LORENZO ROSASCO: So what we want to do now is to move away from local methods and start to do some form of global regularization method. The word regularization I'm going to use broadly as a term to define procedures, statistical procedures and computational procedure that do have some parameters that allow to do from complex model to simple model in a very broad sense. What I mean by complex is something that is potentially going closer to overfitting and by simple model something that is giving me something, which is stable with respect to data.

So we're going to consider the following algorithm. I imagine a lot of you have seen it before. This is called-- it has a bunch of different names-- probably the most famous one is Tikhonov regularization. A bunch of people at the beginning of the '60s thought about something similar either in the context of statistics or solving linear equations. So Tikhonov is the only one for which I can find the picture. The other one was Philips, and then there is Hoerl and other people. They basically thought all about this same procedure.

The procedure basically is based on a functional that you want to minimize based on two terms. So there are several ingredients going on here. First of all, this is f of x. We assume the functional form of-- we try to estimate the function, and we do assume a parametric form of this function, which in this case, just linear. And for the time being, because you can really, you can put it back in, I don't look at the offset. So I just take lines passing through the origin. And this is just because you can prove in one line that you can put back in the offset at zero cost. So for the time being, just think that data are actually standard.

The way you try to estimate this parameter is, on one hand, try to make the empirical error small, and on the other hand, you put a budget on the weights. The reason why you do this-- there are a bunch of way to explain this. Andrei yesterday talked about margin, and different lines, and so on. Another way to think about it is that you can convince yourself-- and we were going to see later-- that if you're in low dimension, a line is a very poor model. Because basically if you have more than a few points-- and they're not standing on the line-- you will not be able to make zero error.

But if the number of points is lower than the number of dimension, you can show that the line actually can give you zero error. It's just a matter of degrees of freedom. You have fewer equations than the actual variables. So what you do is that you actually add a regularization theorem. It's basically a theorem that makes the problem well-posed. We're going to see this in a minute from a different perspective. The easiest one is going to be numerical. We stick to least squares for-- and there is-- so there is an extra parenthesis that I forgot, but before I tell you why we use least squares also let me tell you that-- as somebody pointed out-- there is a mistake here, because this is a minus. It should just be a minus. I'll fix this.

So back, why do you use least squares? OK, so least squares on the one hand, if you're in low dimension especially, you can think of least squares as its way is basic, but it's not a very robust way to measure error, because you squared them. And so just one error can count a lot. So typically, there is a whole literature on robust statistics, where you want to replace least square with something like an absolute value or something like that. It turns out that at least in our experience and when you have high dimensional problem, it's not completely clear how much this kind of instability will occur and will not be cured by just adding some regularization term.

And the computation underlying this algorithm are extremely, extremely simple. So that's why we're sticking to this, because it works pretty well in practice. We actually developed in the last few years some toolbox that you can use. They're pretty much plug and play. And because the algorithm is easy to understand in simpler terms. Yesterday, Andrei was talking about SVM. SVM is very similar in principle. Basically the only difference is that you change the way you measure cost here. This algorithm you can use both for classification and regression, whereas SVM-- the one which was talked about yesterday-- is just for classification.

And because the cost function turns out to be non-smooth-- and non-smooth is basically non-differentiable-- and so the whole math is much more complicated, because you have to learn how to minimize things that are not differentiable. So in this case, you can stick to elementary stuff. And I think I did somewhere, that also because Legendre 200 years ago said that least squares are really great. There is this old story-- who between Gauss and Legendre invented least squares first. And there are actually long articles about this. But anyway, it's around that time. It's around the end of the-- this is when he was born-- it's around the end of the 18th century. So the algorithm is pretty old.

So what's the idea? So back to the case we had before, you're going to take a linear function. So one thing is-- just to be careful-- think about it once. Because if you've never thought about it before, it's good to focus. When you do this drawing, this is not f of x. This line is not f of x. It's f of x equals zero. So I think I made enough time to have a 3D plot. So f of x is actually a plane that cuts through the slide. It's positive, when it's not dotted-- because this points are positive-- and then becomes negative. And this line is where it changes sign.

So the decision boundary is not f of x itself, but it's the level set that corresponds to f of x equals zero. Whereas f of x itself is this one line. If you think in one dimension, the points are just standing on a line. Some here are plus 1. Some here are minus 1. So what is f of x? It's just a line. What is the decision boundary in this case? It will just be one point in this case actually, because it's just one line that cuts the input line in one point. And that's it.

