Interpreting Data with Advanced Statistical Models

by Axel Sirota

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Related

Course Overview

Course Overview

Hi everyone. My name is Axel Sirota. Welcome to my course, Interpreting Data with Advanced Statistical Models. I am a machine learning engineering at ASAPP, statistician, ML fanatic, and I am very excited to present this to you. Machine learning is changing the world, from autonomous cars to intelligent homes. And at the very core of machine learning are advanced statistical models. Statistics is what made machine learning what it is, and it is what will guide us in this trail of predictions. We will build from basic statistics the pillars of supervised and unsupervised learning to help you make a difference. Our journey begins at the basics of machine learning, and it will be a deep dive from the start into how statistics is the power engine of the recommendations we get every day. We will revisit linear regression in a more general pattern, expand to multiple and polynomial features, only to continue learning about classification with logistic regression, SVMs, and Bayesian methods. Finally, we will learn intrinsic patterns of our dataset with unsupervised techniques, such as k-means and PCA. By the end of this course, you will be able to effectively understand how to create an ML application that will be able to revolutionize the problems that appear at your work. You will become the guru of those little tips and tricks that make the difference in the day to day. I hope you will join me on this journey to learn how to interpret data with the Interpreting Data with Advanced Statistical Models course, at Pluralsight.

Getting Started with Machine Learning

Getting Started with Machine Learning?

Hi, welcome to Interpreting Data with Advanced Statistical Models, at Pluralsight. I am Axel, and I am super excited to be your captain in this journey. To give you a brief summary, I am a machine learning engineer, statistician, and an absolute ML fanatic. So let's start this trip. You could ask me, but Axel, why is this module named Getting Started with Machine Learning if this course is about advanced statistical models? And you would be right, but it will come together. However, interpreting data with statistical models, we analyze the data set to see its structure and use that to compare groups or model relationships. In this course, we will take that and go further using more complex models to explain the data set, but also to generalize to unseen data. And that, in a snapshot, is machine learning. So let's go. So, what exactly is machine learning? Well, machine learning is a subset of AI that tries to use past data to learn data relationships and use that to predict new values. What? Let's say that in a nicer way. All the time what we want is to have some application that is able to perform tasks normally requiring human intelligence. Example of those could be, given a picture, is today sunny? Given an audio file, what language is it speaking? Should I buy a stock valued at $10 given this last 6-month performance? Given a lot of characteristics in different animals, how can we build a taxonomy tree? That is artificial intelligence. The catch is that we are not saying how we do those. We could have a lot of complex rules, like all fraud detection systems have right now. Machine learning is a subset of AI that uses past data and statistical models to make those systems, and history has foretold that eventually with enough data, ML outperforms these complex rules. Now we are at the great point where ML is winning in a lot of battles. So it makes sense to be an expert. Therefore, over this course, we will analyze common ML algorithms that leverage statistics to solve complex problems. However, if we think about the various examples and how we can make an algorithm learn to predict those results, in the case of temperature and images, we will have a set of images with different weather and its corresponding temperature. We call this expected output for each data point a label. But in the case of animals, we don't have labels, we just have features and we want to know the intrinsic structure of the data. In the case where we have label data, we talk of supervised learning. In the case where we don't have those, we talk of unsupervised learning. Let's dive deeper into each of those.

Supervised Learning: When We Learn from History

Sometimes, however, we just cannot afford to have those true values. In this case, we talk of unsupervised learning. As we have seen, having those true values, for example, in our data set, leads to well-defined programs that have multiple ways of attacking them. Why isn't that all? Well, having that is called to have annotated data, but annotation a lot of times is expensive, difficult to standardize, difficult to scale, sensitive to design changes, and hard to do right actually. A consequence of this is that almost 0.1 % of data is actually annotated or labeled. Examples of this are chats, audios, or medical histories that most of them are not annotated. So what can we do? Let's look at an example for reference. Let's imagine that as part of an ecological program, we want to understand better Aedes Aegypti distribution based on lagoons properties. In that period we measure the top 52 lagoons of Buenos Aires, Argentina so we can actually group lagoons by their similarity and be more likely to have Aedes Aegypti. Just as reference, Aedes Aegypti is the mosquito of dengue. This has an obvious impact in our efforts of trying to fight back dengue fever in South America. How can we do this? Of course, as always, we would start by taking some measurements that would make sense for the problem. This is where the biologist come in handy. We would end up with a big matrix of 52 times M where M is the number of features we took. One idea would be that based on previous knowledge, we know that there are 3 types of common lagoon types, so we apply some rule to end up with 3 groups of lagoons from the original 52. But how do we know if this is correct? Well, one way of thinking this is groups are correct if we have some measurement of how close the group is. These groups would be okay if this measurement is maximal. That would mean that the groups are actually cohesive. For example, in the case at the left, this measurement wouldn't be as high as the one in the right, because in the left the groups aren't so cohesive. There are some blue dots that appear to be part of the red group actually. In unsupervised learning, as we don't have labels, some things change. We can have a model as the grouping rule, but the notion of cost function needs to change since we don't have true outputs to compare. However, we can have intrinsic cost functions that depend on the data itself, and from there we can train our model. In our example, we had the measurement of how cohesive the groups were. Then what we will find is actually the internal structure of the data set, since our cost function is intrinsic, and that is the key difference. When we try to understand our data set without a given set of layers to optimize to, we go towards unsupervised learning. Another example would be image compression. In this case, we have an image in one format, like jpeg, and we want to compress it to send it via text. The flow would be, we have an image in the original format, then we have some magical rule to reduce the size of the image, and then we have the image in the new format that maintains the proportions. Usually as images are represented as matrices of N.M where N and M are defined by the resolution, also if our picture is colored, usually we will have on each cell three numbers, they are red, green, and blue, or RGB scale. So we end up with a matrix of size, supposing that our image is 12 x 18 x 720, then it would be 1280 x 720 x 3, almost 3 million numbers. That is a lot. The magical rule usually is a way of taking out the redundancy of this picture, such that we can represent this image in, let's say, 300 x 120 matrix, but with a way of transforming back to the original image. So we can keep the resolution, but with a new format. This is how jpeg to png works, for example. Going back, we have three main problems in unsupervised learning, clustering, where we will try to find groups out of our data set like the lagoons; dimensionality reduction, where we will try to reduce the size of our problem to make it more retractable, like an image compression; and anomaly detection, where we try to find if some value is odd, for example, in fraud detection where we want to know if a credit card transaction was fraudulent. On module 8, we will cover unsupervised learning and give hints on ways forward.

Unsupervised Learning: I Don’t Wanna Miss a Thing!

Over this video, we will talk about the most common machine learning problem, supervised learning. For getting context, let's go to a sample scenario. Let's imagine we are entrepreneurs in the medical industry that try to predict the advancement of diabetes by a single blood extraction. If we think how we could do this, it will go like this. Single blood extraction, then we take some measurements, then some complicated magic happens, and we get the advancement of diabetes. Let's imagine a little bit how this would go for humans, after all, we cannot do more than try to reproduce what humans do and use it to get where humans find it difficult. At first we know nothing, so we are given some blood measurements and we will guess how advanced of the disease. And how do we know if we are close? Well, we will have to have next to each set of measurements, the true value. Eventually we will start to see some patterns that help us get better and better aligned with what the true value is. Nobody knows really how this works on a biological level, but we are pretty sure that is how it goes. This pattern emerges a lot in other points, for example, when understanding if some day is sunny, first as children we guess, but later we start to check how many clouds there are, if there is wind, and if we can see the sunshine. Eventually at middle school, we also said that if we see the sun shining and not so many clouds, then it is sunny when the not so many clouds actually is interesting because it's some complicated rule that we cannot really explain how we can say it. What we do when we have a set of values and the true output and use that to get this rule of predicting the output of some other value, is called supervised learning. Why? Because we think of it as we are supervising the learning based on what the expected output is. If we go back to the example of how to make a machine learn how to do this, then we have more ideas. First, a blood extraction can have a lot of analysis. From there, we will take measurements, like glucose level, hematocrit, ______, sex of the person, etc. These first steps passes from an individual to its feature representation, which is how we will see this person in the domain of the problem. We can take as many different characteristics as we want, but more will make the problem more complex, although they can explain more. We will go on this later on. Then we could measure for a given sample of individuals all these measurements and as we said the advancement of diabetes, so we will end up with a table like this. Glucose 80, sex is masculine, there are other things that I measure and the advancement of diabetes is 80%. Then we have another row for sex feminine and the glucose is 130 and the advancement of diabetes is 0.1. Using these, we will have to have an algorithm that actually using all features and some mathematical rule minimizes the predicted output versus the real true output. For example, in our case, we could have a rule that says for all males, diabetes level is glucose over 200, for females it is 2 times glucose over 200. With this rule, the algorithm for the first case would predict a diabetes level of 80/200, 0.4, and for the second case 1.3. Then we would compare this predicted output of 0.4 and 1.3 against the true values, 0.8 and 0.1. As they seem wrong, we need to change the rule until we are satisfied. How we change that rule and when we are satisfied will be interesting topics for the next modules. So stay tuned. In the process of this learning algorithm, we had to do a lot of little things. First, we needed to pass from people to an actual sample and from there to some measurements to represent those people. Later on, we had to find some rule that given all the measurements, we can get a predicted value. We call this a model. Continuing, we need a way of knowing how far away is the predicted value from the true output. That is the cost function. Later on, we used our data set to minimize this cost function to get the best model possible. This is called training. Finally, we need a way of knowing how good this model is, we call this evaluation. This is the common framework for learning algorithms, and we will cover all these points along this course on the whole path. Now let's imagine another problem to attack. We are developers in Tesla and we face the following case, we want to detect from an image if it has a stop sign. One can imagine this is super important, since based on this, our car will stop or actually continue and maybe crash someone. How could this work? Our workflow would be take a picture of the street, then transform into values, then we have some complicated magic, and then we have the question, do I have or don't have a stop sign. And this problem is super similar to the last one. We would need some model to go from the image representation to a yes/no answer, train it by minimizing the cost function, then at the start we would just guess a yes/no and we would need a true response so we can actually measure how far we are from the true value, but there is one fundamental difference. On the diabetes case, we want to predict an ordinal number because the advancement of diabetes could be any real value between 0 and 1. In this case, we call the problem regression. On the stop sign problem, we want to predict a categorical value, yes or no. There are no intermediates, which means some things will be slightly different. This problem is called classification. On modules 4 and 5, we will tackle regression, passing to modules 6 and 7 where we will tackle classification. And what about module 8? Stay tuned for the next video to see what we will cover in module 8.

Demo

Now before I end in this module, let's do a little demo on review how to install Python in Mac and Windows and let's set how to install the virtualenv we will use for the rest of the course. Let's go. Hi, in this super quick demo, let's do a quick catch up of how to install Python and what ______ are we going to use? So in this folder that I already uploaded in the course materials, as the ECAP for this course, we can see that actually there is a file called install_python, so let's see what that looks like. Here my operating system is actually macOS, but you will find that there is actually not much difference in the whole course between Windows and macOS and Ubuntu, actually. So if we check the file install_python, we can see that for Mac users, what we need to do is actually to install this code, here I already input the command, then you need to install Homebrew, which is actually like the ______ for Mac. Later we have to set the path so that actually it finds the Python executable, and finally, we just do install_python. This will install the latest Python. For Windows users, it's way simpler. You just go to that page and it will have the binaries and installers for all the different versions of Python, you just go, click, and follow the instructions, and you are ready to use Python. Later on, with Python already installed, what we will use is actually virtual environments. So, how do we actually use virtual environments? Well, virtualenvs are actually embedded into Python free, which is the version we will use. So all we need to do to create a virtualenv is actually go and call python -m venv and the name of your virtualenv. So I already created one for myself, which is called Pluralsight. If you want to create your own virtualenv named Pluralsight, all you need to actually input is python -m venv Pluralsight, I will put a 2. And this, after some time, will actually create a virtualenv. If we check, now I have a folder called Pluralsight2. Then all you need to is actually get into the virtualenv and install the actually the packages that are in requirements.txt. So to get into a virtualenv, all you need to do is call source and then go into the Pluralsight2 folder, which is my new virtualenv, bin/activate. This will get my inside of virtualenv. So what I have to do now is pip install -r requirements.txt. And with this, basically it will install all the packages that we will use. And that's it. It's super simple, and for Mac it's really, really the same, but in PowerShell. With this, we are ready to go. For the rest of the course, we will not use a terminal, we will use something called Jupyter Notebooks, which are a nice way of in the browser having embedded code and text and it's always agnostic, so we will not have any issues. Thanks.

Summary

And just like that, a module goes away. However, we did cover a lot because we sat foundations on what we will cover in the module. However, this module we learned what machine learning is and how it fits with the statistical models. We discovered supervised learning with examples. In this topic we learned about models, cost functions, training, and evaluation. And finally, when there is no level data, we learned how unsupervised learning can help us. I am so eager to start talking on how these models actually work, but first let's set some common rules on how we can actually find those models and how we decide this is good. Hop on to this ride.

Finding Those Models

How to Learn in Machine Learning? Cost Functions!

