Presentation text

This text must be read with its slide show.

I) What is ML?

Traditional computer science VS ML

In Traditional computer science, we get output by inputting data and a program. As an example, you want to convert a music video from yt into a mp3 file. The data is the music video, the program converts the video into a mp3 file.

ML reverses the process. We input data and output, and we get a program. Definition: designs algorithms that learn how to make decisions based on empirical data.

Let’s say we want to predict if a patient has Alzheimer based on IRM brain pictures. In traditional computer science, we are totally blocked on the program writing, we don’t know how to interpret a brain picture from an Alzheimer patient.

In ML, we associate each photo (data) with its known output to learn the program. The data are examples from the past. The computer builds a model which now knows how looks like an IRM taken from an Alzheimer's patient.

This learning phase is called training. We give to the computer the ability to make decisions based on data without explicit instructions. Then we test the model learned by just giving the data and comparing the prediction with the output to know how well the model does. The goal of an ML model is to predict the right output given any external data.

3 types de ML:

- Supervised (the section we will be focusing on): The algorithm learns by making use of labeled data. The example we just saw was from SML (Supervised ML). It’s when we know the outputs in advance from our data, the label being the outputs
- Unsupervised: algorithms that do not use any outputs in the training phase. PCoA is an Unsupervised ML algo. Basically, it clusters data that seem similar without any human assumptions.

- Reinforcement (AlphaZero in chess, shogi, go) System of reward/punishment. Creation of autonomous agent. Not useful in biology research

II) SML basics

Terminology

feature vector: n-dimensional vector of numerical features that represent some object. We call it Xi, and each of its coordinates is a feature.

Labels (target values): Characteristics on which the feature vectors of a dataset are categorized. These are the values we want to predict given the feature vectors. We call them Yi

Model: A program that can find patterns or make decisions from a previously unseen dataset. Tries to associate the right label to each feature vector.

Classifier: algorithm that is used to map the input data to a specific category. It tries to find the best model.

A dataset is formed of n couples of (Xi,Yi) drawn from the same distribution P. Here it's a representation of a dataset, with 4 feature vectors associated with their labels. Each feature vector has 4 features.

The P distribution is unknown because the ultimate goal of a model is to guess the underlying distribution. You can imagine the dataset as a tiny look at the whole P distribution. The bigger the dataset is, the more it would look like P. We want our model to guess every label given a feature vector from P, even if it isn't in the dataset.

Translation in new terms

(After reflection on how a picture is mapped into a vector, I would say that the dimension of the feature vector is 3 times the number of pixels of the picture. The 3 first coordinates of the vector refer to the R G B values from the RGB code.

2 types of SML algorithms: Classification

Classification: It's when the set of labels Y has a finite size, we can count the labels. It can be 2: binary classification, or much more, 10, 100000: multiclass classifica°. The goal here as we saw before, is to assign the right label to a given feature vector. (Binary classification picture): the colors represent the 2 labels. We want to find a model that

separates points from the 2 labels, each point being a feature vector (features being the abscissas and ordinates)

The labels are not forced to be numeric: nature of a mail {spam,non-spam}, topic of a newspaper article {sport, politics, art..}

Useful cases of classification: spam filter

We don’t know P, our goal is to model it.

We are in a binary classification problem, the set of labels is {spam,non-spam}.

The dataset contains thousands of emails with their known nature, 0 referring to non-spam and 1 to spam.

You have to know that every kind of information can be contained in a vector. That is true because everything you are doing on computers is translated into a sequence of 1 and 0, easily put in a vector.

Here each Xi represents one email or rather the word occurrences from this email. The dimension of the vector here is the number of words in the English dictionary (+80 000). We assign to each coordinate how many times we find a specific word in the mail. We do this for all the mails of the dataset and then we can build a model that assigns weights on each of these words according to their likelihood of being used in spam. With this model, we can predict the nature of external emails.

2 types of SML algorithms: Regression

Def: Y (=set of labels) is an interval of real numbers. Here the goal is to find the model (here the line) that best fits the actual values which are points. Examples: predicting the height of people given their biological features

Useful cases of regression: House Price prediction

We predict at which price we could sell our house given examples from neighbors that really left. The label is the selling price here, and the features could be the surface area, distance to the school, and everything that could affect the price. Each (Xi,Yi) in the dataset is a house that had been sold from the same distribution. Here it could be limited to a neighborhood or a city, but drawing them from the same distribution is primordial. Because a Xi/Xj could have the same features but way different prices since they are not in the same area.