If you were to take a more complicated nonlinear line, it would be more than one point. In two dimension, it becomes one line. In three dimension, it becomes a plane, and so on and so forth. But the important piece-- just a remember, at least once-- then we look at this plot. This is not f of x, but only the set f of x equals zero, which is where you change sign. And that's how you're going to make prediction. You take real valued functions, so you would like-- in principle, in classification, you would allow this function just to be binary.

But optimization with binary functions is very hard. So what you typically do to relax this? You just allow it to be a real valued function, and then you take a sign. When it's positive, you take plus 1. If it's negative, you say minus 1. If it's a regression problem, you just keep it for what it is. And how many free parameters has this algorithm? Well, one. It's lambda for now and w. But w we're going to solve by solving this optimization problem.

How about lambda? Well, whatever we discussed before for k. We would try to sit down and do some bias variance of the composition, see what it depends on, try to see if we can get a grasp on what the theory of this algorithm is. And then we try to see if we can use cross-validation. You can do all these things, so we're not going to discuss much how you choose lambda, but most of you are going to discuss how you can compute the minimizer of this. And this is not a problem, because this is smooth. So you can take the retrospect to w and also this.

So what you can do is just to take the derivative of this, set it equal to zero, and check what happens. So it's useful to do this to other-- just some vectorial notation. We've already seen it before. So you take all the x's and you stack it as rows of the data matrix x of n. So this ny, you just stack them as entries of a vector. You call it yn. Then you can rewrite this term just in this way, as this vector minus this vector here, which you obtain by multiplying the matrix with w. So this norm is the norm in Rn. So this is just simple rewriting. It's useful just because if you now take the derivative of this with respect to w, set it equal to zero, you get this. This is the gradient.

So I haven't set it to zero yet. This is the gradient of the least square part. This is the gradient of the second term. It is still multiplied by lambda. If you set them equal to zero, what you get is this. You take everything with x, so the 2 and the 2 goes away. You took everything with x, and you put it here. There's still the one here with lambda. You put it here. You take this term in x transpose y, and you put it on the other side of the equality. So you take everything with w on one side and everything without w on the other side. And then here, I remove n by multiplying. And so what you get is a linear system. It's just a linear system. So that's the beauty of least squares. Whether you regularize it or not-- in this case for this simple squared loss regularization, all you get is a linear system.

And this is the first way to think about the effect of adding this term. So what is this doing? So just quickly for you a quick linear system recap. You're solving a linear system. I changed notation. This is just a parenthesis, just a little bit. The simplest case you can think of is the case where m is diagonal. Suppose it's just a diagonal matrix, a square diagonal matrix. How do you solve this problem? You have to invert the matrix m. What is the inverse of a diagonal matrix? So it's just another diagonal matrix. On the entries, instead of, say, sigma, you have 1 over sigma or whatever it is.

So what you see is that if m-- you just consider m-- and m is diagonal like this-- this is what you're going to get. Suppose that now some of these numbers are actually small, then when you take 1 over, this is going to blow up. When you apply this matrix to b, what you might have is that if you change the sigmas or the b slightly, you can have an explosion. And if you want, this is one way to understand why adding the lambda would help. And it's another way to look at overfitting, if you want, from a numerical point of view. You take the data. You change them slightly, and you have numerical instability right away.

What is the effect of adding this term? Well, what you see is that instead of just doing m minus 1, you're doing m plus lambda I minus 1. And this is the simple case, where it's diagonal. But what you see is that on the diagonal instead of 1 over sigma 1, you take 1 over sigma 1 plus lambda. If sigma 1 is big, adding this lambda won't matter. If sigma-- for example, sigma d, now think there are order. I'm thinking they are order, and sigma d is small.

If this is small, at some point lambda is going to jump in, make the problem stable at the price of ignoring the information in that sigma, that you basically consider it to be at the same size of the noise or the perturbation or the sample in your data. Does this make sense? So this is what the algorithm is doing. And it's a numerical way to look at stability. But you can imagine that this is an immediate statistical consequence. You change the data slightly, you'll have a big change in your solution and the other way around. And lambda governs this by basically telling you how much this is invertible. So it's a connection between statistical and numerical stability.

Now of course, you can say, this is oversimplistic, because this is just a diagonal matrix. But basically, if you now take matrices that you can diagonalize, conceptually nothing would change. Because basically you would have that if you have a matrix-- so there is a mistake here. There should be no minus 1. If you have an m that you can-- this is just sigma, not minus 1. You can just diagonalize it. And now every operation you want to do on the matrix you can just do on the diagonal. So all the reasoning here will work the same. Only now you have to remember that you have to squeeze the diagonal matrix in between v and v transpose.

