Module 8: Feature Engineering and Overfitting

Video Transcripts

Video 1: Parabolic Model Fitting

Last time, we discussed linear models for regression. In a linear model, predictions are a linear combination of the available features. So for example, if we have three features, the model predicts an output y hat equal to theta one phi one, plus theta two phi two, plus theta three phi three. More generally, we can express the output of the models as follows. If d is the number of features, y hat is the sum from j equals one to d of theta j times phi j. Since we always have the same number of features as parameters, d is also the number of parameters. If we choose, we can also include an extra intercept term, which here I'm calling alpha.

Let's consider a challenge that we've faced with linear models. Here's a plot showing the fuel efficiency versus the engine power of many different models of vehicle. On the x-axis, we have the total engine power in horsepower. If you're not familiar with the term horsepower, horsepower is just equal to 745.7 watts. Now on the y-axis, we have the fuel efficiency in miles per gallon. You may be more familiar with the idea of liters per kilometer. Miles per gallon is just an alternate metric for fuel efficiency that's used in the United States. Higher miles per gallon means better fuel efficiency. The idea being that you're able to travel a larger distance with the same amount of fuel.

Now observing this data, we see a strong inverse correlation. As the engine power grows, the mileage per gallon decreases. This makes intuitive sense

because an engine which is more powerful will tend to be part of a larger vehicle. And since larger vehicles have more mass to move around, we expect them to be less fuel efficient.

Now suppose we want to build a model to relate miles per gallon to horsepower. Our only tool so far is simple linear regression, yielding the model you see here with mean squared error equal to 23.94. We notice that this model fails to capture the shape of the relationship between the fuel efficiency and the engine power. The shape of the relationship, it feels sort of arcing or curved. But because we're using simple linear regression here, we can't possibly get that curvy shape. With what we know so far, it seems impossible to capture this nonlinear relationship between the miles per gallon and our horsepower.

One approach would be to attempt to fit something else instead of a line. For example, what if we tried to fit something like a parabola? Eyeballing the data, the bottom of the parabola, it looks like it's maybe somewhere around x equals 250 and y equals 10. Now looking up as the parabola swoops up to the left, another point on the parabola looks like it might be x equals 75, y equals 27.5. That's just a rough guess, but it'll do for our first attempt. Turns out that if we add the coordinates of the vertex of a parabola and any other point on the parabola. These two points are enough to derive the equation for the parabola. The resulting equation is y equals x minus 250 squared over 1,750 plus 10. I didn't show my work for this basic exercise in algebra, but you might find it interesting to pause this video to derive it yourself.

Plotting this equation, we see the bottom is right here, at the point that I picked. And it arcs up and over through the other point that I picked. It is

instructive to also consider what this equation looks like if we expand it out. This yields y equals x squared over 1,750, plus two-sevenths x, plus 45.7. We can compute the mean squared error on this model. And we get back a value of 21, which is much better than we got using simple linear regression.

Video 2: Nonlinear Features

Naturally, a simple linear regression model could never produce these predictions because simple linear regression models do not have a squared term. Suppose we wanted to generalize our idea of a model with squared features. The naive approach would be to create a new type of model, which I'll call a squared regression model. In such a model, predictions would be the weighted sum of the features, plus the weighted sum of the features squared, plus an intercept term.

In the hypothetical universe where the scikit-learn library included square regression, the code to fit such a model might look like this. We would say, from the nonlinear model library, import SquaredRegression. We would fit a squared regression and we would get some kind of curve like the one we just saw. As it happens, there is no such thing as a squared regression in scikit-learn. And it's for a good reason. Turns out there's a much more clever and simpler solution where we can still use the same linear regression library but get nonlinear behavior.

Now this is going to seem like a trick, but this is really what's done in the real-world and it works very well. All we do is add a squared feature to our model. What do we mean by that? Well, as an example, consider our original DataFrame, where the first row is a vehicle with a 130 horsepower engine and a fuel efficiency of 18 miles per gallon. We're just going to add an

additional feature equal to the horsepower squared, which I will call hp2. For this row, hp2 will be equal to 16,900, because 130 squared is 16,900. On the next row, 165 horsepower results in an hp2 of 27,225.

Now after I've added these features, I say, hey, linear regression model, I'd like you to fit the vehicle data using both hp and hp2. Since the mathematical equations are of exactly the same form as the parabola that we manually fit earlier, we get a nice parabola. The nice thing about this is that we're using the same code as we did for linear regression in an earlier module. The only difference is that we've included a new column in our DataFrame.

