A Machine Learning approach to Gene Expression Analysis for Cancer Classification

Mathematics for Machine Learning

Master Degree in Data Science and Engineering

A.Y. 2020 / 2021

by

Marciano' Vincenzo
282004

Contents

Chapter 1

Abstract

In 1941, Beadle and Tatum published experiments that would explain the basis of the central dogma of molecular biology, whereby the DNA through an intermediate molecule, called RNA, results proteins that perform the functions in cells. Currently, biomedical research attempts to explain the mechanisms by which develops a particular disease, for this reason, gene expression studies have proven to be a great resource.

In a parallel way, Machine Learning techniques spread out in a multitude of applications, derived from other severals Science branches: medical diseases' classification is one of them. From Parkinson to Breast Cancer, more and more improvements have be made through the years. These innovative and truly reliable tools help the scientific community in achieving more informations and speculations to what really matters in a certain genetic disease, in order to enjoy a more accurate diagnosis.

In particular, these wide achievements bring us to 1999: in that year Golub and al.(1) wrote a paper related to the gene expression analysis in order to well classify two types of cancer, acute myeloid leukemia (AML) and acute lymphoblastic leukemia (ALL). This paper is the inspiration of this featured work: here the purpose is to study a considerable selection of ML algorithms in order to achieve the best performance possible, at least for those who are the expected final results in cancer classification.

Chapter 2

Problem Overview

The entire work will rely into 3 different datasets (provided by the Golub study itself) in which the final desired achievement is how to classify a group of patients diagnosed with ALL and AML, in order to assign tumors from the known classes.

The datasets are structured in the follow way:

- *initial* (training, 38 samples)
- *independent* (test, 34 samples)
- *ALL and AML labels* (categorical dataset, 72 samples)

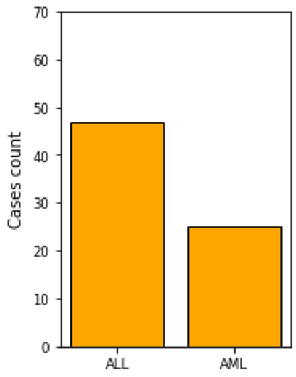
These datasets contain measurements corresponding to ALL and AML samples from Bone Marrow and Peripheral Blood. Intensity values have been re-scaled such that overall intensities for each chip are equivalent.

2.1 Exploratory Data Analysis

In this very first operative section of the work, the attention is moved into the exploration of the data we'll operate with.

Let's begin taking a look to the the labels' dataset: it has 72 istances, each of them referred to a certain patient. A preliminary action that we need to do is to check if the provided dataset has an affordable balanced label set:

Cancer Type	**Count**
ALL	47 65%
AML	25 35%



Since these percentages are not affected by a huge lack of data balance or high ratio between the two label, we can consistently assume that the related dataset is balanced and we can ignore a oversampling/undersampling approach to adjust it.

The next step is to understand better what are the features in our case. Overall, we're dealing with a genomic folder in which every row represents the activation of a certain gene descriptor: in total, 7129 values are shown for each patient, as Table 2.1 suggests (there are represented the first 4 rows and 8 columns for simplicity). This could be tied a little bit.: at a first glance, the first

Gene Description	**Gene Accession Number**	**39**	**54**	**call**	**40**	**call.1**	**42**	**...**
AFFX-BioB-5_at (endogenous control)	AFFX-BioB-5_at	-342	-90	A	-87	A	22	...
AFFX-BioB-M_at (endogenous control)	AFFX-BioB-M_at	-200	-87	A	-248	A	-153	...
AFFX-BioB-3_at (endogenous control)	AFFX-BioB-3_at	41	102	A	262	A	17	...
AFFX-BioC-5_at (endogenous control)	AFFX-BioC-5_at	328	319	A	295	A	276	...
...	...	...	...	...	...	...	...	...

Table 2.1: Appearance of Training Dataset. Shape Training Dataset: 7129 × 78, Shape Test Dataset: 7129 × 70

Patient	**AFFX-BioB-5_at**	**AFFX-BioB-M_at**	**AFFX-BioB-3_at**	**...**
1	-214	-153	-58	...
2	-139	-73	-1	...
3	-76	-49	-307	...
4	-135	-114	265	...
5	-106	-125	-76	...
6	...	...	...	...

Table 2.2: Clean and ready Training Dataset. Shape Training Dataset: 38×7129, Shape Test Dataset: 34×7129

two columns declare the same feature, so we can delete for our convenience the first one; also, a problem emerges when we take a look to the *call. n* columns: for the final goal, those columns' purpose is not stated. A first, even roughly, action is to ignore them since we can't squeeze any further and useful information. Furthermore, going deep with the analysis, **no missing values** in each column are found.

Once these assumptions are made, the process can continue with the *Data Preparation* procedure.

2.2 Data Preparation

The first and more important step to undergo for a good results' evaluation is to transpose both the Training and Test datasets, inverting rows and columns. Done that, we only require to delete the *call. n* features and to proceede with the renaming and indexing. The final result is shown in Table 2.2.

Clearly, the values' scale is still unknown. For the sake of well understanding, we decide to look deeper in the description and look how the descriptor behaves with their values. Table 2.3 well define different values domains: this also underlines the need of scaling those values in order to level each gene descriptor.

2.2.1 Standardization

For our purpose, *StandardScaler* method from *sklearn* package comes in handy: feature standardization makes the values of each data having zero mean (when subtracting the mean in the numerator) and unit-variance. This method is widely used for normalization in many machine learning algorithms (2) (e.g., support vector machines, logistic regression, and artificial neural networks). The general method of calculation is to determine the **distribution mean and standard deviation for each feature**. Then we divide the values (mean is already subtracted) of each feature by its standard deviation:

$$x' = \frac{x - \bar{x}}{\sigma}$$

Gene Accession Number	AFFX-BioB-5_at	AFFX-BioB-M_at	AFFX-BioB-3_at	...
count	38.000000	38.000000	38.000000	...
mean	-120.868421	-150.526316	-17.157895	...
std	109.555656	75.734507	117.686144	...
min	-476.000000	-327.000000	-307.000000	...
25%	-138.750000	-205.000000	-83.250000	...
50%	-106.500000	-141.500000	-43.500000	...
75%	-68.250000	-94.750000	47.250000	...
max	17.000000	-20.000000	265.000000	...

Table 2.3: Value distributions among each gene descriptor

where

- **x** is the original feature vector
- $\bar{x}$ is the mean value for that feature (*mean* in Table 2.3)
- σ is the standard deviation of the x feature (*std* in Table 2.3)

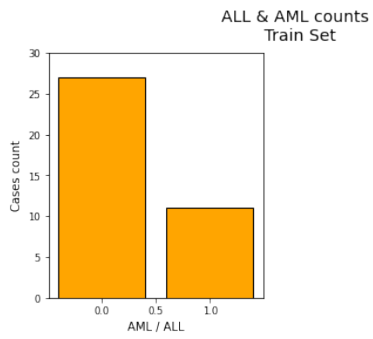
Let's be careful then: the scaling operation is done by $fitting$ the value in the **Training set** and after the transform is applied to it we then directly $transform$ the **Test set**. This relies on a quite simple idea: anything the algorithm learns, must be learned by the training set. Moreover, if we would apply the standardization also to the test set, this leads to a leakage and mismatching of data informations: the test set's mean and standard deviation could influence the overall prediction, accuracy and overfitting.