I'm not saying that this is what you want to do numerically. But I'm just saying that the conceptual reasoning here-- that we tell it that this was the effect of lambda-- is going to hold just the same here. This is m, which you can write like this-- m minus 1 you can write like this. And so this is just going to be the same diagonal terms inverted. And now you see the effect of lambda. It's just the same. So once you grasp this conceptually, for any matrix you can make diagonal, it's the same. And the point is that as long as you have a symmetric positive definite matrix, the reason you can diagonalize it, you just have the same thing squeezed in between v and v transpose.

And that's what we have, because instead of-- because what we have is exactly this matrix here. So instead of-- and you see here that basically this depends a lot on the dimensionality of the data. If the number of points is much bigger than the dimensionality, this matrix in principle could be-- it's easier that is invertible. But if the number of points is smaller than the dimensionality-- how big is this matrix? So xn is-- you remember how big was xn? It was the rows were the points, and the columns were the variable. So how big is this? And we call this

d. We called the length n. So this is--

AUDIENCE: [INAUDIBLE]

LORENZO ROSASCO: --n by d. So this matrix here is how big? Just d by d, and the number of points is smaller than the number dimension. The rank of this-- this is going to be rank-deficient. So it's not invertible. So if the number of points is more, if you're in a high-- so called high-dimensional scenario, where the number of points is more than the number of dimension, for sure you won't be able to invert this. Ordinary least squares will not work. It will be unstable. And then you will have to regularize to get anything reasonable.

So in the case of least squares, just by setting rank to zero and looking in this computation to get a grasp of both. What kind of computation you have to do, and what they mean both from the statistical and the numerical point of view. And that's why that's one of the beauty of least squares. We could stick to a whole derivation of this-- so this is more the linear system perspective. There is a whole literature trying to justify more from a statistical point of view what I'm saying.

You can talk about the maximum likelihood, then you can talk about maximum a posteriori. You can talk about variance reduction and so-called Stein effect. And you can make a much bigger story trying, for example, to develop the whole theory of shrinkage estimators, the bias variance tradeoff of this. But we're not going to talk about that. So this simple numerical stability, statistical stability intuition is going to be my main motivation for considering these schemes.

So let me skip these. I wanted to show the demo, but-- it's very simple. It's going to be very stable, because you're just drawing a one-dimensional line. Then you move on just a bit, because we didn't cover as much as I want in the first part. So first of all, so far so good? Are you all with me about this?

So again, the basic thing if you want-- all the interesting-- so this is the one line, where there is something conceptual happening. This is the one line, where we make it a bit more complicated mathematically. And then all you have to do is to match this with what we just wrote before. That's all. These are the main three things we want to do. And think a bit about dimensionality.

Now if you look at a problem even like this, as I said, this might be misleading-- a low dimension. And in fact, what we typically do in high dimension is that, first of all, you start with the linear model and you see how far you can go with that. And typically, you go a bit further that you might imagine. But still, you can think, why should I just stick to linear decision rule? This won't give me much of a flexibility. So in this case, obviously, it looks like something that would be better, some kind of quadric decision boundary.

So how can you do this? How can you go-- suppose that I give you the code of least squares. And you're the laziest programmer in the world, which in my case is actually not that hard to imagine. How can you recycle the code to fit, to create a solution like this, instead of a solution like this? You see the question? I give you the code to solve this problem, the one I showed you before-- the linear system for different lambdas. But you want to go from this solution to the solution. How could you do that?

So one way you can do it in this simple case is-- this is the example. So the idea is-- you remember the matrix? I'm going to invent new entries of the matrix, not of the points, because you cannot invent points, but of the variables. So what you're going to do, instead of just-- they can say, in this case I call them x1, x2. I'm just in two dimension. These are my data. This is just another example of this. So these are my data-- sorry these are-- let's see what they are. This is one point. X1 and x2 here are just the entry of the point x, so the first coordinate and the second coordinate.

So what you said is exactly one way to do this. And it is-- I'm going to now build a new vector representation of the same points. So it's going to be the same point, but instead of two coordinates I now use three, which are going to be the first coordinate square, the second coordinate square, and the product of the two coordinates. Once I've done this, I forget about how I got this, and I just treat it as new variables. And I take a linear model with that variables. It's a linear model with these new variables, but it's a new linear model with the original variables. And that's what you see here.

So x tilde is this stuff. It's just a new vector representation. And now I'm linear with respect to this new vector representation. But when you write x tilde explicitly, it's some kind of non-linear function of the original variable. So this function here is non-linear in the original variable. It's harder to say than probably to see. Does it make sense? So if you do this, you're completely recycling the beauty of the linearity from a computational point of view while augmenting the power of your model from linear to non-linear.