Comparing the model that I created manually versus sklearn's model, we see they are very similar. They're both parabolas. They both have an output y hat equal to phi 1 times horsepower, plus phi 2 times horsepower squared, plus an intercept term. Though, their phis are different. For example, for my model, phi 2, the coefficient for horsepower squared, is 1 over 1,750. And for scikit-learn, phi 2 was 1 over 812.68. This difference in phi values can be observed visually. For example, we observe that the sklearn's parabola is steeper. And we also see that in terms of error, the sklearn model does better. The model I made up by eyeballing the data, had a mean squared error of 21. But scikit-learn's model, which actually optimizes this to find the absolute best parameters, had an MSE of 18.98.

Video 3: Prediction vs. Inference

Let's consider the two linear regression models that we've built using scikit-learn. Recall that the only difference was the DataFrame that was provided for training. For the left figure, the data provided only a single feature for prediction, the horsepower. For the right figure, that data provided two

features, horsepower and horsepower squared. Both of these linear regression models only makes sense in the range of data that they've seen. Going outside of this range yields strange predictions.

For example, for the left model, it predicts that as you go up to 300 or 400 horsepower engines, fuel efficiency will become negative. Now that obviously doesn't make any sense, but that's what the model would predict. The right model also fails for very high engine power. Specifically, it predicts that as horsepower surpasses 200, the fuel efficiency will start to increase parabolically. In the real-world, this makes no sense. Gigantic vehicles with engine power in the thousands of horsepower should not be expected to have amazingly good fuel efficiency. And yet, this model can actually be useful, so long as we stick to the range in which the model was trained.

This is a good time to bring up the distinction between two distinct goals when creating models, inference and prediction. Prediction means creating a model which can provide accurate predictions for new real-world inputs. For example, I hear about a new vehicle that's been produced it as a 160 horsepower engine. I can give you back a prediction that is likely correct using my model.

By contrast, if inference is our goal then we're trying to understand the true relationship. So in this case, the model would capture the actual economic and physical realities that lead to this observed relationship between fuel efficiency and vehicle size. It would answer the question, why does fuel efficiency drop quickly as horsepower increases, before levelling off at some relatively constant value? Inference is a much harder task.

For our course, we'll concern ourselves almost entirely with prediction, rather than inference. So if you start getting nervous about the fact that your model is making nonsensical predictions for data outside the range in which it was trained, Don't worry, this is perfectly normal.

Video 4: Scikit-Learn Transformers

An interesting question arises. Could we get an even better model by adding the horsepower cubed? After all, going from horsepower to horsepower squared seemed to make a really big difference. So maybe if we add horsepower cubed, we'd do even better. This is easy enough to try. We simply create a new column called horsepower cubed. And for example, for the first row, horsepower is 130, horsepower squared is 16,900, horsepower cubed is a little more than 2 million. Now you may feel like these different features are redundant because, after all, you can compute them from the horsepower column. Even so, including them does give the linear model more expressive power. Linear models cannot generate these higher-order terms during the fitting process. So we're helping them out by providing these higher-order terms in advance.

When we run the code, we see that including horsepower cubed didn't really seem to make much difference. The MSE was very slightly better, and the curve looks a little different, but for the most part it's pretty similar to what we just saw before. But let's not stop there. Let's see what happens when we go to even higher order models. And here, by order, I mean the degree of the generated polynomial features where squared features are degree two, cubed features are degree 3, etc.

We could try this out by manually creating an hp^4 feature, an hp^5 feature, and so forth. However, I'd like us to take a step back and write code that

generalizes more easily. Our goal in the next couple of minutes will be to create this DataFrame shown with an hp, hp^2, and hp^3 column, but without having to manually create each column.

In our new approach, we're going to use a special object in scikit-learn called a transformer. A transformer takes a set of existing features as input and outputs new features. As an example, this code creates a PolynomialFeatures transformer. The constructor for PolynomialFeatures wants to know the degree for the new transformer, which we set to three in this example. The PolynomialFeatures transformer, like all transformers, has a method called fit_transform. It takes the value given and produces a set of input features corresponding to the zeroth, first, second, and third power of the input value. So if the input is 5, it gives me back 1, 5, 25, and 125. In other words, this transformer has automatically generated all the features of degree three from our original value of five.

If we apply this transform function to our entire DataFrame, the result is a large two-dimensional array, with each row corresponding to a sample from our original input DataFrame. For example, observe that the first row is 1, 130, 16,900, and a little more than 2 million. In other words, the features we expect to see.

At this point, we're still getting to our goal, but we have a couple of problems. The first is that the output that we've received is a NumPy array, not a DataFrame. And that's just how transformers work. So to convert our NumPy array provided by the transform into a DataFrame, we simply call pd.DataFrame on the output of the fit transform. We're still not quite there though, because we need to get rid of this extraneous column of all ones on the far left. That leftmost column of all ones is known as a bias feature,

which is always equal to one. And if you don't want that bias feature, you can get rid of it by setting include_bias to False in the constructor for the PolynomialFeatures object. Now that we've changed how the PolynomialFeatures transformer is instantiated, the output of the fit_transform function no longer has this bias column.

