Module 12: Classification and K-Nearest Neighbors

Video Transcripts

Video 1: Introduction to Classification

In this module, we're going to talk about a totally new machine learning problem called classification. To set the stage, let's discuss email in the first decade of this century. Around that time, spam email became a serious problem. Back then, I went to great lengths to hide my true email address using temporary addresses, for example when I'm signing up for websites.

Now here, we see some actual spam emails that I received in the middle of that decade at one of those temporary addresses. You'll notice the text in some of these emails is quite strange. And that's because the senders of these emails were trying to deceive machine learning algorithms that were attempting to detect and block them.

Now these days, I don't bother hiding my email address and yet I still receive virtually no spam. So, why the difference? Well, it's because today's machine learning algorithms have learned really effective patterns that distinguish unwanted spam email from legitimate email. Even with scammers trying to beat the system, the algorithms are just too good.

So, let's formally introduce what we mean by classification. So, let's review our definition of regression, which is that in a regression problem the goal is: Given a set of features, predict a real-valued outcome. By contrast,

determining whether email is spam or not is a different problem called classification. In classification, the goal is: Given a set of features, predict the class to which the sample belongs. For example, given the contents of an email, is it spam or not spam? Or, given the pixels of an image of an animal, which animal is it?

So, in this video, we're going to concern ourselves exclusively with tabular data rather than other types of data, like images or sound files. And so, let's consider this table shown, which gives the income, the total debt, and the status of 225 customers who have taken out a loan with a company. Here, the status is whether or not in the last year, the customer made expected payments on their debts to that company.

'Paid' means all payments were made and 'Did not pay' means that the customer defaulted on the debt, meaning that they failed to pay. Note that our data only has two classes, 'Paid' and 'Did not pay.' And in this module, we're only going to discuss data with two classes. We'll discuss examples of classification with multiple classes like 'dog,' 'cat,' 'horse,' or 'fish' in a future module.

Now, I should note this data is not real-world data. Now naturally, creditors—like banks and credit card companies—would like to predict whether or not a customer will pay back a loan. Or more specifically, the probability that someone will pay or not. Such predictions can be used to derive interest rates for the customer. And in some cases, these interest rates can be tailored to individual customers based on the predicted risk of default, as is common in situations like home loans.

Now, for our current purposes, we'll only try to predict whether or not we believe the customer will pay or not. In other words, we're not taking into account any probabilities for now. So, here, the table represents a past record of many former customers. And we're going to try to build the model using this data to predict the behavior of future customers.

In this plot shown, we see a graphical depiction of our tabular data from before. The customer's income is on the x-axis and the customer's total debt is on the y-axis. The color indicates the status of the customer, whether or not they paid. Now by eye, as humans, we can immediately identify three groups of customers in this historical data. And so naturally, we might imagine that this will help us make predictions about the behavior of future customers.

So, question. Suppose we have a new customer arrive who has an income of $13,000 and a total debt of $400. Without considering any particular algorithm, what do you predict will be the class of this new data point? Naturally, our sense is: Yes, this customer seems firmly ensconced in this nice 'Paid' region. But how do we do that with an algorithm?

Well, there are many approaches. To name a few, nearest neighbors, logistic regression, decision trees, Naive Bayes, random forests, support vector machines, perceptrons, neural networks and so forth. And so, we'll talk about some of these approaches to classification in the coming modules.

Video 2: Nearest Neighbors in Scikit-Learn

Our very first classification algorithm in our course will be the simplest to understand. The 'nearest neighbors' classifier simply says: Which data point in the training set was closest to the data point about which I'm trying to make a prediction? So, for example, for the data point from the exercise at the end of the previous video, our nearest neighbor classifier will simply use the label for the nearest neighbor, highlighted in the figure shown.

Now, if we want to do this in scikit-learn, it's quite similar. Just like our regression models, we start by instantiating an untrained model, as shown in this code. Now, for reasons we'll discuss shortly, we're going to set a hyperparameter called n_neighbors = 1. And we'll do that when we're instantiating the model. Then we train our model, just like usual, using the fit command. And, even though we're doing this totally new task, classification rather than regression, fit will behave more or less exactly as before.

Now, just like a regression estimator, our classification estimator has a predict function. But this time it returns a label rather than a real value. So, as shown here, when we call predict on a model and we give it the input 'Income' equals 13,000 and 'Debt' equals 400, we get back a string label, 'Paid'. In other words, our scikit-learn model thinks that such a customer would pay their debt. So, what did the sklearn model learn exactly when we called fit? Well, it just memorized where all the existing data points are. And, when someone asks for a prediction about a new piece of data, it simply finds the closest, existing data point to that input and then returns the appropriate prediction.

