Università degli Studi di Milano - Bicocca

FACOLTA' DI SCIENZE MATEMATICHE FISICHE E NATURALI
Corso di Laurea in Fisica

Data classification using quantum kernels

Tesi di Laurea Triennale

Relatore:
Dr. Andrea Giachero

Correlatore:
Dr. Leonardo Banchi

Candidata:
Elena Agostoni
Matricola: 814980

2020-2021

23 Novembre 2021

Ai miei nonni

Abstract

Quantum computing is one of the most promising new technologies. Unlike classical computers, quantum computers have the peculiarity of being made up by qubits. They permit us to exploit the properties of quantum mechanics to better the performance and computational capabilities. One of such properties is quantum entanglement, that can be obtained by applying *quantum gates* like the Hadamard and the C-NOT gate.

Furthermore, quantum computing allows us to have, instead of the classical binary state, a superposition of states like $|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where $|0\rangle$ and $|1\rangle$ are the quantum states representing the qubits.

Machine learning is the branch of Artificial Intelligence that takes inspiration from the learning ability of the brain, trying to emulate it via algorithm. *Support vector machines* and *kernels* are used for classification problems, and they belong to the class of *supervised* machine learning methods, meaning that they learn by example.

Support vector machines are a learning model that map a n-dimensional dataset into a space of dimension m, with $m > n$, until the data appear to be separable by a hyperplane.

The kernel function is employed to simplify and speed the operation, because the computational cost rises while m gets bigger. This is the reason why it is convenient to use the quantum counterparts of these classification methods, like *quantum support vector machines* and *quantum kernels*.

Qubits are fragile states, prone to error, but their usage allows a noticeable computational speed-up. Thanks to quantistic properties, quantum computing can manage faster high dimensional spaces.

In the first part of this thesis it is described the meaning of machine learning, focusing on kernel methods and support vector machines and explaining the definitions of support vector, margin and hyperplane and how they are used in data classification. The case of separable data and overlapped data is then given and analysed, together with the fundamental statistical learning notions.

Then, Python codes from *scikit-learn* are shown so that we can familiarise with kernel and SVM classification methods.

After that, quantum computers are introduced, describing the *quantum stack* and how it is possible to go from the qubit hardware to the user interface, touching on the *Qiskit* programming language. After an introduction on quantum computing, are given examples of quantum gates and quantum circuits and their usage.

Then, *quantum machine learning* is explained together with the definitions of quantum kernel, quantum feature map and quantum SVM.

Qiskit, a Python-based programming language, with the `qiskit_machine_learning`, module are used through the IBM Quantum Lab platform, that lets users run code on real quantum computers.

A classification example is then given analysing the `ad_hoc_dataset` with the algorithm support vector machine classification `SVC` from `scikit-learn`. Next, the kernel matrix is computed though the class `QuantumKernel` and `ZZfeaturemap`. The possibility to define a *custom* kernel is shown. Subsequently the matrixes *ad_hoc training kernel* and *ad_hoc testing kernel* are calculated.

After that, the `breast_cancer` dataset from `qiskit.ml.dataset` is analysed with the `QSVM` algorithm and the `ZZfeaturemap`.

Finally for each dataset we compare for the classical and quantum case, finding the *success ratio*, the classification time and kernel matrix. This demonstrates how quantum computing can be more efficient.

Contents

1 An overview of classical machine learning **1**
- 1.1 What is classical machine learning? . . . 1
- 1.2 How does machine learning work? . . . 2
- 1.3 The danger of overfitting and underfitting . . . 3
- 1.4 Supervised learning . . . 3
- 1.5 Support vector machines . . . 5
- 1.6 Fundamental notions of statistical learning . . . 6
  - 1.6.1 The case of separable data . . . 6
  - 1.6.2 The case of overlapping data . . . 7
  - 1.6.3 Solving the problem with Lagrange multipliers . . . 9
- 1.7 Kernels . . . 12
  - 1.7.1 Gram matrix and graph kernels . . . 14

2 An overview of quantum computing **15**
- 2.1 The basis of quantum computing . . . 16
  - 2.1.1 The Bloch sphere . . . 16
  - 2.1.2 Quantum gates . . . 18
  - 2.1.3 An example of a quantum circuit . . . 19
- 2.2 Quantum computers . . . 20
  - 2.2.1 The quantum stack . . . 20
  - 2.2.2 NISQ computers . . . 23
- 2.3 Qiskit . . . 23
  - 2.3.1 The Qiskit *elements* . . . 24

3 Classification methods with scikit-learn **26**
- 3.1 What is scikit learn? . . . 26
- 3.2 The SVC class . . . 26
- 3.3 Multi-class classification . . . 28
  - 3.3.1 One-vs-one . . . 29
  - 3.3.2 One-vs-rest . . . 30

3.3.3 Multi-class classification in Python 30
3.4 Score and probabilities 31
3.5 Unbalanced problems 31
3.6 Kernel Functions 32

4 Quantum machine learning **35**
4.1 Why quantum machine learning? 35
4.2 Quantum computing and classification methods 35
4.2.1 Analyzing the problem 36
4.2.2 Analyzing the components 36
4.2.3 Quantum kernel estimation 37

5 Classification with a Qiskit quantum algorithm **39**
5.1 Quantum kernel classification 39
5.2 A comparison between classical and quantum classification 44

6 Conclusion **47**
Code appendix 49

"Nature isn't classical, dammit,
and if you want to make a simulation of nature,
you'd better make it quantum mechanical,
and by golly it's a wonderful problem,
because it doesn't look so easy."
Richard Feynman (1981)

Chapter 1

An overview of classical machine learning

As Arthur Samuel, the inventor of the term itself, described it, machine learning is the "field of study that gives computers the ability to learn without being explicitly programmed" [Bhavsar et al., 2017]. The year was 1959, far before today's widely spread usage and more in-depth knowledge of machine learning, but this statement is not far from the truth. The question "Can machines think?" needs to be substituted with "Can machines do what we, as thinking entities, can do?" [Turing, 1950].

1.1 What is classical machine learning?

Machine learning is a branch of artificial intelligence which takes inspiration from human intelligence and the unmatched ability it has to learn, and tries to imitate it trough computer algorithms.

In fact, many machine learning tasks like finding and replicating patterns and speech, image recognition and outcome forecasting based on previous knowledge, come naturally for the human brain.

It is often difficult to grasp the difference between machine learning and artificial intelligence. To try and explain this, one could say that machine learning learns and predicts via passive observations, while artificial intelligence needs an agent that interacts with the environment to successfully learn.

Machine learning is a subset of artificial intelligence, and it encases *deep learning* [Fig. 1.1], which uses multiple-layered neutral networks.

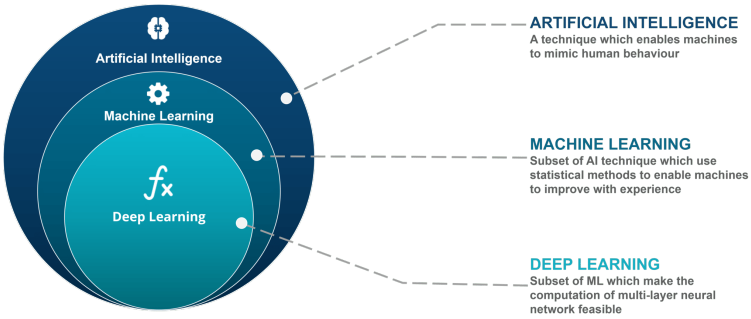


Figure 1.1: Machine learning is a subset of artificial intelligence.

Let us take into account that algorithms that have been successful in the past will continue to be successful in the future. With this assumption, machine learning programs can solve problems that could potentially be more difficult for a human to solve via algorithm than they are for the machine to "learn".

1.2 How does machine learning work?

Quoting Tom Mitchell "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E" [Hackeling, 2014].

Meaning, if one was shooting an arrow and trying to hit the bulls-eye (task T), they could use a machine learning algorithm with data about past arrow shots (experience E). If it has successfully learned, the person can use it to predict future patterns of trajectory for their arrow after it has left the bow, and make the best shot possible (performance measure P).

The first process is to get the algorithm to analyse the input data, so it can find patterns, classify them, or make predictions. Then, a function is implemented to evaluate the accuracy of the aforementioned prediction. The last step is to optimize the process. Weights can be added and adjusted so the model can fit better to the expected result.

This operation will be repeated many times, until the needed threshold of accuracy is reached.

Optimization is a crucial part of machine learning methods. Most of the time a machine learning problem is formulated taking into account a minimization of a *loss function*, which represents the discrepancy between the prediction of the training method and the instances of the presented problem.

The loss function is a function that computes the distance between the *current* output of the algorithm and the *expected* output. It can be categorized into two groups. One for classification (dealing with discrete values, 0,1,2...) and the other for regression (dealing

with continuous values). The loss function is of fundamental importance to evaluate how the algorithm is modeling the data, since it is a way to measure whether the algorithm is properly working [Chollet, 2021]. The measurement is used as a feedback signal to adjust the way the algorithm works. This adjustment step is what we call *learning*.

1.3 The danger of overfitting and underfitting

Let us immagine that we are designing a machine learning model. As it has already been discussed, a model is said to be a good if it generalizes any new input data. Overfitting and underfitting are majorly responsible for the poor performances of the machine learning algorithms.

An overfitted model is too complicated for the dataset, meaning that the model becomes tailored to fit the quirks and random noise in the specific sample [Babyak, 2004]. This means that it will not reflect the overall population. Analysing another sample, it would have its own quirks, and the original overfitted model would not likely adapt to the new data [Fig. 1.2].

The opposite problem would happen if one has an underfitted model, which does not produce relevant results.

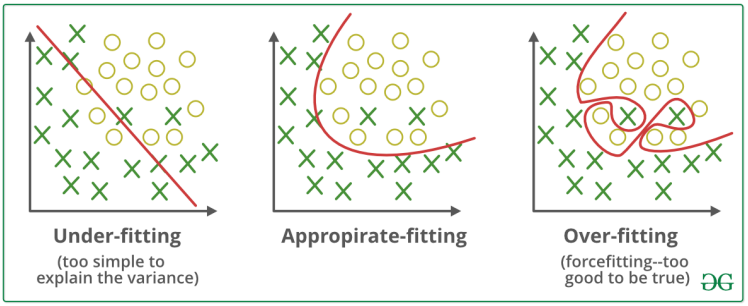


Figure 1.2: The underfitted and overfitted models need to be avoided since they would produce misleading results in classification.

A solution to avoid overfitting is using a linear algorithm if we have linear data. One could also increase training data and reduce the model's complexity.

1.4 Supervised learning

There are three main types of machine learning: supervised, unsupervised and reinforced learning.

Supervised learning is a widely employed type of machine learning and its approach is to learn by example.

Supervised machine learning works by building algorithms that with their mathematical model encapsulate the input data and the desired output data. Such data is known as the *training dataset*, which as one could assume, act as training examples. The training example is represented by a vector, so that the training data is represented by a matrix. The optimization process is iterated and as a result the algorithm will determine the outputs for inputs that were not part of the original set of training data [Fig. 1.3].

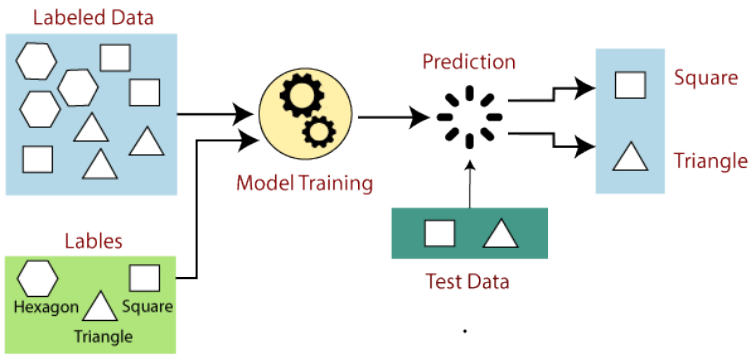


Figure 1.3: The model takes labeled data and uses test data to make a prediction, so that the algorithm will be able to classify and label correctly new incoming data.

The goal is to build a system that learns the mapping between input and output, predicting the output of the system given new inputs. Let us now describe supervised learning more in depth.

If the output takes a finite set of *discrete* values that indicate the labels of the input, the learned mapping leads to the *classification* of the input data. If the output takes *continuous* values, it leads to a regression of the input [e.g. Liu and Wu, 2012]

A learning system needs to go through an estimation process to obtain the model's parameters. The training data for supervised learning needs supervised or labelled information.

Classification algorithms are used when the set of values is limited. An example of such an algorithm is the *support vector machine*.

Supervised machine learning has many advantages, such as the fact that it can be easily used for discriminative pattern classification and for data regression, which are all tasks very useful for humans.

Classification through supervised learning also has some disadvantages, like the fact that not always it is possible to label all data in a supervised way, and even if it were possible, not everything can be described by a discrete label. For example, the margin for separating the two concepts of "hot" and "cold" is not distinct.

To avoid these limitation, one could employ unsupervised learning, reinforced learning or mixed learning approaches.

1.5 Support vector machines

SVMs, or supervised vector machines, are used in supervised machine learning for classification and regression, and outlier detection [Cortes, 1995]. The basis of this classification method is that given a sample of data, made up of two or more different classes, one can find a hyperplane that separates them.

The question is: where should a new data point added to the set lie?

A data point can be represented by a vector, with a number p of dimensions. The function that permits to classify the different types of data is equivalent to the operation of finding a *(p-1)*-dimensional hyperplane that divides them.

Data points falling on either sides of the plane will be attributed to different classes. Since there is not a unequivocal choice for the hyperplane, the optimal one would be where there is the most distance between points of different datasets: the *maximum margin* hyperplane.

Support vectors are data points nearest to the hyperplane margin, and they influence the position and orientation of the hyperplane: by changing the support vectors, one will obtain a different hyperplane. This procedure gives the best chance of new data being correctly classified.

Following such an algorithm, multi-class classification can be performed.