There's one last thing we need to do, rename the columns hp, hp^2, and hp^3, rather than their current names, 0, 1, and 2. The PolynomialFeatures object has a really cool function called get_feature_names_out that will automatically return all the generated names for features based on the original names that it saw when it carried out the fit transform. In this case, when we ask for those names, we see that it gives us hp, hp^2, and hp^3. By providing these names to the DataFrame constructor using the columns keyword, we get a nice DataFrame with useful column names. The names that it came up with are slightly different than the ones I did, as they also include a little carrot to indicate exponentiation. Honestly, I like theirs better, so good job sklearn.

Let's compare these two approaches. In our first approach, we manually created the hp^2 and hp^3 features. And in the second approach, we used this new polynomial transformer. At first glance, this second approach seems way more complicated. There's all this new syntax to learn and understand, and you may be thinking, why are we doing this? I acknowledge that it is more complicated, but the code is also much more general. We can change the degree to 4 or 5 or 50 with just a single number. And as we'll see later, PolynomialFeatures has these additional really useful behaviors that we'll learn about in due time.

Video 5: Scikit-Learn Pipelines

Suppose we have the DataFrame containing only hp, hp^2 and hp^3 of each vehicle. Suppose we then train a linear regression model on this DataFrame using the code shown. Note that I didn't actually specify which features to use in my left argument to the fit function. This is because my DataFrame only contains the hp, hp^2, and hp^3 features. So there's no need to list them out individually. The model will use all three of these available features to fit a model to predict fuel efficiency.

There's one last annoyance we're going to need to tackle before we can use this model. For example, suppose we want to make a prediction for a new 100 horsepower vehicle. Based on what we saw in the previous module, we'd expect to be able to simply provide the value 100 wrapped in double braces. Unfortunately, linear regression is expecting three features and so it crashes. There are a couple of workarounds we might try.

The first is to manually create the squared and cubic features. We can say, okay, hypothetically, if I had a 100 horsepower vehicle, well, it has a 10,000 horsepower squared, and a 1,000,000 horsepower cubed. And manually type in all three of these values. This will work fine, but it's pretty annoying to do in practice. Another workaround is to realize, hey, we have this polynomial transform. Let's just use that. So we'd say, hey, cubic model, please predict the fuel efficiency of a vehicle whose features are the result of applying a degree 3 polynomial transformer to 100.

So we have these two workarounds to use our cubic model. And while the second workaround is somewhat better, I still don't like it. Both still feel unnecessarily verbose. Luckily, there is a better way. This next approach is oh, so gloriously elegant. But it does mean that there's even more

mysterious syntax that you're going to have to learn. This next approach is to use an sklearn pipeline. I use these all the time. The idea is that a scikit-learn pipeline lets you set up a sequence of useful data processing tasks that will be done sequentially.

As an example, the code shown is a two-stage pipeline, where the first stage is a PolynomialFeatures generator, and the second stage is a LinearRegression model. The syntax here is we create a Pipeline object, and each stage of the pipeline has a name, which is just some arbitrary name that you pick as a programmer. In my case, I named my first stage josh_transform, not because it transforms data into Josh, but because I'm Josh and that's the name that I picked. You'll observe that this transformer is just our degree three polynomial transformer from before. Now similarly, my second stage is named josh_regression. This stage is just a standard linear regression model.

Now the really cool part is that after creating this pipeline model, when we call .fit, all of our horsepower data is transformed using the degree three transformer. And then provided to the next stage, the LinearRegression object. The resulting pipeline code does the exact same thing as the code before, where we had to do the transform ourselves and then create the model separately. Observe that the code using the pipeline code is much simpler to read. Why do I like this more? There are a few advantages of this approach.

Most notably, we can apply this model to new data directly. We could just say, pipeline model, please predict the results for a 100 horsepower engine. And it will do so without the need to somehow generate the squared and cubic features ourselves. Another nice advantage is that we don't need to

explicitly create a new DataFrame of new features when we're training the model. And lastly, you don't have to keep track of separate transformer and regression objects that end up floating around in the variable scope of your Jupyter Notebooks.

Now the downside of working with the pipeline model is that it takes a little more syntax for some tasks. For example, suppose we want to know the coefficients for the linear model once it's been fit. In order to do so, we first have to retrieve the specific piece of the pipeline that we're interested in. And we do that using the named_steps attribute of a pipeline object. For example, the line of code shown gives us access to the linear regression model itself. That is, we say, pipeline model, give me the named step called josh_regression. Once we have that piece of the pipeline that we want, you can use it as usual. For example, by requesting the .coef attribute as shown.

