[MUSIC] Hi, in this lecture, we will study
hyperparameter optimization process and talk about hyperparamete
rs in
specific libraries and models. We will first discuss
hyperparameter tuning in general. General pipeline, ways to tuni
ng
hyperparameters, and what it actually means to understand how a
particular
hyperparameter influences the model. It is actually what we will
discuss in this video, and then we will talk about libraries and
frameworks, and see how to tune hyperparameters
of several types of models. Namely, we will first
study tree-based models, gradient boosting decision trees and
RandomForest. Then I'll review important
hyperparameters in neural nets. And finally, we will talk about
linear models, where to find them and how to tune them. Another
class of interesting
models is factorization machines. We will not discuss factorizat
ion
machines in this lecture, but I suggest you to read
about them on the internet. So, let's start with a general
discussion of a model tuning process. What are the most importan
t things to
understand when tuning hyperparameters? First, there are tons of
potential
parameters to tune in every model. And so we need to realize whi
ch
parameters are affect the model most. Of course,
all the parameters are reliable, but we kind of need to select
the most important ones. Anyway we never have time to tune
all the params, that's right. So we need to come up with a nice
subset of parameters to tune. Suppose we're new to xgboost and w
e're trying to find out what
parameters will better to tune, and say we don't even understand
how
gradient boosting decision tree works. We always can search what
parameters
people usually set when using xgboost. It's quite easy to look u
p, right? For example, at GitHub or Kaggle Kernels. Finally, the
documentation sometimes
explicitly states which parameter to tune first. From the select
ed set of parameters
we should then understand what would happen if we
change one of the parameters? How the training process and the t
raining
invalidation course will change if we, for example,
increased a certain parameter? And finally, actually tune
the selected parameters, right? Most people do it manually. Just
run, examine the logs,
change parameters, run again and
iterate till good parameters found. It is also possible to use
hyperparameter optimization tools like hyperopt, but it's usuall
y
faster to do it manually to be true. So later in this video, act
ually discuss
the most important parameters for some models along with some in

tuition how
to tune those parameters of those models. But before we start, I actually want
to give you a list of libraries that you can use for
automatic hyperparameter tuning. There are lots of them actually, and
I didn't try everything from this list myself, but from what I actually tried,
I did not notice much difference in optimization speed on
real tasks between the libraries. But if you have time,
you can try every library and compare. From a user side these
libraries are very easy to use. We need first to define the function
that will run our module, in this case, it is XGBoost. That will run our module with
the given set of parameters and return a resulting validation score. And second,
we need to specify a source space. The range for the hyperparameters where
we want to look for the solution. For example, here we see that a parameter,
it is fix 0.1. And we think that optimal max depth
is somewhere between 10 and 30. And actually that is it,
we are ready to run hyperopt. It can take much time, so
the best strategy is to run it overnight. And also please note that everything
we need to know about hyperparameter's, in this case,
is an adequate range for the search. That's pretty convenient,
if you don't know the new model and you just try to run. But still,
most people tuned the models manually. So, what exactly does it mean to understand how parameter influences the model? Broadly speaking, different values of parameters result
in three different fitting behavior. First, a model can underfit. That is, it is so constrained that
it cannot even learn the train set. Another possibility is that the model is so powerful that it just overfits to the train set and
is not able to generalize it all. And finally, the third behavior is something
that we are actually looking for. It's somewhere between underfitting and
overfitting. So basically, what we should examine
while turning parameters is that we should try to understand if the model
is currently underfitting or overfitting. And then, we should somehow
adjust the parameters to get closer to desired behavior. We need to kind of split all the
parameters that we would like to tune into two groups. In the first group, we'll have
the parameters that constrain the model. So if we increase
the parameter from that group, the model would change its behaviour
from overfitting to underfitting. The larger the value of the parameter,

the heavier the constraint. In the following videos, we'll color such
parameters in red, and the parameters in the second group are doing an opposite
thing to our training process. The higher the value,
more powerful the main module. And so by increasing such parameters, we can change fitting behavior
from underfitting to overfitting. We will use green color for
such parameters. So, in this video we'll be discussing some general aspects of
hyperparameter organization. Most importantly,
we've defined the color coding. If you did not understand
what color stands for what, please watch a part of
the video about it again. We'll use this color coding
throughout the following videos. [MUSIC][MUSIC] In this video, we will talk about
hyperparameter optimization for some tree based models. Nowadays, XGBoost and
LightGBM became really gold standard. They are just awesome implementation
of a very versatile gradient boosted decision trees model. There is also a CatBoost library it
appeared exactly at the time when we were preparing this course, so CatBoost
didn't have time to win people's hearts. But it looks very interesting and
promising, so check it out. There is a very nice
implementation of RandomForest and ExtraTrees models sklearn. These models are powerful, and
can be used along with gradient boosting. And finally, there is a model
called regularized Greedy Forest. It showed very nice results from several
competitions, but its implementation is very slow and hard to use, but
you can try it on small data sets. Okay, what important parameters do
we have in XGBoost and LightGBM? The two libraries have similar parameters
and we'll use names from XGBoost. And on the right half of the slide
you will see somehow loosely corresponding parameter
names from LightGBM. To understand the parameters,
we better understand how XGBoost and LightGBM work at least a very high level. What these models do, these models
build decision trees one after another gradually optimizing a given objective. And first there are many parameters
that control the tree building process. Max_depth is the maximum depth of a tree. And of course, the deeper a tree can be
grown the better it can fit a dataset. So increasing this parameter will lead
to faster fitting to the train set. Depending on the task,
the optimal depth can vary a lot, sometimes it is 2, sometimes it is 27. If you increase the depth and can not get
the model to overfit, that is, the model is becoming better and better on the

validation set as you increase the depth. It can be a sign that there are a lot
of important interactions to extract from the data. So it's better to stop tuning and
try to generate some features. I would recommend to start with a max_depth of about seven. Also remember that as
you increase the depth, the learning will take a longer time. So do not set depth to
a very higher values unless you are 100% sure you need it. In LightGBM,
it is possible to control the number of leaves in the tree rather
than the maximum depth. It is nice since a resulting
tree can be very deep, but have small number of leaves and
not over fit. Some simple parameter controls a fraction
of objects to use when feeding a tree. It's a value between 0 and 1. One might think that it's better
always use all the objects, right? But in practice,
it turns out that it's not. Actually, if only a fraction of objects is used at every duration, then the model is less
prone to overfitting. So using a fraction of objects, the model will fit slower on the train set, but at the same time it will probably generalize
better than this over-fitted model. So, it works kind of as a regularization. Similarly, if we can consider only
a fraction of features [INAUDIBLE] split, this is controlled by parameters
colsample_bytree and colsample_bylevel. Once again, if the model is over fitting, you can try to lower
down these parameters. There are also various regularization parameters, min_child_weight, lambda, alpha and others. The most important one
is min_child_weight. If we increase it,
the model will become more conservative. If we set it to 0,
which is the minimum value for this parameter,
the model will be less constrained. In my experience, it's one of the most important parameters
to tune in XGBoost and LightGBM. Depending on the task,
I find optimal values to be 0, 5, 15, 300, so do not hesitate to try a wide
range of values, it depends on the data. To this end we were discussing
hyperparameters that are used to build a tree. And next, there are two very important
parameters that are tightly connected, eta and num_rounds. Eta is essentially a learning weight,
like in gradient descent. And the num_round is the how many learning steps we want to perform or in other words how many tree's we want to build. With each iteration
a new tree is built and added to the model with
a learning rate eta. So in general,
the higher the learning rate, the faster the model fits to the train set
and probably it can lead to over fitting. And more steps model does,
the better the model fits. But there are several caveats here. I

t happens that with a too high learning
rate the model will not fit at all, it will just not converge. So first, we need to find out if we
are using small enough learning rate. On the other hand,
if the learning rate is too small, the model will learn nothing
after a large number of rounds. But at the same time, small learning rate
often leads to a better generalization. So it means that learning
rate should be just right, so that the model generalize and
doesn't take forever to train. The nice thing is that we can freeze
eta to be reasonably small, say, 0.1 or 0.01, and then find how many rounds we
should train the model til it over fits. We usually use early stopping for it. We monitor the validation loss and exit
the training when loss starts to go up. Now when we found
the right number of rounds, we can do a trick that
usually improves the score. We multiply the number of
steps by a factor of alpha and at the same time,
we divide eta by the factor of alpha. For example, we double the number
of steps and divide eta by 2. In this case, the learning will
take twice longer in time, but the resulting model
usually becomes better. It may happen that the valid parameters
will need to be adjusted too, but usually it's okay to leave them as is. Finally, you may want to
use random seed argument, many people recommend to
fix seed before hand. I think it doesn't make too much
sense to fix seed in XGBoost, as anyway every changed parameter will
lead to completely different model. But I would use this
parameter to verify that different random seeds do not
change training results much. Say [INAUDIBLE] competition,
one could jump 1,000 places up or down on the leaderboard just by training
a model with different random seeds. If random seed doesn't
affect model too much, good. In other case, I suggest you to think
one more time if it's a good idea to participate in that competition as
the results can be quite random. Or at least I suggest you to adjust
validation scheme and account for the randomness. All right,
we're finished with gradient boosting. Now let's get to RandomForest and
ExtraTrees. In fact, ExtraTrees is just a more
randomized version of RandomForest and has the same parameters. So I will say RandomForest
meaning both of the models. RandomForest and ExtraBoost build trees,
one tree after another. But, RandomForest builds each
tree to be independent of others. It means that having a lot of trees
doesn't lead to overfeeding for RandomForest as opposed
to gradient boosting. In sklearn, the number of trees for random

forest is controlled
by N_estimators parameter. At the start, we may want to determine what number
of trees is sufficient to have. That is, if we use more than that,
the result will not change much, but the models will fit longer. I usually first set N_estimators
to very small number, say 10, and see how long does it take
to fit 10 trees on that data. If it is not too long then I set
N_estimators to a huge value, say 300, but it actually depends. And feed the model. And then I plot how the validation
error changed depending on a number of used trees. This plot usually looks like that. We have number of trees on the x-axis and
the accuracy score on y-axis. We see here that about 50 trees
already give reasonable score and we don't need to use more
while tuning parameter. It's pretty reliable to use 50 trees. Before submitting to leaderboard, we can set N_estimators to
a higher value just to be sure. You can find code for this plot, actually, in the reading materials. Similarly to XGBoost, there is a parameter max_depth
that controls depth of the trees. But differently to XGBoost, it can be set to none,
which corresponds to unlimited depth. It can be very useful actually when
the features in the data set have repeated values and important interactions. In other cases, the model with unconstrained
depth will over fit immediately. I recommend you to start with a depth
of about 7 for random forest. Usually an optimal depth for
random forests is higher than for gradient boosting, so do not hesitate
to try a depth 10, 20, and higher. Max_features is similar to call
sample parameter from XGBoost. The more features I use to decipher
a split, the faster the training. But on the other hand,
you don't want to use too few features. And min_samples_leaf is a regularization parameter similar to min_child_weight from XGBoost and
the same as min_data_leaf from LightGPM. For Random Forest classifier,
we can select a criterion to eleviate a split in the tree with
a criterion parameter. It can be either Gini or Entropy. To choose one, we should just try both and
pick the best performing one. In my experience Gini is better more
often, but sometimes Entropy wins. We can also fix random seed using
random_state parameter, if we want. And finally, do not forget to set n_jobs
parameter to a number of cores you have. As by default, RandomForest from sklearn
uses only one core for some reason. So in this video, we were talking
about various hyperparameters of gradient boost and
decision trees, and random forest. In the following video, we'll