A hyperplane defines a *linear classifier*, meaning that it discriminates data into *labels* based on a linear combination of input features. If there is no clear choice of maximum margin hyperplane in a given number of dimensions and the sets of data are not linearly separable, one can map the data into a higher dimension [Fig. 1.4]. This method, on the other hand, needs to maintain the computational load reasonable. One could achieve that by designing an algorithm that ensures that the dot product between two data point vectors can be easily computed in terms of the variables in the original space: this can be achieved using a *kernel function*.

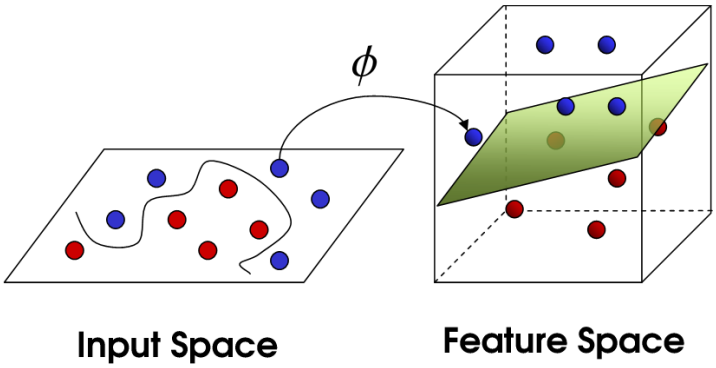


Figure 1.4: When classifying non linearly separable data the optimal hyperplane can be found in a higher dimension.

SVMs have many advantages, as such as being effective in many dimensional spaces, even if the number of dimensions is higher than the number of samples, and being versatile, because different kernel functions can be used. They unfortunately have also flaws: they are not suited for larger datasets, in which case the training time can be really high, and they are less effective if the dataset is noisy or has overlapping classes.

1.6 Fundamental notions of statistical learning

Lets us now develop a better mathematical and theoretical formulation for the support vector machine which produces nonlinear boundaries by constructing a linear boundary in a large, transformed version of the feature space [Hastie et al., 2004].

1.6.1 The case of separable data

It has already been discussed how important it is to obtain an optimal separating hyperplane between separable data. It is also important to understand what to do if the data is not separable by a linear boundary.

The training data consist of N pairs, $(x_1, y_1), (x_2, y_2), ..., (x_N, y_n)$, with $x_i \in \mathrm{R}^p$, and $y_i \in \{-1, 1\}$.

Defining an hyperplane by

$$x : f(x) = x^T\beta + \beta_0 = 0 \tag{1.1}$$

where β is the unit vector. A classification rule introduced by $f(x)$ is the decision function $G(x)$

$$G(x) = sign[x^T\beta + \beta_0] \tag{1.2}$$

$G(x)$ gives the signed distance from a point to the hyperplane. Since the classes are separable, one can find the function $f(x)$ with $y_i f(x) > 0\ \forall i$. Hence we are able to find the biggest margin between training points for classes.

The optimization problem can be written as to capture the concept, keeping in mind that M is the margin,

$$\underset{\beta,\ \beta_0,\ ||\beta||=1}{\text{maximise}}\ M \tag{1.3}$$

subject to $y_i(x_i^T\beta + \beta_0) \geq M,\ i = 1, ..., N$.

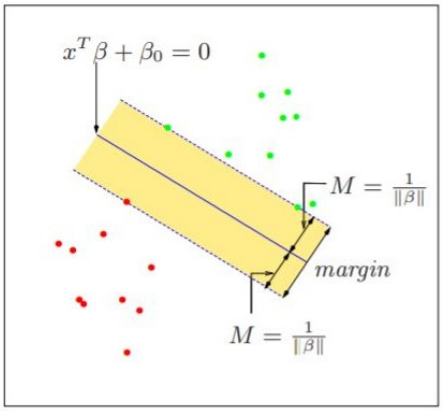


Figure 1.5: Support vector classifier in the separable case.

By dropping the $||\beta|| = 1$ constraint one can write

$$\min_{\beta,\ \beta_0} ||\beta|| \tag{1.4}$$

Subject this time to $y_i(x_i^T\beta + \beta_0) \geq 1, i = 1, ..., N$ with $M = \frac{1}{||\beta||}$ [Fig. 1.5].

This is the usual way of writing the criteria of separation for data.

Such a type of optimization problem is called a *convex* problem, meaning that it deals with the optimization of a complex function and linear inequality constraints.

1.6.2 The case of overlapping data

A new supposition can now be made: the classes will overlap in the feature space. One can still try and solve this problem by maximizing M, but allowing also some points to be on the "wrong" side of the margin.

New variables, called *slack* variables are defined: $\xi = (\xi_1,\ \xi_2,\ ..., \xi_N)$.

One can now modify the formerly analyzed constraints in this way:

$$y_i(x_i^T\beta + \beta_0) \geq M - \xi_i \tag{1.5}$$

or

$$y_i(x_i^T\beta + \beta_0) \geq M(1 - \xi_i) \tag{1.6}$$

With $\forall i,\ \xi_i \geq 0,\ \sum_1^N \xi_i \leq constant$.

Obviously, each choice gives a different solution. The first equation measures the overlap in *actual distance* from the margin. The second equation measures the overlap in *relative distance*, that changes with the width of the margin.

One needs to keep in mind that while the first equation leads to a nonconvex optimization problem, the second one leads to a convex problem and is thus preferable when treating SVMs [Boyd et al., 2004].

Let us now describe the idea behind this formulation. The value ξ_i describes the proportional amount by which the prediction $f(x_i) = x_i^T\beta + \beta_0$ is on the wrong side of its margin. Thus, bounding the sum, we bound the total proportional amount by which predictions fall on the wrong sides of the margin.

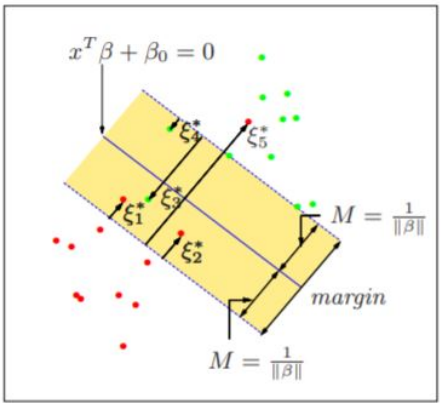


Figure 1.6: Support vector classifier with an overlap

Once again, by dropping the norm constraint, one can simply write $M = \frac{1}{||\beta||}$ [Fig. 1.6], and describe $min||\beta||$ subject to

$$y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i\ \forall i \tag{1.7}$$

$$\xi_i \geq 0,\ \sum \xi_i \leq constant \tag{1.8}$$

Analyzing the latter expression one can note that the points very far into the wrong side of the margin will not make a big difference in the shaping of the boundary.

Such a description is the usual way SVs are defined for the non separable case. It deals with a convex problem, quadratic and with linear inequality constraints.

1.6.3 Solving the problem with Lagrange multipliers

In mathematical optimization, the method of Lagrange multipliers is a strategy for finding the local maxima and minima of a function subject to equality constraints (i.e., subject to the condition that one or more equations have to be satisfied exactly by the chosen values of the variables) [Hoffmann et al., 2012].

Lagrange multipliers can be used to find a solution in classification problems with support vector machines.

One can manipulate the already analyzed expressions and write them in a more convenient way introducing the *cost* parameter, C, defining the *cost function*

$$\min_{\beta,\ \beta_0} \frac{1}{2}||\beta||^2 + C\sum_{1}^{N}\xi_i \tag{1.9}$$

Subject to $\xi_i \geq 0,\ y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i\ \forall i$.

The meaning of the cost function is equivalent to the one of the loss function.

C represents the constant introduced in the non-separable case, and in the separable case C takes the value of infinity: $C = \infty$.

Introducing the Lagrange function,

$$L_p = \frac{1}{2}||\beta||^2 + C\sum_{1}^{N}\xi_i - \sum_{1}^{N}\alpha_i[y_i(x_i^T\beta + \beta_0) - (1 - \xi_i)] - \sum_{i}^{N}\mu_i\xi_i \tag{1.10}$$

Where μ_i is a positive weight relative to ξ_i. The minimization will act onto β, β_0 and ξ_i, meaning that the derivatives will be set to zero, obtaining

$$\beta = \sum_{1}^{N}\alpha_i y_i x_i \tag{1.11}$$

$$\sum_{1}^{N}\alpha_i y_i = 0 \tag{1.12}$$

$$\alpha_i = C - \mu_i,\ \forall i \tag{1.13}$$

Where $\alpha_i,\ \mu_i,\ \xi_i$ are positive $\forall i$. Substituting the expressions above into the Lagrangian function

$$L_D = \sum_{i}^{N}\alpha_i - \frac{1}{2}\sum_{1}^{N}\sum_{1}^{N}\alpha_i\ \alpha_{i'}\ y_i\ y_{i'}\ x_i^T\ x_{i'}^T \tag{1.14}$$

This expression gives the lower bound for any point. Then, it is important to maximise L_D, subject to $0 \leq \alpha_i \leq C$ and $\sum_1^N \alpha_i y_i = 0$.

Other sets of constraints will be given, in addition to the conditions on the derivatives, by the so-called Karush-Kuhn-Tucker conditions:

$$\alpha_i \ [y_i(x_i^T \beta + \beta_0) - (1 - \xi_i)] = 0 \tag{1.15}$$

$$\mu_i \xi_i = 0 \tag{1.16}$$

$$y_i \ (x_i^T \beta + \beta_0) - (1 - \xi_i) \geq 0 \tag{1.17}$$

For $i = 1, ..., N$.

The solution for β will have the form

$$\hat{\beta} = \sum_1^N \hat{\alpha}_i y_i x_i \tag{1.18}$$

$\hat{\alpha}_i$ is different than zero only for those observations i where the constraints given by the Karush-Kuhn-Tucker conditions are met. These are called *support vectors*.

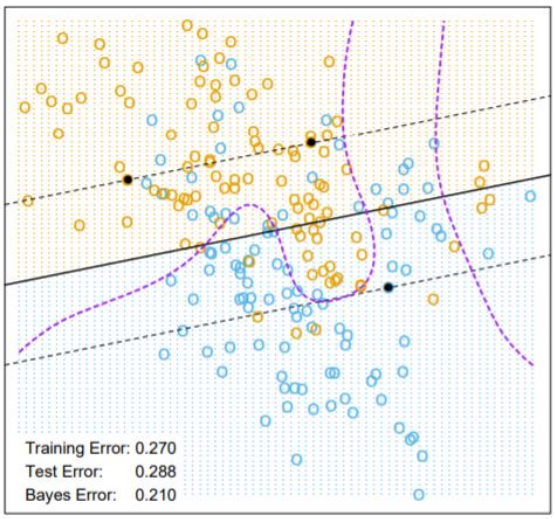
Among these support points, some of them will lie on the margin: $\xi_i = 0$ and $0 \leq \hat{\alpha}_i \leq C$. These values will help solve β_0, using an average of all solutions. The rest of the results will have a $\xi_i > 0$ and $\alpha_i = C$.

Given then, finally, the solutions $\hat{\beta}_0$ and $\hat{\beta}$, one can write the decision function as

$$\hat{G}(x) = sign[\hat{f}(x)] = sign[x^T \hat{\beta} + \hat{\beta}_0] \tag{1.19}$$

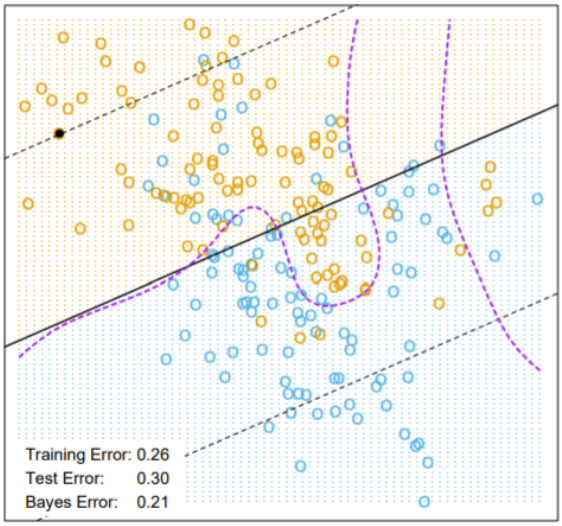
The tuning parameter is the cost parameter, C.

The margin is larger for C=0.01 [Fig.1.8] than it is for C=10000 [Fig.1.7]: larger values of C focus attention more on correctly classified points near the decision boundaries, smaller values of C involve data further away.



$C = 10000$

Figure 1.7: Above illustrated is the case where C=10000. It is apparent that the margin is smaller. Only 65% of the data are support points. The broken purple curve in the background is the Bayes decision boundary.



$C = 0.01$

Figure 1.8: Above illustrated is the case where C=0.01. It is apparent that the margin is bigger. 85% of the data are support points. The broken purple curve in the background is the Bayes decision boundary.

1.7 Kernels

Kernel methods are algorithms that can perform pattern analysis of a dataset. They use a linear classifier to solve a non linear problem, by transforming linearly inseparable data trough a *kernel function*, mapping them into a higher-dimensional space where they become linear.

Let us now give an intuitive explanation on how the kernel method works and why it is so convenient.

$$K(x, y) =< f(\vec{x}), f(\vec{y}) > \tag{1.20}$$

is the Kernel function, and f maps the n-dimensional vectors $\vec{x}$ and $\vec{y}$ into a m-dimensional space given by the scalar product, where $m > n$. To calculate K, $f(\vec{x})$ and $f(\vec{y})$ need to be evaluated first, and then their scalar product. As one can expect, the greater are the dimensions n and m, the more computational effort needs to be exerted, just for the result to again be a number, a scalar.

The kernel *trick* offers a much less "expensive" alternative, depending on the choice of the kernel. For example

$$\vec{x} = (x_{1,2}\,,\ x_3) = (1,\ 2,\ 3)$$

$$\vec{y} = (y_1,\ y_2,\ y_3) = (4,\ 5,\ 6)$$

And $f(\vec{x}) = (x_1x_1,\ x_1x_2,\ x_1x_3,\ x_2x_1,\ x_2x_2,\ x_2x_3,\ x_3x_1,\ x_3x_2,\ x_3x_3)$

$$< f(\vec{x}), f(\vec{y}) > = 1024$$

This step took a lot of algebraic passages, even if $\vec{x}$ and $\vec{y}$ were just 3-dimensional.