Now we can also visualize the entire space of all hypothetical predictions. So, an example is given in this plot shown. For example, we see that income equals 13,000, debt equals 400. That data point in this space is in the 'Paid' class. In other words, it is the same color as the marker that we used for the customers in the 'Paid' class in our scatterplot. Now, the edges of these two colored regions comprise a so-called decision boundary. In other words, it's this infinitesimal transition region between the two colors, or the two predictions.

We see that this boundary is not a continuous surface. For example, we see these islands of each class. Now that brings us to an issue with our model, it's too sensitive. Just a single paid point cast this huge shadow across the decision space, as you see here. More generally, I might just say the decision boundary looks really complicated. And that suggests that we're probably overfitting, because our model variance appears to be quite high.

Video 3: K-Nearest Neighbors

As we saw in the previous video, our model appears to be too sensitive. So, we might think: What's a natural improvement we could make to our nearest neighbor classifier? You might consider actually pausing the video to come up with your own idea. OK, so my idea is, let's consider more than one neighbor. For example, let's find the three nearest neighbors and then do a majority vote for our prediction.

Let's try this out in scikit-learn and see what sort of decision boundaries we'd get in that case. So, to use the three nearest neighbors, we simply set the n_neighbors hyperparameter to 3 when instantiating the classifier.

When we do this, we end up with the decision boundary shown. Here we see, side-by-side, our nearest neighbor classifier and our new 3 nearest neighbors classifier. And we see that the large shadow, cast by the outlier 'Paid' point in the bottom left, is considerably reduced in size. Likewise, several 'Did not pay' islands, they disappear. They used to be inside the 'Paid' region of the decision space.

Now, we can increase the number of neighbors. For example, here we see the 1, 3, 10, and 15 nearest neighbors models. And we notice that, as the number of neighbors increases, the regions, they get smoother and smoother. Now, we can go even higher to, say, 200 neighbors. And in that case, almost the entire space is classified as 'Paid.' And then once we get up to n = 225, then the decision space will become all 'Paid,' since that is the most common class among the entire 225-point dataset.

So, we see there's a tradeoff here in selecting our k. For a very small k, our model is too complex and the variance is too high. But for very large k, our model is too simple. So, in other words, k is a complexity control hyperparameter, where complexity is inversely related to k. The maximum complexity is k = 1, and the minimum complexity is k = n. Now, it may seem counter intuitive that taking into account more neighbors is less complexity. But consider, for example, that for k = n, the model prediction is very simple. We just return the most common class in the training set. By contrast, for k = 1, we have to find exactly which training point in the entire dataset is closest and then return the label of that training point. Naturally, the k = 1 case will yield much higher variance.

Video 4: Choosing K

K is a hyperparameter, just like alpha in regularized linear regression. Thus, we expect to see a plot similar to our usual cartoon of error versus complexity. Now, since complexity increases with k, I'm going to use −k as our x-axis. Now, while our x-axis is quite natural, as I said, we can just use −k, the y-axis will be trickier. We will have to decide what we mean by 'error.' We can't use mean squared error anymore, because our predictions and observations are no longer numerical quantities. And since they're not numerical, we cannot compute a squared difference between the prediction and the observation.

So, one natural choice would be, let's use the misclassification rate. In other words, the fraction of predictions which are incorrect. So, as an example, consider the five-sample table shown. Status here is the same column as earlier and is what we're trying to predict. y hat is our classifier's predictions. And so, we see on this small dataset, four out of the five predictions were correct. And so, we can say, the misclassification rate was 20% or, equivalently, the accuracy rate is 80%. And yes, in case you're wondering, the misclassification rate is always one minus accuracy.

We can compute the accuracy of a set of predictions by using the sklearn.metrics.accuracy_score function. For example, if we call this function on the example in the five-row table from what we just saw, we get 0.8. The get_misclassification_rate_for_k function that I've created here, if you look at it, what it does is it creates a nearest neighbors classifier, which considers the k-nearest neighbors. It then fits the classifier and then it returns one minus the accuracy rate on the training set. Or equivalently, it

gives us the misclassification rate on the training set. And so, using that function, the code here, it sets up an array of 50 different k's from one to 50. We then use that get_misclassification_rate_for_k function to get the misclassification rate for each of the k's. And that yields the table shown.

We see that for k = 1, the misclassification rate on the training set is zero and that's inevitable, right? The closest data point to itself is itself. And thus, the prediction for any training point will always be correct since our classifier will always use the training point's own class as its prediction. Now, we see that as k goes up, the misclassification rate on the training set also goes up. And this matches what we saw earlier when we visualized the decision boundaries for increasing k. As k gets larger, the boundaries get smoother and smoother, leading to additional classification errors on the training set.

So, here we see a visual plot of the misclassification rate versus k. We see that the misclassification rate climbs overall before leveling off just a bit below 45%. And that's because in this dataset, there were 125 customers who paid. In other words, 55.5% paid and 100 who did not pay. And that works out to about 44% who did not pay. Now, since the most common class is 'Paid,' once we reach k = 225, the model will simply predict 'Paid' for every data point. And thus, when applied to all data points, it gets 55% of them right and 44% of them wrong.

