Module 9: Model Selection and Regularization

Video Transcripts

Video 1: Polynomial Features on Multidimensional Data

In a previous module, we saw the relationship shown between the model complexity, the model sensitivity, and the model error.

As our model grows increasingly complex, training error decreases while the model variance increases. This variance eventually drives the validation and test errors back up to very high levels as our model begins to overfit to the training set. We can see real-world versions of this idealized curve by training models of various complexity and measuring the resulting training and validation error.

For example, for our scenario from the overfitting module, the knob that we used to increase the complexity of our models was the polynomial degree. As we increase the polynomial degree, we get a more and more complex model. Plotting the mean squared errors for our training and validation sets gave us the figure shown.

In this module, we'll explore a richer version of the same idea. Specifically, we will see a situation where rather than having seven different models, we have a vast number of models to choose between. Having so many different models to consider will make cross-validation difficult, as we shall soon see.

As our starting point, we'll consider the code we used to generate polynomial features from a dataset that was initially one-dimensional. That is, our original dataset had only one single parameter, the horsepower. But after being fed through the polynomial transform object, we end up with as many features as we specify as our hyperparameter degree.

We can also apply a polynomial features object to data that is initially multi-dimensional. For example, if our initial dataframe has two columns, horsepower and weight, then the output of our polynomial features object with degree two will have five features. The first two features will be the horsepower and weight again. The third feature will be the horsepower squared. The fourth feature will be the horsepower times the weight. And the fifth, and final, feature will be the weight squared. Observe that we've formed all order two combinations of our features.

In case you're curious, the reason in this code we have this backslash is that we need to indicate to Python that the code has continued on the next line. Without this backslash the code would crash.

So increasing the polynomial features degree parameter to three, well now yield one, two, three, four, five, six, seven, eight, nine output features. And these are the same five features upfront from the degree two model. But in addition, we have the horsepower cubed, the horsepower squared times weight, the horsepower times the weight squared, and lastly, the weight cubed.

So the number of generated features we end up with depends both on the degree of the polynomial features object, as well as the number of features that we start with. So for example, suppose we start with this dataframe with three columns: horsepower, weight, and displacement. And we then feed it to a polynomial transform of degree two. In that case, you end up with these nine features.

Maybe pause the video for a moment and convince yourself that the last six features in this dataframe comprise all degree two combinations of the original three features.

So let's discuss how we would conduct simple cross-validation for the scenario shown. One approach would be to do exactly what we did in our previous module on over-fitting, where we find the mean squared error for various choices of our hyperparameter degree. However, this is a crude approach. For example, maybe the horsepower squared feature is useful, but none of the other degree two features are useful. Simply using cross-validation to select degree would fail to identify the fact that a model that only uses horsepower squared is excellent.

So a more exhaustive approach would be to consider every possible combination of features and see which yields the lowest mean squared error. Since we have nine different parameters, this would mean fitting two to the ninth, or 512 different models. Then we'd compute the validation error on each of these models and pick the one that had the lowest mean squared error. Now for 512 models, that isn't that computationally expensive, as long as our dataset is not too large.

The vehicle dataset that I'm using as a running example in this video, it has five numeric features: cylinders, displacement, horsepower, weight, and acceleration. Now if we feed all five of these features into a degree three polynomial features object, we end up with a 55-dimensional feature set. In other words, our dataframe has 55 columns. Trying out all possible combinations would require us to train two to the 55th models, which is quite a lot. And at a millisecond per model, it would take around a million years to completely exhaustively compute the mean squared error for all of these combinations.

Returning back to our idealized cartoon picture of model error versus model complexity, we observe that the space which includes our 55-dimensional model does not neatly fit into the conceptual framework. That is, when we were only adjusting the polynomial degree, it was clear which model was more complex than another. We just compare their degrees. And furthermore, there was only ever one model of any given degree. Now we have a vast landscape of two to the 55th possible models of varying complexity. So the space we need to explore is much more complicated.

So today we will show two alternate approaches which are much faster than exhaustively searching all possible models. The first will be sequential feature selection, and the second will be regularization.

Video 2: Sequential Feature Selection

To set the stage for this video, let's review our previous best model for estimating fuel efficiency from horsepower.

We found in a previous module; the degree two model gave us our best development set error. And recall, when I say development set error, that's just another way of saying validation set error. So here, running the code shown, we get a development set error of 20.98 on our two-feature model. And as a reminder, the two features used by this model, were horsepower and horsepower squared.

Now by contrast, we can of course also fit a model on all of those 55 features that we generated in the previous video. If we fit this linear regression model, we see that the training error is much lower. And this is no surprise because our model has so much more expressive power. However, in this case, we see that the development set error is also quite high — 31.52 — which is much worse than our degree two model that only used horsepower and horsepower squared.

In other words, it seems like our model is abusing the extra expressive power of these 55 available features, resulting in overfitting. And as we just saw in the previous video, one way to find a suitable middle ground between those two models would be to exhaustively search all combinations of the available 55 features. The only problem is that this would take about a million years to complete on a typical laptop.

Now an alternate approach would be to search a subset of this space. And there are many different possible approaches for doing so. One very conceptually simple technique is sequential feature selection. In this approach, we start with zero features initially selected. As this algorithm runs, I will use "phi selected" as the name for the set of features that we have selected so far. So when we start, this set is initially empty. We will then add features one at a time to phi selected until we reach some arbitrarily predetermined number of features. So for example, if we have 55 features to choose from, we might use this process to say, let's find the best four features. The way we decide which feature to add is very natural. As before, let phi selected be the set of features we've decided to add and let phi remaining be the set of features that we have not yet added. So when we first start, phi selected will be empty and phi remaining will be all of the features. Then what we'll do is for each feature phi candidate in phi remaining, we will fit a model that includes all of the features from phi selected as well as phi candidate. We then compute the development set error on this model. Whichever phi candidate has the lowest error will be added to phi selected. And we'll just repeat that process until we get the number of features that we desire.

Now if that seemed a little confusing, let's see a short example. Suppose we have five features to pick from: horsepower, weight, horsepower squared, horsepower times weight, and weight squared. When we start our algorithm, we initially have no features selected and all features remain, as shown. Now to decide which feature to add first, we will fit five models and we will compute the dev set error for each model. These five models are: a model which uses only horsepower, a model which uses only weight, a model which uses only horsepower squared, a model which uses only horsepower times weight, and lastly, a model using only weight squared. Let's suppose that the model which uses horsepower has the lowest mean squared error. In that case, we will select horsepower as our first feature. So now we've selected horsepower as our first feature, and it will stay selected for this procedure forever.