Choosing now in a smart way the kernel

$$K = (< \vec{x}, \vec{y} >)^2 = 1024$$

.

The calculation was much simpler.

Now, after describing in a more intuitive manner the concept of kernel and its usefulness, one can try and lay a better mathematical explanation for them, starting from the now familiar support vectors.

The support vector classifier finds linear boundaries in the input feature space. One can think to enlarge such space using for example a polynomial basis expansion, where the basis could be chosen as

$$h_m(x)\ for\ m = 1, ..., M \tag{1.21}$$

Generally, enlarged boundaries achieve better training-class separation.

Once the basis are selected, the procedure is the same as before, and one can fit the SV classifier using

$$h(x_i) = (h_1(x_i),\ h_2(x_i),\ ...,\ h_M(x_i))\ i = 1, ..., N \tag{1.22}$$

Producing the function

$$\hat{f}(x) = h(x)^T \hat{\beta} + \beta_0 \tag{1.23}$$

The classifier is $\hat{G}(x) = sign(\hat{f}(x))$.

The *support vector machine classifier* is the result of one allowing the dimension of the enlarged space to grow and reach large dimensions, even infinite.

The optimization problem, could be described using inner products between basis

$$L_D = \sum_1^N \alpha_i - \frac{1}{2} \sum_i^N \sum_i^N\ \alpha_i\ \alpha_{i'}\ y_i\ y_{i'}\ \langle h(x_i), h(x_{i'}) \rangle \tag{1.24}$$

The solution function can be written as

$$f(x) = h(x)^T \beta + \beta_0 = \sum_1^N \alpha_i\ y_i\ \langle h(x_i), h(x_{i'}) \rangle + \beta_0 \tag{1.25}$$

The mysterious inner product written in the equations above is none other than the kernel function.

$$K(x, x') = \langle h(x), h(x') \rangle \tag{1.26}$$

Some of the choices for such kernel function are the d-th degree polynomial [Fig. 1.9], the radial basis and the neural network. Their mathematical formulations are, in order of mentioning

$$K(x, x') = (1 + \langle x, x' \rangle)^d \tag{1.27}$$

$$K(x, x') = exp(-\gamma ||x - x'||^2) \tag{1.28}$$

$$K(x, x') = tanh(k_1 \langle x, x' \rangle + k_2) \tag{1.29}$$

The role of C is clearer in the enlarged feature space: many times, perfect separation is in these conditions achievable. A large value of C will discourage any positive ξ_i, and lead to an overfit wiggly boundary in the original feature space. A small value of C will encourage a small value of $||\beta||$, which in turn causes $f(x)$ and hence the boundary to be smoother.

Given for example a radial basis, the Hilbert-space associated with the Gaussian RBF kernel has infinite dimension but the kernel may be readily computed for any pair of points (x, y).

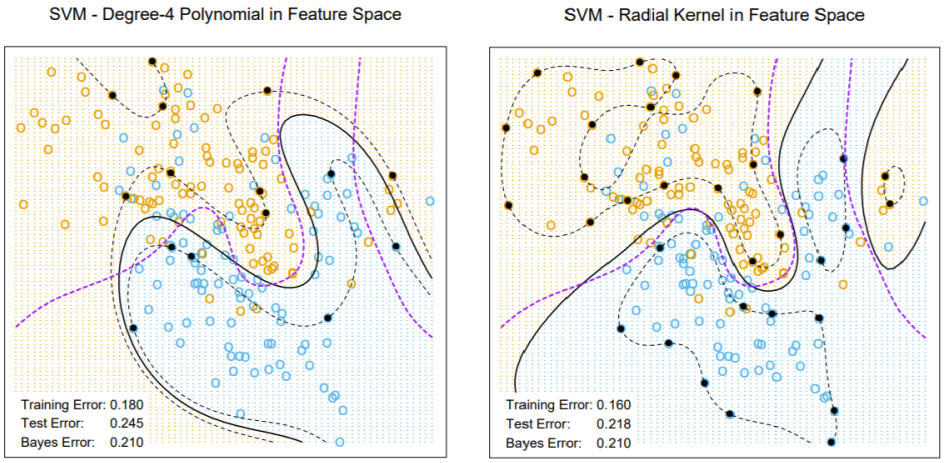


Figure 1.9: The upper plot uses a 4th degree polynomial kernel, the lower a radial basis kernel ($\gamma = 1$). C = 1 worked well in both cases. The radial basis kernel performs the best.

1.7.1 Gram matrix and graph kernels

An important concept in kernel methods is the Gram matrix K, defined with respect to a finite set of data points $x_1, ..., x_m$. The Gram matrix of a kernel K has elements K_{ij}, for $i, j \in \{0, ..., m\}$ equal to the value between pairs of data points given by $K_{ij} = k(x_i, x_j)$. If the Gram matrix of K is positive semidefinite for every possible set of data points, K is a kernel [Schölkopf et al., 1997].

Kernel methods have the desirable property that they do not rely on explicitly characterizing the vector representation $\phi(x)$ of data points: infact, they access data via the Gram matrix K.

Taking a kernel $K : \mathbf{G} \times \mathbf{G} \to \mathbf{R}$, with the help of the Gram matix one can calculate, given objects in a non-empty set of graphs $\mathbf{G}$, the *graph kernel* [Kriege, 2020].

When working with vector data, it is common practice for kernels to compare objects using differences between vector components. The structure of a graph, however, is invariant to permutations of its representation.

Chapter 2

An overview of quantum computing

Quantum computing exploits phenomena of quantum mechanics like state superposition and entanglement. It may be the answer when it comes to achieving the sought-after breakthroughs in science, machine learning, cryptography, risk analysys, etc. [Fig. 2.1], where classical computers' computing power is not enough [Hassija et al., 2020].

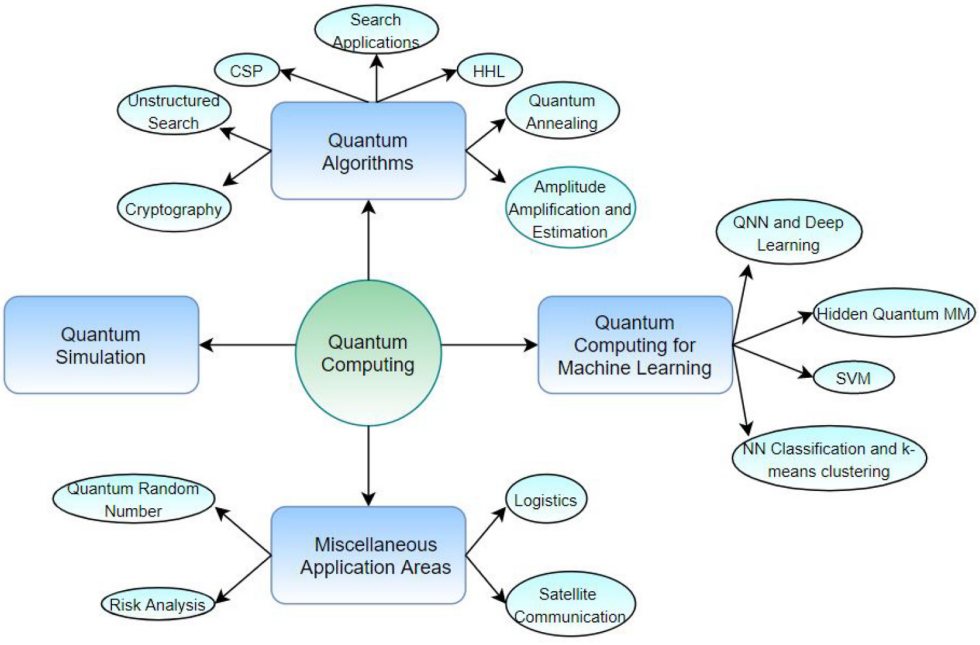


Figure 2.1: The many applications of quantum computing.

2.1 The basis of quantum computing

In contrast form the binary choice between "0" and "1" that classical computers offer, quantum computers are based upon the superposition of the states $|0\rangle$ and $|1\rangle$.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.1}$$

$|0\rangle$ represents the quantum state that will, under the application of logical operators, give 0 as a measurement. The same goes for $|1\rangle$. $|0\rangle$ and $|1\rangle$ are ortonormal and form a basis in the Hilbert space.

One of the differences between quantum and classical computing lies in the fact that while classical bits can only take on the values of 0 and 1, quantum *qubits* can never offer a value with clear certainty (e.g. there cannot be a measurement of 0.5 with $P(0.5) = 1$, and like quantum mechanics teaches, multiple measurement on the same qubits will most likely offer different results.

The propriety of superposition presents the possibility of finding a *computational speed-up* because operations can be executed on many states at the same time.

The superposed state can be written using the Dirac notation as

$$|psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.2}$$

where α and β are complex vectors in the Hilbert space, called amplitudes. Since ψ represents the wave function of the state, the probability, given some value of α and β, can be computed as

$$P(|0\rangle) = |\alpha|^2,\ P(|1\rangle) = |\beta|^2 \tag{2.3}$$

In general, measuring a state $|x\rangle$ in quantum mechanics means to apply

$$p(|x\rangle) = |\,\langle x|\psi\rangle\,|^2 \tag{2.4}$$

It is to be noted that this formula is relative to the probability of a state $|\psi\rangle$ to be measured as $|x\rangle$, whose value is not just limited to be either $|1\rangle$ or $|0\rangle$, and that is because of the superposition property.

In quantum mechanics the procedure of making a measurement can affect the state and make the super-imposed system collapse on a certain value. If another measurement were to be made right after the first one, the state will still be collapsed, and the qubit will practically act like a classical bit.

2.1.1 The Bloch sphere

A Bloch sphere is a geometric representation of a two-level quantum state, with two antipodal points on its surface that correspond to the orthonormal basis vectors $|0\rangle$ and $|1\rangle$: $|0\rangle$

corresponds to "north" while $|1\rangle$ corresponds to "south" [Fig. 2.2]. These can for example represent respectively the spin-states of the electron up and down.

Points right on the surface of the sphere act as pure states, while points inside the sphere serve as mixed states.

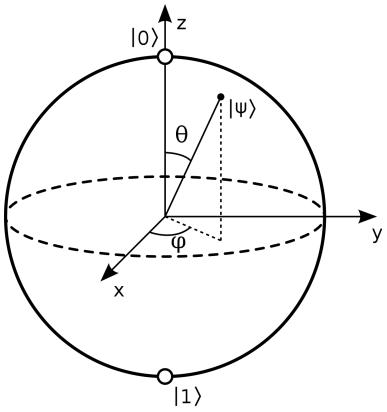


Figure 2.2: The Bloch spere.

A wave function under Born's interpretation is given by a collection of wave packets, each one representing a plane wave. Following electromagnetical laws current is conserved, and in quantum mechanics this leads to probability current conservation, so that $1 = |\alpha|^2 + |\beta|^2$, if the wave function ψ is normalized $\langle\psi|\psi\rangle = 1$. To assure this, quantum gates need to be unitary transformations.

Also, the notation of a general superposed state $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ intends both α and β as complex numbers. The amplitudes can be brought back to real values by adding a relative phase to one of the kets

$$|\psi\rangle = \alpha\,|0\rangle + e^{i\phi}\beta\,|1\rangle \tag{2.5}$$

Where now $\alpha, \beta, \phi \in R$. The amplitudes, with $|\alpha|^2 + |\beta|^2 = 1$ normalization, can be better represented in the Bloch sphere using the variable $\theta \in R$

$$\alpha = cos\frac{\theta}{2},\ \beta = sen\frac{\theta}{2} \tag{2.6}$$

so that the normalization can be still guaranteed. The state representation will now be

$$|\psi\rangle = cos\frac{\theta}{2} + sen\frac{\theta}{2}e^{i\theta}\,|1\rangle \tag{2.7}$$

with $\theta \in [o, \pi]$ and $\theta \in [o, \pi]$, as intended by spherical coordinate systems.

2.1.2 Quantum gates

Position on the Bloch sphere can vary with the application of one or many *quantum gates*. Quantum gates are operations that can change and manipulate qubits between states. They are the building blocks of *quantum circuits* and act on a small number of qubits. One of their properties is that they are always reversible, unlike classical computational gates, implying that quantum gates may be a potential way to improve computational energy efficiency.

To guarantee reversibility, quantum gates need to be unitary operators.

Quantum gates are to be implemented onto quantum circuits, which representation bases itself on the Penrose graphical notation, adopted also by Feynman in 1986 to describe such quantum circuits.

A quantum circuit in its graphic form it is to be read from left to right, as if following a time axis. Horizontal lines represent qubits, doubled lines represent classical bits.

The added elements to this configuration are meant to be interpreted as gates, operations or measurements. An example of quantum gates will be the Pauli matrixes describing the observable corresponding to spin, σ_i, along the i-th axis of $\mathbb{R}$.

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = Pauli - X \tag{2.8}$$

$$\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = Pauli - Y \tag{2.9}$$

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = Pauli - Z \tag{2.10}$$

The Pauli-X gate acts on a single qubit. It represents a rotation of π radians around the x axis on the Bloch sphere [et. al]. It is the quantum equivalent to the NOT gate for classical computers (in respect to the standard basis $|0\rangle$, $|1\rangle$, that distinguishes the Z-direction, meaning that a measure of the eigenvalue 1 corresponds to the classical bit 1 and a measure of -1 corresponds to 0). It maps $|0\rangle$ in $|1\rangle$ and vice-versa. Given this property it is also called *bit-flip* [Fig. 2.3].

The Pauli-Y gate equates to a rotation of π radians around the y axis of the Bloch sphere. It maps $|0\rangle$ in $i\,|1\rangle$ and $|1\rangle$ in $-i\,|0\rangle$ [Fig. 2.3] .

The Pauli-Z gate represents a rotation of π radians around the z axis of the Bloch sphere. It represents a particular case of phase-shift gate with $\phi = \pi$. It leaves $|0\rangle$ unvaried while mapping $|1\rangle$ in $-\,|1\rangle$ and it is also called *phase-flip* [Fig. 2.3].