In the previous module, we learned what machine learning is and how its problems separate into two main categories, supervised and unsupervised learning. For both cases, however, we have a similar procedure. We started with some rule, which were the models, we will learn about different kinds of those in the next modules. Later on with the help of some cost function, we had an idea of how accurate this model was. Given this pair, we can optimize to the best model with the least cost, this process is called training. The output of this process is actually the best model for our dataset. So here we will dive deeper into this process, it's nuances, and how we can get the framework on how to get the best model possible. Of course, the first stepping stone in this process are the cost functions, so let's dive deeper into them. Cost functions are the connections we get from a given model to how accurate it is within a given dataset. To get a better idea, let's go back to something we are familiar from our last course, linear regression. Do you remember that? Linear regression was a technique to find the best line that fits the data. We have said in the previous course to understand the functional relationship between two variables, x and y. Let's imagine again we are real estate brokers and we want to predict house values using square footage. Linear regression was great for this problem because we stated that the price, when the price is a times the square footage plus b. And using any tool, like a jamovi, we can find the best a and b. How do you find a and b now? Now we are going deeper into there than before. In this case, the model is linear and it depends on two parameters, a and b. Suppose initially that we select those at random, for example, a=0 and b=1, therefore the price equals 1. That doesn't seem right. The idea now is to understand how bad it is to update them. In a picture we can see that actually we're pretty far away. So a measurement of how far it could be would be that actual formula, which is called the mean square error. It will make sense in the next module, but for now we can understand that with this we have a way of associating a model to a number of how good it is. This is what we call a cost function. Now what we usually do is apply some rule, more to come on the next video, to get a new set of a and b based on the previous cost. In our example, may we got a= 0.25 and b=-2, so the new model looks like the chart on the right, way better, right? This is the main use of cost functions. It's what allows us to tweak the parameters of a model and improve it until we are satisfied. And when are we satisfied? Well, if we imagine many iterations of this process, what we can have is sampling like the following. Since we expect it to get better and better every time, just to get in context, remember each dot here represents a new selection of a and b and the cost associated, therefore, a new model. Now we can end when we see that the loss actually does not vary that much or continue until it gets the same and we are done. How do we know it will get there? Mathematics is what will tell us that for any given loss and method, it will actually get to a minimum, or maybe not, it depends on the case. This is the way we adapt with the trained model. To catch up then, cost functions are mathematical functions that map models, that depend on multiple parameters, into a real number, such that when the model gets more accurate, the cost gets lower. It has some minimum that will indicate the model is optimal and hopefully it's easy to calculate. This will be our simple tool to train our models. Let's continue this journey on understanding now how we can make that magic update rule in each iteration, that means how, from a given a and b and a cost, we can find the next a and b. See you there.

Finding the Minima: GD and SGD

So now we know how to get the cost of a given model. We rock. The only step we need to know now is how to make that magic update rule. Let's tackle it. To recap from our last video, we needed to predict the price of a house given the square footage. For that we decided the first good model was a linear fit. For getting the best a and b, we started with some random values and then with the help of our cost function associated to this model it got a real number, for example, I don't know, 10. This number represented how good is our model and the lower it is better. The question we have is then, given the full dataset, the previous model, therefore a and b, and the cost of that model, how can we get a new pair of a and b that are better? Let's look at the bigger picture for a second. What we have is a cost function that associates for each pair of a and b a real number being the last. This looks like the following. And what we want to do is find that minimum value of this cost function. So in our case, we are here, represented by the red dot, and what we want to do is actually go down the slope until the minimum. This type of problem is known as mathematical optimization, and luckily for us, it's a super solid problem since the 1920's, and we have many, many, many methods for that. We will analyze the most common two, gradient descent and stochastic gradient descent. The idea behind gradient descent is very simple. There is a sample called a gradient, which basically evaluated at any point and gives you a way of updating the parameters in the direction of the minima. In particular, the formula for gradient descent is basically to get the old step, get this new gradient, which basically the libraries calculate, multiply by an alpha, which will be a hyperparameter, more on that later, and make that difference, which basically the old values, minus the new gradient times the alpha. What will happen then is that during each step of gradient descent, we will calculate that gradient with the full dataset and we will get slowly up to the minimum, as the image shows, which is great. So in recap, gradient descent is an optimization method that calculates the the gradient of the cost function, and based on that updates the parameters in the direction of the minima. Easy, right? However, it has some flaws. To calculate the gradient, we need to iterate through the whole dataset. Imagine we have millions of data points. To get one update, we will need millions of calculations. That's not so great. Also, sometimes it is low because it takes a lot of operations to get to that minima. This is because the gradient becomes smaller and smaller after each iteration. So, in response to those problems, a new method was born, stochastic gradient descent, or SGD. What is SGD? Basically it is to do gradient descent, but in a sample of the whole dataset. The process is as follows, take a sample of size 1 from the dataset, that means a data point, then update by the following rule. The next step is the old step minus the gradient only evaluated in one point times alpha. So, as we can see, this is way faster because for each iteration we only have to make one calculation, which is awesome. And what can happen? The main issue is that the actual trajectory of SGD is more chaotic. How more chaotic? Like way more chaotic. Check this picture. SGD has the characteristic of not always going downward and in the loss also, only eventually in an average. So the graph of the loss against the number of iterations looks like the one in the right. It doesn't always goes down and, actually, the trajectory is more chaotic, but in average it goes where we want. Therefore, why would I ever use gradient descent, right? Well, gradient descent has great characteristics that there are a lot of comparisons results. That means that almost all the time it gets to a nice minimum. That isn't so assured for SGD. And which one should I use? Well, it depends actually, because both have some characteristics that defines them. So for GD, it could be a great case if the dataset is not gigantic, the function is really smooth, and the CPU cost or the time is not actually a problem. However, we could use SGD if the dataset is really big, like big data, or the CPU costs are an issue, or we need to iterate fast and do some agile development, or the cost function is really, really messy like the ones in neural networks. With all this knowledge, we have only one gap in the jar, how to find that alpha and how to tune GD and SGD. For that, we will check some nice tricks and with that the section on how to train your model gets a round up. Incredible how many things we used and did not know in our previous course, right?

Making Things Faster: Feature Scaling and Learning Rates

And we got to the final part of this trilogy of training a model. Now we will learn all those tiny things about alpha and the dataset to make our optimization conversion fast. As we have seen in the last video, both GD and SGD have an alpha hyperparameter in their equations. So we raise the obvious question, what value does that need to have? And of course, as always in life, it depends. But let's analyze our classic house valuation problem. This alpha is what we'll call a learning rate and it reminds basically how our training will work out. We have initials a and b for our linear feet, and a loss cost associated with it. Suppose we selected a really small learning rate alpha, like 0.01. Then what will happen is that actually the next value will be super close to the old one because the term subtracting the old steps was really, really small. As we can see the chart where the two red dots are really, really close. Therefore, what will happen is that we will get to that minimum, but really slow. The image for this would be in the following one where we pass through many, many iterations until we get there. If we increase it a bit, then of course the update will be larger, and then we will go way faster. For example, here in just four updates, we got to the minimum. So the idea would be let's make it big, right? However, check this. What happened here is that the learning rate was so big that actually where we should and at each pass we never approached the minimum. Our method will end in our algorithm, but we would be nowhere close to the right answer and it's so difficult to diagnose this. In summary, we can have all these possibilities where if we go too low, we will never end. If we increase it too much, we would get to a bad solution or actually never converge like the case in red. So it's a lot of tuning, usually one of the main hyperparameters will tune in our algorithms as the learning rate because it is so important. And our solution, which we will not call, but it exists, are some rules that while training advances the update of the learning rate to be bigger based on the slope, so actually doesn't spend so much time closer to a minimum and gets there. However, this is an advanced technique and we can get a lot done without it. Now let's suppose we got to an alpha we like. We train the linear feet and, bam, we love it. We have what it takes. We use the model to predict the values of the next house and the next and the next and life is awesome. But as the only thing that doesn't change in life is change itself, we move to another city. We try to use the exact same algorithm and it takes a lot of time. What happened? We check and yeah the dataset is a lot larger, so this makes sense, but we don't really want to change the algorithm or the learning rate since it was already tweaked. What can we do? Can we use the same algorithm and learning rate in another city? In this problem and a lot more, what we can do is feature scaling. Basically this means that each of our random variables that we got on our sample, we normalize them to have mean 0 and variance 1. Remember about that. In formula, this means that the next x is actually the old x minus the average divided by the variance, and we do this for each feature we have, which are actually random variables, remember. And what why do we do this? Well, there are theoretical and practical reasons. From the practical side, the picture shows it all. When we scale features to mean 0 and variance 1, if we imagine the cost function has a lot of slices, one over the other, each of these slices is more of a circle and less of another shape. In that case, it doesn't matter where our algorithm will pass, depending on the initial value, it passes a straight line to the center, and that's the optimal. If we don't do it, there might be some quick paths, but also some slow ones, and that's what happened to us in another city. In the theoretical side of life, as most of the models we will use assume normality of the variables eventually, we are approximated that normal to a normal with mean 0 and variance 1. This will bring homoscedasticity to a population, which is always another important hypothesis in most tests, and both factors increase the chance of converging rapidly. Doing a quick wrap up, we learned about two of the tiny nuances the training mechanism has, learning rates and feature scaling. Via the correct learning rate, we can get much faster to the correct model without too many iterations. However, we must be careful to avoid overshooting or we won't get to the best minimum. Finally, with feature scaling, we have a nice trick that allows gradient descent and most learning algorithms to perform way faster and better. Now that we have seen all the different parts of how we get to a model, let's wrap it altogether in the house valuation problem we already know from the last course.

How It All Fits: Going Back to the Model!

Oof, that was a lot. Let's do a super quick recap of how it all fits together. Let's remember, we wanted to get the price of a house given the square footage. So the first thing we did, as this is a supervised learning problem, we got a dataset of actual houses with its actual price. Later on, we would do the feature scale so the dataset has mean 0 and variance 1. This is crucial so that the gradient descent goes faster and the end result is better. We proposed a linear model, price equals a times square footage plus b, so we want to find the parameters a and b. We initialize them randomly to say a=0 and b=1. Then we have a cost function that for a given a and b, using the whole dataset, assigns a cost. It would be, for example, 10. With all these, we plug it into gradient descent such that we start iterations, but for these, we need our alpha the learning rate. How we find the best alpha is a subject for the next videos, but for now, let's say we found by someone that the best is 0.1. Then the update rule will be, in that case, the next a and b's, would be the old a and b's minus the gradient in all the dataset times 0.1. After some iterations, what we will find is that the loss gets to a minimum. We know this because the graph of the cost against iterations almost never change. Our final train model are the a and b, such that the cost is minimum, and that, my friends, is how we train a model. This process happens every time for every model, independently if its supervised or unsupervised. This is the key to understanding this course and machine learning. With this training process, what we end up having is one trained model per selection of hyperparameters. What are those hyperparameters? Well, for starters, we have the alpha, the number of iterations we have, the optimization algorithm. Do we want GD, SGD, Adam, another one? So the question now becomes a little bit bigger. How in the world do we select the best combination? Maybe even more important, does it matter? This, of course, depends on the problem. The learning rate usually is very important always. However, the optimization algorithm may or may not be crucial. Sometimes both GD and SGD are okay. Sometimes both suck. Sometimes we don't mind doing a few more iterations. But sometimes we are CPU bound and we'll really be tight in that area. So we need a framework, a way of deciding. We will leave that decision to the next module because this one already is big. That module will cover, between a lot of topics, how to select those hyperparameters, but for now, let's do a demo, right?

Demo: Implementing GD and SGD

Let's make a little stop for a second. Until now we went nonstop in theory mode to discover how to get the best trained model for my problem, and that is awesome, of course. But I wanted to invest some time in polishing up those Python skills and try to recap and ensure how GD and SGD works because it's one of the most common used algorithms and very, very important. In this demo, we will implement GD, implement SGD, and show how they work in a deterministic function where we already know the minimum, so we don't have any randomness. Let's go. For this demo, we will use a deterministic function to implement GD and SGD and understand what will happen with the learning rate. So this is Jupyter Notebook. This is the type of notebook that we will use all along the course. It's great because it allows to input text as you can see in the second block, and code as you can see in the first and third block. Also, it is really nice to play with data to understand what is actually happening, you will use this a lot, and it's also in the browser, so it's really portable. Going into this specific example, we will install all the necessary libraries and what we will do is actually optimize a function x minus 5 to the square plus 3. We already know that the optimal value is x equal 5, therefore y equals 3. So first, what we will do is actually go and see a plot of the data. For that, we will use the function linspace, which actually gets a data point linearly between the first and the second value, the number of data points is the third value, so we are getting 100 data points between 0 and 10 and we are plot Y to be the formula we already know. We use matplotlib as we already know and we know this is a parabola where the optimal is where x equals 5. So for gradient descent, we need the gradient. The gradient, I already give it to you, is actually 2 times x minus 5. So we defined the functions of the cost and the gradient. And for that, we will implement gradient descent. Gradient descent, basically, as we already know, is a iteration, get the current value we already have and remove the learning rate times the gradient evaluated at that x. And that's all. What we will do is actually add back printing such that we know what is going on, and we will return all the costs so that later we will evaluate how the costs go with iterations. So let's start with a random number between 0 and 10, 200 iterations, and a learning rate of 0.01. As we can see, at the 200 iterations, we didn't quite get to the minimum. We get that the x is equal to 5.04 and y equals to 3.001. So we really close, but not close enough. To understand that, let's plot the cost history of our iterations. And we can see the cost always and monotonically goes down, but it didn't quite get to the final point, the learning rate was too low. So let's try it at a bigger learning rate. Now we'll try a learning rate of 0.1 and 50 iterations. As we've already seen, as the learning rate is higher, two things can happen, we can get really quickly to the minimal, or we can be overshooting. We don't know until we try. So let's try that with our implementation of the gradient descent. We do the same things and what will happen is that we actually got to the minimum with 5 degrees of accuracy. That's really good. And if we see the cost over the iterations, we really, after about 10 iterations, we got to the optimal. So we are not overshooting. We are at one of the really, really good values. So to end this demo, you could ask, now what about SGD? Well, the funny thing is that actually the implementation of gradient descent and stochastic gradient descent is actually the same in this case. Why so? Well, because basically it's a deterministic function and we already know all the dataset. We are only evaluating, however, we will see that the SGD idea is exactly the same and we will see how we are going to implement those, and actually the libraries are going to implement those, in the next demos. The important thing is that we've already seen how the learning rate actually changes things, so it's all we have seen in the theory. To end this, let's see if we add 2 more learning rates, 1 and 10, what will happen? So we are going to the 200 iterations and do the same actual procedure. Let's check the graph. And what we can see is that actually, for 0.01, we kind of get to the optimal; for 0.1, which is blue, we get to the optimal; for green, which is linear rate equals 1, we actually are overshooting so we never get to the optimal; and for the learning rate equals 10, we actually got an error, which you can see above, which is overflow encountered in power. That means that we got higher values each time until we got to a math error, the computer couldn't compute such a big value. So we have all the four cases we knew from the theory, the red where we actually went above and above, the green, which was the overshooting; the red, which was the too low; and the blue, which is actually the best. With these, we can wrap up this demo and we saw how to implement gradient descent and stochastic gradient descent and how the learning rates affect the velocity of the training.

