

Week 4

HyperParameter Tuning I

Hi, in this lecture, we will study hyperparameter optimization process and talk about hyperparameters in specific libraries and models. We will first discuss hyperparameter tuning in general. General pipeline, ways to tuning 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 factorization 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 important things to understand when tuning hyperparameters?

- Hyperparameter tuning in general
 - General pipeline
 - Manual and automatic tuning
 - What should we understand about hyperparameters?
- Models, libraries and hyperparameter optimization
 - Tree-based models
 - Neural networks
 - Linear models

- Tree-based models
 - GBDT: XGBoost, LightGBM, CatBoost
 - RandomForest/ExtraTrees
- Neural nets
 - Pytorch, Tensorflow, Keras...
- Linear models
 - SVM, logistic regression
 - Vowpal Wabbit, FTRL
- Factorization Machines (out of scope)
 - libFM, libFFM

First, there are tons of potential parameters to tune in every model. And so we need to realize which 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 we'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 up, right? For example, at GitHub or Kaggle Kernels.

Finally, the documentation sometimes explicitly states which parameter to tune first.

From the selected set of parameters we should then understand what would happen if we change one of the parameters?

How the training process and the training 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 usually faster to do it manually to be true.

How do we tune hyperparameters

1. **Select the most influential parameters**
 - a. There are tons of parameters and we can't tune all of them
2. **Understand, how exactly they influence the training**
3. **Tune them!**
 - a. Manually (change and examine)
 - b. Automatically (hyperopt, etc.)

So later in this video, actually discuss the most important parameters for some models along with some intuition 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.

Hyperparameter optimization software

- A lot of libraries to try:
 - *Hyperopt*
 - Scikit-optimize
 - Spearmint
 - GPyOpt
 - RoBO
 - SMAC3

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.

```
def xgb_score(param):
    # run XGBoost with parameters `param`

def xgb_hyperopt():
    space = {
        'eta' : 0.01,
        'max_depth' : hp.quniform('max_depth', 10, 30, 1),
        'min_child_weight' : hp.quniform('min_child_weight', 0, 100, 1),
        'subsample' : hp.quniform('subsample', 0.1, 1.0, 0.1),
        'gamma' : hp.quniform('gamma', 0.0, 30, 0.5),
        'colsample_bytree' : hp.quniform('colsample_bytree', 0.1, 1.0, 0.1),

        'objective':'reg:linear',

        'nthread' : 28,
        'silent' : 1,
        'num_round' : 2500,
        'seed' : 2441,
        'early_stopping_rounds':100
    }

    best = fmin(xgb_score, space, algo=tpe.suggest, max_evals=1000)
```

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.

Color-coding legend

1. Underfitting (bad)
2. Good fit and generalization (good)
3. Overfitting (bad)

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 behavior 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

Color-coding legend

- A parameter in red
 - Increasing it impedes fitting
 - Increase it to reduce overfitting
 - Decrease to allow model fit easier

powerful the model. 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.

HyperParameter Tuning II

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?

Plan for the video

- Tree-based models
 - GBDT: XGBoost, LightGBM, CatBoost
 - RandomForest/ExtraTrees
- Neural nets
 - Pytorch, Tensorflow, Keras...
- Linear models
 - SVM, logistic regression
 - Vowpal Wabbit, FTRL
- Factorization Machines (out of scope)
 - libFM, libFFM

Tree-based models

Model	Where
GBDT	XGBoost (dmlc/xgboost) LightGBM (Microsoft/LighGBM) CatBoost (catboost/catboost)
RandomForest, ExtraTrees	scikit-learn
Others	RGF (baidu/fast_rg)

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.

BDT	
XGBoost	LightGBM
<ul style="list-style-type: none">• max_depth• subsample• colsample_bytree, colsample_bylevel• min_child_weight, lambda, alpha• eta• num_round	<ul style="list-style-type: none">• max_depth/num_leaves• bagging_fraction• feature_fraction• min_data_in_leaf, lambda_l1, lambda_l2• learning_rate• num_iterations

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 overfit. Some simple parameter controls a fraction of objects to use when fitting 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 at every 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.

GBDT	
XGBoost	LightGBM
<ul style="list-style-type: none"> <code>max_depth</code> <code>subsample</code> <code>colsample_bytree</code>, <code>colsample_bylevel</code> <code>min_child_weight</code>, <code>lambda</code>, <code>alpha</code> <code>eta</code> <code>num_round</code> 	<ul style="list-style-type: none"> <code>max_depth/num_leaves</code> <code>bagging_fraction</code> <code>feature_fraction</code> <code>min_data_in_leaf</code>, <code>lambda_l1</code>, <code>lambda_l2</code> <code>learning_rate</code> <code>num_iterations</code>

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

GBDT	
XGBoost	LightGBM
<ul style="list-style-type: none"> <code>max_depth</code> <code>subsample</code> <code>colsample_bytree</code>, <code>colsample_bylevel</code> <code>min_child_weight</code>, <code>lambda</code>, <code>alpha</code> <code>eta</code> <code>num_round</code> 	<ul style="list-style-type: none"> <code>max_depth/num_leaves</code> <code>bagging_fraction</code> <code>feature_fraction</code> <code>min_data_in_leaf</code>, <code>lambda_l1</code>, <code>lambda_l2</code> <code>learning_rate</code> <code>num_iterations</code>

and added to the model with a learning rate η . 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. It 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 generalizes and doesn't take forever to train.

The nice thing is that we can freeze η to be reasonably small, say, 0.1 or 0.01, and then find how many rounds we should train the model until 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 η by the factor of alpha.

For example, we double the number of steps and divide η 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 other 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.

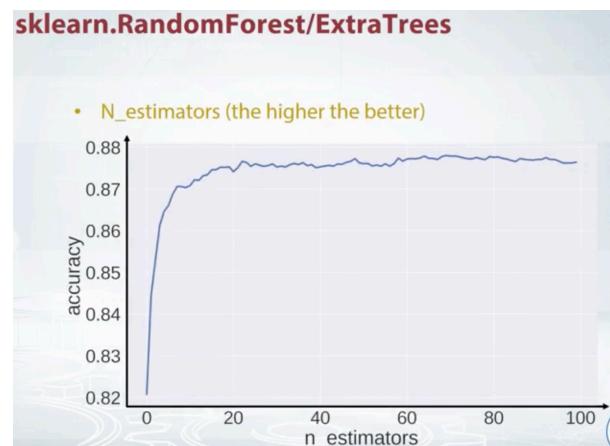
All right, we're finished with gradient boosting. Now let's get to RandomForest and ExtraTrees.

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 overfitting 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 evaluate 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.

sklearn.RandomForest/ExtraTrees

- `N_estimators` (the higher the better)
- `max_depth`
- `max_features`
- `min_samples_leaf`

Others:

- `criterion`
- `random_state`
- `n_jobs`

Advanced Features III

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.

Plan for the lecture: models

- Tree-based models
 - GBDT: XGBoost, LightGBM, CatBoost
 - RandomForest/ExtraTrees
- Neural nets
 - Pytorch, Tensorflow, Keras...
- Linear models
 - SVM, logistic regression
 - Vowpal Wabbit, FTRL
- Factorization Machines (out of scope)
 - libFM, libFFM

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 validation 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?

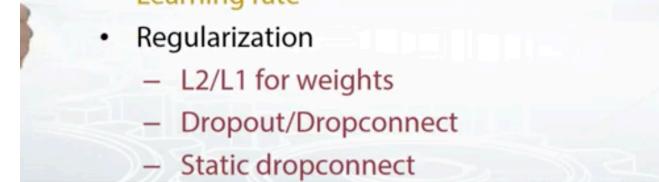
In fact, it turns out that huge batch size leads to more overfitting. Say a batch of 500 objects is large in experience.

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 underfitting, 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.

Neural Nets

- Number of neurons per layer
- Number of layers
- Optimizers
 - SGD + momentum
 - Adam/Adadelta/Adagrad/...
 - In practice lead to more overfitting
- Batch size
- Learning rate
- Regularization
 - L2/L1 for weights
 - Dropout/Dropconnect
 - Static dropconnect



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 vary it 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 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 add a dropout layer. You can vary 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.

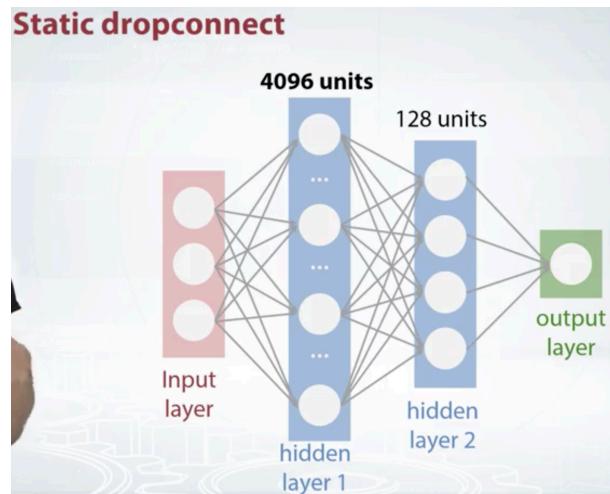
Dropout helps network to find features that really matters, and what never worked for me is to have dropout as the very first layer, immediately after data layer. This way some information is lost completely at the very beginning of the network and we observe performance degradation.

An interesting regularization technique that we used in the SpringLeaf 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 overfit badly.

But now to regularize 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 neural models. Yet, a carefully tuned LightGBM 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 SVM 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, SGDClassifier and SGDRegressor. 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 note that, the parameter C 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.

Linear models

- **Scikit-learn**
 - SVC/SVR
 - Sklearn wraps libLinear and libSVM
 - Compile yourself for multicore support
 - LogisticRegression/LinearRegression + regularizers
 - SGDClassifier/SGDRegressor
- **Vowpal Wabbit**
 - FTRL

For SVM, I usually start a very small C, 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 vary not only random seed, but also other parameters around an optimal value. For example, if optimal depth for XGboost is 5, we can average 3 XGboosts with depth 3, 4, and 5. Wow, it would be better if we could averaged 3 digiboosts with depth 4, 5, and 6.

- **Don't spend too much time tuning hyperparameters**
 - Only if you don't have any more ideas or you have spare computational resources
- **Be patient**
 - It can take thousands of rounds for GBDT or neural nets to fit
- **Average everything**
 - Over random seed
 - Or over small deviations from optimal parameters
 - e.g. average `max_depth=4,5,6` for an optimal 5

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.

Practice Quiz

1- What can we do to deal with overfitting in LightGBM?

- Decrease `min_data_in_leaf`
- Increase `bagging_seed`
- Decrease `num_leaves`
- Increase `bagging_fraction`
- Increase `min_data_in_leaf`

2- What of the following parameter changes speed-up *training process* for LightGBM? That is, time needed for one boosting iteration is decreased.

- Decreasing num_leaves
- Increasing max_bin
- Decreasing verbosity
- Decreasing bagging_fraction

3- How usually training time of SVM changes if we increase the value of parameter "C"?