Training process: démo perceptron

I want to show you a demo of an algo called the Perceptron to see what a training phase is like.

It's a classification algorithm. It really looks like a regular algorithm. We categorize this as a ML algo since all it does is update the parameters w and b until it gets all points correct. Basically, it tries to draw a straight line defined by its parameters that separates the points from the 2 labels

2 parameters: in the algo, there is just w but it's just because they did a trick, they put the information of b in this w (which becomes 2-dimensional) but it's basically the same thing.

Scan all the points in the same order:

When the classifier spots that one point is misclassified, we update (w,b), and we continue. When a scan iteration on all the points has 0% of error → the algo converges and we get the final parameters that define our model learned.

This particular algo builds only models that have 0% training error. Indeed, if the dataset is not linearly separable (you can't separate it with a straight line), for example, if you put a red point in the middle of the blue points, the algo will run endlessly, hoping to find a miracle

The demo I used: https://mlweb.loria.fr/book/en/perceptron.html

Splitting the dataset

Before doing any kind of training, we have to split the datasets into training/testing sets.

The training set is used to find the best model based on the data.

The test set is used to test the model we found. The split is primordial since the test set is used as an objective measure of how well the model does.

Measuring the accuracy of the model on the training set would be like cheating. You created a model on a dataset to have a low error rate on it, it is not fair to measure the accuracy on it.

The test set must be used once in its lifetime since it has to be objective. If you use it a 2nd time, you are biased. Taking a look at the testing set will make you build a model that has a lower error rate because you saw how the test set looks like, and how the data behaves. It's like you already have the answers before your exam.

Training/Testing process: missing set?

After splitting, we try many models on the training set, some work, some don't. When we find one that makes us satisfied, we test it on the test set. But we see there is an issue.

We choose the red model that has 0% of error, but it doesn't do well on the test set. The other model is way better. We get a low training error (error on the training set) and a high testing error (error on the testing set). This is characteristic of overfitting.

Overfitting

def: the model learns the training data by heart, instead of learning the underlying patterns. (top right picture): the model does everything it can to get 0% of error in training, it doesn't respect the distribution. But we are interested in the test error, not the training error. And in overfitting the test error is really high. A good model would be the left one, even if it gets some points wrong. A good model is generalized

On the top right picture, we can easily add circles that respect the distribution but would be misclassified. Overfitted model can't perform well on unseen dataset (overfitting is a common problem in ML, that is not completely removed by what I am introducing you in the next paragraph. There is many more, that I will talk about in the additional pages).

Let's come back to the dataset split. To overcome such situations of not knowing which model to choose, we add an intermediate set, the validation set.

Training/Testing process: missing set!

The idea of the validation set is you can do infinitely back and forth between training and validation set. See the validation set as a test set, but you can actually run as many models as you want on it. We learn many models on the training set, when we think our model is doing fine we test it on the validation set. All the models overfitted to the training set would have a very high error rate on the validation set. The validation set kills overfitted models to the training set.

Final Training/Testing process

This is the real process. Usually, we split in 70/10/20 in the training/Validation/Testing set from the initial dataset. Here the sets are already split. Since we no longer pick the best model on the training set but on the validation set, it no longer suffers from overfitting. When we found the best model according to the training & validation set, we

get the objective error rate by running on the test set, which we call generalized error or test error.

Dataset split: randomly!

You have to be careful about how you are splitting. The sets have to be drawn from the same distribution.

Let's say your dataset is ordered by its set of labels. If you split your dataset with a knife, you would only get here 1 as the label in your test set, very few in your training set, and your model will not be accurate since the set doesn't have the same distribution. You have to randomly fill your 3 sets. You can simply do this with a built-in function in R.