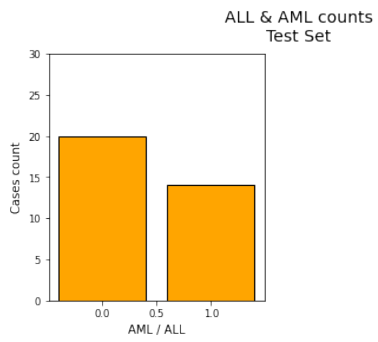
2.2.2 Label allocation

One last job has to be done to end up with this section. The **label dataset** gave us the information related to the distribution of our two classes, resulting as a quite balance set. Nevertheless, we applied some tidying up process either to train and test datasets, so now the question is how to associate the target labels with the patients. Let's try to analyze the range of patients' Id in each dataset along with their shapes:

Dataset	Patients Range	Shape
Training	1:38	38, 7129
Test	39:72	34, 7129
Labels	72	72,2

Since we don't have any duplicates or null values, we can assume without any concern, that the first **38** rows of the label dataset will be devoted to the *Training set*, while the other **34** to $Test\ set$. Then, the labels distributions among the datasets would look like this:





The last task to achieve is letting the label dataset numeric instead of categorical. This is trivial because as the problem suggests we're dealing with **binary classification**: basically we assign to ALL labels the **class 0** and to AML the **class 1**.

Chapter 3

Principal Component Analysis

As a first look to the various sets, a non-negligible issue shows up, in particular to the Training dataset: data dimentional space. As it can been seen, with 7129 features, it's also worth considering whether we might be able to reduce the dimensionality of the dataset. The tool that better satisfies our need is **Principal Component Analysis (PCA)**.

Let's be clear, though: PCA is not the only available reduction approach, **Compressed Sensing** comes in handy in various applications, so we have to decide which of the two methods is necessary. A discriminative reason that allows us to choose is the prior assumption that the original operating vector is *sparse* in some basis, i.e:

$$||x||_0 \coloneqq |\{i \ : \ x_i \neq 0\}| \leq s$$

where s is the number of nonzero elements

Certainly we can compress $\mathbf{x}$ by representing it using s pairs, but actually this is not the case: the dataset, as it's shown in Table 2.2, is clearly stated as a non-sparse features matrix. Also, to be more precise, the method *issparse*(x) from *scipy.sparse* library is used, returning as result what we've already expected: **False**. Confident of our results, we may now procede to evaluate the PCA of our data.

3.1 Eigenvalues and Eigenvectors: a step-by-step approach

In order to well understand the environment behind the correct approach to PCA, we start from scratch, firstly analyzing how to deal with the *optimization problem*:

$$\arg\min \sum_{i=1}^{m} ||x_i - UWx_i||_2^2$$
$$\text{s.t. } W \in \mathbb{R}^{n,d},$$
$$U \in \mathbb{R}^{d,n}$$

Roughly, PCA uses this problem to behave like this:

$$\begin{tikzcd} \mathbf{x} \arrow[r] & \mathbf{W} \arrow[r] & \text{compressed} \end{tikzcd}$$

$$\begin{tikzcd} \text{compressed} \arrow[r] & \mathbf{U} \arrow[r] & \tilde{\mathbf{x}} \end{tikzcd}$$

The $\tilde{x}$ remarks that if we want to reverse the operation, we only achieve an approximation of the input vector (while this does not happen in *Compressed Sensing*). According to the (3) and doing some algebraic calculations, the previous problem becomes:

$$\arg\max \; trace\Big(U^T \sum_{i=1}^{m} x_i x_i^T U\Big)$$
$$\text{s.t.}$$
$$U \in \mathbb{R}^{d,n} : U^T U = I$$

Mathematically speaking, what the equation is saying is to set the matrix U whose columns are the n eigenvectors of the matrix, let's say, **A**$= \sum_{i=1}^{m} x_i x_i^T$, corresponding to the largest n eigenvectors of the matrix A itself, then to set $W = U^T$(3).

Moreover, it's common practice to *allign* the sample data before applying PCA: that is, we apply PCA to:

$$(x_1 - \mu), ..., (x_m - \mu)$$
$$\text{where: } \mu = \frac{1}{m}\sum_{i=1}^{m} x_i$$

Now, with this assumption, an important interpretation of PCA could be **Variance Maximation** (3). Let's say we want this optimization problem:

$$\arg\max \mathrm{Var}[\langle w, x\rangle] = \arg\max \frac{1}{m}\sum_{i=1}^{m} (\langle w, x\rangle)^2$$
$$s.t. w : ||w|| = 1$$

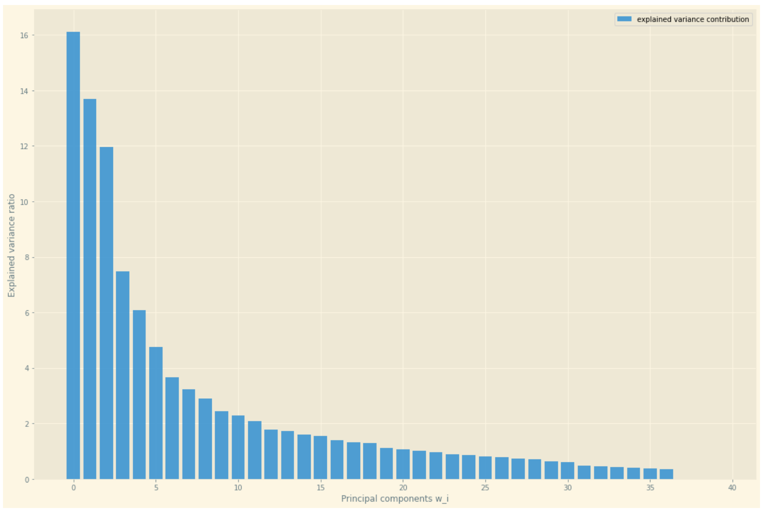
Note that for every unit vector $w \in \mathbb{R}^d$ and for every $i \in m$:

$$(\langle w, x\rangle)^2 = trace(w^T x_i x_i^T w)$$

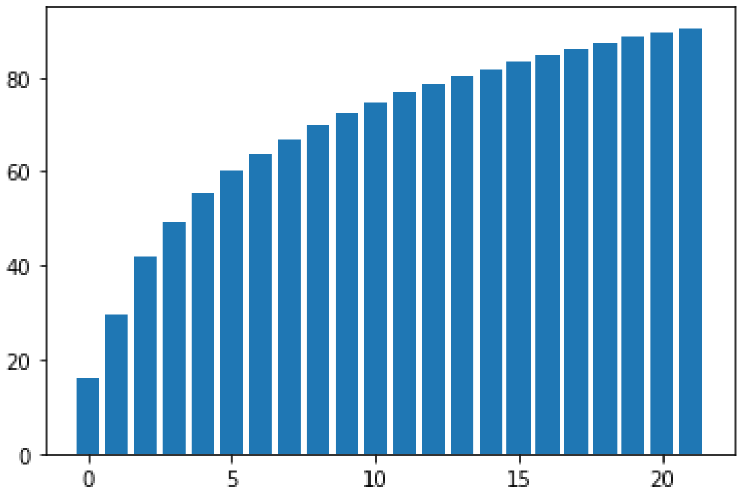
Hence, the optimization problem here coincides with the previous one, allowing us to work with the Variance Maximation due to the **eigendecomposition** of the $Covariance\ Matrix$.