discuss neural networks and linear models. [MUSIC][MUSIC] In this video we'll briefly discuss neural network libraries and then we'll see how to tune hyperparameters for neural networks and linear models. There are so many frameworks, Keras, TensorFlow, MxNet, PyTorch. The choice is really personal, all frameworks implement more than enough functionality for competition tasks. Keras is for sure the most popular in Kaggle and has very simple interface. It takes only several dozen lines to train a network using Keras. TensorFlow is extensively used by companies for production. And PyTorch is very popular in deep learning research community. I personally recommend you to try PyTorch and Keras as they are most transparent and easy to use frameworks. Now, how do you tune hyperparameters in a network? We'll now talk about only dense neural networks, that is the networks that consist only of fully connected layers. Say we start with a three layer neural network, what do we expect to happen if we increase the number of neurons per layer? The network now can learn more complex decision boundaries and so it will over fit faster. The same should happen when the number of layers are increased, but due to optimization problems, the learning can even stop to converge. But anyway, if you think your network is not powerful enough, you can try to add another layer and see what happens. My recommendation here is to start with something very simple, say 1 or 2 layer and 64 units per layer. Debug the code, make sure the training and [INAUDIBLE] losses go down. And then try to find a configuration that is able to overfit the training set, just as another sanity check. After it, it is time to tune something in the network. One of the crucial parts of neural network is selected optimization method. Broadly speaking, we can pick either vanilla stochastic gradient descent with momentum or one of modern adaptive methods like Adam, Adadelta, Adagrad and so on. On this slide, the adaptive methods are colored in green, as compared to SGD in red. I want to show here that adaptive methods do really allow you to fit the training set faster. But in my experience, they also lead to overfitting. Plain old stochastic gradient descent converges slower, but the trained network usually generalizes better. Adaptive methods are useful, but in the settings others in classification and regression. Now here is a question for you. Just keep the size. What should we expect when increasing a batch size with other hyperparameters fixed? In fact, it turns out that huge batch size leads to more overfitting. Say a batch of 500 objects

is large in experience. I recommend to pick a value around 32 or 64. Then if you see the network is
still overfitting try to decrease the batch size. If it is under fitting, try to increase it. Know that is a the number
of outbox is fixed, then a network with a batch
size reduced by a factor of 2 gets updated twice more times
compared to original network. So take this into consideration. Maybe you need to reduce the number of
networks or at least somehow adjust it. The batch size also should not be too
small, the gradient will be too noisy. Same as in gradient boosting,
we need to set the proper learning rate. When the learning rate is too high,
network will not converge and with too small a learning rate,
the network will learn forever. The learning rate should be
not too high and not too low. So the optimal learning rate
depends on the other parameters. I usually start with a huge learning rate,
say 0.1, and try to lower it down till I find one with which network converges
and then I try to revise further. Interestingly, there is a connection
between the batch size and the learning rate. It is theoretically grounded for
a specific type of models, but people usually use it,
well actually some people use it as a rule of thumb with neural networks. The connection is the following. If you increase the batch
size by a factor of alpha, you can also increase the learning
rate by the same factor. But remember that the larger batch size, the more your network is
prone to overfitting. So you need a good regularization here. Sometime ago, people mostly use L2 and
L1 regularization for weights. Nowadays, most people use
dropout regularization. So whenever you see a network overfitting,
try first to a dropout layer. You can override dropout probability and a
place where you insert the dropout layer. Usually people add the dropout layer
closer to the end of the network, but it's okay to add some dropout
to every layer, it also works. Dropout helps network to find features
that really matters, and what never worked for me is to have dropout as the very
first layer, immediately after data layer. This way some information is lost
completely at the very beginning of the network and
we observe performance degradation. An interesting regularization
technique that we used in the [UNKOWN] competition is
static dropconnect, as we call it. So recall that, usually we have an input
layer densely connected to, say 128 units. We will instead use a

first
hidden layer with a very huge number of units, say 4,096 units.
This is a huge network for a usual
competition and it will overfeed badly. But now to regularlize it,
we'll at random drop 99% of connections between the input layer and
the first hidden layer. We call it static dropconnect
because originally in dropconnect, we need to drop random connections at
every learning iterations while we fix connectivity pattern for the network
for the whole learning process. So you see the point, we increase
the number of hidden units, but the number of parameters in the first
hidden layer remains small. Notice that anyway the weight matrix
of the second layer becomes huge, but it turns out to be
okay in the practice. This is very powerful regularizations. And more of the networks with
different connectivity patterns makes much nicer than networks
without static dropconnect. All right, last class of models
to discuss are my neuro models. Yet, a carefully tuned live GPM would
probably beat support vector machines, even on a large, sparse data set. SVM's do not require almost any tuning,
which is truly beneficial. SVM's for classification and regression
are implemented in SK learners or wrappers to algorithms from libraries
called libLinear and libSVM. The latest version of libLinear and
libSVM support multicore competitions, but unfortunately it is not possible
to use multicore version in Sklearn, so we need to compile these libraries
manually to use this option. And I've never had anyone
use kernel SVC lately, so in this video we will
talk only about linear SVM. In Sklearn we can also find logistic and
linear regression with various regularization options and also,
as your declassifier and regressor. We've already mentioned them
while discussing metrics. For the data sets that do not fit in
the memory, we can use Vowpal Wabbit. It implements learning of linear
models in online fashion. It only reads data row by row
directly from the hard drive and never loads the whole
data set in the memory. Thus, allowing to learn
on a very huge data sets. A method of online learning for linear
models called flow the regularized leader or FTRL in short was particularly
popular some time ago. It is implemented in Vowpal Wabbit but there are also lots of
implementations in pure Python. The hyperparameters we usually need
to tune linear models are L2 and L1 regularization of weights. Once again, regularization is denoted

with red color because of the higher the regularization weight is the more
model struggle to learn something. But know that, the parameter see in
SVM is inversely proportional to regularization weight, so
the dynamics is opposite. In fact, we do not need to think about the
meaning of the parameters in the case of one parameter, right? We just try several values and
find one that works best. For SVM, I usually start a very small
seed, say 10 to the power of minus 6 and then I try to increase it,
multiply each time by a factor of 10. I start from small values because
the larger the parameter C is, the longer the training takes. What type of regularization,
L1 or L2 do you choose? Actually, my answer is try both. To my mind actually they are quite similar
and one benefit that L1 can give us is weight sparsity, so the sparsity
pattern can be used for feature selection. A general advise I want to give
here do not spend too much time on tuning hyperparameters, especially
when the competition has only begun. You cannot win a competition
by tuning parameters. Appropriate features, hacks,
leaks, and insights will give you much more than carefully
tuned model built on default features. I also advice you to be patient. It was my personal mistake several times. I hated to spend more then ten minutes
on training models and was amazed how much the models could improve if I
would let it train for a longer time. And finally, average everything. When submitting, learn five models
starting from different random initializations and
average their predictions. It helps a lot actually and
some people average not only random seed, but also other parameters
around an optimal value. For example, if optimal depth for
extra boost is 5, we can average 3 digiboosts with depth 3,
4, and 5. Wow, it would be better if we could averaged 3 digiboosts with depth 4,
5, and 6. Finally, in this lecture, we discussed what is a general pipeline
for a hyperparameter optimization. And we saw, in particular,
what important hyperparameters derive for several models,
gradient boosting decision trees, random forests and extra trees,
neural networks, and linear models. I hope you found something interesting
in this lecture and see you later. [MUSIC][SOUND] Hi, to this moment,
we have already discussed all basics new things which build up to
a big solution like featured generation, validation, minimalist

codings and so on. We went through several
competitions together and tried our best to unite everything
we learn into one huge framework. But as with any other set of t
ools,
there are a lot of heuristics which people often find only with
a trial and
error approach, spending significant time on learning
how to use these tools efficiently. So to help you out here, in
this video we'll share things we
learned the hard way, by experience. These things may vary from
one person to another. So we decided that everyone on class will
present his own guidelines personally, to stress the possible
diversity in a broad issues and to make an accent on different m
oments. Some notes might seem obvious to you,
some may not. But be sure for even some of them or
at least no one involve them. Can save you a lot of time. So, le
t's start. When we want to enter a competition,
define your goals and try to estimate what you can
get out of your participation. You may want to learn more
about an interesting problem. You may want to get acquainted
with new software tools and packages, or
you may want to try to hunt for a medal. Each of these goals wil
l influence what
competition you choose to participate in. If you want to learn m
ore
about an interesting problem, you may want the competition to ha
ve
a wide discussion on the forums. For example, if you are interes
ted in
data science, in application to medicine, you can try to predict
 lung cancer
in the Data Science Bowl 2017. Or to predict seizures in long
term human EEG recordings. In the Melbourne University
Seizure Prediction Competition. If you want to get acquainted
with new software tools, you may want the competition
to have required tutorials. For example, if you want to
learn a neural networks library. You may choose any of competiti
ons with
images like the nature conservancy features, monitoring competit
ion. Or the planet, understanding
the Amazon from space competition. And if you want to try to hun
t for a medal, you may want to check how
many submissions do participants have. And if the points that pe
ople have
over one hundred submissions, it can be a clear sign of legible
problem or difficulties in validation includes an inconsistency
of
validation and leaderboard scores. On the other hand, if there a
re people
with few submissions in the top, that usually means there should
 be
a non-trivial approach to this competition or it's discovered on
ly by few people. Beside that, you may want to pay
attention to the size of the top teams. If leaderboard mostly co
nsists of
teams with only one participant, you'll probably have enough

chances if you gather a good team. Now, let's move to the next s
tep
after you chose a competition. As soon as you get familiar with
the data, start to write down your ideas about
what you may want to try later. What things could work here? Wha
t approaches you may want to take. After you're done, read forum
s and
highlight interesting posts and topics. Remember, you can get a
lot of information
and meet new people on forums. So I strongly encourage you to
participate in these discussions. After the initial pipeline is
ready and you roll down few ideas, you may want
to start improving your solution. Personally, I like to organize
these ideas into some structure. So you may want to sort
ideas into priority order. Most important and
promising needs to be implemented first. Or you may want to orga
nize
these ideas into topics. Ideas about feature generation,
validation, metric optimization. And so on. Now pick up an idea
and implement it. Try to derive some insights on the way. Especi
ally, try to understand why
something does or doesn't work. For example, you have an idea ab
out trying a deep
gradient boosting decision tree model. To your joy, it works. No
w, ask yourself why? Is there some hidden data
structure we didn't notice before? Maybe you have categorical fe
atures
with a lot of unique values. If this is the case, you as well ca
n make a conclusion that
mean encodings may work great here. So in some sense,
the ability to analyze the work and derive conclusions while
you're trying out your ideas will get you on the right track to
reveal hidden data patterns and leaks. After we checked out most
 important ideas, you may want to switch
to parameter training. I personally like the view,
everything is a parameter. From the number of features, through
gradient boosting decision through depth. From the number of lay
ers in
convolutional neural network, to the coefficient you finally
submit is multiplied by. To understand what I should tune and ch
ange first, I like to sort all
parameters by these principles. First, importance. Arrange param
eters from
important to not useful at all. Tune in this order. These may de
pend on data structure,
on target, on metric, and so on. Second, feasibility. Rate param
eters from, it is easy to tune,
to, tuning this can take forever. Third, understanding. Rate par
ameters from, I know what
it's doing, to, I have no idea. Here it is important to understa
nd
what each parameter will change in the whole pipeline. For examp
le, if you increase
the number of features significantly, you may want to change rat
io of columns
which is used to find the best split in gradient boosting decisi

