Machine Learning for Parties

If you are at a party and people start talking “Data Science”, “Machine Learning”, “AI” (as it always happens) this is what you need to know to roam confidently in that environment! The alternative is to learn a couple of decent magic tricks. But hey, if nerdy is your thing, here it is.

The Basics

The best succinct definition of machine learning (ML) I’ve come across is:

“Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules”
- Aurelien Geron.

That’s the general idea. Now let’s talk specifics.

Most of machine learning consists of building models. Models can be a simplified representation of the world, an abstraction of a system, or even an analogy of a process. Generally speaking, models take into consideration a limited set of variables to represent a real-world phenomenon. For instance, we create weather models that use variables such as today’s humidity and wind speed to predict what the temperature will be tomorrow.

Machine learning models are unique in that they learn from data, in order to become better at a given task (e.g. predicting tomorrow’s temperature). Learning doesn’t happen magically though. Instead, we code probability, statistical and mathematical principles into a computer so that, on its own, it can then learn how to improve at performing a task. Machine learning models therefore exist at the intersection between computer programming and statistical learning. A weather model, for instance, would use math and statistics to find how to combine values of humidity and wind speed so as to predict temperature. In broad terms, the process entails three main steps:

- Getting data
- Training a model to do some useful transformation to the data, and
- Producing an outcome from the model that helps us perform the task at hand

Let’s illustrate with an example.

Note: *as you go through this section, you will find many instances where I present an example of a problem that we want to solve, and ask something like “How could we solve this issue?” And then proceed to say “Well, one approach would be to…” as if that approach was in any way an obvious solution. They are not obvious. The approaches and algorithms used in machine learning are the result of years (sometimes decades) of research. Just know that.*

A Classification Problem

Imagine you hate spam like dinosaurs hate meteorites. You want to detect spam emails and remove them from your inbox. One approach to detect those emails coming from a Nigerian prince who offers fortunes of money would be to manually go through each email and delete them as

you open and confirm they are indeed spam. That approach takes time and is risky, as you might accidentally click on a malicious link in the email that steals your information.

A better approach would be to program your email server (e.g. gmail, outlook, yahoo) to identify spam email automatically--this would save time! To do so, you could hard-code rules that flag an email as spam. For example, maybe you have noticed that most spam emails come from a very long and intricate email address. Or maybe any email that has a dollar sign ($) on its title is spam.

What else could indicate spam? The best way to answer this question would be to look at a bunch of spam emails, as well as a bunch of normal emails and compare patterns you observe on each. If you see something consistently showing up on the spam emails but not so much on normal emails (like "$" in the title) you can hard-code that as a rule. You then write some code into your email server that says something like: *IF the sender's email address exceeds 30 characters OR the email title contains "$", THEN flag email as "spam".*

The approach above is better than manually flagging each email; it's faster and safer. But there are several issues with it. If you hard-code rules, whoever is sending you spam could at one point figure out what those rules are and would then be able to change her emails to fool your spam detector. Think that spam emails 10 years ago looked different from spam emails today, because the agents producing spam have adapted and became smarter. If we hard-code rules, then we would also need to update them regularly, which would be cumbersome and, again, time consuming. Another issue is that you may need to come up with hundreds of rules to correctly identify email, which would make hard-coding a very time consuming and intricate process.

Enough preamble already! Is there an efficient and effective way of dealing with spam? Yes indeed. Its name is learning, machine learning.

ML allows us to create a model that makes a machine (our email server in this case) get better at some task (flagging spam email) by learning from data (examples of spam and normal emails). Instead of having to explicitly code rules, the model learns rules on its own! Our machine learning model takes data in the form of an email, it does some mathematical transformations to it, and then it spits out "spam" or "normal". The best part is: if the data changes, the model adapts to it.

We will look into what the aforementioned math transformations are later on, but going back to the 3-step process we outlined above to perform a task, for the case of spam email detection it would look something like this:

- Getting data - collect lots of spam and regular emails
- Train a model - extract useful information from the emails, convert such information into a mathematical representations (because computers like to deal with numbers) and train a model that can differentiate spam emails from regular ones
- Produce an outcome - give a new email to the model, and have it predict if it is spam

Now that we know the big-picture steps, how do we magically train these models to classify email? Also, what if we wanted to predict something else other than if an email is spam or not? To answer those questions, we turn to **Regression.**

Regression

Flagging spam email is a **Classification** type of problem: you are classifying each email as spam or normal. Whenever you have distinct classes that a variable can take on, you are doing classification (e.g. labeling pictures of animals, predicting which team will win a soccer match). But what if you wanted to create a model that predicts how much electricity will be consumed by a city on a given day? You could create consumption categories like “little”, “medium” and “high”, and predict one of the three for a given day. However, it would be better if we could get a real number (say in megawatts, MW). In that case, we are confronted with a regression problem in which we want to predict a numeric value. In other words, we are dealing with continuous data (i.e. variables that can be measured at decimal levels).

