[MUSIC] Hi, in this lesson, we will talk about a major part of any competition. The metrics that are used to evaluate a solution. In this video, we'll discuss why there are so many metrics and why it is necessary to know what metric is used in a competition. In the following videos, we will study what is the difference between a loss and a metric? And we'll overview and show optimization techniques for the most important and common metrics. In the course, we focus on regression and classification. So we only discuss metric for these tasks. For better understanding, we will also build a simple baseline for each metric. That is what the best constant to predict for that particular method. So metrics are an essential part of any competition. They are used to evaluate our submissions. Okay, but why do we have a different evolution metric on each competition? That is because there are plenty of ways to measure equality of an algorithm and each company decides for themselves what is the most appropriate way for their particular problem. For example, let's say an online shop is trying to maximize effectiveness of their website. The thing is you need to formalize what is effectiveness. You need to define a metric how effectiveness is measured. It can be a number of times a website was visited, or the number of times something was ordered using this website. So the company usually decides for itself what quantity is most important for it and then tries to optimize it. In the competitions, the metrics is fixed for us and the models and competitors are ranked using it. In order to get higher leader board score you need to get a better metric score. That's basically the only thing in the competition that we need to care about, how to get a better score. And so it is very important to understand how metric works and how to optimize it efficiently. I want to stress out that it is really important to optimize exactly the metric we're given in the competition and not any other metric. Consider an example, blue and red lines represent objects of a class zero and one respectively. And say we decided to use a linear classifier,and came up with two matrix to optimize, M1 and M2. The question is, how much different the resulting classifiers would be? Actually by a lot. The two lines here, the solid and the dashed one show the best line your boundaries for the two cases. For the dashed, M1 score is the highest among all possible hyperplanes. But M2 score for the hyperplane is low. And we have an opposite situation for the solid boundary. M2 score is the highest, whereas M1 score is low. Now, if we know that in this particular competition, the ranking is based on M1 score, then we need to o ptimize M1 score and so we should submit the prediction. Predictions of the model with dash boundary. Once again, if your model is scored with some metric, you get best results by optimizing exactly that metric. Now, the biggest problem is that some metrics cannot be optimized efficiently. That is there is no simple enough way to find, say, the optimal hyperplane. That is why sometimes we need to train our model to optimize something different than competition metric. But in this case we will need to apply various heuristics to improve competition metric score. And there's another case where we need to be smart about the metrics. It is one that train and the test sets are different. In the lesson about leaks, we'll discuss leader board probing. That is, we can check, for example, if the mean target value on public part of test set is the same as on train. If it's not, we would need to adapt our predictions to suit rest set better. This is basically a specific metric optimization technique we apply, because train and test are different. Or there can be more severe cases where improved metric validation set could possibly not result into improved metric on the test set. In these situations, it's a good idea to stop and think maybe there is a different way to approach the problem. In particular, time series can be very challenging to forecast. Even if you did a validation just right. [INAUDIBLE] by time, rolling windows, fill the distribution in the future can be much different from what we had in the train set. Or sometimes, there's just not enough training data, so a model cannot capture the patterns. In one of the compositions I took part, I had to use some tricks to boost my score after the modeling. And the trick was as a consequence of a particular metric used in that competition. The metric was quite unusual actually, but it is intuitive. If a trend is guessed correctly, then the absolute difference between the prediction and the target is considered as an error. If for instance, model predict end value in the prediction horizon to be higher than the last value from the train side but in reality it is lower, then the trend is predicted incorrectly, and the error was set to absolute difference squared. So if we predict a value to be above the dashline, but it turns out to be below or vice versa, the trend [INAUDIBLE] to be predicted incorrectly. So this metric carries a lot more about correct trend to be predicted than about actual value you predict. And that is something it was possible to exploit. There were several times series was to forecast, the horizon to predict was wrong, and the model's predictions were unreliable. Moreover, it was not possible to optimize this metric exactly. So I realized that it would be much better to set all the predictions to either last value plus a very tiny constant, or last value minus very tiny constant. The same value for all the points in the time interval, we are to predict for each time series. And design depends on the estimation. What is more likely the values in the horizon to be lower than the last known value, or to be higher? This trick actually took me to the first place in that competition. So finding a nice way to optimize a metric can give you an advantage over other participants, especially if the metric is peculiar. So maybe I should formulate it like that. We should not forget to do kind of exploratory metric analysis along with exploratory data analysis. At least when the metric is an unusual one. So in this video we've understood that each business has its own way to measure ineffectiveness of an algorithm based on its needs, and therefore, there are so many different metrics. And we saw two motivational examples. Why should we care about the metrics? Well, basically because it is how competitors are compared to each other. In the following videos we'll talk about concrete metrics. We'll first discuss high level intuition for each metric and then talk about optimization techniques. [MUSIC]In this video, we will review the most common ranking metrics and establish an intuition about them. Although in a competition, the metric is fixed for us, it is still useful to understand in what cases one metric could be preferred to another. In this course, we concentrate on regression and classification, so we will only discuss related metrics. For a better understanding, for each metric, we will also build the most simple baseline we could imagine, the constant model. That is, if we are only allowed to predict the same value for every object, what value is optimal to predict according to the chosen metric? Let's start with regression task and related metrics. In the following videos, we'll talk about metrics for classification. First, let us clarify the notation we're going to use throughout the lesson. N will be the number of samples in our training data set, y is that the target, and y-hat is our model's predictions. And y-hat and y with index i are the predictions, and target value respectively for i-th object. The first metric we will discuss is Mean Square Error. It is for sure the most common metric for regression type of problems. In data science, people use it when they don't have any specific preferences for the solution to their problem, or when they don't know other metric. MSE basically measures average squared error of our predictions. For each point, we calculate square difference between the predictions of the target and then average those values over the objects. Let's introduce a simple data set now. Say, we have five objects, and each object has some features, X, and the target is shown in the column Y. Let's ask ourselves a question. How will the error change if we fix all the predictions but want to be perfect, and we'll derive the value of the remaining one? To answer this question, take a look at this plot. On the horizontal line, we will first put points to the positions of the target values. The points are colored according to the corresponding rows in our data table. And on the Y-axis, we will show the mean square error. So, let's now assume that our predictions for the first four objects are perfect, and let's draw a curve. How the metric value will change if we change the prediction for the last object? For MSE metric, it looks like that. In fact, if we predict 25, the error is zero, and if we predict something else, then it is greater than zero. And the error curve looks like parabola. Let's now draw analogous curves for other objects. Well, right now it's hard to make any conclusions but we will build the same kind of plot for every metric and we will note the difference between them. Now, let's build the simplest baseline model. We'll not use the features X at all and we will always predict a constant value Alpha. But, what is the optimal constant? What constant minimizes the mean square error for our data set? In fact, it is easier to set