3.2 Choosing the k-dimensions

We're now fully prepared to go on with the reduction. Thanks to (4) powerful tools, we're able to compute the eigenvectors and eigenvalues of our Covariance matrix: recalling that the *trace* of the Covariance matrix explains the total variance of the data, we may now be interested in understanding how much dimension are really involved in the overall variance distribution. As the following figure suggests, all the eigenvalues are sorted in a decrescent fashion to enhance better which principal component spreads the majority of the entire variance. Surprisingly, from the 37-th eigenvalue the contribution is roughly equal to 0 (the x-axis range is reduced to 40 for the sake of better visualization):

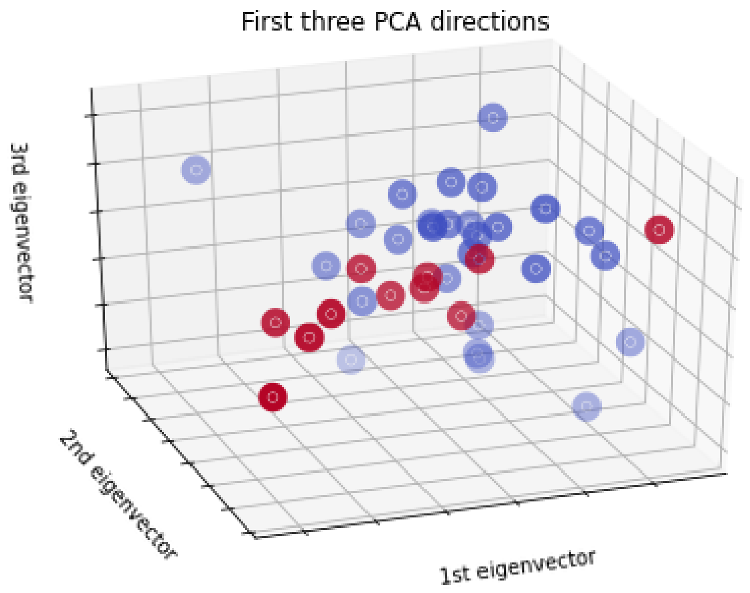


Let's also run in parallel the $pca(x).explained\_variance\_[k]$ method (5), iteratively: setting a k, let's say, to **90%** of the total explained variance ratio (that is a valuable threshold), the function returns a $k = \mathbf{22}$, as it's depicted here:



Practically speaking: a 22-dimension new data, instead of a 7129-features-descriptive dataset it's really not bad, taking in mind that this process "costed" to us only the 10% less of the entire variance ratio.

However, a 22-dimensional space is quite difficult to represent, so let's reduce for a moment the dimension and let's display it in a *3-dimensional* space. This helps a lot in the visualization of the PCA behavioural background:



N.B. This is a "dummy" representation of our features' space: in fact, we can see- the agglomerative behaviour of our classes here for the **first 3** dimensions, but just for simplicity. Visually, we don't know (at least for our tools) anything about the distribution of the other 19 dimensions.

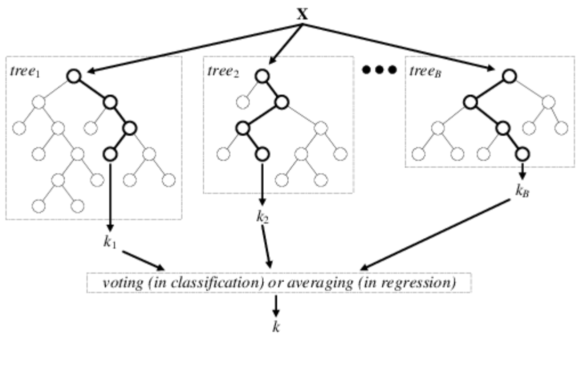
Chapter 4

Machine Learning Models

In this section we are going to analyze several **Machine Learning Models** in order to achieve the best configuration for our task. In particular, each model will give a $confusion$ *matrix*, which parameters are described in the *Appendix* section. Thanks to these results, we can generalize which model will be the best for our purposes.

4.1 Random Forest Classifier

The first model we are approaching to is **Random Forest Classifier**. A random forest is a classifier consisting of an ensemble of trees (a collection of decision trees) where each is constructed by the following process: we apply an algorithm **A** on the training set **X** and an additional random vector γ sampled *iid* from some distribution. This is followed by the **majority vote** over the predictions of the group of trees: making it simple, we create a *Dividi et Impera* approach, that (as we'll see later) brings fairly good results.



4.1.1 Bootstrap

Importing $RandomForestClassifier$ class (5), we see that between all its parameters, a boolean one will capture our attention: **Bootstrap** (which default value is $True$).

Bootstrap is a widely applicable and extremely powerful statistical tool that can be used to quantify the uncertainty associated with a given estimator or statistical learning method (6). The training algorithm of Random Forest Classifier applies the *bootstrap aggregating* procedure, also called **Bagging**, on its trees. Let be:

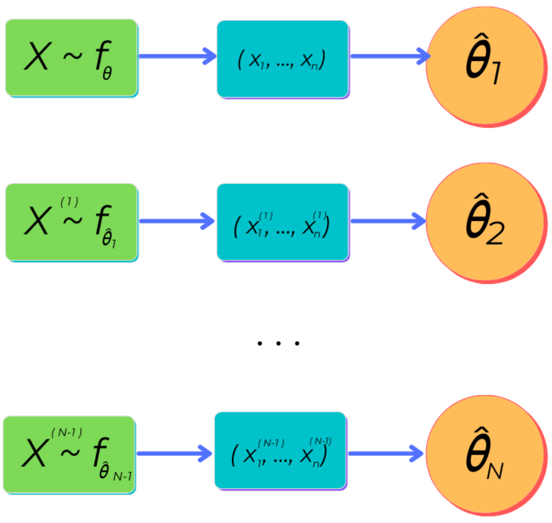
$$X = x_1, ..., x_n \rightarrow \text{our training set}$$

$$Y = y_1, ..., y_n \rightarrow \text{our target set}$$

Also, we can rewrite X as a function f_θ where θ is the parameter to estimate:

$$X \sim f_\theta$$

Now, Bootstrap goal is to repeatedly select (N times) a **random sample with replacement** of the training set X and fit the trees to this updated data. Roughly speaking, the process is the following:


$X \sim f_\theta$
$(x_1, ..., x_n)$
$\hat{\theta}_1$
$X^{(1)} \sim f_{\hat{\theta}_1}$
$(x_1^{(1)}, ..., x_n^{(1)})$
$\hat{\theta}_2$
. . .
$X^{(N-1)} \sim f_{\hat{\theta}_{N-1}}$
$(x_1^{(N-1)}, ..., x_n^{(N-1)})$
$\hat{\theta}_N$

So, in the first step the sample X is given and we simply have to find $\hat{\theta}_1$; then, we feed the algorithm with a brand-new sample $X^{(1)}$, generated by the previously found parameter and this repeats until we reach the $\hat{\theta_N}$ estimate.

Eventually, the unseen samples (like our *Test dataset*) can be predicted by two different approaches:

- **majority vote** - *Classification Task* (like our datasets);
- **averaging predictions** among the test set - *Regression Task*. In this case, we have:

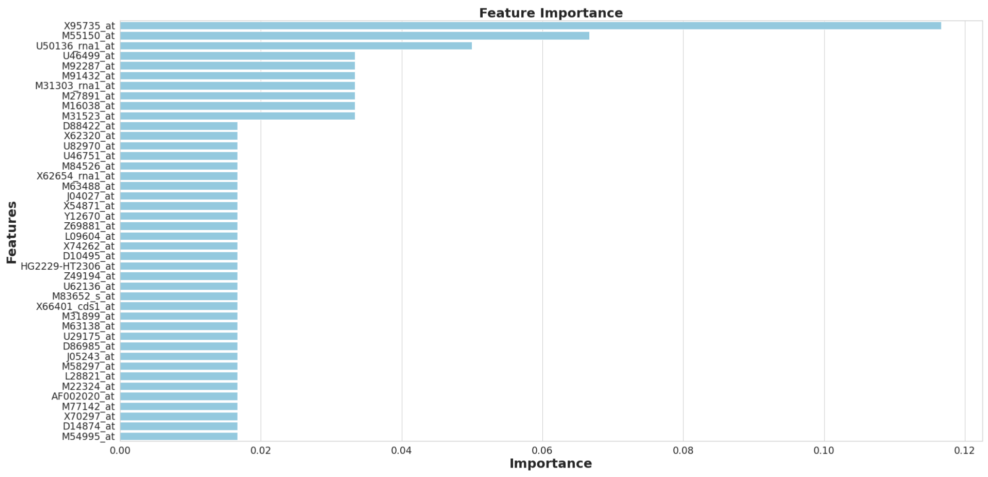
$$\hat{\theta}^* = \frac{\sum^N \hat{\theta}_i}{N}$$

That's where the power of Bootstrap comes from: we create several "tiny" predictors (tree) that *individually* lead to an high noise sensitivity; however, if they are combined together and work, let's say, *cooperatively*, the average of many trees is no more noise-friendly, as long as the trees are not correlated. That's why Bootstrap sampling is a **de-correlated way** to approach each tree family: by showing them different training sets everytime, it decreases the overall variance, still maintaining the same bias.

Also, let's suppose we have some features that are presented as the *strongest correlated* candidates for the target predictions: this is quite dangerous, because it may lead to a correlated behaviour for each predictor. That's why Random Forest Classifier uses another useful tool, that is the *random split* in generating each subtree: this process is essential in exploiting at a maximum level the algorithm, because it drives to several situations with a gained surplus in accuracy (7).

4.1.2 Tuning

Let's get back to our application. We have to create the baseline for our Random Forest Classifier. So, firstly we fit the model with our data, then we may have a look in those who are the most **important features** for the model:



This picture considers only the 41 *gene descriptors* with a non-null **Gini Index** metric. With *Entropy (Information Gain)* and *Train Error*, Gini Index yet represents another gain definition:

$$C(a) = 2a(1 - a)$$

We then proceed with the **tuning process**. One can be interested in understand how the PCA reduced set could be useful in our situation: nevertheless, we need to take in mind that each child tree better improves its learning by studying original (or at least scaled) features. For the experiment, *4 setups* are provided:

1 Baseline Random Forest,

2 Baseline Random Forest with PCA,

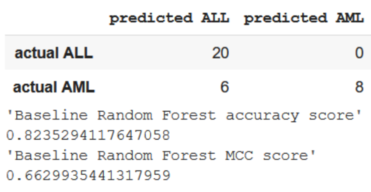
3 Hyperparameter Tuned Random Forest,

4 Hyperparameter Tuned Random Forest with PCA

For each of them, two metrics are used to evaluate the efficiency: **Accuracy Score** and **Matthews Correlation Coefficient**. The latter is contextually very important: this metric is preferred to *F1 score* for several reasons in computational biology, but roughly speaking the power of MCC comes from its evaluation score that is high only if our classifier is doing well on both the negative and the positive elements [(8) (9)]. Since we are dealing with cancer types diagnosis, this is imperative.

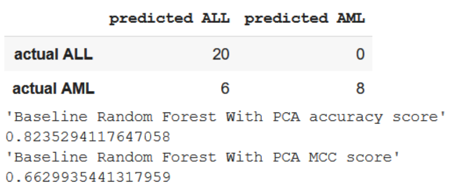
4.1.3 Results

With this concept in mind, we initialize the hyperparameters , listed in Table 4.1, to run a *RandomizedSearchCV* (5) on our data. Then, a confusion matrix for each model has to be created and as a final step results are printed, along with their metrics' scores:



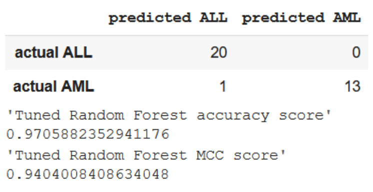
	predicted ALL	predicted AML
actual ALL	20	0
actual AML	6	8

```
'Baseline Random Forest accuracy score'
0.8235294117647058
'Baseline Random Forest MCC score'
0.6629935441317959
```



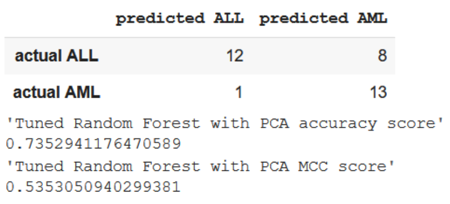
	predicted ALL	predicted AML
actual ALL	20	0
actual AML	6	8

```
'Baseline Random Forest With PCA accuracy score'
0.8235294117647058
'Baseline Random Forest With PCA MCC score'
0.6629935441317959
```



	predicted ALL	predicted AML
actual ALL	20	0
actual AML	1	13

```
'Tuned Random Forest accuracy score'
0.9705882352941176
'Tuned Random Forest MCC score'
0.9404008408634048
```



	predicted ALL	predicted AML
actual ALL	12	8
actual AML	1	13

```
'Tuned Random Forest with PCA accuracy score'
0.7352941176470589
'Tuned Random Forest with PCA MCC score'
0.5353050940299381
```

Hyperparameter	**Values**
n_ estimator	np.linspace(start=100, stop=1000, num=10)
max_features	log2, sqrt, 0.6, 0.65, 0.7, 0.75, 0.8
max_ depth	np.linspace(start=1, stop=15, num=15)
min_ samples_ split	np.linspace(start=2, stop=50, num=10)
min_ samples_ leaf	np.linspace(start=2, stop=50, num=10)
bootstrap	True, False

Table 4.1: List of RF's hyperparameters choices.