Importance of distribution P

US Army developed a model that can recognize military/civilian vehicles with pictures as feature vectors. They looked for data to train & test. They spent the day taking pictures of vehicles. They trained the classifier on the data, then tested it. And it turns out they got 100% accuracy. This is uncommon to have such great accuracy. They used it in the real world, and they got 50% accuracy, which is equal to random guessing. What happened? In fact, they took pictures of military vehicles in the morning at the military base, and the civilian ones in the afternoon. As sunlight isn't the same throughout the day, the model only made its choice depending on sunlight.

Distribution of your data must represent real-world systems otherwise all the models built on it would be useless.

Introduction to Random Forest

What is a binary tree?

Before talking about RF (Random Forest) you must know what is a binary decision tree.

They are made of 3 different types of nodes: the root which is the top node, decision node, these two always lead to 2 further nodes. And there are the leaves which don't lead to any.

To get an idea of how we use a tree I am presenting you the Binary Search algorithm. It's basically how you use a dictionary. You input your search word "ecology" in the algorithm, and you compare it to the median word of the dictionary: the middle word

when you cut the dictionary exactly in 2. Let’s say it’s "machine". "Ecology" is before machine in the alphabetical order so we go to the left. Now we just look at the dictionary subset that ends at the word machine. We look for the median word, let’s say it’s fact. Ecology is before the fact, and so on until we find our word. We progress in the tree until we hit a leaf

Really fast algorithm since it’s the way you use a dictionary instinctively. A linear search would be looking at each word in the dictionary until finding your search word

Let’s see in our case: We have a dataset with 4 feature vectors associated with their Yi. Decision nodes impose conditions on features. We go down along the tree until we hit pure leaves: subspaces that contain points that belong to one label. Let’s visualize what a pure leave is on the graph dataset. Circles and crosses represent the 2 labels. The 4 compartments are pure leaves since each contains points that belong to only one label. The algo splits the dataset until each leaf is pure. That means that each leaf in the tree (you can see on the left) refers to only one label, 0 or 1.

Predicting the label of a point means looking at which leaf the test point is and assigning the environment label.

Decision trees look good, but in fact, we call it a weak learner, which means it’s just slightly better than random guessing, around 55% of accuracy. Why? Because of overfitting. It’s very highly dependent on the decision nodes, their order, and which features are compared. Each tree is built as having 0 in training error because it learned the data but the test error is high, the definition of overfitting we saw before.

In the end, a tree is just more right than doing 50/50. And this property is the general idea behind Random Forests.

Basically, the idea is: as each tree is slightly right, running thousands of trees and taking the average is way more reliable

Random Forest

Let’s say the set of labels is 0/1, it is easily extendable to a multi-class problem.

We want to run thousands of trees. The difficulty is we need as many datasets as we have trees, otherwise, we would have thousands of duplicate trees. The trick is, from our initial dataset, we draw m new datasets of the same size with replacement. It allows us to have the amount of data we want while respecting the distribution. The idea is to

get more data while keeping the same distribution P behind the initial dataset D → this trick is called bootstrapping.

The columns are in fact the new datasets, I couldn't represent the features and labels. Since we drew with replacement, we can have the same feature vector several times in a dataset.

For each of these datasets, we select a fixed number of features. Here we select 2. We learn an overfit decision tree (built the way I told you in the previous part), which means training error equals 0. These features are the ones compared in their respective trees.

Then we want to test a point. For the RF, it means running the whole forest on it. This is not a big deal since each tree is so fast to compute.

Let's see what is the predicted label on the most left tree: x2 is above 4.9 (x2 = 6.2) so we go to the right, we hit a 1 label leaf → predicted label: 1

We get the labels of each tree and we return the one in majority. Here we have 2 times 1 and just one 0, so we assign 1 as the label of X5.

It seems maybe magical, why we would have a great accuracy?

Random Forest: Why does it work?

It works because of the weak law of large numbers, it is the basic intuition of probabilities. It states that if you repeat an experiment independently a large number of times and average the result, what you obtain should be close to the expected value (espérance). Take the example of a coin toss: the more you repeat the experience of coin tossing, the more likely you will have 50/50 in heads/tails. It just says that as n grows, which is the number of trees, our prediction tends towards the expected value of our prediction.