Historical Context:

The word *regression* comes from *regress,* which is defined as the tendency to approach or revert to a mean. It was coined by Francis Galton in the 19th century to refer to the tendency of fathers’ sons to regress (or revert) to the average height in the population.

If the male population height is 1.65 meters tall, or 5’4 31/32” in English units… How in the world are English units still a thing! U.S., Liberia and Myanmar, stop the madness. Dear reader, I promise that’s the last case of silly units you will see in this book. So, if the average man is 165 cm tall, and a father is 180 cm tall, Galton observed that his son would be closer to the mean than him. The son could be 175 cm tall, for instance. If the father is 160, the son might be 162. He generalized this via a linear approximation, meaning, a linear model which we now call *linear regression.*

Back to the classification vs regression example. Similarly, to the email classification problem, we can get a bunch of data from many days when we know what the electricity consumption was and create rules to predict the electricity for the following day. What would those rules look like? First, using common sense we can conclude that the following might be good predictors: temperature, whether it’s a weekend or a weekday, population size, and number of hours with sunlight, among others. We could create a model that says the daily electricity consumption is a linear function[1] as follows:

$$E = 10 * avg\_temp + 0.5 * sun\_hour + 100 * population$$

Which reads: *daily electricity is equal to the average temperature times 10, plus the hours of sun exposure times 0.5, plus the population size times 100.* That returns a real number. Let’s say it’s January 5 of 2019 and we use that equation to predict 15 MW will be consumed in San Francisco on January 7. We basically took values of variables we can measure or safely estimate—such as temperature and population size two days from now—to predict a variable we are not able to measure directly—like electricity consumption two days from now. The best model would be that which gets the predicted value as close as possible to the actual values.

In the above example the *rules* are the selection of which variables to include, like temperature and population size, as well as the actual values that we multiplied our variables by: 10, 0.5 and 100, which are called **Coefficients**. To avoid having to do trial and error to determine which rules

best predict electricity consumption, we create a model that creates those rules by finding coefficients minimizing the discrepancy between predicted values and actual ones.

To get our model to learn these rules, we need to gather observations where we know what the outcome is. In this case we are predicting electricity, so say we gather two years worth of observations where we measured the variables mentioned above and then our response variable (electricity). We create a model that tries different rules and ends up selecting the best set of rules, meaning it selects which variables to keep in our model, and what weight (a.k.a. coefficients) to assign to each (again, that's the 10, 0.5 and 100 values above). All of this with the goal of getting predictions that most closely approximate the actual values.

In this way, ML uses algorithms to automate the process of finding rules that allow our machines to perform a task, like classifying emails or predicting electricity consumption.

The two examples above are part of a big subset of problems called **Supervised Learning**. It's called that because we are using observations where we know the response variable (electricity in the example above) to train our models. Meaning, the model *supervises* how well the model is doing by comparing predicted vs actual values.

There is also **Unsupervised Learning**. If we didn't know what the electricity consumption was for any of the observations in our two years worth of data, we would not be able to supervise our model's performance during training and we would be dealing with an *unsupervised* problem. Usually the goal with unsupervised learning is to infer a structure or patterns in the data, as opposed to predicting something. But more on that later.

Now, the two examples of tasks described so far (classifying emails and predicting electricity consumption) are just two out of thousands of applications. Let's look at several others to understand the reach of ML.

Facebook Photo Tagging

Say your name is Mr. Facebook and one of your favorite things to do is to tag faces on photos. How do you do that? You could ask users to do that themselves, but ain't nobody got time for that. A better alternative is to train a system that automatically does that. But how?

First, you gather *a lot* of photos that are already tagged. Second, you convert that photo into a format our computer models can understand: numbers. In particular, you represent each pixel as a number. For simplicity, let's say we represent each pixel as a value from 0 to 10, with 0 being white and 10 being black (everything in between is a darker or lighter gray). In this case we will assume our images are 1000 x 1000 pixels (it's a square image with a total of 1,000,000 pixels). That is what we feed into our model: a matrix containing 1000 rows and 1000 columns, each with a value from 0 to 10. This is a classification problem where we want our system to classify faces from a given set of options (in this case, Facebook users). That all sounds very magical, so let's break it down and talk about the mathematical transformations mentioned earlier.