Training time increases

Graded Quiz

Question 1

Which hyperparameters are first to tune in sklearn's RandomForest?

Correct answers:

- n_estimators, max_depth, min_samples_split Yes! These parameters are important. The first one should just be sufficiently large, you do not actually need to tune it.

Incorrect answers:

- n_jobs, random_state, verbose. Some of these parameters can even change the result of the training but only because of randomness involved. They are for sure not the parameters to tune.
- bootstrap, oob_score, warm_start. These parameters are not what you want to tune in the model!

Question 2

Suppose you fit LightGBM to your train data and check performance on the validation set. The train set consists of 500 rows and 1000 different features and validation set consist of 50 objects. You run automatic hyperparameter optimization method overnight and in the morning you select the best parameters, produce results for the test set and submit to the leaderboard. We also know that test set comes from the same distribution as train and validation sets.

Correct answers:

- There is a high chance of overfitting to the validation set. That is, there is a high chance that score on the test set will be bad. This is because we've tried too much hyperparameters while the dataset is small and the number of features is large. **Correct!** This is because of [multiple comparisons fallacy](#).

Incorrect answers:

- There is a low chance of overfitting to the validation set. That is, there is a high chance that score on the test set will be good. This is because we found good parameters on validation set and test set is similar to the test set. The fact that test and validation sets come from the same distribution does not make them similar in terms of optimal hyperparameters.

Question 3

Suppose you want to find a good set of hyperparameters for a dataset with 1000 points and have resources to do fitting 2000 times. Which method of model selection your should use?

Correct answers:

- k-Fold scheme (i.e. split data into k part, use k-1 parts for training and the last one for quality estimation; repeat for each part). Yes! It is easy to note than we will check $2000 / k$ sets of hyperparameters -- this number still big enough (for typical k in $[3,..,10]$), but this scheme is much more robust.

Incorrect answers:

- Hold-Out scheme (i.e. divide data into two parts, use first for model fitting and second for estimation of quality). See previous question for detailed explanation.
- Select model by quality on training set (i.e. fit model on whole dataset and measure quality on the same data). This is overfitting. Such way cannot be used to estimating quality of model.
- Leave-one-out validation (i.e. fit model for all points except one and estimate quality for this single point; repeat for every point). Since LOO requires 1000 fittings for single set of hyperparameters, we can estimate only 2 combinations of hyperparameters -- it's way too small.

Question 4

Suppose you train Neural Network with SGD and see that it overfits data. Which of the following actions can help you to regularize model?

Correct answers:

- Insert (or increase rate of) Dropout layers inside NN. Yes, Dropout is a known way to regularize model via dropping some randomly selected features at every step.
- Add (or increase) Weight Decay. Weight Decay is essentially the same thing as L2 Regularization.
- Reduce number of parameters (e.g. remove some layers). Reducing number of parameters make model less flexible and harder to overfit

Incorrect answers:

- Change optimization method to Adam. Adam is a better optimizer than SGD, thus overfits training data better and faster.
- Add more layers. No, this will increase complexity of neural net and do not introduce any regularization effect.

- [Tuning the hyper-parameters of an estimator \(sklearn\)](#)
- [Optimizing hyperparameters with hyperopt](#)
- [Complete Guide to Parameter Tuning in Gradient Boosting \(GBM\) in Python](#)

Tips And Tricks

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 tools, 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 moments. 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, let'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 will influence what competition you choose to participate in. If you want to learn more about an interesting problem, you may want the competition to

Before you enter a competition

Define your goals. What you can get out of your participation?

1. To learn more about an interesting problem
2. To get acquainted with new software tools
3. To hunt for a medal

have a wide discussion on the forums. For example, if you are interested 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 competitions with images like the nature conservancy features, monitoring competition. Or the planet, understanding the Amazon from space competition. And if you want to try to hunt for a medal, you may want to check how many submissions do participants have. And if the points that people 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 are people with few submissions in the top, that usually means there should be a non-trivial approach to this competition or it's discovered only by few people. Beside that, you may want to pay attention to the size of the top teams. If leaderboard mostly consists 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 step 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? What approaches you may want to take. After you're done, read forums 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 organize 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. Especially, try to understand why something does or doesn't work. For example, you have an idea about trying a deep gradient boosting decision tree model. To your joy, it works. Now, ask yourself why? Is there some hidden data structure we didn't notice before? Maybe you have categorical features with a lot of unique

values. If this is the case, you as well can 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 you enter a competition: Everything is a hyperparameter

Sort all parameters by these principles:

1. Importance
2. Feasibility
3. Understanding

Note: changing one parameter can affect the whole pipeline

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 layers in convolutional neural network, to the coefficient you finally submit is multiplied by. To understand what I should tune and change first, I like to sort all parameters by these principles. First, importance. Arrange parameters from important to not useful at all. Tune in this order. These may depend on data structure, on target, on metric, and so on. Second, feasibility. Rate parameters from, it is easy to tune, to, tuning this can take forever. Third, understanding. Rate parameters from, I know what it's doing, to, I have no idea. Here it is important to understand what each parameter will change in the whole pipeline. For example, if you increase the number of features significantly, you may want to change ratio of columns which is used to find the best split in gradient boosting decision tree. Or, if you change number of layers in convolution neural network, you will need more reports to train 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

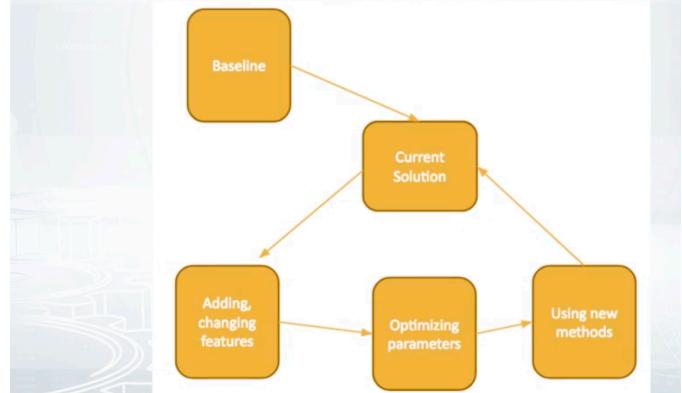
loading and preprocessing. I usually don't pay much attention to some sub optimal usage of computational resources but this particular case is of crucial importance. Doing things right at the very beginning will make your life much simpler and will allow you to save a lot of time and computational resources. 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. Down casting 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

- ▶ Do basic preprocessing and convert csv/txt files into hdf5/npy for much faster loading
- ▶ Do not forget that by default data is stored in 64-bit arrays, most of the times you can safely downcast it to 32-bits
- ▶ Large datasets can be processed in chunks

Performance evaluation

- ▶ Extensive validation is not always needed
- ▶ Start with fastest models - LightGBM



only with some marginal improvements. Same logic applies to initial model choice. I usually start with LightGBM, find some reasonably good parameters, and evaluate performance of 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 as, fast and dirty, always better. Try focusing on what is really important, 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 situations 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 transformed 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.

Initial pipeline

- Start with simple (or even primitive) solution
- Debug full pipeline
 - From reading data to writing submission file
- "From simple to complex"
 - I prefer to start with Random Forest rather than Gradient Boosted Decision Trees

Best Practices from Software Development

- Use good variable names
 - If your code is hard to read -- you definitely will have problems soon or later
- Keep your research reproducible
 - Fix random seed
 - Write down exactly how any features were generated
 - Use Version Control Systems (VCS, for example, git)
- Reuse code
 - Especially important to use same code for train and test stages

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 ideas from them to generate new features.

I usually start the competition 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 follow it.

But sometimes I just try to generate as many features as possible, put them in extra boost

- **Read forums and examine kernels first**
 - There are always discussions happening!
- **Start with EDA and a baseline**
 - To make sure the data is loaded correctly
 - To check if validation is stable
- **I add features in bulks**
 - At start I create all the features I can make up
 - I evaluate many features at once (not “add one and evaluate”)
- **Hyperparameters optimization**
 - First find the parameters to overfit train dataset
 - And then try to trim model

Code organization: keeping it clean

- **Very important to have reproducible results!**
 - Keep important code clean
- **Long execution history leads to mistakes**

```
In [521]: pr = bst.predict(X_val)
           smape(y_val, pr)

Out[521]: 0.042878536778555423
```

- **Your notebooks can become a total mess**

```
s = qq.sum(1)
ss = s[:,3]/qq.var()
sss = ss[0]
```

and study what helps and what does not. When tuning parameters, I first try to make model overfit to the training set and only then I change parameters to constrain the model.

I had situations when I could not reproduce one of my submissions. I accidentally changed something in the code and I could not remember what exactly, so nowadays I'm very careful about my code and script.

Another problem? Long execution 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.

One notebook per submission (and use git)

- main-2lv-submit1.ipynb
- main-2lv-submit2-1.ipynb
- main-2lv-submit2.ipynb
- main-2lv-submit3.ipynb

Before creating a submission restart the kernel

- Use “Restart and run all” button

I found a convenient way to validate the models that allows to use validation code with minimal changes to retrain a model on the whole dataset. In the competition, we are provided with training and test CSV files. You see we load them in the first cell. In the second cell, we split training set and actual

- Split *train.csv* into *train* and *val* with structure of *train.csv* and *test.csv*
- ```
1. train = pd.read_csv('data/train.csv')
 test = pd.read_csv('data/test.csv')

 from sklearn.model_selection import train_test_split

 # train set into new train and validation
 train_train, train_val = train_test_split(train, random_state=660)

 # save to disk
 train_train.to_csv('data/val/train.csv')
 train_val.to_csv('data/val/val.csv')
```

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.

- When validating, set it at the top of the notebook

```
train_path = 'data/val/train.csv'
test_path = 'data/val/val.csv'
```

- To retrain models on the whole dataset and get predictions for test set just change

```
train_path = 'data/train.csv'
test_path = 'data/test.csv'
```

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.

- **I use a library with frequent operations implemented**

- Out-of-fold predictions
- Averaging
- I can specify a classifier by its name

```
param = {
 'C':1.2,
}
res = trylib(train_2lv, y_train,
 'lsvc', param,
 one = False, skf_seed = 660, skf = 4,
 test_mode='whole', X_test=test_2lv).res
```

## KazAnova's Competition pipeline I

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 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 and modeling, up until this point. So, let's take all these steps one by one.

# Pipeline

**Understand the problem** (1 day)

**Exploratory analysis** (1-2 days)

*Define cv strategy*

**Feature engineering**  
(until last 3-4 days)

**Modelling** (until last  
3-4 days)

**Ensembling** (last 3-4 days)

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, LightGBM. 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, is it root mean square error, is it 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.

# Understand broadly the problem

- Type of problem
- How BIG is the data (How much I need?)?
- Hardware needed (CPUs, GPUs, RAM, Disk space)
- Software Needed (TF, sklearn, Lightgbm, xgboost)
- What is the metric being tested on?
- Previous code relevant?