As a small test of your understanding, let's consider briefly this puzzle. Suppose I fit a pipeline model exactly like we just saw. And now, suppose we reach inside of the pipeline, and we get the named step called josh_regression. The challenge for you is to fill in this blank so that the result would be the prediction for the miles per gallon of a vehicle with the 100 horsepower engine. Feel free to pause the video, or just wait a moment and I'll spoil answer.

Now for this example, turns out we have to provide it with all three features. Either by manually generating them, or by using a polynomial transformer. Now you might say, what? I thought the whole point of all this new syntax that we're learning is I don't have to create the higher-order features myself. And yes, that is true. So long as you provide the input to the pipeline as a

whole. However, in this contrived exercise, I've pulled the regression stage out by itself, which requires you to provide the three features explicitly.

Now to wrap up this entire scikit-learn pipelines idea, I just want to kind of go over what we saw. We saw that a scikit-learn pipeline model has multiple stages. And in our example, we just had two. The first stage was this polynomial transformer, and the second stage was this linear regression model. And we can use a pipeline both for fitting and predicting. And the really nice thing about a pipeline is we never have to call the polynomial transform methods ourselves. We never had to call fit transform. Now while the syntax is awkward at first, scikit-learn pipelines will ultimately make your life much easier. And that will be especially true once we have pipelines of more than two stages.

Video 6: Order 0 through 6 Models on Vehicle Data

Now that we're armed with this nice idea of a pipeline, we can explore higher degree model territory. If we want to try out a new degree, like degree four or five, we simply set the degree parameter to the value of our choosing. So here we see plots and mean squared errors for models of order zero, one, two, three, four, and five. Our degree 0 model has a mean squared error of 60.76. Our simple linear regression model has a mean squared error of 23.94. And our degree 2 or squared model has an MSE of 18.98. And the same for our other degrees.

Now we note that as we move on to degree four and five, the mean squared error continues to decrease slightly. In other words, the model gets better and better. Increasing model complexity seems like a strictly better proposition here. As degree goes up, error goes down, with no exceptions. And in fact, you can prove that, at least on a computer without rounding

error, bringing the degree higher and higher will always bring the error lower and lower. This cartoon plot shows the relationship between the error or the loss and the model's complexity. And this trend will hold true for almost any model, for almost any problem, not just this specific example. Here, I'm annotating the line with the words "Training error", since we're discussing the error on the data that was used to train the model, as opposed to future data that we have not yet collected.

Now as it turns out, there was a subtle but very important downside of increasing the model complexity. Observe that the curve gets a little bit more wiggly as the additional polynomial degrees give more expressive power to our linear model. The result is that the model gets more and more sensitive to the data. To get a better sense of what I mean when I say sensitivity, let's consider the following experiment.

These two figures show two squared models fit to very similar datasets. In fact, the only difference between those datasets is that I move one of the points with x equals 160 up to around y equals 26 in this right figure. Observe that these two models of order two are quite similar. By contrast, here we see degrees six models fit on the exact same two data sets. Even though these two datasets differ in only a single point, the shape of the model curve is quite different. That shift in a single data point caused that significant change in the predictions made by the model. It is in this sense that the model is sensitive to the data. Even a relatively minor perturbation to the input data results in a significant change in prediction.

In machine learning, the word variance is often used to represent the sensitivity of a model to the training data. Based on this experiment, we say that the order two model has low variance, and the order six model has high

variance. This is important because anytime you collect real-world data, whether its surveys conducted via door-to-door knocking, or pictures of human faces on a social media platform, or records of financial transactions, you'll get some degree of random noise. We saw this when we talked about random variables and about sampling.

So going back to our cartoon picture, we see that while model complexity does decrease the training error, it also increases the sensitivity of the model to all the tiny details of the data. In some of these details are really just noise. Being overly sensitive to our data is also known as overfitting. We'll see shortly why it can be a serious problem.

Video 7: The Dangers of Overfitting

To understand why overfitting is dangerous, let's consider a little toy problem. So we're going to do a regression model, but this time on only a four data point set. So those data points are 0,0, 1,3, 2,2, and 3,1, as shown on this plot. Now it turns out that because we only have four data points, that a degree three model can exactly, perfectly model this data. In other words, there's a theta one, theta two, theta three, and alpha for this model that goes through all of these points with absolutely zero mean squared error. We can generalize this and say that if we have an N datapoint input set, that a degree N minus one model will also have mean squared error equal to zero.

Now to see why this is the case, all we need to do is plug our four points into our model equation and then solve for theta one, theta two, theta three, and alpha. For example, let's plug in 0,0 into our equation. We get alpha equals zero. If we plug in x equals one, y equals three, into our model equation, well, we replace all the x's by one, giving us theta one, plus theta

two, plus theta three, plus alpha. And we know that that must equal three. Then we plug in x equals two into our equation, giving us two theta one, plus four theta two, plus eight theta three, plus alpha equals two. And I'll leave the fourth equation for you to think about to check your understanding.