Now, the other extreme, when k = 0, the misclassification rate is zero, for reasons we just discussed earlier. Now, we note that this plot has complexity that decreases as we move to the right, because that is also the direction where k is increasing. So, if we want to better match our old

cartoon plot, we should simply plot −k on the x-axis. So, once we do that, once we plot −k on the x-axis, that yields the flipped plot shown.

Here, we see that as model complexity increases, the misclassification rate decreases. We observe that this plot now matches the intuition that we're used to for our cartoon plot of error versus complexity. For low complexity models, the error is high, dropping as the model complexity increases. Now, unlike linear regression, the improvement here, you'll notice, is not monotonic. In other words, it's possible that a more complex model actually yields a, an increase in error. But, if you look closely here, you'll see that those little upticks in error are generally quite small in magnitude. Note that this plot does not tell us which k to pick. We really don't want to simply pick the lowest error k, because then we run the risk of overfitting.

So, how do we know the best k? Well, we can simply apply cross-validation, as usual. In scikit-learn, that means using our old friend GridSearchCV to identify the k with the lowest validation error. So, this code shown, it applies five-fold cross-validation to find the best n_neighbors. Here, notice that I say this dictionary has all the k's I want to try out between one and 225. And I picked 225 as my high end because that's the number of samples in the dataset. Note that we are scoring here based on accuracy, not misclassification rate.

Recall that what GridSearchCV does, is it tries to maximize the score. So, if we want to minimize the error, or in other words, the misclassification rate, we therefore want to maximize the accuracy, which is one minus the misclassification rate. Now, after fitting our model, we ask our model_finder object for its best estimator. And it tells us, it has trained a model with

n_neighbors = 3. To dissect how GridSearchCV made this choice, we can request the mean test score, which will yield 225 scores, one for each value of k that we tried.

Note that for large k's, the accuracy is 55.5% as we'd expect. For very large k, the test fails, returning NaN, not a number. This is because our temporary training sets in our five-fold cross-validation procedure only have four-fifths times 220…225 or 180 points. For such a training set that is 180 points, it doesn't make sense to have a k greater than 180. After all, if you only have 180 neighbors to look at, it doesn't make any sense to consider your 181 closest neighbors. In other words, for a training set of size 180, the maximum k is 180. So, any k greater than 180 fails, and I mentioned it briefly before, but if you have never seen this before, NaN means not a number.

So, here we see the plot of the cross-validation accuracy versus k. In other words, it's just a plot of the numbers that we just saw. Now, if you look very carefully, you might be able to see the cross-validation accuracy is maximized for k = 3. That tells us that k equal to three is our best choice, at least according to our five-fold cross-validation procedure. So, in order to bring this particular figure in line with our usual cartoon plot of validation error versus complexity, we again need to do some contortions.

Specifically, we'll need to plot −k on the x-axis instead of k, so that the complexity is increasing as we move to the right, just like before. And we'll also need to plot the misclassification rate. In other words, one minus accuracy on the y-axis. So, in effect, we've now flipped our plot from before, from left to right, and we flipped it up to down. And similar to before, if you

look very, very closely, you might see in the bottom left that the error is minimized for k = 3. Plotting the training misclassification rate and the cross-validation misclassification rate on the same axes, we get the figure shown. If we look at the training set, we see the smaller the k, the better with dramatic differences as k shrinks. By contrast, looking at the cross-validation error, we see that the impacts of decreasing k are relatively modest and they reach their optimal value at k = 3 before going back up slightly when we get to k = 2 and k = 1.

So, let's zoom in and look only at the smallest 40 k. In other words, k equals one to 40. That gives us this figure. If we compare this figure with our usual cartoon of training and validation error versus model complexity, we see the usual trend where training error generally decreases as the model complexity increases. But validation error has a more complex story where it initially drops before increasing again. Now, if we look at the rightmost part of this curve, then we can get a sense of what happens as the model starts to overfit. We observe that the magnitude of the overfitting error is quite small, even at the maximum possible model complexity of k = 1.

Video 5: Predict_proba and Decision Thresholds

As we've seen, classifiers provide a prediction about the most likely class. So, here what we're going to do, is look at a small example where I'm only going to set aside 10 random samples, as shown, that have the income and debt levels that we see in the table. Next to those values, separately, I'm showing the results of a 10-nearest neighbors classifier. Now, here I've chosen ten for clarity. And that's despite the fact that we just did five-fold cross-validation and found n_neighbors = 3 was the optimal value. Instead, I'm picking ten and I'm doing so for pedagogical clarity for reasons that will

become really clear in a moment. Scikit-learn models, they provide a function called predict_proba, which returns not just predictions about the class of a given sample, but also information about the level of confidence of the model. The first column here then, we look at the result of this function, is an estimate of the probability that the sample belongs to class 0.