It's still parametric in the sense that in this case-- what I mean by parametric is that we still fix a priori the number of degrees of freedom of our problem. It was true now I make it three. More general I could make it p, but the number of numbers I have to find is fixed a priori. It doesn't depend on my data, and it's fixed. But I can definitely go from linear to non-linear. So let's keep on going. So from the simple linear model we already went quite far, because we basically know that with the same computation we can now solve stuff like this.

Let's take a couple of steps further. So one is-- appreciate that really the code is just the same. Instead of x, I have to do a pre-processing to replace x with this new matrix x tilde, which is the one which instead of being n by d, is now n by p where p is this new number of variables that I invented.

Now it's useful to just get the feeling of what is the complexity of this method. And this is a very quick complexity recap. Here basically, the product of two numbers is going to count one. And then when you take product of vectors of matrices, you just count on any real number multiplication you do. And this is a quick recap. If I multiply two vectors of size p, the cost p, matrix vector is going to be np. Matrix matrix is going to be n square p. You have n vectors. And one-to-one, other n vectors. And they are size p, so each time you have-- it costs you p. And you have to do n against n. So it's going to be n square p.

And the last one is-- this is a much-- less clear to just look at it like this. But roughly speaking, the inversion of a matrix costs roughly speaking n cube in the worst case. It's just to give you a feeling of what the complexity are. So it makes sense? It's a bit quick, but it's simple. If you know it, OK. Otherwise, you just take this on the side, when you think about this.

So what is the complexity of this? Well, the matrix-- you have to multiply this times this, and this is going to cost you nd or np. You have to build this matrix. This is going to cost you n square d or n square p. And then you have to invert. These are going to be n cube. So-- sorry, p cubed, because with this matrix is going to be-- or d cube, because this matrix is d by d. So this is, roughly speaking, the cost.

So now look at this. This is-- I take this. In this case, p is the new variable, otherwise d. So in this case, I have p cube, and then I have p square n. But one question is what if n is much-- and that's a fact-- what if n is much smaller than p? If n is a 10, do I really have to pay quadratic or even cubic in the number of dimension to solve this problem? Because in some sense, it looks I'm overshooting things a bit. Because I'm inverting a matrix, yes, but this matrix is really a rank n. It only has n rows that are linearly independent at most. It might be less, but at most it has n.

So can I break the complexity of this? Linear system have to solve, you just use the table I showed you before. Check the computation. These are the computation you have to do. And one observation here is you pay really a lot in the dimension, the number of variables or the number of features you invented. And this might be OK, when p is smaller than n. But one thing-- this seems wrong intuitively, when n is much smaller than p. Because the complexity of the problem, the rank of the problem is just n. The matrix here has n rows and d or p columns depending on which representation you take. And so the rank of the whole thing is at most n, if n is much smaller.

So now the red dot appears. And what you can do is proving this one line. So let's see what they do, and then I'll tell you how you can prove it. And it's an exercise. So you see here if you invert this, then you have to multiply x transpose y times the inverse of this matrix, which is what's written in here. So I claim that this equality stands. Look what it does. I take this x transpose. I move it in front. But then if I do this, you clearly see that I'm messing around with dimensions. So what you do is that you have to switch the order of the two matrices in the middle. Now from a dimensionality point of view, at least, I still see that this matrix and this matrix have the same dimension.

How do you prove this? Well, you basically just need to do SVD. You take the singular-value decomposition of the matrix Xn. You plug it in, and you just compute things. And you check that this side of the equality is the same of this side of the equality. So there's nothing more than this, but we're going to skip this. So you just take this as a fact. It's a little trick. Why do I want to do this trick? Because look, now what I say is that my w is going to be x transpose of something.

What is this something? So w is going to be X transpose of this thing here. How big is this vector? So how big is this matrix first of all? So remember, Xn was how big?

AUDIENCE: N by d.

LORENZO ROSASCO: N by d or p. How big is this?

AUDIENCE: N by n.

LORENZO ROSASCO: N by n. So how big is this vector? It's n by 1. So now I have to-- I found out that my w can always be written as x transpose c, where c is just an n-dimensional vector. I rewrote it like this, if you want. So what is the cost of doing this? Well, this was the cost of doing this? But now you just have to do-- so let's say what is the cost of doing this thing here above the bracket? Well, if this one was p cube p square n, this one will be how much?