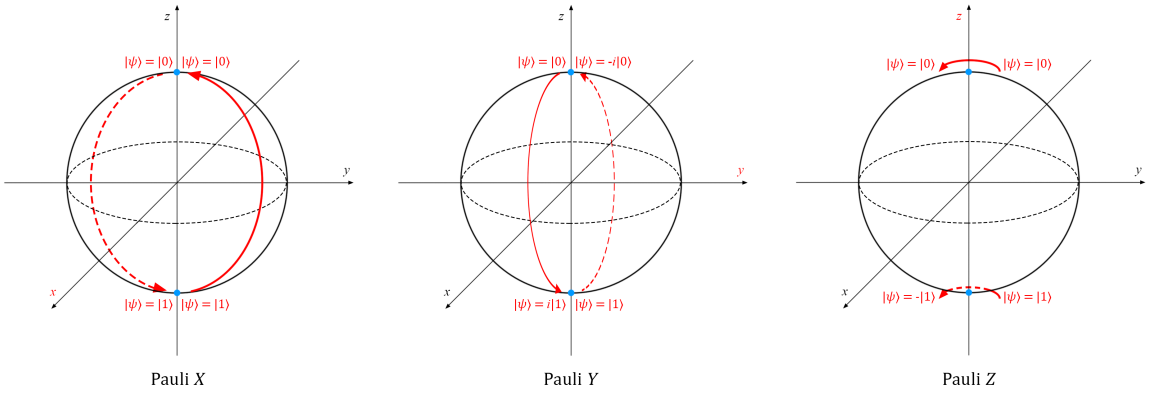


Figure 2.3: Depiction of how each Pauli gate acts on the Bloch sphere

2.1.3 An example of a quantum circuit

A quantum circuit is a series of logical gates that run onto qubits under a certain algorithm. Lets us give an example of a quantum circuit that will help to transform two pure $|0\rangle$ states into one of the four Bell states which are maximally entangled qubits [Fig. 2.4].

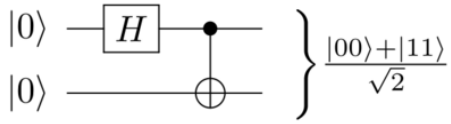


Figure 2.4: The circuit shown above entangles two qubits in state $|0\rangle$ into a Bell state.

Starting from the left, one has the two $|0\rangle$ states. Running from the left to the right, the Hadamard gate takes action in the way represented in Fig. 2.5.

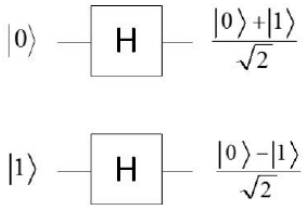


Figure 2.5: How the Hadamard gate modifies the states.

Where

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.11}$$

After the H-gate, the CNOT (controlled not) gate is applied, which flips the target qubit, indicated with the cross enclosed in the circle, to $|1\rangle$, but only if the control qubit, that is represented with the black dot, is also $|1\rangle$. The result is an entangled state.

2.2 Quantum computers

As the name would suggest, quantum computers are machines that exploit quantum mechanic principles to perform different tasks. Their basic unit of memory is called "qubit", quantum bit, made up of physical systems like polarization of a photon, spin of the electron, and many more. The choice between two or more systems can be made depending on how prone each system is to be bothered by noise, how long it stays in a desired state, or how easy it is to operate on each qubit.

Unlike classical computers, which are today used in everyday life while not paying any mind to the separation between hardware and user interface, the best version of quantum-architecture is still in the works.

2.2.1 The quantum stack

Everything between user and hardware is called *stack*. The quantum computer stack [Fig. 2.6], for example should take into account, while converting user inputs into hardware manipulation, the many errors quantum circuits fall victim of. In particular, to try and manage those errors, the *quantum firmware* layer is fundamental. It is a set of protocols that help and bridge the divide between algorithm and hardware, which is unfortunately still imperfect. Its aim is to determine how the physical hardware has to be manipulated if one wants to perform efficient and error-mitigated operations.

In a classical computer the bits can take values of "1", or "0". They are manipulated through changes in the electrical voltage in the circuits, "on" or "off", ad this permits to practically do everything a classical computer needs to do. Obviously, one needs billions of bits to perform tasks, and this is one of the biggest differences between classical and quantum computing, since in the latter only needs two qubits and their superposition.

A number n of classical bits represent n binary digits, while n qubits can represent even as far as 2^n coefficients, even if the output values will again be n.

There exist many types of quantum computers. Two of the most competitive variants are ion traps (used by the company IonQ) and superconducting qubits (used by IBM) [Linke et al., 2017] [Fig. 2.7].

Ion traps are based on electromagnetic fields that can fix in a given position, or "trap", an ion, that is then ready to be manipulated. The qubits are given by two different energy states of the ion, which can be reached through the stimulation of a laser. The ions that are in the higher energy state emit light when measured, unlike the ones in the fundamental state. Ion traps have a gate time of microseconds. Faster gate times mean speediness of the algorithms, and superconducting qubits measure at the nanosecond range.

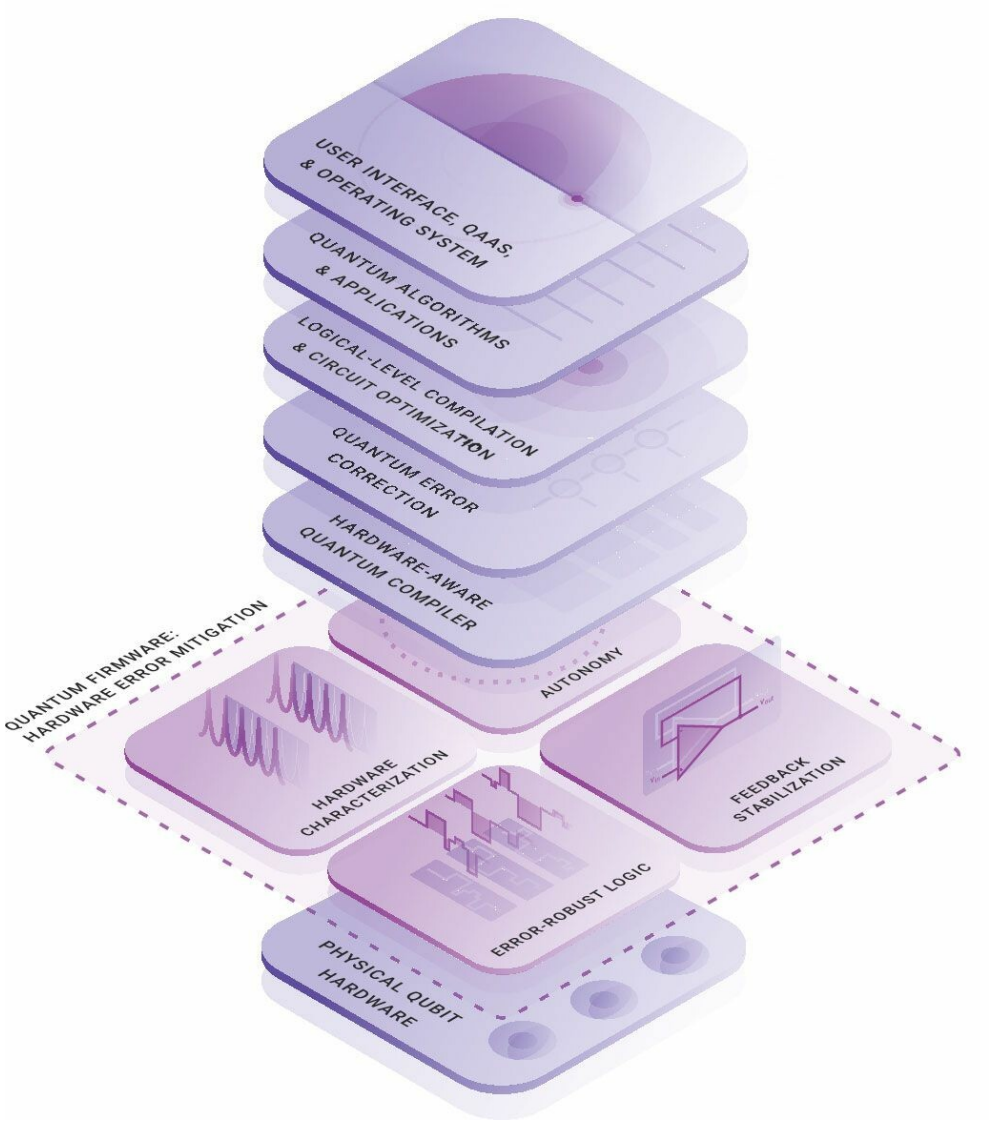


Figure 2.6: The quantum stack representation.Going from bottom to top, the graphic starts from hardware ad reaches user interface, passing from error correction.

Superconducting qubits, that work at a temperature of almost absolute zero, use a resistance-free current running and oscillating in a looped circuit. In this configuration the excitation of the current is given by a microwave signal.

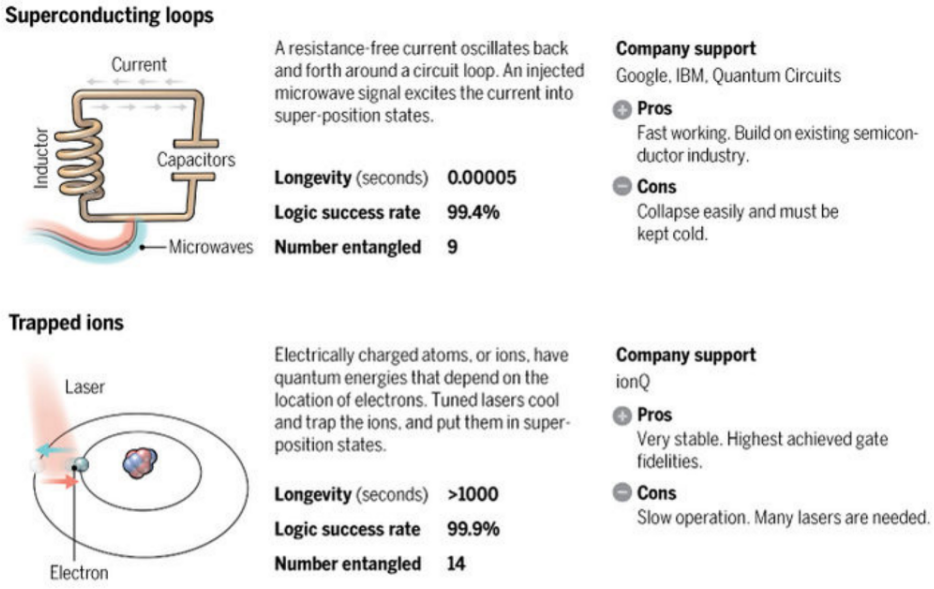


Figure 2.7: The two types of most utilized qubits.

Other than gate time, one needs to evaluate also the coherence time, which is for ion traps up to seconds and for superconducting circuits hundreds of microseconds. This implies that the former could perform more complex or longer algorithms.

It is interference by magnetic fields, radiation or unstable temperature, that cause decoherence of the state (e.g. a $|1\rangle$ gets flipped into $|0\rangle$).

These problems are and need to remain at hardware level, on the lower levels of the quantum stack, meaning the user should be able to work without taking them into account.

It is quantum firmware that focuses on trying to alleviate errors, working on low-level quantum tasks, like mitigating imperfection in resonance frequency of the qubits. This still does not cover the whole of the errors that need intervention, and this is where *quantum error correction* (QEC) comes to play.

QEC "spreads" information onto many qubits, forming a so called *logical qubit*, as to try and hide fallacy on a single problematic qubit. Then, an algorithm identifies errors and corrects them so that the logical qubit does not loose the information it contains: while the single qubits decohere, the logical one does not.

Furthermore, redundancy can help, meaning storing the information multiple times and, in the case that some values are found to disagree, take it and substitute it with the one

given by the majority.

Next on the quantum stack is the logical-level compilation and circuit optimization.

A quantum circuit is a series of quantum gates that run onto qubits under a certain algorithm, keeping in mind that the gates act onto logical qubits for error mitigation.

These gate operations can be united to form whole quantum algorithms, that exploit the underlying quantum nature of the hardware itself. Even if the quantum computer can run, as it has been already explained, up to 2^n coefficients, the measurements can give only as many outputs as the inputs, and that number is n.

This particular limitation makes fully taking advantage of the quantum properties of the system much more difficult than it seems.

A smart choice is to use a *variational* quantum algorithm, that both uses classical and quantum computing, each one where is more convenient.

The top of the quantum stack has now been reached, and user interface needs to be investigated.

Access to quantum computers in day to day life is still only limited to cloud-based infrastructures made available by companies such as IBM with its IBM Quantum Experience, that also offers Qiskit, a Python-based open-source development kit.

2.2.2 NISQ computers

The *noisy intermediate-scale quantum*, NISQ for short, era is the current state of the art in the fabrication of quantum processors. The leading quantum processors contain about 50 to a few hundred *noisy* qubits, which are not advanced enough to reach fault-tolerance nor large enough to profit sustainably from quantum primacy [Bharti et al., 2021].

"Until scalable fault-tolerant universal quantum computers are available in maybe 20 years, quantum computing will be based on noisy quantum processors. Fortunately, machine learning also likes noise. In the next ten years, quantum machine learning is going to be the biggest application of quantum computing. " - Wittek [2018].

Even if quantum primacy is not going to happen anytime soon, quantum computers are starting to be part of the heterogeneous mix of hardware to advance artificial intelligence.

Variational Quantum Algorithms, or VQAs, which use a classical optimizer to train a parametrized quantum circuit, have emerged as a leading strategy to address the constraints of NISQ computers. VQAs have now been proposed for essentially all applications that researchers have envisioned for quantum computers, and they appear to the best hope for obtaining quantum advantage [Cerezo et al., 2021]

2.3 Qiskit

Qiskit is an open source development kit used to work with quantum computer, at the level of circuits and algorithms. It presents as a collection of tools for creating quantum

programs to be run on cloud simulators as such as IBM quantum experience, which founded the Qiskit project.