The second column shown, is the model's estimate of the probability of the sample belongs to class 1. And for sklearn models, the 0th class is the class that comes earlier in alphabetical order. And so, since 'Did not pay' comes before 'Paid' in alphabetical order, then 'Did not pay' is the 0th class. So, for example, for this top sample, the model is 100% certain that this person will pay back the loan. And, as another example, for the third sample, the model is only 70% certain that the person will pay back the loan. And as a last example, for the seventh sample, the model is 50% certain that the person will pay back the loan. Note that the results of the predict function are 'Did not pay' for the specific sample.

So, the probabilities for K-NearestNeighbors, they come from the votes from each of the nearest neighbors. So, if two of the ten nearest neighbors defaulted on their loans and eight of them paid, we'd be 80% sure that the person we're considering will pay back their loan. In effect, that behavior of each available training sample is used as a vote. And so, since we have ten nearest neighbors, note that our probabilities, they're always multiples of 0.1. For a binary classifier, scikit-learn will return class 0 if p is less than or equal to 0.5. And class 1 only if p is greater than 0.5. We can think of 0.5 as a threshold that we must exceed to belong to class 1. So, this cutoff point of 0.5 is an arbitrary choice by scikit-learn.

So, note that for that seventh sample we discussed earlier, even though the model is 50% certain that the person will pay back their loan, scikit-learn's predict method returns 'Did not pay' on the sample. And that's because, for scikit-learn, the threshold of 0.5 must be exceeded. Suppose we decided to classify someone as likely to pay only if the probability is greater than 80%. Another way to put that is that we've established a probability threshold of 0.8.

In this example, since the third and fifth row have probability 0.7, which does not exceed 0.8, they will be placed in the 'Did not pay' class. And again, note you must exceed the threshold to be considered in the 'Paid' class. Ties go to 'Did not pay.' So, if we decided to classify someone as likely to pay only if the probability is greater than 70%, in other words we set our threshold to 0.7, then we would still predict that the folks in the third and the fifth rows would be 'Did not pay' customers even though the model prediction accuracy was 70%. So again, ties, they go to class 0.

Video 6: Classifier Metrics

To set the stage for our next topic of discussion, let's discuss a medical context. We can think of a coronavirus test for patients at a medical clinic as a classification model. It takes in some set of features about the patient, and it produces a prediction that may or may not be correct. Before, we used accuracy to assess the quality of a classifier. But accuracy is not always the right metric.

For example, in the year 2020, when the coronavirus first started spreading, almost no people in the world had the disease. However, a fake test that simply said 'not infected' would have extremely high accuracy. After all, if

only 100,000 people have the disease out of 1.4 billion, the accuracy of this bad test would be 1.399 billion out of 1.4 billion, or around 99.993%. This is an example of a situation with imbalanced classes. In other words, there are different numbers of samples from each class. In cases with imbalanced classes, accuracy does not necessarily give a good assessment of the usefulness of a model.

Consider the somewhat mysterious two-by-two grid of numbers shown, 349, 7, 118, and 126. This is an example of a confusion matrix. So, in the context of coronavirus infection, the truth axis, up versus down, is whether or not someone is actually infected. And the prediction axis, left versus right, is whether or not our test predicts that they are infected. The confusion matrix gives us a more thorough picture of classifier usefulness.

For a binary classifier, in other words when there's only two classes, the confusion matrix gives us the count of a few things. The first is, the number of true negatives. These are situations where the model correctly predicts the patient is healthy. There are 349 of those. Next are false positives, here called FP. There were seven of those. The model predicted the patient was sick, but they were not. Next, we have the false negatives, or FN. There's 118 of those. And the model predicted that the patient is healthy, but actually they were sick. And then, lastly, we have the true positives indicated with TP here. There are 126 of those. And those are places where the model correctly predicted that a patient is sick.

Now, unlike accuracy—which is just a single, scalar value—a confusion matrix is a more thorough picture. It carries more information and it will retain its usefulness even for wildly imbalanced classes. So, to test your

understanding of this brand-new idea I just dropped on you, called a confusion matrix, consider the following five questions. I'd like you to pause the video and actually really try to work these out and it'll take a few minutes.

The first is: What was the total accuracy of our model? How many patients were considered sick by the model? How many infected patients were there in reality? Of patients without coronavirus, what fraction did the model predict correctly? And then lastly, a little math question. Give a formula for accuracy, in terms of these four parts of our matrix, TN, FP, FN, and TP.

OK, so the answers to those five questions, and I hope you paused and thought about it. First, if we want to compute the total accuracy of this model, we have (126 + 349)/(118 + 7 + 126 + 349). How many patients were considered sick by the model? 126 + 7, or 133. How many infected patients where they are in reality? Well, that's 118 + 126 = 244. That fourth question. Patients without coronavirus, what fraction do we predict correctly? That's 349 out of 356, or 98%. And then lastly, that formula, well, it's just what we did in the first problem. TP plus TN, all of that, divided by the sum of every box in our confusion matrix.