I have that this matrix will say p by p, and then this vector was p by 1. Whereas here, my matrix is n by n, and the victory is n by 1. So you basically have that these two numbers swap. Instead of having this complexity, now you have a complexity, which is n cube. And then you have n square p, which sounds about right. It's linear in p. You cannot avoid that. You have to look at the data at least once. But then it's polynomial only in the small quantity of the two. So in some sense, what you see is that, depending on the size of n, of course, you still have to do this multiplication. But this multiplication is just n, nd, or np.

So let's just recap what I'm telling you. This is a lot more mathematical fact I put. I have a warning here. The first thing is the question should be clear. Can I break the complexity of this in the case when n is smaller than p or d? This is relevant because the question came out a second ago, which was should I always explode the dimension of my features? And here what you see is that-- well, at least for now we see that even if you do, you don't pay more than linearly in that.

And the way you prove it is A, you observe this factor, which, again, I measured if you're curious, to show how you do it, but it's a one line. And 2, you observe that once you have this, if you just rewrite w, you can write w as a x transpose c. And to find a c-- which is now you basically re-parametrize-- and to find the new c is going to cost you only n cube n square p. So you do exactly what you wanted to do. And basically, what you see now is that whenever you do least squares, you can check the number of dimensions, the number of points, and always re-parametrize the problem in such a way that complexity is depending linearly on the bigger of the two and polynomially on the smaller of the two. So that's good news.

Oh, I wrote it. So this is where we are right now. So if we're lost now, you're going to become completely lost in one second. Because this is what we want to do. We want to introduce kernel in the simplest possible way, which is the following. So look at-- this is what we find out. We discovered, we actually proved a theorem. And the theorem says that the w's that are output by the least squares algorithm are not any possible d-dimensional vectors, but they're always vectors that I can write as the combination of the training set vectors. So xi is long d or p, and I've summed them up with these weights.

And the w's that are going to come out of least squares are always of that form. They cannot be of any other form. This is called the representer theorem. It's the basic theorem of so-called kernel methods. It shows you that the solution you're looking for can be written as a linear superposition of these terms. If you now write-- this is just the w. Let's just write down f of x. F of x is going to be x transpose w, just the linear function. And now you can-- if you write it down, you just get this.

By linearity you can-- so w is written like this. You multiply by x transpose. This is a finite sum. So you can let x transpose inside the sum. This is what you get. Are you OK? So you have x transpose times a sum. This is the sum of x transpose multiplied by the rest. Why do we care about this? Because basically the idea of kernel methods-- in this very basic form-- is what if I replace this inner product, which is a way to measure similarity between my functions, with another similarity.

So instead of mapping each x into a very high dimensional vector and then taking product-- which is itself, if you want another way, as I said, of measuring similarity in your product, distances between vectors-- what if I just define it, instead of by an explicit mapping, by redefining the inner product. So this k here is the k similar to the one we had in the previous-- in the very first slide. And it's-- re-parametrize the inner product. Change the inner product, and then I want to use everything else.

So we need to question-- we need to answer two question. The first one is if I give you now a procedure that whenever you would want to do x transpose x does something else called ax comma x prime. How do you change the computations? This is going to be very easy. But also what are you doing from a modeling perspective?

So from the computational point of view, it's very easy, because you see here you always had that you have to build a matrix whose entries were xi transpose xj. So it was always a product of two vectors. And what you do now is that you do the same. So you build the matrix kn, which is not just xn, xn transpose but is a new matrix whose entries are just this. This is just a generalization. If I put the linear kernel, I just get back in what we had before. If you put another kernel, you just get something else.

So from a computational point of view, you're done for this computation of c. You have to do nothing else. You just replace this matrix with these general matrix. And if you want to now compute s-- so w you cannot compute anymore, because you don't know what's an x by itself. But if you want to compute f of x, you can, because you've just to plug-in-- So you know how to compute the c. And you know how to compute this quantity, because you have just to put the kernel there. So the magic here is that you never ever point x in isolation.

You always have a point x multiplied by another point x. And this allows you to replace vectors by-- in some sense, this is an implicit remapping of the points by just redefining the inner product. So what you should see for now is just that the computation that you've done to compute f of x in the linear case you can redo, if you replace the inner product with this new function. Because A, you can compute c by just using this new matrix in place of this. And B, you can replace f of x, because all you need is to replace this inner product with this one and put the right weights, which you know how to compute.

From a modeling perspective what you can check is that, for example, if you choose here this polynomial kernel-- which is just x transpose x prime plus 1 elevated to the d-- if you take, for example, d equal 2, this is equivalent to the mapping I showed you before, the one with explicit monomials as entries. This is just doing it implicitly. If you're in low-dimensional, if you're low-dimensional, if n is very big, and the dimensions are very small, the first way might be better. But if n is much bigger, this way would be better.