on tree. Or, if you change number of layers
in convolution neural network, you will need more reports to tra
in it,
and so on. So let's see, these were some
of my practical guidelines, I hope they will prove useful for
you as well. Every problem starts with data loading and
preprocessing. I usually don't pay much attention to
some sub optimal usage of computational resources but this parti
cular
case is of crucial importance. Doing things right at the very be
ginning
will make your life much simpler and will allow you to save a lo
t of time and
computational resources. I usually start with basic data
preprocessing like labeling, coding, category recovery,
both enjoying additional data. Then, I dump resulting data into
HDF5 or
MPI format. HDF5 for Panda's dataframes,
and MPI for non bit arrays. Running experiment often require
a lot of kernel restarts, which leads to reloading all the data.
 And loading class CSC files
may take minutes while loading data from HDF5 or MPI formats
is performed in a matter of seconds. Another important matter is
 that by
default, Panda is known to store data in 64-bit arrays, which is
unnecessary in most of the situations. Downcasting everything to
 32 bits will
result in two-fold memory saving. Also keep in mind that Panda's
 support
out of the box data relink by chunks, via chunks ice parameter
in recess fee function. So most of the data sets may be
processed without a lot of memory. When it comes to performance
evaluation, I
am not a big fan of extensive validation. Even for medium-sized
datasets
like 50,000 or 100,000 rows. You can validate your models
with a simple train test split instead of full cross validation
loop. Switch to full CV only
when it is really needed. For example,
when you've already hit some limits and can move forward only wi
th
some marginal improvements. Same logic applies to
initial model choice. I usually start with LightGBM,
find some reasonably good parameters, and evaluate performance o
f my features. I want to emphasize that
I use early stopping, so I don't need to tune number
of boosting iterations. And god forbid start ESVMs,
random forks, or neural networks, you will waste too
much time just waiting for them to feed. I switch to tuning the
models,
and sampling, and staking, only when I am satisfied
with feature engineering. In some ways, I describe my approach a
s,
fast and dirty, always better. Try focusing on what is really im
portant,
the data. Do ED, try different features. Google domain-specific

knowledge. Your code is secondary. Creating unnecessary classes and personal frame box may only make
things harder to change and will result in wasting your time, so keep things simple and reasonable. Don't track every little change. By the end of competition, I usually
have only a couple of notebooks for model training and to want notebooks
specifically for EDA purposes. Finally, if you feel really uncomfortable
with given computational resources, don't struggle for weeks, just rent a larger server. Every competition I start with
a very simple basic solution that can be even primitive. The main purpose of such solution
is not to build a good model but to debug full pipeline from very beginning of the data to the very end when we write
the submit file into decided format. I advise you to start with construction of the initial pipeline. Often you can find it in baseline
solutions provided by organizers or in kernels. I encourage you to read carefully and
write your own. Also I advise you to follow from simple
to complex approach in other things. For example, I prefer to start
with Random Forest rather than Gradient Boosted Decision Trees. At least Random Forest
works quite fast and requires almost no tuning
of hybrid parameters. Participation in data science competition implies the analysis of data and generation of features and manipulations with models. This process is very similar in spirit
to the development of software and there are many good practices that I advise you to follow. I will name just a few of them. First of all, use good variable names. No matter how ingenious you are,
if your code is written badly, you will surely get confused in it and
you'll have a problem sooner or later. Second, keep your research reproducible. FIx all random seeds. Write down exactly how
a feature was generated, and store the code under version control system like git. Very often there are situation when you need to go back to the model that you built two weeks ago and edit to the ensemble width. The last and probably the most important thing, reuse your code. It's really important to use the same
code at training and testing stages. For example, features should be prepared
and transforming by the same code in order to guarantee that they're
produced in a consistent manner. Here in such places are very difficult to catch, so it's better to be very careful with it. I recommend to move reusable code into
separate functions or even separate model. In addition, I advise you to read
scientific articles on the topic of the competition. They can provide you with
information about machine and correlated things like for example

how
to better optimize a measure, or AUC. Or, provide the main
knowledge of the problem. This is often very useful for
future generations. For example, during Microsoft Mobile
competition, I read article about mobile detection and used idea
s from
them to generate new features. >> I usually start the competitio
n by
monitoring the forums and kernels. It happens that a competition
starts,
someone finds a bug in the data. And the competition data is
then completely changed, so I never join a competition
at its very beginning. I usually start a competition with
a quick EDA and a simple baseline. I tried to check the data for
various leakages. For me, the leaks are one of the most
interesting parts in the competition. I then usually do several
submissions
to check if validation score correlates with publicly the board
score. Usually, I try to come up with a list
of things to try in a competition, and I more or less try to fol
low it. But sometimes I just try to generate
as many features as possible, put them in extra boost and
study what helps and what does not. When tuning parameters,
I first try to make model overfit to the training set and only t
hen I
change parameters to constrain the model. I had situations when
I could not
reproduce one of my submissions. I accidentally changed somethin
g in
the code and I could not remember what exactly, so nowadays I'm
very
careful about my code and script. Another problem? Long executio
n history in notebooks leads
to lots of defined global variables. And global variables surely
lead to bugs. So remember to sometimes
restart your notebooks. It's okay to have ugly code, unless you
do not use this to produce a submission. It would be easier for
you to get into this code later if
it has a descriptive variable names. I always use git and
try to make the code for submissions as transparent as possible.
I usually create a separate notebook for every submission so I
can always run
the previous solution and compare. And I treat the submission
notebooks as script. I restart the kernel and
always run them from top to bottom. I found a convenient way to
validate
the models that allows to use validation code with minimal chang
es to retrain
a model on the whole dataset. In the competition, we are provide
d
with training and test CSV files. You see we load them in the fi
rst cell. In the second cell, we split
training set and actual training and validation sets, and
save those to disk as CSV files with the same structure as given
train CSV and
test CSV. Now, at the top of the notebook,

with my model, I define variables. Path is to train and test sets. I set them to create a training and validation sets when working with
the model and validating it. And then it only takes me to switch those paths to original train CSV and test CSV to produce a submission. I also use macros. At one point I was really tired of typing import numpy as np, every time. So I found that it's possible to define
a macro which will load everything for me. In my case, it takes only five symbols to type the macro name and this macro immediately loads me everything. Very convenient. And finally, I have developed my library
with frequently used functions, and training code for models. I personally find it useful, as the code,
it now becomes much shorter, and I do not need to remember how to import a particular model. In my case I just specify a model with its name, and as an output I get all the information
about training that I would possibly need. [SOUND] [MUSIC]Hello everyone. This is Marios. Today I would like to show you the Pipeline or like the approach I have used to tackle more than 100 machine learning competitions in cargo and obviously has helped me to do quite well. Before I start, let me state that I'm not claiming this is the best pipeline out there, is just the one I use. You might find some parts of it useful. So roughly, the Pipeline is, as you see it on the screen, here this is a summary and we will go through it in more detail later on. But briefly, I spend the first day in order to understand the problem and make the necessary preparations in order to deal with this. Then, maybe one, two days in order to understand a bit about the data, what are my features, what I have available, trying to understand other dynamics about the data, which will lead me to define a good cross validation strategy and we will see later why this is important. And then, once I have specified the cross validation strategy, I will spend all the days until 3-4 days before the end of the competition and I will keep iterating, doing different feature engineering and applying different machine bearing models. Now, something that I need to to highlight is that, when I start this process I do it myself, shut from the outside world. So, I close my ears, and I just focus on how I would tackle this problem. That's because I don't want to get affected by what the others are doing. Because I might be able to find something that others will not. I mean, I might take a completely different approach and this always leads me to gain, when I then combine with the rest of the people. For example, through merges or when I use other people's kernels. So, I think this is important, because it gives you the chance to create an intuitive approach about the data, and then also leverage the fact that other people have different approaches and you will get more diverse results. And in the last 3 to 4 days, I would start exploring different ways to combine all the models of field, in order to get the best results. Now, if people have seen me in competitions, you should know that you might have noticed that in the last 3-2 days I do a rapid jump in the little box and that's exactly because I leave assembling at the end. I normally don't do it. I have confidence that it will work and I spend more time in feature engineering a

nd modeling, up until this point. So, let's take all these steps one by one. Initially I try to understand the problem. First of all, what type of problem it is. Is it image classification, so try to find what object is presented on an image. This is sound classification, like which type of bird appears in a sound file. Is it text classification? Like who has written the specific text, or what this text is about. Is it an optimization problem, like giving some constraints how can I get from point A to point B etc. Is it a tabular dataset, so that's like data which you can represent in Excel for example, with rows and columns, with various types of features, categorical or numerical. Is it time series problem? Is time important? All these questions are very very important and that's why I look at the dataset and I try to understand, because it defines in many ways what resources I would need, where do I need to look and what kind of hardware and software I will need. Also, I do this sort of preparation along with controlling the volume of the data. How much is it. Because again, this will define how I need to, what preparations I need to do in order to solve this problem. Once I understand what type of problem it is, then I need to reserve hardware to solve this. So, in many cases I can escape without using GPUs, so just a few CPUs would do the trick. But in problems like image classification of sound, then generally anywhere you would need to use deep learning. You definitely need to invest a lot in CPU, RAM and disk space. So, that's why this screening is important. It will make me understand what type of machine I will need in order to solve this and whether I have this processing power at this point in order to solve this. Once this has been specified and I know how many CPUs, GPUs, RAM and disk space I'm going to need, then I need to prepare the software. So, different software is suited for different types of problems. Keras and TensorFlow is obviously really good for when solving an image classification or sound classification and text problems that you can pretty much use it in any other problem as well. Then you most probably if you use Python, you need scikit learn and XGBoost, Lighgbm. This is the pinnacle of machine learning right now. And how do I set this up? Normally I create either an anaconda environment or a virtual environment in general, and how a different one for its competition, because it's easy to set this up. So, you just set this up, you download the necessary packages you need, and then you're good to go. This is a good way, a clean way to keep everything tidy and to really know what you used and what you find useful in the particular competitions. It's also a good validation for later on, when we will have to do this again, to find an environment that has worked well for this type of problem and possibly reuse it. Another question I ask at this point is what the metric I'm being tested on. Again, is it a regression program, is it a classification program, it is root mean square error, it is mean absolute error. I ask these questions because I try to find if there's any similar competition with similar type of data that I may have dealt with in the past, because this will make this preparation much much better, because I'll go backwards, find what I had used in the past and capitalize on it. So, reuse it, improve it, or even if I don't have something myself, I can just find other similar competitions or explanations of these type of problem from the web and try to see what people had used in

order to integrate it to my approaches. So, this is what it means to understand the problem at this point. It's more about doing the screen search, this screening in order to understand what type of preparation I need to do, and actually do this preparation, in order to be able to solve this problem competitively, in terms of hardware, software and other resources, past resources in dealing with these types of problems. Then I spent the next one or two days to do some exploratory data analysis. The first thing that I do is I see all my features, assuming a tabular data set, in the training and the test data and to see how consistent they are. I tend to plot distributions and to try to find if there are any discrepancies. So is this variable in the training data set very different than the same variable in the task set? Because if there are discrepancies or differences, this is something I have to deal with. Maybe I need to remove these variables or scale them in a specific way. In any case, big discrepancies can cause problems to the model, so that's why I spend some time here and do some plotting in order to detect these differences. The other thing that I do is I tend to plot features versus the target variable and possibly versus time, if time is available. And again, this tells me to understand the effect of time, how important is time or date in this data set. And at the same time it helps me to understand which are like the most predictive inputs, the most predictive variables. This is important because it generally gives me intuition about the problem. How exactly this helps me is not always clear. Sometimes it may help me define a gross validation strategy or help me create some really good features but in general, this kind of knowledge really helps to understand the problem. I tend to create cross tabs for example with the categorical variables and the target variable and also creates unpredictability metrics like information value and you see chi square for example, in order to see what's useful and whether I can make hypothesis about the data, whether I understand the data and how they relate with the target variable. The more understanding I create at this point, most probably will lead to better features for better models applied on this data. Also while I do this, I like to bin numerical features into bands in order to understand if there nonlinear R.A.T's. When I say nonlinear R.A.T's, whether the value of a feature is low, target variable is high, then as the value increases the target variable decreases as well. So whether there are strange relationships trends, patterns, or correlations between features and the target variable, in order to see how best to handle this later on and get an intuition about which type of problems or which type of models would work better. Once I have understood the data, to some extent, then it's time for me to define a cross validation strategy. I think this is a really important step and there have been competitions where people were able to win just because they were able to find the best way to validate or to create a good cross validation strategy. And by cross validation strategy, I mean to create a validation approach that best resembles what you're being tested on. If you manage to create this internally then you can create many different models and create many different features and anything you do, you can have the confidence that is working or it's not working, if you've managed to build the cross validation strategy in a consistent way with what you're being t