Forget for a second about the image classification example and let's talk about building a model that classifies someone as tall or short. Our input will be a table with three columns: name, height and label. Each row is therefore a person's name, its height and a short/tall label (e.g. Peter, 181 cm, Tall | Anna, 156 cm, Short | Shaq, 216 cm, Tall). Before feeding this into the model, we do what? Convert data into the language of computers. So we convert "Tall" and "Short" to numeric representations. Let's say *Tall* is converted to 1 and *Short* is converted to 0. If we feed our model a table with 1000 entries and it so happens that everyone at or above 178cm is labeled *Tall*, and everyone below 178 is labeled Short. We then train a model that given height as an input, returns 1 if it is greater than or equal to 178 cm or 0 if height is less than 178. Using simple

code we can then convert “1” and “0” to “Tall” and “Short”, respectively. Just like that, our model can successfully predict tall vs short. Again, note that these models operate exclusively with numbers: numbers in and numbers out.

To a certain extent, *tall* and *short* are subjective terms though. So what if we saw examples below 178 cm that were labeled as *tall*. Say that we had different people label those entries, and we found some cases where 175 cm is Tall and others were 175 cm is Short. Same for 186 cm, or any other number. In that case the cutoff would not be so clear. That’s not too big of an issue though. Most of the models used in ML applications **operate with probabilities** more than with clear-cut rules. So, if we asked our model to label someone who is 197 cm, it might output 0.91, which is a value much closer to 1 than to 0 (remember that we chose 0 to represent *short*, and 1 to be *Tall*), and therefore we label it as *Tall*. If we tried to label someone who is 180 cm we might see that our model outputs 0.58, which means it’s closer to 1 than it is to 0, but not for much. That means that our model is not as certain that this person is *Tall*!

This brings up an important consideration that is common in the field: often, the consequences of mistakenly choosing one label are greater than those of mistakenly choosing the other label. In this example, it is more common for people to feel offended if they are called short, than if they are called tall*,* so it is preferable to mistakenly label someone as tall. To prevent our model from getting us in trouble we decide that it will only classify someone as *Short* if it is highly confident that he is short. In model language that translates to saying that only probability values close to 0 will be labeled *Short*. How close is close enough? We get to decide. Since we really don’t want to offend, let’s say only values at 0.20 or lower will be labeled *Short.* What we are effectively doing here is moving our threshold level, or level of tolerance. In this way, we get to make our own rule to determine what the cutoff for *Short* vs *Tall* is. Let’s continue with the assumption that the model is 50% certain when the height is 178 cm, and also assume that the model considers someone measuring 169 cm to have a probability of 30% of being tall. Since that 30% (0.30) is higher than our chosen cutoff of 0.20, then it would classify that person as *Tall.* Not a bad day for short people.

The main points of the simple example above are: first, models work with numerical structures, and develop mathematical rules to produce a numeric output. They don’t really understand what tall and short means, that’s for us humans to interpret. Second, most models provide a likelihood and an associated level of confidence, which means they are not guaranteed to provide the right outcome. The quality of the result depends on the data at hand and how well our model can use it to perform a given task. And third, human judgement is necessary and important at many steps in the process of building models. Now, back to Facebook and tagging faces.

We said earlier that we feed a matrix of 1000 x 1000 to our face-tagging model, but to simplify things a little, let’s instead say we flatten the image so that we take the second row of pixels and add it at the end of the first, then we add the third. So on and so forth until we have our matrix converted into a single line with 1,000,000 values from 0 to 10 representing the pixels. Let’s also say that there are only two options for tagging a face: you or your friend who has a thick unibrow.

Columns 1 through 1,000,000 will be pixels, and column 1,000,001 will be a 1 if the picture is of you, or a 0 if the picture is of your unibrow friend. We are assuming that all pictures have either you or your friend, but not both (this is just for explanation purposes, in practice you can of course create a model that has images with many friends and can successfully tag each of them). Let’s get to it.

- We gather 2500 pictures of you and 2500 pictures of your unibrow friend. You have so many pictures because you both want to become Instagram models.
- Convert each photo to a grey-scale and flatten the image so that we get a table with 5000 rows (one per picture) and 1,000,001 columns (1,000,000 representing pixels, and the last

column representing the label: you or unibrow friend). Notice any similarities with the Tall vs Short classification in the previous example? In both cases we created a table where each row is an observation and each column is a feature. In the case of Tall vs Short we only had one feature: height. In this case we have 1,000,000 features: individual pixels. In both cases we have one column with labels 0 or 1.