the derivative of our total error with respect to that constant to zero, and find it from this equation. What we'll find is that the best constant is the mean value of the target column. If you think you don't know how to derive it, take a look at the reading materials. There is a fine explanation and links to related books. But let us constructively check it. Once again, on the horizontal axis, let's denote our target values with dot and draw a function. How the error changes is if we change the value of that constant Alpha? We can do it with a simple grid search over a given range by changing Alpha intuitively and recomputing an error. Now, the green square shows a minimum value for our metric. The constant we found is 10.99, and it's quite close to the true mean of the target which is 11. In fact, the value we got deviates from the true mean value only because with the grid search, we get only approximate answer. Also note that the red curve on the second plot is uniformly same and average of the curves from the first plot. We are finished discussing MSE metric itself, but there are two more related metrics used frequently, RMSE and R_squared. And we will briefly study them now. RMSE, Root Mean Square Error, is a very similar metric to MSE. In fact, it is calculated in two steps. First, we calculate regular mean square error and then, we take a square root of it. The square root is introduced to make scale of the errors to be the same as the scale of the targets. For MSE, the error is squared, so taking a root out of it makes total error a little bit easier to comprehend because it is linear now. Now, it is very important to understand in what sense RMSE is similar to MSE, and what is the difference. First, they are similar in terms of their minimizers. Every minimizer of MSE is a minimizer of RMSE and vice versa. But generally, if we have two sets of predictions, A and B, and say MSE of A is greater than MSE of B, then we can be sure that RMSE of A is greater RMSE of B. And it also works in the opposite direction. This is actually true only because square root function is non-decreasing. What does it mean for us? It means that, if our target the metric is RMSE, we still can compare our models using MSE, since MSE will order the models in the same way as RMSE. And we can optimize MSE instead of RMSE. In fact, MSE is a little b it easier to work with, so everybody uses MSE instead of RMSE. But there is a little bit of difference between the two for gradient-based models. Take a look at the gradient of RMSE with respect to i-th prediction. It is basically equal to gradient of MSE multiplied by some value. The value doesn't depend on the index I. It means that travelling along MSE gradient is equivalent to traveling along RMSE gradient but with a different flowing rate and the flowing rate depends on MSE score itself. So, it is kind of dynamic. So even though RMSE and MSE are really similar in terms of models scoring, they can be not immediately interchangeable for gradient based methods. We will probably need to adjust some parameters like the learning rate. Now, what if I told you that MSE for my models predictions is 32? Should I improve my model or is it good enough? Or what if my MSE was 0.4? Actually, it's hard to realize if our model is good or not by looking at the absolute values of MSE or RMSE. It really depends on the properties of the dataset and their target vector. How much variation is there in the target vector. We would probably want to measure how much our model is better than the constant baseline. And say, the desired metrics should give us zero if we are no better than the baseline and one if the predictions are perfect. For that purpose, R_squared metric is usually used. Take a look. When MSE of our predictions is zero, the R_squared is 1, and when our MSE is equal to MSE over constant model, then R_squared is zero. Well, because the values in numerator and denominator are the same. And all reasonable models will score between 0 and 1. The most important thing for us is that to optimize R_squared, we can optimize MSE. It will be absolutely equivalent since R_squared is basically MSE score divided by a constant and subtracted from another constant. These constants doesn't matter for optimization. Lets move on and discuss another metric called Mean Absolute Error, or MAE in short. The error is calculated as an average of absolute differences between the target values and the predictions. What is important about this metric is that it penalizes huge errors that not as that badly as MSE does. Thus, it's not that sensitive to outliers as mean square error. It also has a little bit different applications than MSE. MAE is widely used in finance, where $10 error is usually exactly two times worse than $5 error. On the other hand, MSE metric thinks that $10 error is four times worse than $5 error. MAE is easier to justify. And if you used RMSE, it would become really hard to explain to your boss how you evaluated your model. What constant is optimal for MAE? It's quite easy to find that its a median of the target values. In this case, it is eight. See reading materials for a proof. Just to verify that everything is correct, we again can try to Greek search for an optimal value with a simple loop. And in fact, the value we found is 7.98, which indicates we were right. Here, we see that MAE is more robust than MSE, that is, it is not that influenced by the outliers. In fact, recall that the optimal constant for MSE was about 11 while for MAE it is eight. And eight looks like a much better prediction for the points on the left side. If we assume that point with a target 27 is an outlier and we should not care about the prediction for it. Another important thing about MAE is its gradients with respect to the predictions. The grid end is a step function and it takes -1 when Y_hat is smaller than the target and +1 when it is larger. Now,

the gradient is not defined when the prediction is perfect, because when Y_hat is equal to Y, we can not evaluate gradient. It is not defined. So formally, MAE is not differentiable, but in fact, how often your predictions perfectly measure the target. Even if they do, we can write a simple IF condition and return zero when it is the case and through gradient otherwise. Also know that second derivative is zero everywhere and not defined in the point zero. I want to end the discussion with the last note. Well, it has nothing to do with competitions but every data scientists should understand this. We said that MAE is more robust than MSE. That is, it is less sensitive to outliers, but it doesnt mean it is always better to use MAE. No, it does not. It is basically a question. Are there any real outliers in the dataset or there are just, let's say, unexpectedly high values that we should treat just as others? Outliers have usually mistakes, measurement errors, and so on, but at the same time, similarly looking objects can be of natural kind. So, if you think these unusual objects are normal in the sense that they're just rare, you should not use a metric which will ignore them. And it is better to use MSE. Otherwise, if you think that they are really outliers, like mistakes, you should use MAE. So in this video, we have discussed several important metrics. We first discussed, mean square error and realized that the best constant for it is the mean targeted value. Root Mean Square Error, RMSE, and R_squared are very similar to MSE from optimization perspective. We then discussed Mean Absolute Error and when people prefer to use MAE over MSE. In the next video, we will continue to study regression metrics and then we'll get to classification ones.[SOUND] In the previous video,
we started to discuss regression metrics. In this video,
we'll talk about three more metrics, (R)MSPE, MAPE, and (R)MSLE. Think about the following problem. We need to predict,
how many laptops two shops will sell? And in the train set for
a particular date, we see that the first shop sold 10 items, and
the second sold 1,000 items. Now suppose our model predicts
9 items instead of 10 for the first shop, and
999 instead of 1,000 for the second. It could happen that off by
one error in the first case, is much more critical
than in the second case. But MSE and MAE are equal to one for
both shops predictions, and thus according to those metrics, these
off by one errors are indistinguishable. This is basically because MSE and
MAE work with absolute errors while relative error can
be more important for us. Off by one error for
the shops that sell ten items is equal to mistaking by 100 items for
shops that sell 1,000 items. On the plot for MSE and MAE, we can see that all the error curves have
the same shape for every target value. The curves are kind of shifted
version of each other. That is an indicator that metric
works with absolute errors. The relative error preference
can be expressed with Mean Square Percentage Error, MSPE in short, or
Mean Absolute Percentage Error, MAPE. If you compare them to MSE