ested on so consistency is the key word here. The first thing I ask is, "Is time important in this data?" So do I have a feature which is called date or time? If this is important then I need to switch to a time-based validation. Always have past data predicting future data, and even the intervals, they need to be similar with the test data. So if the test data is three months in the future, I need to build my training and validation to account for this time interval. So my validation data always need to be three months in the future and compared to the training data. You need to be consistent in order to have the most consistent results with what you are been tested on. The other thing that I ask is, "Are there different entities between the train and the test data?" Imagine if you have different customers in the training data and different in the test data. Ideally, you need to formulate your cross validation strategy so that in the validation data, you always have different customers running in training data otherwise you are not really testing in a fair way. Your validation method would not be consistent with the test data. Obviously, if you know a customer and you try to predict it, him or her, why you have that customer in your training data, this is a biased prediction when compared to the test data, that you don't have this information available. And this is the type of questions you need to ask yourself when you are at this point, "Am I making a validation which is really consistent with what am I being tested on?" The other thing that is often the case is that the training and the test data are completely random. I'm sorry, I just shortened my data and I took a random part, put it on training, the other for test so in that case, is any random type of cross validation could help for example, just do a random K-fold. There are cases where you may have to use a combination of all the above so you have strong temporal elements at the same time you have different entities, so different customers to predict for past and future and at the same time, there is a random element too. You might need to incorporate all of them do make a good strategy. What I do is I often start with a random validation and just see how it fares with the test leader board, and see how consistent the result is with what they have internally, and see if improvements in my validation lead to improvements to the leader board. If that doesn't happen, I make a deeper investigation and try to understand why. It may be that the time element is very strong and I need to take it into account or there are different entities between the train and test data. These kinds of questions in order to formulate a better validation strategy. Once the validation strategy has been defined, now I start creating many different features. I'm sorry for bombarding you with loads of information in one slide but I wanted this to be standalone. It says give you the different type of future engineering you can use in different types of problems, and also suggestions for the competition to look up which was quite representative of this time. But you can ignore these for now. Look at it later. The main point is different problem requires different feature engineering and I put everything when I say feature engineering. I put the day data cleaning and preparation as well, how you handle missing values, and the features you generate out of this. The thing is, every problem has its own corpus of different techniques you use to derive or create new features. It's not easy to k

now everything because sometimes it's too much, I don't remember it myself so what I tend to do is go back to similar competitions and see what people are using or what people have used in the past and I incorporate into my code. If I have dealt with this or a similar problem in the past then I look at my code to see what I had done in the past, but still looking for ways to improve this. I think that's the best way to be able to handle any problem. The good thing is that a lot of the feature engineering can be automated. You probably have already seen that but, as long as your cross validation strategy is consistent with the test data and reliable, then you can potentially try all sorts of transformations and see how they work in your validation environment. If they work well, you can be confident that this type of feature engineering is useful and use it for further modeling. If not, you discard and try something else. Also the combinations of what you can do in terms of feature engineering can be quite vast in different types of problems so obviously time is a factor here, and scalability too. You need to be able to use your resources well in order to be able to search as much as you can in order to get the best outcome. This is what I do. Normally if I have more time to do this feature engineering in a competition, I tend to do better because I explore more things.And the modeling is pretty much the same story. So, it's type problem has its own type of model that works best. Now, I don't want to go through that list again, I put it here so that you can use it for reference. But, again, the way you work this out is you look for literature, you sense other previous competitions that were similar and you try to find which type of problem, which type of model or best for its type of problem. And it's not surprise that for typical dataset, when I say typical dataset I mean, tabular dataset rather boosting machines in the form of [inaudible] turned to rock fest for problems like aim as classification sound classification, deep learning in the form of convolutional neural networks tend to work better. So, this is roughly what you need to know. New techniques are being developed so, I think your best chance here or what I have used in order to do well in the past was knowing what's tends to work well with its problem, and going backwards and trying to find other code or other implementations and similar problems in order to integrate it with mine and try to get a better result. I should mention that each of the previous models needs to be changed sometimes differently. So you need to spend time within this cross-validation strategy in order to find the best parameters, and then we move onto Ensembling. Every time you apply your cross-validation procedure with a different feature engineering and a different joint model, it's time, you saved two types of predictions, you save predictions for the validation data and you save predictions for the test data. So now that you have saved all these predictions and by the way this is the point that if you collaborate with others that tend to send you the predictions, and you'll be surprised that sometime that collaboration is just this. So people just sending these prediction files for the validation and the test data. So now you can find the best way to combine these models in order to get the best results. And since you already have predictions for the validation data, you know the target variable for the validation data, so you can explore different ways to combine them. The metho

ds could be simple, could be an average, or already average, or it can go up to a multilayer stacking in general. Generally, what you need to know is that from my experience, smaller data requires simple ensemble techniques like averaging. And also what tends to show is to look at correlation between predictions. So find it here that work well, but they tend to be quite diverse. So, when you use fusion correlation, the correlation is not very high. That means they are likely to bring new information, and so when you combine you get the most out of it. But if you have bigger data there are, you got pretty must try all sorts of things. What I like to think of is it is that, when you have really big data, the stacking process that impedes the modeling process. By that, I mean that you have a new set of features this time they are predictions of models, but you can apply the same process you have used before. So you can do feature engineering, you can create new features or you can remove the features/ prediction that you no longer need and you can use this in order to improve the results for your validation data. This process can be quite exhaustive, but well, again, it can be automated to some extent. So, the more time you have here, most probably the better you will do. But from my experience, 2, 3 days is good in order to get the best out of all the models you have built and depends obviously on the volume of data or volume of predictions you have generated up until this point. At this point I would like to share a few thoughts about collaboration. Many people have asked me this and I think this is a good point to share. These ideas has greatly helped me to do well in competitions. The first thing is that it makes things more fun. I mean you are not alone, you're with other people and that's always more energizing, it's always more interesting, it's more fun, you can communicate with the others through times like Skype, and yeah I think it's more collaborative as the world says, it is better. You learn more. I mean you can be really good, but, you know, you always always learn from others. No way to know everything yourself. So it's really good to be able to share points with other people, see what they do learn from them and become better and grow as a data scientist, as a model. From my experience you score far better than trying to solve a problem alone, and I think these happens for mainly for two ways. There are more but these are main two. First you can cover more ground because, you can say, you can focus on ensembling, I will focus on feature engineering or you will focus on joining this type of model and I will focus on another type of model. So, you can generally cover more ground. You can divide task and you can search, you can cover more ground in terms of the possible things you can try in a competition. The second thing is that every person sees the problem from different angles. So, that's very likely to generate more diverse predictions. So something we do is although we kind of define together by the different strategy when we form teams, then we would like to work for maybe one week separately without discussing with one another, because this helps to create diversity. Otherwise, if we over discuss this, we might generate pretty much the same things. So, in other words, our solutions might be too correlated to add more value. So, this is a good way in order to leverage the different mindset each person has in solving these problems. So, for one week, each one works separately and then after some point

, we start combining or work more closely. I would advise people to start collaborating after getting some experience, and I say here two or three competitions just because Cargo has some rules. Sometimes, it is easy to make mistakes. I think it's better to understand the environment, the competition environment well before exploring these options in order to make certain that, no mistakes are done, no violation of the rules. Sometimes new people tend to make these mistakes. So, it's good to have this experience prior to trying to collaborating. I advise people to start forming teams with people around their rank because sometimes it is frustrating when you join a high rank or a very experienced team I would say. It's bad to say experience from rank, because you don't know sometimes how to contribute, you still don't understand all the competition dynamics and it might stall your progress, if you join a team and you're not able to contribute. So, I think it's better to, in most cases, to try and find people around your rank or around your experience and grow together. This way is the best form of collaboration I think. Another tip for collaborating is to try to collaborate with people that are likely to take diverse approaches or different approaches than yourself. You learn more this way and it is more likely that when you combine, you will get a better score. So, such for people who are sort of famous for doing well certain things and in order to get the most out of it, to learn more from each other and get better results in the leader board. About selecting submissions, I have employed a strategy that many people have done. So normally, I select the best submissions I see in my internal result and the one that work best on the leader board. At the same time, I also look for correlations. So, if two submissions, they tend to be the same pretty much. So, the one that was the best submission locally, was also the best on leader boards, I try to find other submissions that still work well but they are likely to be quite diverse. So, they have low correlations with my best submission because this way, I might capture, I might be lucky, it maybe be a special type of test data set and just by having a diverse submission, I might be lucky to get a good score. So that's the main idea about this. Some tips I would like to share now in general about competitive modeling, on land modeling and in Cargo specifically. In these challenges, you never lose. [inaudible] lose, yes you may not win prize money. Out of 5000 people, sometimes it's difficult to be, almost to impossible to be in the top three or four that gives prizes but you always gain in terms of knowledge, in terms of experience. You get to collaborate with other people which are talented in the field, you get to add it to your CV that you try to solve this particular problem, and I can tell you there has been some criticists here, people doubt that doing these competitions stops your employ-ability but I can tell you that i know many examples and not want us, they really thought the Ocean Cargo like Master and Grand-master that just by having kind of experience, they have been able to find very decent jobs and even if they had completely diverse backgrounds to the science. So, I can tell you it matters. So, any time you spend here, it's definitely a win for you. I don't see how you can lose by competing in these challenges. You mean if this is something you like right. The whole predictive modeling that the science think. Coffee tempts to shop, because you tend to spend

longer hours. I tend to do this especially late at night. So it definitely tells me something to consider or to be honest any other beverage will do: depends what you like. I see it a bit like a game and I advise you to do the same because if you see it like a game, you never need to work for it. If you know what I mean. So it looks a bit like NRPT. In some way, you have some tools or weapons. These are all the algorithms and feature engineering techniques you can use. And then you have this core leader board and you try to beat all the bad guys and to beat the score and rise above them. So in a way does look like a game. You know you try to use all the tools, all the skills that you have to try to beat the score. So, I think if you see it like a game it really helps you. You don't get tired and you enjoy the process more. I do advise you to take a break though, from my experience you may spend long hours hitting on it and that's not good for your body. You definitely need to take some breaks and do some physical exercise. Go out for a walk. I think it can help most of the times by resting your mind this way can actually help to do better. You have more rested heart, more clear thinking. So, I definitely advise you to do this, generally don't overdo it. I have overnighted in the past but i advise you not to do the same. And now there is a thing that I would like to highlight is that the Cargo community is great. Is one of the most open and helpful helpful communities have experience in any social context, maybe apart from Charities but if you have a question and you posted on the forums or other associated channels like in Slug and people are always willing to help you.That's great, because there are so many people out there and most probably they know the answer or they can help you for a particular problem. And this is invaluable. So many times i have really made use of this, of this option and it really helps. You know this kind of mentality was there even before the serine was gamified. When I say gamified, now you get points by sharping in a way by sharing code or participating in discussions. But in the past, people were doing without really getting something out of it. It maybe the open source mentality of data science that the fact that many people participating are researchers. I don't know but it really is a field that sharing seems to be really important in helping others. So, I do advise you to consider this and don't be afraid to ask in these forums. Another thing that I do at shops, is that after the competition has ended irrespective of how well or not you've done, is go and look for other people and what they have done. Normally, there are threads where people share their approaches, sometimes they share the whole approach would go to sometimes it just give tips and you know this is where you can upgrade your tools and you can see what other people have done and make improvements. And in tandem with this, you should have a notebook of useful methods that you keep updating it at the end of every competition. So, you found an approach that was good, you just add it to that notebook and next time you encounter the same or similar competition you get that notebook out and you apply the same techniques at work in the past and this is how you get better. Actually, if i now start a competition without that notebook, i think it will take me three or four times more in order to get to the same score because a lot of the things that I do now depend on stuff that i have done in the past. So, it's definitely helpf