Now this yields a system of four linear equations and there are only four unknowns. And so, assuming that these equations are linearly independent, that means that there exists a single unique solution. And that solution is theta one equals nineteen-thirds, theta two equals minus four, theta three equals two-thirds, and alpha equals zero. If we plot the resulting curve, we see that it is a perfect model of our data.

We can also do this in sklearn. So if the arbitrary data we have is stored as a four row, two column DataFrame, we can just call fit using the pipeline shown. Then, when we request the coefficients and the intercept from that scikit-learn model with this code, we see that they are what I showed on the previous slide. For example, theta one is nineteen-thirds, and so forth.

Now you may notice that alpha is 10 to the minus 15th, but it's supposed to be 0. And that's just because there's some rounding error in a real computer. But for linear model prediction purposes, 10 to the minus 15th and 0, they're effectively the same number.

Now another approach, if you have linear algebra background, is as follows. What we're trying to do is solve the equation Y hat equals phi theta, where phi is a matrix of all of our features, and theta is a column vector of all of our parameters. To solve for theta, we can left multiply both sides of this equation by the matrix inverse of phi. In other words, compute the inverse

of all of our features, multiply them by the output values that we're trying to predict, and we'll get back our thetas.

You can actually also carry out this computation yourself in Python without relying on scikit-learn at all. You could do this using NumPy's linear algebra solver or, less efficiently, by explicitly inverting the phi matrix and carrying out a matrix multiplication using the matrix multiplication operator in Python.

Note that this linear algebra perspective, it's only for those of you who already have some prior background in linear algebra. If you don't, then don't be too concerned about these ideas. This is just another way of understanding why a degree three model on degree four data yields a perfect fit.

What we've just observed is that if we have four data points, we could always find a four parameter model, three coefficients and one intercept, that fits perfectly. Similarly, suppose we have a 100 data points, each representing an observation about a vehicle. In theory, if we create a degree 99 model based on the horsepower alone, of each vehicle, we will be able to perfectly fit our data. It will have an error of literally zero, but unfortunately that model will be totally useless. The problem that we're facing here is overfitting.

In this case, our model is just memorizing the data that it sees, and it can't handle new situations at all. And to really see why overfitting is a problem, let's consider the process of fitting a degree five model to a random sample of six vehicles from our full data set. No matter which six vehicles we pick, this one or this one, we'll always get a perfect fit. So if we pick these six vehicles on the left, we get this weird model, which predicts negative fuel

efficiency for some values of horsepower, but it's going to be perfect for the six provided data points.

Similarly, this other model, it makes totally ridiculous predictions about horsepower vehicles in the 170 horsepower range. But for the six data points that it saw, it has zero mean squared error. If we overlay one of those perfect fits on the dataset rather than just examples, guess what? It has terrible error on the real-world data. Sure, the model gets zero error on our training set. In other words, on the data it got to see. But on a bigger collection of real-world data, the model will do terribly because this model is really, really overfit.

A similar phenomenon can occur even if we don't use a transformer. For example, suppose I'm trying to predict the price of houses based on various features. If the number of numerical features we have available exceeds the number of house data points, and we use the linear regression model, we'll see the exact same problem, where the training error is zero, but the model does not generalize well to new house observations. While these examples are extreme, even on real-world problems, you will always need to be concerned with overfitting. Indeed, the rest of this module is all about keeping overfitting under control.

Video 8: Using Feature Data to Detect Overfitting

Next, we're going to loosely derive a widely popular technique for detecting overfitting, known as simple cross-validation. For this exercise, I'm going to restrict ourselves to just 35 data points because it's easier to see what's going on. We'll start by calling this line of code to collect a 35 data point sample of our vehicle data, which I also show as a scatter plot. Now as before, we're going to try fitting a bunch of different models of different

degrees. And then we'll also compute the mean squared error on the training set.

The get_MSE_for_degree_k_model function shown fits a pipeline model of degree k. It computes the mean squared error and then it returns the mean squared error value. This table here, it shows the results of this function for each k. For example, a degree k equals 0 model has a training mean squared error of 72 on our original 35-point dataset. A degree k equals 1 model has a training mean squared error of 28 on our 35-point dataset, and so forth. As we observed earlier, this table shows that the mean squared error decreases monotonically as k increases.

We can also look at this visually by creating a plot. And if we do so, we see that as the degree goes up, the mean squared error on the training set goes lower and lower and lower. And from this plot, we have no idea if we're overfitting.