Summary

That indeed was a lot. Thank you for staying with me. From now on, we will go problem by problem understanding how to use each of the models that are more common. However, that is super less theoretical work than what we learned this module. In this module, to do a little wrap up, we understood how to train a model via cost function, how to use an optimization algorithm, and how to tweak the learning rate. With all these, we can advance further and tackle the first big problem, regression. This will be super fun.

Predicting Linear Relationships with Regression

Back to the Basics: Linear Regression Again

And we got to the first big problem, regression. Regression is a supervised learning technique to predict real numbers based on a number of features. Let's tackle it on. Let's imagine we are entrepreneurs in the medical system. Our objective is simple, we want people to be healthy. In that purpose, we want to design a system that from a single blood extraction can effectively predict how advanced diabetes is, even if we are not diagnosed. This is crucial for a lot of people. So let's get a good model. Our engineering friend already designed a capsule that allows to extract blood and from there take 10 measurements, including age, sex, body mass index, average blood pressure, and six other protein levels. The only thing we are left to do is design this model. Let's remember linear regression for a bit. This will allow us to start with a familiar example and expand to the unfamiliar settings. However, don't panic, most of the hard stuff already was learned. In simple linear regression, we have a random variable X, called explanatory variable, and a random variable Y, called dependent variable. The idea was to find the functional relationship between those and we model it by Y = f of X plus an epsilon where we said epsilon was a normal distributed random variable with mean 0. Sounds familiar. Well of course, it's the familiar setting of supervised learning. In the case of linear regression, we said f of X equals a times x plus b, where a and b were parameters of the regression. And previously, we used software for finding those. In the previous videos, we understood what the software did in the background, so now we know how to find a and b on our own. If we remember matrices from high school, then we can represent these as f of X equals a times x transposed, plus b, where a is a 1x1 matrix and b is a 1x1 vector. Why would we do that? Well, for a couple of reasons. The first is that now we can implement evaluation of the whole dataset in one line in Python, if we consider x to be a row vector of all values. The second will be that as we add more features and we pass the multiple regression, this formula will stay the same in the vectorized way. This is what we call the general linear model. It's how we represent any linear model. Going back again to linear regression, how that we have a model, we need a cost function. If we remember the previous cost correctly, the main cost function for linear regression is the mean squared error. That is the sum where all values of the difference between the real value and the predicted value by the line to the square. Of course, then it depends on the parameters. In pictures, it's as follows, where we are adding up all the values in red. Now this cost function has the following characteristics. It is bowl shaped. It is super regular. It has a unique minimum, and it is convex. With all these characteristics, one can tell that both GD and SGD will work. So for now, let's go with GD. We then already have our training in place. When we execute these for a number of iterations, we will end up with the correct a and b and we have our model. We will check the extra stuff later, like assumptions and so on. But for now, let's go dive deep into one of those technical points that we have from the model before, hyperparameter optimization.

Hyperparameter Optimization: Train/Dev/Test Sets

And now, going back on track on how to find the best model, let's attack the problem of hyperparameter optimization. Suppose we want the best house valuation algorithm. What can we do? We can have so many different learning rates, so we would like to understand which one is the best. For the sake of having an example in mind, let's imagine we are in our linear model and we want to test an alpha of 0.01, 0.1, and 1. We already decided that we will do 30 iterations and we will train the linear model. Therefore, in the end, we would end up with three trained models, and for example, some measurement of the loss of each one. Which one do we choose? We could say, for example, the one with less loss, right? However, there are at least two problems with this approach, which seems intuitive. One problem we will have is that we will know how good our model will be for our current dataset. However, if a new house comes, we actually don't know how good our model would be since we haven't tried it out. This, as silly as it sounds, is one of the biggest problems in ML, which is that our dataset is not representative of the actual distribution of the data. This is called overfitting. The second one, a more intricate, is that we made optimization of the parameters a and b with this dataset. Therefore, this cost estimation is already biased. What this means is that the cost calculation is performed not independently of alpha, so we cannot actually use that as a way of comparing performances of alpha. This is why we need some external offline metric to evaluate our model. So, what can we do? If we calculate this metric in the same dataset we used to optimize our parameters, then we are in the same mess. This is where the hero appears in our story. The new dataset, the dev set. Another dataset? If I only have one. Well, the usual way is splitting the actual dataset into training and dev set as the image shows. This dev set is untouched during training, so we can use it to actually measure how good the model is with this metric in an unbiased dataset. In this way, for each learning rate, we will get an unbiased estimation of how good it is. In the picture, we can see this working and how in the end we will arrive to one number per hyperparameter. How this number is calculated, which is called the metric, is dependent on the problem. If we are classifying, it can be the number of correct predictions versus the real ones. If we are in regression, the actual cost function in the new dev set. The important thing is that we end up with one and only one number per hyperparameter. So this, plus a business idea of what we want to optimize, get us to the choose of the correct final choice of hyperparameter. And now we know how to get to one and only one model. Great, right? We have one tiny detail left out. What happens if we want to know how good is this final model in the real world? How accurate it is? As we cannot measure that in the dev set, because of overfitting again, we need another final dataset, the test set. Which after getting to a final model and training again with all the train and dev sets merged, will serve us well in accurately estimating the model performance. To recap, we then have usually three datasets. The train set for training the model, that means for finding the correct parameters. The dev set for hyperparameter tuning, that means to find the correct hyperparameters. And finally, the test set, for evaluating the final model performance. It depends on the dataset how much we put into each because in normal datasets, the training set is around 70% of the data. But if we go into deep learning, it can be as high as 97% of the data, because we will have many, many data points, which is actually awesome. With this, the final flow for finding a model gets completed. We start with a dataset. We split into train, dev, and test sets. Using some searching mechanism, we will have a selection of end hyperparameters to compare. For each of those, we train the model using the train set, then with the dev set we get its performance. We compare the performance of the train models in the dev set and we choose the best hyperparameters for us. Be aware that that if we have, for example, two or three different types of hyperparameters, this can lead to a lot of training runs. Then with the full trained plus dev set, we retrain our model with this choice of hyperparameters fixed. Finally, we then measure its performance in the test set. These will be its actual estimation of real live performance. And we got there to how to, in an unbiased way, find the right model and estimate its performance. Incredible, right? It is way more complicated than what one initially thinks. I am super proud of you now because as simple or not as it sounds, getting this right sets you apart from many, many, many machine learning developers.

Demo: Perform Simple Linear Regression

This demo did come fast. As the hard theoretical part is done, it is time to start getting some practice with all the contents and how the pipeline works as we discover more models. In this demo, we will perform a simple linear regression in a Pythonic way and we will split into train test sets to evaluate the model performance. Sounds simple, but it is important to get these things good in a practical way. Let's use all this linear regression knowledge to know how to get the advancement of diabetes by the first feature of the diabetes dataset. For this demo, first we will load the dataset. For that, scikit-learn has already some prefilled methods that basically, be it a dataset API, we are able to load a lot of datasets. In this case, the diabetes dataset. As we already told, as we are doing simple linear regression, we will use only the first feature. We will leave for a future demo to use more features. And finally, we will use the train test split method from scikit-learn to split our dataset into train and test sets. There is also another one if we wanted the dev set, but here we will not use hyperparameters, so we are okay. As a final note, and this is only for this demo, and this is different from the other things that are in general, is that we need to reshape our data frames to actually be only one row. That is because we are using only one feature. In the rest of the course, we will use more. Continuing, we can verify that actually we have 353 rows in the training set and 89 rows in the test set. So let's go and train our model. For training a linear regression model in scikit-learn, it's really easy. We have to use the linear model API and just instantiate the linear regression option. And that's it, it has a fit method where we input the train, dataset x and the train labels, y. And we are done. Also, it is nice enough that if we put all the test cases, it will predict all the predictions. And with that, we will calculate the coefficients. We will get the mean squared error, remember that's an estimation of the cost function, and their square. Finally, let's plot this. We can see, it's not an actually great model. It doesn't have a really good R squared and the mean square error is really high. But this is the first case where we did linear regression. This is awesome because also it was really simple. Let's continue to learn more on multiple regression and come back to this example to see more of how can we predict diabetes.

What if I Want More Variables? Multiple Regression to the Rescue!

Linear regression is fun, right? However, it would be awesome to be able to use all those variables from the diabetes dataset, so let's try to tackle that. Let's imagine we would like to include age and body mass to our linear model. That means we would try a model diabetes advance equals a times age plus b times body mass plus c plus epsilon, where, as always, epsilon is a normal variable with mean 0. If we remember the general linear model, we could do the following trick. Diabetes advance equals the matrix a, b, times the matrix, age, bodymass transposed plus C plus epsilon, where we just group the terms together for a matricial product. Do not panic is this is unfamiliar to you, just bear with me this second, we will get to a known ground. If we trust this trick, then we are back to the linear model, but now x is not a random variable, but a series of random variables where each row would be each variable, in charts this would mean that the first row will be that age for person 1, 2, and 3, and the second row we will have the body mass for person 1, 2, and 3. With this new layout, we are back to the linear model. So actually, to train a multiple regression is exactly the same as linear regression, but instead of numbers, we now have matrices. Doing the mental map, if before we iterated samples of x be numbers, we estimated numbers a and b, and now our cost function did operations with numbers. Now we iterate samples of x, being rows, one per feature. We estimate a and b, where b is a number, but a is a matrix. And now the cost function operates with vectors. Considering the square of a vector to be its square norm. And with that mental map, we are back to before. Lucy for us, most of this internal abstraction is transparent to us via the layers in software. What we need to know is which format to put the data at the start, which is exactly what we saw. Let's put this new concept into practice and check that it's as easy as 1-2-3.

Demo: Perform Multiple Linear Regression

Now we go fast, right? As I foretold you, the main workflow and pipeline is the hard stuff, then it's just a matter of picking the pieces and a couple of new ideas for the new models. In this new demo, we'll perform multiple linear regression, and check that actually as part of the general linear model, it is the same as the simple case. And we get to the second demo. In this demo, we will include all the features from the diabetes dataset to try to predict the advancement of diabetes. For that, we will import all the packages again, we will do all of the same, but the only difference is the check that the diabetes_x now doesn't only include the first feature, but all of them. We can verify that the shapes of the training and test set is the same. But there is one difference, and that is that the number of variables that the training set has is actually 10. Now we have 10 columns, 1 per feature. This should actually improve our model, right? And how do we do multiple regression? It is exactly the same code as you can check. All these abstractions that happen actually are all taken care of by scikit-learn, which is awesome. You may ask me, and then why did I have to learn how to do this and how it works and what are the differences? And I think the main point here is that by knowing all of these, you will become a better debugger when these models don't work. There is a difference between just telling you what to do and how it works because when you know how it works, you become not just a doer, but a maker. Continuing with our demo, if we get the coefficients, mean squared error and variance score, we can check that actually by including all the variables, we got 10 coefficients, of course, 1 per variable, but the mean squared error reduced from over 5000 to less than 3000 and the variance score, which is the R squared, increased from 0.01 to 0.45. That means that we improved a lot our model and now we explain a lot of the variance just by adding a lot of features.

What No One Talks About: Assumptions

No model from me would be complete without the statistical way of life, right? In other words, let's talk about what no one talks about, assumptions of the model. As I have already mentioned before, one of the biggest pitfalls I see common machine learning fail is to never validate assumptions of their model. Only to find themselves with a model that doesn't work in production as they expected, even though there was no overfitting. But you are not a common dev. You are special, you are at Pluralsight. So, let's understand what needs to happen to apply the linear regression model or, as a matter of fact, a general linear model. In GLM, we normally ask for linearity, normality, homoscedasticity, no outliers, and no collinearity. That's a lot. So let's go one by one understanding how to check those and what do they bring? Linearity, normally it's understandable as we going a linear model, and to check that, we normally check the residue plot after the regression. Normality, we know from the previous course, we have the Shapiro-Wilk test. Remember homoscedasticity, this meant that the variance of all the variables is actually the same. One nice way to attack this problem that, in fact, helps us a lot, is to feature scale. This is also a reason why feature scaling rocks. It brings all the variables to the same unit variance. Yay for us. Homoscedasticity is not a problem in our case. Regarding outliers, let's wait on this one until module 8 where we will know more about clustering. And finally, we have collinearity. Collinearity means, basically, that two variables say the same. And how do we measure that? Well, correlation of course. As we know from the last course, it is super nice to start seeing how, in fact, we use all the tools from the previous course in new ways. When two variables have high collinearity, then what will happen is that their correlation will be close to 1. So what exists for multiple regression is the correlation matrix, as the one in the left. For example, here in this dataset, we can tell that age and preg are apparently highly correlated. How do we know that? Well, because this matrix is all the 2x2 correlations and we can see that age and preg have like a greenish kind of yellow number, therefore we can check that the correlation is over 0.75, kind of. Now the issue is, what adds correlation to our problem? Well, it adds variance. Since our model will be more unstable, therefore less likely to converse to a minimum. And what can we do if we find two variables are highly correlated? We have two main options. The easy one is to drop one. This has been a common practice for some time, but we found out that sometimes the variance explained by both variables is a little bigger than each other's contributions separately, so it is not always that simple. The second approach will be to use the techniques from module 8 to create new variables, which are a linear combination of both. We will see more on this later. With all these, we made a pretty strong pass for the assumptions and checks what has to do in regression. Now let's make a quick demo in action, and with that, we finish the module.