and MAE, you will notice the difference. For each object, the absolute error is divided by the target value, giving relative error. MSPE and MAPE can also be thought as weighted versions of MSE and MAE, respectively. For the MAPE, the weight of its sample is inversely proportional to it's target. While for MSPE, it is inversely proportional to a target square. Know that the weight do not sum up to one here. You can take a look at this individual error plus for our individual sample dataset. Now, we see the course became more flat as the target value increases. It means that, the cost we pay for a fixed absolute error, depends on the target value. And as the target increases, we pay less. So having talk about definition and motivation behind MSPE and MAPE. Let's now think, what are the optimal constant predictions for these matrix? Recall that for MSE, the optimal constant is the mean over target values. Now, for MSPE, the weighted version of MSE, in turns out that the optimal constant is weighted mean of the target values. For our dataset, the optimal value is about 6.6, and we see that it's biased towards small targets. Since the absolute error for them is weighted with the highest weight, and thus inputs metric the most. Now the MAPE, this is a question for you. What do you think is an optimal constant for it? Just use your intuition here and knowledge from the previous slides. Especially recall that MAPE is weighted version of MAE. The right answer is, the best constant is weighted median. It is not a very commonly used quantity actually, so take a look for a bit of explanation in the reading materials. The optimal value here is 6, and it is even smaller than the constant for MSPE. But do not try to explain it using outliers. If an outlier had a very, very small value, MAPE would be very biased towards it, since this outlier will have the highest weight. All right, now let's move on to the last metric in this video, Root Mean Square Logarithmic Error, or RMSLE in short. What is RMSLE? It is just an RMSE calculated in logarithmic scale. In fact, to calculate it, we take a logarithm of our predictions and the target values, and compute RMSE between them. The targets are usually non-negative but can equal to 0, and the logarithm of 0 is not defined. That is why a constant is usually added to the predictions and the targets before applying the logarithmic operation. This constant can also be chosen to be different to one. It can be for example 300 depending on organizer's needs. But for us, it will not change much. So, this metric is usually used in the same situation as MSPE and MAPE, as it also carries about relative errors more than about absolute ones. But note the asymmetry of the error curves. From the perspective of RMSLE, it is always better to predict more than the same amount less than target. Same as root mean square error doesn't differ much from mean square error, RMSLE can be calculated without root operation. But the rooted version is more widely used. It is important to know that the plot we see here on the slide is built for a version without the root. And for a root version, an analogous plot would be misleading. Now let's move on to the question about the best constant. I will let you guess the answer again. Just recall that, Just recall what is the best constant prediction for RMSE and use the connection between RMSLE and RMSE. To find the constant, we should realize that we can first find the best constant for RMSE in the log space, will be the weighted mean in the log space. And after it, we need to get back from log space to the usual one with an inverse transform. The optimal constant turns out to be 9.1. It is higher than constants for both MAPE and MSPE. Here we see the optimal constants for the metrics we've broken down. MSE is quite biased towards the huge value from our dataset, while MAE is much less biased. MSPE and MAPE are biased towards smaller targets because they assign higher weight to the object with small targets. And RMSLE is frequently considered as better metrics than MAPE, since it is less biased towards small targets, yet works with relative errors. I strongly encourage you to think about the baseline for metrics that you can face for first time. It truly helps to build an intuition and to find a way to optimize the metrics. So, in this video, we will discuss different metrics that works with relative errors. MSPE, means square percentage error, MAPE, mean absolute percentage error, and RMSLE, root mean squared logarithmic error. We'll discussed the definitions and the baseline solutions for them. In the next video, we will study several classification matrix. [MUSIC][MUSIC] In the previous videos, we discussed metrics for regression problems. And here, we'll review classification metrics. We will first talk about ac curacy, logarithmic loss, and then get to area under a receiver operating curve, and Cohen's Kappa. And specifically Quadratic weighted Kappa. Let's start by fixing the notation. N will be the number of objects in our dataset, L, the number of classes. As before, y will stand for the target, and y hat, for predictions. If you see an expression in square brackets, that is an indicator function. It fields one if the expression is true and zero if it's false. Throughout the video, we'll use two more terms hard labels or hard predictions, and soft labels or soft predictions. Usually models output some kind of scores. For example, probabilities for an objects to belong to each class. The scores can be written as a vector of size L, and I will refer to this vector as to soft predictions. Now in classification we are usually asked to predict a label for the object, do a hard prediction. To do it, we usually find a maximum value in the soft predictions, and set class that corresponds to this maximum score as our predicted label. So hard label is a function of soft labels, it's usually arg max for multi class tasks, but for binary classification it can be thought of as a thresholding function. So we output label 1 when the soft score for the class 1 is higher than the threshold, and we output class 0 otherwise. Let's start our journey with the accuracy score. Accuracy is the most straightforward measure of classifiers quality. It's a value between 0 and 1. The higher, the better. And it is equal to the fraction of correctly classified objects. To compute accuracy, we need hard predictions. We need to assign each object a specific table. Now, what is the best constant to predict in case of accuracy? Actually, there are a small number of constants to try. We can only assign a class label to all the objects at once. So what class should we assign? Obviously, the most frequent one. Then the number of correctly guessed objects will be the highest. But exactly because of that reason, there is a caveat in interpreting the values of the accuracy score. Take a look at this example. Say we have 10 cats and 90 dogs in our train set. If we always predicted dog for every object, then the accuracy would be already 0.9. And imagine you tell someone that your classifier is correct 9 times out of 10. The person would probably think you have a nice model. But in fact, your model just predicts dog class no matter what input is. So the problem is, that the base line accuracy can be very high for a data set, even 99%, and that makes it hard to interpret the results. Although accuracy score is very clean and intuitive, it turns out to be quite hard to optimize. Accuracy also doesn't care how confident the classifier is in the predictions, and what soft predictions are. It cares only about arg max of soft predictions. And thus, people sometimes prefer to use different metrics that are first, easier to optimize. And second, these metrics work with soft predictions, not hard ones. One of such metrics is logarithmic loss. It tries to make the classifier to output two posterior probabilities for their objects to be of a certain kind, of a certain class. A log loss is usually the reason a little bit differently for binary and multi class tasks. For binary, it is assumed that y hat is a number from 01 range, and it is a probability of an object to belong to class one. So 1 minus y hat is the probability for this object to be of class 0. For multiclass tasks, LogLoss is written in this form. Here y hat ith is a vector of size L, and its sum is exactly 1. The elements are the probabilities to belong to each of the classes. Try to write this formula down for L equals 2, and you will see it is exactly binary loss from above. And finally, it should be mentioned that to avoid in practice, predictions are clipped to be not from 0 to 1, but from some small positive number to 1 minus some small positive number. Okay, now let us analyze it a little bit. Assume a target for an object is 0, and here on the plot, we see how the error will change if we change our predictions from 0 to 1. For comparison, we'll plot absolute error with another color. Logloss usually penalizes completely wrong answers and prefers to make a lot of small mistakes to one but severer mistake. Now, what is the best constant for logarithmic loss? It turns out that you need to set predictions to the frequencies of each class in the data set. In our case, the frequencies for the cat class is 0.1, and it is 0.9 for class dog. Then the best constant is vector of those two values. How do I, well how do I know that is so? To prove it we should take a derivative with the respect to constant alpha, set it to 0, and find alpha from this equation. Okay, we've discussed accuracy and log loss, now let's move on. Take a look at the example. We show ground truth target value with color, and the position of the point shows the classifier score. Recall that to compute accuracy score for a binary task, we usually take soft predictions from our model and apply threshold. We can see the prediction to be green if the score is higher than 0.5 and red if it's lower. For this example the accuracy is 6 or 7, as we misclassified one red object. But look, if the threshold was 0.7, then all the objects would be classified correctly. So this is kind of motivation for our next metric, Area Under Curve. We