtained via microwave resonators, then they could be a source of *systematic error*. We can't introduce parameters to correct these errors, so IBM calibrates its devices two times a day to reduce their influence. For each gate it is defined a *fidelity factor* simply as the squared module of the inner product between the state we expected and the one we obtain: if they form a δ "angle", then $\gamma = cos^2(\delta)$ is the fidelity factor. Then we can simply obtain the *error rate* as $1 - \gamma$; error rates are avaiable on the IBM Q website. Main characteristics are summarized in the following table.



	Tenerife (IBM Q Experience)	**Tokyo (IBM Q Network)**	**Poughkeepsie (IBM Q Network)**	**IBM Q System One (In preparation for the IBM Q Network)**
Relaxation (T1) in microseconds	51.1	84.3	73.2	73.9
mean	57.7	148.5	123.3	**132.9**
best	42.3	42.2	39.4	38.2
worst				
Dephasing (T2) in microseconds	25.9	49.6	66.2	69.1
mean	40.2	78.4	123.6	**100.8**
best	10.6	24.3	10.8	39.2
worst				
Two-qubit (CNOT) error rates x10-2	4.02	2.84	2.25	1.69
mean	2.24	1.47	1.11	**0.97**
best	5.76	7.12	6.61	2.85
worst				
Single-qubit error rates x10-3	1.65	1.99	1.07	0.41
mean	0.69	0.64	0.52	**0.19**
best	3.44	6.09	2.77	0.82
worst				

Figure 2.5: fundamental metrics of the quantum devices in four recent IBM Q systems. Source :https://www.ibm.com/blogs/research/2019/03/power-quantum-device/

Chapter 3

Quantum machine learning : qSVM

The art of developing algorithms for a potential quantum computer is to use elementary gates in order to create a quantum state that has a relatively high amplitude for states that represent solutions for the given problem. A measurement in the computational basis then produces such a desired result with a relatively high probability. In this chapter we will present different prototypes of quantum hardware implementations to improve classical algorithms performances. In particular the quantum SVM implemented by IBM will be discussed in detail.

3.1 Quantum machine learning

Talking about *quantum machine learning*, we mean a new interdisciplinary research area between quantum physics and informatics. It mainly concernes the possibility to apply quantum hardwares to obtain faster (or, in general, better) implementations of machine learning algorithms. Infact as seen before, the main problem about classical machine learning is the big amount of data and resourches we need to train our model. This limitation is connected to the sequential nature of classical computation, not able to perform parallel operations[1]. Since quantum mechanical systems of N qubits "live" in a 2^N dimensional Hilbert spaces with finite linear operators defined on them, the most natural way to face this problem seems to compute it in this "quantum space". Before proceeding with the tractation, an important specification must be done : saying *quantum machine learning* we mean *quantum enhanced machine learning*, i.e. the execution of a quantum version of an algorithm on classical data (properly encoded). Infact there isn't nowadays a *full quantum* computer: they only exist hybrid architecture, integrat-

[1]Actually, there exist different tecniques of parallel computing using "classical" logic. Rather than execute an algorithm as an ordered succession of instruction to solve a problem, they divide the same problem in multiple independent parts, solved in parallel.

ing both quantum and classical processors. For example a quantum device could have access to data differently form classical computers, implementing a qRAM (*quantum RAM*, so called to underline its analogous function to clasical RAM). Given n bits, we can access $N = 2^n$ different memory cells identified with an univocal code. With qubits the situation is very similar (information is stored in different cells addressed by binary strings), but given a superimposed state as input, we can obtain a superposition of addressed registers as output. To make an example[16], if we denote a the *access register* and $|j\rangle_a$ one of its states associated to a stored address, then if $\sum_j \psi_j |j\rangle_a$ is the input of the qRAM, the output will be $\sum_j \psi_j |j\rangle_a |D_j\rangle_d$, where $|D_j\rangle_d$ id the j^{th} state from the data register d. If the memory array (both clasical or quantum) is disposed in a d dimensional lattice, then $O(\sqrt[d]{N})$[16] actions are required to access the memory address. Actually such an architecture would be computationally expansive and noisy due to decoherence times. Despite this, it is possible to implement a qRAM that requires $Olog(N)$ operations to address a memory address.