Now, since we wanted three features, we're not done yet. So we need to try out each of the four remaining next features and see which one is best. So we will fit four models. The first uses only horsepower and weight. The second model uses only horsepower and horsepower squared. The third model uses only horsepower and weight. And the fourth model uses only horsepower and weight squared. We compute the mean squared error for each of these four models on the dev set. And we look for the model that has the lowest mean squared error. So suppose that horsepower and weight together yield the lowest mean squared error. This means we'll move weight into our selected feature set.

So now we've selected both horsepower and weight. And because in this procedure, feature selection is permanent, they will both stay selected forever.

So next, to find our third and final parameter, we fit three models and compute their dev set errors. Just as before, each of these models has the two features from phi selected, namely horsepower and weight, and one additional feature from phi remaining, as shown on the screen. Now suppose that of those three models, the one that contains horsepower, weight, and horsepower squared has the lowest mean squared error. This means we will move horsepower squared into our selected set.

Now since at the beginning we said we wanted three features; we are now done. We have completed forward sequential feature selection.

Video 3: Sequential Feature Selection in Scikit-learn

Scikit-learn provides a transformer called sequential feature selection that takes a high-dimensional dataset and outputs a lower dimensional dataset using the sequential feature selection process that we've just described.

To do this in scikit-learn, we'll start by explicitly generating all degree three combinations of our 55 numeric features, yielding the 55-dimensional dataset from before.

The next step is going to seem a little bit out of nowhere. We need to create two lists of numbers. The first list will be all of the sample numbers for the training set. And the second list will be all the sample numbers to use for the dev set. The reason we're doing this strange thing is that these lists are required by the sequential feature selection transformer if we want to use the simple cross-validation procedure that we've been using in our course so far.

There are many potential ways to generate these lists of numbers. The approach that I have used is to first create a list of all numbers from zero to 371, then to shuffle them, and then to use NumPy split from before.

So now that we have these lists, we can create our sequential feature selection transformer. Like other scikit-learn objects, its constructor takes a number of different arguments. The first argument is the estimator that we're ultimately going to be using. In our case, we are generating our estimators using the scikit-learn LinearRegression object. So we provide an instantiated but untrained linear regression object that the sequential feature selector will use.

Next is the scoring argument. This tells the sequential feature selector how to compute the loss of the estimator. Here we say neg_mean_squared_error. And the reason we use the negative of the mean squared error is that the sequential feature selector will pick the feature with the highest score. Now since we want to minimize mean squared error, that means we want to maximize the negative of the mean squared error. It's a common trick.

Now, other functions are also valid as well for scoring. So see the documentation for the get score method if you'd like to explore other options.

Next, we provide cv, which stands for cross-validation. And here we provide a list containing a list containing our training and dev indices. If you look at the documentation for sequential feature selection, you'll see there are many, much more complicated things that we can provide to the cv argument. But for our purposes, since we're using simple cross-validation, our argument is a list of a list of our samples.

Lastly, we tell it the number of features that we want to select, and I have arbitrarily chosen four here.

So the next step is just like using any Sklearn transformer, we simply call fit_transform on our data. And then we convert the output of fit_transform into a DataFrame, as we've seen before.

This yields the four-dimensional datasets shown. Here we see that the four best features, at least according to sequential feature selection, are cylinder times weight, horsepower times acceleration, cylinder squared times acceleration, and displacement times weight squared.

Whether or not this model reflects some underlying physical reality or just happens to be empirically useful for this specific 392 row dataset is impossible to say. All we can say is that these specific set of four features minimize the dev set error according to this sequential feature selection process. In other words, don't read too much into what these columns are named. Our model here is hopefully useful for prediction, but it's very unlikely to be useful for inference at all.

To compare this brand new four parameter model to our earlier models, we can compute the mean squared error on the dev set. And while we're at it, let's compute the training set error as well, though, of course, we should never use that to compare and choose between different models. Now if you look at this code that I'm using to compute the errors, you'll see it looks a little bit different than our previous mean squared error predictions. That's because for this approach, we put our training indices and our dev indices into separate lists, instead of splitting the original dataset into two entirely different training and dev set dataframes. Because of this, I'm using iloc to retrieve the training and development sets when computing the respective errors. Now if that doesn't quite make sense right now, don't worry, you'll have a chance to practice this on the homework. We see that the model that uses the best four features has a dev set error of only 16.44. In other words, given a choice of these models, the four feature model seems best.

Now before we reflect on this sequential feature selection process, I have a quick check of your understanding that I'd like you to try. How many models did we fit in total during our sequential feature selection example on the 55-dimensional dataset? In other words, how many models do we need to create to decide which of the four features we want to keep? Pause the video and think about it and I'll give a hint in a couple of seconds. First, consider how many models we had to fit to select our first feature and then, and then work from there.

Okay, so the answer is 214. The reason is to pick the first parameter, we had to fit 55 models. Then for the second parameter, we had to fit 54 models to choose among the 54 remaining features. Then to pick the third parameter, we had to fit 53 models. And then finally 52 models to pick our final parameter. At that point we were done. And so the total number of models is 55, plus 54, plus 53, plus 52, which is 214.

So now let's compare the sequential feature selection approach to the exhaustive approach from before. In this approach, we greedily explored just a small fraction of the gigantic space of all possible feature combinations. The sequential feature selection algorithm is called the greedy algorithm, because it makes a sequence of greedy choices that are best at the moment, but which might not yield the overall best result. In general, greedy algorithms are not guaranteed to find the global optimum, but they are generally quite fast.

We see this clearly here because we considered only 214 models out of a total of 2^55 possibilities. So rather than taking a million years to complete, the process was done in less than a second. Now it's worth noting that sequential feature selection with N equals four doesn't even necessarily give us the best choice of four features. We only considered 214 combinations, but there's actually 341,055 different combinations of four features.

It's also worth noting that each time we add a feature, there's no guarantee whatsoever that our dev set error will go down. And in fact, as you'll see on the homework, as we add features, we usually see the dev set error go down at first, then fluctuate up and down as more features are added before finally going back up once we've added all the features to the model and we're just overfitting.

Before we saw that our sequential feature selection picked four possible features out of the 55 features it had choices of. And an interesting question you may have is, well, will we always get the same four features if we do this again? And it turns out the answer's no. Any feature selection process, of course, depends on the samples that are selected. So if you run the code shown on the screen multiple times, you will get different samples each time. And since we have different samples from our original dataset, there's a chance that just by luck, you'll end up selecting a different set of four features. This is especially true if there happens to be redundancy between some of our features.