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.

## Do some (exploratory data analysis) EDA

- Plot histograms of variables. Check that a feature looks similar between train and test.
- Plot features versus the target variable and vs time.
- Consider univariate predictability metrics (IV,R,auc)
- Binning numerical features and correlation matrices

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 tested 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

# Decide a cross validation Strategy

- This step is critical. Its success is a good indication for what is going to happen in the competition.
- *People have won by just selecting the right way to validate.*
- Is time is important? Split by time. **Time-based validation.**
- Different entities than the train. **Stratified validation.**
- Is it completely random. **Random validation** (random K-fold).
- **Combination** of all the above.
- Use test leader board to test.



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

# Feature engineering

- The type of problem defines the feature engineering.
- **Image classification:** Scaling, shifting, rotations, CNNs. Suggestion [previous data science bowls](#).
- **Sound classifications:** Fourier , Mfcc, specgrams, scaling . [Tenso flow speech recognition](#)
- **Text classification:** Tf-idf, svd, stemming, spell checking, stop words' removal, x-grams, [StumbleUpon Evergreen Classification](#).
- **Time series:** Lags, weighted averaging, exponential smoothing . [Walmart recruitment](#).
- **Categorical :** Target enc, freq, one-hot, ordinal, label encoding. [Amazon employee](#)
- **Numerical :** Scaling , binning, derivatives ,outlier removals, dimensionality reduction. [Africa soil](#).
- **Interactions:** multiplications, divisions, group-by features . Concatenations. [Homesite](#).
- **Recommenders:** Features on transactional history. Item popularity, frequency of purchase. [Acquire Valued Shoppers](#).
- This process **can be automated** using selection with cross validation.



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 know 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.

## KazAnova's Competition pipeline II

(It seems this should be the last video but is in the wrong place!!!)

### Modeling

- The type of problem defines the feature engineering.
- **Image classification:** CNNs (Resnet, VGG, densenet...)
- **Sound classifications:** CNNs(CRNN), LSTM
- **Text classification:** GBMs, Linear, DL, Naïve bayes, KNNs, LibFM, LIBFFM
- **Time series:** Autoregressive models, ARIMA, linear, GBMs, DL, LSTMs
- **Categorical features:** GBMs, Linear models, DL , LibFM, libFFm
- **Numerical Features:** GBMs, Linear models, DL, SVMs
- **Interactions:** GBMs, Linear models, DL
- **Recommenders:** CF, DL, LibFM, LIBFFM, GBMs



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 XGboost 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 methods 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.

# Ensembling

- All this time, **predictions** on internal validation and test **are saved**.  
(If **collaborating** with others, this is the point where everyone passes on their predictions as .csv files)
- Different ways to combine from **averaging** to multilayer **stacking**.
- Small data requires simpler ensemble techniques (like averaging).
- Helps to average a few **low-correlated predictions** with good scores.
- Bigger data can utilize stacking.
- **Stacking process repeats** the modelling process.



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

## Tips on collaboration

- It makes it more fun.
- You learn more.
- You score better.
- You gain in at least 2 ways. First you can cover more ground. Second every person seizes the problem from different angles leading to more thorough solutions.
- Start collaborating after getting some experience (maybe 2-3 competitions) to understand the dynamics.
- Start with people around your “rank”.
- Look for people that are likely to do different things well or that specialize in certain areas.



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 Kaggle 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.

## Selection final submissions

- Normally select the **best submission locally and best on leader board**.
- It is good to **monitor correlations**. If correlations are too high and submissions exist with high scores but significantly lower correlations, they could be considered too.

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 critics here, people doubt that doing these competitions stops your employability 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

# Final tips

- In these challenges **you never lose**. You may not win prize money, BUT you always gain in terms of knowledge, experience, meeting/collaborating with talented people in the field, boost your CV.
- Coffee is kind of a must when you do this!
- See it **like a game**...you have some tools ...or "weapons", you can get a score and you try to beat the "bad guys" score!
- **Take a break** often to rest your mind – do some physical exercise as it is unhealthy sitting on chair many hours.
- The kaggle community may be the most kind, helpful community I have ever experienced in any social context.
- After the competition look for people sharing approaches.
- Create a notebook with useful methods and update it.



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 helping 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 helpful, consider creating this notebook or library of all the approaches 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 with you and thank you very much for bearing with me and to see you next time.

[FarOn's framework for Kaggle competitions "kaggletils"](#)

[28 Jupyter Notebook tips, tricks and shortcuts](#)

## Advanced Features II

Hi everyone. This video is dedicated to the following advanced feature engineering techniques. Calculating various 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 have a chunk of data for some CTR task. Let's forget about target variable and focus on given features. Namely, User\_ID, unique identifier of a user, Page\_ID, an identifier of a page user visited, Ad\_price, item prices in the ad, and Ad\_position, relative position of an ad on the web page. The most straightforward way to solve this problem is to label encode the Ad\_position and fit some classifier. It would be a very good classifier that could take into account all the hidden relations between variables.

Some data for CTR task

|    | User_id | Page_id | Ad_price    | Ad_position  |
|----|---------|---------|-------------|--------------|
| 0  | 4       | 6       | 165.977125  | Bottom_right |
| 1  | 4       | 6       | 34.5395640  | Bottom_right |
| 2  | 4       | 6       | 29.1963786  | Bottom_left  |
| 3  | 4       | 6       | 79.4373729  | Bottom_left  |
| 4  | 4       | 6       | 290.534595  | Bottom_right |
| 5  | 4       | 6       | 314.412660  | Bottom_right |
| 6  | 4       | 6       | 138.9007639 | Bottom_right |
| 7  | 4       | 6       | 107.4711914 | Bottom_right |
| 8  | 4       | 6       | 242.1089786 | Bottom_left  |
| 9  | 4       | 7       | 27.16719836 | Bottom_left  |
| 10 | 4       | 7       | 413.5421978 | Bottom_right |

But no matter how good it is, it still treats all the data points independently. And this is where we can apply feature engineering. We can imply that an ad with the lowest price on the page will 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 features relevant to such an implication. We can add lowest and highest prices for every user and page pair. Position of an ad with the lowest price could also be of use in such case.

Here's one of the ways to implement statistical features with pandas. If our data is stored in the data frame df, we call groupby method like this to get maximum and minimum price values. Then store this object in gb variable, and

```
: gb = df.groupby(['user_id', 'page_id'], as_index=False).agg(
 {'ad_price': {'max_price': np.max, 'min_price': np.min}})
gb.columns = ['user_id', 'page_id', 'min_price', 'max_price']
gb
```

|   | user_id | page_id | min_price | max_price  |
|---|---------|---------|-----------|------------|
| 0 | 4       | 6       | 73.711548 | 474.637726 |
| 1 | 4       | 7       | 35.465202 | 121.542191 |

```
: df = pd.merge(df, gb, how='left', on=['user_id', 'page_id'])
```

then join it back to the data frame df.

I want to emphasize that you should not stop at this point. It's possible to add other useful features not necessarily calculated within user and page pair. It could be how many pages user has visited, how many pages user has visited during 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 can drastically increase the quality of the models. But what if there 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 useful information. On the other hand, the method is more flexible. We can fine tune things like the size of relevant neighborhood or metric. The most common and natural example of neighborhood analysis arises from property pricing. Imagine that you need to predict rental prices.

You would probably have some characteristics like floor space, number of rooms, presence of a bus stop. But you need something more than that to create a really good model. It could be the number of other houses in different neighborhoods like in 500 meters, 1,000 meters, or 1,500 meters, or average price per square meter in such neighborhoods, or the number of schools, supermarkets, and parking lots in such neighborhoods. The distances to the closest objects of interest like subway

### More features

- How many pages user visited
- Standard deviation of prices
- Most visited page
- Many, many more

### Neighbors

- Explicit group is not needed
- More flexible
- Much harder to implement

### Neighbors

- Number of houses in 500m, 1000m,..
- Average price per square meter in 500m, 1000m,..
- Number of schools/supermarkets/parking lots in 500m, 1000m,..
- Distance to closest subway station

stations 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 neighborhoods 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 SpringLeaf 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.

### KNN features in Springleaf

- Mean encode all the variables
- For every point, find 2000 nearest neighbors using Bray-Curtis metric

$$\Sigma |u_i - v_i| / \Sigma |u_i + v_i|$$

- Calculate various features from those 2000 neighbors

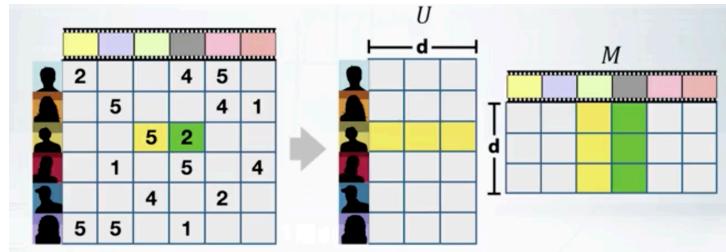
- Mean target of nearest 5,10,15,500, 2000 neighbors
- Mean distance to 10 closest neighbors
- Mean distance to 10 closest neighbors with target 1
- Mean distance to 10 closest neighbors with target 0

## Matrix Factorization

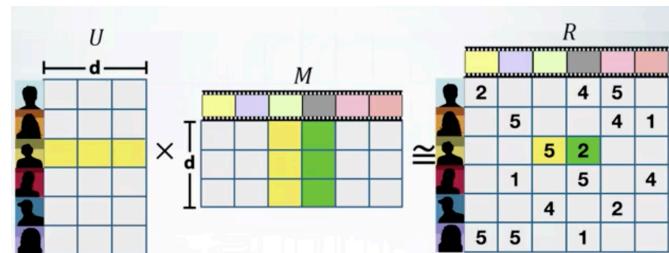
In this video I will talk about the application of matrix factorization technique in feature extraction.

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  $i$  has given a rating to item  $j$ . Assume that  $i$ th user have some features  $U_i$ . And  $j$ th item have is corresponding feature  $M_j$ . And scalar product of these features produce a rating  $R_{ij}$ . 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 measure of sadness or something similar. But generally you should consider them as some abstract features, which we can use to encode user in the same way as we did before with labeling encoder or one-hot encoder. Specifically our assumption about scalar product is the following. If we represent all attributes of user and items as matrices, their matrix product will be very close to the matrix with ratings. In other words, we try to find matrices  $U$  and  $M$ , such as their product gives the matrix  $R$ . That is why, this approach is called matrix factorization or matrix composition. In previous example, we used both row and column related features.



But sometimes we need only features correspond to rows. Let's consider another example. Suppose that we are processing texts, do you

|               | cat | is | dog | animal | nature |
|---------------|-----|----|-----|--------|--------|
| Cat is animal | 1   | 0  | 0   | 1      | 0      |
| Dog is animal | 0   | 1  | 1   | 1      | 0      |
| Cat is dog    | 1   | 1  | 1   | 0      | 0      |

remember how we usually classify text? We extract bag of word features and each document was described by a large sparse vector. If we do matrix factorization over these sparse features, we will get the representation for index displayed in yellow, and terms displayed in green.