shouldn't fix the threshold for it, but this metric kind of tries all possible ones and aggregates those scores. So this metric doesn't really cares about absolute values of the predictions. But it depends only on the order of the objects. Actually, there are several ways AUC, or this area under curve, can be explained. The first one explains under what curve we should compute area. And the second explains AUC as the probability of object pairs to be correctly ordered by our model. We will see both explanations in the moment. So let's start with the first one. So we need to calculate an area under a curve. What curve? Let's construct it right now. Once again, say we have six objects, and their true label is shown with a color. And the position of the dot shows the classifier's predictions. And for now we will use word positive as synonym to belongs to the red class. So positive side is on the left. What we will do now, we'll go from left to right, jump from one object to another. And for each we will calculate how many red and green dots are there to the left, to this object that we stand on. The red dots we'll have a name for them, true positives. And for the green ones we'll have name false positives. So we will kind of compute how many true positives and false positives we see to the left of the object we stand on. Actually it's very simple, we start from bottom left corner and go up every time we see red point. And right when we see a green one. Let's see. So we stand on the leftmost point first. And it is red, or positive. So we increase the number of true positives and move up. Next, we jump on the green point. It is false positive, and so we go right. Then two times up for two red points. And finally two times right for the last green point. We finished in the top right corner. And it always works like that. We start from bottom left and end up in top right corner when we jump on the right most point. By the way, the curve we've just built is called Receiver Operating Curve or ROC Curve. And now we are ready to calculate an area under this curve. The area is seven and we need to normalize it by the total plural area of the square. So AUC is 7/9, cool. Now what AUC will be for the data set that can be separated with a threshold, like in our initial example? Actually AUC will be 1, maximum value of AUC. So it works. It doesn't need a threshold to be specified and it doesn't depend on absolute values. Recall that we've never used absolute values while constructing the curve. Now in practice, if you build such curve for a huge data set in real classifier, you would observe a picture like that. Here curves for different classifiers are shown with different colors. The curves usually lie above the dashed line which shows how would the curve look like if we made predictions at random. So it kind of shows us a baseline. And note that the area under the dashed line is 0.5. All right, we've seen that we can build a curve and compute area under it. There is another total different explanation for the AUC. Consider all pairs of objects, such that one object is from red class and another one is from green. AUC is a probability that score for the green one will be higher than the score for the red one. In other words, AUC is a fraction of correctly ordered pairs. You see in our example we have two incorrectly ordered pairs and nine pairs in total. And then there are 7 correctly ordered pairs and thus AUC is 7/9. Exactly as we got before, while computing area under the curve. All right, we've discussed how to compute AUC. Now let's think what is the best constant prediction for it. In fact, AUC doesn't depend on the exact values of the predictions. So all constants will lead to the same score and this score will be around 0.5, the baseline. This is actually something that people love about AUC. It is clear what the baseline is. Of course there are flaws in AUC, every metric has some. But still AUC is metric I usually use when no one sets up another one for me. All right, finally let's get to the last metric to discuss, Cohen's Kappa and it's derivatives. Recall that if we always predict the label of the most frequent class, we can already get pretty high accuracy score, and that can be misleading. Actually in our example all the models will fit, will have a score somewhere between 0.9 and 1. So we can introduce a new metric such that for an accuracy of 1 it would give us 1, and for the baseline accuracy it would output 0. And of course, baselines are going to be different for every data, not necessarily 0.9 or whatever. It is also very similar to what r squared does with MSE. It informally saying is kind of normalizes it. So we do the same here. And this is actually already almost Cohen's Kappa. In Cohen's Kappa we take another value as the baseline. We take the higher predictions for the data set and shuffle them, like randomly permute. And then we calculate an accuracy for these shuffled predictions. And that will be our baseline. Well to be precise, we permute and calculate accuracies many times and take, as the baseline, an average for those computed accuracies. In practice, of course, we do not need to do any permutations. This baseline score can be computed analytically. We need, first, to multiply the empirical frequencies of our predictions and grant those labels for each class, and then sum them up. For example, if we assign 20 cat labels and 80 dog labels at random, then the baseline accuracy will be 0.2*0.1 + 0.8*0.9 = 0.74. You can find more examples in actually. Here I wanted to explain a nice way of thinking about eliminator as a baseline. We can also recall that error is equal to 1 minus accuracy. We could rewrite the formula as 1 minus model's error/baseline error. It will still be Cohen's Kappa, but now, it would be easier to derive weighted Cohen's Kappa. To explain weighted Kappa, we first need to do a step aside, and introduce weighted error. See now we have cats, dogs and tigers to classify. And we are more or less okay if we predict dog instead of cat. But it's undesirable to predict cat or dog if it's really a tiger. So we're going to form a weight matrix where each cell contains The weight for the mistake we might do. In our case, we set error weight to be ten times larger if we predict cat or dog, but the ground truth label is tiger. So with error weight matrix, we can express our preference on the errors that the classifier would make. Now, to calculate weight and error we need another matrix, confusion matrix, for the classifier's prediction. This matrix shows how our classifier distributes the predictions over the objects. For example, the first column indicates that four cats out of ten were recognized correctly, two were classified as dogs and four as tigers. So to get a weighted error score, we need to multiply these two matrices element-wise and sum their results. This formula needs a proper normalization to make sure the quantity is between 0 and 1, but it doesn't matter for our purposes, as the normalization constant will anyway cancel. And finally, weighted kappa is calculated as 1- weighted error / weighted baseline error. In many cases, the weight matrices are defined in a very simple way. For example, for classification problems with ordered labels. Say you need to assign each object a value from 1 to 3. It can be, for instance, a rating of how severe the disease is. And it is not regression, since you do not allow to output values to be somewhere between the ratings and the ground truth values also look more like labels, not as numeric values to predict. So such problems are usually treated as classification problems, but weight matrix is introduced to account for order of the labels. For example, weights can be linear, if we predict two instead of one, we pay one. If we predict three inst ead of of one, we pay two. Or the weights can be quadratic, if we'll predict two instead of one, we still pay one, but if we predict three instead of one, we now pay for. Depending on what weight matrix is used, we get either linear weighted kappa or quadratic weighted kappa. The quadratic weighted kappa has been used in several competitions on Kaggle. It is usually explained as inter-rater agreement coefficient, how much the predictions of the model agree with ground-truth raters. Which is quite intuitive for medicine applications, how much the model agrees with professional doctors. Finally, in this video, we've discussed classification matrix. The accuracy, it is an essential metric for classification. But a simple model that predicts always the same value can possibly have a very high accuracy that makes it hard to interpret this metric. The score also depends on the threshold we choose to convert soft predictions to hard labels. Logloss is another metric, as opposed to accuracy it depends on soft predictions rather than on hard labels. And it forces the model to predict probabilities of an object to belong to each class. AUC, area under receiver operating curve, doesn't depend on the absolute values predicted by the classifier, but only considers the ordering of the object. It also implicitly tries all the thresholds to converge soft predictions to hard labels, and thus removes the dependence of the score on the threshold. Finally, Cohen's Kappa fixes the baseline for accuracy score to be zero. In spirit it is very similar to how R-squared beta scales MSE value to be easier explained. If instead of accuracy we used weighted accuracy, we would get weighted kappa. Weighted kappa with quadratic weights is called quadratic weighted kappa and commonly used on Kaggle.