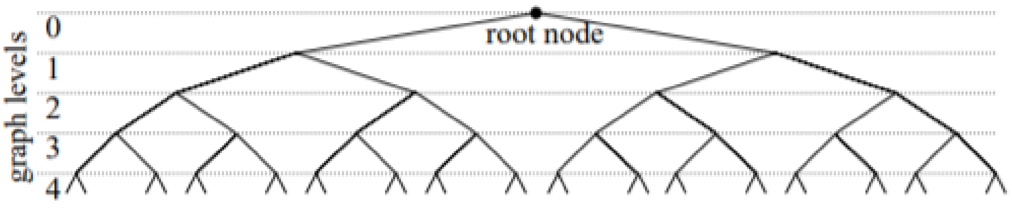


Figure 3.1: [16]A RAM can be represented as a graph. The lowest level represents the data register. At each node "0/1" can be interpretes as "go left/right", so n nodes can address 2^n memory cells. In a qRAM it is possible to cross more paths at the same time

But quantum machine learning algorithms don't only benefit of a different memory addressign system : the main difference from the classical ones is that they are based on *soubroutines* which act as *oracles* (i.e. black boxes) and perform numerous tasks in less time than any classical program. Some of them are :*quantum Fourier transformation* (QFF) , *Detush-Jozsa algorithm*[2] or *Groover algorithm*[3] :all of them perform task a classical computer can do, but in reduced times. We have to inform the reader about the existence of some quantum algorithm which can perform in polynomial time tasks resulting P or $NP - C$ for classical ones (at the "state of the art") : for example, with *Shor's algorithm* we can perform the integer factorization in polynomial time.

[2]Given a constant or balanced function $f : \{0,1\}^m \to \{0,1\}$, the algorithm can determine if f was constant or balanced from the output

[3]Roughly speaking, if a function $y = f(x)$ can be evaluated on a quantum computer, Grover's algorithm calculates x when given y

3.2 qSVM

3.2.1 qSVM implementations

In literature it is posible to find different possible implementations of the SVM classificator using a quantum hardware. For example, a first implementation prensented in the early 2000's was mainly based on a variant of *Grover search algorithm*[21].
Alternatively it could be possible to differently formulate the SVM problem form 1.17 : rather than use KKT conditions, one can solve a linear system[22]: it is called LS-SVM. Then a quantum algorithm to solve linear systems could be used, leading to the implementation of a *quantum LS-SVM*, which requires a qRAM[23]. Also IBM implemented two different versions of the classificator[11]: a *quantum variational algorithm* and a *Quantum kernel estimation*[12].

3.2.2 IBM qSVM

Now we focus our attention on the *quantum kernel extimation* methon introduced above and implemented by Qiskit. It is a quantum enhanced algorithm, which requires classical data to perform (i.e. it doesn't need a qRAM). It is implemented using an integrated classical/quantum circuit. In this chapter we describe its quantum circuit.
The biggest difference from the classical counterpart lies in the algorithm itself : given a N dimensional feature space and M samples, a classical SVM , according to 1.17, has to compute $\frac{M(M-1)}{2}$ scalar products to build the kernel matrix (each one of them requirung $O(N)$ operations) , then it requires $O(M^3)$ [25]operations to solve the quadratic problem in the dual space. Eventually, we have to [25] $O(log(\epsilon^{-1}poly(N, M)))$ (fixed an accuracy ϵ) operations to train our classical SVM. On the other hand, a qSVM can perform "naturally" scalar products (evaluating the Kernel matrix) in parallel ,endig up with a total of $O\log_2(NM)$ [25]. To perform the scalar product, we use the *SWAP TEST ROUTINE* implementing the circuit in figure 3.2.