But also you can use stuff like this, like a Gaussian kernel. And in that case, you cannot really write down explicitly the explicit map, because it turns out that it's infinite-dimensional. The vectors you would need to write down, to write down the explicit variable version of-- embedding version of this is infinite-dimensional. So this is a-- if you use this, you get the truly non-parametric model. If you think of what is the effect of using this, it's quite clear if you plug them here. Because what you have is that in one case you have a superposition of linear stuff, a superposition of polynomial stuff, or a superposition of Gaussians.

So same game as before. So same dataset we train. I take kernel least squares-- which is what I just showed you-- compute the c inverting that matrix, use the Gaussian kernel-- the last of the example-- and then compute f of x. And then we just want to plot it. So this is the solution. The algorithm depends on two parameters. What are they?

AUDIENCE: Lambda.

LORENZO Lambda, the regularization parameter, the one that appeared already in the linear case-- and

ROSASCO: then--

AUDIENCE: Whatever parameter you've chosen [INAUDIBLE].

LORENZO ROSASCO: Exactly. Whatever parameters there is in your kernel. In this case, it's the Gaussian, so it will depend on this width. Now suppose that I take gamma big. I don't know what big is. I just do it by hand here, so we see what happens. If you take gamma-- sorry gamma, sigma big, you start to get something very simple. And if I make it a bit bigger, it will probably start to look very much like a linear solution. If I make it small-- and again, I don't know what small is, so I'm just going to try. I's very small. You start to see what's going on.

And if you go in between, you really start to see that you can circle out individual examples. So let's think a second what we're doing here. It is going to be again other hand-waving explanation. Look at this equation. Let's read out what it says. In the case of Gaussians, it says, I take a Gaussian-- just a usual Gaussian-- I center it over a training set point, then by choosing the ci I'm choosing whether it is going to be a peak or a valley. It can go up, or it can go down in the two-dimensional case. And by choosing the width, I decide how large it's going to be.

If I do f of x, then I sum up all this stuff, which basically means that I'm going to have these peaks and these valleys and I connect them in some way. Now you remember before that I pointed out within the two-dimensional case what we draw is not f of x, but f of x equal to zero. So what you should really think is that f of x in this case is no longer an upper plane, but it's this surface. It goes up, and it goes down. And it goes up, and it goes down. So in the blue part, it goes up, and in the orange part, it goes down into valley.

So what you do is that right now you're taking all these small Gaussians, and you put them in around blue and orange point, and then you connect their peaks. And by making them small, you allow them to create a very complicated surface. So what did we put before? So they're small. They're getting smaller, and smaller, and smaller. And they go out, and you see the-- there is a point here, so they circle it out here by putting basically Gaussian right there for that individual point.

Imagine what happens if my points-- I have two points here and two points here-- and now I put a huge Gaussian around each point. Basically, the peaks are almost going to touch each other. So what you're imagine is that you get something, where basically the decision boundary has to look like a line, because you get something which is so smooth. It doesn't go up and down all the time. It's going to be-- And that's what we saw before, right? And again, I don't remember what I put here.

So this is starting to look good. So you really see that somewhat something nice happens. Maybe if I put-- five is what we put before maybe. So basically what you're basically doing is that you're computing the center of mass of one class in the sense of the Gaussians. So you're doing a Gaussian mixture on one side, a Gaussian mixture on the other side, you're basically computing the center of masses, and then you just find the line that separates the center of masses. That's what you're doing here, and you just find this one big line here.

So again, so we're not playing around with the number of points. We're not play around with lambda. But because this is basically what we already saw before. All I want to show you right now is the effect of the kernel. And here I'm using the Gaussian kernel, but-- let's see-- but you can also use the linear kernel. This is the linear kernel. This is using the linear least squares. If you now use the Gaussian kernel, you give yourself the extra possibility. Essentially, what you see is that if you put the Gaussian which is very big, in some sense you get back the linear kernel.

But if you put the Gaussian which is very small, you allow yourself to this extra complexity. And so that's what we gain with this little trick that we did of replacing the inner product with this new kernel. We went from the simple linear estimators to something, which is-- It's the same thing-- if you want-- that we did by building explicitly these monomials of higher power, but here you're doing it implicitly. And it turns out that it's actually-- there is no explicit version that you can-- You can do it mathematically, but the feature representation, the variable representation of this kernel would be an infinitely long vector.