Although we can somehow use representation for terms, we are interested only in representations for index. Now every document is described by a small, dense vector.

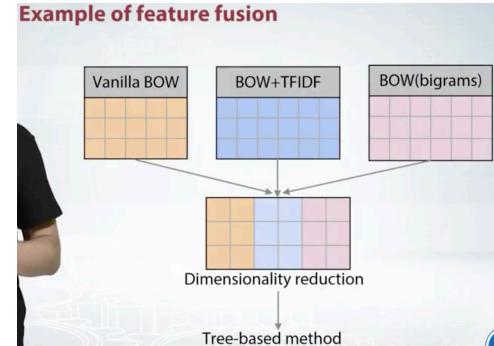
These are our features, and we can use them in a way similar to previous example. This case is often called dimensionality reduction. It's quite an efficient way to reduce the size of feature metric, and extract real valued features from one-hot encoded categorical ones.

In competitions we often have different options for pre-processing. For example, using text data, you have vanilla bag of words, bag of words with tf-itf weighing, bag of bigrams and so on. Using matrix optimization 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 together with other features 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 others as is. Besides reduction you can use such approach for getting another presentation of the same data. This is especially useful for ensembles since it provides diversity of fitted models and leads to a better generalization. Of course matrix factorization is a lossy of transformation, in other

|               | cat | is | dog | animal | nature |
|---------------|-----|----|-----|--------|--------|
| Cat is animal | 1   | 0  | 0   | 1      | 0      |
| Dog is animal | 0   | 1  | 1   | 1      | 0      |
| Cat is dog    | 1   | 1  | 1   | 0      | 0      |

|               |        |
|---------------|--------|
| Cat is animal | cat    |
| Dog is animal | is     |
| Cat is dog    | dog    |
|               | animal |
|               | nature |

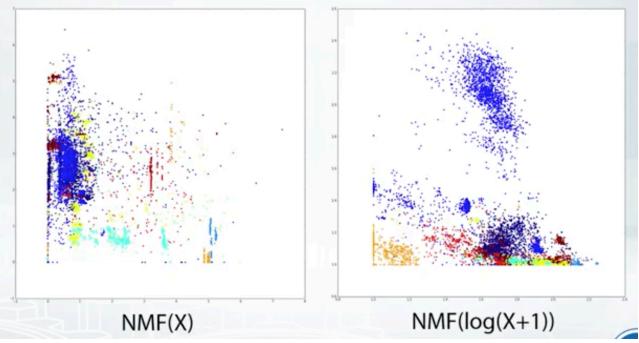


- Can be applied only for some columns
- Can provide additional diversity
  - Good for ensembles
- It is a **lossy** transformation. Its' efficiency depends on:
  - Particular task
  - Number of latent factors
    - Usually 5-100

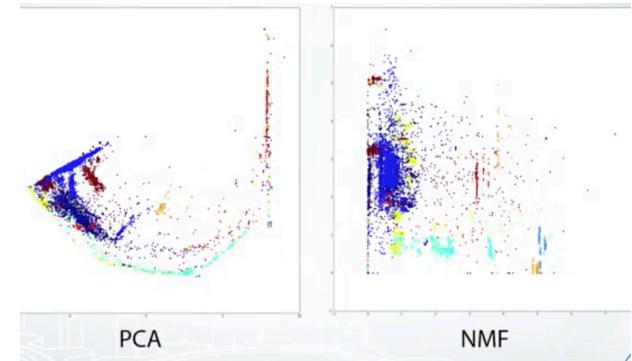
words we will lose some information after the search reduction.

Efficiency of this approach heavily depends on a particular task and choose a number of latent factors. 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 500.

Matrix factorization is similar in spirit to linear models.  
You can use the same transformation tricks.



Now, let's switch from general idea to particular implementations. Several matrix factorization methods are implemented in SKlearn library, the most famous ones are SVD and PCA. In addition, their use included Truncated SVD, which can work with sparse matrices. It's very convenient for example, in case of text datasets. Also there exists a so called non-negative matrix factorization, or NMF. It impose an additional restrictions that all hidden factors are non-negative, that is either zero or a positive number. It can be applied only to non-negative matrices. For example bag of words matrix where a row represented occurrence of each word in the document.



NMF has an interesting property, it transforms data in a way that makes data more suitable for decision trees. Take a look at the picture from Microsoft Malware Classification Challenge. It can be seen that NMF transform data from lines parallel to the axis.

A few more notes on matrix factorizations. 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 NMF, I advise you to apply the factorization to logarithmic transformed data. Here's another plot from the 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.

Reading 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.

Wrong way:

```
pca = PCA(n_components=5)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.fit_transform(X_test)
```

Right way:

```
X_all = np.concatenate([X_train,X_test])
pca.fit(X_all)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
```

To sum up, matrix composition 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.

## Feature Interactions

The main topic of this video is Feature Interactions. You will learn how to construct them and use in problem solving. Additionally, we will discuss them for feature extraction from decision trees.

Let's start with an example. Suppose that we are building a model to predict the best advertisement banner to display on a website. Among available features, there are two categorical ones that we will concentrate on.

### Example: banner selection

The category of the advertising banner itself and the category of the site the banner will be showing on. Certainly, we can use the features as two

| ... | category_ad   | category_site | ... | is_clicked |
|-----|---------------|---------------|-----|------------|
| ... | auto_part     | game_news     | ... | 0          |
| ... | music_tickets | music_news    | ..  | 1          |
| ... | mobile_phones | auto_blog     | ... | 0          |

independent ones, but a really important feature is indeed the combination of them.

We can explicitly construct the combination in order to incorporate our knowledge into a model. Let's construct new feature named `ad_site` that represents the combination.

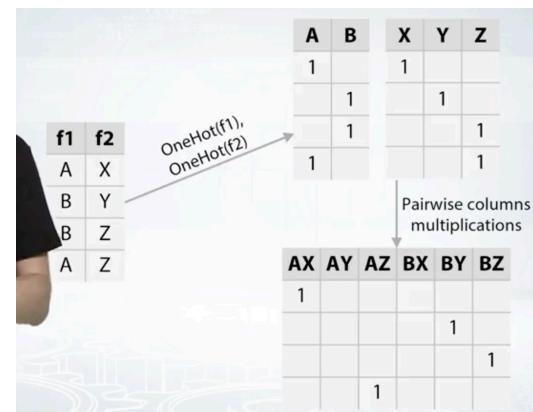
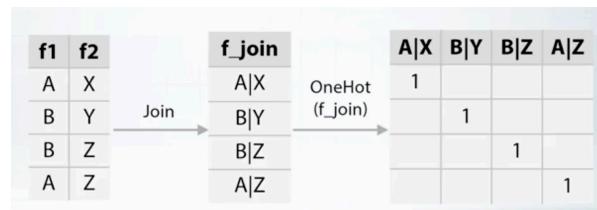
It will be categorical as the old ones, but set of its values will be all possible combinations of two original values. From a technical point of view, there are two ways to construct such interaction. Let's look at a simple example.

Consider 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 data points.

The first approach is to concatenate the text values of `f1` and `f2`, and use the result as a new categorical feature `f_join`. We can then apply the OneHot according to it.

The second approach consist of two steps. Firstly, apply OneHot and connect to features `f1` and `f2`. Secondly, construct new metrics by multiplying each column from `f1` encoded metrics to each column from `f2` encoded metrics. It worth mentioning that both methods results in practically the same new feature representations.

| ... | <code>category_ad</code> | <code>category_site</code> | ... | <code>is_clicked</code> |
|-----|--------------------------|----------------------------|-----|-------------------------|
| ... | auto_part                | game_news                  | ... | 0                       |
| ... | music_tickets            | music_news                 | ..  | 1                       |
| ... | mobile_phones            | auto_blog                  | ... | 0                       |
| ... | <code>ad_site</code>     |                            | ... | <code>is_clicked</code> |
| ... | auto_part game_news      |                            | ... | 0                       |
| ... | music_tickets music_news |                            | ..  | 1                       |
| ... | mobile_phones auto_blog  |                            | ... | 0                       |



In the above example, we can consider as interactions between categorical features, but similar ideas can be applied to real valued features.

For example, having two real valued features  $f_1$  and  $f_2$ , interactions between them can be obtained by multiplications of  $f_1$  and  $f_2$ .

In fact, we are not limited to use only multiplication operation. Any function taking two arguments like sum, difference, or division is okay. The following transformations significantly enlarge feature space and makes learning easier, but keep in mind that it makes overfitting easier too.

It should be emphasized that for tree based algorithms such as the random forest or gradient boost decision trees it's difficult to extract such kind of dependencies. That's why the above transformation are very efficient for tree based methods.

Let's discuss practical details now. pairwise feature generation approaches greatly increase the number of the features. If there are  $N$  original features, there will be  $N$ -square. And will be even more features if several types of interaction are used.

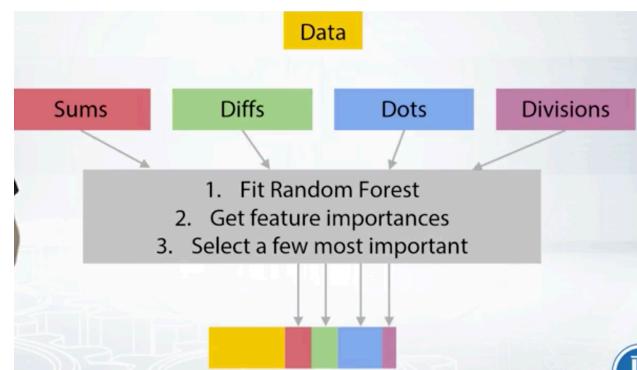
| $f_1$ | $f_2$ | $f_{\text{join}}$ |
|-------|-------|-------------------|
| 1.2   | 0.0   | 0.0               |
| 3.4   | 0.1   | 0.34              |
| 5.6   | 1.0   | 5.6               |
| 7.8   | -1.0  | -7.8              |

Mul

- We have a lot of possible interactions –  $N^2$  for  $N$  features.
  - a. Even more if use several types in interactions
- Need to reduce its' number
  - a. Dimensionality reduction
  - b. Feature selection

There are two ways to moderate this, either do feature selection or dimensionality reduction. I prefer doing the selection since not all but only a few interactions often achieve the same quality as all combinations of features.

For each type of interaction, I construct all piecewise feature interactions. Fit random forests over them and select several most important features. Because number of resulting features for each type is relatively small, It's possible to join them together along with



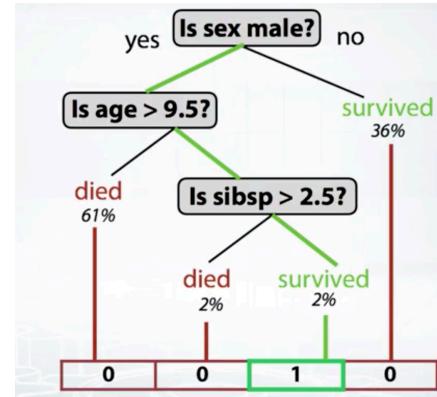
original features and use as input for any machine learning algorithm usually tree based method.