- We train a model so that when we give it a new image it can accurately output 0 if it is a picture of your friend, or 1 if it is you.

Image #	Pixel 1	Pixel 2	Pixel 3	...	Pixel 1,000,000	Label
1	0	3	2	...	1	1
2	1	4	2	...	3	0
3	3	8	10	...	6	0
4	8	6	5	...	4	0
5	4	7	4	...	10	1
6	1	7	2	...	2	0
...	...	...	...	...	...	1
5000	0	8	0	...	8	1

That all sounds good, but still a little magical. How does the model actually learn? The actual process is a little intricate, but I will illustrate the general idea by doing what an educated person should not: focusing on your friend's unibrow.

If we compared one image of your face and one of your friend's, to the computer they will look similar. They will both share patterns of darker pixels at the eyes, nostrils, mouth, and conveniently for this example, at the eyebrows. There will be a difference there though, assuming you don't have a unibrow, your friend's picture will show darker pixels (values of 8s, 9s and 10s) between where the two eyebrows are located compared to yours (values of 1s, 2s and 3s).

That pattern will be observed on all images, and it will be picked up by the model. The model will learn to create a certain rule stating that when those pixels are dark, it's more likely that the person in the photo is your friend. How does it create this rule in practice? Similarly to the electricity consumption model we discussed earlier, it will do so by giving a certain weight to those columns. It could maybe multiply those columns by a negative value, so that pictures of your friend predict a label that is closer to 0 when compared to pictures of you. In that way, the model has learned to extract knowledge from the input by doing some mathematical operation to them (i.e. multiplying each column by a weight, and summing all of the products). Those weights that the system multiplies the features by are saved, and in essence this becomes the model.

The process of determining the actual weights is even more intricate and depends on which algorithm is being used. The beautiful details of how that happens, and how to get the results to perform best, are what this book is about.

Categorizing Yelp Reviews as Positive or Negative

We looked at images, now let's look at language. Given a set of Yelp reviews for a restaurant, can we train a model that tells us whether a review is positive or negative? Yes we can.

Similarly to the pictures example above, we first need to take the crucial step of representing our inputs as a set of numbers, so the computer model can work with them. There are different options for doing so. Here we will use a simple approach, which is to use a word counter. Imagine we have the phrase "The food there sucks". We can represent this as a series of numbers by creating a vocabulary of words. In this case we will use a simple vocabulary of 10 words. It will be:

[a, chair, floor, food, in, restaurant, the, there, sucks, water].

It's a small vocabulary, but what's important is that it includes all the words in our review "The food there sucks". Our representation of this sentence will therefore be:

[0, 0, 0, 1, 0, 0, 1, 1, 1, 0]

where we have a 1 if the word in our vocabulary appears in this particular sentence, 0 otherwise. If we had another review, say "The water in the restaurant sucks" then our numerical representation would be

[0, 0, 0, 0, 1, 1, 2, 0, 1, 1]

where each number is counting how many times each word appeared in the sentence ("the" is there twice, so we have a 2 in its place). If we had a dataset with 100 reviews, we could have a vocabulary of, say 1000 words and represent each sentence in a similar way. We would have a table with 100 rows and 1000 columns. Each column would represent a word in the vocabulary, and each row would represent a review. We would also have a column at the end (column number 1001) with a positive-or-negative label: let's choose 1 for positive and 0 for negative.

The model would then train and learn weights so that it predicts values as close to the actual values as possible. For instance, it will realize that if it gives high weights to columns of positive words, like "terrific", "delicious", "amazing", the predicted value for reviews that contain those words would be higher than if they didn't have those words. Therefore, positive reviews will get a larger predicted value than negative reviews would, which is what we want because positive ones are labeled as 1 and negative ones are labeled as 0 (and 1 is of course higher than 0).

In practice, we first convert all reviews into the vocabulary numerical representations, and then multiply each by the model weights. This returns a number of arbitrary magnitude. To illustrate, let's assume that after training the model the weights for our columns in the 10-word-vocabulary example end up being:

Vocabulary:	[a	chair	floor	food	in	restaurant	the	there	sucks	water]
Converted Sentence:	[0	0	0	1	0	0	1	1	1	0]
Weights Learned:	[0.6	0.5	0.6	0.8	0.7	0.5	0.4	0.7	- 0.1	0.7]

In that case, we would get an output of 1.8 by multiplying "The food there sucks" by its weights (1*0.8 + 1*0.4 + 1*0.7 + 1*-0.1 = 1.8).