Qiskit follows the circuit model for universal quantum computing and can be used on any quantum hardware that can be described with such.

The main programming language for Qiskit is Python, and tutorials and didactic Jupyter notebooks with code are freely available on the Qiskit website.

Qiskit is made up of numerous components that all work in unison to enable quantum computing, with the goal of making it available for everybody interested in learning, regardless of their level of expertise.

The user can write code for many aims, and then run it on real quantum computers or simulators. Software can be developed both at the machine code level or at abstract level suitable for users with less experience.

2.3.1 The Qiskit *elements*

The Qiskit Framework is divided into *elements* [Fig. 2.8], each one representing the pillars of quantum computing software, and named after Latin terms for natural elements, nodding to quantum computing's roots in nature and how it works through quantum physics.

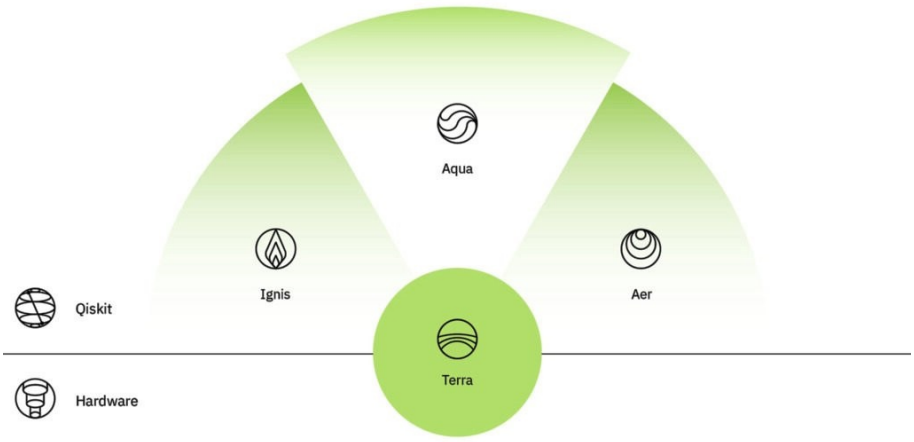


Figure 2.8: The Qiskit framework and its elements

Qiskit *Terra*, is the foundation on which the rest of Qiskit lies. It provides a base for composing quantum programs at the level of circuits and pulses, aiming to optimize them and to manage the execution of experiments. *Terra* defines the interfaces for a desirable user experience.

Qiskit *Aer* permeates all Qiskit elements: it is of fundamental importance since it deals with simulators, emulators and debuggers. *Aer* helps understand the limits of classical

processors when imitating quantum computation, and offers the user a high performance simulator for quantum circuit.

Qiskit *Ignis* fights errors and noise, while improving gates and computing in the presence of such problems.

Qiskit *Aqua* is where algorithms for quantum computing are built. It concerns real world applications: artificial intelligence, chemistry, finance and many more.

Chapter 3

Classification methods with scikit-learn

3.1 What is scikit learn?

Scikit-learn is a free software machine learning library for the Python programming language [Pedregosa et al., 2011]. It features various classification, regression and clustering algorithms including support vector machines.

It is largely written in Python, meaning that it works well with the numerical and scientific libraries NumPy and SciPy.

3.2 The SVC class

In Scikit-Learn, the `SVC` class is capable to perform classification on a dataset. Together with `NuSVC` and `LinearSVC`, `SVC` helps achieve binary and multi-class classification [Fig. 3.1].

`SVC` uses training vectors $x_i \in \mathbf{R}^p$, with $i = 1, ..., n$ divided in two classes and a vector $y \in \{1, -1\}^n$. One wants to find $w \in \mathbf{R}$ such that the prediction given by $sign(w^T \phi(x) + b)$ is correct for most examples. The procedure is the same as the one that has already been described: solving the primal problem, maximizing the margin, adding a penalty C when a sample is misclassified while allowing them to be at a distance from their correct margin, since most of the time data is not completely separable. C controls the strength of this penalty.

α_i are the dual coefficients that are upper-bounded by C. The dual representation underlies the fact that support vectors are mapped into a higher dimensional space thanks to the kernel function.

After solving the optimization problem, `decision_function` represents, in a fitted classifier or outlier detector, a "soft" score for each sample in relation to each class.

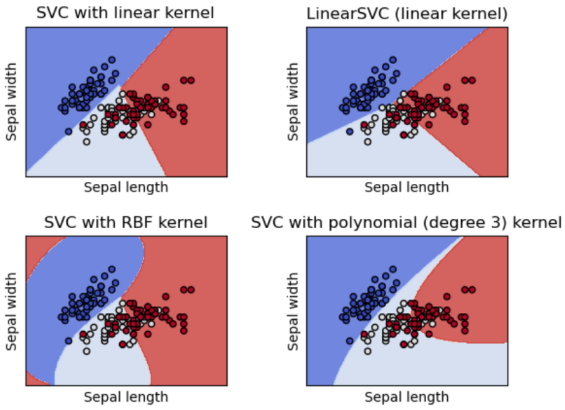


Figure 3.1: Different types of kernel functions and their relative boundary. *Sepal length, sepal width* are features in the `iris_dataset`, which contains four features (length and width of sepals and petals) of 50 samples of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). These measures were used to create a linear discriminant model to classify the species.

The input is the observed data X.

The `decision_function` is $\sum_i y_i \alpha_i K(x_i, x) + b$ and the predicted classes correspond to its sign.

`dual_coef` holds $y_i \alpha_i$, `support_vectors_` holds the support vectors x_i and `intercept_` holds b.

`NuSVC`, or $\nu - SVC$, is a similar method to `SVC` but introduces a new parameter $\nu \in (0, 1]$ that substitutes C, and is is meant to regulate the number of support vectors and the margin errors, which is a sample data that lies on the "wrong" side of the margin, meaning it is either misclassified or correctly classified but does not lie beyond the margin.

The `LinearSVC` cannot be represented through a dual form, so it cannot hold inner products between samples. This means that the kernel trick cannot be applied and the `kernel` parameter is not accepted. Only the linear kernel is supported by `LinearSVC`. In this condition, `LinearSVC` is a faster implementation of the linear kernel.

`SVC`, `NuSVC` and `LinearSVC` take as inputs the arrays `x` of shape `(n_samples, n_features)` that hold the training samples and `y` of shape `n_samples` representing training samples.

The following code will give an example on how to implement SVC with the forementioned x and y arrays, with samples {0, 1}.

```
[ ]: from sklearn import svm
     X = [[0, 0], [1, 1]]
     y = [0, 1]
     clf = svm.SVC()
     clf.fit(X, y)
     SVC()
```

After being fitted the model can be used to predict new values:

```
[ ]: clf.predict([[2., 2.]])
     array([1])
```

Revising the decision function, $\sum_i y_i \alpha_i K(x_i, x) + b$, one can see it depends on the support vectors which act as a subset of the training data. The support vectors have properties that could be found in `support_vectors_`, `support_` and `n_support_`. The first attribute gives the support vectors, the second gives the indices of the support vectors and the third one gives the number of support vectors. They will be used as such

```
[ ]: #support vectors
     clf.support_vectors_
     array([[0., 0.],
            [1., 1.]])

     #indices
     clf.support_
     array([0, 1]...)

     #number
     clf.n_support_
     array([1, 1]...)
```

3.3 Multi-class classification

When there are more than two class instances in input training data, it becomes much more difficult to analyze such data, train the model, and predict results. This is the case where one needs to use multi-class classification [Fig. 3.2].

Multi-class classification allows to categorize the test data into multiple class labels present in trained data as a model prediction, and is mainly composed by the "one-vs-one" and "one-vs-rest" approaches [Bishop, 2006].

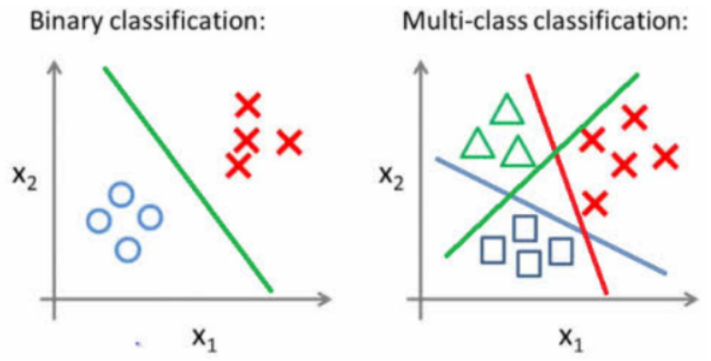


Figure 3.2: This picture clearly shows the difference between binary and multi-class classification. Where in the first instance there is a certain choice for the hyperplane, in the second case there is not an intuitive choice of hyperplane to be made.

3.3.1 One-vs-one

The "one-versus-one" approach [Fig. 3.3] consists of a number of $\frac{n\_classes*(n\_classes-1)}{2}$ classes to be constructed, while each class trains the data from two other classes. Meaning that this approach splits a multi-class classification into one binary classification problem per each pair of classes. The `SVC` and `NuSVC` utilize the approach just described.

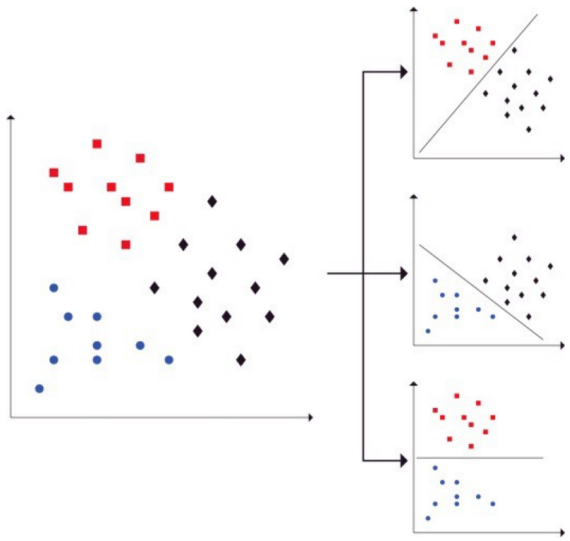


Figure 3.3: "one-vs-one" approach with $\frac{n\_classes*(n\_classes-1)}{2} = 3$. One classifier predicts one class label. The model with the majority counts is concluded as a result.

3.3.2 One-vs-rest

Using the `decision_function_shape` option the "one-vs-one" classifiers are transformed into a "one-vs-rest" decision function which will have the shape of `(n_samples, n_classifiers)`. This other type of approach splits a multi-class classification into one binary classification problem per class [Fig. 3.4].

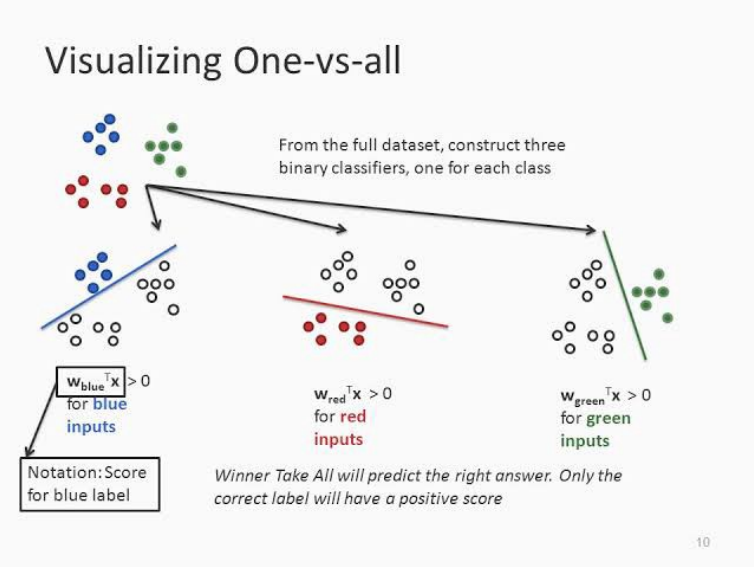


Figure 3.4: The "one-vs-all" approach, also called "all-vs-rest".

3.3.3 Multi-class classification in Python

Let us now take a look at how the multi-class classification is implemented in Python.

In the case of "one-vs-rest", `LinearSVC` has attributes `coef_` which has shape `(n_classes, n_features)`, and `intercept_` which has shape `(n_classes)`.

The "one-vs-one" case, applied with `SVC` and `NuSVC`, is a little bit more complex. The attributes `coef_` and `intercept_` have respectively the shape of (`n_classes*(n_classes-1) / 2, n_features`) and of `n_classes * (n_classes-1) / 2)`.

The shape of the dual coefficient `dual_coef_` is `(n_classes-1, n_SV)`.

```
[ ]: X = [[0], [1], [2], [3]]
     Y = [0, 1, 2, 3]
     clf = svm.SVC(decision_function_shape='ovo')
     clf.fit(X, Y)
     SVC(decision_function_shape='ovo')
     dec = clf.decision_function([[1]])
     dec.shape[1] # 4 classes: 4*3/2 = 6
     6
     clf.decision_function_shape = "ovr"  # one-vs-rest
     dec = clf.decision_function([[1]])
     dec.shape[1] # 4 classes
     4
```

```
[ ]: lin_clf = svm.LinearSVC()
     lin_clf.fit(X, Y)
     LinearSVC()
     dec = lin_clf.decision_function([[1]])
     dec.shape[1]
     4
```

In the last cell of the code, it is to be noted that `LinearSVC` was used with a "one-vs-rest" approach, thus training `n_classes` models.

3.4 Score and probabilities

`SVC` and `NuSVC` methods employ the `decision_function` that gives per-class scores for each sample. If `probability = True`, the methods `predict_proba` and `predict_log_proba` enable the class membership probability estimates.

In the binary case, the probabilities are calibrated through *Platt scaling*, which works by fitting a logistic regression model to a classifier's scores. It gives a way of transforming the outputs of a classification model into a probability distribution over classes. Platt's scaling is not without defects: it is very expensive for large datasets and has some inconsistencies with the probability estimates. For example, in the case of binary classification a sample with an output of `predict_proba` less than 0.5 could be predicted to belong to the positive class, while it could be labeled as belonging to the negative class if the output is more than 0.5.