During the video, we have examined method to construct second order interactions. But you can similarly produce third order or higher.

- We looked at 2nd order interactions.
- Such approach can be generalized for higher orders.
- It is hard to do generation and selection automatically.
- Manual building of high-order interactions is some kind of art.

Due to the fact that number of features grow rapidly with order, it become difficult to work with them. Therefore, high order interactions are often constructed semi-manually. And this is an art in some ways.

Additionally, I would like to talk about methods to construct categorical features from decision trees. Take a look at the decision tree. Let's map each leaf into a binary feature. 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 ensemble of them. For example, a random forest, then such operation 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 apply method which takes as input feature metrics and returns corresponding indices of leaves.  
xgboost, also support it via a parameter pred\_leaf in predict method.

In sklearn:

```
| tree_model.apply()
```

In xgboost:

```
| booster.predict(pred_leaf=True)
```

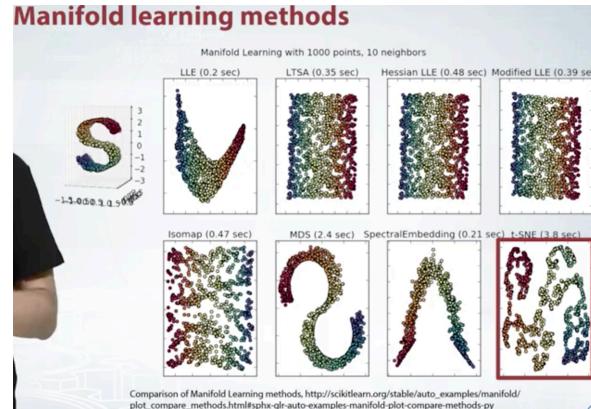
I suggest reading library documentations in order to get more information about these methods and APIs. In the end of this video, I will tackle the main points. We examined method to construct an interactions of categorical features. Also, we extend the approach to real-valued features. And we have learned how to use trees to extract high order interactions.

## tSNE

Today, we will discuss tSNE method for visualizing data and generating features. At the end of this video, you will be able to use tSNE in your products. In the previous video, we learned about matrix factorization technique that is spiritually(?) very close to linear models.

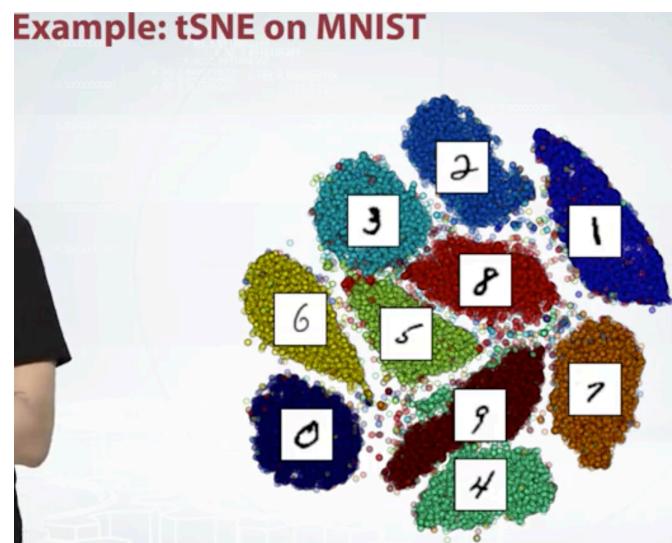
In this video, we will touch the subject of non-linear methods of dimensionality reduction. That says in general are called manifold learning.

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. tSNE result is placed at the right bottom corner on the slide. tSNE is the main topic of the lecture, as it tells of how tSNE 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. ? shows that these clusters are meaningful and corresponds to the target numbers well. Moreover, neighbor clusters corresponds



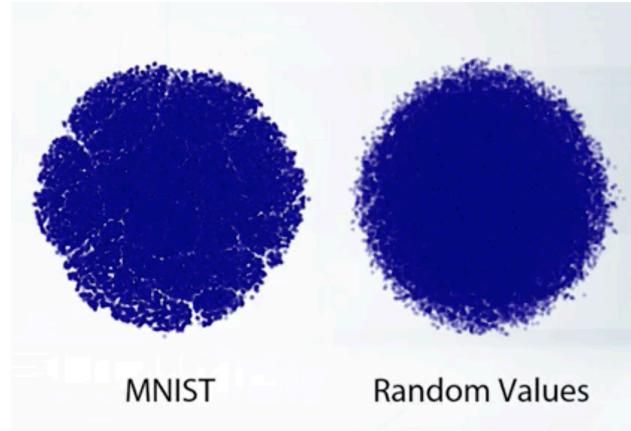
to a visually similar numbers.

For example, cluster of three is located next to the cluster of five which in turn is adjustment to the cluster of six and eight.

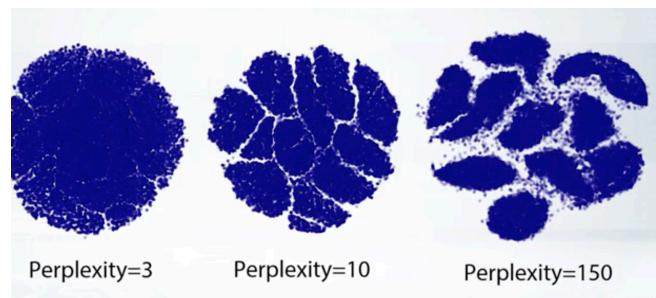
If data has explicit structure as in case of MNIST dataset, it's likely to be reflected on tSNE plot. For the reason tSNE is widely used in exploratory data analysis. However, do not assume that tSNE is a magic one that always helps. For example, a misfortune choice of hyperparameters may lead to poor results.

Consider an example, in the left there is presented a tSNE projection of exactly the same

MNIST data as in previous example, only perplexity parameter has been changed. On the right, so it present a tSNE projection of random data. We can see that choice of hyper 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 intuition of how tSNE works. I strongly advise you to take a look at it.

In addition to EDA, tSNE can be considered as a method to obtain new features from data. You should just concatenate the transformed coordinates to the original feature matrix.

Now practical details. As it has been shown earlier, the results of tSNE algorithm strongly depends on hyperparameters. It is good practice to use several projections with different perplexities.

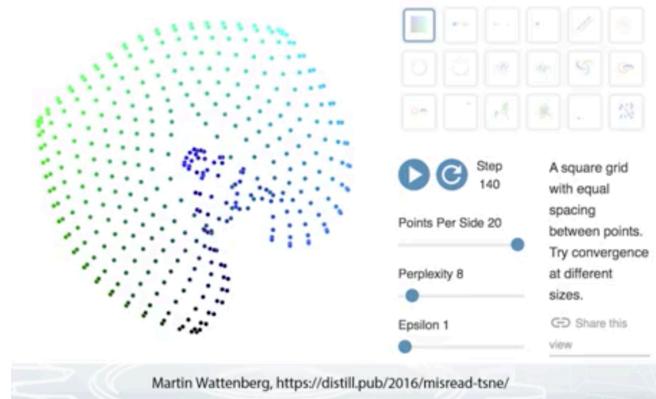
In addition, because of stochasticity of this methods results in different projections even with the same data and hyperparameters. This means the **train and test sets should be projected together rather than separately**. Also, tSNE will run for a long time if you have a lot of features. If the number 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

## How to Use t-SNE Effectively

Although extremely useful for visualizing high-dimensional data, t-SNE plots can sometimes be mysterious or misleading. By exploring how it behaves in simple cases, we can learn to use it more effectively.



It is good practice to use several

### Practical Notes

- Result heavily depends on hyperparameters (perplexity)
  - Good practice is to use several projections with different perplexities (5-100)
- Due to stochastic nature, tSNE provides different projections even for the same data\hyperparams
  - Train and test should be projected together
- tSNE runs for a long time with a big number of features
  - it is common to do dimensionality reduction before projection.

structure, then it likely be reflected 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 EDA, tSNE is working very well as a feature for fitting models.

### Question 1

Imagine that we apply `X = PCA(n_components=5).fit_transform(data)` and `data` has shape `(5000, 53)`. What is the shape of `X`?

Correct answers:

- (5000, 5). Yes, it should be `(n_samples, n_components)`.

Incorrect answers:

- (5, 53). No, we cannot transform 5000 samples into 5.
- (5, 5000). No, we cannot transform 5000 samples into 5.
- (53, 5). No, we cannot transform 5000 samples into 53.

### Question 2

To which data NMF is NOT applicable?

Correct answers:

- Standartized matrix. "Standartized" means that every feature column has zero mean and unit variance. This implies that we have negative values and cannot apply NMF.

Incorrect answers:

- Bag-of-words matrix. Since BoW matrix is non-negative matrix, we \*can\* apply NMF to it.
- One-Hot encoded feature. Since this matrix contains only 0's and 1's -- it is non-negative and we can apply NMF.

### Question 3

Suppose we have 2 categorical features: **f1** with A possible values and **f2** with B possible values. How many values will their interaction have?

Correct answers:

- Less or equal to A \* B. True. Sometimes some value (e.g. \*a\*) from A cannot be used with some value (eg \*b\*) from B. In this case, we have no chance to see \*ab\* combination. If all value from A can be used with all values from B -- we will get A\*B new possible values.

Incorrect answers:

- Exactly A + B. No, it's too small.
- Exactly A \* B. Not exactly, sometimes some value (e.g. \*a\*) from A cannot be used with some value (eg \*b\*) from B. In this case, we have no chance to see \*ab\* combination.
- max(A, B). No, it's way too small.

#### **Question 4**

Imagine we have 2 categorical features represented as integers: **f1** with all values in range [0, 1000] and **f2** with values in range [0, 100]. What is the correct way to build their interaction?

Correct answers:

- `f1.astype(str) + "_" + f2.astype(str)`. Yes, this is the right answer

Incorrect answers:

- `f1 + f2`. There are some problems. For example, if  $f1+f2=100$ : it is an interaction of 0 and 100, 100 and 0, or 90 and 10?
- `f1.astype(str) + f2.astype(str)`. There is some problems. For example, "123": it is an d interaction of "1" and "23" or "12" and "3"?
- `(f1 + f2).astype(str)`. It is essentially the same way as just `f1+f2`.

#### **Question 5**

What is a correct way to get t-SNE projection of train and test data?

Correct answers:

- Apply t-SNE to concatenation of train and test and split projection back. This this the right way since train and test will projected in the same way.

Incorrect answers:

- Apply t-SNE to the train and after that to the test. This this case we will have 2 different projections.
- Apply t-SNE to the test first and after to train. This this case we will have 2 different projections.
- Doesn't matter, all variants will produce the same result. No, since tSNE results in randomized projection

#### **Question 6**

Is it possible to do t-SNE projection into 20-dimensional space?