Demo: Evaluate a Regression Model

And we got to the final demo. We now know everything we need to evaluate correctly a regression model, check if it works, and validate its assumptions. In this demo, we will validate the assumptions of the regression model. We will evaluate the significance of it and, finally, we will get a final performance estimate for our model of diabetes progression. We will check how assumptions are really important in linear regression. We will check the normality assumption and the collinearity assumption. First, of course, we will import all the necessary packages we will use, we are used to that. For the normality assumption, we will generate a synthetic dataset. For that, we have already a method in scikit-learn that is called make regression, which basically outputs the dataset that the linear fit is great with just some noise. Later, what we will do is actually transform our data so that our dataset is not normal, it's exponential. And we will have two sets of labels or expected outputs. The actual real one, which is good, and then we have the transform one. But we will invert those because first we will transform, so first we will have one that is exponential and then we will have one that is transformed because it applies the algorithm and, therefore, we go back to the initial case, which is a good fit. Going back, and just to make sure we are okay, we will call the target distribution, the exponential one, and then the transformed one is the one that we go back. And therefore, we can see that the first one, the target distribution, is exponential and not normal. And what can happen in the linear model? We know from the first course that the linear fit shouldn't go well. And what we can see is that it's that case. We can see that the R squared goes from 0.41 to 0.65, just for doing a transformation and going with the normality. But what's more important is that the residue plot for the first model is really bad, you can check that there is not a random distribution around the line of all the residuals. This indicates also nonlinearity, as we know. Nonlinearity is associated with a lack of generalization of our model, so it's actually not useful, and this is a way to know it. When we have normal data, then we can actually generalize. For collinearity, what we will do is get the diabetes dataset again and again run the linear regression. We are using stats models instead of scikit- learn, just because it outputs this nice summary of all the necessary statistics we have on the model. The ones we will check is actually the R squared, which is 0.51. We have an f statistic, which is the ANOVA of the linear regression. This is good because it's actually significant. And we have the coefficients and the significance of each of the coefficients of each of the variables. The difference between this case and the last course case is that in the last course we supposed that all the data available was the one that was had. In this case, we are assuming that this data is just a part of all of the other data. Also, for our dataset that goes into this model, we are splitting to train and to set usually. What does that mean? It means that the p values, in this case, should include all those assumptions. What that says is that we cannot check the significance of each of the variables and the coefficients as a hypothesis test for each of the variables. However, we can use that information because usually what will happen is that having a high non-significance and a high condition number will usually mean that there is some collinearity, even though we are not doing a hypothesis test per se, because our hypothesis should include also the separation of train and test set, for example. For doing that, what we will do is calculate the variation inflation factor of each of the variables. We already have a method in scikit-learn called variance inflation factor. So we are great. By doing that, we can check that we drop two variables, so actually that instability was due to some collinearity. The variance inflation factor, really small and really short, calculates the variance by dropping that variable. So if that variance actually doesn't change so much, what happens it that it didn't explain so much of the model. What that means is that actually some other variable explains the same as this one. What that indicates is collinearity. So by doing the linear regression with two examples removed, what happened is that we had the same explanation of variance, 0.521, but we have a really lower condition number, which actually indicates us that this model is more stable, and therefore, generalizes better.

Summary

And we got to the end of this module. We started machine learning and statistics meshed together in action to a great mix. In this module, we have learned how we optimize hyperparameters components via train/dev and test sets. We understood how to perform multiple linear regression. Later on we analyzed what and where were the assumptions of the models. And how to get performance metrics on a model. Put into practice all the theory from the last module. This was awesome. Regression is a passionate topic and we are just scratching its surface. Let's complete this journey, adding some nonlinearity to the mixer.

Understanding Regression Models in Depth

Non-linear Regression: Polynomial Features

I think so far we have a pretty good mental map of how to develop a machine learning algorithm. In fact, for regression we have a pretty good idea of how it works and how it can file. So what comes next? Let's extend the power of multiple regression with a nice trick that will allow us to fit nonlinearity. Switching roles again, now we are part of IRS. We are in charge of collecting taxes for all agricultural sales in New York State. That was a dark time in US history, when people did not want to make a correct statement of their sales to evade taxes. So we have the duty of getting a correct price model so we can verify people are actually criminals. In the past, people tried and failed with linear models. So it is up to us, the experts in regression, to save the day. What happens when the linear model is not enough? How do we know? In the following picture, we have the residue plot from a linear fit that clearly went wrong. We know it is that case because we don't have a random distribution of residues. So what we can do is add terms to that regression. Polynomial regression is just a special case of multiple regression when the new features are polynomial features. What does this mean? Let's imagine a quadratic function like the one shown. If you rename the feature x squared to be zed, then this looks like the following. And, in fact, if we make a new x matrix with zed and the old x, then this can look like the following. Which basically is multiple regression. So there is nothing under the table. We are just using the same trick to simulate nonlinearity. Let's go with an example. In the following picture, we try to estimate the following known function, which clearly is nonlinear. If we test a linear model, we can see it's not enough. So let's go quadratic. If we add a new feature to be a quadratic term and we use multiple regression, we get the following charts, which is way better. In fact, in the domain we are checking, a quadratic model appears to be a great fit. And the rest of the plot is quite random around 0. So we're great. Let's check adding a cubic term. It appears we don't gain so much over the last one. The cost is practically the same and we have a more complex model. As a matter of fact, in this example, we see a lot of the things that will repeat itself. There are a couple of points we need to take care in polynomial regression as we've seen. While adding more terms, it may be appearing we have to be aware of our fitting. Also, as adding more terms, we have a more complex model, which affects its parsimony. We will see this later. And finally, the polynomial features have high collinearity with themselves, so we must be cautious if we don't want to have a model that is highly unstable.

Overfitting: A Great Responsibility Conveys a Great Regularization

There was a time life was wonderful, a miracle, magical, and our models performed just as we trained them, but sometimes that just doesn't happen. Let's imagine, as an example, we want to predict the following function, which we already know. So in blue dots we have a sample of size 30 of the function with some tiny randomness. We can try a linear regression as we always do. We do everything we stated in this module at the last course, and we end up with the following chart. Apparently it doesn't fit so much. What happened here is super simple. Our true function is too far away from the best possible linear model. In machine learning this is known as bias. Why does it happen? On the picture we have a representation of what is happening. We have our true function, which is what we try to estimate. However, with linear functions, we only can estimate functions to the right of the red line, which is the subspace of functions well estimated by linear functions. Our actual model that we train is the best possible model restricted to our space, which is the blue dot. The distance between the blue and the black dot is actually the bias. We want this to be as little as possible, such that we are mathematically able to predict our distribution, even if computationally it takes a lot of time. In this case, without putting so fun into that, we have more degrees and we try to fit with a larger polynomial. This one seems correct, but maybe we can reduce the bias a little bit more, right? So if we continue to add degrees, what happened? In the spirit of trying to reduce the bias so much, we went the other way. This function is too complicated, and what will happen actually is that if we get a new data point, we may be super far away from the actual model. So what we end up having is a model that only works for the training set. This is over fitting. If we go back to our pictorial representation, we can add the variance, which basically is all the possible models we represent given the closest fit. Having high variance might seem cool, since we get closer to the true function. But also, we get far away and it is random actually. So we want to reduce the variance. And how can we diagnose if we have a bias or variance problem? We have three cost measurements, the cost in train set, the cost in dev set, and the cost in test set. Usually if the cost is way too high in the train set, we have high bias. If the cost between train and dev sets are too different, this means we over fit and we specialize too much in the training set. This means we have high variance. And finally, if the cost in dev and train sets are similar, but way different than in test sets, then what we have is another problem, it means that actually something is wrong with our procedure. So the point estimation of the errors between dev and test were too different. A way to reduce this issue will be cross validation. The issue we just saw was that when we tried to reduce the bias, we increased the variance. So what can we do? Here is where regularization comes in place. As you probably have foretold, it's difficult and maybe sometimes we don't want to reduce the number of features we use. We want the complex model, such that we get high predictive power. Regularization is a technique that allows us to reduce the variance by sacrificing a little bit in the bias. Usually this works by adding some term to the cost function that depends on how big the values are. So what we try to do is reduce the absolute value of the coefficients, which end up reducing the chance of having high variance. With regularization, the amount of space actually looks like this. Where regularization restricts a little bit amount of space, adding more bias, but reducing variance a lot. How much do we regularize? Well, you may already guess it. The amount of regularization is done via hyperparameter. We have to tune. This is where the art of developing machine learning applications becomes black magic. To do a quick recap, we discover that now we know how to get a new model. We can have two different types of errors, bias and variance, based on how complicated our model is. It is an eternal tradeoff we'll have to deal in our lives. But regularization comes to aid us, so we don't have to change the background model so much. It is time for a demo, right? Let's put all this in practice.

Demo: Linear Regression with Regularization

Oof, that was a lot. I admit that this video is highly technical. But trust me, this video will set you apart from the rest in terms of understanding how to assess and develop your machine learning application. For doing a nice ______ regularization and train test error, we will do a quick demo. Over this demo, we will understand how regularization works by affecting the parameters found in training, which therefore reduces variance, we will understand how these will imply that the train and test errors will get closer, and we will understand how the whole train dev test pipeline works in action. Let's go. In this case, we will check regularization, which is a super interesting topic and so key in so many models. What we will do is check how regularization makes our model more robust. How it is done by basically putting all the absolute values of the coefficients in a lower scale. What that will mean is that we are not so affected by all the other extreme values. This is not the only way of adding ______, but it's a really nice and easy one. The first thing we will do is importing all the necessary packages, and our train and test sets are actually going to be super simple, just two points. Why so, you ask? Well, what we will do is actually run a linear regression and what is called in scikit-learn as rich regression, which is basically a linear regression with regularization. Where the alpha term there is not the alpha learned in red, but is actually what we called lambda. That's the standard ______ ______ in machine learning, which is basically the strength of the regularization. And what we will do is basically add some noise to this train and test sets so that we can check how it varies the models for linear regression and linear regression with regularization. For that, basically what we are adding is some random noise to the train set, then fitting and then plotting for each of the models. Let's check how that went. If we check for the linear regression, as we know, it changed a lot. Always linear regression has a lot of this changing in the slope, which is basically changing the variance of the model based on the noise. We already know this, linear regression has this issue. What happened with regularization? Basically it wasn't affected at all. Because regularization adds the term to the cost function that makes the linear regression to not listen to those noises. Awesome, right? Let's check how this works. For that, we will create a dataset that goes well with the regression fit. We already have seen this in the last module. And what we will do is check how the coefficients work with the strength of the regularization. And this is the final plot. We can check that as the regularization parameter goes bigger, the values of the coefficients tend to go to the same place, which is 0, so their absolute value gets decreased and decreased until they get really close to 0. Also, what happens is that usually we will get a curve like the one in the right of the error right. With almost no regularization, the error right, between the test and the train set, can be somewhat small or high depending on its generalized ability of the model. When the regularization parameter is really low, what will happen is that the difference between the actual true coefficients of the model and the value set we got, it's really close. That makes sense because actual regularization isn't making such an effect and what will happen is that as we increase that, the difference will get bigger. This doesn't mean that this is a bad thing. This means that we are not so inclined to overfit in the training set, and, therefore, we can generalize more. However, as all things in life, there is a tipping point. In this case, around 1 or 2, when we go higher than that in the regularization parameter, their error goes way up and we actually aren't predicting anything, which goes hand in hand with all the coefficients going to 0 in the chart in the left. So this is how regularization works. It's interesting that just by adding a single term to the cost function, we have such a high difference in the models that we have.

Demo: Perform Polynomial Regression

As we learn this new trick, all of the sudden a lot more problems can be solved with regression. Why? Well, there are a lot of theorems in math that tell us that any good function can be as close as we want to a polynomial. So why we don't just stick with polynomials and wait for this demo to find out? In this demo, we'll perform a polynomial regression. We will understand how collinearity and overfitting starts to act with more terms. And we will understand the effects of outliers in this regression. In this demo, we will use everything we have learned to actually predict the sizes of the pumpkins. This is really important for the IRS because we need the true taxes. For that, and as always, first we will import all the necessary packages and later on we will create our dataset based on that CSV file that we have already in the GitHub repo of the course. What we are doing is some preprocessing, which are used to. We will assign the item size to the actual item size from in letters that we have in the CSV. Later on, we will get the actual prices, the average of the high and the low price because those are market terms. And with that, we got the final dataset. Finally, we will drop all the null values and missing values because those always exist. And we will have the dataset like the following, where for each pumpkin sale, we will have the size, the price, and a class we will assign. We have 104 training data points. So what we will do is split into train and test set and train polynomial regression for that. As we already know, we will use the train test split method from scikit-learn for getting the train and the test sets, and we will try degrees 1, 2, and 4. Remember always, degree 1 is linear regression. And we will plot everything to see what happens. This is an interesting case because for using the polynomial features in the linear regression model, what we have to do is to define a pipeline object in scikit-learn. A pipeline object in scikit-learn is basically to train different models or features, such that our data passes through that pipe and ends being fitted. However, the interface of the pipeline model is the same as the model object, so it's the same. After having our pipeline model fitted, we will round up the predicted sizes such that we have an integer value and we will compare those with the actual sizes. Let's see how that went. For linear regression, we got a really high true rate, but not 100% accuracy. However, for degree 2 and degree 4, we actually got all the cases right within one class of the sizes. This is great because we are having a really good model. Let's see how is the stability of that model because these models tend to be somewhat unstable. If we check, what we are going to do is fill the correlation matrix for our features. And what we can check is that our data is really highly correlated. This is known because basically x squared is just raising x to the square. So with the information of x, we have the information of all the features. With the information of x squared, we have the information of x to one-fourth and so on and so forth. So it is no surprise that this dataset is highly correlated. If we add just one outlier, check that for the 20th row where adding our outlier and doing the same thing, let's see what happens. Our models perform a lot worse. This is a clear sign of overfitting. Polynomial models tend to overfit a lot. Check how for degree 4 and degree 2, with adding just 1 outlier, we have a lot of false cases right now. This is really bad, right? So what we need is actually some kind of regression that is robust, that doesn't hear outliers so much. And this is where spline regression will enter.

Outliers Strike Again: Spline Regression as Local Regressor

As we have seen in the demo, sometimes polynomial regression is really good at the center of the data, but as we move to stream values, it is a poor performer. One of the reasons for this is that it doesn't have too many parameters, so it cannot adjust to this super complex function. Let's see what we can do about it. Let's go back to a known example. This is a cubic regression over the ______ true function we already know. As we can see, it's a really a really good fit. What we will do is add one outlier. Let's see what happens. Incredible, right? It is a completely different fit. And just because of one outlier. What we have witnessed is the nature of highly collinear regression. It is super non-robust. From here, we have two clear pathways. The first one is a stallion robust regression, which is a super interesting topic that we will not cover in this course. The second one is spline regression. The idea of spline regression is to partitionate the model space into smaller chunks of equal size. In one dimension it would be the real line into similar intervals. Then what one does is perform a polynomial regression into each chunk, providing everything sticks together. It works because each polynomial regression has enough degrees of freedom to also fix the derivatives in the borders. But how does it work in an example? Trying to predict the same function, we will compare a polynomial and spline regression with one outlier only. As we can check, the polynomial regression on the left is completely lost since it doesn't predict a single value. However, the spline regression on the right is local. The part of the regression that has outlier, of course, is nothing good. But that doesn't affect the rest of the regression. Therefore, we can check that we have an actual good predictor in the rest. This is what allows us to say in a way that it is more robust. Let's see how this one works in a demo, and later let's talk about how to compare both models.