Notice three things:

1. The weight corresponding to "sucks" is negative, because the model would learn that "sucks" is usually a negative word, associated with negative reviews, and therefore would make its weight so that the end result is closer to 0 for sentence that say "sucks" (because we had labeled positive as 1 and negative as 0).
2. The resulting 1.8 is greater than 1. In fact, if the sentence was longer, it could be 17, 103, or any other number. But we want a number between 0 and 1! Meaning, we have a problem because 1.8 is closer to 1 than to 0, but we know that it should be closer to 0 because we can tell "The food there sucks" is a negative review. Is this an error? No. In practice we use a function that *compresses* numbers of arbitrary magnitudes to values

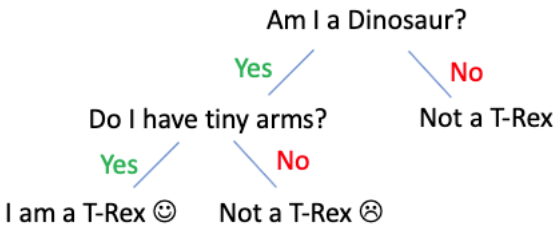
between 0 and 1. The details are technical, but know that if we actually trained a model and got 1.8 for "The food there sucks" it will be transformed to something like 0.2 once we passed it through this shrinking function. Since 0.2 is closer to 0 than 1, we accurately label this review as negative.

3. The image and review classification problems just described illustrate how we go from input to output. That's ML at its core: without explicitly coding any rules, we create models that become better at a task (in these cases classification) by coming up with weights (a.k.a. learning) that transform input data into an output useful to us.

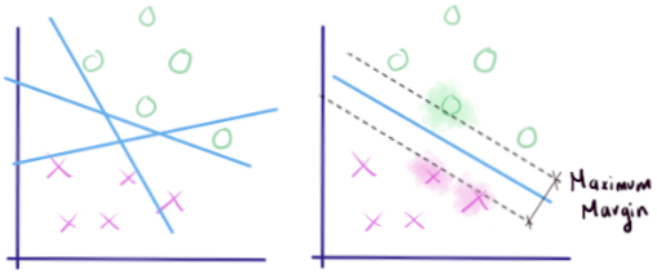
More Algorithms

The way ML models learn is via algorithms. An algorithm is a set of ordered steps that are followed to solve a particular problem. They are the engine that allows models to learn. There are *many* algorithms and models used in ML. Some of them learn by coming up with weights, just like in our examples above, but some operate differently. We will cover the most popular ones throughout the book, but here is a preview:

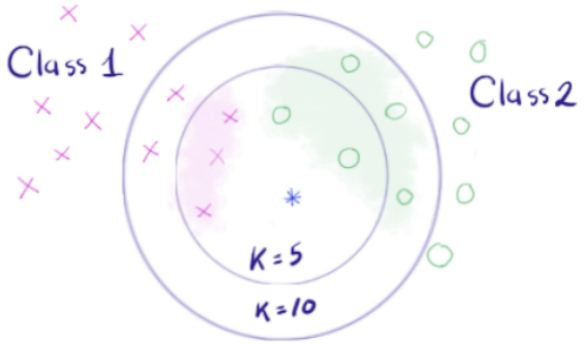
- **Decision Trees -** they use a tree-like structure to classify or predict values. Once the decision tree structure is created, which the model does by looking at labeled observations, we can feed a new unlabeled observation and see what the model will output. In the simple example below the options are: T-Rex, or Not T-Rex. Instead of weights, we have thresholds that define whether we go one way or another at each node. Decision trees can also be used to solve regression problems (when we have continuous values instead of categories).



- **Support Vector Machines (SVM) -** Allow us to classify observations by learning the dividing line that maximizes the space between itself and observations on either side. Once the model learns such line, one can classify observations by seeing on which side of the line they lie. The same concept can be applied for regression, but it's more often used for classification and it's easier to visualize that way.



- **K-Nearest Neighbor (KNN) -** The KNN algorithm is primarily used for classification, although it can be applied in regression as well. It assigns a new observation to the category class that most of the k-nearest observations belong to. In the image below we see that if we choose *k* to be 5, then our new blue-asterisk observation will be classified as belonging to Class 1 (pink). But, if we extend k to include the 10 closest observations then we would classify it as belonging to Class 2 (green). This may seem like it's a very arbitrary approach but in practice, and following some basic guidelines, it can provide good results. Contrary to the rest of the algorithms we have seen so far, this one is not model-based. Meaning, we don't need to train a model with observations in order to classify new ones. You only need to find the closest ones and assign observations to the majority class in that space.