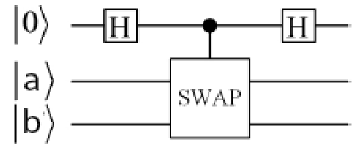


Figure 3.2: The SWAP test is implemented with two Hadamard gates applied to the control qubit and a C-SWAP (*controlled SWAP*). Controlled gates perform the operation between the target qubits if the control qubit state is $|1\rangle$

Given a system of three qubits, respectively in the states $|0\rangle$, $|a\rangle$, $|b\rangle$, at the stage 2 we obtain $|\Psi\rangle = \frac{1}{\sqrt{2}}(|0ab\rangle + |1ab\rangle)$. Now the C-SWAP gate leads us to the superposition of : $\frac{1}{\sqrt{2}}|0ab\rangle \rightarrow \frac{1}{\sqrt{2}}|0ab\rangle$ and $\frac{1}{\sqrt{2}}|1ab\rangle \rightarrow |1ba\rangle$. Now we apply the second H gate to the first qubit and the final state becomes :
$|\Psi\rangle = \frac{1}{2}|0\rangle[|ab\rangle + |ba\rangle] + \frac{1}{2}|1\rangle[|ab\rangle - |ba\rangle]$. This means we can evaluate $P(|0\rangle) = |\langle 0|\Psi\rangle|^2 = \frac{1}{2} + \frac{1}{2}|\langle a|b\rangle|^2$. Then we can have a direct misure (repeating the cycle several times) of $|\langle a|b\rangle|$. Another way to implement the same idea (the one adopted by IBM) consists in encoding each "classical" string of bit $\boldsymbol{x}$ with a suitable map $\Phi(\boldsymbol{x})$. Let's say $\boldsymbol{x}$ is an element of ur test set, then we apply the feature map $\Phi(\boldsymbol{x})$ and we can perform on it an unitary transformation of the form $U_{\Phi(\boldsymbol{x})}$ and then we can implement it all as

$$\mathcal{U}_{\Phi(\boldsymbol{x})} = U_{\Phi(\boldsymbol{x})}H^{\otimes n}U_{\Phi(\boldsymbol{x})}H^{\otimes n} \qquad n = \text{number of qubits} \tag{3.1}$$

such that the state associated to these qubits will be $|\Phi(\boldsymbol{x})\rangle = \mathcal{U}_{\Phi(\boldsymbol{x})}|0\rangle^{\otimes n}$. In the same way we can encode on a second qubit another element from the sample, let's say $\boldsymbol{z}$, and then we can define its feature map $\mathcal{U}_{\Phi(\boldsymbol{z})}$. Then if we want to evaluate the module of the inner product $|\langle\Phi(\boldsymbol{z})|\Phi(\boldsymbol{x})\rangle| = \left|\langle 0|^{\otimes n}\mathcal{U}^{\dagger}_{\Phi(\boldsymbol{z})}\mathcal{U}_{\Phi(\boldsymbol{x})}|0\rangle^{\otimes n}\right|$, we just have to implement the right circuit (see fig. 3.3). This means that, as with the SWAP TEST, we can extimate the module of the inner product between our vectors $\Phi(\boldsymbol{x})$ and $\Phi(\boldsymbol{z})$ (i.e. the Kernel matrix) just measuring the frequency of the state $|0\rangle^{\otimes n}$ at the end of our circuit (assuming $|0\rangle^{\otimes n}$ as initial state).

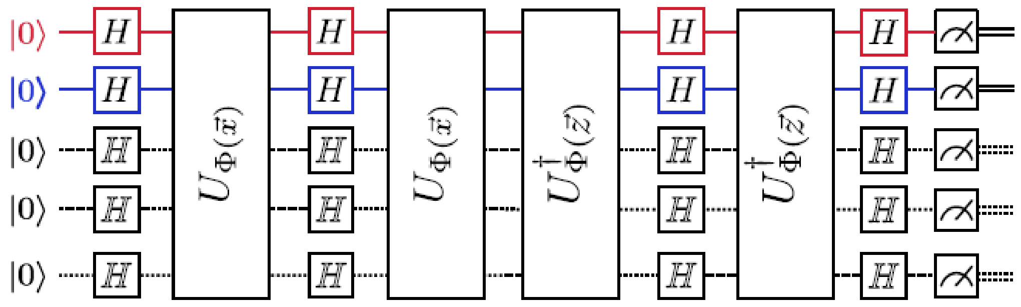


Figure 3.3: [12]Given a map $\mathcal{U}_{\Phi(\boldsymbol{x}_i)}$ for any classical input $\boldsymbol{x}_i$, it is possible to perform the inner product between them.