Correct answers:

- Yes, why not. You can do tSNE projection into arbitrary space.

Incorrect answers:

- No, only 2-dim or 3-dim projections are possible. Wrong! Despite tSNE is quite often used for visualization purposes, you don't limit to use only 2- or 3-dimensional projections.

Matrix Factorization:

- [Overview of Matrix Decomposition methods \(sklearn\)](#)

t-SNE:

- [Multicore t-SNE implementation](#)
- [Comparison of Manifold Learning methods \(sklearn\)](#)
- [How to Use t-SNE Effectively \(distill.pub blog\)](#)
- [tSNE homepage \(Laurens van der Maaten\)](#)
- [Example: tSNE with different perplexities \(sklearn\)](#)

Interactions:

- [Facebook Research's paper about extracting categorical features from trees](#)
- [Example: Feature transformations with ensembles of trees \(sklearn\)](#)

## Ensembling

This will be the first video in a series that we will be discussing on ensemble methods for machine learning.

What is ensemble modeling? I think with this term, we refer to combining many different machine learning models in order to get a more powerful prediction. There are various ensemble methods. Here we'll discuss a few. Those that we encounter quite often, in predictive modeling 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.

### Examined ensemble methods

- Averaging (or blending)
- Weighted averaging
- Conditional averaging
- Bagging
- Boosting
- Stacking
- StackNet

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

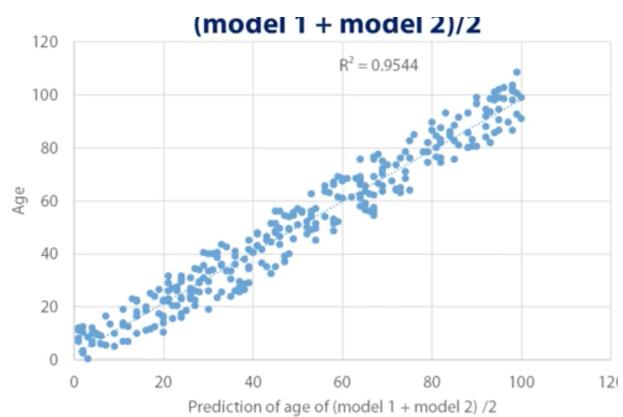
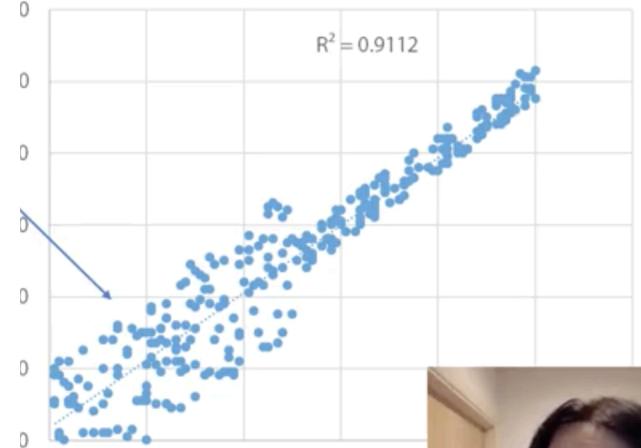
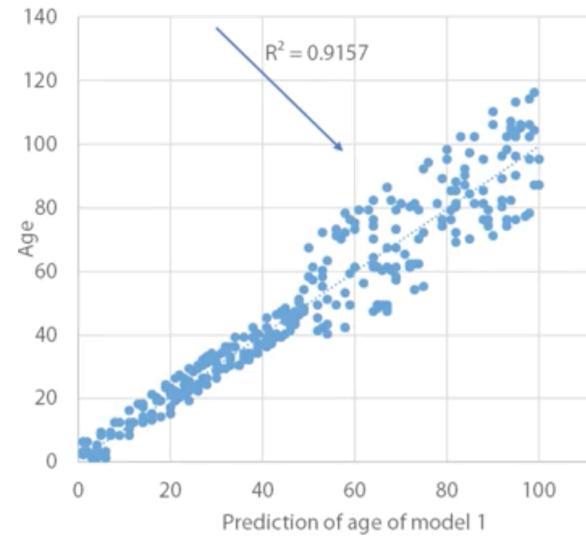
the two, the actual age in our prediction, looks like in the graph, as in the graph. So you can see that the model boasts quite a higher R-square of a value of 0.91, but it doesn't do so well in the whole range of values. So when age is less than 50, the model actually does quite well. But when age is 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 have 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.

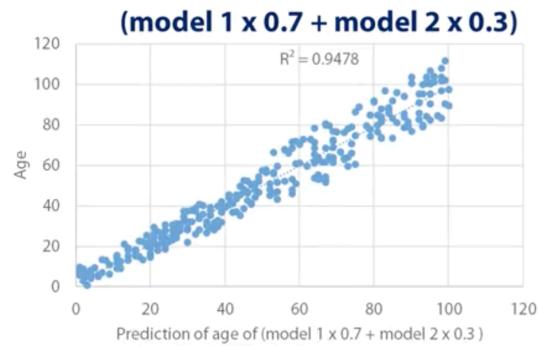
So what will happen if we were to try to combine this two with a simple averaging method, in other words, just say  $(\text{model 1} + \text{model two}) / 2$ , so a simple averaging method.

The end result will look as in the new graph. So, R-square has moved to 0.95, which is a considerable improvement versus the 0.91 we had before, and as you can see, on average, the points tend to be closer with the reality. So the average error is smaller. However, as you



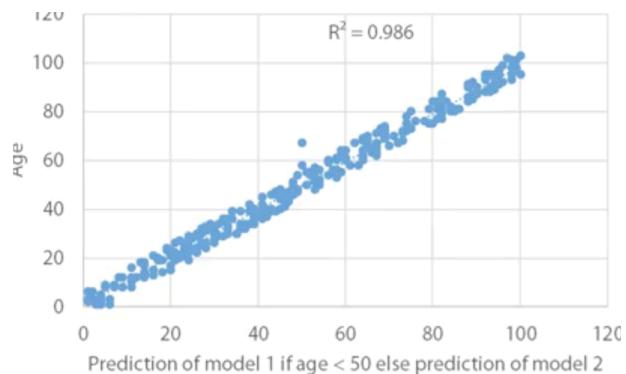
can see, the model doesn't do better as an individual models for the areas where the models were doing really well, nevertheless, it does better on average.

This is something we need to understand, that there is potentially a better way to combine these models. We could try to take a weighting average. So say, I'm going to take 70% of the first model prediction and 30% of the second model prediction. In other words,  $(\text{model 1} \times 0.7 + \text{model 2} \times 0.3)$ , 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 predictive power and it doesn't make sense to rely more in 1.



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 model that scores really well when age is less than 50, and another model that scores 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, hopefully you found it useful.



## Bagging

Bagging refers to averaging slightly different versions of the same model as a means to improve the predictive power. A common and quite successful application of bagging is the Random Forest. Where you would run many different versions of decision trees in order to get a better prediction. Why should we consider bagging? Generally, in the modeling process, there are two main sources of error. There are errors due to bias often referred to as under-fitting, and errors due to variance often referred to as overfitting. In order to better understand this, I'll give you two opposite examples. One with high bias and low variance 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 know this person is quite rich and we're trying to find him, this person who'll buy a racing or an expensive car. Our model has high variance, has high bias if it says that this person is young and I think he's not going to buy an expensive car. What the model has done here is that it hasn't explored 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 underfitted. However, this is also associated with low variance because this relationship, the fact that a young person generally doesn't buy an expensive car is generally true so we would expect this information to generalize well enough in unforeseen 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 lives 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 explored a lots of information about the training data. However, it is making the mistake that every person that has these characteristics is going to buy a car. Therefore, it generalizes for something that it shouldn't. In other words, it has already exhausted the information in the training data and the results are not significant. So, here, we actually have high variance but low bias. If we were to visualize the relationship between prediction error and model complexity, it would look like that.

When we begin 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 generalizable. They are simple. However, after a point, any improvements 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 its utmost value. By making slightly different or let say randomized models, we ensure that the predictions do not read very high variance. They're generally more generalizable. We don't over exhaust the information in the training data. At the same time, we saw before that when you average slightly different models, we are generally able to get better predictions and we can assume that in 10 models, we are still able to find quite significant information about the training data. Therefore, this is why bagging tends to work quite well and personally, I always use bagging. When I say, "I fit a model," I have actually not fit a model I have fit a bagging 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 procedures so by changing the seed you ensure that they are made slightly differently. At the same time, you can run a model with less rows or

you could use bootstrapping. Bootstrapping is different from row subsampling in the sense that you create an artificial dataset so you might let's say data row the training data three or four times. You create a random dataset from the training data. A different form of randomness can be imputed with shuffling. There are some algorithms, which are sensitive to the order of the data. By changing the order you ensure that the models become quite different. Another way is to date a random sample of columns so bid models on different features or different variables of the data. Then you have model-specific

### Parameters that control bagging?

- Changing the seed
- Row (Sub) sampling or Bootstrapping
- Shuffling
- Column (Sub) sampling
- Model-specific parameters
- Number of models (or bags)
- (Optionally) parallelism

parameters. For example, in a linear model, you will try to build 10 different let's say logistic regression with slightly different regularization 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 principle, the more bags you put, it doesn't hurt you. It makes results better but after some point, performance start plateauing. So there is a cost benefit with time but, in principle, more bags is generally better and optionally, you can also apply parallelism. Bagging models are independent to each other, which means you can build many of them at the same time and make full use of your computation power. Now, we can see an example about bagging but before I do that, just to let you know that a bagging estimators that scikit-learn has in Python are actually quite cool. Therefore, I recommend them.

This is a typical 15 lines of code that I use quite often. They seem really simple but they're actually quite efficient. Assuming you have a training at the test dataset and to target variable, what you do is you specify some bagging parameters. What is the model I'm going to use at random forest? How many bags I'm going to run? 10. What will be my seed? One. Then you create

an object, an empty object that will save the predictions and then you run a loop for as many bags as you have 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 average of these predictions. This is the end of the session. In this session, we discussed bagging as a popular form of ensembling. We saw bagging in association with variants and bias and we also saw in the example about how to use it.

## Examples of bagging

### **BaggingClassifier** and **BaggingRegressor** from Sklearn

```
train is the training data
test is the test data
y is the target variable
model=RandomForestRegressor()
bags=10
seed=1
create array object to hold bagged predictions
bagged_prediction=np.zeros(test.shape[0])
#loop for as many times as we want bags
for n in range (0, bags):
 model.set_params(random_state=seed + n)# update seed
 model.fit(train,y) # fit model
 preds=model.predict(test) # predict on test data
 bagged_prediction+=preds # add predictions to bagged prediction
#take average of predictions
bagged_prediction/= bags
```

## Boosting

Boosting is a form of weighted averaging of models where each model is built sequentially in a way that it takes into account previous model performance. In order to understand this better, remember that before, we discussed about bagging, and we saw that we can have it at many different models, which are independent to each other in order to get a better prediction. Boosting does something different. It says, now I tried to make a model, but I take into account how well the previous models have done in order to make a better prediction.