Now, confusion matrices are great and they're thorough, but they're nontrivial to interpret and read. There's many ways that we can take a confusion matrix and just turn it into a number and use that to maybe rank models. One of those is just the accuracy which we've seen before, but there are others that are interesting.

So, the first I'll introduce is something called precision. And that's TP over TP plus FP, or in this case, 126/126 + 7, or 94.7%. Another I'll introduce is the recall. And that's TP over TP plus FN. That one for this matrix, is 126/126 + 118, or 51.6%. A third, totally arbitrary, thing I'm bringing up is something called the specificity. This is TN over TN plus FP, which is 349/349 + 7, or 98%. And, by the way, the recall is often also called the sensitivity.

OK, so each of these three concepts that I just brought up out of nowhere, they're just different summaries of the confusion matrix. To get some intuition for where they're useful, let's review each of them in turn. So, the precision tells us how many of the people we tested positive actually have coronavirus. It's for this matrix 94.7%. Now, note that the highlighted rectangle, here, that defines this quantity is vertical in orientation. The recall tells us, well, of those with coronavirus, they actually have it, how many do we correctly predict? And that's 51.6%. And here, note that the highlighted rectangle, here, that defines this quantity is horizontal in orientation. Lastly, the specificity tells us, of those who do not have coronavirus, how many do we correctly predict? And here it's quite high, 98%. Note that the highlighted rectangle that defines this quantity is also horizontal in orientation.

OK, so to test your understanding of these arbitrary definitions, I'd like you to compute the precision, the recall, and the specificity of these two classifiers below. Now, there are some tricky situations here. So, do your best to resolve them. I recommend you pause the video and I'll give you my answers.

Now, for the first classifier we have the accuracy, which I computed as 80%. The precision is 50/70, or 71%. The recall is 100%. And the specificity is 30/50, or 60%. Now, for that second classifier, we have, OK, the accuracy is 95%. But then when we get to precision, we have a kind of funny situation. Precision is supposed to be, of the predicted ones, how many were actually ones? But there were no predicted ones. So, what do you do? Well, it turns out the answer is, it's just undefined. Recall is straightforward for this one, it's just 0% and specificity is a 100%.

OK. So, I should say, it takes a lot of practice with these metrics for them to become intuitively useful. But if you got this numerical exercise correct, you've at least understood the mathematical definition of each. Note that just because two models have similar accuracies, their other metrics are not necessarily similar. For example, consider the two possible coronavirus tests shown. These two models have accuracy of around 79%, which you can verify by pausing the video and computing that from the confusion matrices. However, their other metrics are quite different. For example, precision is 126/126 + 7, or 95% on the left. But it is 164/164 + 46, or 78% on the right. In other words, for predictions that a patient is sick, the first model with 95% precision, it's much more likely to be correct in its prediction.

By this point, I've hopefully convinced you that accuracy is not everything. And that there are other metrics that have some intuitive desirability. So, given these metrics, which of the two is better? How do we know which metric or metrics that we should be focusing on? Should we sometimes focus entirely on trying to optimize recall, precision, specificity, some combination of these?

Well, focusing entirely on one of these metrics will often yield a model that's not all that useful. For example, let's say we just wanted to maximize recall. One way to do that would be simply classify every living person on this earth as having coronavirus. That is obviously not a useful model, but it gets you a 100% recall. So, instead, we'll focus on the tradeoffs between these metrics. Because, as we just saw, focusing on only a single metric can give us models that aren't particularly useful. And so, we'll explore how to evaluate these tradeoffs in our next video.

Video 7: Choosing a Classifier Metric

One of the most commonly explored tradeoffs amongst these metrics, is between precision and recall. So, let's consider what these two terms mean in the context of our loan model. In that context, the precision is: If we predict that a person will pay their loan, what is the chance that we're correct? And the recall is, well, if some person out there will pay, what is the chance we correctly identify them as someone who will pay?

Now, in a business context, we want both. High precision, it means that we won't make many bad loans. In other words, our customers to whom we give loans are likely to pay us back. And high recall means that we're not missing any potential quality customers. In other words, we don't want to overlook an application from an interested customer who will ultimately pay back the loan and thus be profitable.

So, imagine an optimistic classifier. And optimistic classifier will tend to think people pay their bills and it will have high recall but low precision. And on the extreme end, a very optimistic classifier. It would disqualify only the

most dangerous- looking customers. That would result in high recall because the model would correctly identify essentially every customer who is likely to pay back their loan. But it would have low precision because it would treat many dangerous customers as safe. This yields a high-risk situation where we'd give out tons of loans, but many of the customers would default.