[MUSIC]In this video, we will discuss what is the loss and what is a metric, and what is the difference between them. And then we'll overview what are the general approaches to metric optimization. Let's start with a comparison between two notions, loss and metric. The metric or target metric is a function which we want to use to evaluate the quality of our model. For example, for a classification task, we may want to maximize accuracy of our predictions, how frequently the model outputs the correct label. But the problem is that no one really knows how to optimize accuracy efficiently. Instead, people come up with the proxy loss functions. They are such evaluation functions that are easy to optimize for a given model. For example, logarithmic loss is widely used as an optimization loss, while the accuracy score is how the solution is eventually evaluated. So, once again, the loss function is a function that our model optimizes and uses to evalua te the solution, and the target metric is how we want the solution to be evaluated. This is kind of expectation versus reality thing. Sometimes we are lucky and the model can optimize our target metric directly. For example, for mean square error metric, most libraries can optimize it from the outset, from the box. So the loss function is the same as the target metric. And sometimes we want to optimize metrics that are really hard or even impossible to optimize directly. In this case, we usually set the model to optimize a loss that is different to a target metric, but after a model is trained, we use hacks and heuristics to negate the discrepancy and adjust the model to better fit the target metric. We will see the examples for both cases in the following videos. And the last thing to mention is that loss metric, cost objective and other notions are more or less used as synonyms. It is completely okay to say target loss and optimization metric, but we will fix the wording for the clarity now. Okay, so far, we've understood why it's important to optimize a metric given in a competition. And we have discussed the difference between optimization loss and target metric. Now, let's overview the approaches to target metrics optimization in general. The approaches can be broadly divided into several categories, depending on the metric we need to optimize. Some metrics can be optimized directly. That is, we should just find a model that optimizes this metric and run it. In fact, all we need to do is to set the model's loss function to these metric. The most common metrics like MSE, Logloss are implemented as loss functions in almost every library. For some of the metrics that cannot be optimized directly, we can somehow pre-process the train set and use a model with a metric or loss function which is easy to optimize. For example, while MSPE metric cannot be optimized directly with XGBoost, we will see later that we can resample the train set and optimize MSE loss instead, which XGBoost can optimize. Sometimes, we'll optimize incorrect metric, but we'll post-process the predictions to fit classification, to fit the communication metric better. For some models and frameworks, it's possible to define a custom loss function, and sometimes it's possible to implement a loss function which will serve as a nice proxy for the desired metric. For example, it can be done for quadratic-weighted Kappa, as we will see later. It's actually quite easy to define a custom loss function for XGBoost. We only need to implement a single function that takes predictions and the target values and computes first and second-order derivatives of the loss function with respect to the model's predictions. For example, here you see one for the Logloss. Of course, the loss function should be smooth enough and have well-behaved derivatives, otherwise XGBoost will drive crazy. In this course, we consider only a small set of metrics, but there are plenty of them in fact. And for some of them, it is really hard to come up with a neat optimization procedure or write a custom loss function. Thankfully, there is a method that always works. It is called early stopping, and it is very simple. You set a model to optimize any loss function it can optimize and you monitor the desired metric on a validation set. And you stop the training when the model starts to fit according to the desired metric and not according to the metric the model is truly optimizing. That is important. Of course, some metrics cannot be even easily evaluated. For example, if the metric is based on a human assessor's opinions, you cannot evaluate it on every iteration. For such metrics, we cannot use early stopping, but we will never find such metrics in a competition. So, in this video, we have discussed the discrepancy between our target metric and the loss function that our model optimizes. We've reviewed several approaches to target metric optimization and, in particular, discussed early stopping. In the following videos, we will go through the regression and classification metrics and see the hacks we can use to optimize them.[SOUND] So far we've discussed different metrics, their definitions, and intuition for them. We've studied the difference between optimization loss and target metric. In this video, we'll see how we can efficiently optimize metrics used for regression problems. We've discussed, we always can use earl stopping. So I won't mention it for ever metrics. But keep it in mind. Let's start with mean squared error. It's the most commonly used metric for regression tasks. So we should expect it to be easy to work with. In fact, almost every modelling software will implement MSE as a loss function. So all you need to do to optimize it is to turn this on in your favorite library. And here are some of the library that support mean square error optimization. Both XGBoost and LightGBM will do it easily. A RandomForestRegresor from a scaler and also can split based on MSE, thus optimizing individually. A lot of linear models implemented in siclicar, and most of them are designed to optimize MSE. For example, ordinarily squares, reach regression, regression and so on. There's also SGRegressor class and Sklearn. It also implements a linear model but differently to other linear models in Sklearn. It uses [INAUDIBLE] gradient decent to train it, and thus very versatile. Well and of course MSE was built in. The library for online learning of linear models, also accepts MSC as lost function. But every neural net package like PyTorch, Keras, Flow, has MSE loss implemented. You just need to find an example on GitHub or wherever, and see what name MSE loss has in that particular library. For example, it is sometimes called L two loss, as L to distance in Matt Luke's using. But basically for all the metrics we consider in this lesson, you may find plaintal flames since they were used and discovered independently in different communities. Now, what about mean absolute error. MAE is popular too, so it is easy to find a model that will optimize it. Unfortunately, the extra boost cannot optimize MAE because MAE has zero as a second derivative while LightGBM can. So you still can use gradient boosting

decision trees to this metric. MAE criteria was implemented for RandomForestRegressor from Sklearn. But note that running time will be
quite high compared with MSE Corte. Unfortunately, linear models from SKLearn including SG Regressor can not
optimize MAE negatively. But, there is a loss called Huber Loss, it is implemented in some of the models. Basically, it is very similar to MAE,
especially when the errors are large. We will discuss it in the next slide. In [INAUDIBLE], MAE loss is implemented, but under a different name
that's called quantile loss. In fact, MAE is just a special case of quantile loss. Although I will not go into the details here, but just recall that MAE is somehow connected to median values and
median is a particular quantile. What about neural networks? As we've discussed MAE is not
differentiable only when the predictions are equal to target. And it is of a rare case. That is why we may use any model
train to put to optimize MAE. It may be that you will not find MAE
implemented in a neural library, but it is very easy to implement it. In fact, all the models need is a loss function gradient with respect to predictions. And in this case,
this is just a set function. Different names you may encounter for
MAE is, L1 that fit and a one loss, and sometimes people
refer to that special case of quintile regression as
to median regression. A lot, a lot of,
a lot of ways to make MAE smooth. You can actually make up your own smooth
function that have upload that loops like MAE error. The most famous one is Huber loss. It's basically a mix between MSE and MAE. MSE is computed when the error is small,
so we can safely approach zero error. And MAE is computed for large errors given robustness. So, to this end, we discuss the libraries
that can optimize mean square error and mean absolute error. Now, let's get to not ask
common relative metrics. MSPE and MAPE. It's much harder to find the model
which can optimize them out of the box. Of course we can always can use,
either, of course we can always either implement a custom loss for
an integer boost or a neural net. It is really easy to do there. Or we can optimize different metric and
do early stopping. But there are several specific
approaches that I want to mention. This approach is based on the fact that
MSP is a weighted version of MSE and MAP is a weighted version of MAE. On the right side,
we've sen expression for MSP and MAP. The summon denominator just
ensures that the weights are summed up to 1, but it's not required. Intuitively, the sample weights are