It looks like the **Hyperparameters Tuned Random Forest Classifier** achieved the best in *Accuracy* but mostly important in *MCC*, with a high score of **0.94** (the range of MCC is between -1 and +1). Its list of hyperparameters' values is depicted here:

Hyperparameter	**Value**
n_ estimator	400
max_features	0.6
max_ depth	3
min_ samples_ split	12
min_ samples_ leaf	2
bootstrap	True

Also the two baselines worked well, with same results in both the two metrics. The worst is the Hyperparameters Tuned classifier with PCA, however we expected it since we discuss about the working procedure between a RF and dimensionally reduced data.

4.2 Support Vector Machines

As long we went through the overall work, the high dimensionality of our data is no longer a mistery. For this reason in this section we are going to exploit **Support Vector Machines** known as **SVM**, a very useful tool to tackle the *sample complexity* and *computational complexity* challenges raised up by the high dimensionality of feature space.

4.2.1 "Vanilla" SVM

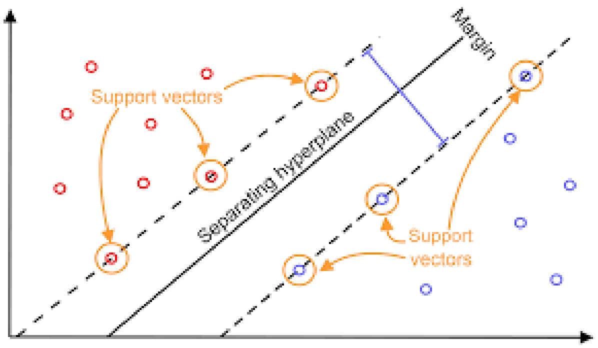
SVM paradigm overcomes to sample complexity by searching for *"large margin"* separators through **halfspaces**: a halfspace separates a training set with large margin if all the examples are not only on the correct side of the dividing hyperplane but also *far away* from it.

Let's call **S** a training set of size m, where each $x_i \in \mathbb{R}^{n,d}$ and each target $y_i \in \pm 1$. This set is **linearly separable** if:

$$\forall i \in m, y(\langle w, x \rangle + b) > 0$$

Hence, we have to find the separator across certain points, however, as an example, there are too many lines as the best candidate one which divides two sets both of two points.

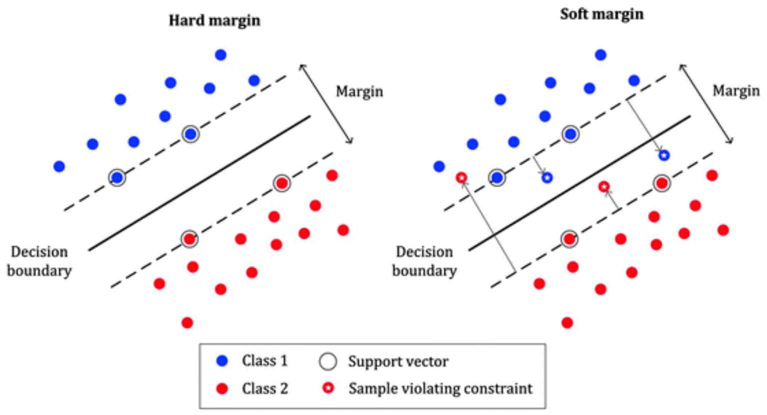
Our intuition is to overcome this choice by introducing the definition of **margin**: a margin of a hyperplane with respect to a training set is defined to be the *minimal distance* between a point in the Training Set and the hyperplane itself. In the case of a *large* margin, it will separate the training set even if we add some noisy perturbation to it.



The "Support Vectors" words come from the associating feature vectors that lie on the margin edges, i.e. the *support* of the belonging class, and for which:

$$y(\langle w, x\rangle + b) = \pm 1$$

From here, two definitions of SVM are presented:



- **Hard SVM (strong assumption)** → the learning rule in which we return a hyperplane that separates the training set with the largest possible margin:

$$\arg\max \min |\langle w, x\rangle + b|$$

$$s.t.\ \forall i, y_i(\langle w, x\rangle + b) \geq 1$$

$$(w, b) : ||w|| = 1, i \in [m]$$

  where the output of this SVM is a solution of the previous equation. Intuitively, hard-SVM searches for **w** of minimal norm among all the vectors that separate the data and for which $|\langle w, x_i\rangle + b| \geq 1$ for each i, enforcing the margin to be equal to 1. Therefore, finding the largest margin halfspace is equivalent to finding w whose norm is minimal.

- **Soft SVM (relaxed)** → with hard-SVM we introduced some basic and reliable strong concepts, however as its formulation assumes, we *have* to deal with a linearly separable training set. This in very unlucky to happen. Nevertheless Soft-SVM comes in handy for our problem: it can be viewed as a relaxation of the Hard-SVM rule, applied even if the training set is not linearly separable.

  As the theory behind the Optimization algorithms illustrates, a **natural relaxation** allows the costraint to be *violated* for some examples in the set. In our case we replace each constraint into:

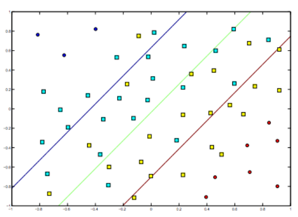
$$y_i(\langle w, x\rangle + b) \geq 1 \rightarrow y_i(\langle w, x\rangle + b) \geq 1 - \xi_i$$

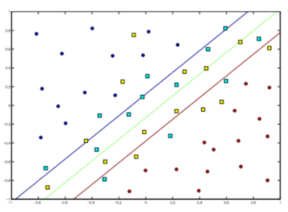
  with ξ_i measuring **how much each constraint is being violated**.