Currently, it is possible to implement the following feature maps from 3.1.
$U_{\Phi(\boldsymbol{x})} = \exp\{(i\sum_{S\subseteq[n]}\Phi_S(\boldsymbol{x}))\prod_{i\in S}Z_i\}$, with :

1. *FirstOrderExpansion* :$S \in \{0, 1, \ldots n-1\}$, so $\Phi_i(\boldsymbol{x}) = x_i$

2. *SecondOrderExpansion*: $S \in \{0, 1, \ldots n-1, (0,1), (0,2) \ldots (n-2, n-1)\}$ and $\Phi_i(\boldsymbol{x}) = x_i$, $\Phi(\boldsymbol{x})_{ij} = (\pi - x_i)(\pi - x_j)$

3. *PauliZExpansion*: $S \in \{\binom{n}{k}\, combinations\,, k = 1 \ldots n\}$ and $\Phi_S(\boldsymbol{x}) = x_i$ if k=1, $\Phi_S(\boldsymbol{x}) = \prod_S (\pi - x_j), j \in S$otherwise

From S we can see that only interactions between couples of qubits are implemented. The corresponding circuit between the qubits i and j can be seen in figure .

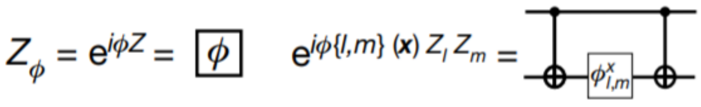

$Z_\phi = e^{i\phi Z} = \boxed{\phi}$
$e^{i\phi\{l,m\}(\mathbf{x})\, Z_l Z_m} =$
$\phi^x_{l,m}$

Figure 3.4: [12]Two qubit operations from the feature map $U_{\Phi(\boldsymbol{x})}$ are implemented with CNOT and Z gates.

Chapter 4

Experiments

We have now developed all necessary instruments to implement a quantum support vector machine algorithm both on a real and simulator backend. We will test its performances on the benchmark datasets *Iris* , *Wine* and *Digits*, avaiable online on the *UCI Repository*. For each of them we measured the accuracy scored by both classical and quantum implementations of the algorithm on the same set. Then we analyzed differences between simulated and real results obtained with the qSVM. All algorithms are implemented using `Python 3.7.4.` qSVM has been implemented by `Qiskit.aqua.0.6.0` module, SVM by `scikit-learn 0.21.3` module.

4.1 Datasets

We decided to use three different datasets[24] for different motivations :

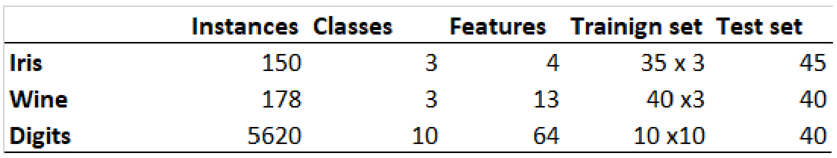
1. Iris dataset is a benchmark dataset to classify iris flower starting from measurements of petal dimensions.It is a very simple dataset. It contains a small number of attributes and instances, so it is a good starting point to test a classification algorithm.

2. Wine dataset consists of measures about physical properties from three different types of wine (alcohol levels, magnesium concentration, ...). It contains a restricted numeber of classes, but each instance has several features associated. It could give us important informations about algorithm ability to manage big feature spaces [1].

[1]Actually, as said before and as will be better explained later, each dataset has been preprocessed via PCA reducing the number of features to two. What we really tested is indeed the ability of the algorithm to find the correct hyperplane even if some information is lost

3. Digits dataset is the "hardest one" used. This dataset contains handwritten digits. Each instance originates from a block of 32x32 bitmaps, divided into nonoverlapping blocks of 4x4. The number of "on" pixels is counted in each bloch, so we reduced to a 8x8 matrix as instance. The aim is to recognize the input digit. It contains a big amount of instances, several features per one and numerous classes. With this dataset we could have a 360 degree overview of the (q)SVM performances.