indicating how important the object is for us while training the model. The smaller the target, is the more important the object. So, how do we use this knowledge? In fact, many libraries accept sample weights. Say we want to optimize MSP. So if we can set sample weights to the ones from the previous slide, we can use MSE laws with it. And, the model will actually optimize desired MSPE loss. Although most important libraries like XGBoost, LightGBM, most neural net packages support sample weighting, not every library implements it. But there is another method which works whenever a library can optimize MSE or MAE. Nothing else is needed. All we need to do is to create a new training set by sampling it from the original set that we have and fit a model with, for example, I'm a secretarian if you want to optimize MSPE. It is important to set the probabilities for each object to be sampled to the weights we've calculated. The size of the new data set is up to you. You can sample for example, twice as many objects as it was in original train set. And note that we do not need to do anything with the test set. It stays as is. I would also advise you to re-sample train set several times. Each time fitting a model. And then average models predictions, if we'll get the score much better and more stable. The results will, another way we can optimize MSPE, this approach was widely used during Rossmund Competition on Kagle. It can be proved that if the errors are small, we can optimize the predictions in logarithmic scale. Where it is similar to what we will do on the next slide actually. We will not go into details but you can find a link to explanation in the reading materials. And finally, let's get to the last regression metric we have to discuss. Root, mean, square, logarithmic error. It turns out quite easy to optimize, because of the connection with MSE loss. All we need to do is first to apply and transform to our target variables. In this case, logarithm of the target plus one. Let's denote the transformed target with a z variable right now. And then, we need to fit a model with MSE loss to transform target. To get a prediction for a test subject, we first obtain the prediction, z hat, in the logarithmic scale just by calling model.predict or something like that. And next, we do an inverse transform from logarithmic scale back to the original by expatiating z hat and subtracting one, and this is how we obtain the predictions y hat for the test set. In this video, we run through regression matrix and tools to optimize them. MSE and MAE are very common and implemented in many packages. RMSPE and MAPE can be optimized by either resampling the data set or setting proper sample weights. RMSLE is optimized by optimizing MSE in log space. In the next video, we will see optimization techniques for classification matrix. [MUSIC][MUSIC] In this and the next video, we will discuss, what are the ways to optimize classification metrics? In this video, we will discuss logloss and accuracy, and in the next one, AUC and quadratic-weighted kappa. Let's start with logloss, logloss for classification is like MSE for aggression, it is implemented everywhere. All we need to do is to find out what arguments should be passed to a library to make it use logloss for training. There are a huge number of libraries to try, like XGBoost, LightGBM, Logistic Regression, and [INAUDIBLE] classifier from sklearn, Vowpal Wabbit. All neural nets, by default, optimize logloss for classification. Random forest classifier predictions turn out to be quite bad in terms of logloss. But there is a way to make them better, we can calibrate the predictions to better fit logloss. We've mentioned several times that logloss requires model to output exterior probabilities, but what does it mean? It actually means that if we take all the points that have a score of, for example, 0.8, then there will be exactly four times more positive objects than negatives. That is, 80% of the points will be from class 1, and 20% from class 0. If the classifier doesn't directly optimize logloss, its predictions should be calibrated. Take a look at this plot, the blue line shows sorted by value predictions for the validation set. And the red line shows correspondent target values smoothed with rolling window. We clearly see that our predictions are kind of conservative. They´re much greater than two target mean on the left side, and much lower than they should be on the right side. So this classifier is not calibrated, and the green curve shows the predictions after calibration. But if we plot sorted predictions for calibrated classifier, the curve will be very similar to target rolling mean. And in fact, the calibrator predictions will have lower log loss. Now, there are several ways to calibrate predictions, for example, we can use so-called Platt scaling. Basically, we just need to fit a logistic regression to our predictions. I will not go into the details how to do that, but it's very similar to how we stack models, and we will discuss stacking in detail in a different video. Second, we can fit isotonic regression to our predictions, and again, it is done very similar to stacking, just another model. While finally, we can use stacking, so the idea is, we can fit any classifier. It doesn't need to optimize logloss, it just needs to be good, for example, in terms of AUC. And then we can fit another model on top that will take the predictions of our model, and calibrate them properly. And that model on top will use logloss as its optimization loss. So it will be optimizing indirectly, and its predictions will be calibrated. Logloss was the only metric that is easy to optimize directly. With accuracy, there is no easy recipe how to directly optimize it. In general, the recipe is following, actually, if it is a binary classification task, fit any metric, and tune with the binarization threshold. For multi-class tasks, fit any metric and tune parameters comparing the models by their accuracy score, not by the metric that the models were really optimizing. So this is kind of early stopping and the cross validation, where you look at the accuracy score. Just to get an intuition why accuracy is hard to optimize, let's look at this plot. So on the vertical axis we will show the loss, and the horizontal axis shows signed distance to the decision boundary, for example, to a hyper plane or for a linear model. The distance is considered to be positive if the class is predicted correctly. And negative if the object is located at the wrong side of the decision boundary. The blue line here shows zero-one loss, this is the loss that corresponds to accuracy score. We pay 1 if the object is misclassified, that is, the object has negative distance, and we pay nothing otherwise. The problem is that, this loss has zero almost everywhere gradient, with respect to the predictions. And most learning algorithms require a nonzero gradient to fit, otherwise it's not clear how we need to change the predictions such that loss is decreased. And so people came up with proxy losses that are upper bounds for these zero-one loss. So if you perfectly fit the proxy loss, the accuracy will be perfect too, but differently to zero-one loss, they are differentiable. For example, you see here logistic loss, the red curve used in logistic regression, and hinge loss, loss used in SVM. Now recall that to obtain hard lab els for a test object, we usually take argmax of our soft predictions, picking the class with a maximum score. If our task is binary and soft predictions sum up to 1, argmax is equivalent to threshold function. Output 1 when the predictions for the class one is higher than 0.5, and output 0 when the prediction's lower. So we've already seen this example where threshold 0.5 is not optimal, so what can we do? We can tune the threshold we apply, we can do it with a simple grid search implemented with a for loop. Well, it means that we can basically fit any sufficiently powerful model. It will not matter much what loss exactly, say, hinge or log loss the model will optimize. All we want from our model's predictions is the existence of a good threshold that will separate the classes. Also, if our classifier is ideally calibrated, then it is really returning posterior probabilities. And for such a classifier, threshold 0.5 would be optimal, but such classifiers are rarely the case, and threshold tuning helps often. So in this video, we discussed logloss and accuracy, in the next video we will discuss AUC and quadratic weighted kappa. [MUSIC]So in the previous video, we've discussed Logloss and Accuracy. In this video we'll discuss Area Under Curve, AUC, and (Quadratic weighted) Kappa. Let's start with AUC. Although the loss function of AUC has zero gradients almost everywhere, exactly as accuracy loss, there exists an algorithm to optimize AUC with gradient-based methods, and some models implement this algorithm. So we can use it by setting the right parameters. I will give you an idea about this method without much details as there is more than one way to implement it. Recall that originally, classification task is usually solved at the level of objects. We want to assign 0 to red objects, and 1 to the green ones. But we do it independently for each object, and so our loss is pointwise. We compute it for each object individually, and sum or average the losses for all the objects to get a total loss. Now, recall that AUC is the probability of a pair of the objects to be ordered in the right way. So ideally, we want predictions Y^ for the green objects to be larger than for the red ones. So, instead of working with single objects, we should work with pairs of objects. And instead of using pointwise loss, we should use pairwise loss. A pairwise loss takes predictions and labels for a pair of objects and computes their loss. Ideally, the loss would be zero when the ordering is correct, and greater than zero when the ordering is not correct, incorrect. But in practice, different loss functions can be used. For example, we can use logloss. We may think that the target for this pairwise loss is always one, red minus green should be one. That is why there is only one term in logloss objective instead of two. The prob function in the formula is needed to make sure that the difference between the predictions is still in the 0,1 range, and I use it here just for the sake of simplicity. Well, basically, XGBoost, LightGBM have pairwise loss we've discussed implemented. It is straightforward to implement i