ul, consider creating this notebook or library of all the approa ches or approaches that have worked in the past in order to have an easier time going on. And that was what I wanted to share wi th you and thank you very much for bearing with me and to see yo u next time, right.Hi everyone. This video is dedicated to the f ollowing advanced feature engineering techniques. Calculating va rious statistics of one feature grouped by another and features derived from neighborhood analysis of a given point. To make it a little bit clearer, let's consider a simple example. Here we h ave a chunk of data for some CTR task. Let's forget about target variable and focus on human features. Namely, User_ID, unique i dentifier of a user, Page_ID, an identifier of a page user visit ed, Ad_price, item prices in the ad, and Ad_position, relative p osition of an ad on the web page. The most straightforward way t o solve this problem is to label and call the Ad_position and fe ed some classifier. It would be a very good classifier that coul d take into account all the hidden relations between variables. But no matter how good it is, it still treats all the data point s independently. And this is where we can apply feature engineer ing. We can imply that an ad with the lowest price on the page w ill catch most of the attention. The rest of the ads on the page won't be very attractive. It's pretty easy to calculate the fea tures relevant to such an implication. We can add lowest and hig hest prices for every user and page per ad. Position of an ad wi th the lowest price could also be of use in such case. Here's on e of the ways to implement statistical features with paid ads. I f our data is stored in the data frame df, we call groupby metho d like this to get maximum and minimum price values. Then store this object in gb variable, and then join it back to the data fr ame df. This is it. I want to emphasize that you should not stop at this point. It's possible to add other useful features not n ecessarily calculated within user and page per. It could be how many pages user has visited, how many pages user has visited dur ing the given session, and ID of the most visited page, how many users have visited that page, and many, many more features. The main idea is to introduce new information. By that means, we ca n drastically increase the quality of the models. But what if th ere is no features to use groupby on? Well, in such case, we can replace grouping operations with finding the nearest neighbors. On the one hand, it's much harder to implement and collect usef ul information. On the other hand, the method is more flexible. We can fine tune things like the size of relevant neighborhood o r metric. The most common and natural example of neighborhood an alysis arises from purposive pricing. Imagine that you need to p redict rental prices. You would probably have some characteristi cs like floor space, number of rooms, presence of a bus stop. Bu t you need something more than that to create a really good mode l. It could be the number of other houses in different neighborh oods like in 500 meters, 1,000 meters, or 1,500 meters, or avera ge price per square meter in such neighborhoods, or the number o f schools, supermarkets, and parking lots in such neighborhoods. The distances to the closest objects of interest like subway st ations or gyms could also be of use. I think you've got the idea . In the example, we've used a very simple case, where neighborh oods were calculated in geographical space. But don't be afraid to apply this method to some abstract or even anonymized feature

space. It still could be very useful. My team and I used this method in Spring Leaf competition. Furthermore, we did it in supervised fashion. Here is how we have done it. First of all, we applied mean encoding to all variables. By doing so, we created homogeneous feature space so we did not worry about scaling and importance of each particular feature. After that, we calculated 2,000 nearest neighbors with Bray-Curtis metric. Then we evaluated various features from those neighbors like mean target of nearest 5, 10, 15, 500, 2,000 neighbors, mean distance to 10 closest neighbors, mean distance to 10 closest neighbors with target 1, and mean distance to 10 closest neighbors with target 0, and, it worked great. In conclusion, I hope you embrace the main ideas of both groupby and nearest neighbor methods and you would be able to apply them in practice. Thank you for your attention.[MUSIC] Hi everyone, in this video I will
talk about the application of matrix factorization technique
in feature extraction. You will see a few application
of the approach for feature extraction and
we will be able to apply it. I will show you several examples
along with practical details. Here's a classic example
of recommendations. Suppose we have some information about
user, like age, region, interest and items like gender, year length. Also we know ratings that
users gave to some items. These ratings can be organized in a user
item matrix with row corresponding to users, and columns corresponding to items,
as shown in the picture. In a cell with coordinates i, j,
the user or agent can be chooser i, give the item j. Assume that our user
have some features Ui. And jth item have is
corresponding feature Mj. And scalar product of these
features produce a rating Rij. Now we can apply matrix factorization
to learning those features for item and users. Sometimes these features
can have an interpretation. Like the first feature in item can
be measured of or something similar. But generally you should consider them
as some extra features, which we can use to encode user in the same way as we did
before with labeling coder or coder. Specifically our assumption about
scale of product is the following. If we present all attributes
of user and items as matrixes, the matrix product will
be very close to the matrix's ratings. In other words,
which way to find matrix's U and M, such as their product
gives the matrix R. This way, this approach is called matrix
factorization or matrix composition. In previous examples, we used both row and
column related features. But sometimes we don't let
the features correspond to rows. Let's consider another example. Suppose that we are texts, do you
remember how we usually classify text? We extract features and each document
was described by a large sparse reactor. If we do matrix factori

zation over
these parse features, we will get the representation for index d
isplayed
in yellow, and terms displayed in green. Although we can somehow
use representation for jumps, we are interested only
in representations for dogs. Now every document is described
by a small, dense reactor. These are our features, and we can us
e
them in a way similar to previous example. This case is often ca
lled
dimension energy reduction. It's quite an efficient way to reduc
e
the size of feature metric, and extract real valued features
from categorical ones. In competitions we often have
different options for purchasing. For example, using text data,
you can run back of big rams and so on. Using matrix optimizatio
n technique, you are able to extract features
from all of these matrices. Since the resulting matrices will be
small, we can easily join them and use togetherness of the featu
res
in tree-based models. Now I want to make a few comments
about matrix factorization. Not just that we are not
constrained to reduce whole matrix, you can apply factorization
to a subset
of a column and leave the other as is. Besides reduction you can
use pressure boards for getting another presentation
of the same data. This is especially useful for example since it
provides velocity
of its models and leads to a better. Of course matrix factorizat
ion
is a loss of transformation, in other words we will lose some
information after the search reduction. Efficiency of this appro
ach heavily
depends on a particular task and choose a number of latent facto
rs. The number should be considered as
a hyper parameter and needs to be tuned. It's a good practice to
choose
a number of factors between 5 and 100. Now, let's switch from ge
neral idea
to particular implementations. Several matrix factorization
methods are implemented in circuit as the most famous SVD and PC
A. In addition,
their use included TruncatedSVD, which can work with sparse matr
ices. It's very convenient for example,
in case of text datasets. Also there exists a so called non-nega
tive
matrix factorization, or NMF. It impose an additional restrictio
ns that
all hidden factors are non-negative, that is either zero or a po
sitive number. It can be applied only to
non-negative matrixes. For example matrix where all represented
occurence of each word in the document. NMF has an interesting p
roperty, it transforms data in a way that makes
data more suitable for decision trees. Take a look at the pictur
e from
Microsoft Mobile Classification Challenge. It can be seen that N

MF transform data
forms lines parallel to the axis. A few more notes on matrix fac
torizations. Essentially they are very
similar to linear models, so we can use the same transformation
tricks as we use for linear models. So in addition to standard N
MF, I advise you to apply
the factorization to transform data. Here's another plot from th
e competition. It's clear that these two transformations
produce different features, and we don't have to choose the best
one. Instead, it's beneficial
to use both of them. I want to note that matrix factorization
is a trainable transformation, and has its own parameters. So we
should be careful, and
use the same transformation for all parts of your data set. Read
ing and transforming each
part individually is wrong, because in that case you will get
two different transformations. This can lead to an error
which will be hard to find. The correct method is shown below,
first we need to the data information on all data and
only then apply to each individual piece. To sum up, matrix comp
osition is a very
general approach to dimensional reduction and feature extraction
. It can be used to transform
categorical feature into real ones. And tricks for linear models
are also
suitable for matrix factorizations. Thank you for your attention
. [MUSIC] [SOUND]Hi, everyone. The main topic of this video is F
eature Interactions. You will learn how to construct them and us
e in problem solving. Additionally, we will discuss them for fea
ture extraction from decision trees. Let's start with an example
. Suppose that we are building a model to predict the best adver
tisement banner to display on a website. Among available feature
s, there are two categorical ones that we will concentrate on. T
he category of the advertising banner itself and the category of
the site the banner will be showing on. Certainly, we can use t
he features as two independent ones, but a really important feat
ure is indeed the combination of them. We can explicitly constru
ct the combination in order to incorporate our knowledge into a
model. Let's construct new feature named ad_site that represents
the combination. It will be categorical as the old ones, but se
t of its values will be all possible combinations of two origina
l values. From a technical point of view, there are two ways to
construct such interaction. Let's look at a simple example. Cons
ider our first feature, f1, has values A or B. Another feature,
f2, has values X or Y or Z, and our data set consist of four dat
a points. The first approach is to concatenate the text values o
f f1 and f2, and use the result as a new categorical feature f_j
oin. We can then apply the OneHot according to it. The second ap
proach consist of two steps. Firstly, apply OneHot and connect t
o features f1 and f2. Secondly, construct new metrics by multipl
ying each column from f1 encoded metrics to each column from f2
encoded metrics. It was nothing that both methods results in pra
ctically the same new feature representations. In the above exam
ple, we can consider as interactions between categorical feature
s, but similar ideas can be applied to real valued features. For
example, having two real valued features f1 and f2, interaction

s between them can be obtained by multiplications of f1 and f2. In fact, we are not limited to use only multiply operation. Any function taking two arguments like sum, difference, or division is okay. The following transformations significantly enlarge fea ture space and makes learning easier, but keep in mind that it m akes or frequent easier too. It should be emphasized that for th ree ways algorithms such as the random forest or gradient boost decision trees it's difficult to extract such kind of dependenci es. That's why they're buffer transformation are very efficient for three based methods. Let's discuss practical details now. Wh ere wise future generation approaches greatly increase the numbe r of the features. If there were any original features, there wi ll be n square. And will be even more features if several types of interaction are used. There are two ways to moderate this, ei ther do feature selection or dimensionality reduction. I prefer doing the selection since not all but only a few interactions of ten achieve the same quality as all combinations of features. Fo r each type of interaction, I construct all piecewise feature in teractions. Feature random forests over them and select several most important features. Because number of resulting features fo r each type is relatively small. It's possible to join them toge ther along with original features and use as input for any machi ne learning algorithm usually to be by use method. During the vi deo, we have examined the method to construct second order inter actions. But you can similarly produce throned order or higher. Due to the fact that number of features grow rapidly with order, it has become difficult to work with them. Therefore high order directions are often constructed semi-manually. And this is an art in some ways. Additionally, I would like to talk about metho ds to construct categorical features from decision trees. Take a look at the decision tree. Let's map each leaf into a binary fe ature. The index of the object's leaf can be used as a value for a new categorical feature. If we use not a single tree but an e nsemble of them. For example, a random forest, then such operati on can be applied to each of entries. This is a powerful way to extract high order interactions. This technique is quite simple to implement. Tree-based poodles from sklearn library have an ap ply method which takes as input feature metrics and rituals corr esponding indices of leaves. In xgboost, also support to why a p arameter breed leaf in predict method. I suggest we need to coll aborate documentations in order to get more information about th ese methods and IPIs. In the end of this video, I will tackle th e main points. We examined method to construct an interactions o f categorical features. Also, we extend the approach to real-val ued features. And we have learned how to use trees to extract hi gh order interactions. Thank you for your attention.Hi, everyone . Today, we will discuss this new method for visualizing data in tegrating features. At the end of this video, you will be able t o use tSNE in your products. In the previous video, we learned a bout metaphysician technique that is predatory very close to lin ear models. In this video, we will touch the subject of non-line ar methods of dimensionality reduction. That says in general are called manifold learning. For example, look at the data in form of letter S on the left side. On the right, we can see results of running different manifold learning algorithm on the data. Th is new result is placed at the right bottom corner on the slide.