3.5 Unbalanced problems

In some cases one might need to classify data while needing to give more importance to certain classes or certain values, more *weight* [Fig. 3.5]. In cases like these it is useful to use the `class_weight` and the `sample_weight` parameters.For example, `SVC` implements `class_weight` in the `fit` method.

`sample_weight` can instead be implemented on all `SVC`, `LinearSVC`, `NuSVC` and many more and its effect is to weight the single sample.

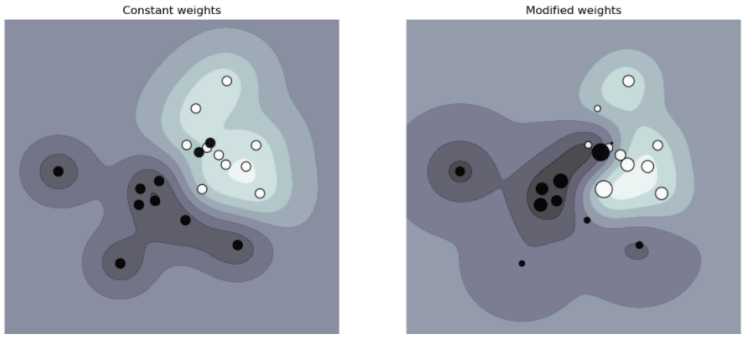


Figure 3.5: The first image describes a "balanced" problem, where the weights are constant throughout the samples. In the second problem each sample has a weight proportional to the dimension of the circle.

3.6 Kernel Functions

As it has already been discussed, there exist many types of kernel functions. Let us revise some of the most relevant examples:

$$Linear\ kernel : \langle x, x' \rangle \tag{3.1}$$

$$Polynomial\ kernel : (\gamma \langle x, x' \rangle + r)^d \tag{3.2}$$

$$Radial\ basis\ kernel : exp(-\gamma ||x - x'||^2) \tag{3.3}$$

All of these can be specified by the `kernel` parameter.

```
[ ]: linear_svc = svm.SVC(kernel='linear')
     linear_svc.kernel
     'linear'
     poly_svc = svm.SVC(kernel='polynomial')
     poly_svc.kernel
     'polynomial'
     rbf_svc = svm.SVC(kernel='rbf')
     rbf_svc.kernel
     'rbf'
```

Figure 3.6: Examples of Python codes for the implementation of different types of kernel.

There is the possibility to define *custom kernels*. This can be done by passing a function to the `kernel` parameter.

The arguments of the kernel will be two matrixes: `(n_samples_1, n_features)` and `(n_samples_2, n_features)`. They will return a kernel matrix of shape `(n_samples_1, n_samples_2)`.

Let us now try to classify the `wine_dataset` with a custom kernel that will be indicated in the code itself [Fig. 3.7].

The wine_dataset describes the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. This will give a multi-class classification with 3 different classes. The analysis determined the quantities of 13 constituents found in each of the three types of wines, meaning 13 features: alcohol, malic acid, ash, alcalinity of ash, magnesium, etc.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

wine = datasets.load_wine()
X = wine.data[:, :2]
Y = wine.target

def my_kernel(X, Y):
    """
    We create a custom kernel:
                (2  0)
    k(X, Y) = X (    ) Y.T
                (0  1)
    """
    M = np.array([[2, 0], [0, 1.0]])
    return np.dot(np.dot(X, M), Y.T)

h = 0.02  # step size in the mesh

clf = svm.SVC(kernel=my_kernel) # we create an instance of SVM and fit out data.
clf.fit(X, Y)

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)  #colour plot

plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Paired, edgecolors="k")   # Plot also the training points
plt.title("3-Class classification using Support Vector Machine with custom kernel")
plt.axis("tight")
plt.show()
```

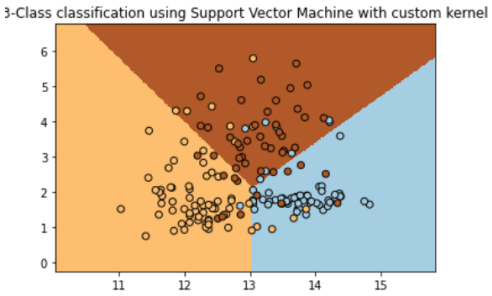


Figure 3.7: Example of Python codes for the classification of the `wine_dataset` with a custom kernel.

Chapter 4

Quantum machine learning

4.1 Why quantum machine learning?

Quantum machine learning is a field of quantum computing where the main aim of research is to unify the techniques and advantages of both quantum computing and machine learning. That means, for example, using AI to try and better error correction for quantum algorithms, while speeding up the computing time it takes to train classical networks.

The reason this new approach came into consideration is the fact that every year the volume of stored data grows by 20% [Schuld et al., 2014], and the need to speed up computational time grows with it.

While it is still a proposal based on strong assumptions and mainly heuristic algorithms, quantum speed-up is one of the attractive feature in quantum computing applied in machine learning.

One example would be Shor's algorithm, that claims to factor integers in polynomial time, instead of superpolynomial time, required by classical computers, meaning basically ad exponential speed-up.

4.2 Quantum computing and classification methods

In classical computing, kernel methods and support vector machines are already a very well known and used source of pattern recognition and data classification. However, there are limitations to their use.

When the feature space becomes exceedingly large, the usage of SVMs and kernels becomes computationally more and more expensive. This is where quantum computing comes to help, because as it has already been illustrated, it gives a prospect of computational speed-up and exploitation of exponentially large quantum state space thought controllable entanglement and interference[Liu, 2021].

Let us propose experimental methods on superconducting qubits, that will take advantage of the large dimensionality of quantum Hilbert spaces to obtain enhanced solutions. These methods have been proposed to be utilizable near-term [Havlíček et al., 2019].

The algorithm that lays the bases of such methods takes on the original problem of supervised learning dealing with the construction of a classifier.

4.2.1 Analyzing the problem

Let us take a training set T and a test set S, where S is a subset of a set $\omega \in R^d$. T and S are mapped as such $m : T \cup S \to \{-1, +1\}$. It is assumed that there is a correlation between the lables given for training and a map $m(\vec{s}) = \tilde{m}(\vec{s})$, where $\vec{s} \in S$ are the test data.

A labeling function uses support vector machines that, as it has already been explained, maps data into a higher dimensional space called feature space, and is then separated with an hyperplane.

A problem one could encounter while trying to utilize SVMs classifiers, kernels or similar methods is that it is not so straightforward to implement classical data onto quantum environment, meaning using the quantum state space as the feature space.

This is done by mapping the data unto a quantum state $\Phi : \vec{x} \in \Omega \, |\Phi(\vec{x})\rangle \langle\phi(\vec{x})|$.

One approach would be to use the quantum computer to estimate the kernel function of the quantum feature space directly and implement a classical SVM. A critical condition is that the kernel cannot be estimated classically.

4.2.2 Analyzing the components

The first component to be analyzed, that will be fundamental in the completion of the task, is the quantum feature map [Fig. 4.1].

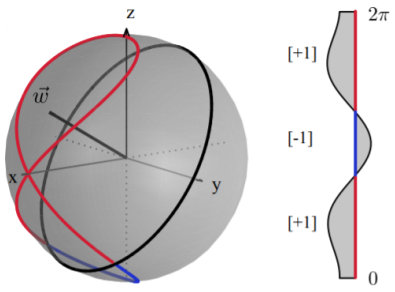


Figure 4.1: Above is the representation of a quantum feature map for a single qubit, where classical data in $\omega = (0, 2\pi]$.

Classifiers based on quantum circuit do not give quantum advantages if the kernel is too simple: one needs to implement a map based on circuits that cannot be implemented classically. An example of employable circuit is the Hadamard gate, which will be employed in the way described by Fig. 4.2.

Defining the feature map on n-qubits generated by the unitary transformation

$$\mathcal{U}_\Phi(\vec{x}) = U_\Phi H^{\otimes n} U_\Phi H^{\otimes n} \tag{4.1}$$

Where H is the Hadamard gate and $\mathcal{U}$ is a diagonal gate in the Pauli-Z basis.

$$U_\Phi = exp(i \sum_S \phi_s \vec{x} \prod_i Z_i) \tag{4.2}$$

Any diagonal unitary U_ϕ can be used if it is possible for it to be implemented efficiently.

Now the data needs to be evaluated. As it has just been described, the data generated are artificial, *ad hoc*. This data can be fully separated by the feature map. Using for example the map for n=d=2, meaning for 2-qubits.

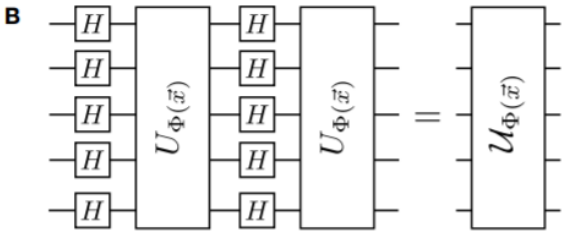


Figure 4.2: An example of how the circuit with implementation of $\mathcal{U}$ and U_ϕ would work.

To summarize, the circuit U_ϕ will be made up of single-qubit or two-qubit units, diagonal in the computational basis. In the case of these experiments, the training T data and the testing S data are both artificially generated so that they are tailored to be classifiable with the former described feature map.

4.2.3 Quantum kernel estimation

Using quantum kernels the SVMs can be implemented directly, and they will be classical SVMs. The quantum computer will be used twice while carrying out this process.

The first step of this procedure, and the first time using the quantum computer, is to estimate the kernel $K(\vec{x_i}, \vec{x_j})$ for all pairs of training data $\vec{x}_i, \vec{x}_j \in T$, $T = \{\vec{x}_i, ..., \vec{x}_t\}$, with $t = |T|$. The label will be $y_i = m(\vec{x}_i)$.

Maximizing the Lagrangian

$$L_d(\alpha) = \sum_1^N \alpha_i - \frac{1}{2}\sum_1^t y_i y_j \alpha_i \alpha_i K(\vec{x}_i, \vec{x}_j) \tag{4.3}$$

subject to $\sum_1^t \alpha_i y_i = 0$ and $\alpha_i \geq 0, \forall i$. This problem is convex when $K(\vec{x}_i, \vec{x}_j)$ is positive definite.

The solution will be noneother than the vector $\alpha = (\alpha_1, ..., \alpha_t)$.

Let us use the quantum computer a second and last time, to estimate the kernel for $\vec{s} \in S$ and build the classifier

$$\tilde{m}(\vec{s}) = sign(\sum_1^N y_i \alpha_i^* K(\vec{x}_i, \vec{s}) + b) \tag{4.4}$$

The more values of α^* are null, the cheaper the evaluation of the classifier will be, since the kernel function only needs to be evaluated in the case of α_i^* is greater than zero.

Now let us analyse how the quantum computer is used to estimate $K(\vec{x}_i, \vec{s})$. The kernel represents the fidelity between different feature vectors, and there exist many different methods to estimate such a fidelity between the quantum states. Since the states in the feature space are not arbitrary, one could evaluate the overlap from the transition amplitude between the two instantaneous eigenstates

$$|\langle \Phi(\vec{x}) | \vec{z} \rangle|^2 = |\langle 0^n | \mathcal{U}_\Phi^\dagger \mathcal{U}_\Phi | 0^n \rangle| \tag{4.5}$$

Where $|0^n\rangle$ is the initial reference state. Then, measuring the final state in the Z-basis a number R of times, the number of all-zero stings 0^n will give the frequency of this transition probability.

There is once again the need to apply the error-mitigation protocol to the first order to estimate the kernel matrix. Using this protocol, one obtains a number of support vectors α_i that are very similar to the noise-free case.

Chapter 5

Classification with a Qiskit quantum algorithm

5.1 Quantum kernel classification

A classification problem will be analysed using the Qiskit softwares `qiskit-terra`, `qiskit-aer` and `qiskit-machine-learning`.

The dataset is the `ad_hoc_dataset` (source in the Code Appendix). This is a "special" dataset that, as the name would suggest, was made *ad hoc* to reach the specific aim of the experiment, without reflecting any real-world features. Because of this, one cannot enunciate the classes belonging to it, because it does not contain any explicitly characterised classes. It was just meant to act as an exercise to understand the extent of quantum computing's potential. One could imagine that the two classes will be "red" and "blue".

The markdown cells that happen to be in between some of the code blocks will explain in detail the functions and operations in the code.

```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.svm import SVC
from sklearn.cluster import SpectralClustering
from sklearn.metrics import normalized_mutual_info_score

from qiskit import BasicAer
from qiskit.circuit.library import ZZFeatureMap
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit_machine_learning.algorithms import QSVC
from qiskit_machine_learning.kernels import QuantumKernel
from qiskit_machine_learning.datasets import ad_hoc_data

seed = 12345
algorithm_globals.random_seed = seed
```

`ad_hoc_dimension` declares that the dimension of the dataset is 2, N=2.

`Train_features` and `train_labels` represent the variables to be passed onto the training algorithm.

The aim for "train" is to find the best fitting parameters of a certain $f(x)$ so that this $f(x)$ describes the training data in the most accurate possible way.

`Test_features`, `test_labels` are used to check the goodness of the fitting of $f(x)$ onto new points, different than the test ones.

`ad_hoc_total` is a martix that gives us an example of already classified data relative to their margin. One needs to check how the data described above will fit into this `ad_hoc_data`.

_

```
adhoc_dimension = 2
train_features, train_labels, test_features, test_labels, adhoc_total = ad_hoc_data(
    training_size=20,  #number of points in the trainig set
    test_size=5,       #number of points in the test set
    n=adhoc_dimension,
    gap=0.3,
    plot_data=False, one_hot=False, include_sample_total=True
)
```

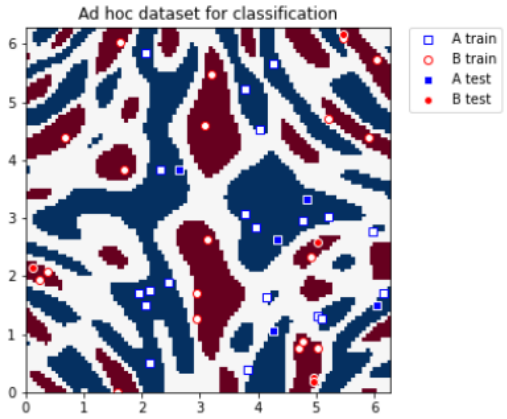
Let us now plot the function. `ad_hoc_data` will be translated into a `numpy` matrix. The scatter plots are 4 because the algorithm wants firstly to plot the point and then the colour, because for both the test data and the training set there will be "red" and "blue". The training points, represented by the emptied out indicators, were the ones used to calculate the margin regions. Those regions were, in this "didactic" example, already implemented in `ad_hoc_data`.