As a quick demonstration, here we see the results of four different runs of sequential feature selection. You'll notice there's some overlap, but the choices made by sequential feature selection are not particularly stable between each run of this algorithm. Now since some of these features have some correlation, for example, a vehicle with high horsepower also tends to be heavier, it's no surprise that the selection of features is a bit unstable. There are techniques, that we will not describe in today's videos, for identifying and removing features which are two highly correlated, before we even begin this entire feature selection process. You'll have a chance to explore some of these issues in more detail on the homework.

Now there are many variants of the sequential feature selection idea. And one very obvious approach is reverse feature selection. In this approach, you start with all of the features and you take one out at a time until you have only as many as you desire. We can also do a process which both adds and removes features. So maybe we start with no features, then we add five features, and then we randomly alternate between adding and removing a feature, repeating this process for some fixed number of iterations.

We also see variance in the technique used to decide that one feature set is better than another. In this video, we use the error on the development set, but other possibilities have been explored including applying statistical tests to the feature themselves. All of these variants are beyond the scope of our course, but you might see them elsewhere, including in the scikit-learn library.

Video 4: A First Look at Regularization

So far, we've discussed three different models. The first was the model that we had from a previous module, which tries to predict fuel efficiency from horsepower and horsepower squared — just those two features. Then today, we generated a model which had 55 features, which for all degree three combinations of all five numeric features on our original dataset. Now this model had great training error, but looking at the development set error, we can see that it was badly overfit. Then lastly, we have the model that we just created using sequential feature selection, where we took the 55 features from the previous scenario, and then we use sequential feature selection to pick four features to keep.

In our new approach, called regularization, we're going to keep all of our features. The difference is that we will take steps to ensure that we don't use them as much. Now I mean that only loosely speaking, but it gives us some intuition for where we're heading.

So fundamentally, regularization is an approach for controlling the complexity of a model. As I mentioned before, we're going to keep all of our features, even if they were a huge number of them. These features could be from a dataset that is extremely rich on its own, like say, a housing dataset where we have many different features about a given house. Or it could be high dimensional because we generated a bunch of features from an initially lower dimensional dataset, like for example, our vehicle data today. We will see that for a regularized model, we can control complexity with just a single magical parameter — alpha. Now in a previous video, I used alpha for the intercept term of a linear regression model. But in this video, I'm using alpha in a totally different sense to mean the complexity parameter of a regularized model.

This time, I'm going to show you how to build such a model in scikit-learn first, and it will feel like magic. And then only afterwards I'll show you how it's working.

We'll start by recalling that when we fit a linear regression model on five features, we get back five parameters corresponding to the weight of each feature. For example, the code shown fits a linear regression model on our five numeric features in our vehicle dataset. We can then see the parameters by requesting the coefficients from the fit model. Suppose we now wanted to use this linear regression model to predict the fuel efficiency of an observed vehicle. Its prediction would be negative 0.228 times the number of cylinders in the engine, plus negative 0.0038 times the displacement of the engine of the vehicle, and so forth.

Now, let's create a regularized model. The only difference is that instead of creating an object of type LinearRegression, we create an object of type Ridge. Ridge is also a linear regression model. The syntax for creating and fitting such a model is exactly the same, but it requires an additional parameter — alpha. Now for now, you can think of alpha as the complexity control parameter. And that as alpha grows, the model will become less complex and our theta values will get smaller, as we'll soon see.

Now, different choices of alpha will affect the parameters that we get back. For example, our ridge regression model here has an alpha of 0.001 and we get the feature shown. You may notice that these parameters look almost exactly the same as the linear regression model from earlier. By contrast, if we increase alpha to 100, the resulting parameters are somewhat different.

Now we could do a little computational experiment by fitting a bunch of different ridge regression models, each with different alphas, and observing how the parameters change as we move from alpha equals 0.01 to alpha equal to 10,000. Observe that as alpha gets bigger, the parameters overall tend to get smaller and smaller. Now note that the magnitude of some of the parameters, like displacement, actually increases. But if we add another column showing the sums of the squares of the parameter, we observe a strict trend that as alpha increases, the sum of the squares decreases.

As a quick check of your understanding, I want you to reflect on the following question. As alpha increases, which parameters become less important to the model? Pause the video for a moment and think about this.

Since the output of our model is just the weighted sum of our features and the weights are given by the corresponding parameter, we can think about the magnitude of the parameter as the degree to which the model cares about each feature. So we see that as alpha goes to 10,000, the weight of the cylinders and acceleration features, they dropped by almost two orders of magnitude. We can interpret this as meaning that for large alpha, our model cares less and less about these two features.

It's worth noting that we can't compare the magnitude of two different columns of the dataframe to assert the importance of a particular feature. After all, the scales of the features of our model are different. For example, the cylinder values, they're all small integers. But by contrast, the weight is a large number in the thousands representing the weight of the vehicle in pounds.

Now we can keep increasing the alpha parameter up to even higher and higher values. And you'll see that the magnitudes of the weights in the features, they will tend to get smaller, and smaller, and smaller. Now individual features, like weight in this example, might not follow this trend in the short-term. But the sum of the squares of the features will inexorably decrease so that if we were to increase alpha yet further here, we would eventually see the weight feature shrink as well.

It is in this sense that we mean that ridge regression models are simpler. Even though the model still has the expressive power of all of the input features, it loses its ability to use them as much. And we'll cover how that happens exactly in some detail soon.

By contrast, let's consider the small alpha limit. As alpha gets smaller and smaller, the constraint on our model effectively disappears so that in this specific case, as alpha reaches 0.01, our ridge regression model's parameters are almost exactly equal to the linear regression model's parameters.

One way to think about the role of alpha then, is that it tries to keep the complexity of the model in check. The model wants to be very complex, using all of its features, as much as it can. But if the alpha value is large, the sums of the squares of the features will be forced to be smaller. If we pick a very large alpha, our parameters will be constrained and they will shrink closer and closer to zero. By contrast, if we pick alpha equal to zero, the model is not constrained at all and we just end up with standard linear regression.

So next, let's augment our experimental table with the training mean squared error. This will help us to understand how alpha prevents overfitting. We see that as alpha increases, in other words, the model is more and more constrained, the training error increases. For example, if alpha is 10 to the eighth, we see that our parameters are relatively small and our training error reaches an MSE of 22.

This trend is easier to see if we make a graphical plot. And so here on the x-axis we have alpha and on the y-axis we have the mean squared error on the training set. Note that I've chosen a logarithmic scale for the x-axis. And this is because our alpha value spanned many orders of magnitude. Now as alpha decreases, the complexity of our model actually goes up and our training error goes down. In other words, it's a lot like our usual figure showing complexity versus error, only it's backwards.