We have n trees, some are predictive, some don't. Some are just doing random predictions because they are based on unpredictive features. By the weak law of large numbers, the expected value of the sum of the unpredictive trees prediction is to have 50/50 in 0/1 (same as having 50/50 in heads/tails in coin tossing). They are doing errors that are so different that they average out in the end. And only the predictive trees play a role in the final prediction. Here X4 is a predictive feature, you are more likely to get 1 as a prediction if you selected X4 for your tree. Then when you do the average prediction of the whole forest, the predictive and unpredictive trees, we would get 1.

Strength of Random Forest

1) Curse of Dimensionality

The main obstacle in ML applied to biology is that algorithms have to deal with very high dimensional datasets, which is restrictive.

Imagine the unit hypercube (dotted cube in the top right).

Hypercube means a d-dimensional object. It means that if d=1 it's a line, d=2 it's a square, and d=3 it's a cube.

All training data is sampled uniformly within this hypercube: it means there is no cluster of points anywhere. We construct an other smaller hypercube with l as its edge length as it contains the k nearest neighbors of the test black point. k is a parameter and we choose k = 10 here. Here are the 10 nearest neighbors of the black point, then we build the smaller hypercube that contains them. We call this edge length l.

Animation on the right: blue points are uniformly distributed. We put a random test point, we spot its 10 nearest neighbors (10 lower distances according to the Euclidian distance) we put in red. Then we construct the l hypercube in yellow as it's the smaller hypercube that contains the 10 nearest neighbors of the test point;

Now let's get into it with d=1. the 2 hypercubes are lines, and l has an arbitrary size. As we are using a uniform distribution, a point as much as the chance to be found anywhere in the interval. And as our interval is scaled to 1, we can think in terms of probabilities. We are looking at what is the probability that a point is within the l hypercube, which means landing in the yellow section. It is actually the size of the yellows section, which is l. The probability of finding a point in the l hypercube is l.

Let's see what is happening in 2d. To be in the l hypercube, a point has to land in the 2 yellow sections which is much harder. Mathematically, the probability of being in this square is l times l.

In 3d, it's the same thing you have to land in the yellow part one additional time, which is way harder. You can spot the pattern now that the probability that a point is in the square is given by l^d with d as the dimension.

You know that a number below 1 to the power of something growing tends to 0, and this is really fast. We can translate this theoretically as "In an infinite dimensional space, the probability of finding a point within the l hypercube is actually 0. It's impossible". Since

it’s the power that is growing, it is exponentially harder to put a point in the l hypercube as d is increasing.

We can quantify and visualize this phenomenon with the table: here we assume we have 1000 points in the unit hypercube and we build the smaller hypercube as it still contains the 10 nearest neighbors.

The values in the table are computed by this formula:

$$\ell^d \approx \frac{k}{n} \Rightarrow \ell \approx \left(\frac{k}{n}\right)^{\frac{1}{d}}$$

See k/n as the proportion of points within the l hypercube. As l^d decreases exponentially as d grows, l must increases exponentially to still be equal to k/n (since k/n doesn’t vary). We compute the d-dimensional root on the left equation to get the right one.

According to the table, when d=10, the size of the l hypercube has to be 63% of the unit hypercube to keep the property of containing the 10 nearest neighbors of the test point. For d= 100, the smaller hypercube has almost the size of the unit hypercube. Imagine that in this hypercube there are only ten points, and the 990 others are right on the edge. It’s so difficult to catch a point within the l hypercube that it has to be as big as the unit hypercube. Every point is so far away from each other.

Distribution of distances: these parabolas represent the distribution of distances between every point of the unit cube and we measure their frequency in %. As d increases, all distances concentrate in a tiny range: every point is far away as we saw before, but there are uniformly far away: all the points are equidistant! The distance no longer means smth in high dimension. Algorithms like Nearest Neighbors, Perceptron no longer hold in high dimensional data.

How do we overcome this curse? By simply not computing any distance. By chance, RF does not (In fact it can become an issue if we don’t cap the depth of the decision trees).

2) Easy to use

Here I show the code I wrote to analyze the dataset of Sophie. I want you to understand that running RF is so easy and you should definitely try She sampled microbial populations from cultivars of apple trees, and I am trying to predict which month each sample has been taken. It's of course not a useful case but it's for the demo. We are in a multiclass classifier situation since the label here is time and contains these 3 months.