The space of function that is built as a combination of Gaussians is not finite-dimensional. For polynomials, you can check that the space of function, it basically is a polynomial in d. If I ask you how big is the function space that you can build using this-- well, this is easy. It's just d-dimensional. With this, well, this is a bit more complicated, but you can compute. For this, it's not easy to compute, because it's infinite. So it in some sense is a non-parametric model. What does it mean?

Of course, you still have a finite number of parameters in practice. And that's the good news. But there is no fixed number of parameters a priori. If I give you a hundred points, you get a hundred parameters. If I give you 2 million points, you get 2 million parameters. If I give you 5 million points, you get 5 million parameters. But you never hit a boundary of complexity, because these are in some sense as an infinite-dimensional parameter space.

So of course, I see that here there are some of the part that I'm explaining are complicated, especially if this is the first time you see them. But the take-home message should be essentially from least squares, I can understand what's going on from a numerical point of view and bridge numerics and statistics. Then by just simple linear algebra, I can understand the complexity-- how I can get complexly-- which is linear in the number of dimension or the number of points.

And then by following up, I can do a a little magic and go from the linear model to something non-linear. The deep reason why this is possible are complicated. But as a take-home message, A, the computation you can check easy. It remained the same. B, you can check that what you're doing is now allowing yourself to take a more complicated model, it's combination of the kernel functions. And then even just by playing with these simple demos, you can understand a bit what is the effect. And that's what you intuitively would expect. So I hope that it would get you close enough to have some awareness, when you use this.

And of course, you can put-- when you abstract from the specificity of this algorithm, you build an algorithm with one or two parameters-- lambda and sigma. And so as soon as you ask me how you choose those, well, we go back to the first part of the lecture-- bias-variance, tradeoffs, cross-validation, and so on and so forth. So you just have to put them together.

There is a lot of stuff I've not talked about. And it's a step away from what we discussed, so you've just seen the take-home message part, but we could talk about reproducing kernel hybrid spaces, the functional analysis behind everything I said. We can talk about Gaussian processes, which is basically the probabilistic version of what I just showed you now. Then we can all see the connection with a bunch of math like integral equations and PDEs. There is a whole connection with the sampling theory a la Shannon, inverse problems and so on. And there is a bunch of extension, which are almost for free.

You change the loss function. You can make the logistic, and you take kernel logistic regression. You can take SVM, and you get kernel SVM. Then you can also take more complicated output spaces. And you can do multiclass, multivariate regression. You can do regression. You can do multilabel, and you can do a bunch of different things. And these are really a step away. These are minor modification of the code. And you can do a bunch of stuff.

So the good thing of this is that with really, really, really minor effort, you can actually solve a bunch of problem. I'm not saying that it's going to be the best algorithm ever, but definitely it gets you quite far.

So again we spent quite a bit of time thinking about bias-variance and what it means and used least squares and just basically warming up a bit with this setting. And then in the last hour or so, we discussed least squares, because it allows to just think in terms of linear algebra, which is something that-- one way or another-- you've seen in your life. And then from there, you can go from linear to non-linear. And that's a bit of magic, but a couple of parts-- which are how you use it both numerically and just from a practical perspective to go from complex models to simple models and vice versa-- should be-- is the part that I hope you keep in your mind.

For now, our concern has just been to make predictions. If you hear classification, you want to have good clarification. If you hear regression, you want to do good regression. But you didn't talk about-- we didn't talk about understanding how did you do good regression? So a typical example is the example in biology. This is, perhaps, a bit old. This is micro-arrays. But the idea is the datasets you have is a bunch of patients. For each patient, you have measurements, and the measurements correspond to some gene expression level or some other biological process.

The patients are divided in two groups, say, disease type A and disease type B. And based on the good prediction of whether a patient is disease type A or B, you can change the way you cure it or you address the disease. So of course, you want to have a good prediction. You want to be able-- when a new patient arrive-- to say whether it's going to-- this is type A or type B. But oftentimes, what you want to do is that you want to use this not as the final tool, because unless deep learning can solve this, you might go back and study a bit more the biological process to understand a bit more.

So you use this as more statistical tools like measurements, like the way you can use a microscope or something to look into your data and get information. And in that sense sometimes, it's interesting to-- instead of just saying is this patient going to be more likely to be disease type A or B, it's to go in and say, ah, but when you make the prediction, what are the process that matters for this prediction? Is this gene number 33 or 34, so that I can go in and say, oh, these genes make sense, because they're in fact related to these other processes, which are known to be related, involved in this disease.