So we can reorient the plot by keeping the training mean squared error on the y-axis. But now plotting one over alpha on the x-axis. If we do this, the plot now matches our usual cartoon shown, where higher complexities is on the right and lower complexity is on the left. And to emphasize, the reason it was backwards before is that bigger alpha was lower complexity. So now we're effectively just flipping the plot left to right so that now complexity increases as we move over to the right.

Now in the experimental table that I generated, I only picked alpha values that were powers of 10, but they don't have to be powers of ten, they could be anything. So now in this figure, I've picked a smoother range of 100 different alpha values. And we're able to see that this alpha parameter, it smoothly affects the complexity of our model.

Recall that with polynomial degree, complexity came in these big discrete jumps — degree equals two, degree equals three, and so forth. By contrast here, alpha allows us to smoothly affect the complexity of the model. And it's interesting to observe that there's these different numerical regimes in the plot. There's this steeply dropping mean squared error as one over alpha increases. Then it levels off for a little bit, and then it becomes just a bit steeper before decaying to a fixed asymptote somewhere around MSE of 17.1.

And by the way, the reason that the mean squared error levels off at that specific value is that's the mean squared error of a standard linear regression model. After all, for very large values of one over alpha, regularization stops doing anything, and we just have standard linear regression.

So at this point, I've covered all of the really important intuitions for how regularization works. And so an important question pops up. How do we know which alpha to pick?

Now before we talk about this topic, we should talk a little bit about how regularization actually works mathematically.

Video 5: How Regularization Works

Let's now take a quick look at how regularization actually works. I'm only going to cover the most important and essential details that you'll need to apply regularization in a practical context. That's because a full mathematical treatment of the subject is just too intricate for us to be able to cover in our course.

So as a warm-up exercise, I want to review how to compute the mean squared error for a model, given the data used to train the model, and the parameters of the model. Suppose we have an input dataframe with two features, phi 1 and phi 2. And for the sake of having a manually workable example, let's assume this dataframe is tiny and only has two rows.

Suppose that our model has two parameters, theta one and theta two, where theta one is three and theta two is equal to two. Assume we also have an intercept term, b, which is equal to minus two. I'd like you to compute the predictions y hat sub i made by the model. That is, what is y hat sub one and what is y hat sub two? And then after that, I'd like you to compute the mean squared error. Now for your convenience, I put all the most important equations on the slide. And I encourage you to pause the video and think about the answer. Now you'll need to really, really understand this for the next part of this video to make sense. So I really encourage you to actually pause and take the minute or two to work out the math.

Let's check your answer. First, we compute y hat sub one, which is just three times the first feature, plus two times the second feature, plus b. That gives us three times one, minus two times two, minus two, which is just negative three. Our second prediction is three times one, plus two times three, minus two, which is equal to seven.

So now that we have our predictions, we can compute the mean squared error. The total squared error is minus three, minus negative three, which is zero, squared, plus 12 minus seven, which is five, squared. So zero plus five squared is 25. And then we divide that total squared error by the number of samples, giving us 25 divided by two, which equals 12.5.

So if you got the answers minus three, seven, 12.5, then you've understood this perfectly and you're ready to proceed. If not, pause this video and work it out yourself again before continuing on.

So next, we remind ourselves how linear regression picks its parameters. For simple linear regression, where a model has only one parameter, theta, and an intercept, b, our model fitting procedure chooses the b and theta that together minimize the mean squared error. Here we see the exact same mean squared error formula that we used in our exercise just now. Now, of course, for this model, since y hat is equal to theta times phi sub i plus b, we can also expand out the equation to show how the mean squared error depends on theta and b.

Now for a model with multiple parameters, the model fitting process instead finds b, theta 1, theta 2, and so on up to theta d, such that the mean squared error is minimized. And here, d is the number of different features in our model. Now as before, I've shown the mean squared error equation with y hat expanded to explicitly show the effects of our thetas and b on the mean squared error. Now this equation may look large and intimidating, but it's really not so bad. And in fact, this big, ugly equation is the exact equation that you just used to compute the value, 12.5.

Now ridge regression is a very small twist on the idea of linear regression. Just like linear regression, it finds b, it finds theta one, and theta two, and so on up to theta d, such that an equation is minimized. However, now, instead of minimizing the mean squared error, the model fitting procedure minimizes the sum of the mean squared error plus alpha times the sum of the squares of the parameters. We call this new function that we want to minimize our objective function.

Now notice this new term here on the left side showing the objective function includes alpha times the sum of theta one squared, plus theta two squared, plus dot, dot, dot, plus theta d squared. This term is often called the penalty term. Now because this is also the L2 norm of the model parameters, the process of adding this term to a linear regression model is also sometimes called L2 regularization. The terms ridge regression and L2 regularization are interchangeable for most purposes.

Now we could also write this extra term in sigma notation in order to make it more consistent with the right side of the equation. This equation looks even bigger, and uglier, and scarier. But again, it's not as bad as it looks, as we'll see on our next exercise.

So again, we return to the scenario from before. We have a two row dataframe with the exact same model. Theta one is three, theta two equals two, and b equals minus two. Recall that the mean squared error for the scenario was 12.5. Your job now is to compute the ridge regression objective function, if alpha equals two. Recall the right side of this equation, starting from this plus is just the mean squared error. So pause the video and see what you come up with.

Okay? Although I don't actually need to do this because we already know the mean squared error, I'm going to solve this totally from scratch using the big, scary, ugly equation.

So the first thing we do is we figure out y hat sub one and y hat sub two. That's the same thing we did in the previous exercise. And we get y hat sub one is minus three. And y hat sub two is seven. These computations come from the highlighted part of the big scary equation. Now that we have our predictions, we can compute the mean squared error. And again, we already did this in the previous exercise, giving us the number 12.5.

What I'd like to emphasize is that the right side of this equation, starting after the plus, is the mean squared error. And while all the symbols here may seem intimidating, if you just look back at the route computation we completed before, it's really not so bad.

So now the last step is just to add our penalty term. And actually this is quite simple. We add together the squares of our parameters, three squared plus two squared, giving us 13. And then we multiply that by alpha equals two, giving us 26. Adding 26 to 12.5, we get 38.5. So what does this 38.5 mean, you may ask? Well, that's the value that our model fitting procedure wants to minimize by picking theta one, theta two, and b. Note that alpha is set in stone and is not allowed to change during fitting.

So one last understanding check. Consider the two models below with the theta and b values shown. The first model has mean squared error ten and the second model has mean squared error three. Which model will ridge regression consider to be better? In other words, if the ridge regression model fitting procedure were faced with this pair of choices, which of these choices would it take? Pause the video, compute whatever you might need to compute, and come up with an answer.