We resume in detail their characteristics in the following table :



	Instances	Classes	Features	Trainign set	Test set
Iris	150	3	4	35 x 3	45
Wine	178	3	13	40 x3	40
Digits	5620	10	64	10 x10	40

Figure 4.1: Characteristics of used datasets. In this table we also reported the dimension of the trainig set and the test set used for our experiments

4.2 Experiment setting

First of all, whe had to preprocess datasets. Currently IBM processors can implement only two/tree features qSVM, due to internal topology, as evident from figure of a quantum processor. For our experiment, we used a two features implementation (i.e. we set $|S| < 2$ from feature maps described in section 3.2).

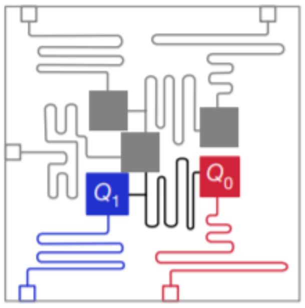


Figure 4.2: [12]The internal structure of a 5 qubit quantum processor used by IBM. No more than three qubit can be connected. In our experiments we only used two qubits, to be coherent with figure 2.2.

To do so, we applied PCA to all datasets and reduced the number of features to two; results are shown figure 4.3

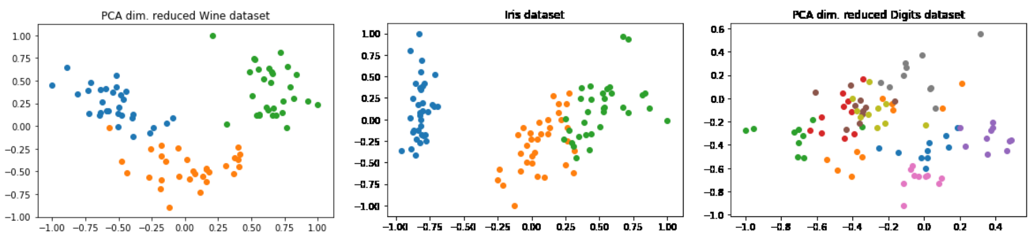


Figure 4.3: Feature distribution of the three dataset after the application of PCA.

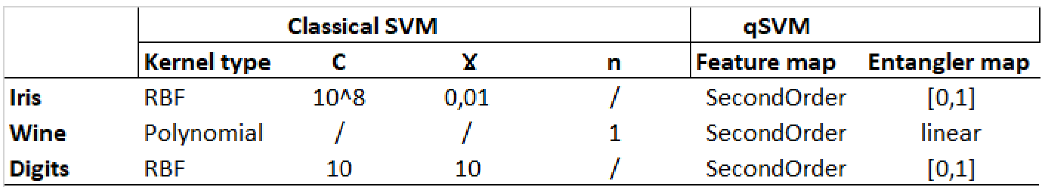
For each dataset different kernels (with different parameters) have been tested.For the qSVM, due to the long times of execution, this "parameter testing" has been executed using the `StatevectorSimulator`, one of the simulators implemented by *Qiskit Aer*. It is faster than the others cause it doesn't take in account any source of noise and/or error, then it only performs the circuit once and returns the probability of final states. It provides a good idea about the scoring of parameters, even if it isn't absolutely reliable to measure the accuracy of the quantum chip. For classical SVM we looked for the best configuration among (referring to section 1.6):

1. Polynomial kernel of degree n, $n \in [1, 10]$
2. Radial basis function kernel (RBF) in the form $K(\boldsymbol{x}^i, \boldsymbol{x}^j) = \exp\left(-\gamma \|\boldsymbol{x}^i - \boldsymbol{x}^j\|^2\right)$ $C \in [10^{-2}, 10^{10}], \gamma \in [10^{-9}, 10^3]$. For each of them 13 evenly spaced values have been tested. C comes from 1.20.