Now another visualization that we can try is to plot each model over the dataset. And here I'm showing the degree zero, one, two, and six models. The degree zero model is just a flat line. The degree one is a sloped line, degree two is a parabola, and degree six is this more wiggly shape. Now just staring at this, one has the creeping sense that yeah, this degree six model does seem a little overfit. More specifically, we believe that if we collected more data, in other words, we drew more samples from the original distribution of data, that our model would make poor predictions on that new data.

So suppose we collect these nine new orange data points in the future. These plots show those nine new points overlaid on the existing models of degree zero, one, two, and six. Now very importantly, we do not give those

models a chance to refit themselves using those nine new data points. After all, what could be a better test of a model than being faced with a new real-world observation that it has never seen.

For example, we see that the degree six model performs really poorly on that one new orange datapoint around x equals 190. In other words, the error on that point is high. Now to more thoroughly compare our models, we can compute the mean squared error for all of our models on just those nine new points, without making any changes to the model's parameters. That gives us this table. From this table, we see that our degree two model seems to actually be the best. In other words, even though the degree two model's training error is higher than the more complex models, on a new dataset with never before seen data, the model two performed best. Now, since that's what our model is going to be used for, that is, looking at new data, that's the model we should probably pick.

Now this procedure works quite well. We collect a bunch of data, we fit a bunch of models, and then we wait a while for new observations to arrive or to be collected. And then we use that second round of observations to select between models. Now the obvious downside of this approach is we have to wait for the new data after we fit the model. In some contexts, acquiring this additional data is too expensive. It'll take too long. Or maybe it's actually just impossible because we're modelling an event from the past that will never recur.

Video 9: Simple Cross-Validation

Rather than waiting for future data to estimate the generalizability of our model, cross-validation allows us to do so with data that we already have.

In this video, we'll discuss a flavor of cross-validation called simple cross-validation.

So in simple cross-validation, we split our data into two random, non-overlapping sets. The first set is our training set, which we use to train our models. The second is our development set. And we use that to compare models after they are already trained. In other words, the development set is data that we hide from ourselves, in effect, pretending like it comes from the future.

Unfortunately, the terminology around this concept is confusing and it's used inconsistently between different researchers and practitioners. I should note that the development set is also known as a validation set, and sometimes as a cross-validation set. For the purposes of our course, we will use these terms interchangeably. But be aware that in the real-world, you may find that you need to choose a specific term in order to communicate well with other practitioners.

Splitting the data is a straightforward process. First we shuffle the data, and we do that to randomize the order of our observations. And to do so, what I'm doing in code is calling sklearn.utils.shuffle. Then I use the NumPy split function, which splits an array into two or more pieces. Here, my second argument, the split function, is the number 25 in braces, which tells NumPy that the first split should be of length 25. I assign this first split to a variable called training_set. And then the remaining rows, numbering ten in total, will be returned as the second split, which I assigned to a variable called dev_set.

Whatever technique that you use to generate a training and dev set, it is extremely important to make sure that the splits of the data are random,

which I achieved by shuffling the data. You really don't want to have the training set be the first 80% of your observations, and the dev set be the last 20%. That's because the data that you get, it may be ordered in some way.

So for example, imagine you're trying to predict house prices from a dataset ordered from low to high price. In that case, every single observation in the dev set is going to have a price that's higher than any observation in the training set. In other words, you've hidden the top 20% of expensive houses from your model while it's being trained. Meaning that it is unlikely to perform well for this range of house prices.

There are other more user-friendly libraries out there to create training dev set splits, and you're welcome to use those instead. Here, in this example, I'm showing both steps of the process explicitly, shuffling and splitting, to better explain what's going on.

Now I should note that, because my original 35 data points were themselves a random sample, I don't actually need to shuffle here. But I think it's good practice to do so, just in case your data has order that you're not expecting.

So now here, we see our training and dev sets, which have 25 and 10 rows respectively. We train our models on the 25 data points shown. And then we re-evaluate their performance using the 10 dev set points. And just like we did earlier, first we are going to plot our training mean squared errors as a function of model degree. As expected, the training mean squared error decreases monotonically.

Now just like before, we can also plot our different models on top of the training data. Here, I'm again only showing the degree zero, one, two, and

six models. And as before, the degree six model seems overfit. Now close observers of the video will also notice that this model, trained on a 25 data point subset, is slightly different than the model we got earlier, when we trained on the original 35-point dataset. Now since our degrees six model has very high variance, that means it is very sensitive to data it receives. Now that's quite unlike our degree zero, one, and two models, which have lower variance. They don't care nearly so much about the difference in datasets. And thus, they look a lot more like their cousins from the previous video.

So which of our seven models is best? The degree zero, one, two, three, four, five or six model. That's where the validation set comes in. Recall that a validation set is just the same thing as a dev set, though I'm using both of these terms in this video to get you used to hearing both. We simply compute the mean squared error on our ten validation set points shown in orange. The mean squared error values for each are shown on the table that you see here. We see a similar trend as before, where the mean squared error initially decreases as the model grows more complex. But starting with the degree three model, the mean squared error starts increasing with a huge jump in error once we reach degree five.