This new algorithm is the main topic of the lecture, as it tells of how this really works won't be explained here. But you will come to look at additional materials for the details. Let's just say that this is a method that tries to project points from high dimensional space into small dimensional space so that the distances between points are approximately preserved. Let's look at the example of the tSNE on the MNIST dataset. Here are points from 700 dimensional space that are projected into two dimensional space. You can see that such projection forms explicit clusters. Coolest shows that these clusters are meaningful and corresponds to the target numbers well. Moreover, neighbor clusters corresponds to a visually similar numbers. For example, cluster of three is located next to the cluster of five which in chance is adjustment to the cluster of six and eight. If data has explicit structure as in case of MNIST dataset, it's likely to be reflected on tSNE plot. For the reason tSNE is widely used in exploratory data analysis. However, do not assume that tSNE is a magic want that always helps. For example, a misfortune choice of hyper parameters may lead to poor results. Consider an example, in the center is the least presented a tSNE projection of exactly the same MNIST data as in previous example, only perplexity parameter has been changed. On the left, for comparison, we have plots from previous right. On the right, so it present a tSNE projection of random data. We can see as a choice of hybrid parameters change projection of MNIST data significantly so that we cannot see clusters. Moreover, new projection become more similar to random data rather than to the original. Let's find out what depends on the perplexity hyperparameter value. On the left, we have perplexity=3, in the center=10, and on the right= 150. I want to emphasize that these projections are all made for the same data. The illustration shows that these new results strongly depends on its parameters, and the interpretation of the results is not a simple task. In particular, one cannot infer the size of original clusters using the size of projected clusters. Similar proposition is valid for a distance between clusters. Blog distill.pub contain a post about how to understand and interpret the results of tSNE. Also, it contains a great interactive demo that will help you to get into issues of how tSNE works. I strongly advise you to take a look at it. In addition to exploratory data analysis, tSNE can be considered as a method to obtain new features from data. You should just concatenate the transformers coordinates to the original feature matrix. Now if you've heard this about practical details, as it has been shown earlier, the results of tSNE algorithm, it strongly depends on hyperparameters. It is good practice to use several projections with different perplexities. In addition, because of stochastic of this methods results in different projections even with the same data and hyperparameters. This means the train and test sets should be projected together rather than separately. Also, tSNE will run for a long time if you have a lot of features. If the number of features is greater than 500, you should use one of dimensionality reduction approach and reduce number of features, for example, to 100. Implementation of tSNE can be found in the sklearn library. But personally, I prefer to use another implementation from a separate Python package called tSNE, since it provide a way more efficient implementation. In conclusion, I want to remind you the basic

points of the lecture. TSNE is an excellent tool for visualizing data. If data has an explicit structure, then it likely be [inaudible] on tSNE projection. However, it requires to be cautious with interpretation of tSNE results. Sometimes you can see structure where it does not exist or vice versa, see none where structure is actually present. It's a good practice to do several tSNE projections with different perplexities. And in addition to EJ, tSNE is working very well as a feature for feeding models. Thank you for your attention.Hello everyone, this is Marios Michailidis, and this will be the first video in a series that we will be discussing on ensemble methods for machine learning. To tell you a bit about me, I work as Research Data Scientist for H2Oai. In fact, my PhD is about assemble methods, and they used to be ranked number one in cargo and ensemble methods have greatly helped me to achieve this spot. So you might find the course interesting. So what is ensemble modelling? I think with this term, we refer to combining many different machine learning models in order to get a more powerful prediction. And later on we will see examples that this happens, that we combine different models and we do get better predictions. There are various ensemble methods. Here we'll discuss a few, those that we encounter quite often, in predictive modelling competitions, and they tend to be, in general, quite competitive. We will start with simple averaging methods, then we'll go to weighted averaging methods, and we will also examine conditional averaging. And then we will move to some more typical ones like bagging, or the very, very popular, boosting, then stacking and StackNet, which is the result of my research. But as I said, these will be a series of videos, and we will initially start with the averaging methods. So, in order to help you understand a bit more about the averaging methods, let's take an example. Let's say we have a variable called age, as in age years, and we try to predict this. We have a model that yields prediction for age. Let's assume that the relationship between the two, the actual age in our prediction, looks like in the graph, as in the graph. So you can see that the model boasts quite a higher square of a value of 0.91, but it doesn't do so well in the whole range of values. So when age is less than 50, the model actually does quite well. But when age is more than 50, you can see that the average error is higher. Now let's take another example. Let's assume we have a second model that also tries to predict age, but this one looks like that. As you can see, this model does quite well when age is higher than 50, but not so well when age is less than 50,

nevertheless, it scores again 0.91. So we have two models that h
ave
a similar predictive power, but they look quite different. It's
quite obvious that they do
better in different parts of the distribution of age. So what wi
ll happen if we
were to try to combine this two with a simple averaging method,
in other words, just say (model 1 + model two) / 2,
so a simple averaging method. The end result will look
as in the new graph. So, our square has moved to 0.95,
which is a considerable improvement versus the 0.91 we had befor
e,
and as you can see, on average, the points tend to
be closer with the reality. So the average error is smaller. How
ever, as you can see, the model doesn't
do better as an individual models for the areas where the models
were doing really well, nevertheless, it does better on average.
 This is something we need to understand, that there is potentia
lly a better
way to combine these models. We could try to take a weighting av
erage. So say, I'm going to take 70% of
the first model prediction and 30% of the second model predictio
n. In other words,
(model 1x0.7 + model 2x0.3), and the end result would
look as in the graph. So you can see their square is no better
and that makes sense, because the models have quite similar pred
ictive power and
it doesn't make sense to rely more in one. And also it is quite
clear that
it looks more with model 1, because it has better predictions
when age is less than 50, and worse predictions
when age is more than 50. As a theoretical exercise, what is the
theoretical best we could get out of this? We know we have a mod
el that scores
really well when age is less than 50, and another model that sco
res really
well when age is more than 50. So ideally, we would like to
get to something like that. This is how we leverage the two
models in the best possible way here by using a simple
conditioning method. So if less than 50 is one I'll just
use the other, and we will see later on that there are ensemble
methods
that are very good at finding these relationships of two or more
 predictions
in respect to the target variable. But, this will be a topic for
another discussion. Here we discuss simple averaging methods, ho
pefully you found it useful, and
stay here for the next session to come. Thank you very much.Hell
o everyone. This is Marios Michailidis and we will continue our
discussion in regards to ensemble methods. Previously, we saw so
me simple averaging methods. This time, we'll discuss about bagg
ing, which is a very popular and efficient form of ensembling. W
hat is bagging? bagging refers to averaging slightly different v
ersions of the same model as a means to improve the predictive p
ower. A common and quite successful application of bagging is th
e Random Forest. Where you would run many different versions of

decision trees in order to get a better prediction. Why should w e consider bagging? Generally, in the modeling process, there ar e two main sources of error. There are errors due to bias often referred to as underfitting, and errors due to variance often re ferred to as overfitting. In order to better understand this, I' ll give you two opposite examples. One with high bias and low va riance and vice versa in order to understand the concept better. Let's take an example of high bias and low variance. We have a person who is let's say young, less than 30 years old and we kno w this person is quite rich and we're trying to find him, this p erson who'll buy a racing or an expensive car. Our model has hig h variance, has high bias if it says that this person is young a nd I think he's not going to buy an expensive car. What the mode l has done here is that it hasn't explore very deep relationship within the data. It doesn't matter that this person is young if it has a lots of money when it comes to buying a car. It hasn't explored different relationships. In other words, it has been u nderfitted. However, this is also associated with low variance b ecause this relationship, the fact that a young person generally doesn't buy an expensive car is generally true so we would expe ct this information to generalize well enough in a foreseen data . Therefore, the variance is low in this example. Now, let's try to see the other way around, an example with high variance and low bias. Let's assume we have a person. His name is John. He li ves in a green house, has brown eyes, and we want to see he will buy a car. A model that has gone so deep in order to find these relationships actually has a low bias because it has really exp lored a lots of information about the training data. However, it is making the mistake that every person that has these characte ristics is going to buy a car. Therefore, it generalizes for som ething that it shouldn't. In other words, it has already exhaust ed the information in the training data and the results are not significant. So, here, we actually have high variance but low bi as. If we were to visualize the relationship between prediction error and model complexity, it would look like that. When we beg in the training of the model, we can see that the training error make the error in that training data gets reduced and the same happens in the test data because the predictions are easily gene ralizable. They are simple. However, after a point, any improvem ents in the training error are not realized into test data. This is the point where the model starts over exhausting information , creates predictions that are not generalizable. This is where bagging actually comes into play and offers it's utmost value. B y making slightly different or let say randomized models, we ens ure that the predictions do not read very high variance. They're generally more generalizable. We don't over exhaust the informa tion in the training data. At the same time, we saw before that when you average slightly different models, we are generally abl e to get better predictions and we can assume that in 10 models, we are still able to find quite significant information about t he training data. Therefore, this is why bagging tends to work q uite well and personally, I always use bagging. When I say, "I f it a model," I have actually not fit a model I have fit a baggin g version of this model so probably that different models. Which parameters are associated with bagging? The first is the seed. We can understand that many algorithms have some randomized proc

edures so by changing the seed you ensure that they are made sli ghtly differently. At the same time, you can run a model with le ss rows or you could use bootstrapping. Bootstrapping is differe nt from row sub-sampling in the sense that you create an artific ial dataset so you might let's say data row the training data th ree or four times. You create a random dataset from the training data. A different form of randomness can be imputed with shuffl ing. There are some algorithms, which are sensitive to the order of the data. By changing the order you ensure that the models b ecome quite different. Another way is to dating a random sample of columns so bid models on different features or different vari ables of the data. Then you have model-specific parameters. For example, in a linear model, you will try to build 10 different l et's say logistic regression with slightly different regularizat ion parameters. Obviously, you could also control the number of models you include in your ensemble or in this case we call them bags. Normally, we put a value more than 10 here but, in princi ple, the more bags you put, it doesn't hurt you. It makes result s better but after some point, performance start plateauing. So there is a cost benefit with time but, in principle, more bags i s generally better and optionally, you can also apply parallelis m. Bagging models are independent to each other, which means you can build many of them at the same time and make full use of yo ur computation power. Now, we can see an example about bagging b ut before I do that, just to let you know that a bagging estimat ors that scikit-learn has in Python are actually quite cool. The refore, I recommend them. This is a typical 15 lines of code tha t I use quite often. They seem really simple but they're actuall y quite efficient. Assuming you have a training at the test data set and to target variable, what you do is you specify some bagg ing parameters. What is the model I'm going to use at random for est? How many bags I'm going to run? 10. What will be my seed? O ne. Then you create an object, an empty object that will save th e predictions and then you run a loop for as many bags as you ha ve specified. In this loop, you repeat the same. You change the seed, you feed the model, you make predictions in the test data and you save these predictions and then, you just take an averag e of these predictions. This is the end of the session. In this session, we discussed bagging as a popular form of ensembling. W e saw bagging in association with variants and bias and we also saw in the example about how to use it. Thank you very much. The next session we will describe boosting, which is also very popu lar so stay in tune and have a good day.Hello, everyone. This is Marios Michailidis. And today, we'll continue our discussion wi th ensemble methods, and specifically, with a very popular form of ensembling called boosting. What is boosting? Boosting is a f orm of weighted averaging of models where each model is built se quentially in a way that it takes into account previous model pe rformance. In order to understand this better, remember that bef ore, we discussed about biking, and we saw that we can have it a t many different models, which are independent to each other in order to get a better prediction. Boosting does something differ ent. It says, now I tried to make a model, but I take into accou nt how well the previous models have done in order to make a bet ter prediction. So, every model we add sequentially to the ensem ble, it takes into account how well the previous models have don