Demo: Perform a Spline Regression

For this quick demo, we will understand how to perform a spline regression and in the middle we will understand how to extend the model pipeline with custom models. Let's go and do the spline. In the previous demo, we did polynomial regression and it worked great. However, the model ended up being a little bit unstable because when adding just one outlier, it didn't go so well. What we will do right now is adding some spline regression so we have some spice in life. First and foremost, as always, we will import all the necessary packages. And later we will have this auxiliary method that for some given knots, it will construct all thee spline polynomials. This is based in some optimization package that uses math, so we don't have to take a lot of care into that. And with that, we get to the first case where scikit-learn doesn't provide us a model. There isn't a spline regression model in scikit-learn. But what we can do is create our own. For that, it's super simple. We only have to inherit from the TransformerMixin class and we have to reimplement the fit and transform methods. For that, the only thing we will do is just give some knots, put those knots in some line, and then construct all the necessary polynomials. And then it's super simple to create your own model in scikit-learn. Later on, we will import all the data into all the preprocessing we did before, so no surprise here. We will reproduce the case in polynomial features, just to get that again. As we can see in the chart, the polynomial regression with one outlier didn't go so well. We can see that there is a lot of false cases in all the polynomials. However, the main difference here is that now we will do spline regression for degrees 1, 2, and 4. And let's see how that works. What we can check is that for degrees 1 and 2, there is actually a 100% accuracy. That is awesome. And why is that? Well, it makes a lot of sense because if the outlier is between two nodes, all the other spline regressions are going to go okay. And when we have this outlier between these two nodes, that's the only regression that cannot go so well. In this case, it went, but it may not happen. However, and this is important in case it will be only one case in all of those. That's the difference with the polynomial regression where it could be any number of cases because it's not robust and it affects all the regression, not just the local part. Another interesting topic and the final one, I might add, is that for degree 4 we have falses. What that means is that as we are having a high degree polynomial in a spline regression, which are a lot of polynomial regressions, we are having a high type 1 error, and that is what we are seeing, which in other words is overfitting. So all the contents from the both courses mesh together and make sense.

Model Selection: Let the Simplest Model Win

And after a whole pipeline one gets the final train model with its pros and cons. The only issue is that we usually don't just have one model, we have many. How can we choose? Model selection is an art. It has black magic and every person has its taste because mainly there are no rules. So what I will do is tell you what I think are a couple of ways of comparing models. Probably from different model families, like polynomial and spline regression. There are a number of different approaches to compare models. Adjusted R squared, use parsimony marginality, AIC and BIC, and best test performance, where the first three are intrinsic and the last one is extrinsic. Let's talk a little bit about each one. As we all know, R squared is a proportion of the variance explained by our current model. This means that it gets closer to 1, we explain more and more of the variance. However, of course, if we add infinity variables, we can easily end up with an R squared of 1. So statisticians came up with an adjusted R squared, which is basically the same one, but with a penalty given by the number of degrees of freedom that we have. For example, in this case, on the left, although the R squared of a quadratic model is not as high as the one of a degree 28, the adjusted one it is because it is a good model based on the number of degrees of freedom. Another good principle for model selection is marginality and parsimony. Marginality means that when a model includes a function of some explanatory variables as new variables, it should always include the original variables. What does this mean? That if in our model we include a term like X squared, we should always include X. If we have a term called X dot times Y, we should include X and Y. On the other hand, parsimony means when in doubt, always select the simpler model, which basically means that if everything is kind of the same, select the simpler model. The issue is, which is the simpler? In easy problems, it means less parameters to estimate. However, in complex problems, it means a lower VC dimension, a concept we will not tackle in this course. Okay awesome, but this is zero helpful since I need the metrics. How do I measure how parsimony is a model? And adjusting R squared is not always the best metric. So statisticians came out with an intrinsic measurement of the complexity and entropy of a model called AIC. And as Bayesian people are always in fight with frequency since the beginning of time, they came up with one of their own, BIC. In this example, we then again have again the degree 2 and degree 28 polynomials, we can see that the best model is the one is the left because it has a lower AIC, which measures the complexity of the model. It would be the same is we used the BIC, just another formula. We will check the formulas in the demos, but just understand that this is a good rule of thumb of an intrinsic metric. Up until now, we analyze intrinsic metrics, but they have one issue, they do not directly relate to the performance we actually want to measure. What do I mean? Let's imagine we are back again to the pumpkin sale past problem. We always optimize the training based on MSE. It is quite odd to later choose the best model based on a metric that is not MSE, right? That is the idea of extrinsic metrics. Use the metric we care about. Easy, right? We take a validation set, we measure MSE for both models, and we select the one that has a lower MSE, right? Well, it's not that simple, for two reasons mainly. The first and foremost is that if we do that, we cannot measure the final model performance in the validation set, since it would be biased by the selection of the model itself. The second reason, which is more important and more subtle, is that we get an estimation of the MSE that we only get for a sample of size 1. That's it. We have two random variables, the MSE for the model 1 and the MSE for the model 2. So in order to compare them, we must use all the tools from the previous course, and for that we need replicas. So we actually need a way of generating, in a statistical ______ way, more samples for MSE of each model. This is the magic of cross validation. Cross validation is a super interesting topic and works like this. Suppose you want to get an estimate of something, anything, and that estimate needs some dataset. In our previous case, it was the MSE for each model, however it can be the MSE for any hyperparameter, for example. The image to keep in mind is the following. The idea is to split the dataset into K folds of size k, such that short k times big K equals N, the number of data points. In the first iteration, we assign one of these folds to be the test set, and the rest is a training and dev set. Then you train your algorithm, in our case polynomial regression and spline regression, with the train dev set, and evaluate the MSE in the new test set. That is a randomly chosen fold, save this value. Then do K more iterations. In each select randomly one of the folds to be the test set, the rest of the training set and repeat. In the end, we will have K measurements of each random variable for the performance of each model. Voila. Now we can do a test and compare them to see if they actually defer. Magic, right? It's black magic. The important thing to keep in mind is that everything, and I mean everything, needs to enter into the cross validation. When I say everything, let me repeat I mean everything. If you dump one explanatory variable based on some data, that process is part of the training, and so on and on. One of the biggest mistakes people make is don't include everything in the cross validation because of time savings. After that, we get a nice method to compare models, but if we don't include everything, we get a biased estimator. To do a quick recap, we saw four main methods of selecting models, adjusted R squared, which is a penalty on the complexity of the model for the R squared, AIC and BIC, parsimony and marginality, and using extrinsic metrics and cross validation.

One or more of these actually can point in the same direction, so it doesn't hurt to check these methods. Let's see them in action and let's find that super model to the tax fraudulent people.

Demo: Comparing Models

And we got to the final demo of the module. We have two models, so we will compare both using AIC and adjusted R squared. Cross validation is one of the main takes here. In the meantime, we will learn how to implement cross validation in Python. And it's one of the main takes, since it expands to many, many, many places. So let's do it the right way. Let's do a quick model selection of which algorithm we should use for our size selection in the pumpkin sale for the IRS. And for that, first and foremost, as always, we will import the necessary packages, and what we will do, so I will do a quick pass through, is prepare the dataset and run basically the polynomial and spline regression with and without one outlier, as we have seen in the last two demos. And I will calculate for both the R squared and the AIC. For the AIC, I created a method, which basically calculates the AIC for a given model, so we can use that. And I will get to the interesting part, which is basically the comparison. For the model without the outlier, we can check and given the AIC criterion, we should use the polynomial regression because we can check that the AIC is lower, it's 1.27 against 2.73. If we use the adjusted R squared, we can check that also we should use the polynomial regression model because the R squared is 0.75 against the 0.49. What this tells us is that most of the criteria go the same ways, but we should still check all of them. Let's add one outlier to a training set and see how that will change everything. If we add one outlier, the AIC for the poly model went up from 1.27 to 3.22. Why? Because we now have a lot more uncertainty because a polynomial regression is not robust. Therefore, we have more entropy. However, you can check that for the spline regression, as we know from the theory it shouldn't be affected and it isn't. The AIC went from 2.73 to 2.80, which is almost nothing. So given the AIC criteria, we should now choose the spline regression if we expect our dataset to have outliers. If we use the adjusted R squared, we can check that the polynomial regression went from 0.75 to 0.35. Incredible, right? So, again, what happened for the spline regression is that the R squared almost didn't move from 0.49 to 0.47, because we are testing a non-robust model against a robust model. So now you've seen the R squared criteria, we should still select the spline model. It's really interesting because what we ended up checking is that if we expect to have outliers, we should use spline regression. But if we don't, or we are really good doing some data preprocessing, we should have the polynomial model, and actually, if we check, the polynomial model is the best of all if we don't expect outliers. It's really interesting to do model selection, and this is just a simple case. But from there, you can have tons of models and adding and decreasing degrees. So it gets a little more complicated, but it's really fun. With this, we end the demo for this module. It was really interesting and I think we got a lot of tools from that example. Thanks.

Summary

And just like that, another module that goes away. We ended regression for this course and we learned a lot. In particular, in this module, we learned about polynomial and spline regressions as special cases of multiple regression. We learned about bias variance tradeoff and how that adds regularization hyperparameters to get the best generalizable model possible. We understood how to linearize and use that for modeling complex relationships. And we learned how to compare any models, not just regression ones, using intrinsic and extrinsic metrics. I am super proud of us. And next, we will tackle the other big problem in supervised learning, classification. Let's go.

The Problem of Correct Classification

What Does the Boundary Look Like?

Classification is one of the biggest problems in machine learning. I think the reason for that is that this space was where neural networks started gaining more and more and more performance, up to the point to reaching human level performance. Over this module, we will start to analyze what does it mean to create a classifier and how to frame it in what we already know. Now, we are anthropologists at The British Museum and a new shipwreck was found near the coast of Virginia. It seems to be a transatlantic boat, very similar to Titanic, but totally under the books and transporting illegal refugees alongside common folks. In a spirit to fix and complete the registers, we are given the task of building a classifier of who may have survived and lives nowadays. Also, given features and previous studies, it appears that a classifier from Titanic shipwreck may be useful for this. So we will build one for Titanic and use that for this new shipwreck. Let's imagine we try to classify something. For the sake of the demo, we try to predict, given a person, if it survived the Titanic shipwreck. And for simplicity, let's imagine that we do that by predicting it by ticket fare. Based on the previous knowledge, we have that most first-class citizens were saved, at least that's what James Cameron got me thinking. So the scatter plot could look like the following. Where we can see that the survivals are in blue and the died are in orange. So what can we do? One idea we could have is let's make a linear regression, and then define some hyperparameter threshold, such that if a predicted value is greater than this threshold, we assign survived, and if not, we assign died. In pictures, it would look like following the threshold. Therefore, as we do our linear regression, we can see that we assign all fares above 100, kind of, to be survived, and all fares below that to be died. However, there are a couple of problems. The first one is that linear regression actually can output values way outside interval 0 and 1. Also linear regression doesn't output the probability, therefore we cannot interpret as a likelihood of survival. So what can we do? Here is where logistic regression comes into play. The idea of logistic regression is to apply a function to linear regression such that all results get between 0 and 1. This function is the logistic function and it's plotted here. As we can see from its graph, on the left all negative values go below 0.5 and go to 0. And all positive values are more than 0.5 and as they are bigger, they are closer to 1. The idea will be if the output is higher than some threshold, then we assign 1. We will interpret the output of this regression to be the probability of assigning 1. This simple idea will prove to be super useful in classification. The line that will split the dataset into the ones that we assign 1 and the ones that we assign 0 is called decision boundary. How does this look in logistic regression? Well, if the threshold, for example, was 0.5, the logistic function assigns 0 to negative values and 1 to positive values. Therefore, we will have A times X plus b equals 0. So the decision boundary in one dimension is just one point. In our case, like for example, 120. However, if instead of just using fare, we used fare and age, for example, our scatter plot would be the following. In this case, the decision boundary actually defines a line. But it can actually be of any type of shape. For example, in this case, this is a typical example of a classification decision boundary, and we also can see that in black we have a really good fit, but in green we have a clear overfit.

If You Need to Classify, Try the Star: Logistic Regression in Depth!

So now we know about logistic regression as a classifier. We added this logistic function and everything appears to work, but how do we train it? Training logistic regression is the same as training any other learning algorithm. We need a cost function, an optimizer, and return the hyperparameters. The only difference is that we already have one hyperparameter to tune this threshold. The cost function for logistic regression is called the cross entropy and looks like this. For example, here we are putting the loss when the true label is 1. We have another loss when the true label is 0, and we combine them. The idea is quite simple, we want a label that is actually 1 to be 1. So we make the cost of putting a 0 super big, and if the true label is a 0, we do the other way around. Luckily for us, Python will handle all the formulas. I just want you to know how it looks like. So to recap, we have our logistic regression model. It has two parameters, the A matrix, which has many parameters, and b, the intercept. And it has one hyperparameter, the threshold. In the training, we find the parameters used in the cross entropy cost function and using any optimizer we want. Since the cross entropy is nonconvex, there is at least one caveat. It may happen then from different initial points, we get to different minima. So we need to take extra care with the learning rate to not be too small. After training, we end up with a model that outputs the probability of classifying 1 and we assign 1, or not, based on the threshold. Later on with help of the dev set, we will tune the learning rate and the threshold, but one question arises, how do we measure how good this model is here? One good idea of measuring how good our model is, is by going back to what may happen. We have this 2 x 2 table where if we say what the model said and what the truth is. Imagine when we tried to classify, if we should trigger a fire alarm. In that case, a false positive would be that the alarm triggered, but actually there was no fire. This is not good, but not that bad. This is our old type 2 error. On the other hand, a false negative is super bad because there was a fire, but no alarm was triggered. This is a type 1 error. So it's easy what to do, right? Well, not so much. Imagine, again, we classify if we should trigger the alarm and we define that a good metric to be is the accuracy. That means all the good values against all values. Just as remembering, back to our fire alarm system, we go and we find, for example, that it has a 99% accuracy. That's awesome, right? We should stick with this classifier. Now imagine we actually have 0.3 % of cases with an actual fire. What would be the accuracy of the following classification system? The classification is disconnect alarm. That means never trigger the alarm. We would have 99.7 % accuracy. The issue here is that accuracy assumes that both classes are evenly distributed, and most cases it is not. So accuracy isn't such a big metric. In this period, two other metrics were born, precision, which is the percentage of true positives against all predicted positives and recall which is the true positives against all actual positives. In the example above, the truth table for classification to algorithm would be 997 cases, the truth was actually no fire and we never triggered, and 3 cases the truth was there was fire and we didn't trigger. In this case, the precision and the recall would be 0. So this is great because if we care a lot about predicted 1, if it is, we can aim to a higher precision. On the other side, we care about not missing too many positive cases, we aim for a higher recall, and how can we make that tradeoff? With a threshold. If the threshold is super high, for example 0.99, then we will only output 1 if we are super sure. Therefore, we will have a super high precision. An example of this is the correct on the keyboard that Google makes. It will only suggest a correction if it's super sure. On the other hand, if the threshold is low, like 0.01, then we will only output 0 if we are super sure. Therefore, we are having a high recall. It is an eternal tradeoff. But what should we optimize? In order to have one, and only one, metric, there exists a final metric, the F1 score. It is basically the harmonic mean between precision and recall and it has that formula. It ponders both, and this is the metric we will try to optimize. Let's see all this in action in a demo.