Beyond Classification and Regression

Although classification and regression have historically been the most commonly encountered ML applications (i.e. tasks to perform), there are many others. Some of them are becoming increasingly more relevant with our newly-found ability to capture more and more data. Here are some of those applications:

Principal Component Analysis (PCA)

Often used as a pre-processing step to reduce the dimension of our dataset. If, for example we have 10 features that we can use to predict the weather (e.g. temperature, humidity, etc) but wanted to be able to visualize them, we could either find a way to plot 10 variables together (hint: it can't be done, at least not in a way that makes sense to humans because we visualize everything in our 3D perception of the world); or we could reduce the dimension of the dataset by combining features in a way that captures most of their information in less dimensions. That's what PCA allows us to do: represent many variables in fewer dimensions by combining them in some way, without losing much information. PCA is an *unsupervised* learning process (when we don't have labels to "supervise" the model while learning).

Clustering

It consists of a broad class of methods used for discovering subgroups in the data. It basically clusters data points together so that we can learn grouping patterns in it. Clustering is an unsupervised learning process.

Recommender Systems

To put it simply, these are the systems that allows Netflix to know which shows to recommend to you. They learn similarities between items (e.g. movies) and provide recommendations based on those similarities. Alternatively, they can also provide recommendation based on users' attitudes towards items. To explain the specifics, assume I watched the movies *Titanic, Hercules* and *Transformers* and rated them as 4, 5 and 2 on a scale from 1 to 5. What movie could Netflix recommend to me next? Well, if another user called Binger has watched the same 3 movies and gave a similar score to them (say 5, 5 and 2), then it might mean we have a similar taste in movies. If it just so happens that Binger also watched *Interstellar* and rated it with a 4, it would make sense to recommend that movie to me. The way the model would learn this is by "plotting" my reviews of *Titanic*, *Hercules* and *Transformers* on a 3D graph and noticing that Binger's reviews of the same movies lie close to my point in the graph. This high level of similarity would then indicate that I might like what he likes. Note: the model does not really *plot* reviews on a 3D graph, but instead finds proximity using vectors, matrices and other fun math stuff we will cover extensively later on.

Natural Language Processing

This is how Siri and Alexa are spying on you. Systems that perform tasks related to analyzing, interpreting or generating language structures. These can be written or spoken languages. Some examples of applications are:

- Machine Translation - Translate content from one language to another
- Automatic Summarization - Models that take text documents and create a (hopefully) coherent summary.
- Sentiment Analysis - One example is the Yelp reviews categorization we explained above.
- Voice Recognition - Taking sounds from spoken language and interpreting it.

Time Series Analysis

Time series are sequences of data points taken at equally spaced points in time. They are different to many other datasets in that (1) observations are correlated between them, meaning they are not independent (think of how the weather today is likely to be similar to the weather tomorrow); and (2) most time series have a seasonality trend, meaning the data contains variations at specific time frames (think of how although the weather fluctuates throughout the year, we know that summer months are hotter than winter months every year, so we observe a seasonal pattern, in this case yearly).

Reinforcement Learning

Think "robots". Reinforcement learning systems are the series of models that allow an agent (e.g. robot) to make observations and take actions within an environment. An example is self-driving vehicles. The agent (car) makes observations (measures distances, recognizes objects, etc., via sensors) to take actions (accelerate, break, turn the wheel) within an environment (roads).These agents learn by trial and error, where basically every time they do something good they get rewarded, and with every mistake they get punished. Rewards and punishments are merely representations in the code, as if it was a computer game and we programmed the car to subtract 10 points when we hit a curb, subtract 100 when we hit a dog, or add 1 point for every minute it keeps moving within the road and without hitting anything. The agent will then act to optimize its reward (i.e. get as high of a score as possible) and in that way avoid hitting dogs and learn to stay on the road.

Anomaly Detection
Models that can recognize observations that are outside of the norm. Mostly used in fraud detection, manufacturing and data centers monitoring. Anomaly detection is similar in some way to the classification problems we saw earlier, but in this case one of the categories is way more prevalent than the other. We classify an observation as an anomaly by computing a probability that it fall outside of the distribution that the rest belong to. If the probability is high enough, it is considered an anomaly.