There are **two** main boosting 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 order to understand it better. Let's say we have a tabular data set, with four features. Let's call them  $x_0$ ,  $x_1$ ,  $x_2$ , and  $x_3$ , 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  $\text{pred}$ . These predictions have a certain margin of error. We can calculate these absolute error, and when I say absolute error, is absolute of  $y$  minus our prediction. You can see there

| Rownum | <b>x0</b> | <b>x1</b> | <b>x2</b> | <b>x3</b> | <b>y</b> | <b>pred</b> | <b>abs.error</b> | <b>weight</b> |
|--------|-----------|-----------|-----------|-----------|----------|-------------|------------------|---------------|
| 0      | 0.94      | 0.27      | 0.80      | 0.34      | 1        | 0.80        | 0.20             | 1.20          |
| 1      | 0.84      | 0.79      | 0.89      | 0.05      | 1        | 0.75        | 0.25             | 1.25          |
| 2      | 0.83      | 0.11      | 0.23      | 0.42      | 1        | 0.65        | 0.35             | 1.35          |
| 3      | 0.74      | 0.26      | 0.03      | 0.41      | 0        | 0.40        | 0.40             | 1.40          |
| 4      | 0.08      | 0.29      | 0.76      | 0.37      | 0        | 0.55        | 0.55             | 1.55          |
| 5      | 0.71      | 0.76      | 0.43      | 0.95      | 1        | 0.34        | 0.66             | 1.66          |
| 6      | 0.08      | 0.72      | 0.97      | 0.04      | 0        | 0.02        | 0.02             | 1.02          |

are predictions which are very, very far off, like row number five, but there are others like number six, which the model has actually done quite 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, and we say that this weight is 1 plus the absolute error. There are 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 what 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 can almost interpret weight has the number of times that a certain row appears in my data. So let's say weight was 2, this means that this row appears twice, and therefore, has bigger contribution to the total error. You can keep repeating this process. You can, again, calculate a new error based on this error. You calculate new weights, and this is how you sequentially add models to the ensemble that take into account how well each model has done in certain cases, maximizing the focus from where the previous models have done more wrong.

There are certain parameters associated 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 averaging. 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 models a little bit. It is very good to control over-fitting.

The second parameter we look at is the number of estimators. This is quite important. And normally, there is an inverse relationship, an opposite relationship, with the learning rate. So the more estimators we add to these type of ensemble, the smaller learning rate we need to put. This is sometimes quite difficult to find the right values, and we do it with the help of cross-validation. 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 performance, we find this to be 0.1. What we can do then is,

## Weight based boosting parameters

- Learning rate (or shrinkage or eta)
- Number of estimators
- Input model – can be anything that accepts weights
- Sub boosting type:
  - AdaBoost – Good implementation in sklearn (python)
  - LogitBoost - Good implementation in Weka (Java)

let's say, we can double the number of estimators, make it 200, and divide the learning rate by 2, so we can put 0.05, and then we take 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 its 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.

### **Residual based Boosting**

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 won 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

| Rownum | x0   | x1   | x2   | x3   | y | pred | error |
|--------|------|------|------|------|---|------|-------|
| 0      | 0.94 | 0.27 | 0.80 | 0.34 | 1 | 0.80 | 0.20  |
| 1      | 0.84 | 0.79 | 0.89 | 0.05 | 1 | 0.75 | 0.25  |
| 2      | 0.83 | 0.11 | 0.23 | 0.42 | 1 | 0.65 | 0.35  |
| 3      | 0.74 | 0.26 | 0.03 | 0.41 | 0 | 0.40 | -0.40 |
| 4      | 0.08 | 0.29 | 0.76 | 0.37 | 0 | 0.55 | -0.55 |
| 5      | 0.71 | 0.76 | 0.43 | 0.95 | 1 | 0.34 | 0.66  |
| 6      | 0.08 | 0.72 | 0.97 | 0.04 | 0 | 0.02 | -0.02 |

| Rownum | x0   | x1   | x2   | x3   | y     |
|--------|------|------|------|------|-------|
| 0      | 0.94 | 0.27 | 0.80 | 0.34 | 0.2   |
| 1      | 0.84 | 0.79 | 0.89 | 0.05 | 0.25  |
| 2      | 0.83 | 0.11 | 0.23 | 0.42 | 0.35  |
| 3      | 0.74 | 0.26 | 0.03 | 0.41 | -0.4  |
| 4      | 0.08 | 0.29 | 0.76 | 0.37 | -0.55 |
| 5      | 0.71 | 0.76 | 0.43 | 0.95 | 0.66  |
| 6      | 0.08 | 0.72 | 0.97 | 0.04 | -0.02 |

| Rownum | x0   | x1   | x2   | x3   | y     | new pred | old pred |
|--------|------|------|------|------|-------|----------|----------|
| 0      | 0.94 | 0.27 | 0.80 | 0.34 | 0.2   | 0.15     | 0.80     |
| 1      | 0.84 | 0.79 | 0.89 | 0.05 | 0.25  | 0.20     | 0.75     |
| 2      | 0.83 | 0.11 | 0.23 | 0.42 | 0.35  | 0.40     | 0.65     |
| 3      | 0.74 | 0.26 | 0.03 | 0.41 | -0.4  | -0.30    | 0.40     |
| 4      | 0.08 | 0.29 | 0.76 | 0.37 | -0.55 | -0.20    | 0.55     |
| 5      | 0.71 | 0.76 | 0.43 | 0.95 | 0.66  | 0.24     | 0.34     |
| 6      | 0.08 | 0.72 | 0.97 | 0.04 | -0.02 | -0.01    | 0.02     |

To predict Rownum=1 we would say :

$$\text{Final prediction} = 0.75 + 0.20 = \mathbf{0.95}$$



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 to make certain that every model has the right contribution. If you intent to put many, then you need to make sure that your models have very, very small contribution. Again, you decide these parameters based on cross-validation and the logic is very similar as explained before. Other things that work really well is taking a subset of rows or a subset of columns when you build its model.

Actually, there is no reason why we wouldn't use this with the previous algorithm. The way its based, it is more common with this type of boosting, and internally works quite well. For input 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 common or more successful right now in a predictive modeling context is the gradient based, which is actually what I explained with you how the prediction and you don't move 100 percent with that direction if you apply the learning rate. The other very interesting one, which I've actually find it very efficient especially in classification problems is the dart. Dart, it imposes a drop out mechanism in order to control the contribution of the trees. This is a concept derived from deep learning where you say, "Every time I make a new prediction in my sample, every time I add a new estimate or I'm not relying on all previous estimators but only on a subset of them." Just to give you an example, let's say we have a drop out rate

#### Residual based boosting parameters

- Learning rate (or shrinkage or eta)

$$\text{predictionN} = \text{pred0} + \text{pred1} * \text{eta} + \dots + \text{predN} * \text{eta}$$

#### Residual based boosting parameters

- Learning rate (or shrinkage or eta)
- Number of estimators
- Row (sub) sampling
- Column (sub) sampling
- Input model – better be trees.
- Sub boosting type:
  - Fully gradient based
  - Dart

of 20 percent. So far, we have built 10 trees, we want to or 10 models and then we try to see, we try to build a new, an 11th one. What we'll do is we will randomly exclude two trees when we generate a prediction for that 11th tree or that 11th model. By randomly excluding some models, by introducing this kind of randomness, it works as a form of regularization. Therefore, it helps a lot to make a model that generalizes 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 improve on different parts of these algorithms. One really successful application especially in the comparative predictive modeling world is the Xgboost. It is very scalable and it supports many loss functions. At the same time, is available in all major programming 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 loss functions. Another interesting case is the Gradient Boosting Machine from H2O. What's really interesting about this implementation is that it can handle categorical variables out of the box and it also comes with a real set of parameters where you can control the modeling process quite thoroughly. Another interesting case, which is also fairly new is the Catboost. What's really good about this is that it comes with the strong initial set of parameters. Therefore, you don't need to spend so much time tuning. As I mentioned before, this can be quite a time consuming process. It can also handle categorical variables out of the box.

Ultimately, I really like the Gradient Boosting Machine implementation 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 of this video.

## 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 to a 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 first 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 collect 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 meta model, and the models we run into, we call them base model or base learners.

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 architectural, 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

| A    |      |      |      |   |
|------|------|------|------|---|
| x0   | x1   | x2   | xn   | y |
| 0.17 | 0.25 | 0.93 | 0.79 | 1 |
| 0.35 | 0.61 | 0.93 | 0.57 | 0 |
| 0.44 | 0.59 | 0.56 | 0.46 | 0 |
| 0.37 | 0.43 | 0.74 | 0.28 | 1 |
| 0.96 | 0.07 | 0.57 | 0.01 | 1 |

| B    |      |      |      |   |
|------|------|------|------|---|
| x0   | x1   | x2   | xn   | y |
| 0.89 | 0.72 | 0.50 | 0.66 | 0 |
| 0.58 | 0.71 | 0.92 | 0.27 | 1 |
| 0.10 | 0.35 | 0.27 | 0.37 | 0 |
| 0.47 | 0.68 | 0.30 | 0.98 | 0 |
| 0.39 | 0.53 | 0.59 | 0.18 | 1 |

| C    |      |      |      |   |
|------|------|------|------|---|
| x0   | x1   | x2   | xn   | y |
| 0.29 | 0.77 | 0.05 | 0.09 | ? |
| 0.38 | 0.66 | 0.42 | 0.91 | ? |
| 0.72 | 0.66 | 0.92 | 0.11 | ? |
| 0.70 | 0.37 | 0.91 | 0.17 | ? |
| 0.59 | 0.98 | 0.93 | 0.65 | ? |

Train algorithm **0** on A and make predictions for B and C and save to **B1, C1**

Train algorithm **1** on A and make predictions for B and C and save to **B1, C1**

Train algorithm **2** on A and make predictions for B and C and save to **B1, C1**

| B1    |       |       |   |
|-------|-------|-------|---|
| pred0 | pred1 | pred2 | y |
| 0.24  | 0.72  | 0.70  | 0 |
| 0.95  | 0.25  | 0.22  | 1 |
| 0.64  | 0.80  | 0.96  | 0 |
| 0.89  | 0.58  | 0.52  | 0 |
| 0.11  | 0.20  | 0.93  | 1 |

| C1    |       |       |   |        |
|-------|-------|-------|---|--------|
| pred0 | pred1 | pred2 | y | Preds3 |
| 0.50  | 0.50  | 0.39  | ? | 0.45   |
| 0.62  | 0.59  | 0.46  | ? | 0.23   |
| 0.22  | 0.31  | 0.54  | ? | 0.99   |
| 0.90  | 0.47  | 0.09  | ? | 0.34   |
| 0.20  | 0.09  | 0.61  | ? | 0.05   |

Train algorithm **3** on B1 and make predictions for C1



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 store 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 repeat 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 its 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 data-set, 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 predictions from C1. And this is how we combine different models with stacking, to hopefully make better predictions 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 people not very experienced with Python hopefully can understand this.