Now, while it's not necessary to compute the actual objective function, I'm going to do here, do so here, so we get maximum clarity.

The top model's objective function here, its value is 10 plus alpha, times the sum of the squares of the parameters. Since the parameters are three squared, and four squared, and alpha is two, you can work out that the penalty term is 50. And that gives us a total objective function of 60. By contrast, the sum of the squares of the parameters for the bottom model is a 100, which we then multiply by alpha equal to two, yielding 200. We then add three, yielding a final objective function value of 203.

So whereas linear regression would have considered the bottom model better because the mean squared error was lower, on a dev set let's say, ridge regression would consider the top model better because the parameters are not so large. And note that our intercept term is not included in our penalty term.

So earlier we did an experiment where as we picked larger and larger alpha, we got smaller and smaller parameters. This process, by the way, is sometimes called shrinkage. One useful interpretation is that the penalty term establishes a trade-off between the size of our parameters and the error of our model. In other words, we can think of alpha as setting a budget for our model. When alpha is very small, the budget is effectively unlimited and we're just doing normal linear regression. However, when alpha becomes large, the penalty term becomes more sensitive to the magnitudes of the parameters and the overall model size budget, it becomes constrained.

We can also see this effect graphically if we plot the objective function for a model for various choices of alpha. Consider the tips dataset from an earlier module. We want to predict the amount a table will tip based on the total cost of a meal. In the leftmost figure, we see a plot showing the mean squared error for a tips dataset as a function of our parameter theta, which is the percentage tip chosen by our model. We see that the best choice is somewhere slightly less than 15 percent. Now if instead of linear regression, we use ridge regression with alpha equals one, we see that the function we're optimizing is somewhat flatter and shifted to the left. When we increase alpha to ten, the curve becomes yet flatter and shifted even further to the left. For alpha equals 100, the curve is much flatter and the optimal parameter has also moved to the left, landing somewhere around 0.1. In other words, it predicts somewhere around a ten percent tip.

Now while we can't plot the objective function for really high dimensional models, something similar happens when you apply ridge regression in those spaces. So specifically, as alpha grows, the location of the minimum of the objective function shifts closer and closer to the origin, closer to all zeros.

Video 6: Scaling

Now that we understand how regularization works, I'd like to point out a really important issue that will need to be taken care of. The penalty term affects all of our parameters equally, regardless of the scale of the parameter.

So for example, here is a large 55-feature dataframe. And we see that some of the features are small, like cylinders and acceleration, but others are large, like horsepower squared times weight, or weight cubed. Now if we have a limited parameter budget constrained by alpha, it is cheaper to spend our budget on features like horsepower cubed, since the corresponding theta only needs to be relatively small to take advantage of this large feature. Or in other words, our penalty term, it more heavily penalizes features like cylinders and acceleration, which are small in value.

Now one way to deal with this problem is to rescale our data. There are a number of different techniques for doing so. So I will highlight here the most commonly used in the context of regularization. This technique is known as standardization. When we standardize a column of a dataframe, we simply subtract out the mean of that column and then divide the result by the standard deviation of that column. In other words, we're converting each feature into a Z-score. This process is applied to each column of the dataframe independently.

Scikit-learn provides a transformer that we can use called StandardScaler. And it works just like the previous transformers. We first instantiate a StandardScaler object, and then we call fit_transform method on the DataFrame of interest. Here I have extra code that takes the result and it turns it into a dataframe with the appropriate name. And when we do so, we see that the columns are now Z-scores. For example, the first entry in the displacement column is 1.07. And this means that on the original scale, the displacement of this sample was approximately one standard deviation above the mean of all the other displacement column values.

Now since all of our features are Z-scores, they are also on approximately the same scale. As a result, regularization penalties will be applied to each column in a more equitable manner.

Now I should also note that now that we've applied a standardization to our dataframe, the mean of every column is now zero and the variance of every column is now one, and one nice thing about this process is that we don't lose any information. So if for some reason we wanted to invert the transform, we can use the inverse transform method of the standard scaler.

Now just like any scikit-learn building block, we can place a standard scaler in a pipeline. And this is useful because it avoids the need to explicitly generate a dataframe that has been rescaled. So for example, here, we augment our previous ridge model pipeline with the new scale step that comes after the polynomial feature step. So now when we call fit on our scale ridge model, the object will first add all of the new features and then it will rescale by standardizing them. And then finally, you will pass the standardized data to the ridge regression model.

You don't have to apply a standard scaler. And ridge regression may provide reasonable results even if you don't scale your data. However, if you neglect to scale, you'll see sometimes you'll end up with a warning. Now I'll say, I don't 100% know why this warning arises. But I believe the situation is that these wildly different scales of our dataframe are leading to significant issues with rounding error.

Now, even if you don't get a warning, it is possible that neglecting to include a scaler in your pipeline may result in a model which is inferior to model that includes a scalar. After all, an unscaled input dataframe will be penalized in an unfair manner, restricting your model from doing the best job that it can.

Video 7: Grid Search CV

Since alpha is a hyperparameter, our choice of alpha will affect the performance of our model. We already have one great technique for picking a good alpha, namely simple cross-validation. In this procedure we'll simply try out a bunch of different alphas and pick the one that gives us the minimum development set mean squared error.

Here I show the results of a little computational experiment where I tried a bunch of different alphas, and then computed the resulting training and dev set errors. We again see our familiar curve, showing that as complexity increases from left to right, our training set error decreases. For our dev set model, which our model never gets to see, we see that the dev set error initially decreases before increasing once the model becomes too complex and the model starts to overfit. For this specific experiment, we notice that there's a wide range of alpha values that are all about equally good. Now, keep in mind that the x-axis is one over alpha, so that the left end of this range is around alpha equal to 100 and the right end of this range is somewhere around alpha equal to 0.01. Eyeballing it, it looks like the minimum is reached at something like ten, or 20, or so.

Now I won't go over this code in detail, but I'm providing it here so you can see exactly how I generated that dataframe of mean squared error values that are used to produce the figure that you just saw. Once we have the dataframe of all results, we can get the best alpha that we found by sorting by the dev set error. And then retrieving the zeroth row. We see that the optimal alpha was around 23.

Now, while writing code like this to iterate over all alpha's works, we can save ourselves some work by using a special scikit-learn estimator called GridSearchCV. GridSearchCV is one of my favorite things to use in scikit-learn because it allows you to very conveniently compare many different hyperparameters, combinations of hyperparameters, and even many entirely different models, all in a few lines of code.