Generative Deep Learning
These systems go beyond analyzing and learning from data, and are instead capable of generating synthetic new data that resembles the actual real observations. Some of its uses are in the realm of language models, where Generative models have been used to create text. Currently, a large number of applications of Generative models is in the development of images. Be it for artistic purposes (look up Deep Dream) or to increment the size of your training datasets by creating images that are slightly different to the original ones, this area has developed to the point that we can now generate synthetic images of human faces that are indistinguishable from actual photos of faces.

All of the applications above have different purposes and rely on different processes. If there are so many different types of systems and such a vast range of applications, why did we spend so much time talking about the specific cases when we create rules by assigning weights to features? That's not the way in which (to mention a few) Recommender Systems, Reinforced Learning, or Generative Systems work!

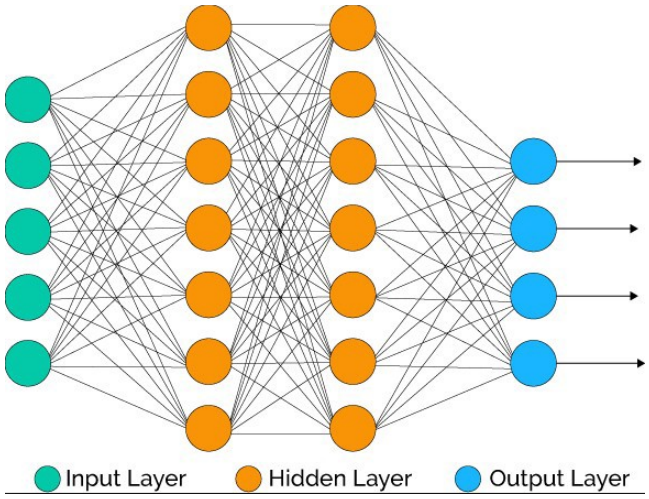
The reason we emphasized the specific cases when we use weights to transform an input into an output is because (1) the core concepts behind those algorithms expand to most computer learning methods, and (2) because of two terms you may have heard before: "Neural Networks" and "Deep Learning". What are they and why should you care?

Artificial Neural Networks

Before going into what they are, let's talk about why they are relevant. Artificial Neural Networks (ANN) have recently become one of the most powerful tools to train computers, for a couple of main reasons: (1) we have much powerful computers than we used to; and (2) we have way more data at our disposal than previously. Therefore, ANN are now able to learn from data and solve problems that we could not in the past. With the amount of research being put into them currently, they are only expected to keep improving. We won't have super smart robots by 2050, but we are making progress.

ANN are a simplified abstraction of how our brains work. They consist of neurons (illustrated below as circles) and connections between them (lines). Data is entered into the network at the input layer (green circles) and it undergoes a series of operations by passing through the network via the hidden layers (orange circles), to eventually reach the output layer (blue circles) and produce an output (arrows). The number of neurons at each layer, as well as the number of layers

are determined on a case by case basis. When we have more than one hidden layer we are doing **Deep Learning**, whereas only one hidden layer is often referred to as Shallow Learning.



Similarly to our classification and regression examples, ANN allow machines to improve at performing a specific task by learning from data. The way in which these networks improve at a task is by tweaking their parameters, which are nothing more and nothing less than weights. This is why we focused on weights so much in the first parts of this section. Let's break down what neural networks do exactly.

To explain neural networks, I need to introduce my second favorite definition of ML:

"Machine Learning is, technically: searching for useful representations of some input data, within a predefined space of possibilities, using guidance from a feedback signal"
- Francois Chollet

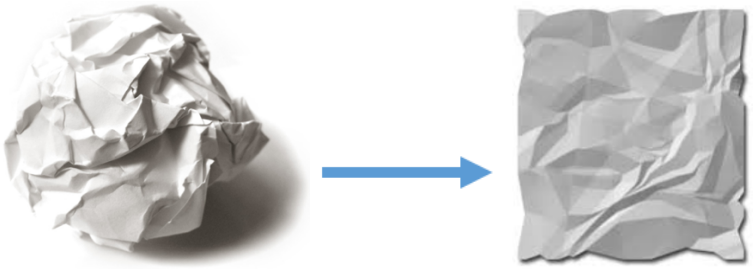
That's a very accurate explanation, but also a little technical, so let's use a more visual and geometric interpretation to understand it in the context of neural networks. Imagine you have two pieces of paper crumpled together, one is white and the other is green. Think of how your fingers would need to move to separate the two papers. First move this corner here, then pull this other corner there. So on and so forth. At the end, you are able to separate the two sheets of paper. If you can think of the paper and your hands existing on a 3D graph, you would essentially be moving an edge from, say coordinate (2, 0.5, -8) to (5, 3, 19), and repeating this process for different parts of the paper located at different coordinates.