We can see the source of this huge error in this plot right here. Consider the rightmost orange data point. Our model suggests that a car with this engine power should have a fuel efficiency of somewhere between 40 and 50 miles per gallon. But the real answer was around 12. This egregious error drives up our validation set error for this degree six model.

Here we see a plot of the training error and the validation error against the polynomial degree. We see that the training error continually decreases with

the polynomial degree. However, the validation set error dips at first, but then it goes back up as the models start to overfit. As we saw in the table before, cross-validation indicates that our best choice is a second-order model.

Here we see an idealized version of that curve. As the model complexity grows, the training error decreases. But by contrast, the dependence of the validation error on the complexity is more of a bowl shape. Initially decreasing as the model gains more expressive power, reaching some bottom, and then increasing back up as the model begins to abuse its surfeit of expressive power and it starts overfitting. In other words, this increase in validation error is caused by the increased model variance.

So what does that tell us? It tells us that when faced with a choice between different models, we should pick the model that minimizes the validation error. In other words, that sits at the bottom of the red curve. It is a common novice mistake to pick based only on the training error, which will lead you to pick models which are over-fit.

Now I should note that these curves are idealized. You will see that on a real computer, on real data, they don't always behave exactly like this. Sometimes kind of funny stuff happens as models get really complex. For example, as the polynomial degree goes very high. And that's just because of the fact that the computer starts encountering lots of rounding errors. I encourage you to play around with this and maybe try out some really high degree models just to see what happens.

Now before we end this video, I want to introduce one last very important piece of vocabulary that you should know. The word hyperparameter. In a machine learning model, a hyperparameter is a value that controls the

learning process itself. So for example, in this video, we built seven models, each of which had a hyperparameter called degree that controlled how many polynomial features were generated. This term leads to a nice way of thinking about the machine learning model training process. We use the training set to select parameters, and we use the development set to select hyperparameters. Or more generally, we use the validation set to decide between different models.

Video 10: Test Sets

Let's suppose you want to build a model to predict the sale price of a diamond from various attributes, like the diamond size, its shape, its opacity, whatever else. Let's assume you wanted to create a linear regression model. Suppose you split the data into training and development sets, and then you then train models of various complexity on the training set. And then finally, you use the development set to pick the best hyperparameters. In other words, those which yield the lowest development set error. This yields the mean squared error values shown.

Now suppose we train a deep learning model, which is something we'll cover later. And that gives us the training and development mean squared error shown. Which model is better? In other words, if you need to deploy this model in a real-world business context to predict prices for diamonds you've never seen before, which one would you trust more?

The better choice here is probably the deep learning model. And that's not just because deep learning is hyped up. I mean that empirically. Because when we look at the development set error, the deep learning model did better. Yes, the linear regression model here did have better training mean squared error, but that's irrelevant. Maybe it's just overfit. So if we want to

use this for real business purposes, we're going to be using new real-world data. And that means that the development error is the better measure to compare.

It's worth noting that this process of comparing two models is subject to random error. In other words, running this experiment again with a different assignment of samples to our training and development set, might show the opposite trend. And we'll talk a bit more about this in a future module.

Now that we've decided that our deep learning model is our best model for pricing diamonds, we want to tell our colleagues about the performance of our model. Ideally, we would provide a mean squared error value. that is an unbiased estimate of my model's performance. By unbiased, we loosely mean that it's an honest assessment. That for a randomly chosen diamond out there in the actual world, this is about the squared error you would expect on average from my model.

Now the validation mean squared error that we've provided would be a reasonable measure, but it is ever so slightly biased in our favor. The idea here is that we tried many different linear regression models, and deep learning models, and then reported the very best one that we found.

Now as a very rough analogy, imagine trying to identify the golfer with the longest drive. In other words, who can hit a ball the farthest. You get the best ten players you know, and you have them hit tons, and tons, and tons of balls. And eventually one of them emerges as the winner who is able to hit the farthest.

Now suppose I wanted to provide an unbiased estimate of just how far our champion can hit. Maybe so we can put it up on our web page or

something. Now using his average drive distance from the distance competition, is a reasonable thing to do. But we have to recognize that there was some luck in the competition process. A better, unbiased estimate would be to have him come back and hit another few 100 drives as a final assessment of his skill. After all, maybe he didn't really quite earn the distance that he achieved in the competition. Maybe he got just a little lucky because the wind was slightly favorable. This last trial of repeated attempts provides a less biased estimate of that golfer's ability to hit a ball.