On the other hand, imagine a pessimistic classifier that tends to think people will not pay their loans. It will have a low recall, but high precision. And in the most extreme case, imagine this extremely pessimistic classifier, which only classifies the very safest customers as part of the 'Paid' class. That results in low recall because we'd miss a vast fraction of good customers. But the precision would be great because we'd be very unlikely to make a mistake when we did approve someone as a good customer. That yields a low-risk, low-reward situation, where we only give out loans to a tiny number of possible customers because we're just being too conservative.

So, how do we adjust the precision and the recall of a model? One approach would be to try to adjust hyperparameters like n_neighbors to maximize some sort of weighted combination of precision and recall. And we could do that using cross-validation. We'll explore that a bit on the homework, but I'm not going to discuss that any further in this video because there's nothing truly new to discuss there. Instead, in this video, I'll talk about a very different, and alternate possibility, which is to adjust our decision threshold. So, what's that? Well, earlier we discussed probability thresholds for classifiers. For example, the model shown marks samples as 'Paid' only if the probability of belonging to the paid class is greater than 0.7. So, let's

use T to represent a probability threshold. So, below we see the confusion matrix for a K-NearestNeighbors model for k = 30 on our dataset if we set T = 0.5. As a thought exercise, consider the following question. As we increase T, what do you think tends to happen to precision? And similarly, what do you think tends to happen to our recall? See these highlighted rectangles, here, for a mnemonic reminder of the definitions of precision and recall. If we think about this exercise in the context of loans, as we increase T, our model gets more and more pessimistic about customers' ability to pay.

We see that as T increases, the precision gets better. The model only classifies data as 1 if it is very sure. And at the same time, recall gets worse. The model will classify fewer and fewer data as class 1, in other words, as 'Paid.' So, here, we see the confusion matrices for our loan payback model with n_neighbors = 30, as we increase T from 0.2 to 0.4 to 0.6 to 0.8. And indeed, in line with what we just saw on our exercise, the precision grows from 70 to 86, to 91%, and then it slips a bit down to 90%. By contrast, the recall drops from 98% to 90% to 79% to 68%. And note that the trend was monotonically decreasing for recall, but that precision's dependence on T was not monotonic. And you'll have a chance to explore this issue more thoroughly on the homework.

Now, we can also plot this table of precision and recall values as a graphical plot. Here, I annotate some of the points in this precision-recall space with the T-value that was used to generate that precision and recall. And we see visually that, as we adjust T from high to low, our precision-recall curve arcs from the top left of the precision-recall plot to the bottom right. Now, scikit-learn provides a really useful function called

precision_recall_curve that will generate precision and recall values for various thresholds for a classifier.

We can plot the resulting values on a curve using Plotly, yielding a curve like the one shown. Now again, note that small T-values are in the bottom left. That's high recall, low precision. And then, high T-values are in the top left. Those have low recall and high precision. Given such a curve for a classifier, we can use it to select our threshold T. Now, ideally, this involves leveraging some domain expertise. For example, how do we translate the precision and the recall into a projection about profits?

Now, the tradeoffs may vary widely. For example, imagine you're building a classifier to detect bombs. In that case, recall is going to be a lot more important than precision, because failure to detect an explosive device is catastrophic. But occasionally having some false alarms, that's just going to be somewhat annoying or costly for the entity using your system. Now, ideally, as I've noted, your choice of threshold is based on this kind of domain knowledge. But, sometimes, the choice of T is based more on simple rules of thumb, or eyeballing the plot.

So, as an example, for that precision-recall curve that we just generated for n_neighbors = 30. Suppose that we know we want a high precision for our model, but we still want a classifier that's useful. OK, so in that case, we might use the highlighted point where T = 0.5. For this threshold, precision is still fairly good. It's 90%, not as good as elsewhere on the plot. But we have significantly better recall, 85%, than those other regions of the curve towards the top left. And then, going to even lower thresholds, we have the issue that the precision drops off rapidly.

Now, the other way that we can use a precision-recall curve is to compare two classifiers independent of decision thresholds. So, as an example, let's consider the world's worst model. This model here, it simply picks a random number between zero and one for its predict_proba values. In other words, it ignores the input features and simply makes up an arbitrary probability of the samples belonging to the positive class. Using that same precision-recall curve function, we can generate its precision-recall curve. Now, such a model is often called a 'No-Skill' model for the reason that it has no predictive skill. And, on any dataset, the No-Skill model will roughly approximate a horizontal line with a precision equal to the fraction of items that are in the positive class. For the example of our Paid/Did not pay dataset, our precision-recall curve is, roughly speaking, a horizontal line with precision = 0.55, as shown.