And doing that, you use just as a little tool, then you use other ones to get a picture. And then you put them together. And then it's mostly on the doctor, or the clinician, or the biostatistician to try to develop better understanding. But you do use these as tools to understand and look into the data. And in that perspective, the word interpretability plays a big role. And here by interpretability I mean I not only want to make predictions, but I want to know how I make predictions and tell you, come afterwards with an explanation of how I picked the information that were contained in my data. So so far it's hard to see how to do it with the tools we had.

So this is basically the field of variable selection. And in this basic form, the setting where we do understand what's going on is the setting of linear models. So in this setting basically, I just rewrite what we've seen before. You have x is a vector, and you can think of it, for example, as a patient. And xj are measurements that you have done describing this patient. When you do a linear model, you basically have that by putting a weight on each variables, you're putting a weight on each measurement.

If a measurement doesn't matter, you think you might put here a zero. And it will disappear from the sum. If the measurement matters a lot, then here you might get a big weight. So one way to try to get the feeling of which measurements are important and which are not and to try to estimate and model, a linear model, where you get the w, but ideally we would like to get the w, which has many zeros. You don't want to fumble with what's small and what's not. So if you do least squares the way I showed you before, you would get a w. Then you would get-- most of them you can check that it will not be zero.

In fact, none of them will be zero in general. And so now you have to decide what's small and what's big, and that might not be easy. Oops, what happened here? So funny enough, this is the name I found on how-- I don't remember the name of the book. It's the name that was used to describe the process of variable selection, which is a much harder problem, because you don't want to make predictions. But you want to go back and check how you make the prediction. And so it's very easy to start to get overfitting and start to try to squeeze the data until you get some information.

So it's good to have a procedure that will give you somewhat a clean procedure to extract the important variables. Again, you can think of this as a-- basically, I want to build an f, but I also want to come up with a list or even better weights that tell me which variables are important. And often this will be just a list, which is much smaller than d, so that I can go back and say, oh, measurement 33, 34, and 50-- what are they? I could go in and look at it. Notice that there is also a computational reason why this would be interesting.

Because of course, if d here is 50,000-- and what I see is that, in fact, I can throw away most of these measurements and just keep 10-- then it means that I can hopefully reduce the complexity of my computation, but also the storage of the data, for example. If I have to send you the datasets after I've done this thing, I've just to send you this teeny tiny matrix. So interpretability is one reason, but the computational aspect could be another one.

Another reason that I don't want to talk too much is also-- remember that we had this idea, where we said we could document the complexity of a model by inventing features, and he said do I always have to pay the price of making it big? Well, I basically-- if you what-- I was pointing at-- I said, no, not always, because I was thinking of kernels. These, if you want give you another way potentially to go around in which what you do is that, first of all, you explode the number of features.

You take many, many, many, many, and then you use this as a preliminary step to shrink them down to a more reasonable number. Because it's quite likely that among these many, many measurements, some of them would just be very correlated, or uninteresting, or so on and so forth. So this dimensionality reduction or computational or interpretable model perspective is what stands behind the desire to do something like this.

So let's say one more thing and then we'll stop. So suppose that you have an infinite computational power. So the computation are not your concern, and you want to solve this problem. How will you do it? Suppose that you have the code for least squares. And you can run it as many times as you want. How would you go and try to estimate which variables are more important?

AUDIENCE: [INAUDIBLE] possibility of computations.

LORENZO ROSASCO: That's one possibility. What you do is that you have-- you start and look at all single variables. And you solve least squares for all single variables. Then you take all couples of variables. Then you get all triplets of variables. And then you find which one is best. From a statistical point of view there is absolutely nothing wrong with this, because you're trying everything. And at some point, you find what's the best. The problem is that it's combinatorial. And you see that when you're in dimension a few more then-- very few, it's huge. So it's exponential.

So it turns out that doing what you just told me to do, which is what I asked you to tell me to do, which is this brute force approach is equivalent to do something like this is again a regularization approach. Here I put what is called the zero norm. The zero norm is actually not a norm. And it is just functional. It's a thing that does the following thing. If I give you a vector, you've to return the number of components different from zero, only that.

So you go inside and look at each entry, and you tell if they are different from zero. This is absolutely not convex. And so this is the reason why this problem is equivalent-- it becomes a computation not feasible. So perhaps, we can stop here. And what I want to show you next is essentially-- if you have this-- and you know that in some sense, this is what you would like to do, if you could do it computationally, but you cannot-- so how can you find approximate version of this that you can compute in practice? And we're going to discuss two ways of doing it. One is greedy methods and one is convex relaxations.