Demo: Classify with Logistic Regression

And we got to the demo. Here we will see everything in action. By analyzing the Titanic shipwreck, we will make a survival classifier with logistic regression, and finally, we will analyze metrics to see its performance. Hi, and welcome of the first demo of module 6. Here we will analyze the Titanic shipwreck, just to give an aid for this British ______ anthropologist like ourselves that we need to understand more of this new shipwreck. We will do that by creating a logistic regression classifier, but it's not only that, because to give a little twist, we will start seeing cross validation in action in a lot of places. First and foremost, as always, we will install all the necessary packages and we will load all the datasets into memory. So we can see that for training, our dataset is of length 891 and we have a lot of variables, the passenger id, if it survived, the age, the fare it paid, if it traveled alone, where did it embark, if it was a male or female, and if it was a minor or not. For the test dataset, of course, we have all the same columns, but 418 data points. The first thing to check in logistic regression is if we want to add all the variables. We can imagine that actually if we put all the three embark variables, that is kind of the same, right? Because if it hasn't embarked in S and if it hasn't embarked in C, then obviously it embarked in Q. So we have some kind of collinearity here and there. For that, we will use recursive feature elimination, which is an extremely important package from the feature selection in scikit-learn. Basically what it will do is you give me your model and I will do cross validation. Just removing one of all the features, so I put all the features, then I will remove one. I will do cross validation on all those, so I select the one that went worse, and then it will do iteratively until the significance in variance explained is actually significant. So how do we do that? Actually it's super simple. From the training dataset, we just define the X and Y. We define the model to be the logistic regression and we just say, hey we want an RFE model and here we will say please select 8 attributes. Later we will do some other cross validation to see if that 8 is optimal. And as a matter of fact, with only that, when we feed the RFE model, it will output the selected features. In this case, where the ones printed. Now, of course, you may tell me, but you hard coded the 8. Of course, so there is another object called RFECV, which does cross validation between the RFE objects and the model you input and the data, given the scoring that we are interested in, and with all that, the RFECV object will go one by one, adding one number of features, two number of features, three number of features, four number of features, and doing an RFE over the whole dataset. So just to give an example in your head, if we are setting it to 4, for example, we went to the cross validation in 4, that means that we start with 10, then we remove 1, then we remove the second, all that until we move 6. If we see the following plot, we can check that actually the cross-validation score, which is actually the ______ in this case, it's highest in 8, so the optimal number of features to have is 8. And with the RFE that we did before, we actually know which are those features, so we select them. If we do a correlation matrix between those features, let's analyze that actually they aren't so highly correlated. So awesome. The only ones that appear to be somewhat correlated are the embarking ports, but that makes sense. Continuing, what we will do now is actually create our logistic regression classifier and that also is super simple, we just use a train test split method that we always have used, we use the logistic regression model again, but now we fit with the logistic regression, we get the predictions and we get the probabilities. What we will output here is all the scores, which gave the ROC curve. This is a really interesting curve that I want to put a couple of minutes explaining. This is the ROC curve. So let's see all these outputs. What we got is that logistic regression accuracy is 0.78, which is really good. The auc is 0.839, which I will explain right now what that is, and another measurement of loss, which is the log_loss, is 0.50, which doesn't tell us that much. The ROC curve basically tells us moving the threshold of the logistic regression, how much recall versus precision will work. So, for example, in blue, we have that actually setting a threshold of 0.071, that will guarantee us that we have a recall of almost 0.962, which is awesome. The only thing is that we will have a precision of 20%. So as always, you know, this is actually a tradeoff. It depends on what we are looking for. This is the business case we have to check now. The ROC curve is basically this curve that gives you, for each of the thresholds, how recall and precision will work out. The AUC is the area under that curve. The perfect AUC is basically 1 because actually the perfect classifier would be 1, actually with some threshold. It is all in the upper left of the chart where the false positive rate would be 0, therefore excellent prevision, and the recall will be 1, therefore excellent recall also. But that's the ideal right? So we want a high AUC, and that is another measurement of how good our classifier is, independent of the threshold. Now that we have our classifier and we know the ROC curve of this classifier, let's analyze its performance. For that, what we will do is instantiate again a new logistic regression object, but we will use the cross_val_score method from scikit-learn. What this will do is do cross validation with different scores. In this case, we will do it with accuracy and with a negative log loss, which is the measurement, the cross-entropy log loss of the logistic regression. Also, we will use the AUC. This will give us, also, which is the actual AUC of the logistic regression classifier. If we can see with cross validation, the results vary little bit from the ones above, but that makes sense because the ones above is actually a sample of size 1 and with cross validation we have an unbiased estimation. So the accuracy by cross validation of our logistic regression classifier is 0.80, the log loss of 0.45 and the auc 0.85. So they are kind of similar to the above results. That means that also our classifier is quite repetitive, which is also another good quality. Finally, let's see which of the scores is actually the optimal and if we should or not add some regularization. For that, we will test with a regularization parameter C that will go from 3 to 0.1 in descending order, and we will try with the scores accuracy, AUC, and negative log loss. For that, we will instantiate from the model selection library in scikit-learn the grid search cross validation. What this will do is the perfect tool to do some hyperparameter tuning, because it will do cross validation, but it understands that it will get some dictionaries and it will have to test all of those. Or maybe not the grid search one, of course, it will do. And then after that, we will have all the values for all the combinations and it will select the best. So in our case, as we are trying to use some of the scorings, what we will do is we will associate the grid search CV object, of course, the model is the logistic regression, and check that we are going to say that after that, when it selects the best, it will refit by optimizing the accuracy. This is really an interesting part because then at the end we will have a model that is optimized by accuracy with the best scorings and regularization parameters. Then we fit, and this may take some time in your computer. In mine it took like, you know, 2 minutes. And let's chart what we have. What we got from here is that the best parameters is 2.80 for the regularization parameters, and the best score in accuracy was 0.8069 for that regularization parameter. Of course, it will be a little lower than the one above because we are including regularization, but that is not a bad thing because now with regularization our model will be more generalizable. And we can see for the multiple scores how it went. So in this chart, we can check that for regularization parameters 2.8, the score in accuracy is around 0.85, for regularization parameter 2.8, the score in AUC is around 0.85, in accuracy is 0.81 and in log less is 0.45. And actually, this was kind of the same among all regularization parameters. Therefore we can select any of those as the scoring, and it will work. I hope you had a lot of fun because in this demo not only we used logistic regression, but we do feature selection with cross validation and recursive feature elimination. We did some hyperparameter tuning with cross validation, as we see on the theoretical part of the module, and finally, also we did some cross validation to analyze the performance. So we did a quick overview of all the parts of the pipeline from before the model to after the model.

Classify Multiple Categories: One vs. All Classification

That is awesome, but let's get honest. I want to predict multiple things, Axel, how is this useful? Let's check a matrix. Suppose we have the following case where we want to predict embarking port based on age and fare, if it makes sense at all, what do we do? The strategy we will take is basically one versus all classification. What is that? For each category, I create a logistic regression classifier in that category as positive. After one pass, we would have k classifiers, one per class. When a new data point comes, calculate the probability of each classifier. After this, we would have k probabilities, one per class. Assign to the biggest ______. How does that work? Let's look at an example with Titanic. First, we check how many categories we have in embarked. There are three. S for SouthHampton, C for Cherbourg, and Q for Queenstown. Next, we grab SouthHampton and we create a logistic regression for classifying it to assign SouthHampton. For example, for this logistic regression, if we take fare 200 and age 20, then it would predict the probability of being embarked in SouthHampton to be less than 0.5, therefore assigns negative. Next, we grab Cherbourg and we do the same thing, check that the decision boundary for each logistic regression will be different. And finally, we do the same with Queenstown. After we have the three classifiers, we get a new person, it pays the fare of 150 and is 50 years old. What do we do? We check the output of the free logistic regressions in this input. For example, in our case, logistic regression for S is 0.65, for C is 0.85, and Q is 0.12. As the highest is C, we assign that person to be embarking Cherbourg. This is how person subclassification works and this is a way of classifying multiple categories. This approach has its drawbacks, however. Mainly when there are multiple classes, we start to have highly skewed distributions for each class, there are way less positives than negatives. When having multiple classes, the chance of type 1 error goes higher. This is why when there are many, many classes, maybe more than four or five, we can pass to another multiclass classifier, like the ones at the end of the module, or go to another extreme with neural networks and ______ for classifying. However, that is topic for another course. If we make a quick wrap up, we learn about how to take a binary classifier and make it classify multiple classes. We made an actual example with Titanic embarking port and learned that when the number of classes goes big, as each classifier is one versus the rest, we start to have skewed classes. Let's see this in action in a demo where we will classify flowers. After that, we will learn a native multiclass classifier.

Demo: Multiclass Classification with Multinomial Logistic Regression

And we got to another demo. However, we will abandon for a time Titanic, an awesome move, to go to flowers. In particular, we will perform multiclass classification via one versus all classification and we will analyze multinomial logistic regression as another option. Hello and welcome to the last demo of module 6. Here we will classify flowers from the iris dataset, and with that we will try to win a trip. So as always, first and foremost, we import all the necessary packages and then we will load the iris dataset from the dataset's API in scikit-learn. We will use only two features, just to plot the decision boundary, which is also _____. Of course, if we would want a full classifier, we would use all the features. We can do that later. So what we will do is try out two types of multi-binary classification. One is multinomial logistic regression, which what it will do is it will put all the necessary features into our logistic regression, but it already knows that there are more than two classes inside it. So we don't need to take care about anything. If we fit this classifier, then let's see how it went. We can check that actually it does a pretty good job, right? So we know that there is one class that it did really well and then it has some difficulty actually classifying between the iris versicolor and the iris virginica. But that makes sense because if you can see, they are kind of mixed. So actually, just as easy as that, we did a pretty good job. What we will do right now is try out how one versus all classification works in scikit-learn. For that, we can see that actually it changes only one line, instead of saying multiclass equals multinomial, we'll do multiclass equals OPR. That's it. The rest of the code is simple. What this will do is it will generate actually three logistic regression classifiers, one per class of one versus all, and then it will output the class of the highest project as we have seen in the module. And actually, if you check the classification, it is somewhat similar. There is a little more less separation between the iris versicolor and the iris virginica. But actually we did super well. This demo was really fast, but the only difference between multiclass classification with logistic regression and binary classification with logistic regression is just one string, the code is the same. So we don't need to take care of anything. I just wanted to demonstrate how easy it is to expand to this new knowledge. Thanks all.

Summary

Classifying is a super exciting world. This is just the tip. Over this module, we started analyzing the problem of classification. We learned why regression isn't the best fit for classification. We learned about logistic regression and how it detects a problem, and we discussed about performance evaluation metrics for classification. In the next module, we will go deeper into the world of classification. I will present you the other two main paradigms of classification, robust and Bayesian classification. Hop on.

Large Margin and Bayesian Classification

Maximizing the Actual Information: Bayesian Attack!

In the previous module, we learned logistic regression, and it was awesome. However, logistic regression is to classification as linear regression to regression. It's the first model. As such, it suffers from two main factors, assumptions and outliers. Let's check those and what to do. Let's take a look at what can happen and what can we do. One of the issues with frequentists models have is that they have a lot of assumptions. One of the most common assumptions is normality, homoscedasticity, and that we have a really large sample to use central limit theorems. However, this is not always the case. What can we do? The usual answer is get more data, man. What I want to do in this module is to show that there are other alternatives in statistics, machine learning, and life, it's just a matter of having a wide perspective. One of these approaches is Bayesian approach. That basically says I will use almost everything I have as this is the truth, and based on that I will decide the next step. Check that this is different from there exists a North Star with a node, so we continue going forward until we say we've found it. One of the main classifiers people use is Naive Bayes. So I wanted to present it to you. We will use Bayes Theorem. Do you remember that is the one in the formula? This is what we will use. Let's check in an example. Let's imagine we are on a scavenger hunt. So we take a past data chart, we check if a known record of traveling for that area, the temperature of the water, if it's a strong current, the ocean, and if we found a treasure, and so on. So have a new place that doesn't have records of known travelers, it has a low temperature, but it doesn't have strong currents and it's in the Indic. Are we going to find a treasure? In frequentist world, we would need like, I don't know, easily 1000 treasures to actually have enough data to say are we going to find a treasure. But suppose we only have 20. The idea of Naive Bayes is that the Bayes Theorem to the stream. So we say the probability of finding a treasure given that is basically to multiply the probability of how each value, even that we found the treasure, divided by the product of each value restricted to what we know times the probability of the treasure. That seems like a lot of formulas. But the idea is that now we have some probabilities that we know how to calculate, especially Python, and why are there are so many types of Naive Bayes? Well it depends on how we calculate those probabilities. If we calculate those counting cases, it's the Standard Bayes. If we treat it as a sampling problem, it's Binomial Bayes. And if we think every probability is from a Gaussian distribution, it's Gaussian Bayes. However, the main idea is this one, and the good one, we will have a spoiler, almost no assumptions. Only one, and every important, we have used above without telling so. Awesome, right? This neat idea can help us make a great classifier, since with this probability we will compare to a threshold and, bam, we have it. Let's see this in action with a demo.