To use the GridSearchCV estimator, we first create an unfit model. Then we create a Python dictionary of all the hyperparameters and the values that we want to try. And in this case, since the hyperparameter that we want to explore is part of a name step, we have to prepend the hyperparameter name with the name of the name step, followed by two underscores. You'll get practice with this on the homework. So here, to try out a range of alphas between ten to the minus fifth and ten to the fourth, we include an entry in our dictionary that maps josh_regression__alpha to an array containing all the values that we want to try.

So next we create our GridSearchCV object, which you'll notice has arguments that are similar but not quite the same as our sequential feature selection transformer from before. We provide our unfit estimator, the parameters we want to try, the metric to be used to score the models, and lastly, our training and dev set indices.

Now the very last step is to call the fit method of our GridSearchCV object. I've named my GridSearchCV object model finder. Note that here I'm providing the entire dataset, not just the training set, because this search process requires model finder to be able to carry out simple cross-validation. In other words, it needs to have access to the development set.

So when the GridSearchCV is done, in other words, the fit method completes execution, the usual next step is to ask for the best estimator that was found. We do this with the aptly named best estimator property. Here we see that the best model had an alpha equal to 23.1, which is exactly what we saw when we did this process ourselves. The difference is that using a GridSearchCV object is much less tedious than what we had to do earlier. It's worth noting that we could at this point use the best model.predict method in order to make useful real-world predictions.

Now if we want to understand what our model is doing, we can get back the coefficients of our model as usual. We simply ask our best model for the name step called josh_regression, which will give us access to the ridge estimator that was found by GridSearchCV. We then use the .coef feature and we get back a bunch of numbers. And we can convert those into a dataframe so we can understand which weights go with which features. And that will allow us to see the relative importance of each of the features. Now keep in mind that we rescaled all of our features so that they became Z-scores, so interpreting these weights is a bit non-trivial.

Now GridSearchCV, it also allows us to see the cross-validation scores that it computed to make its decision. In other words, how it picked alpha equals 23. This can be very helpful sometimes when you're trying to debug a result that does not make sense. To see the cross-validation results, use the cv results property. And you'll get back a large summary of different aspects of the fitting process. These include things like the time it took to do each fit, the rankings of each choice of hyperparameter, and more. Now here, for this example, I've restricted my set of alphas to only three values to make the resulting results easier to interpret. The split0_test_ score, that property gives you the mean squared error of each choice of alpha. So for example, for the choice of alpha equal to 1,000, the mean squared error was 19.4. The mean_test_score, that's redundant with the split0_test_score for simple cross-validation. In a later video, we'll see a variant of cross-validation where that's not always the case. The final property in this list is the rank_test_score, giving the rel, giving you the relative rank of each choice of alpha. So for example, the middle choice, alpha equal to one was the best, which makes sense since its mean squared error was only 17.9.

Now one little warning I have to throw at you before I move on. Technically, a GridSearchCV object is an estimator, meaning that we can call model finder.predict on some data. I've seen students do this many times where they assume that model finder is now equivalent to the best estimator that was discovered. However, that is not the case. Actually, model finder.predict will actually just use whatever choice of hyperparameters it tried most recently. So if you want the best model rather than the most recent model, you should call model finder.best estimator. Don't call model finder.predict.

Now before we move on past ridge regression, let's compare the quality of our ridge regression model to our previous three models. Here in this table, I give the training error, the development set error, and the number of features for all of these four models. We observe that the ridge regression model performs well on the unseen development set, despite the fact that it has 55 features. It appears that regularization has done its job of preventing overfitting. For this specific experiment, we see that our sequential feature selection model actually did slightly better.

Now this process, and this is extremely important, of comparing models is itself subject to variance. At the very beginning of our experiment, I created a specific training and development set that was shared by all of our model evaluation processes, yielding the data for the four models, as shown in this table.

However, if I generate another different training and dev set, we can end up with dramatically different mean squared errors. For example, here I show the results of another experiment that was done in exactly the same way, except for which samples were randomly assigned to the training set versus the dev set. In the results table for this second experiment, we see two really big differences from before. The first is that that massive 55 parameter linear regression actually performed better than our two parameter linear regression, suggesting that this time around the 55 parameter model was not as grossly overfit as before. The second difference is that in the second experiment our ridge regression model performed the best.

On the homework, you'll have a chance to explore this issue more deeply, including how to use k-fold cross-validation, which we'll cover soon, to reduce this variance and dev set error.

Video 8: LASSO Regression

Earlier in the course, we saw there are many different loss functions we can select. For example, L1 loss, L2 loss, or other stranger possibilities like Huber loss.

Now similarly, there are many different choices for a penalty term. Drawing inspiration from the loss functions earlier, one obvious choice is to create a penalty term, which is the sum of the absolute values of the parameters times alpha. That is called L1 regularization, as opposed to what we saw with ridge regression, which was L2 regularization. Now, applying L1 regularization is common enough that it has a special name, LASSO regression.

I won't go over why it's called LASSO regression, but you're welcome to look that up online. Now you might say, why don't they just call it L2 regression and L1 regression instead of having these strange and hard to remember names — ridge regression and LASSO regression. Sorry, don't blame me, and somebody else named them.

So using LASSO regularization in Sklearn is extremely easy. All we do is instead of instantiating a ridge object, we instantiate a LASSO object. In this pipeline that I'm showing, I'm calling the LASSO regression step josh_lasso. We then fit as normal, either directly or by using GridSearchCV, which will give us a model just like before.

Now one note is it's pretty common to get convergence warnings when you fit a LASSO model. And the reason is that an L1 regularization results in a harder problem to solve numerically, which we might talk about briefly in a future module. Now, despite those warnings, you will see that the resulting models are still often quite good.

Now LASSO, it is more than an idle mathematical curiosity. When I run the same grid search as before, only now using LASSO regression, instead of ridge regression, as our final step, we get a model with a very interesting set of coefficients. Looking at them, we see that most of them are zero. Now this should seem a little shocking, with ridge regression nothing like that happened. Ridge regression just forced the parameters to be smaller. By contrast, LASSO regression forces the parameters to be smaller, but also forces many of the parameters to zero.

We can convert the coefficients into a dataframe just like before. And once we've done so, we could do a little bit of dataframe magic to keep only the columns which are non-zero. Here we see that the seven features that LASSO regression decided were worth its budget where displacement, horsepower, weight, horsepower times acceleration, displacement times horsepower squared, displacement times horsepower times weight, and acceleration cubed.

In other words, LASSO has automatically selected features, in this case, seven out of the original 55. And what's fascinating is that it did so not through any kind of sequential feature selection process like before. Instead, all of those zeros emerged organically as part of a global optimization process. Now when I first saw this, I was quite shocked. Somehow changing the penalty term from the sum of the squares of the parameters to the sum of the absolute values has resulted in feature selection. It boggles the mind. Now unfortunately, understanding why that happens is well beyond the scope of our class. But if you want to try searching the Internet, the key phrase to search for is norm ball.