Now in machine learning, that final estimate of skill is ideally done on a third dataset, often called the test set. In many real-world competitions, this test set is publicly available, but the correct predictions are hidden from the teams participating in the contest. So for example, if we held a contest to predict fuel efficiency from engine power, we would provide a large dataset with two columns, one for the engine power, and one for the fuel efficiency, similar to what we saw before. Individual competitors would then split the dataset into a training and development set. And then they would train the models, and then select the best one using the dev set.

We'd also provide a smaller test set, which would only have the engine power column, hiding the fuel efficiency column so nobody can cheat. We'd have competitors then submit their best predictions for this hidden fuel efficiency. And then we would announce the winner based on the results. In some situations, there is no common test set.

For example, maybe we're trying to create a model that predicts whether or not a given person survived the sinking of the Titanic. Because the full dataset surrounding the passengers of the Titanic has been available for a long time, and there will be no additional sinkings of the Titanic, we need to

create our test set if we want to provide an unbiased estimate of model quality. The idea here is simple. At the beginning, rather than splitting into two sets, we split our data into three sets, a training set, a development set, and a test set. And we make sure not to use the test set in any way at all until the very, very end of our project.

Now we could do that very easily in code. Just like before, we shuffle, then we use scikit-learn to split the data. And so, here, suppose our original diamond set had 2,000 observations. This command will split it into sets of 1,500 for training, 300 for cross-validation, and 200 for testing. The 1,500 points are used to pick parameters for the model. The 300 points are used to pick between different models, including any hyperparameters. And then, at the very end, after we've picked our best model, we compute the mean squared on our test set exactly one time, and we report that value.

In this example, when we compute our test set error, we get 3.3 million. Meaning for this dataset, the difference in dollars squared between our predictions and the true diamond prices was around 3.3 million. Often, this test set error will be larger than our training set error, and about the same as the validation set error. And this 3.3 million is a number that we can spread widely and people will understand immediately what it means. For an arbitrarily selected diamond, from the same distribution as the test set, the average squared error and diamond price will be around $3.3 million squared.

Now in common practice, you may find that you never need the test set at all. If the only thing you need to do is pick the best model, And you don't care about providing an unbiased estimate of the model quality, a test set is unnecessary. So for example, for my golfing example earlier, a TV network

running this contest would probably just use the best competition results. Their goal is hype, not accuracy. So having slightly better numbers for the top golfer makes for better business.

Adding test error to our cartoon picture from earlier, we see that the test error follows a very similar curve to the validation error. For sufficiently large validation and test sets drawn from the exact same distribution, we would expect these curves to overlap perfectly. However, different samples from the space of all possible data will tend to have these curves not quite in alignment.

One important distinction is that whereas we often compute the entire red curve and pick the bottom, we should never, never, never do this for the test set. In other words, we should not compute the entire green curve. By contrast, we should pick a model and only use the test set once. The error that we get back is the intersection of the orange dotted line and this secret test error curve.

Now you can of course, if you'd like, compute the entire test set curve, but selecting a model based on that curve means you've just used it as a development set. So now, if you wanted an unbiased estimate of performance, you're going to have to go collect yet another dataset. And that just doesn't make any sense to do.

Video 11: Conclusion

Let's finish up by reflecting on what we've learned in this module. The primary lesson was that complex features can greatly increase the expressive power of a linear model. For example, we saw we could generate polynomial features using the PolynomialFeatures object provided

by scikit-learn. This allows a linear model to fit parabolic, cubic, or even arbitrarily complex relationships.

We also saw some programming idioms for machine learning in scikit-learn. The first important idiom was the idea of a transformer, such as PolynomialFeatures. The second idiom was the scikit-learn pipeline, which we can use to encapsulate an entire sequence of machine learning operations into a single coherent object.

We discussed how increasing model complexity affects the model's performance. Specifically, we saw that the training error will decrease as complexity increases. And that with enough model complexity, training error will actually decrease to zero. Now at the same time, as complexity increases, a model's variance also increases. In other words, tiny changes to our input dataset can result in dramatic changes to the predictions that our model will make. And for very high levels of complexity, then our model would begin to overfit our data, which will yield poor predictions when the model is applied to subsequent real-world observations.

Now to avoid that overfitting, we traditionally use a process known as cross-validation. And today, we talked about the simplest kind of cross validation, which is just named simple cross-validation. With this technique, we separate our data into two sets, a training set, which we use to pick parameters, and a validation set that we use to pick hyperparameters, or which we used to choose between competing models.

Then lastly, we said that if we want a final unbiased assessment of our model's quality, we can compute the average loss over a test set that is not used at any other point in the process. That is, the only thing that you use your test set for is that unbiased assessment of model quality. Now if you

don't need the unbiased numerical assessment, that's fine. Then you just don't need a test set.

All right, well, that's it for this module, and I will see you next time.