A neural network does the same thing. You provide input data (crumpled paper) and would like to be able to transform this input data in such a way that you can cleanly separate two classes (white and green papers). Remember, the input data is simply a series of numbers representing something else. Each feature is a dimension. For instance, if we were trying to classify between two breeds of dogs we could use three features (to keep it consistent with the three dimensions that papers and hands exist in) such as: height, hair color, and ears shape.

With ANN, you are basically searching for a different and useful representation of those features. Representing the crumpled papers as two separate sheets so you can clearly identify them means doing a series of transformations to them. With paper, the way to that useful representation is via finger movements, with neural network it is via mathematical operations. For

instance, converting numbers 3, 4, 5 into 6, 8, 10 took 3 odd numbers, applied an operation to them, and turned them into even numbers. If you just *loved* even numbers, that mathematical operation would be of interest to you. What operation is that? It was simply multiplying by two.

Similarly, each neuron in an ANN contains a set of weights that multiply whatever input it receives and produces an output. This way, each neuron is in fact performing a transformation, just like your fingers would transform the crumpled paper.



To exemplify, let's assume we have the following input for a particular patient in a hospital:

[height = 180 cm, weight = 200 kg, age = 42 years, Glucose Level = 120 mg/dL, Gender = M]

and we would like to predict whether this person will suffer from diabetes or not. We take these 5 variables and multiply its values by weights at each neuron. Maybe the uppermost weight in the first hidden layer illustrated above multiplies the input by [0.4, 2, 10, -1, 4]. We do one operation after another, and use some aggregation function to eventually end up with a single value that will allow us to categorize as Diabetes (1) or No Diabetes (0). The following image shows this Diabetes example in action. We show 7 observations of patients that we know already whether they have diabetes or not. We transform the data by passing it through the network so that eventually we end up with the table in the middle. We then aggregate these values and end up with values that are cleanly separable (like the white and green paper). In this case the model found the "rule" that every observation with a final result of 200 or greater has diabetes (we could have also said anything greater than 300 in this case).

How do we determine what weights that transform inputs into outputs? That's where the feedback signal from our definition by Chollet comes in. Neural Networks iterate over different values for its weights to find the set of weights that result in the smallest discrepancy between predicted and actual values. In the example below the weights we found allowed us to get 100% accuracy. Meaning, it correctly predicted every one of the 7 observations. That's a pretty good thing! The way networks get to the optimal value for each weight is via an algorithm called backpropagation. But more on that later.

Patient #	*Height*	*Weight*	*Age*	*Glucose*	*Gender*	***Diabetes***		After Some Transformations						Final Result
1	180	200	42	120	1	0	...	54	4600	-29.4	240	9	...	198.2
2	150	190	31	160	0	1	...	45	4370	-21.7	320	0	...	300.5
3	170	174	35	110	1	0	...	51	4002	-24.5	220	9	...	197.54
4	160	140	25	100	0	0	...	48	3220	-17.5	200	0	...	189.9
5	183	240	39	170	1	1	...	54.9	5520	-27.3	340	9	...	332.2
6	169	150	30	80	0	0	...	50.7	3450	-21	160	0	...	132.7
7	176	229	40	168	1	1	...	52.8	5267	-28	336	9	...	316.14

Conclusion

In this way, neural networks, and most of machine learning problems are simply a set of transformations on our data to produce an output that achieves some task. Be it classifying patients as diabetic or not diabetic, or predicting the electricity consumption of a city. Models can be shaped with great flexibility and are thus capable of performing a vast array of tasks. They can tag you and your friends on photos, translate languages, detect cancerous tumors from photos, train a robot to learn how to walk from scratch, and detect fraudulent credit card behaviors.

What other problems can be solved using ML? The following and many more:

- Predicting the number of calories in a plate based off pictures of them
- Beating chess world champions
- Writing songs
- Performing medical diagnosis
- Program smart personal assistant computers
- Determining the mood of customers via facial recognition
- Interacting with chatbots that respond like humans
- Flagging inappropriate content on social media
- Identifying disinformation online
- Deciding what advertising to show to which customers
- Anticipating consumer market trends
- Conversing with a computer
- And many others…

What other applications can you think of?