Now the punchline here is that while LASSO regression is often slower and less numerically stable than ridge regression, if you want a model that only uses a relatively small number of features then LASSO is one potential way to achieve that goal.

Video 9: K-fold Cross-Validation

In our last new content video of the day, we're going to talk about another technique for cross-validation, this one called k-fold cross-validation.

Now before we do that, let's review simple cross-validation. Before we do anything with simple cross-validation, we split our data into two sets, a training set and the validation set. The samples in the validation set are never used for training at all. Then, to determine the quality of a particular hyperparameter, we train our model on the training set. And then we evaluate the quality based on the model's error on the validation set.

So in this figure, assuming we're doing a ridge or a LASSO regression model and we need to pick an alpha, we see that our choices of alpha yield three different sets of parameters. Now in order to decide which of these is best, we check the mean squared error for each of these parameter sets. And in this case, the middle set of parameters for alpha, the one that has alpha equals one, that one has the lowest mean squared error on the dev set of only 317. Thus, let's use that hyperparameter. We want alpha equals one, and that will give us the theta one and theta two that are shown.

Now one downside of using simple cross-validation is that some of our samples are never used for training. In some cases, data can be exceedingly hard to come by. K-fold cross-validation, an alternate technique allows us to use all of our data while still avoiding overfitting. The downside as we'll see, is that k-fold cross-validation can be much more computationally expensive, meaning it'll take much more time to select the best model.

So in this new k-fold cross-validation approach, we do not split our data into separate training and dev sets. Instead, our data is split into k different groups, often called folds. In this example, I show an example where k equals five. So we split our data into five-folds, meaning this is five-fold cross-validation. Then to determine the quality of a particular hyperparameter, we pick a fold which we call the validation fold. Then we train the model on all but this fold. And then finally compute the error on the validation fold.

Now if you're paying close attention, you'll realize that what we just did is the exact same thing as simple cross-validation. But here's where the difference comes in. We then repeat that process for all k possible choices of the validation fold. So the overall quality of the current model that we're analyzing would just be the average of the k different validation fold errors.

So for example, in this annotated figure, we see that for the first iteration, the leftmost fold is used as a validation set. Then for the second iteration of this process, the first, third, fourth, and fifth fold are used as the training data, leaving the second fold to be used as temporary validation data. We then repeat this process a third, and a fourth, and a fifth time. And then to compute the cross-validation error, we average the error computed across each of those five little experiments, yielding an overall quality assessment for the choice of hyperparameters under consideration.

You'll notice that in order to compute this score, we had to actually fit five different models. By contrast, in simple cross-validation, we only had to fit one model to assess a particular choice of hyperparameters.

So as an example, suppose we want to evaluate the quality of the hyperparameter choice alpha equals one. To do this using five-fold cross-validation, we will first use folds two through five as our training set to fit ridge regression. This yields theta one equal to 10.2 and theta two equal to 1.3, as shown. We then evaluate the quality of this model using the first fold as our temporary validation set, giving us a mean squared error of 457 on this fold. Now since we did not use fold one, the model cannot be overfit to fold one. That's the nice thing about it.

Now to continue this process, we then fit another model also with alpha equal to one. But the difference is this time we're going to use folds one, three, four, and five as the training set. And that gives us back theta one equals 9.9 and theta two equals 1.2. Now this model is similar, but not identical to the model that we trained earlier on a different set of folds. We then evaluate the quality of that model, giving us a mean squared error of 456.

Now we repeat this process a total of five times, yielding five different ridge regression models with alpha equal to one. Ultimately, those five models will then have five different mean squared error values, each that are computed on a different temporary validation set. In order to give an assessment of the overall quality of this hyperparameter choice, we simply compute the average of these five MSEs, yielding 489.8.

Now to test your understanding, consider here the full output of a three-fold cross-validation procedure for four different choices of hyperparameters. For example, in the top left, we see that for alpha equal to 0.01, the three different models that we computed had mean squared errors of one, three, and six, respectively, on validation folds one, two, and three. Given what we see on the slide, which alpha should we pick? So pause the video and then I will tell you the answer.

So from what we see here, the best alpha was alpha equals one, as it had the lowest average mean squared error of all of our possible choices of hyperparameter.

So the very last step after picking our hyperparameter is to train our model. Now you might think, why are we training a new model? We actually already have three that we've train. So while true, each of those models was only trained on two thirds of the available data. By using all of the data, we can get a more accurate model.

So another question comes up. How do you know which k to pick? In other words, how do you know how many cross-validation folds you need? This is actually a very deep question that's an active area of research. And so I'll say that common choices include k equals five, k equals ten, and k equals n, where n is the number of samples that you have. Now, as you'll see on the homework, as you increase k, the noise in your cross-validation errors will tend to decrease.

And that very special case where k equals n is also known as leave one out cross-validation. In this approach our temporary validation sets are always going to be exactly one sample and every single data point gets a chance to be the validation set.

Now while leave one out approach to cross-validation makes your cross-validation errors less noisy, it is also extremely computationally expensive. If you have 10,000 data points, it means that for every hyperparameter that you want to evaluate, you have to fit 10,000 models.

Lastly, let's discuss how to conduct k-fold cross-validation in scikit-learn. Now writing code from scratch that performs k-fold cross-validation is really tedious. But luckily, it is very easy in scikit-learn. The only difference is that rather than providing a list of a list of training indices and dev set indices, as we did before, we're only going to give a single number as the CV argument. So for here in this example, we see that if we set CV equal to five in the constructor of this GridSearchCV object. Then the resulting search, when we call fit, it's going to be five-fold cross-validation. In other words, it's going to be maybe five times as slow as simple cross-validation. But the quality of the cross-validation results will be better.

Video 10: Conclusion

We've finished another module of machine-learning, whoo. We've covered a lot of ground, which I will summarize somewhat briefly.

So in a previous module, we saw how nonlinear features can greatly expand the expressive power of linear models. The central idea of this module was how to harness all of that expressive power in a way that yields models that perform well on unseen data. Now some data is inherently high-dimensional, like the housing dataset or data like images. And other data is high-dimensional because we apply preprocessing steps, like polynomial features. We saw in either case, we can apply two different approaches for intelligently harnessing that model complexity and preventing overfitting.

The first was sequential feature selection, which was a greedy algorithm that explored only a small subset of the possible combinations of features that we could use for our model. The inspiration with this approach was that searching all combinations would just take too long. And when sequential feature selection terminates, we're left with only a small subset of the features which we then use to fit a standard linear regression model.