```
plt.figure(figsize=(5, 5))
plt.ylim(0, 2 * np.pi)
plt.xlim(0, 2 * np.pi)
plt.imshow(np.asmatrix(adhoc_total).T, interpolation='nearest',
           origin='lower', cmap='RdBu', extent=[0, 2 * np.pi, 0, 2 * np.pi])

plt.scatter(train_features[np.where(train_labels[:] == 0), 0], train_features[np.where(train_labels[:] == 0), 1],
            marker='s', facecolors='w', edgecolors='b', label="A train")
plt.scatter(train_features[np.where(train_labels[:] == 1), 0], train_features[np.where(train_labels[:] == 1), 1],
            marker='o', facecolors='w', edgecolors='r', label="B train")
plt.scatter(test_features[np.where(test_labels[:] == 0), 0], test_features[np.where(test_labels[:] == 0), 1],
            marker='s', facecolors='b', edgecolors='w', label="A test")
plt.scatter(test_features[np.where(test_labels[:] == 1), 0], test_features[np.where(test_labels[:] == 1), 1],
            marker='o', facecolors='r', edgecolors='w', label="B test")

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.title("Ad hoc dataset for classification")

plt.show()
```

The plot shows the decision boundaries by which the model classifies the data. In two dimensions, it acts as the hyperplane with the relative maximised margin.

In the plot one can see a good separation between red and blue entries.

The test points, represented by the filled in indicators, fall into the coloured parts nicely, indicating a good fitting, but not an overfitting that would imply the algorithm to be not general if one were to try and see the classification outcome on a new and different data point.



Knowing that a feature map is an algorithm that tries to learn and predict which category the data point belongs to, let us introduce the `ZZfeaturemap` [Fig. 5.1]. It is a quantum feature map which converts classical data into quantum data. It represents a second-order Pauli-Z evolution circuit, where the mapping function ϕ is a non-linear function with the shape of

$$\phi(x, y) = (\pi - x)(\pi - y) \tag{5.1}$$

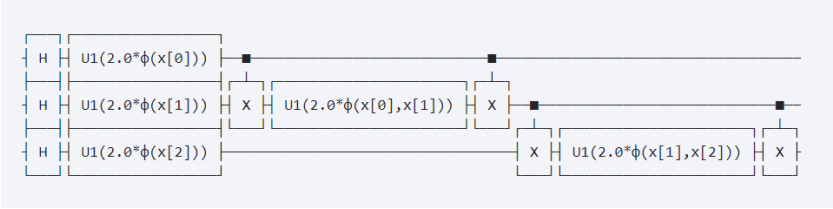


Figure 5.1: ZZfeaturemap for 3 qubits and 1 repetition.

`ad_hoc_feature_map` utilizes the function `ZZFeatureMap`. Then `ad_hoc_backend` initializes the backend to run the experiment. Finally, `ad_hoc_kernel` uses the forementioned `feature_map` and `quantum_instance` to employ a quantum kernel.

```
adhoc_feature_map = ZZFeatureMap(feature_dimension=adhoc_dimension,
                                 reps=2, entanglement='linear')

adhoc_backend = QuantumInstance(BasicAer.get_backend('qasm_simulator'), shots=1024,
                                seed_simulator=seed, seed_transpiler=seed)

adhoc_kernel = QuantumKernel(feature_map=adhoc_feature_map, quantum_instance=adhoc_backend)
```

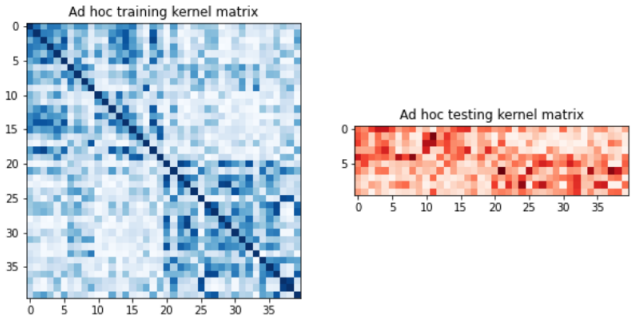
The scikit-learn svc algorithm allows us to define a custom kernel by providing the kernel as a callable function. We can do it using the QuantumKernel class in qiskit. The resukt will be the classificaton test score, which indicates how well the test data fall into the classification regions. The closer this value is to 1, the better the result is.

```
adhoc_svc = SVC(kernel=adhoc_kernel.evaluate)
adhoc_svc.fit(train_features, train_labels)        #fit means train
adhoc_score = adhoc_svc.score(test_features, test_labels)

print(f'Callable kernel classification test score: {adhoc_score}')
```

```
Callable kernel classification test score: 1.0
```

In the above section we trained the `ad_hoc_SVC` with `train_data` and `train_labels`. Now we want to see the result, so we plot the resulting kernel matrixes. They represent the matrix given by the scalar product between each couple of vector data representing the features in the case of a custom kernel.

```
adhoc_matrix_train = adhoc_kernel.evaluate(x_vec=train_features)
adhoc_matrix_test = adhoc_kernel.evaluate(x_vec=test_features,
                                          y_vec=train_features)

fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].imshow(np.asmatrix(adhoc_matrix_train),
              interpolation='nearest', origin='upper', cmap='Blues')
axs[0].set_title("Ad hoc training kernel matrix")
axs[1].imshow(np.asmatrix(adhoc_matrix_test),
              interpolation='nearest', origin='upper', cmap='Reds')
axs[1].set_title("Ad hoc testing kernel matrix")
plt.show()

adhoc_svc = SVC(kernel='precomputed')
adhoc_svc.fit(adhoc_matrix_train, train_labels)
adhoc_score = adhoc_svc.score(adhoc_matrix_test, test_labels)

print(f'Precomputed kernel classification test score: {adhoc_score}')
```



Lets us now analyze a qiskit extension to the SVC from scikit-learn. The procedure will be the same as the formerly implemented code.

```
qsvc = QSVC(quantum_kernel=adhoc_kernel)
qsvc.fit(train_features, train_labels)
qsvc_score = qsvc.score(test_features, test_labels)

print(f'QSVC classification test score: {qsvc_score}')
```

```
QSVC classification test score: 1.0
```

The classification test score is 1, which is the maximum value possible. This is consistent with the aim of `ad_hoc_data`.

5.2 A comparison between classical and quantum classification

This new experiment regards the analysis of a real-world case, using `breast_cancer` dataset, which compares cases of malignant and benignant tumor.

Unlike the last experiment, a similar but less didactic approach will be taken in the classification. The code in this instance is in fact much slimmer and can lend itself to the addition of a direct comparison between classic and quantum cases.

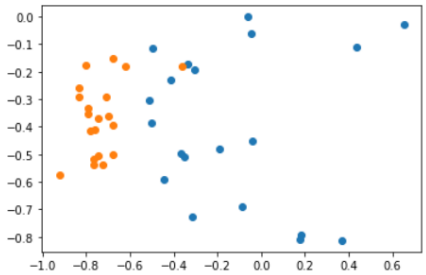
One expects the quantum case to have a faster classification time with better results than the classic case.

Furthermore, the kernel matrixes are different in the two cases.

Let us begin and analyze the `breast_cancer` dataset. It is a multivariate dataset with real characteristics whose features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. The classes will be $M = malignant$ and $B = benignant$. The features are 10 in total: area, perimeter, radius, texture, smoothness, compactness, concavity, concave points, symmetry and fractal dimension. Just two of them will be used, at random, to be plotted in a 2D graph.

```
import matplotlib.pyplot as plt
import numpy as np

from qiskit import BasicAer
from qiskit.circuit.library import ZZFeatureMap
from qiskit.aqua import QuantumInstance, aqua_globals
from qiskit.aqua.algorithms import QSVM
from qiskit.aqua.utils import split_dataset_to_data_and_labels, map_label_to_class_name

import time

seed = 10599
aqua_globals.random_seed = seed
```

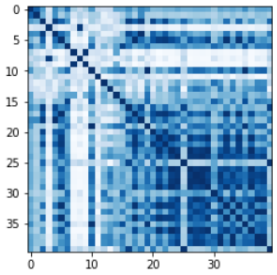
```
from qiskit.ml.datasets import breast_cancer

feature_dim = 2
sample_total, training_input, test_input, class_labels = breast_cancer(
    training_size=20,
    test_size=10,
    n=feature_dim,
    plot_data=True
)
```



```
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='linear')
qsvm = QSVM(feature_map, training_input, test_input)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)

t0 = time.time()
result = qsvm.run(quantum_instance)
t1 = time.time()
print(f'Train + test time: {(t1-t0):.2f} s')

print(f'Testing success ratio: {result["testing_accuracy"]}')
```

```
Train + test time: 22.21 s
Testing success ratio: 0.9
```

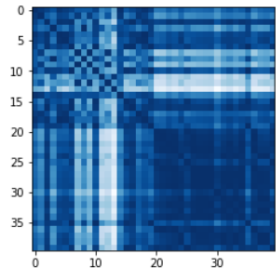
```
kernel_matrix = result['kernel_matrix_training']
img = plt.imshow(np.asmatrix(kernel_matrix),interpolation='nearest',origin='upper',cmap='Blues')
```



Let us analyse the classical case for a comparison.

```
from qiskit.aqua.algorithms import SklearnSVM

t0 = time.time()
result = SklearnSVM(training_input, test_input).run()
t1 = time.time()

print(f'Train + test time: {(t1-t0):.2f} s')

print(f'Testing success ratio: {result["testing_accuracy"]}')

kernel_matrix = result['kernel_matrix_training']
plt.imshow(np.asmatrix(kernel_matrix), interpolation='nearest', origin='upper', cmap='Blues');
```

```
Train + test time: 0.03 s
Testing success ratio: 0.85
```

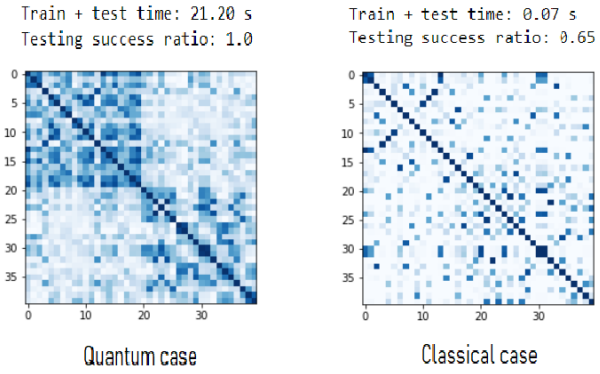


The testing success ratio is better in the quantum case, demonstrating the aim of the experiment.

The kernel matrixes appear to be different, as one would expect from the different classification scores. The quantum kernel matrix is relative to a custom kernel while the classical kernel matrix employes a linear kernel. The coloured entries are given by the scalar product of the feature vectors.

An unexpected result is the classification time, which is much larger in the quantum computing case. This seems to go against the computational speed-up, but the slow time can be easily justified by saying that these computations were made onto quantum *simulators*, which cannot give the same advantage of real quantum computers.

Avoiding explicitly showing the code for the ad_hoc_dataset, since it will be the same, one could try and evaluate again the classification times and kernel matrixes in both the quantum and classical cases.



Again, while the classification score is much better in the quantum case, the classification time is evidently shorter in the classical case. The justification for this time discrepancy is the same that was given above.

Chapter 6

Conclusion

The experiments that were overviewed in this thesis all have the underlying aim to demonstrate the advantage of quantum computing.

As far as the classification score goes, the aim has been successfully reached, since in every experiment the classification score was either 1, the maximum value possible (section 5.1), or better than the classical case (section 5.2).

Nonetheless, a few remarks need to be made.

The dataset `ad_hoc_data` in the experiment was carefully picked to achieve better performance in the quantum case. Indeed, this dataset does not have any real-world features but acts as an exercise to be able to understand the extent of quantum computing's performance.

The `breast_cancer` dataset, defining malignant and benignant tumors, was instead used to give a real-world comparison between a classic classification score and a quantum classification score.

As expected, the classification for the quantum cases had a better score.

The kernel matrixes were plotted, giving two different plots for the classical and quantum cases. This is to be expected given the different kernels that were utilized (custom kernel for the quantum case and linear kernel for the classical case).

The computing time was also calculated, giving results that do not line up with the computational speed-up proposed in quantum computing: the time in the classical case was smaller. To justify this, one needs to keep in mind that the analysed experiments concerned classification tasks and *not* quantum primacy, and were subsequently ran utilizing quantum simulators, not real quantum computers.

This means that the classification time reported in the experiments may not fully serve as a representation of a "real" quantum case.

Keeping in mind such constraints, the results reflected the expectations and the didactic goal was reached successfully.

An option to further better the classification score (where possible) and time would be to

run the experiments on the quantum computers offered by IBM quantum lab, and to follow algorithms, like the ones proposed by Liu [2021], that are proven to achieve a rigorous and robust quantum speed-up in supervised machine learning.

Until the noisy intermediate-scale (NISQ) era of quantum computers is still the state of the art, quantum computing primacy over classical computing will remain an open debate.

Nonetheless, the ferment that this new prospective has brought inside (and outside) of the scientific community has been hardly matched by recent discoveries. Not only that, it poses an interesting challenge on human's technology against the rules and properties of nature, because as Feynman said, "Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy".

Code appendix

Listing 6.1: Ad_hoc_dataset

```
import numpy as np
import scipy
from qiskit.aqua import aqua_globals, MissingOptionalLibraryError