Demo: Multilabel Intent Classification with Naive Bayes

In this fast demo, we will go and make an NLP task to actually get an intent classification. This is a super famous problem in machine learning. What we will do is we will implement Naive Bayes for this and we will get a glimpse of doing NLP in the middle. So quick and fast. Hi, let's do some intent classification. This is a really interesting NPL topic. And I am really excited to bring some NLP task to you, even though we will not be able to cover all the necessary parts to understand it fully. What classifying intention is actually from a given text, have what it actually meant. It can be at the level of this is a question, this is some type of code, this is a description, or it can be up to normal text, for example, I want to pay my bill and knowing that is my bill. This is a really interesting topic and one of the star problems in NLP. First and foremost, as always, we will import all the necessary packages, lower data, and do some preprocessing. This is not actually all. Also, we will create some helper maps from all the categories we have, which in this case are only three, but it could be any, to ids. And what we will do is the part, the NLP magic. The NLP magic is using the tfid vectorizer. What this will do is inside it has a pretrained neural network that it will partition our text into ngrams, which are basically where n is the number of characters and it will split our text into how many features, depending on the number of characters it has, so, for example, hello, and we are doing three grams, it will be hel and lo. And with that it will pass through an encoder, which is basically a function in neutral networks that will create some features for us, given that text. So what we will have in the end is basically a function that, given some text, it will output all the relevant features. These features can be things like it has one space, it is actually talking in English, it has this many stop words, etc. They are relevant to the text itself, given our training set. With that, we can check that, for example, with the features.shape, we can see that we have 6000 texts, but instead of just having the text, we transform that into 1322 features. Of course, if I needed to actually go deeper in this topic, we need to cover NLP and neural networks. And that's out of scope, but I only needed this part such that our test could actually make sense. Later on, what we will create is the wrapper predict function that just basically predicts given a classifier and what we will do is use the train test split method, as always, and we will transform first all the questions in the training set to its feature representation and with that we will instantiate the object Multinomial Naive Bayes. What that will do is just do a Naive Bayes, expecting there are more than two classes. The only little thing that we need to take into account is that for creating our features, we also input not only the text, but the number of times it appears. For that we're using the count vectorizer. With that, we have already a classifier. We applied to the test set and we could check that, for example, for the row 263, we are predicting that it was a description. So let's see the accuracy. Let's check how the predictions go with the test set. And we can check that actually our accuracy is 0.97, which means 97%. That is awesome. So we did really good. Of course, the big magic is how it transforms from text to features. That is like another course, but given that, Naive Bayes does a really good job giving features and classes to actually do a classification, and it is really simple, right? Awesome.

Large Margin Classification: Outliers!

I'm going to the other side of the world, both logistic regression and Naive Bayes have an issue, how sensitive they are to outliers. So let's check a little bit of robust ML. In this video, we will learn about support vector machines, or SVMs, that are robust classifications. What does this mean? Let's check the following decision boundary. Of course, we have two classes, right? However, we could have the green or the yellow decision boundary, which one do we choose? The idea of SVM is to find the yellow line. How? Let's see. The idea will be to try a boundary, then find the closest points to the line and calculate the distance. We call that the margin. What we will try is to maximize the margin. If we do that, then what we will have is a robust estimation. I know that's a long shot, but you will have to trust me, maximizing the margin is the best we can do. And how do we do that? Let's go back to a logistic loss. What this did was to give a huge penalty when the probability of x was assigned to the wrong label. How? Well, if we say zed to be A times x plus b, then the idea was if zed is below 0, then the logistic loss of zed will be below 0.5 and the loss will be super big. However, we did not push for having super big values of zed, just positives. This did not push for the margin to be large enough, it was just in the writing giving the correct answer, which in turn, sounds like overfitting, right? What we will do in SVM is change that loss function to one that pushes the result a little harder. If we try this new loss for once, it is super easy since it's linear. So it will be super fast to compute, but also, it will push to not only have positive values of zed, but also values bigger than a threshold. Here we set to 1. In this way, we can ask for a higher margin because we have to be really sure of a label to assign a 1, but also we want to prevent big values of coefficients since that also goes to overfitting. So what SVM will do is to have a new cost, which is basically a hyperparameter C times the SVM loss we just saw plus the norm of the vector of the coefficients. So if the coefficients get too high, the cost will also be too high. In this way, also the term that goes with C will push for a bigger margin and the other term will actually push for a high regularization and low variance. This is why SVM is so famous, because it attacks both problems at the same time. So how does that look in the end? When C is not that big, we get the green line. But if it is too big, then we have the cyan line. So way to go. In this way, SVM clearly creates a decision boundary by simple geometry. Note, however, there are two things to take into account, the first one is that this SVM will only be able to work if the data is linearly separable. That means that there exists that margin. The second one is that we only find the boundary, we don't find probabilities, and by the way, finding those are a pain with a fivefold calculation. So SVM may not say as the probability of an event. It will just give us a good margin. Let's attack problem one with one of the biggest wins of ML, the kernel trick.

Passing from Linear Boundaries to Nonlinear Ones: Kernel Trick

There is one simple trick that allows us to classify nonlinear boundaries. This is the kernel trick. As you may have noticed, a lot of the times in machine learning, we do a technique for linear cases and then we add tricks for the nonlinear cases. Let's imagine we want to classify the following data. In SVM we likely fail as we saw it, since there is no line that can separate those classes, right? However, imagine that we include as features for SVM, X1 squared and the Y label, which is the X2 squared. What can happen? Then we can find with SVM that red circle, since it is X1 squared plus X2 squared equals 1, which is a linear equation with the new features. So actually, SVM is way more powerful than what we thought. This case is actually "linearly separable". In this case, the decision boundary is quite complicated. Now what we could do with the SVM trick is that actually we could add to the SVM more polynomial terms, and that will work, however, it gets quite messy, right? Couldn't there be a way that we can easily and with our features find a nice boundary? The kernel trick gets the previous idea of using another set of features, but adds the idea of then being another thing. In specific, the idea is let's define landmarks, l1 to lm. The decision boundary found by SVM will look like the following, a theta vector times F for all the new features plus a value and intercept that has sub zero equals a value c. Where, again, F is the matrix of features. Up until now, we used the x values for features, but it doesn't have to be that way. The idea is for the new features to be a similarity function output of a value x to the landmarks, and that similarity is called the kernel. So what this will do eventually is generate a neighborhood of points surrounding the landmarks, where F1 will be closer to 1 and the rest will be closer to 0. In pictures, we have the following diagram. Where there are green stars are the landmarks. The green dots are points that are close to any landmark. So it is assigned to my positive class in the end calculation because they get a high score, and the red dots are the ones that are far away from all the landmarks, and therefore will get a low score. In this way, you see these new features from kernels and SVM, we can find a boundary that actually is just the surroundings of the landmarks with a high margin. A thing we would have lost in the case of adding more polynomial features only. Neat, right? How can we choose those in our case? Well, of course, we choose our training points to be the landmarks and we pull them in two ways, the positive cases get a +1 landmark and the negative cases get a -1 landmark, and voila. The only question remaining is what kernel should I use? Well, there are two or three known kernels. The linear kernel, that's normal SVM, the Gaussian kernel, also called RPF, or the sigmoid kernel. It depends highly the case, but one can later research depending on one's problem. To do a quick recap of these videos, we learned what SVM is and how it tries to maximize the margin. We learned that by doing this, it makes itself a robust estimator. We found how adding features to SVM, we can create a nonlinear decision boundary and how with the help of a kernel trick, we can make those simple without overfitting. Awesome, right? We already covered the three main types of specification, frequentist, Bayesian, and large margin. It's time to go to unsupervised learning, right?

Demo: Classify Iris with SVM

Whoa, whoa, whoa. Wait, wait. First, let's do a demo, right? Of course I wouldn't introduce a big concept like SVM without the demo. Please, you know me. In this demo, we will reclassify the iris dataset but perform it with SVMs and compare it with logistic regression. Bon voyage. And we got to the last demo of this module. What we will is actually really simple. We will do the same thing we did for the last demo of module 6, which is to classify the iris dataset, but instead of using logistic regression, multinomial, or ______, what we will use is SVM with different kernels. So we will use the kernel trick to see how actually now we can predict some nonlinearity. For that, and always, we will import all the necessary packages and later we will have this method called make meshgrid. This is a common method that scikit learn has somewhere in their examples and it's really useful to actually get these 2x2 charts that have all the classification. Also we will have a sample method that basically predicts the class and gets the contour for us. All of this is already given by scikit learn, so it's awesome. We reloaded the iris dataset, we will get the first two features, just as before, but here we will actually instantiate four models. One will be a linear SVM, so basically SVM with regularization, and we will input the regularization parameter as 1. Then we will have a linear SVM without regularization. Then we will have a Gaussian SVM, we are using the kernel trick here, and then a polynomial SVM. Let's see how that went. We'll plot the contours for all the models and we can see that actually the SVM with and without regularization kind of go in the same place. So it makes sense actually. However, the Gaussian SVM and the polynomial SVM using the kernel trick, have nonlinear boundaries, so this is great for us because we can start predicting all these nonlinear boundaries that make sense for our problem. In this case, for example, the Gaussian SVM did really good because we could split a little bit more of those intrinsic classes of iris virginica and iris versicolor. With this, we can give a nice wrap up to module

7, which actually was small, but it was powerful because SVM can give us a lot and it asks for so little.

Summary

This was a great module. Quick and fast. I enjoyed it a lot because it's one of my favorite topics. Most of all, this is our last video in the realms of supervised learning. So say goodbye to those pretty labels, since they are not coming back. Before abandoning the ship of supervising, let's do a quick recap of what we learned. We learned about the other two approaches to classification and when they are useful. The Bayesian approached, we used Bayes Theorem to get a classifier with almost no assumptions, and we used large margin theory and geometry to have a classifier that doesn't output probabilities, but gives robustness. Finally, we expanded on that idea with kernels to get nonlinear boundaries. With all these concepts we are the masters of classification. One nice topic we did not cover of time was neural networks. So that goes to another course. For now, we have one last module ______, unsupervised learning. Let's see each other there.

The Subtle Art of Not Needing Labels: Unsupervised Learning

Distance and Covariance Matrices

Over all the course, we've been having annotated data, and our task was to give the next prediction. This was awesome and we learned so much. But now it's time to switch tasks and get out of the comfort zone. It's time for some unsupervised learning. This module will finish with one super demo. We will in the core of cancer research and we want to assess how the mutation in the DNA are affecting the protein expression landscape in breast cancer. Genes in our DNA are first transcribed into RNA molecules, which then are translated into proteins. Changing the information content of DNA has impact on the behavior of the protein, which is the main functional unit of cells, taking care of cell division, DNA repair, enzymatic reactions, and signaling, for example. In order to help understand cancer, we will use everything we got from this course to get a better understanding of this dataset. Most unsupervised learning techniques try to find internal structure of the dataset. The first step to do this is a little field trip to understand distance and covariance matrices. In all this course we have a feature matrix, X, which was N times M where N was the size of the dataset and M was the number of features. Therefore, we can imagine it as M rows being samples of M different random variables. For example, in the following chart we have three random variables, each with a sample size of 2, temperature, humidity, and altitude. The first way of understanding this matrix would be to run the descriptive of the previous course to each of these random variables. However, as we now know, there are relationships between these variables that are hidden. One way of analyzing those is with distance matrices. What is that? In the following picture, where d sub i and j equals the distance between variable i and variable j. So, for example, d sub temp, humid will be the distance between temperature and humidity. How do we calculate this matrix? Well, there are multiple ways. For continuous random variables, we have Euclidean matrix, Manhattan distance, Mahalanobis distance, and Likelihood ratio. For categorical random variables, we have Chi Square distance and Jaccard distance. The ones we will check more are the first ones. Basically this matrix are functions that map a pair of variables, sample to a number. This number has the property that if you are way different, then it is bigger, and it is always positive. Now as always, there is a catch. If we standardize our dataset, then the distance gets broken, because standardization doesn't respect most metrics. So we must be careful. For example, here we have some dataset of economic variables for Latin American countries. When standardizing Argentina, instead of being closer to Peru, Columbia, Mexico or Brazil, it ends up being closer to Chili, Uruguay, and Venezuela. So which metric should I choose? It depends on the variable. If the variables are continuous, have no 0 values, and are in the same scale, Euclidean or Manhattan are great. If the maximum should be when there is nothing in common or we want to see similarity, we use Bray Curtis. For binary data, we use Bray Curtis or Sorensen. And when we want to analyze occurrence of events, we can use Jaccard. That was the distance matrix. This will be useful to understand, for example, to which cluster should the point be based on these metrics? On another topic, we now know how far things are. What about how dispersed they are? This is where the covariance matrices come into play. This one has a niche point that covariance of one variable with another. Which is a way of knowing how associated they are. Why is it useful? Because we will need it in many, many, many unsupervised learning applications. For two reasons mainly, the sum of the elements in the diagonal is called the generalized variance, and it will be our measurement of how much variance are we explaining. Remember ANOVA always. Also, it's eigenvalues and eigenvectors will determine which linear combinations of variables explain the intrinsic structure the most? If you don't know eigenvalues and eigenvectors, don't panic please. It's just a mathematical technique software will solve for us. To make a quick catchup, we started to check on how to start understanding the intrinsic structure of our dataset. We learned about distances, which are key having different ways of seeing our data. We discovered distance and covariance matrices that will be useful later, and we saw how a standardization can be harmful in these cases. With all this, we are super ready to check out our first application, clustering.