Comparing this K-NearestNeighbors model where n_neighbors is 30 to our No-Skill model, we can see that the nearest neighbors model's precision-recall curve, it's just, it's a lot better, right? The precision-recall tradeoff is superior. Now, that's no surprise since our no-skill model is just randomly guessing. Now, if we want to see just how much better our K-NearestNeighbors model is to this No-Skill model, we can overlay the precision-recall curves on top of each other on the same axis. Now, if we wanted to choose between these two models based on these curves, we could say the K-NearestNeighbors model, its curve is to the top right of the random model. So, it's clearly better. That's better precision-recall tradeoff.

Now, we could also do a numerical summary of the entire precision-recall curve. And we could do that for model comparison. Now, for example, one

way to take an entire precision-recall curve and summarize it with a number, is just to compute the area under that curve. And so, the area under this No-Skill curve, is clearly much smaller than the area under our nearest neighbors classifier. By the way, an optimal classifier would be the one where the area was one. In other words, the precision-recall tradeoff was stuck towards the top right of this plot.

Now, there's another common tradeoff in machine learning, where we trade off between the recall and one minus specificity. We're not going to talk about this alternate tradeoff explicitly in these videos because the ideas are quite similar to the precision-recall tradeoff. But we'll give you a chance to explore this idea on the homework. This alternate tradeoff is also known as a receiver operator characteristic curve, or just ROC curve for short. And, in the context of a ROC curve, be aware that the recall is often called the true positive rate. And 1 − specificity is often called the true negative rate. Now, just like a precision-recall curve, you can take a ROC curve and summarize it by the area that it encloses. And, for a ROC curve, being closer to the top left is better as you'll see on the homework.

Now, it's worth noting that ROC curves are sufficiently popular that GridSearchCV can compare two models based on the area under their ROC curve. For example, this code here, it tries different values of k and it compares them based on the area under their respective ROC curves under the cross-validation technique. We observe that for this scoring technique, the optimal k is 6. And that's different from the k that we computed when we used accuracy as our cross-validation score. On the homework, you will explore how the cross-validation, ROC area under curve, varies as a function of k.

Video 8: Regression Example: Using Nearest Neighbors for Regression

The central idea for classification in this module was k-nearest neighbors. Now, while it's not super common in practice, we can also use the k-nearest neighbors idea for regression. So, suppose we want to predict debt from income on the same dataset as before. Earlier, in a previous module, we saw how to do this with linear regression and ridge regression. But we can also do it with nearest neighbors and the idea is quite natural.

Consider this figure shown. Let's suppose we have a new customer whose income is $15,800. So, what debt level will our model predict? Well, in the nearest neighbors case, we just look at the single nearest neighbor where k = 1. And the nearest neighbor then, looking at this plot of data, is clearly the point that is rightmost in the original dataset. In other words, the person with the highest income. And so, our nearest neighbors model will predict a value of $343, which is the same as the debt for the highest income person in the actual training dataset.

Now, we can do this same idea in scikit-learn by using the sklearn.neighbors.KNeighborsRegressor. Creating, fitting, and using a KNeighborsRegressor is nearly exactly the same as before, except for the name of the class and also the names of the input and the output columns that we select from our DataFrame. We observe that our regressor outputs a numerical value rather than a label. Now, we can plot the output of this model, just like we did back when we were plotting linear regression curves. So, for k = 1, we see that the line moves up and down quite erratically. But, given the density of the data, it's kind of hard to see what's going on exactly.

So, let's zoom in. If we look only at the range between x = 4,000 and x = 5,000, we have the following curve. And the predictions our model makes are almost exactly as you would expect. The output is always equal to the debt value for the closest input. Now, just like before, we can increase k. So, for example, here we see the output of a KNeighborsRegressor with n_neighbors = 50. Each prediction is just the average of the debts of the 50 closest neighbors in the training set. And we end up with a much smoother plot.

And, just as before with classification, behavior will start to get really poor for very large k. For k = 220, the model is too simple. The output is almost completely flat. So, if we wanted to select an optimum k, we would just use cross-validation as before to identify a value of k that maximizes the metric of our choosing. Now, in this case, since this is a regression problem, mean squared error is what we'd most likely choose as our loss.

Now, as noted earlier, KNeighbors regression is not used much in practice, but we'll explore the idea on this week's homework just to build a deeper understanding of machine learning overall.

Video 9: Conclusion

In this module, we saw a totally new machine learning problem called classification. In a classification problem, we try to predict which class a sample belongs to, rather than some numerical attribute of that sample. We focus only on problems that had two classes, one of which we call the positive class, and the other, which we call the negative class. Now, in general, classification can involve more classes. For example, given a

picture of an animal, which animal is it? But today we just talked about two-class classification.

Our approach today to solving this problem, was k-nearest neighbors. We saw that when k = 1, when we want to know how to classify a sample, we simply find the observation in our training set that is closest to the sample, and then we return the class of that observation. And if we take k to be greater than one, then we're just taking a vote among the k-nearest neighbors. We saw that sklearn classifiers have this predict_proba function, which for k-nearest neighbors, returns the result of this vote. In effect, giving the model's level of confidence that the item belongs to each class. Sklearn also gives us a predict method which gives us the most likely label without that probability associated.