In the same way we tested different configurations for the qSVM among kernels presented in section 3.2 and the following `entangler maps` :

1. `full` entangler map : all qubits are entangled, according to the topology of the processor used
2. `linear` entangler map : each qubit is entanged with the next one
3. `[0,1]` only q_0 and q_1 qubits are entangled

The best configurations found for each dataset are reported in the following table:



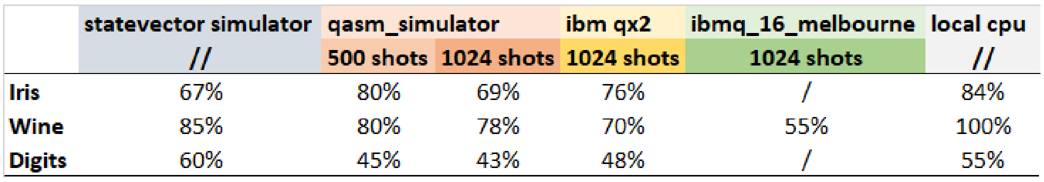
	Classical SVM				qSVM	
	Kernel type	C	γ	n	Feature map	Entangler map
Iris	RBF	10^8	0,01	/	SecondOrder	[0,1]
Wine	Polynomial	/	/	1	SecondOrder	linear
Digits	RBF	10	10	/	SecondOrder	[0,1]

Figure 4.4: Kernel configurations which scored the best accuracy on each dataset. We used these parameters in our experiments.

Fixed the algorithms, they have been trained and tested on the same sets, then accuracies achieved have been compared.

4.3 Results and discussion

Both algoriths have been used on the same datasets, each one with the best parameters provided by fig 4.4. Accuracies scored by simulators, quantum chips and local CPU (classical SVM) are resumed in figure 4.5



	statevector simulator	qasm_simulator		ibm qx2	ibmq_16_melbourne	local cpu
	//	500 shots	1024 shots	1024 shots	1024 shots	//
Iris	67%	80%	69%	76%	/	84%
Wine	85%	80%	78%	70%	55%	100%
Digits	60%	45%	43%	48%	/	55%

Figure 4.5: Accuracies obtained on each dataset for each implementation of the algorithm. We also reported results from simulators as sources of interesting considerations. `statevector simulator` (as well as classical SVM) executes the algorithm just one time

First of all, it is evident the great difference between results from the two different simulatos: we remark that we used `statevector_simulator` only to get an idea about the accuracy of the current configuration of the algorithm, since it is much faster than `qasm_simulator`. Infact, just to get an idea, the first one required, on average, about 6 minutes, while the second took about 3h (averaged on 1024 shots). This evidences the exponential time required by a classical computer to simulate a quantum one, since the classical SVM required lesser than one second.

From the difference between accuracies scored on the same simulator but increasing the number of repetitions, we see that results strongly change. It is probably due to the presence of final $|0\rangle$ states with low probability of occurrence, then a great number of repetitions is needed to accurately extimate it measuring the frequency. This leads to a strange situation, since for the *wine* and *digits* dataset it could seem that accuracy scored with 1024 run is worse than the one obtained with 500 run. This probably depends on the fact that the qSVM algorithm tries to approximate the probability measuring a frequeny (see section 3.2), which fluctuates around this value (*Bernoulli theorem*). Then it finds a "wrong" hyperplane for the training instances, but a "lucky" one for the validation set. That's why we only executed 1024 times the algorithm on real backends.

It seems that the system used to simulate errors and noise isn't too accurate : results from `qasm_simulator` and real backends are quite different. For the Iris dataset this led again to a strange situation cause of the same motivations told before: the wrong hyperplane found during the trainig phase works fine on the specific training set, even if it is just a fortuity. This strange result can be a great starting point to impove quantum systems, cause good simulation of a real system means a deep understanding of all causes of noise and error.

Then another big difference arises among quantum processors: it is evident that `ibmq_16_melbourne` is less accurate than `ibm_qx2`: this is probably because our algorithm is explicitly designed to work on a five qubit architecture, so the 9 unused qubits are just source of noise.