The second approach we discussed was regularization. In this approach, our model still has access to all of the features, unlike sequential feature selection. And unlike sequential feature selection, regularization is not a greedy algorithm. It is a globally optimizing algorithm that finds the minimum in a really high-dimensional space.

So for example, if we have 55 different parameters, a regularized linear model finds the point in this 55-dimensional space that minimizes the regularization objective function. For very small alpha, that objective function is just the same as the usual mean squared error from linear regression. And so our regularized models are equivalent to linear regression. But for very large alpha, we start to get penalized as we move away from the origin of the parameter space. And so the optimal model parameters will shrink towards zero.

We saw two different flavors of regularization in this model. The first penalized the sum of the parameter squared. And we called that L2 regularization. The other flavor penalized the sum of the magnitudes of the parameters, and we call that L1 regularization. Now there are other penalty terms you could come up with other than L1 and L2, but we won't discuss them in our course. L2 regularization is faster and more numerically stable. Whereas L1 regularization sometimes has trouble with optimization for reasons that we'll touch on a bit in a future module. However, L1 regularization does give you some degree of automatic feature selection. Now we won't discuss why automatic feature selection happens as a byproduct of L1 regularization because the topic is just too technical for our course.

We also saw that when building regularized models, that we have to scale our data, for example, by standardizing it into Z-scores. At least if we want to get the best possible results.

We also saw a really nice library for hyperparameter and model selection called GridSearchCV. This is a special scikit-learn module that automatically performs cross-validation, making the search for optimal hyperparameters much more convenient, that is, rather than manually tabulating a bunch of development set mean squared errors, GridSearchCV does that for us. And GridSearchCV could be used not just for hyperparameter selection, but also for comparing completely different choices of model.

Lastly, we saw a new flavor of cross-validation, which we called k-fold cross-validation. K-fold cross-validation does the same thing that our older simple cross-validation procedure does, namely, it gives us a way to estimate how well a model generalizes. The difference is that k-fold cross-validation, we use all of our data for both training and cross-validation. The downside is that this cross-validation procedure takes longer than simple cross-validation, which we saw before.

Okay, everybody that is it for today. So I will see you next time.

Video 11: Human Resources Applications

You now have the tools to build and use predictive models. And while these tools come in many shapes and sizes, they're all designed to forecast and predict outcomes based on the data that you provide. Thus, the outputs, or the predictions, are always a function of the inputs, or the data.

The point of this lesson is to consider situations when you might want outputs, or predictions, that look different than the data you use to forecast, recognizing that there may be some benefit from learning or exploration.

The practical application here is to consider designing and implementing algorithms in the context of a human resources decision. Businesses are increasingly looking for ways to automate labor-intensive processes and recruiting is a prime example. Companies often receive hundreds or even thousands of applications for a single job listing, and the vast majority of these applications are not qualified for the listing. So companies might consider an automated screening tool for screening applicants based on qualifications. And that's fairly straightforward to do. But what do we mean by qualified?

The simplest version of this would be to consider qualifications based on inputs. For example, does the applicant have a college degree? Is that degree in a relevant field? Do they have relevant work experience, et cetera?

A more nuanced and arguably more data-driven approach might try to determine, using data and modeling techniques, what applicant characteristics have been traditionally correlated with success among our existing employees?

For example, imagine you have performance scores of your current employees on the job. And you have some observable characteristics of these same employees when they applied for your position. You could then relate the current performance metrics to observable characteristics of the same applicants prior to hiring them. However, these models will not be able to do a great job of predicting performance of applicants that look very different from those currently in your workforce. This isn't great for workforce diversity. It may also not be great for performance or productivity.

Thus, we might want a different tool, a tool that optimizes hiring based on existing data, but also prioritizes hiring candidates that look different from the existing workforce in an effort to gather more data and learn from them. We could even have this learning process feed back into the model for future applicants. This is a form of what in data science we refer to as a bandit problem. To find the best workers over time, firms must balance exploitation, or selecting from groups with proven track records, with exploration, basically selecting from under-represented groups to learn about quality.

A group of economists from MIT and Columbia recently built a resume screening algorithm that values exploration by evaluating candidates according to their statistical upside potential. Basically, this means selecting candidates with characteristics for which there is more statistical uncertainty. They are able to show that this approach improves hiring quality of selected candidates, while also increasing demographic diversity relative to the firm's existing hiring practices. The same is not true for traditional supervised learning-based algorithms, which improve hiring quality, but select far fewer black and Hispanic applicants.

The researchers built three different resume screening algorithms, two were based on a supervised learning algorithm, and one is based on a contextual bandit approach. And they try to evaluate the candidates that each algorithm selects relative to the actual interview decisions made by the firm's human resume screeners. They observed data on an applicant's demographics, their, their education, their work history. And the goal here is to maximize the quality of applicants who are selected for an interview.

The first algorithm uses a static supervised learning approach based on a logit LASSO model.

The second algorithm builds on the same baseline as the LASSO model, but it updates the training data it uses throughout the test period with the hiring outcomes of the applicants it chooses to interview. While this updating process allows the LASSO model to learn about the quality of the applicants it selects, it's myopic in the sense that it is not incorporate the value of this learning into its selection decisions.

The last approach that the researchers use implements what is known as an upper confidence bound, contextual bandit algorithm. The LASSO algorithms evaluate candidates based on their point estimates of hiring potential. This new algorithm selects applicants based on the upper bound of the confidence interval associated with those point estimates. That is, there's going to implicitly be an exploration bonus that's increasing in the algorithm's degree of uncertainty about quality. Exploration bonuses are going to be higher for groups of candidates who are under-represented in the algorithm's training data, because the model will have less precise estimates for these groups.

There are two main results in the paper. The supervised learning, or LASSO models, differ markedly in the demographic composition of the applicants they select to be interviewed. Implementing an upper confidence bound model would more than double the share of interviewed applicants who are black or Hispanic from ten percent to 24 percent. Both the static and the updating LASSO models, however, would both dramatically decrease the combined share of black and Hispanic applicants to two and five percent, respectively.

The second main result is that all of the machine learning models increase hiring yield relative to human recruiters. However, this last comparison is not experimental in nature. Ideally, one would like to use a randomized controlled trial or an AB test to randomly assign applicants into one of four resume screening tools and potentially a control group — a human recruiter LASSO, supervised LASSO, or upper confidence bound models. And then we would evaluate ex-post performance. This would allow the researchers to estimate causal effects of each different treatment arm and evaluate relative performance differences of all applicants, holding all else equal.