e in order to make better predictions. There are two main boosti ng type of algorithms. One is based on weight, and the other is based on residual error, and we will discuss both of them one by one. For weight boosting, it's better to see an example in orde r to understand it better. Let's say we have a tabular data set, with four features. Let's call them x0, x1, x2, and x3, and we want to use these features to predict a target variable, y. What we are going to do in weight boosting is, we are going to fit a model, and we will generate predictions. Let's call them pred. These predictions have a certain margin of error. We can calcula te these absolute error, and when I say absolute error, is absol ute of y minus our prediction. You can see there are predictions which are very, very far off, like row number five, but there a re others like number six, which the model has actually done qui te well. So what we do based on this is we generate, let's say, a new column or a new vector, where we create a weight column, a nd we say that this weight is 1 plus the absolute error. There a re different ways to calculate this weight. Now, I'm just giving you this as an example. You can infer that there are different ways to do this, but the overall principle is very similar. So w hat you're going to do next is, you're going to fit a new model using the same features and the same target variable, but you're going to also add this weight. What weight says to the model is , I want you to put more significance into a certain role. You c an almost interpret weight has the number of times that a certai n row appears in my data. So let's say weight was 2, this means that this row appears twice, and therefore, has bigger contribut ion to the total error. You can keep repeating this process. You can, again, calculate a new error based on this error. You calc ulate new weights, and this is how you sequentially add models t o the ensemble that take into account how well each model has do ne in certain cases, maximizing the focus from where the previou s models have done more wrong. There are certain parameters asso ciated with this type of boosting. One is the learning rate. We can also call it shrinkage or eta. It has different names. Now, if you recall, I explained boosting as a form of weighted averag ing. And this is true, because normally what this learning rate. So what we say is, every new model we built, we shouldn't trust it 100%. We should trust it a little bit. This ensures that we don't have one model generally having too much contribution, and completely making something that is not very generalizable. So this ensures that we don't over-trust one model, we trust many m odels a little bit. It is very good to control over fitting. The second parameter we look at is the number of estimators. This i s quite important. And normally, there is an inverse relationshi p, an opposite relationship, with the learning rate. So the more estimators we add to these type of ensemble, the smaller learni ng rate we need to put. This is sometimes quite difficult to fin d the right values, and we do it with the help of cross-validati on. So normally, we start with a fixed number of estimators, let 's say, 100, and then, we try to find the optimal learning rate for this 100 estimators. Let's say, based on cross-validation pe rformance, we find this to be 0.1. What we can do then is, let's say, we can double the number of estimators, make it 200, and d ivide the learning rate by 2, so we can put 0.05, and then we ta ke performance. It may be that the relationship is not as linear

as I explained, and the best learning rate may be 0.04 or 0.06 after duplicating the estimators, but this is roughly the logic. This is how we work in order to increase estimators, and try to see more estimators give us better performance without losing so much time, every time, trying to find the best learning rate. Another thing we look at is the type of input model. And generally, we can perform boosting with any type of estimator. The only condition is that it needs to accept weight in it's modeling process. So I weigh to say how much we should rely in each role of our data set. And then, we have various boosting types. As I said, I roughly explained to you how we can use the weight as a means to focus on different rows, different cases the model has done wrong, but there are different ways to express this. For example, there are certain boosting algorithm that do not care about the margin of error, they only care if you did the classification correct or not. So there are different variations. One I really like is the AdaBoost, and there is a very good implementation in sklearn, where you can choose any input algorithm. I think it's really good. And another one I really like is, normally, it's only good for logistic regression, and there is a very good implementation in Weka for Java if you want to try. Now, let's move onto the our time of boosting, which has been the most successful. I believe that in any predictive modeling competition that was not image classification or predicting videos. This has been the most dominant type of algorithm that actually has one most in these challenges so this type of boosting has been extremely successful, but what is it? I'll try to give you again a similar example in order to understand the concept. Let's say we have again the same dataset, same features, again when trying to predict a y variable, we fit a model, we make predictions. What we do next, is we'll calculate the error of these predictions but this time, not in absolute terms because we're interested about the direction of the error. What we do next is we take this error and we make it adding new y variable so the error now becomes the new target variable and we use the same features in order to predict this error. It's an interesting concept and if we wanted, let's say to make predictions for Rownum equals one, what we would do is we will take our initial prediction and then we'll add the new prediction, which is based on the error of the first prediction. So initially, we have 0.75 and then we predicted 0.2. In order to make a final prediction, we would say one plus the other equals 0.95. If you recall, the target for this row, it was one. Using two models, we were able to get closer to the actual answer. This form of boosting works really, really well to minimize the error. There are certain parameters again which are associated with this type of boosting. The first is again the learning rate and it works pretty much as I explained it before. What you need to take into account is how this is applied. Let's say we have a learning rate of 0.1. In the previous example, where the prediction was 0.2 for the second model, what you will say is I want to move my prediction towards that direction only 10 percent. If you remember the prediction was 0.2, 10 percent of this is 0.02. This is how much we would move towards the prediction of the error. This is a good way to control over fitting. Again, we ensure we don't over rely in one model. Again, how many estimators you put is quite important. Normally, more is better but

you need to offset this with the right learning rate. You need t o make certain that every model has the right contribution. If y ou intent to put many, then you need to make sure that your mode ls have very, very small contribution. Again, you decide these p arameters based on cross-validation and the logic is very simila r as explained before. Other things that work really well is tak ing a subset of rows or a subset of columns when you build its m odel. Actually, there is no reason why we wouldn't use this with the previous algorithm. The way its based, it is more common wi th this type of boosting, and internally works quite well. For i nput model, I have seen that this method works really well with this increase but theoretically, you can put anything you want. Again, there are various boosting types. I think the two most co mmon or more successful right now in a predictive modeling conte xt is the gradient based, which is actually what I explained wit h you how the prediction and you don't move 100 percent with tha t direction if you apply the learning rate. The other very inter esting one, which I've actually find it very efficient especiall y in classification problems is the dart. Dart, it imposes a dro p out mechanism in order to control the contribution of the tree s. This is a concept derived from deep learning where you say, " Every time I make a new prediction in my sample, every time I ad d a new estimate or I'm not relying on all previous estimators b ut only on a subset of them." Just to give you an example, let's say we have a drop out rate of 20 percent. So far, we have buil t 10 trees, we want to or 10 models and then we try to see, we t ry to build a new, an 11th one. What we'll do is we will randoml y exclude two trees when we generate a prediction for that 11th tree or that 11th model. By randomly excluding some models, by i ntroducing this kind of randomness, it works as a form of regula rization. Therefore, it helps a lot to make a model that general izes quite well enough for same data. This concept tends to work quite well because this type of boosting algorithm has been so successful. There have been many implementations to try to impro ve on different parts of these algorithms. One really successful application especially in the comparative predictive modeling w orld is the Xgboost. It is very scalable and it supports many lo ss functions. At the same time, is available in all major progra mming languages for data science. Another good implementation is Lightgbm. As the name connotes, it is lightning fast. Also, it is supported by many programming languages and supports many los s functions. Another interesting case is the Gradient Boosting M achine from H2O. What's really interesting about this implementa tion is that it can handle categorical variables out of the box and it also comes with a real set of parameters where you can co ntrol the modeling process quite thoroughly. Another interesting case, which is also fairly new is the Catboost. What's really g ood about this is that it comes with the strong initial set of p arameters. Therefore, you don't need to spend so much time tunin g. As I mentioned before, this can be quite a time consuming pro cess. It can also handle categorical variables out of the box. U ltimately, I really like the Gradient Boosting Machine implement ation of Scikit-learn. What I really like about this is that you can put any scikit-learn estimator as a base. This is the end o f this video. In the next session, we will discuss docking, whic h is also very popular, so stay tuned.Continuing our discussion

with ensemble
methods, next one up is stacking. Stacking is a very, very popular form of ensembling using
predictive modeling competitions. And I believe in most competitions,
there is a form of stacking in the end in order to boost
your performance as best as you can. Going through the definition of stacking,
it essentially means making several predictions
with hold-out data sets. And then collecting or
stacking these predictions to form a new data set,
where you can fit a new model on it, on this newly-formed data set from predictions. I would like to take you through
a very simple, I would say naive, example to show you how,
conceptually, this can work. I mean, we have so far seen that you
can use previous models' predictions to affect a new model, but
always in relation with the input data. This is a new concept because we're
only going to use the predictions of some models in order
to make a better model. So let's see how these could
work in a real life scenario. Let's assume we have three kids,
let's name them LR, SVM, KNN, and
they argue about a physics question. So each one believes the answer to
a physics question is different. First one says 13, second 18, third 11, they don't know how to
solve this disagreement. They do the honorable thing,
they say let's take an average, which in this case is 14. So you can almost see the kids, there's different models here,
they take input data. In this case,
it's the question about physics. They process it based on
historical information and and they are able to output an estimate,
a prediction. Have they done it optimally, though? Another way to say this is
to say there was a teacher, Miss DL, who had seen this discussion,
and she decided to step up. While she didn't hear the question,
she does know the students quite well, she knows the strength and
weaknesses of each one. She knows how well they have done
historically in physics questions. And from the range of values they have
provided, she is able to give an estimate. Let's say that in this concept, she knows
that SVM is really good in physics, and her father works in the department
of Physics of Excellence. And therefore she should have
a bigger contribution to this ensemble than every other kid,
therefore the answer is 17. And this is how a meta model works,
it doesn't need to know the input data. It just knows how the models
have done historically, in order to find the best
way to combine them. And this can work quite well in practice. So, let's go more into

the methodology of stacking. Wolpert introduced stacking in 1992
, as a meta modeling technique
to combine different models. It consists of several steps. The f
irst step is,
let's assume we have a train data set, let's divide it into two
parts; so
a training and the validation. Then you take the training part,
and you train several models. And then you make predictions for
the second part, let's say the validation data set. Then you col
lect all these predictions,
or you stack these predictions. You form a new data set and
you use this as inputs to a new model. Normally we call this a m
eta model,
and the models we run into, we call them base model or base lear
ners. If you're still confused about stacking,
consider the following animation. So let's assume we have three
data sets A,
B, and C. In this case, A will serve
the role of the training data set, B will be the validation data
 set, and C will be the test data sets where we
want to make the final predictions. They all have similar archit
ectural,
four features, and one target variable we try to predict. So in
this case,
we can choose an algorithm to train a model based on data set 1,
 and then we make predictions for
B and C at the same time. Now we take these predictions, and
we put them into a new data set. So we create a data set to stor
e the
predictions for the validation data in B1. And a data set called
 C1 to save
predictions for the test data, called C1. Then we're going to re
peat the process, now we're going to choose
another algorithm. Again, we will fit it on A data set. We will
make predictions on B and
C at the same time, and we will save these predictions
into the newly-formed data sets. And we essentially append them,
we stack them next to each other, this is where stacking takes i
ts name. And we can continue this even more,
do it with a third algorithm. Again the same, fit on A,
predict on B and C, same predictions. What we do then is we take
 the target
variable for the B data set, or the validation datadset,
which we already knew. And we are going to fit a new model on B1
with the target of the validation data, and then we will make pr
edictions from C1. And this is how we combine
different models with stacking, to hopefully make better predict
ions for
the test or the unobserved data. Let us go through an example,
a simple example in Python, in order to understand better,
as in in code, how it would work. It is quite simple, so even pe
ople not very experienced with
Python hopefully can understand this. The main logic is that we
will use
two base learners on some input data, a random forest and a line
ar regression. And then, we will try to combine