n any neural net library, and for sure, you can find implementations on GitHub. I should say that in practice, most people still use logloss as an optimization loss without any more post processing. I personally observed XGBoost learned with loglosst to give comparable AUC score to the one learned with pairwise loss. All right. Now, let's move to the last topic to discuss. It is Quadratic weighted Kappa metric. There are two methods. One is very common and very easy, the second is not that common and will require you to implement a custom loss function for either XGBoost or neural net. But we've already implemented it for XGBoost, so you will be able to find the implementation in the reading materials. But let's start with the simple one. Recall that we're solving an ordered classification problem and our labels can be found of us integer ratings, say from one to five. The task is classification as we cannot output, for example, 4.5 as an answer. But anyway, we can treat it as a regression problem, and then somehow, post-process the predictions and convert them to integer ratings. And actually quadratic weights make Kappa as somehow similar to regression with MSE loss. If we allow our predictions to take values between the labels, that is relax the predictions. But in fact, it is different to MSE. So if relaxed, Kappa would be one minus MSE divided by something that really depends on the predictions. And it looks like everyone's logic is, well, there is MSE in the denominator, we can optimize it, and let's don't care about denominator. Well, of course it's not correct way to do it, but it turns out to be useful in practice. But anyway, MSE gives us flat values instead of integers. So now, we need somehow to convert them into integers. And the straightforward way would be to do rounding all the predictions. But we can think about rounding as of applying a threshold. Like if the value is greater than 3.5 and less than 4.5, then output 3. But then we can ask ourselves a question, why do we use exactly those thresholds? Let's tune them. And again, it's just straightforward, it can be easily done with grid search. So to summarize, we need to fit MSE loss to our data and then find appropriate thresholds. Finally, there is a paper which suggests a way to relax classification problem to regression, but it deals with this- hard to deal with part in denominator that we had. I will not get into the details here, but it's clearly written and easy to understand paper, so I really encourage you to read it. And more, you can find loss implementation in the reading materials, and just use it if you don't want to read the paper. Finally, we finished this lesson. We've discussed that evaluation or target metric is how all submissions are scored. We've discussed the difference between target metric and optimization loss. Optimization loss is what our model optimizes, and it is not always the same as target metric that we want to optimize. Sometimes, we only can set our model to optimize completely different to target metric. But later, we usually try to post-process the predictions to make them better fit target metric. We've discussed intuition behind different metrics for regression and classification tasks, and saw how to efficiently optimize different metrics. I hope you've enjoyed this lesson, and see you later.[MUSIC] Hi, everyone. In this section, we'll cover a very
powerful technique, mean encoding. It actually has a number of names. Some call it likelihood encoding,

some target encoding, but in this course, we'll stick with plain mean encoding. The general idea of this technique is to add new variables based on some feature to get where we started,. In simplest case, we encode each level of categorical variable with corresponding target mean. Let's take a look at the following example. Here, we have some binary classification task in which we have a categorical variable, some city. And of course, we want to numerically encode it. The most obvious way and what people usually use is label encoding. It's what we have in second column. Mean encoding is done differently, via encoding every city with corresponding mean target. For example, for Moscow, we have five rows with three 0s and two 1s. So we encode it with 2 divided by 5 or 0.4. Similarly, we deal with the rest of cities, pretty straightforward. What I've described here is a very high level idea. There are a huge number of pitfalls one should overcome in actual competition. We went deep into details for now, just keep it in mind. At first, let me explain. Why does it even work? Imagine, that our dataset is much bigger and contains hundreds of different cities. Well, let's try to compare, of course, very abstractly, mean encoding with label encoding. We plot future histograms for class 0 and class 1. In case of label encoding, we'll always get total and random picture because there's no logical order, but when we use mean target to encode the feature, classes look way more separable. The plot looks kind of sorted. It turns out that this sorting quality of mean encoding is quite helpful. Remember, what is the most popular and effective way to solve machine learning problem? Is grading using trees, [INAUDIBLE] OIGBM. One of the few downsides is an inability to handle high cardinality categorical variables. Trees have limited depth, with mean encoding, we can compensate it, we can reach better loss with shorter trees. Cross validation loss might even look like this. In general, the more complicated and non linear feature target dependency, the more effective is mean encoding, okay. Further in this section, you will learn how to construct mean encodings. There are actually a lot of ways. Also keep in mind that we use classification tests only as an example. We can use mathematics on other tests as well. The main idea remains the same. Despite the simplicity of the idea, you need to be very careful with validation. It's got to be impeccable. It's probably the most important part. Understanding the correct linkless validation is also a basis for staking. The last, but not least, are extensions. There are countless possibilities to derive new features from target variable. Sometimes, they produce significant improvement for your models. Let's start with some characteristics of data sets, that indicate the usefulness of main encoding. The presence of categorical variables with a lot of levels is already a good indicator, but we need to go a little deeper. Let's take a look at each of these learning logs from Springleaf competition. I ran three models with different depths, 7, 9, and 11. Train logs are on the top plot. Validation logs are on the bottom one. As you can see, with increasing the depths of trees, our training care becomes better and better, nearly perfect and that's a normal part. But we don't actually over feed and that's weird. Our validation score also increase, it's a sign that trees need a huge number of splits to extract information from some variables. And we can check it for mortal dump. It turns out that some features have a tremendous amount of split points, like 1200 or 1600 and that's a lot. Our model tries to treat all those categories differently and they are also very important for predicting the target. We can help our model via mean encodings. There is a number of ways to calculate encodings. The first one is the one we've been discussing so far. Simply taking mean of target variable. Another popular option is to take initial logarithm of this value, it's called weight of evidence. Or you can calculate all of the numbers of ones. Or the difference between number of ones and the number of zeros. All of these are variable options. Now, let's actually construct the features. We will do it on sprinkled data set, suppose we've already separated the data for train and validation, X_tr and X val data frames. These called snippet shows how to construct mean encoding for an arbitrary column and map it into a new data frame, train new and val new. We simply do group by on that column and use target as a map. Resulting commands were able [INAUDIBLE]. It is then mapped to tree and validation data sets by a map operator. After we've repeated this process for every call, we can fit each of those model on this new data. But something's definitely not right, after several efforts training AOC is nearly 1, while on validation, the score set rates around 0.55, which is practically noise. It's a clear sign of terrible overfitting. I'll explain what happened in a few moments. Right now, I want to point out that at least we validated correctly. We separated train and validation, and used all the train data to estimate mean encodings. If, for instance, we would have estimated mean encodings before train validation split, then we would not notice such an overfitting. Now, let's figure out the reason of overfitting. When they are categorized, it's pretty common to get results like in an example, target 0 in train and target 1 in validation. Mean encodings turns into a perfect feature for such categories. That's why we immediately get very good scores on train and fail hardly on validation. So far, we've grasped the concept of mean encodings and walked through some trivial examples, that obviously can not use mean encodings like this in practice. We need to deal with overfitting first, we need some kind of regularization. And I will tell you about different methods in the next video. [MUSIC][MUSIC] In previous video, we realized that mean encodings cannot be used as is and requires some kind of regularization on training part of data. Now, we'll carry out four different methods of regularization, namely, doing a cross-validation loop to construct mean encodings. Then, smoothing based on the size of category. Then, adding random noise. And finally, calculating expanding mean on some parametrization of data. We will go through all of these methods one by one. Let's start with CV loop regularization. It's a very intuitive and robust method. For a given data point, we don't want to use target variable of that data point. So we separate the data into K-node intersecting subsets, or in other words, folds. To get mean encoding value for some subset, we don't use data points from that subset and estimate the encoding only on the rest of subset. We iteratively walk through all the data subsets. Usually, four or five folds are enough to get decent results. You don't need to tune this number. It may seem that we have completely avoided leakage from target variable. Unfortunately, it's not true. It will become apparent if we perform leave one out scheme to separate the data. I'll return to it a little later, but first let's learn how to apply this method in practice. Suppose that our training data is in a DFTR data frame. We will add mean encoded features into another train new data frame. In the outer loop, we iterate through stratified K-fold iterator in order to separate training data into chunks. X_tr is used to estimate the encoding. X_val is used to apply estimating encoding. After that, we iterate through all the columns and map estimated encodings to X_val data frame. At the end of the outer loop we fill train new data frame with the result. Finally, some rare categories may be present only in a single fold. So we don't have the data to estimate target mean for them. That's why we end up with some nans. We can fill them with global mean. As you can see, the whole process is very simple. Now, let's return to the question of whether we leak information about target variable or not. Consider the following example. Here we want to encode Moscow via leave-one-out scheme. For the first row, we get 0.5, because there are two 1s and two 0s in the rest of rows. Similarly, for the second row we get 0.25 and so on. But look closely, all the resulting and the resulting features. It perfect splits the data, rows with feature mean equal or greater than 0.5 have target 0 and the rest of rows has target 1. We didn't explicitly