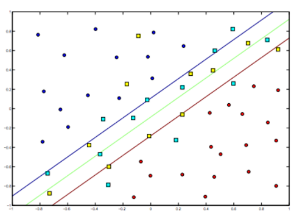
Obviously, this requires a *trade-off* between margin width and constraints' violations. This is measured by a parameter λ. Hence, the problem to be solved is:

$$min\Big(\lambda||w||^2 + \frac{1}{m}\sum_{i=1}^{m}\xi_i\Big) \rightarrow min(\lambda||w||^2 + L_S^{hinge}((w,b))$$

where $L_S^{hinge}((w,b)$ is the **averaged hinge loss** on set S. The purpose is clear: since a Soft-SVM algorithm has a bias toward *low norm* separators, the objective function we want to minimize penalizes either for **training errors** and **large norm**. Here that's how the *regularization parameter* λ in a linear SVM works:


($\lambda = 0.1$)


($\lambda = 1$)


($\lambda = 1000$)

This can be applied also to the **homogeneous hyperplane**, situation that from now on we'll consider, for the sake of simplicity. An homogeneous case of the starting problem consists in setting the bias equal to 0:

$$\min ||w||^2$$

$$s.t.\ \forall i,\ \ y_i(\langle w, x\rangle) \geq 1$$

4.2.2 Kernel Trick

Many properties of SVM have been obtained by considering the *dual* of the previous problem, i.e. working with the Lagrangian relaxation of it.

With some algebraic calculations, we eventually notice that if α (parameter of the relaxed constraint function $g(w)$ used for our purpose) is fixed, the optimization problem with respect to $\mathbf{w}$ is unconstrained and the objective is differentiable, thus:

$$w = \sum_{i=1}^{m} \alpha_i y_i x_i$$

The game is done: we make a very powerful assumption here, stating that the solution must be in the *linear span of the examples*. So the dual problem becomes a problem in which only **inner products between instances** are involved.

As a matter of fact, it's convenient to remark that we are very often dealing with multi-dimensional feature space, so we need a function that undergoes through this issue, to understand how to *map* our space properly. The common solution to this concern is **kernel based learning**:

$$K(x, x') = \langle\phi(x), \phi(x')\rangle$$

where a kernel is the key to solve our problem, since contextually speaking is used to describe inner products in a feature space. Here one can imagine of:

- $K \rightarrow$ specifying similarity between instances
- $\phi \rightarrow$ mapping the domain set into a space in which similarities are expressed with inner products

Hyperparameter	**Value**
C	0.1, 1, 10, 100
kernel	linear, rbf, poly, sigmoid
gamma (only for non-linear kernel)	1, 0.1, 0.01, 0.001, 0.00001, 10
decision_function_shape	one-vs-rest (Fixed)

Table 4.2: Hyperparameters' values for SVM Tuning.

The advantage is clear: thanks to this *trick* we implement linear separators in high dimensional spaces without having to specify points in that space or expressing the embedding ϕ explicitly. As (3) states, there exists a vector $\alpha \in \mathbb{R}^m$ such that:

$$w = \sum_{i=1}^{m} \alpha_i \phi(x_i) \rightarrow \textbf{optimal solution of the problem}$$

Computing Kernels should be efficient, much more than computing $\phi(x)$ and $\phi(x')$ explicitely.

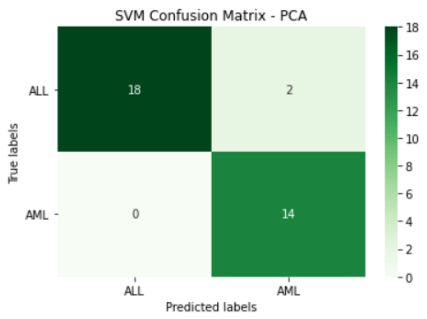
4.2.3 Results

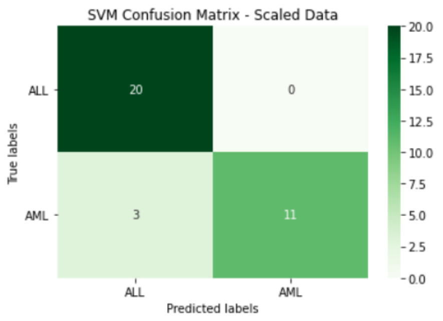
Two experiments have been conducted: one with the previously achieved PCA and one with the scaled data. A *GridSearchCV* (5) has been exploited in evaluating the best hyperparameters, in the range of values showed in Table 4.2.

According to (5), the strenght of *lambda* parameter shown in the previous section is inversely proportional to C while *gamma* is the γ parameter related to non-linear kernels, defining the *influence* of a single sample in which low values meaning 'far away' and high values meaning 'very close'; for what concerns the *decision function shape* parameter, we choose to select "ovr" since this is a binary classification problem and the documentation suggests intuitively to work with each class as a **one vs rest**.

The results obtained are illustrated here along their confusion matrix:

Dataset	Accuracy	MCC	kernel
PCA	0.941	0.887	*linear*
Scaled	0.912	0.827	*linear*





Both the approaches perform very well, however the power of PCA allows us to gain a **3%** in accuracy and a ∼0.9 MCC score, becoming the best set to perform with.

4.3 Naive Bayes

A **generative approach** assumes that the underlying distribution over the data has a specific parametric form and our goal is to estimate the parameters of the model (*parametric density estimation*). Thus, if we succeed in learning this kind of distribution accurately we are considered to be experts in the sense that we can predict by using the **Bayes** optimal classifier: for example, sometimes is computationally easier to estimate the parameters of the model than to learn a discriminative predictor.

The **Naive Bayes classifier** is a classical demonstration of how generative assumptions and paramter estimations simplify the learning process. In our workcase, we have to predict a label $y \in 0, 1$ starting from a vector of features $\mathbf{x}$, in which each x_i is represented in the binary space. The *optimal Bayes classifier* is stated as following:

$$h_{Bayes}(x) = \arg\max \ \mathbb{P}_\theta[Y = y|X = x]$$

$$\text{with } y \in 0, 1$$

In the Naive Bayes model we make the $Naive$ assumption that given the label, the features are **independent** of each other. That is:

$$\mathbb{P}_\theta[Y = y|X = x] = \prod_{i=1}^{d} \mathbb{P}_\theta[Y = y|X = x]$$

4.3.1 Maximum Likelihood Estimator

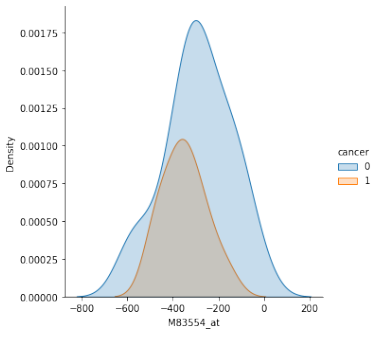
Still, we have to understand how to comupute our probability of interest. Here comes in help the **Maximum Likelihood Estimator**.