the results, starting with a meta learner, again, it will be linear regression. Let's assume we again have
a train data set, and a target variable for
this data set, and a test data set. Maybe the code seems a bit intimidating,
but we will go step by step. What we do initially is we take the train
data set and we split it in two parts. So we create a training and
a valid data set out of this, and we also split the target variable. So we create ytraining and
yvalid, and we split this by 50%. We could have chosen something else,
let's say 50%. Then we specify our base learners,
so model1 is the random forest in this case, and
model2 is a linear regression. What we do then is we fit
the both models using the training data and the training target. And we make predictions for
the validation data for both models, and at the same time we'll make
predictions for the test data. Again, for both models,
we save these as preds1, preds2, and for the test data,
test_preds1 and test_preds2. Then we are going to
collect the predictions, we are going to stack the predictions and
create two new data sets. One for validation,
where we call it stacked_predictions, which consists of preds1 and preds2. And then for the data set for
for the test predictions, called stacked_test_predictions, where we stack test_preds1 and test_preds2. Then we specify a meta learner, let's call it meta_model,
which is a linear regression. And we fit this model on the predictions
made on the validation data and the target for the validation data, which
was our holdout data set all this time. And then we can generate predictions for the test data by applying this model
on the stacked_test_predictions. This is how it works. Now, I think this is a good
time to revisit an old example we used in the first session,
about simple averaging. If you remember,
we had a prediction that was doing quite well to predict age when
the age was less than 50, and another prediction that was doing
quite well when age was more than 50. And we did something tricky,
we said if it is less than 50, we'll use the first one, if age is more
than 50, we will use the other one. The reason this is tricky is
because normally we use the target information to make this decision. Where in an ideal world, this is what
you try to predict, you don't know it. We have done it in order to show what
is the theoretical best we could get, or yeah, the best. So taking the same predictions and applying stacking, this is what the
end

result would actually look like. As you can see,
it has done pretty similarly. The only area that there is some
error is around the threshold of 50. And that makes sense, because the model
doesn't see the target variable, is not able to identify
this cut of 50 exactly. So it tries to do it only
based on the input models, and there is some overlap around this area. But you can see that stacking
is able to identify this, and use it in order to
make better predictions. There are certain things you need to
be mindful of when using stacking. One is when you have time-sensitive data,
as in let's say, time series, you need to formulate your
stacking so that you respect time. What I mean is, when you create
your train and validation data, you need to make certain that your train
is in the past and your validation is in the future, and ideally your
test data is also in the future. So you need to respect this
time element in order to make certain your model generalizes well. The other thing you need to look at is,
obviously, single model performance is important. But the other thing that is
also very important is model diversity, how different
a model is to each other. What is the new information each
model brings into the table? Now, because stacking, and depending
on the algorithms you will use for stacking, can go quite deep
into exploring relationships. It will find when a model is good, and when a model is actually bad or
fairly weak. So you don't need to worry too much
to make all the models really strong, stacking can actually extract
the juice from each prediction. Therefore, what you really need to focus
is, am I making a model that brings some information,
even though it is generally weak? And this is true, there have been many
situations where I've made, I've had some quite weak models in my ensemble,
I mean, compared to the top performance. And nevertheless, they were actually
adding lots of value in stacking. They were bringing in new information
that the meta model could leverage. Normally, you introduce
diversity from two forms, one is by choosing a different algorithm. Which makes sense, certain algorithms capitalize on
different relationships within the data. For example, a linear model will
focus on a linear relationship, a non-linear model can capture
better a non-linear relationships. So predictions may come a bit different. The other thing is you can
even run the same model, but you try to run it on different
transformation of input data, either less features or
completely different transformation. For example, in one data se

t you may treat categorical
features as one whole encoding. In another,
you may just use label in coding, and the result will probably p
roduce
a model that is very different. Generally, there is no limit to
how many models you can stack. But you can expect that
there is a plateauing after certain models have been added. So i
nitially, you will see some
significant uplift in whatever metric you are testing on every
time you run the model. But after some point, the incremental
uplift will be fairly small. Generally, there's no
way to know this before, exactly what is the number of models
where we will start plateauing. But generally, this is a affecte
d by how
many features you have in your data, how much diversity you mana
ged
to introduce into your models, quite often how many
rows of data you have. So it is tough to know this beforehand,
but generally this is
something to be mindful of. But there is a point where adding mo
re
models actually does not add that much value. And because the me
ta model, the meta model will only use predictions of other mode
ls. We can assume that the other
models have done, let's say, a deep work or
a deep job to scrutinize the data. And therefore the meta model
doesn't need to be so deep. Normally, you have predictions with
are correlated with the target. And the only thing it needs to d
o is
just to find a way to combine them, and that is normally not so
complicated. Therefore, it is quite often that
the meta model is generally simpler. So if I was to express this
in a random forest context, it will have lower depth than what w
as the
best one you found in your base models. This was the end of the
session,
here we discussed stacking. In the next one, we will discuss a v
ery
interesting concept about stacking and extending it on multiple
levels,
called stack net. So stay in tune.We can continue our discussion
 with StackNet. StackNet is a scalable meta modeling methodology
 that utilizes stacking to combine multiple models in a neural n
etwork architecture of multiple levels. It is scalable because w
ithin the same level, we can run all the models in parallel. It
utilizes stacking because it makes use of this technique we ment
ioned before where we split the data, we make predictions so som
e hold out data, and then we use another model to train on those
 predictions. And as we will see later on, this resembles a lot
in neural network. Now let us continue that naive example we gav
e before with the students and the teacher, in order to understa
nd what conceptually, in a real world, would need to add another
 layer. So in that example, we have a teacher that she was tryin
g to combine the answers of different students and she was outpu
tting an estimate of 17 under certain assumptions. We can make t
his example more interesting by introducing one more meta learne

r. Let's call him Mr. RF, who's also a physics teacher. Mr. RF b elieves that LR should have a bigger contribution to the ensembl e because he has been doing private lessons with him and he know s he couldn't be that far off. So he's able to see the data from slightly different ways to capitalize on different parts of the se predictions and make a different estimate. Whereas, the teach ers could work it out and take an average, we could create or we can introduce a higher authority or another layer of modeling h ere. Let's call it the headmaster, GBM, in order to shop, make b etter predictions. And GBM doesn't need to know the answers that the students have given. The only thing he needs to know is the input from the teachers. And in this case, he's more keen to tr ust his physics teacher by outputting a 16.2 prediction. Why wou ld this be of any use to people? I mean, isn't that already comp licated? Why would we want to ever try something so complicated? I'm giving you an example of a competition my team used, four l ayer of stacking, in order to win. And we used two different sou rces of input data. We generated multiple models. Normally, exit boost and logistic regressions, and then we fed those into a fo ur-layer architecture in order to get the top score. And althoug h we could have escaped without using that fourth layer, we stil l need it up to level three in order to win. So you can understa nd the usefulness of deploying deep stacking. Another example is the Homesite competition organized by Homesite insurance where again, we created many different views of the data. So we had di fferent transformations. We generated many models. We fed those models into a three-level architecture. I think we didn't need t he third layer again. Probably, we could have escaped with only two levels but again, deep stacking was necessary in order to wi n. So there is your answer, deep stacking on multiple levels rea lly helps you to win competitions. In the spirit of fairness and openness, there has been some criticism about large ensembles t hat maybe they don't have commercial value, they are confidentia lly expensive. I have to add three things on that. The first is, what is considered expensive today may not be expensive tomorro w and we have seen that, for example, with the deep learning, wh ere with the advent of GPUs, they have become 100 times faster a nd now they have become again very, very popular. The other thin g is, you don't need to always build very, very deep ensembles b ut still, small ensembles would still really help. So knowing ho w to do them can add value to businesses, again based on differe nt assumptions about how fast they want the decisions, how much is the uplift you can see from stacking, which may vary, sometim es it's more, sometime is less. And generally, how much computin g power they have. We can make a case that even stacking on mult iple layers can be very useful. And the last point is that these are predictive modeling competitions so it is a bit like the Ol ympics. It is nice to be able to see the theoretical best you ca n get because this is how innovation takes over. This is how we move forward. We can express StackNet as a neural network. So no rmally, in a neural network, we have these architecture of hidde n units where they are connected with input with the form of lin ear regression. So actually, it looks pretty much like a linear regression. So whether you have a set of coefficients and you ha ve a constant value where you call it bias in neaural networks, and this is how your output predictions which one of the hidden

units which are then taken, collected, to create the output. The concept of StackNet is actually not that much different. The only thing we want to do is, we don't want to be limited to that linear regression or to that perception. We want to be able to use any machine learning algorithm. Putting that aside, the architecture should be exactly the same, could be fairly similar. So how to train this? In a typical neural network, we use bipropagation. Here in this context, this is not feasible. I mean in the context of trying to make this network work with any input model because not all are differentiable. So this is why we can use stacking. Stacking here is a way to link the output, the prediction, the output of the node, with target variable. This is how the link also is made from the input features with a node. However, if you remember the way that stacking works is you have some train data. And then, you need to divide it into two halves. So, you use the first part called, training, in order to make predictions to the other part called, valid. If we, assuming that adding more layers gives us some uplift, if we wanted to do this again, we would have re-split the valid data into two parts. Let's call it, mini train, and mini valid. And you can see the problem here. I mean, assuming if we have really big data, then this may not really be an issue. But in certain situations where we don't have that much data. Ideally, we would like to do this without having to constantly re-split our data. And therefore minimizing the training data set. So, this is why we use a K-Fold paradigm. Let's assume we have a training data set with four features x0, x1, x2, x3, and the y variable, or target. If we are use k-fold where k = 4, this is a hyper-parameter which is what to put here. We would make four different parts out of these datasets. Here I have put different colors, colors to each one of these parts. What we would do then in order to commence the training, is we will create an empty vector that has the same size as rows, as in the training data, but for now is empty. And then, for each one of the folds, we would start, we will take a subset of the training data. In this case, we will start with red, yellow, and green. We will train a model, and then we will take the blue part, and will make predictions. And we will take these predictions, and we will put them in the corresponding location in the prediction array which was empty. Now, we are going to repeat the same process always using this rotation. So, we are now going to use the blue, the yellow, and the green part, and we will keep to create a model, and we will keep the red part for prediction. Again, we will take these predictions and put it into the corresponding part in the prediction array. And we will repeat again with the yellow, and the green. Something that I need to mention is that the K-Fold doesn't need to be sequential as a date. So, it would have been shuffled. I did it as this way in order to illustrate it better. But once we have finished and we have generated a whole prediction for the whole training data, then we can use the whole training data, in order to fit one last model and make now predictions for the test data. Another way we could have done this is for each one of the four models we were making predictions for the validation data. At the same time, we could have been making predictions for the whole test data. And after four models, we will just take an average at the end. We'll just divide the test predictions by four. But a different way to do it

, I have found this way I just explained better with neural networks, and the method where you use the whole training data to generate predictions for test better with tree-based methods. So, once we finish the predictions with the test, you can start again with another model this time. So you will generate an empty prediction, you will stack it next to your previous one. And you will repeat the same process. You will essentially repeat this until you're finished with all models for the same layer. And then, this will become your new training data set and you will generally begin all over again if you have a new layer. This is generally the concept. Though we could say this, in order to extend on many layers, we use this K-Fold paradigm. However, normally, neural networks we have this notion of epochs. We have iterations which help us to re-calibrate the weights between the nodes. Here we don't have this option, the way stacking is. However, we can introduce this ability of revisiting the initial data through connections. So, a typical way to connect the nodes is the one we have already explored where you have it input nodes, each node is directly related with the nodes of the previous layer. Another way to do this is to say, a node is not only affected, connected with the nodes of the directly previous layer, but from all previous nodes from any previous layer. So, in order to illustrate this better, if you remember the example with the headmaster where he was using predictions from the teachers, he could have been using also predictions from the students at the same time. This actually can work quite well. And you can also refit the initial data. Not just the predictions, you can actually put your initial x data set, and append it to your predictions. This can work really well if you haven't made many models. So that way, you get the chance to revisit that initial training data, and try to capture more informations. And because we already have meta l-models present, the model tries to focus on where we can explore any new information. So in this kind of situation it works quite well. Also, this is very similar to target encoding or many encoding you've seen before where you use some part of the data, let's say, a code on a categorical column, given some cross-validation, you generate some estimates for the target variable. And then, you insert this into your training data. Okay, you don't stack it, as in you don't create a new column, but essentially you replace one column with hold out predictions of your target variable which is essentially very similar. You have created the logic for the target variable, and you are essentially inserting it into your training data idea.