use target variable, but our encoding is biased. Furthermore, this effect remains valid even for the KFold scheme, just milder. So is this type of regularization useless? Definitely not. In practice, if you have enough data and use four or five folds, the encodings will work fine with this regularization strategy. Just be careful and use correct validation. Another regularization method is smoothing. It's based on the following idea. If category is big, has a lot of data points, then we can trust this to [INAUDIBLE] encoding, but if category is rare it's the opposite. Formula on the slide uses this idea. It has hyper parameter alpha that controls the amount of regularization. When alpha is zero, we have no regularization, and when alpha approaches infinity everything turns into globalmean. In some sense alpha is equal to the category size we can trust. It's also possible to use some other formula, basically anything that punishes encoding software categories can be considered smoothing. Smoothing obviously won't work on its own but we can combine it with for example, CD loop regularization. Another way to regularize encodence is to add some noise without regularization. Meaning codings have better quality for the [INAUDIBLE] data than for the test data. And by adding noise, we simply degrade the quality of encoding on training data. This method is pretty unstable, it's hard to make it work. The main problem is the amount of noise we need to add. Too much noise will turn the feature into garbage, while too little noise means worse regularization. This method is usually used together with leave one out regularization. You need to diligently fine tune it. So, it's probably not the best option if you don't have a lot of time. The last regularization method I'm going to cover is based on expanding mean. The idea is very simple. We fix some sorting order of our data and use only rows from zero to n minus one to calculate encoding for row n. You can check simple implementation in the code snippet. Cumsum stores cumulative sum of target variable up to the given row and cumcnt stores cumulative count. This method introduces the least amount of leakage from target variable and it requires no hyper parameter tuning. The only downside is that feature quality is not uniform. But it's not a big deal. We can average models on encodings calculated from different data permutations. It's also worth noting that it is expanding mean method that is used in CatBoost grading, boosting to it's library, which proves to perform magnificently on data sets with categorical features. Okay, let's summarize what we've discussed in this video. We covered four different types of regularization. Each of them has its own advantages and disadvantages. Sometimes unintuitively we introduce target variable leakage. But in practice, we can bear with it. Personally, I recommend CV loop or expanding mean methods for practical tasks. They are the most robust and easy to tune. This is was regularization. In the next video, I will tell you about various extensions and practical applications of mean encodings. Thank you. [MUSIC][SOUND] In the final video, we will cover various generalizations and extensions of mean encodings. Namely how to do meaning coding in regression and multiclass tasks. How can we apply encoding to domains with many-to-many relations. What features can we build based on target we're able in time series. And finally, how to encode numerical features and interactions of features. Let's start with regression tasks. They are actually more flexible for feature encoding. Unlike binary classification where a mean is frankly the only meaningful statistic we can extract from target variable. In regression tasks, we can try a variety of statistics, like medium, percentile, standard deviation of target variable. We can even calculate some distribution bins. For example, if target variable is distributed between 1 and 100, we can create 10 bin features. In the first feature, we'll count how many data points have targeted between 1 and 10, in the second between 10 and 20 and so on. Of course, we need to realize all of these features. In a nutshell, regression tasks are like classification. Just more flexible in terms of feature engineering. Men encoding for multi-class tasks is also pretty straightforward. For every feature we want to encode, we will have n different encodings where n is the number of classes. It actually has non obvious advantage. Three models for example, usually solve multi-class task in one versus old fashion. So every class had a different model, and when we feed that model, it doesn't have any information about structure of other classes because they are merge into one entity. Therefore, together with mean encodings, we introduce some additional information about the structure of other classes. The domains with many-to-many relations are usually very complex and require special approaches to create mean encodings. I will give you only a very high level idea, consider an example. Binary classification task for users based on apps installed on their smartphones. Each user may have multiple apps and each app is used by multiple users. Hence, many-to-many relation. We want to mean encode apps. The hard part we need to deal with is that the user may have a lot of apps. So let's take a cross product of user and app entities. It will result in a so called long representation of data. We will have a role for each user app pair. Using this table, we can naturally calculate mean encoding for apps. So now every app is encoded with target mean, but how to map it back to users. Every user has a number of apps, so instead of app1, app2, app3, we will now have a vector like 0.1, 0.2, 0.1. That was pretty simple. We can collect various statistics from those vectors, like mean, minimal, maximum, standard deviation, and so on. So far we assume that our data has no inner structure, but with time series we can obviously use future information. On one hand, it's a limitation, on the other hand, it actually allows us to make some complicated features. In data sets without time component when encoding the category, we are forced to use all the rules to calculate the statistic. It makes no sense to choose some subset of rules. Presence of time changes it. For a given category, we can't. For example, calculate the mean from previous day, previous two days, previous week, etc. Consider an example. We need to predict which categories users spends money. In these two example we have a period of two days, two users, and three spending categories. Some good features would be the total amount of money users spent in previous day. An average amount of money spent by all users in given category. So, in day 1, user 101 spends $6, user 102, $3. Therefore, we feel those numbers as future values for day 2. Similarly, with the average amount by category. The more data we have, the more complicated features we can create. In practice, it is often been official to mean encode numeric features and some combination of features. To encode a numeric feature, we only need to bin it and then treat as categorical. Now, we need to answer two questions. First, how to bin numeric feature, and second how to select useful combination of features. Well, we can find it out from a model structure by analyzing the trees. So at first, we take for example, [INAUDIBLE] model and raw features without any encodings. Let's start with numeric features. If numeric feature has a lot of [INAUDIBLE] points, it means that it has some complicated dependency with target and its was trying to mean encode it. Furthermore, these exact split points may be used to bin the feature. So by analyzing model structure, we both identify suspicious numeric feature and found a good way to bin it. It's going to be a little harder with selecting interactions, but nothing extraordinary. First, let's define how to extract to way interaction from decision tree. The process will be similar for three way, four way arbitrary way interactions. So two features interact in a tree if they are in two neighbouring notes. With that in mind, we can iterate through all the trees in the model and calculate how many times each feature interaction appeared. The most frequent interactions are probably worthy of mean encoding. For example, if we found that feature one and feature two pair is most frequent, then we can concatenate that those feature values in our data. And mean encode resulting interaction. Now let me illustrate how important interaction encoding may be. Amazon Employee Access Challenge Competition has a very specific data set. There are only nine categorical features. If we blindly fit say like GBM model on the raw features, then no matter how we return the parameters, we'll score in a 0.87 AUC range. Which will place roughly on 700 position on the leaderboard. Furthermore, even if we mean encode all the labels, we won't have any progress. But if we fit cat boost model, which internally mean encodes some feature interactions, we will immediately score in 0.91 range, which will place us onto win this position. The difference in both absolute AUC values and relative leaderboard positions is tremendous. Also note that cat boost is no silver bullet. In order to get even higher on the leader board, would still need to manually add more mean encoded interactions. In general, if you participate in a competition with a lot of categorical variables, it's always worth trying to work with interactions and mean encodings. I also want to remind you about correct validation process. During all local experiments, you should at first split data in X_tr and X_val parts. Estimate encodings on X_tr, map them to X_tr and X_val, and then regularize them on X_tr and only after that validate your model on X_tr / X_val split. Don't even think about estimating encodings before splitting the data. And at submission stage, you can estimate encodings on whole train data. Map it to train and test, then apply regularization on training data and finally fit a model. And note that you should have already decided on regularization method and its strength in local experiments. At the end of this section, let's summarize main advantages and disadvantages of mean encodings. First of all, mean encoding allows us to make a compact transformation of categorical variables. It is also a powerful basis for feature engineering. Then the main disadvantage is target rebel leakage. We need to be very careful with validation and irregularization. It also works only on specific data sets. It definitely won't help in every competition. But keep in mind, when this method works, it may produce significant improvements. Thank you for your attention. [MUSIC]