Now, unlike regression models, classification models do not use mean squared error for comparing models, because the predictions of the models are not numerical in the case of classification. The most descriptive difference between two different classification models is given by their confusion matrices. Now, we don't necessarily want a big matrix, so there's a number of different ways to compact a confusion matrix into just one value that summarizes the model's quality. And these include, for example, the accuracy, the precision, the recall, and the specificity, and many others that we've not discussed that you can go learn about some other time.

Now, accuracy is the most intuitive metric. It's just how many did we get right or wrong. But it's not always the best metric. For situations with imbalanced classes, we have to be especially careful. The example we gave was detecting coronavirus very early in the 2020 pandemic. We saw there's

tradeoffs between our different metrics. For example, optimistic loan models have high recall but low precision.

We also saw that these metrics are a function of the decision threshold. By generating a curve showing the precision and recall for various values of T, we were able to visually see the tradeoffs between these two metrics for a given model. We also briefly saw a curve showing the tradeoffs between recall and 1 − specificity, which we called a ROC curve. Such curves can be used to select T and they can also be used to compare two different models.

Now, this topic of evaluating classifiers is vast with really important domain-specific details that drive decision making in real-world organizations. So, while we've only scratched the surface of this topic, we'll explore some more on the homework. And when you go onto working on real-world projects, you'll have an opportunity to deepen your knowledge in classifier evaluation considerably.

Lastly, we observed that even though nearest neighbors was an idea that we developed together to solve the classification problem, we can also use this engine for the problem of regression. You'll see that other such engines can also be used for many different domains for machine learning problems. Now, nearest neighbors is not particularly popular for regression or classification in the real world. But it is a useful stepping stone as we move forward towards bigger and better ideas. And with that, we're done. See you next time.

Video 10: Refugee Resettlement and Prediction

It is important to apply data science tools with an eye towards the specific business or policy environment. Ask yourself when you're evaluating an approach, what is the theory of change that I can impact with a model? The answer will often make the data science job a lot harder, but it will also make the results a lot more impactful.

A nice example of this comes from a great academic paper that studies whether refugee resettlement could be done more effectively by using an algorithm to assign refugees to locations. The goal is to try to figure out how to resettle refugees so they can have the best chance of success in their new country. They focused on refugees arriving in the U.S. and Switzerland, two countries where there's good data available. The setting seems to lend itself to the use of a prediction model. We'd need to predict outcomes. For example, we want to determine how likely a refugee is to be employed within two months, or six months, or one year later.

We've learned precisely the modeling tools we need to accomplish that goal. It's not that straightforward, though, because we're not really do anything about the average outcome. The theory of change underlying the approach is to ask whether a refugee could be reassigned to another location and be more successful. Not just how successful will they be on average.

At a simple level, think about Switzerland where many languages are spoken. A refugee from a French-speaking country may have a very different outcome in the French-speaking portion of the country than where

German or Italian are the primary languages. Of course, we don't need a model to figure that out, but a model can tell us much more subtle factors that might have the same effect and hopefully pick up on the language effects that we expect. The key is that we need to find match effects. The paper has an important advantage in doing this.

It turns out that refugee resettlement is basically done randomly, subject to availability and population, but—importantly—not based on expected outcome. Thus, they turn the prediction problem into a question of matching. That is, we want to predict employment outcomes for each location based on rich observable characteristics about each refugee. They use exactly the tools we are learning in this course to build a supervised model predicting outcomes. They then map these predictions into how refugee resettlement rules could be rewritten to adhere to the algorithmic predictions and maximize employment outcomes.

The results are striking. Were refugee agencies to rely on an assignment algorithm, they predict the U.S. could increase employment of refugees by 41%. In the Swiss setting, they're able to assess long-run outcomes and find that employment three years later could be increased by 73% using algorithmic assignment. The efficacy and cost-effectiveness of this kind of approach is a dramatic improvement over far more costly interventions, such as retraining programs.

These results are striking and demonstrate the value of finding a setting where random assignment allows them to use standard prediction tools to estimate matches. These kinds of matching predictions are important in a lot of data science applications. For example, consider a sales organization

trying to assign different salespeople to certain types of leads. If there are some salespeople who are better at different types of leads, this is commonly the case, we want to do more than just predict lead quality on average. We want to allocate each salesperson to the best lead for them.

Here we want to look for randomization or near randomization and could use a similar approach with our prediction tools. And good news for us, it's common for companies to use relatively random rules in a lot of settings. For example, often leads are given at random to all salespeople. The example of refugee placement is interesting and important in its own right. It also underscores the many settings in which predicting matches matter and demonstrate why data scientists need to be thinking carefully about the downstream business or policy objectives.