Now we look at the differences between accuracy performances between classical and quantum SVM implemented on the classical and quantum CPU. First of all, some specifications should be done about the datasets used: they are benchmark datasets composed of little amounts of instances, so a serious statistical analysys can't be done (we are trying to evaluate the global accuracy of our classificator basing the result on just up to 40 instances), but the obtained results surely are a good starting point to get an idea about the state of the art of quantum computers. Excpet for the wine dataset, accuracy results are comparable, even if the qSVM is less accurate on average. This can be due to inner noises and errors or to the fact that the feature map used is just inappropriate: real advantages of qSVM become evident when the suitable feature map is too hard to be classically evaluated.

Eventually, we must talk about the strong limitation imposed on the number of features: this led to the loss of big amounts of information about the dataset (e.g. for the Digits we tried to "resume" 64 features in just two. Even from figure 4.1 was clear that it wasn't enough to distinguish all classes).
One final consideration must be done about times taken from these two implementations: even if the quantum speed up is surely achievede thanks to this "direct evaluation" of the kernel matrix, the user can't benefit of this. Infact, as explained before, IBM Q processors are public, so the access is provided with a system of queueing and personal credits, then each implementation of the qSVM required several days to be performed.

Conclusions and future developments

In this short thesis we presented main concepts about machine learning and supervised learning, presenting an implementation of a multiclass classifier with the SVM algorithm. We did it because these tecniques are wideley used in reasearch (high energy, astrophysics, new materials, chemistry...) and then we presented a possible improvement of them using a quantum computer. So we presented basic concepts about quantum computation theory and why it should exponentially speed up execution time of classical algorithms (or even perform tasks not possible to a classical computer). In particular, a detailed discussion about the qSVM has been done, explaining the circuit implemented by IBM. Nevertheles, with a simple experiment, we found that we are still far away from a quantum computer able to perform "real life" tasks , even if leading companies in the sector such as Google, Microsoft or NASA (as well as IBM) are collaborating with academic researchers to find new ways to power up their quantum computers (to achieve a better controll of the qubits, a better isolation from the environment, bigger number of qubits per chip ...). This means that contemporaneous tecnologies aren't still enough advanced to manage a full working superconductive qubit, with very small times of coherence. Then maybe a change of paradigma could occurr (we remark that the transmon isn't the only existing implementation of the qubit. Trapped ions, polarized photons, atomic spins of molecules... are valid theorical alternatives, even if the transmon presents the advantage of being tunable with simple electronic devices) But this is a field in continuous evolution: IBM has already announced the first commercial line of 20 qubit quantum computers (*IBM Q System One*), while on 18 September 2019 it announced that it will soon make avaiable a 53 qubit system on IBM Q Network. With such a system it could be possible to demonstrate the *quantum supremacy*[2]. Industries and researchers will benefit of the advantage of a computer able to manage with huge amounts of

[2]Quantum supremacy is achieved when a formal computational task is performed with an existing quantum device, but the same task cannot be performed using any known algorithm running on an existing classical supercomputer in a reasonable amount of time

data and to simulate real quantum system. Chemistry, biology as well as physics will surely benefits of these properties.

Bibliography

[1] Dr. Christine Corbett Moran. *Mastering Quantum Computing with IBM QX*. Packt publishing, 2019.

[2] Ethem Alpaydin. *Introduction to Machine Learning, Second Edition*. The MIT Press, 2010.

[3] Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning, Second Edition*. Packt publishing, 2017.

[4] David McMahon. *Quantum computing explained*. IEEE Computer Society, 2007.

[5] Benenti Giuliano, Giulio Casati, Giuliano Strini. *Principles Of Quantum Computation And Information - Volume I: Basic Concepts*. World Scientific Publishing, 2004.

[6] B. J. Chelliah, S. Shereyasi , A. Pandey, K. Singh. *Experimental Comparison of Quantum and Classical Support Vector Machines*. IJITEE. ISSN: 2278-3075, Volume-8 Issue-6, April 2019

[7] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, W. D. Oliver. *A quantum engineer's guide to superconducting qubits*. Appl. Phys. Rev. 6, 021318 (2019).
`https://doi.org/10.1063/1.5089550`

[8] Havenstein, Christopher; Thomas, Damarcus; and Chandrasekaran, Swami. *Comparisons of Performance between Quantum and Classical Machine Learning*." SMU Data Science Review: Vol. 1 : No. 4 , Article 11, 2018.
`https://scholar.smu.edu/datasciencereview/vol1/iss4/11`