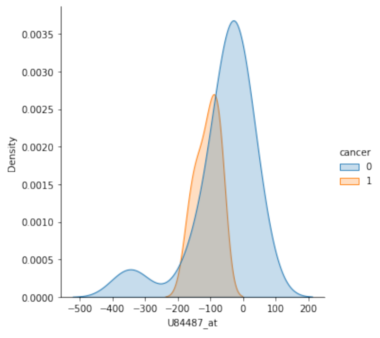
Let X be a continuous random variable. Then, we define the Likelihood as *log of the density* of the probability of X at x, that is: given a training set $\mathbf{S}$ sampled by $\mathbb{P}_\theta$, the likelihood of S will be defined as

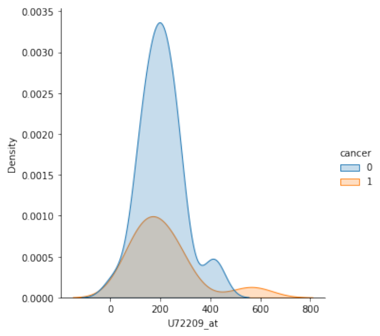
$$L(S;\theta) = \sum_{i=1}^{m} log(\mathbb{P}_\theta(x_i))$$

$$\text{where MLE is: } \frac{dL(S;\theta)}{d\theta} = 0$$

As an example, let's consider 3 different gene descriptor features in our dataset, picked randomly. Their distributions look like the followings:







These function shapes are quite familiar: the key idea for our prior assumption is to model the data as a *Gaussian Random Variable*. This fact is also extended to the other 7126 features, since other random tries with the gene descriptors confirm what we've just suggest.

A Gaussian RV, for which the density of a sample X is parametrized by $\theta = (\mu, \sigma)$, is defined as follows:

$$\mathbb{P}_\theta(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

After some mathematical simplifications, we obtain the **Maximum Likelihood Estimates**:

$$\hat{\mu} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

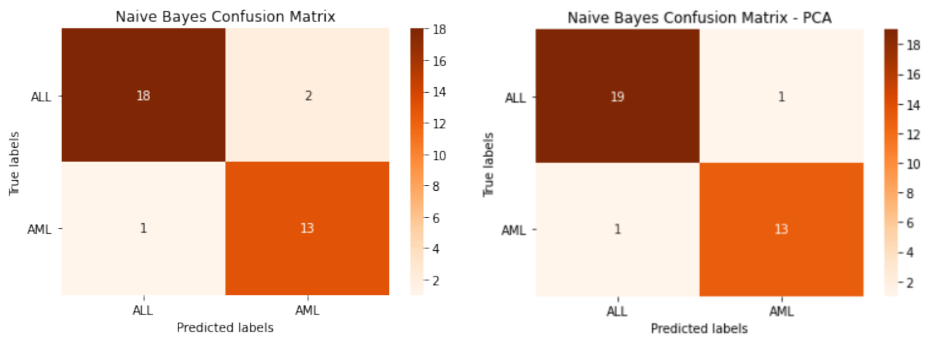
$$\hat{\sigma} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_i - \hat{\mu})^2}$$

For these reasons and as the previous pictures suggest, we are going to approach to a Naive Bayes Classifier by using *GaussianNB* (5).

4.3.2 Results

We fed the classifier by the contribution of the raw Training Set and the PCA-applied Set. Confusion matrix for both the approaches has been evaluated and the results computed are:

Dataset	**Accuracy**	**MCC**
PCA	0.941	0.879
Training Set	0.912	0.821

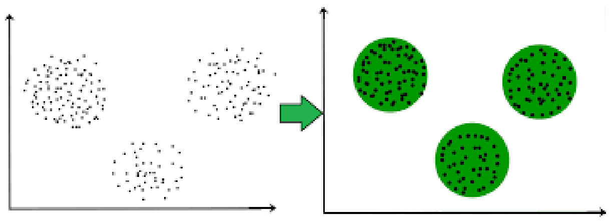


These scores remark again how PCA behaves well in a high-dimensional space prediction. Surprisingly, both the *Accuracies* are equal to those achieved by the SVM model, while for *MCC* we obtain a slightly lower result either for PCA and raw data.

4.4 Clustering (K-Means)

Clustering is one of the most widely used techniques for exploratory data analysis, across all disciplines (e.g. computational biology and genomics). Intuitively, clustering is the task of grouping a set of instances in order to make similar objects ending up in the same group, viceversa for dissimilar objects. Still, it's not clear how to come up with a more precise definition (3).

Let's suppose we have two parallel lines of points and we would like to cluster them into two clusters: we may have a clustering method that emphasizes not having far-away points share the same cluster or another method that relies on unseparating close-by point. Furthermore, there is the lack of *ground truth* for clustering, which is a common problem in **unsupervised learning** methods, like this one: there are no labels to predict, yet we wish to organize the data in some meaningful way.



4.4.1 k-Means

A widely used approach to clustering starts by defining a *cost function* over a parametrized set of possible clusterings and the goal (G) is to find for a given input (X,d) a clustering C so that $G((X, d), C)$ is **minimized**. However, this kind of optimization problem is related to **NP-Hard**, for these reasons when people talk about a *k-type-clustering* is referring to an **approximation** algorithm rather then the corresponding exact solution of the minimization problem. So, the common objective of these clustering methods is to choose a parameter k such that it's the most suitable for the given problem.

k-means objective function is one of the most popular clustering approaches: in k-Means the data in partitioned into $C_1, ..., C_k$ disjointed sets in which each C_i is related to a peculiar centroid μ_i. The i-th centroid is defined as:

$$\mu_i(C_i) = \arg\min \sum_{x \in C_i} d(x, \mu)^2$$

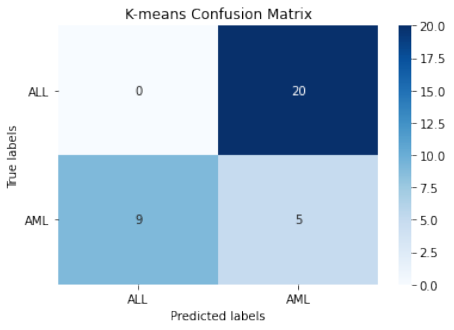
while the $objective\ function$ is:

$$G_{k-means}((X, d), (C_1, ..., C_k)) = \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_i(C_i))^2$$

There are other k-clustering approaches such as *k-medoids* and *k-medians*, very robust to noise and outliers. Since in our dataset none of these issues has occurred, we may proceede by exploiting *k-means* instead.

4.4.2 Curse of dimensionality

Let's put our data in a vanilla-fashion *KMeans* (5).