[docs]def ad_hoc_data(training_size, test_size, n, gap, plot_data=False):
    """ returns ad hoc dataset """
    class_labels=[r'A', r'B']
    count=0
    if n==2:
        count=100
    elif n==3:
        count=20  # coarseness of data separation

    label_train=np.zeros(2*(training_size+test_size))
    sample_train=[]
    sample_a=[[0 for x in range(n)] for y in range(training_size+test_size)]
    sample_b=[[0 for x in range(n)] for y in range(training_size+test_size)]

    sample_total=[[[0 for x in range(count)] for y in range(count)]
    for z in range(count)]

    # interactions=np.transpose(np.array([[1, 0],[0, 1],[1, 1]]))

    steps=2*np.pi/count

    # sx=np.array([[0, 1], [1, 0]])
    # X=np.asmatrix(sx)
    # sy=np.array([[0, -1j], [1j, 0]])
    # Y=np.asmatrix(sy)
    s_z=np.array([[1, 0], [0, -1]])
    z_m=np.asmatrix(s_z)
    j_m=np.array([[1, 0], [0, 1]])
    j_m=np.asmatrix(j_m)
    h_m =np.array([[1, 1], [1, -1]]) / np.sqrt(2)
    h_2=np.kron(h_m, h_m)
    h_3=np.kron(h_m, h_2)
    h_m=np.asmatrix(h_m)
    h_2=np.asmatrix(h_2)
    h_3=np.asmatrix(h_3)

    f_a=np.arange(2**n)

    my_array=[[0 for x in range(n)] for y in range(2**n)]

    for arindex,_in enumerate(my_array):
```

```
    temp_f=bin(f_a[arindex])[2:].zfill(n)
    for findex in range(n):
        my_array[arindex][findex]=int(temp_f[findex])

my_array=np.asarray(my_array)
my_array=np.transpose(my_array)

# Define decision functions
maj=(-1) ** (2 * my_array.sum(axis=0) > n)
parity=(-1) ** (my_array.sum(axis=0))
# dict1=(-1) ** (my_array[0])
d_m=None
if n==2:
    d_m=np.diag(parity)
elif n==3:
    d_m=np.diag(maj)

basis=aqua_globals.random.random((2**n, 2**n))+\
    1j * aqua_globals.random.random((2**n, 2**n))
basis=np.asmatrix(basis).getH()*np.asmatrix(basis)

[s_a, u_a]=np.linalg.eig(basis)

idx=s_a.argsort()[::-1]
s_a=s_a[idx]
u_a=u_a[:, idx]

m_m=(np.asmatrix(u_a)).getH()*np.asmatrix(d_m)*np.asmatrix(u_a)

psi_plus=np.transpose(np.ones(2))/np.sqrt(2)
psi_0=1
for k in range(n):
    psi_0=np.kron(np.asmatrix(psi_0), np.asmatrix(psi_plus))

sample_total_a=[]
sample_total_b=[]
sample_total_void=[]
if n==2:
    for n_1 in range(count):
        for n_2 in range(count):
            x_1=steps*n_1
            x_2=steps*n_2
            phi=x_1*np.kron(z_m, j_m)+x_2*np.kron(j_m, z_m)+\
                (np.pi-x_1)*(np.pi-x_2)*np.kron(z_m, z_m)
            u_u=scipy.linalg.expm(1j*phi)  # pylint: disable=no-member
            psi=np.asmatrix(u_u)*h_2*np.asmatrix(u_u) * np.transpose(psi_0)
            temp=np.real(psi.getH()*m_m*psi).item()
            if temp>gap:
                sample_total[n_1][n_2]=+1
            elif temp<-gap:
```

```
            sample_total[n_1][n_2]=-1
        else:
            sample_total[n_1][n_2]=0

# Now sample randomly from sample_Total a number of times training_size
+testing_size t_r=0
while t_r<(training_size+test_size):
    draw1=aqua_globals.random.choice(count)
    draw2=aqua_globals.random.choice(count)
    if sample_total[draw1][draw2]==+1:
        sample_a[t_r]=[2*np.pi*draw1/count, 2*np.pi*draw2/count]
        t_r+=1

t_r=0
while t_r<(training_size+test_size):
    draw1=aqua_globals.random.choice(count)
    draw2=aqua_globals.random.choice(count)
    if sample_total[draw1][draw2]==-1:
        sample_b[t_r]=[2*np.pi*draw1/count, 2*np.pi*draw2/count]
        t_r+=1

sample_train=[sample_a, sample_b]

for lindex in range(training_size+test_size):
    label_train[lindex]=0
for lindex in range(training_size+test_size):
    label_train[training_size+test_size+lindex]=1
label_train=label_train.astype(int)
sample_train=np.reshape(sample_train, (2*(training_size+test_size), n))
training_input={key:(sample_train[label_train==k, :])[:training_size]
                for k, key in enumerate(class_labels)}
test_input={key: (sample_train[label_train==k, :])[training_size:(
    training_size+test_size)] for k, key in enumerate(class_labels)}

if plot_data:
    try:
        import matplotlib.pyplot as plt
    except ImportError as ex:
        raise MissingOptionalLibraryError(
            libname='Matplotlib',
            name='ad_hoc_data',
            pip_install='pip install matplotlib') from ex

    plt.show()
    fig2=plt.figure()
    for k in range(0, 2):
        plt.scatter(sample_train[label_train==k, 0][:training_size],
                    sample_train[label_train==k, 1][:training_size])

    plt.title("Ad-hoc Data")
```

```
            plt.show()

    elif n==3:
        for n_1 in range(count):
            for n_2 in range(count):
                for n_3 in range(count):
                    x_1=steps*n_1
                    x_2=steps*n_2
                    x_3=steps*n_3
                    phi=x_1*np.kron(np.kron(z_m, j_m), j_m)+\
                        x_2*np.kron(np.kron(j_m, z_m), j_m)+\
                        x_3*np.kron(np.kron(j_m, j_m), z_m)+\
                        (np.pi-x_1)*(np.pi-x_2)*np.kron(np.kron(z_m, z_m), j_m)+\
                        (np.pi-x_2)*(np.pi-x_3)*np.kron(np.kron(j_m, z_m), z_m)+\
                        (np.pi-x_1)*(np.pi-x_3)*np.kron(np.kron(z_m, j_m), z_m)
                    u_u=scipy.linalg.expm(1j*phi)  #pylint: disable=no-member
                    psi=np.asmatrix(u_u)*h_3*np.asmatrix(u_u)* np.transpose(psi_0)
                    temp=np.real(psi.getH()*m_m*psi).item()
                    if temp>gap:
                        sample_total[n_1][n_2][n_3]=+1
                        sample_total_a.append([n_1, n_2, n_3])
                    elif temp <-gap:
                        sample_total[n_1][n_2][n_3]=-1
                        sample_total_b.append([n_1, n_2, n_3])
                    else:
                        sample_total[n_1][n_2][n_3]=0
```

Listing 6.2: breast_cancer

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.decomposition import PCA
from qiskit.aqua import MissingOptionalLibraryError


[docs]def breast_cancer(training_size, test_size, n, plot_data=False):
    """returns breast cancer dataset"""
    class_labels=[r'A', r'B']
    data, target=datasets.load_breast_cancer(return_X_y=True)
    sample_train, sample_test, label_train, label_test=\
        train_test_split(data, target, test_size=0.3, random_state=12)

    # Now we standardize for gaussian around 0 with unit variance
    std_scale=StandardScaler().fit(sample_train)
    sample_train=std_scale.transform(sample_train)
    sample_test=std_scale.transform(sample_test)

    # Now reduce number of features to number of qubits
```

```
    pca=PCA(n_components=n).fit(sample_train)
    sample_train=pca.transform(sample_train)
    sample_test=pca.transform(sample_test)

    # Scale to the range (-1,+1)
    samples=np.append(sample_train, sample_test, axis=0)
    minmax_scale=MinMaxScaler((-1, 1)).fit(samples)
    sample_train=minmax_scale.transform(sample_train)
    sample_test=minmax_scale.transform(sample_test)

    # Pick training size number of samples from each distro
    training_input={key:(sample_train[label_train==k, :])[:training_size]
                    for k, key in enumerate(class_labels)}
    test_input={key:(sample_test[label_test==k, :])[:test_size]
                for k, key in enumerate(class_labels)}

    if plot_data:
        try:
            import matplotlib.pyplot as plt
        except ImportError as ex:
            raise MissingOptionalLibraryError(
                libname='Matplotlib',
                name='breast_cancer',
                pip_install='pip install matplotlib') from ex
        for k in range(0, 2):
            plt.scatter(sample_train[label_train==k, 0][:training_size],
                        sample_train[label_train==k, 1][:training_size])

        plt.title("PCA dim. reduced Breast cancer dataset")
        plt.show()

    return sample_train, training_input, test_input, class_labels
```

Listing 6.3: ZZfeaturemap

```
from typing import Callable, List, Union, Optional
import numpy as np
from .pauli_feature_map import PauliFeatureMap


[docs]class ZZFeatureMap(PauliFeatureMap):

    def __init__(
        self,
        feature_dimension: int,
        reps: int=2,
        entanglement: Union[str, List[List[int]], Callable[[int],
        List[int]]]="full",
        data_map_func: Optional[Callable[[np.ndarray], float]]=None,
```

```
        insert_barriers: bool=False,
        name: str="ZZFeatureMap",
    ) -> None:
        if feature_dimension<2:
            raise ValueError(
                "The ZZFeatureMap contains 2-local interactions and cannot be"
                f"defined for less than 2 qubits. You provided {feature_dimension}."
            )

        super().__init__(
            feature_dimension=feature_dimension,
            reps=reps,
            entanglement=entanglement,
            paulis=["Z", "ZZ"],
            data_map_func=data_map_func,
            insert_barriers=insert_barriers,
            name=name,
        )
```

Bibliography

Parth Bhavsar, Ilya Safro, Nidhal Bouaynaya, Robi Polikar, and Dimah Dera. Chapter 12 - machine learning in transportation data analytics. In Mashrur Chowdhury, Amy Apon, and Kakan Dey, editors, *Data Analytics for Intelligent Transportation Systems*, pages 283–307. Elsevier, 2017. ISBN 978-0-12-809715-1. doi: https://doi.org/10.1016/B978-0-12-809715-1.00012-2. URL `https://www.sciencedirect.com/science/article/pii/B9780128097151000122`.

A. M. Turing. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX (236):433–460, 10 1950. ISSN 0026-4423. doi: 10.1093/mind/LIX.236.433. URL `https://doi.org/10.1093/mind/LIX.236.433`.

Gavin Hackeling. *Mastering Machine Learning With Scikit-Learn*. Packt Publishing, 2014. ISBN 1783988363.

Francois Chollet. *Deep learning with Python*. 2021.

Michael A Babyak. What you see may not be what you get: a brief, nontechnical introduction to overfitting in regression-type models. *Psychosomatic medicine*, 66(3):411–421, 2004.

Qiong Liu and Ying Wu. Supervised learning. 01 2012. doi: 10.1007/978-1-4419-1428-$6_4$51.

Vapnik Vladimir Cortes, Corinna. Support-vector networks. 09 1995. doi: https://doi.org/10.1007/BF00994018.

Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The elements of statistical learning: Data mining, inference, and prediction. *Math. Intell.*, 27:416–426, 11 2004. doi: 10.1007/BF02985802.

Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

L.D. Hoffmann, G.L. Bradley, P. David Sobecki, and M. Price. *Applied Calculus for Business, Economics, and the Social and Life Sciences, Expanded Edi-*

tion, Media Update. McGraw-Hill Education, 2012. ISBN 9780073532370. URL https://books.google.it/books?id=6y7cZwEACAAJ.

Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Kernel principal component analysis. In Wulfram Gerstner, Alain Germond, Martin Hasler, and Jean-Daniel Nicoud, editors, *Artificial Neural Networks — ICANN'97*, pages 583–588, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69620-9.

Johansson Fredrik D. Morris Christopher Kriege, Nils M. A survey on graph kernels. *Applied Network Science*, 2020.

Vikas Hassija, Vinay Chamola, Adit Goyal, Salil S. Kanhere, and Nadra Guizani. Forthcoming applications of quantum computing: peeking into the future. *IET Quantum Communication*, 1(2):35–41, 2020. doi: https://doi.org/10.1049/iet-qtc.2020.0026. URL https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-qtc.2020.0026.

MD SAJID ANIS et. al.

Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences*, 114(13):3305–3310, 2017. ISSN 0027-8424. doi: 10.1073/pnas.1618020114. URL https://www.pnas.org/content/114/13/3305.

Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and Alán Aspuru-Guzik. Noisy intermediate-scale quantum (nisq) algorithms, 2021.

Peter Wittek, 2018. URL https://scholar.google.es/citations?user=I8mtxEAAAAAJhl=en.

M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and et al. Variational quantum algorithms. *Nature Reviews Physics*, 3(9): 625–644, Aug 2021. ISSN 2522-5820. doi: 10.1038/s42254-021-00348-9. URL http://dx.doi.org/10.1038/s42254-021-00348-9.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011. URL http://jmlr.org/papers/v12/pedregosa11a.html.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. An introduction to quantum machine learning. *Contemporary Physics*, 56(2):172–185, Oct 2014. ISSN 1366-5812. doi: 10.1080/00107514.2014.964942. URL http://dx.doi.org/10.1080/00107514.2014.964942.

Yunchao Liu. A rigorous and robust quantum speed-up in supervised machine learning. *Nature Physics*, 2021.

Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, Mar 2019. ISSN 1476-4687. doi: 10.1038/s41586-019-0980-2. URL http://dx.doi.org/10.1038/s41586-019-0980-2.

Ringraziamenti

Ringrazio il mio relatore Dr.Andrea Giachero e corelatore Dr.Leonardo Banchi per la loro disponibilità nell'aiutarmi e per i preziosi consigli, nonchè per avermi dato l'opportunità di aver potuto approfondire un argomento che così tanto mi affascinava.

Ringrazio la mia famiglia, che mi ha sempre sostenuto nei miei studi e nelle mie giornate, per avermi insegnato l'etica del lavoro e la determinazione, sia nei periodi belli che in quelli più brutti.

Ringrazio Alessandro, per aver condiviso con me in questi anni vita e università.