I put this image just to show you that it's easy to implement, we separate the data into training/testing sets in a few lines, and we execute the algo. It has only 2 main parameters (in fact you can tune it deeper but that is way more shady). It appears we don't have to do any kind of optimization, the best mtry value (number of selected features) is always the ceiling of the square of the dimension of the dataset. Don't ask questions about why is it this way. And the greater the nb of trees is, the better will be the classifier (because the weak law of large numbers is increasingly powerful as the number of experiments grows). This means you don't have to try every value of parameters.

If you remember what I wrote on splitting set, we used the validation set to kill models that used parameters that made them overfitted. As we don't have to select them, we don't need a validation set.

We run the algo on the training set and we get this confusion matrix. A confusion matrix is a way to know how many errors the algo did here on the training set and which one. The rows are our prediction, the columns are the actual values. Among all the samples of the training set, we predicted that 37 samples were taken in August (row1). It appears they were effectively taken in August (column1). Diagonal values are good predictions. We have 0% of training error. You can calculate the accuracy by doing the sum of the values of the diagonal on the sum of all the numbers in the matrix.

As you remember, we are not interested in the training error, but in the test or generalized error. This is what we are doing here. And we have still 0% of error. This means that we get 100% correct on external data. Obviously, it works great since we have "date" as a feature.

Test results: We can analyze our results in several ways. We can know the prediction and the label of each sample and on the right, we can observe the evolution of the training error along the number of trees. We see we got perfect accuracy by only taking tens of trees it's very quick.

3) Highlight the predictive features

The main reason why I showed you the Random Forest algorithm is that you can extract the predictive features. And this very easily with these 2 functions:

- importance: give a kind of predictive score for each label
- varimp: graphical version

MeanDecrease accuracy is the Measure of the accuracy of a RF model by excluding each variable. To give an understandable translation, it means that if we remove the date feature, we would misclassify around 167 additional observations since MeanDecreaseAcuracy(date) = 167.54

4) Easily interpretable

Let's say you want to predict if someone has colorectal cancer given his microbiome. The set of labels is {Cancer sufferer,Healthy}. Let's say you are using Neural Networks, which is another algorithm that is not interpretable at all, It's kind of a black box. you take a sample from a patient and you get a result. That's it. The model might be great, maybe it doesn't get wrong. But you can't know what is happening right there. In research finding the best model is not your goal, it's to spot the underlying patterns. Neural Networks have no explanatory power.

On the contrary, Random Forest is highly interpretable since we know exactly how it works. And we get further interpretation with its ability to spot predictive features.

I hope I made you understand clearly why this algorithm is powerful and why you should consider it. But it's important to not overinterpret the kind of results you would get from ML methods.

IV) What can we deduce from ML?

I use a 2nd time the example of colorectal cancer. We have a dataset that contains the ASV found in the microbiome of patients suffering from cancer or not. We find with our model the most predictive features, which are bacteria here, using the taxonomy table. I oversimplified the process but let's say it works this way, this is not the point I want you to focus on. I took 3 random names of bacteria just to acknowledge that they are the most predictive features.

Can we say these bacterias are the cause of the cancer? Absolutely not! because ML is just about predictions, seeking underlying correlations. One of these bacteria can be characteristic of a healthy person, who knows? Since we only track the predictive

power. But even if we prove in another way that bacteria is always found in cancer sufferers, we can’t conclude this is a cause. Let’s see why.

Does ML make classical statitics useless?

To understand it, watch these implications. Heat increases the consumption of ice cream. It appears as well that it increases the murder rate. But the increase in ice cream consumption is obviously not the cause of an increasing rate of murder, but they are a bit correlated since they have at least one same cause. This said ice cream consumption is a good feature to predict murder rate in a machine learning model, even though none is a cause of the other.

There could be many unknown causes but they remain hidden in your results. You need classical statistics and especially causal inference to analyze the nature of all these correlations.

What can we deduce from ML? (continued)

To come back to our colorectal cancer example, we can only deduce that these bacterias have a strong predictive power in recognizing cancer sufferers, that’s it.

You can absolutely run these models to just have an idea of bacteria implied in what you are studying. It’s a real help in finding future hypotheses, and research paths to dig.