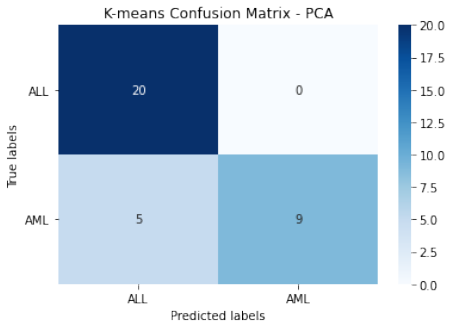
The classifier works very poorly: with an *Accuracy* of **14.7%** and a *MCC* score of **-0.717**, it certainly wouldn't be used for a diagnonis. However, this behaviour doesn't surprise at all: we are dealing with a convergence of the distances between each data, due to its many thousands of features available. Multiple dimensions are hard to think in, impossible to visualize, and, due to the exponential growth of the number of possible values with each dimension, complete enumeration of all subspaces becomes intractable with increasing dimensionality. The *concept of distance* becomes less precise as the number of dimensions grows, since the distance between any two points in a given dataset converges. The discrimination of the nearest and farthest point in particular becomes meaningless:

$$\lim_{d \to \infty} \frac{dist_{max} - dist_{min}}{dist_{min}} = 0$$

For this reason, more than ever, a PCA-based approach to bring down the amount of features to a relatively small set could provide to *k-Means* a good evaluating process.

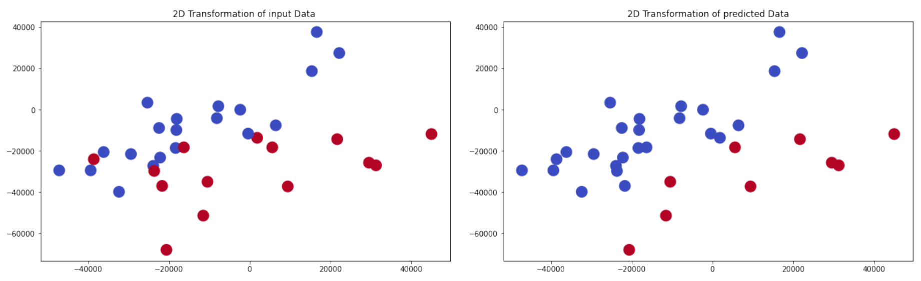
4.4.3 Results

We may now be interested to understand the correctness of the previous assumption. Let's call back our 22-dimensional PCA set and feed the algorithm with it:



As we expected, the classifier performs widely better than a vanilla one: with a **85.3%** in *Accuracy* and a *MCC* of **0.717**, the reduced dimension idea goes in the right way, allows us to achieve a very high gain in both the scores. However, until now it's presented as the "worst", taking in mind the previous classifiers' results. Still, it maintains its fairly good usability.

Finally, in the following pictures the predictions vs the true data are illustrated. Again, this is simplified representation using the first two eigenvectors, only for a simpler visualization purpose:



Chapter 5

Conclusions

Machine Learning applied to biogenomics is the evolution of research in terms of efficiency, reliability and time wasting. In this work several tools have been proposed, not only from a merely applicative point of view but also from a mathematical perspective. We need to take in mind that there is no "it's good because it is done this way": eventually, a deeper study can prevent some misleading assumptions that in medicine and in particular in cancer diagnosis are not allowed at all.

Overall, the models behaved very well. Among those, **Random Forest Classifier** with a fine *Hyperparameters Tuning* on the scaled dataset has achieved the best accuracy and MCC score possible. Moreover, a further study may be led by the usage of Neural Networks and the relative approach to graph paths between all the genes' values. For any doubt feel free to check my repository. As a final point, the following table remarks all the best results achieved for each model:

Classifier	**Accuracy**	**MCC**
Hyperparameters Tuned Random Forest (Scaled Data)	**97%**	**0.94**
SVM (PCA)	94.1%	0.89
Gaussian Naive Bayes (PCA)	94.1%	0.88
k-Means Clustering (PCA)	85.3%	0.72

N.B at the same accuracy score we prefer a higher MCC since it says more about the label predictions and how the classifier behaves about the *False Positive* and *False Negative* achieved.

Chapter 6

Appendix

6.1 Coding

Python is the programming language used for this work and it is more intuitive than other programming languages with many frameworks, libraries, and extensions that simplify the implementation of different functionalities: this allows it in being highly suitable for complex machine learning tasks. Moreover Python has a great community support.

The code has been completely written on a Google Colab Notebook, a Python development environment that runs in the browser using Google Cloud.

The most used common libraries for this analysis are:

- *PANDAS*: it's a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license (10).

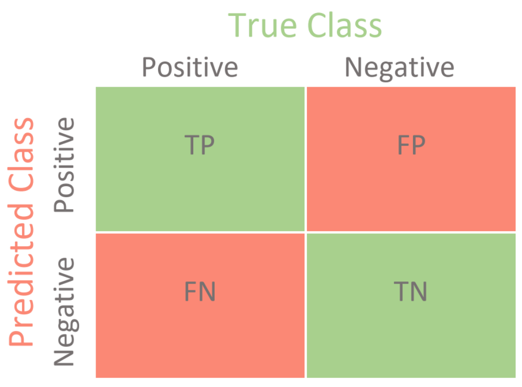
- *NumPy*: Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays (4).

- *SciKit*: Scikit-learn (formerly scikits.learn and also known as sklearn) is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy (5).

- *Seaborn*: Seaborn is a Python data visualization library based on *matplotlib* (11). It provides a high-level interface for drawing attractive and informative statistical graphics (12).

6.2 Metrics

In machine learning, the **Confusion Matrix** consists in a table that tells us about the performances of the used algorithms. In our study, the configuration used is: rows represent the instances in an actual class while each column represents the instances in a predicted class.



Where:

- **TP** are the True Positives
- **TN** are the True Negatives
- **FP** are the False Positives
- **FN** are the False Negatives

For our purposes, two metrics are adopted:

- **Accuracy**
$$\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{P}+\mathrm{N}}=\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}$$
- **Matthews Correlation Coefficient (MCC)**
$$\frac{\mathrm{TP}\times\mathrm{TN}-\mathrm{FP}\times\mathrm{FN}}{\sqrt{(\mathrm{TP}+\mathrm{FP})(\mathrm{TP}+\mathrm{FN})(\mathrm{TN}+\mathrm{FP})(\mathrm{TN}+\mathrm{FN})}}$$

The Matthews correlation coefficient (MCC) or phi coefficient is used in machine learning as a measure of the quality of binary (two-class) classifications, introduced by biochemist Brian W. Matthews in 1975.

The MCC doesn't depend on which class is the positive one, which has the advantage over the F1 score to avoid incorrectly defining the positive class.

Bibliography

[1] S. D. Golub TR, "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. science.," 1999 Oct 15.

[2] J. Grus, *Data Science from Scratch Sebastopol, CA: O'Reilly pp. 99, 100.*

[3] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms.* USA: Cambridge University Press, 2014.

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*, vol. 112. Springer, 2013.

[7] T. K. Ho, "A data complexity analysis of comparative advantages of decision forest constructors," *Pattern Analysis & Applications*, vol. 5, no. 2, pp. 102–112, 2002.

[8] D. Chicco, "Ten quick tips for machine learning in computational biology," *BioData mining*, vol. 10, no. 1, pp. 1–17, 2017.

[9] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC genomics*, vol. 21, no. 1, pp. 1–13, 2020.

[10] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020.

[11] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[12] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021.