Clustering: Hierarchical and Non-hierarchical

Clustering is one of the most important applications of unsupervised learning. Let's un-mystify it. Why is clustering useful? Because applications from taxonomy trees and find species, to understand better metabolic diseases, to make some astronomical analysis and understand better how things like stars, planets, and meteors behave. The basis, it can be applied in so many places that it doesn't have almost any assumptions, and it bases everything in internal structure of the data. In clustering, what we want is basically that image. We want to split our data into clusters so that those could be the labels. In a way, this can be done beforehand to find labels and then have annotators annotate. One more application, it is endless. And how do we define a cluster? Well, an intuitive idea is that the distance between each data point inside the cluster is smaller than the distance with data points from another cluster. So we need a distance metric first. And that will vary immensely how we cluster. Check the picture. The algorithm is the same. The difference is how we measure distances. Once we have a metric, then it's as easy as starting somewhere and seeing which are close to me. There are two fundamental techniques for this, hierarchical and non-hierarchical clustering. In hierarchical clustering, which is also called agglomerative clustering, we have elements in a nested way, like the image. We start first cluster with the two elements closest to each other. We calculate the distance between this cluster and the other elements. Here we will define what is this distance. We will create a new cluster or append an existing one between two elements of least distance. We append if the new cluster as an element is closest to another new element. And we continue. Easy, right? We end up with this picture, which is basically a dendrogram. And how do we measure distance between clusters? Remember we can imagine a data point as a cluster of size 1. This is the linkage. In simple linkage, we measure distance by the closest elements. In complete linkage, we measure distance by the farthest way elements. In average linkage, we average out all the distances between elements from one cluster to the other. In centroid linkage, we will measure the distance between the center of each cluster. In ward linkage, we measure the diagonal of the covariance matrix of merging the clusters. So what we minimize is the variance of the clusters. This is my preferred method. So in the end, we get the dendrogram of how everything links up to a unique group of all. From here there are multiple ways of generating clusters. One usual way is saying that the cut line is where we reach 50% of the total distance. Another way is where we see a big distance jump in the dendrogram. The final one is where we have clusters of equal size. This is less preferred since it may generate artificial clusters. The other big way of doing clustering is via non-hierarchical clustering. It is called like that because we don't create a hierarchy to append to clusters. The most famous one here is called k means, and that's the one we will learn today. K means is, we start with k randomly chosen landmarks for starting our clusters. We assign every element to a cluster based on minimal distance, whichever we choose. After this step, we have k clusters, but they may not be optimal. We calculate the centers of each cluster. This is done by averaging them out. These will be then new landmarks for a new iteration. And then we continue until there is no reassignment in step 2. Done. In an example, we check that we start with all the data points in the first picture. Then we select randomly three landmarks. These are the starts of my clusters. Check that we assigned each point to its closest landmark. We end up with three clusters, green, brown, and red. Later, we recalculate the landmarks to the new crosses. These crosses are the centroids of the clusters. Now we would reassign. For example, the data point above the red landmark will now be assigned to red instead of brown. Therefore, we continue with each data point. In the end, we get the final picture where we have the three good clusters. They are good because there was no reassignment. But this has a flaw. We start with k landmarks. How many clusters do we want? How do we define k? There are three main ways of finding this out. The first one is using the elbow method as the image says. Basically, measure the sum of the diagonal elements of the covariance matrix of the clusters after clustering for several k's and select the one that forms an elbow. The second one would be to run hierarchical clustering, check the dendrogram, and find out yourself the optimal cut based on what we've seen before. The final one is use theoretical knowledge you already have for this problem. For example, you already may know that there has to be, I don't know, five groups. Some final remarks on clustering would be try always several distances, it may surprise you how much they can differ. There are many, many more non-hierarchical clustering methods. It depends on the geometry of the problem. And watch out for clusters with one or two elements. Those could be multivariance outliers. And regarding outliers, this one is a biggie, but I will tell you the reason. As we grow in both features, what happens with our models in supervised learning, for example, is that variance increases. We know this now. Therefore, we are more likely to overfit. If we have outliers in our dataset, this means that our train model not only will overfit, but also will have a high bias, so terrible. A way of checking outliers, we kind of mentioned during the course, but now we formally see it, is by clustering and check for one or two data point clusters, or to take the Mahalanobis distance of one point against all the other points and check if this value is over five. Five is the threshold people normally use, but it can be any number. So another application of clustering, it also helps us to find outliers. This is rock and roll.

Compression: PCA and CA

Changing topics altogether, let's do a quick overview of another big topic in unsupervised learning, dimensionality reduction. The idea of dimensionality reduction is simple, given a dataset with n variables probably correlated, find a new dataset with p variables that explain the same and have a way of going back. And why is this useful? Well, many, many ways. We can compress data to be able to send it easily. We can use this technique before any supervised learning in order to make computations faster, and it is cool. And how does this work? The idea is that given the original variables, X1 to Xm, the algorithm will find new linear combinations of the variables called C1 to Cm, with the following characteristics. They will be independent. They will be ordered from first to last by how much variance they explain. More on this later. They will resume the information of the original m variables. And how does this work? Basically we will find the eigenvalues and eigenvectors of the covariance matrix. These eigenvalues will represent how much variance it is explained by the eigenvectors. So we will order the eigenvectors and name them principal components. One thing to take into account, as different variables have different scales, we need to do feature scaling before doing PCA. In this example of this iris dataset, we can check that applying scaling led to clear designation of the labels. So, for example, imagine we have four variables. We run PCA and we find the following chart where we will have four principle components. If we check, we can give the first two principal components and retain 98% of all information, but now in two dimensional. Now PCA offers the chance of doing a B plot, which is a way of seeing how the old variables project in the new ones. The following is a standard B plot and we can see that with principal component 1, we explain almost the same as the variables 4, 3, and a lot of 1. So they were super collinear. The variable 2 and part of variable 1 are more different and split their information within principal component 1 and principal component 2. In this super easy way, we ended up with two variables instead of four that we can even scatter plot our dataset and continue from here with, for example, clusters, and using those clusters as labels we can have a nice classification problem. So we start to have this pipeline of models. Awesome, right? Some closing remarks on PCA. One thing is this is not a statistic test. We are not doing inference. We can use PCA to find new variables that we can use in other techniques where collinearity was an issue. Remember in regression where we said we could drop a variable or have a new variable based on linear combinations of the collinear ones? Basically I was saying we would do PCA beforehand, before the linear regression. However, we must always try to use the original raw data at least once to see if they work before jumping to PCA. I have seen many times machine learning developers pipelining PCA before doing an algorithm without even trying if the original data works. Another thing to take into account is that outliers are really bad for PCA, so cleaning beforehand will be useful. And finally, the number of principal components usually set by the amount of variance we want to explain and the complexity of the dataset. And why is this video named PCA and CA? Basically because if we have categorical data, CA does the same. The only difference is we will use chi square distance instead of other distance, the components are extracted not by variance explained but by percentage of lack of independence explained, and in this way we can end up with components that are independent. All this will make up the correspondence analysis. As a nice catch up of PCA and CA, in PCA we optimize for the variance explained. We end up with principal components that are independent and orthogonal. We use the covariance matrix, so standardization is a big thing here. And we use it in continuous variables. In CA, on the other hand, we optimize the inertia. That means the association between rows and columns in a double entry table. This means that we optimize by lack of independence. We end up with independent components that are not orthogonal. We use the chi square distance matrix, not the covariance one. And we use it for categorical values and double entry tables. Let's see PCA as an example of both in action in a demo.

Demo: Perform PCA

In this demo, we will shortly perform PCA on the iris dataset. We will analyze how different distances affect PCA and we will analyze how standardization affects PCA. This is our last single topic demo. So I hope you enjoyed it. What we have left is to analyze cancer with all we have learned. And now, we will use PCA for a quick visualization. As we already know, PCA is great for many things. We can use it to visualize a dataset better. We can use it as the first part of the pipeline of models to actually have a better classifier with less complexity or without needing regularization. Also, it is great for getting the intrinsic pattern of our data. In our case, we will get the iris dataset and up to now we always use only the first two features. Remember, now we will do a three-dimensional contour plot, but first we will pass through PCA to see if we can get a good visualization of our data without actually needing to just select the first three or three at random. For that, first and foremost, as always, we will import all the necessary packages. We will load the iris dataset, and now, for the first time, we will not select only the first two features. Later we will instantiate the PCA object with three components because now we will do a 3D contour plot, and we will fit with our data. Finally, we will transform our original data and put it in the same variable to transform data with three components. This is actually the projection from the original ten-dimensional data to the compressed three-dimensional data. And we will plot that. We can see that actually we have a really nice plot where we can see if, for example, the first component clearly could separate setosa, versicolor, and virginica. We did a really good job. You can imagine that if we input these into a logistic regression, we actually will get a really nice classification without actually needing some nonlinear trick or some kernel or whatever, just by plugging first PCA.

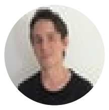
Demo: Breast Cancer at a Glance

It appears only yesterday we started this course and we got to the end. It's incredible, right? In this demo, we will see how we change supervised and unsupervised learning to analyze breast cancer. More, more, more could be done and said, but this is a nice wrap up of the course and I hope to see you again. For this demo, we will inspire in this excellent Kaggle kernel. Hi, welcome to the last demo of this course. Incredible, right? It was so fast. In this demo, we will use the excellent Kaggle kernel from Pete Bleackley to cluster breast cancer proteins. This is the exact same kernel that you can find online. I dare not touch it because I think it's really, really good and I want to illustrate the points of what he thought. So first and foremost, as always, we import all the necessary packages. Peter created this reduced dimension class, which basically does PCA. So we have a PCA class for that. He didn't like the PCA class that actually came from scikit learn because you had to input the number of components, and he wanted to learn that too. Also, he has this big class where he has the methods that we will use, principal components to get the dendrogram, and so we will get the hierarchical clustering. The cluster proteins, where we will get a k means clustering. And finally, it will cluster the patient activity and it will train some clinical models. For that, it will instantiate basically a lot of logistic regressions. If you check, also it instantiated a LassoLars regression. That makes sense when the output is actually an integer with some order. So it does make sense. It's a lot like logistic regression, but when you try to classify, for example, age, which actually has an order. Later on he gives some visualizations. With all of this, let's see how the dataset behaves. The first thing he did, actually, is get the principal components to reduce the dimensionality to two dimensions, and then to have a dendrogram. From here, we can see that actually these proteins cluster into eight nice groups. How did he get that eight? Well actually, we got that from the 50% rule in the maximum variance. Let's remember that the hierarchical cluster, and there are many ways to actually create the clusters, and one of those is actually get the middle of the table, which is the 50% of the maximal distance. When he used that number 8 clusters to plug it into k means, we got the clusters of all the proteins. This, as we know, basically needs the number of initial landmarks for the k means clustering. There are multiple ways of getting that number and one of those, and the most used one, is using hierarchical clustering. Later on, with that, he got and clustered all the patients given the activity, and that's really interesting because always the historical analysis was done the other way around. For that, he did a PCA projection of the patients into the cluster activity of the proteins to try to classify how given in which cluster of protein you are, what happens to you. Usually it was done the other way around. And later on, he used that to train a lot of classifiers and we will get all those realizations of what that can actually give us. So, for example, we can see that actually we can predict a lot based on gender, PAM50, RPPA, vital status, OS time, metastasis, if there was a node, if you had a tumor, and so on. However, there are other features that actually couldn't be explained by this model, and it is okay. So the rest of this kernel basically we will analyze for each of these variables how it explains for the different patients given the actual cluster of protein activity. And this is end. So, for example, for gender, we can get that actually it is way likely for females to have the class 0 protein activity of breast cancer. This could make sense because breast cancer is usually most common in females. Later on, given the ER Status, so this is the status that they get when they go to ER the first time, we can see that, for example, a high ER Status, that means that they triage weekly, it's really found in clusters 2, 4, and 5. The same with PR Status, but that makes sense because it's really related, those variables. And then we go, for example, to another status of different proteins, HER2 and tumor, and this one is really interesting. So, for example, what he found doing a multinomial logistic regression is that T1 tumors are actually most likely in patients with activity in clusters 4 and 5. This is really interesting. Later T2 tumors are most likely in patients with activity in cluster 5. And T3 tumors are most likely in patients in cluster 0. Also, he found a similar relationship given the nodes of those tumors. And the metastasis is really less likely in patients with activity in cluster 4, but are more likely in patients in cluster 0 and 3. Finally, we can go to the mRNA values, which actually explain a lot of the cluster 1 protein activity, but don't explain a lot of cluster 2. And the RPPA clusters behave similarly, are most likely patients with activity in cluster 4 and the RPPA clusters are least likely in patients with activity in cluster 5. So with that, we got a lot of variables that were actually predicted by classify in the cluster of the protein activity, and from there, when a new patient comes, we can check in which cluster of protein activity falls into and we can do a nice prediction of what type of breast cancer is the person going to have, or if it is deadly or not. This is really interesting, right? Of course, we could put all this information into a new model, but for that, the usual way is plug all this information into a neural network. So bear with me that this is out of scope for this course, but we will meet again. Thanks a lot for being with me and staying here. I think this kernel is excellent because it shows how logistic regression, clustering, PCA, everything meshes together to get a great analysis of our data.

Summary

Unsupervised learning is an incredibly interesting, unexplored, and fun world where multivariant statistics, some browse techniques, Bayesian approaches, all come together to basically try to get some information, since we don't have labels. In this module, we just gave a glimpse of what is out there in the wild, wild world. Basically we introduced the formal problem of unsupervised learning. We discovered what measurements can help us find intrinsic structure of our data. As a special case, we introduce clustering where in its both flavors, what we optimize is the inter versus intra cluster distance. Finally, we cover PCA and CA as a dimensionality reduction technique that tries to optimize variance or inertia of variables. There are many, many more, but if we try to cover them all, it will be a whole course in itself. So we just give a glimpse. The important aspect to take home is that the procedure is the same and we still optimize for cost, but with other objectives in mind. It was so fun to make this course and I really hope it was fun for you too, my friend. I hope you learned so much and you can always ping me whatever you want. I try to answer as fast as I can in order to spread the machine learning view of the world, but always trying to keep faithful to the statistics part. As I mentioned in our previous course, machine learning is changing the world, so I hope you join the ride. Thank you, thank you, thank you so much for seeing this course and I hope to see you again. Bye.

Course author



Axel Sirota

Axel Sirota has a Masters degree in Mathematics with a deep interest in Deep Learning and Development Lifecycle. After researching in Probability, Statistics and Machine Learning optimization, he...

Course info

Level	Advanced
Rating	★★★★★
My rating	★★★★★
Duration	3h 9m

Released	10 Sep 2019

Share course