[9] J. Eisert, M.M. Wolf. *Quantum computing*, 2006.
`arXiv:quant-ph/0401019`

[10] M. Schuld, I. Sinayskiy, F. Petruccione. *An introduction to quantum machine learning*, 2014.
`arXiv:1409.3097`

[11] Seth Lloyd, Masoud Mohseni, Patrick Rebentrost. *Quantum algorithms for supervised and unsupervised machine learning*, 2013.
`arXiv:1307.0411`

[12] Vojtech Havlicek, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, Jay M. Gambetta. *Supervised learning with quantum enhanced feature spaces*, 2018.
`arXiv:1804.11326`

[13] Patrick J. Coles, Stephan Eidenbenz, Scott Pakin, Adetokunbo Adedoyin, John Ambrosiano, Petr Anisimov, William Casper, Gopinath Chennupati, Carleton Coffrin, Hristo Djidjev, David Gunter, Satish Karra, Nathan Lemons, Shizeng Lin, Andrey Lokhov, Alexander Malyzhenkov, David Mascarenas, Susan Mniszewski, Balu Nadiga, Dan O'Malley, Diane Oyen, Lakshman Prasad, Randy Roberts, Phil Romero, Nandakishore Santhi, Nikolai Sinitsyn, Pieter Swart, Marc Vuffray, Jim Wendelberger, Boram Yoon, Richard Zamora, Wei Zhu *Quantum Algorithm Implementations for Beginners*, 2018.
`arXiv:1804.03719`

[14] David C. McKay, Thomas Alexander, Luciano Bello, Michael J. Biercuk, Lev Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, Jay M. Gambetta *Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments*, 2018
`arXiv:1809.03452`

[15] A. Barenco (Oxford), C. H. Bennett (IBM), R. Cleve (Calgary), D. P. DiVincenzo (IBM), N. Margolus (MIT), P. Shor (AT&amp;T), T. Sleator (NYU), J. Smolin (UCLA), H. Weinfurter (Innsbruck) *Elementary gates for quantum computation*, 1995
`arXiv:quant-ph/9503016`

[16] Vittorio Giovannetti, Seth Lloyd, Lorenzo Maccone *Quantum random access memory*, 2007
`arXiv:0708.1879`

[17] Qiskit API documentation, 2019 :
`https://qiskit.org/documentation/`

[18] *Learn Quantum Computation using Qiskit*, 2019
`https://community.qiskit.org/textbook/`

[19] A collection of Jupyter notebooks showing how to use Qiskit that is synced with the IBM Q Experience , 2019
`https://github.com/Qiskit/qiskit-iqx-tutorials`

[20] Cynthia Rudin. *Kernels, MIT 15.097 Course Notes.*
`https://ocw.mit.edu/courses/sloan-school-of-management /15-097-prediction-machine-learning-and-statistics-spring-2012/ lecture-notes/MIT15_097S12_lec13.pdf`

[21] Davide Anguita, Sandro Ridella, Fabio Rivieccio, Rodolfo Zunino *Quantum optimization for training support vector machines, Neural Networks* Volume 16, Issues 5?6, 2003, Pages 763-770, ISSN 0893-6080,
`https://doi.org/10.1016/S0893-6080(03)00087-X`

[22] Suykens, Johan, Vandewalle, Joos *Least Squares Support Vector Machine Classifiers. Neural Processing Letters* ,1999 9. 293-300. 10.1023/A:1018628609742.

[23] Li Zhaokai, Liu Xiaomei, Xu Nanyang, Du jiangfeng *Experimental Realization of Quantum Artificial Intelligence* ,2014
`arXiv:1410.1054`

[24] Dua, Dheeru and Graff, Casey *UCI Machine Learning Repository*. Irvine, CA: University of California, School of Information and Computer Science ,2019
`http://archive.ics.uci.edu/ml`

[25] Patrick Rebentrost, Masoud Mohseni, Seth Lloyd *Quantum support vector machine for big data classification*, 2014.
`arXiv:1307.0471`