## Stacking example

```
#split train data in 2 parts, training and validation.
training,valid,ytraining,yvalid = train_test_split(train,y,test_size=0.5)
#specify models
model1=RandomForestRegressor()
model2=LinearRegression()
#fit models
model1.fit(training,ytraining)
model2.fit(training,ytraining)
#make predictions for validation
preds1=model1.predict(valid)
preds2=model2.predict(valid)
#make predictions for test data
test_preds1=model1.predict(test)
test_preds2=model2.predict(test)
#Form a new dataset for valid and test via stacking the predictions
stacked_predictions=np.column_stack((preds1,preds2))
stacked_test_predictions=np.column_stack((test_preds1,test_preds2))
#specify meta model
meta_model=LinearRegression()
#fit meta model on stacked predictions
meta_model.fit(stacked_predictions,yvalid)
#make predictions on the stacked predictions of the test data
final_predictions=meta_model.predict(stacked_test_predictions)
```



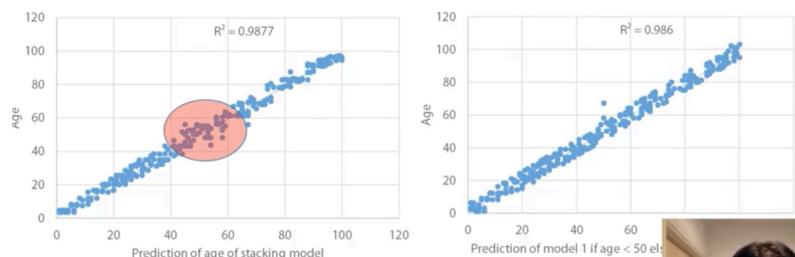
The main logic is that we will use two base learners on some input data, a random forest and a linear regression. And then, we will try to combine the results, starting with a meta learner, again, it will be linear regression.

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 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 set you may treat categorical features as one whole encoding. In another, you may just use label in coding, and the result will probably produce a model that is very different.

## Things to be mindful of

- With time sensitive data – respect time
- Diversity as important as performance
- Diversity may come from:
  - Different algorithms
  - Different input features
- Performance plateauing after N models
- Meta model is normally modest

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 initially, 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 affected by how many features you have in your data, how much diversity you managed 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 more models actually does not add that much value.

And because the meta model, the meta model will only use predictions of other models. 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 which are correlated with the target. And the only thing it needs to do 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 was 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 very interesting concept about stacking and extending it on multiple levels, called stack net

## StackNet

We can continue our discussion with StackNet. StackNet is a scalable meta modeling methodology that utilizes stacking to combine multiple models in a neural network architecture of multiple levels. It is scalable because within the same level, we can run all the models in parallel. It utilizes stacking because it makes use of this technique we mentioned before where we split the data, we make predictions on some 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 gave before with the students and the teacher, in order to understand what conceptually, in a real world, would need to add another layer. So in that example, we have a teacher that she was trying to combine the answers of different students and she was outputting an estimate of 17 under certain assumptions. We can make this example more interesting by introducing one more meta learner. Let's call him Mr. RF, who's also a physics teacher. Mr. RF believes that LR should have a bigger contribution to the ensemble because he has been doing private lessons with him and he knows 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 these predictions and make a different estimate. Whereas, the teachers could work it out and take an average, we could create or we can introduce a higher authority or another layer of modeling here. Let's call it the headmaster, GBM, in order to shop, make better 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 trust his physics teacher by outputting a 16.2 prediction. Why would this be of any use to people? I mean, isn't that already complicated? Why would we want to ever try something so complicated? I'm giving you an example of a competition my team used, four layer of stacking, in order to win. And we used two different sources of input data. We generated multiple models. Normally, exit boost and logistic regressions, and then we fed those into a four-layer architecture in order to get the top score. And although we could have escaped without using that fourth layer, we still need it up to level three in order to win. So you can understand 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 different transformations. We generated many models. We fed those models into a three-level architecture. I think we didn't need the third layer again. Probably, we could have escaped with only two levels but again, deep stacking was necessary in order to win. So there is your answer, deep stacking on multiple levels really helps you to win competitions. In the spirit of fairness and openness, there has been some criticism about large ensembles that maybe they don't have commercial value, they are confidentially expensive. I have to add three things on that. The first is, what is considered expensive today may not be expensive tomorrow and we have seen that, for example, with the deep learning, where with the advent of GPUs, they have become 100 times faster and now they have become again very, very popular. The other thing is, you don't need to always build very, very deep ensembles but still, small ensembles would still really help. So knowing how to do them can add value to businesses, again based on different assumptions about how fast they want the decisions, how much is the uplift you can see from stacking, which may vary, sometimes it's more, sometime is less. And generally, how much computing power they have. We can make a case that even stacking on multiple layers can be very useful. And the last point is that these are predictive modeling competitions so it is a bit like the Olympics. It is nice to be able to see the theoretical best you can get because this is how innovation takes over. This is how we move forward. We can express StackNet as a neural network. So normally, in a neural network, we have these architecture of hidden units where they are connected with input with the form of linear regression. So

actually, it looks pretty much like a linear regression. So whether you have a set of coefficients and you have a constant value where you call it bias in neural 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  $x_0$ ,  $x_1$ ,  $x_2$ ,  $x_3$ , 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 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 metal-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

## Ensembling Tips and Tricks

Now, I'll give you a few tips that are going to help you to make a good ensemble, or at least it help you to start with. And I mentioned before when I talked about stacking, what's quite important is to introduce diversity, and one way to do this is based on the algorithms you use. So an architecture that I have found quite useful is to always make two or three gradient boosted trees. And what I'm about to do here is I haven't either tried different implementations, or I try to make one with bigger depth, one with middle depth, and one with low depth, and then I try to tune the parameters around it in order to make them have as similar performance as possible. Again, hoping that the end result will be some models which are fairly diverse to each other.

Another thing that I like is, I'm using neural nets either from Keras or PyTorch. And again, what I try to do is, I try to make one which is fairly deep that take three hidden layers, and one which is middle, like two hidden layers, and one that has, let's say, only one hidden layer. Again, I try to diversify, I try to make them slightly different, in order to try and get new information. Then I use a few ExtraTrees or Random Forest, most of the time they work quite well, they normally add. I also add a few linear models or ridge regression. I also like

linear support vector machine from scikit-learn. KNN models tend to want quite nice value in many problems, which is surprising because if you look at them individually, they rarely have good performance by compared to an exit boost that they generally tend to add quite some value in a meta modeling context. I personally like factorization machines, I find them quite useful, especially libFM that factorizes all pairwise interactions. And if your data permits, so if your data is not so big, I also find useful support vector machines with some sort of non-linear kernel, like RBF. They don't work quite well, especially in regression. The other way you can introduce diversity is with the

## 1<sup>st</sup> level tips

- Diversity based on algorithms:
  - 2-3 gradient boosted trees (lightgb, xgboost, H2O, catboost)
  - 2-3 Neural nets ( keras, pytorch)
  - 1-2 ExtraTrees/Random Forest (sklearn)
  - 1-2 linear models as in logistic/ridge regression, linear svm (sklearn)
  - 1-2 knn models (sklearn)
  - 1 Factorization machine (libfm)
  - 1 svm with nonlinear kernel if size/memory allows (sklearn)
- Diversity based on input data:
  - Categorical features: One hot, label encoding, target encoding
  - Numerical features: outliers, binning, derivatives, percentiles, s
  - Interactions : col1\*/+-col2, groupby, unsupervised



transformations you make in your own input data, whereas, you can actually run exactly the same models just by having slightly different input data, is enough to generate diversity. So, what we look at for categorical features are train one hot encoding, label encoding as in replacing the categorical values with an index. I can use likelihood encoding or target encoding, or I can replace categories with frequency or counts. For numerical features, I try to take care of outliers or not. I bin the variables from X to Z, from Z to all, and so on. I use derivatives which is a way to smoothen your variables or I could use percentiles or scaling, these are different ways I use to

change my numerical features in different models. And then I also explore interactions, like column one, multiple column two, or column one plus column two.

I can go up to three or four levels so explore all possible interactions. Another way to explore interactions is with groupby statements where you say, given all categories of a categorical feature, compute the average of another variable, for example. In certain situations, this works quite well. The other thing that I would do is, I would try and supervise techniques like k-means, or SVM, or PCA, or numerical features. Again, it tends to add value quite often.

Now, on every other layer, what you need to keep in mind is that you need to make your algorithms smaller or shallower, more constrained. What does this mean? In gradient boosting trees, means you need to put very small depth, like two or three. Linear models, you need to put high regularization. Extra Trees, just don't make them too big, they tend to work quite well. Shallow neural networks, again, I normally put one layer maximum two layers here, with not that many

## Subsequent level tips

- Simpler (or shallower) Algorithms:
  - ❑ gradient boosted trees with small depth (like 2 or 3)
  - ❑ Linear models with high regularization
  - ❑ Extra Trees
  - ❑ Shallow networks (as in 1 hidden layer )
  - ❑ knn with BrayCurtis Distance
  - ❑ Brute forcing a search for best linear weights based on cv
- Feature engineering:
  - ❑ pairwise differences between meta features
  - ❑ row-wise statistics like averages or stds
  - ❑ Standard feature selection techniques
- For every 7.5 models in previous level we add 1 in meta (emp
- Be mindful of target leakage

hidden neurons. We can try KNN with BrayCurtis distance or

sometimes it's actually better to brute force the best linear weights we could use. Actually, in a cross-validation try to find the best linear weights, again, symmetric. We can also deploy different feature engineering in this subsequent level. One thing we could do is, we could create pairwise differences between the model's predictions, and this can help because they tend to be quite correlated. So when you create the differences, you essentially force the model to focus on what each new model brings. And you can also create row-wise statistics like averages or standard deviations of all the models. This is almost like an ensemble, you create some ensemble features yourself. You can also deploy standard feature selection techniques, so any techniques you would use to find which features are important. You could use it here to find which models are important and exclude them. A rule which empirically, let's say as a rule of thumb, I've found to work quite well. I mean, not with 100 percent confidence but as a general idea, is that for every seven point five models in one layer, we add one model in the subsequent layer. So if we have seven models, we'll have one metamodel. If we have 15 models, then we will have two metamodels and so on. What we need to be very mindful is that, even though we use this hold-out mechanism, we still might introduce leakage. How we can control this is by selecting the right K, when I mentioned K in the cross-validation. So when we select a very high value there, this means that each model would use more training data when it makes predictions and therefore it might not generalize very well. At the same time, it will exhaust all information about the training data. So again, there is this bias variance threshold where you try to find. So, there is no easy way to spot a mistake here. Normally, you have a test data set, and if you see in your cross-validation that you have a next improvement, that in your test data you don't see it, then you need to go back and try to reduce the number of K-Folds. And hopefully, this will generalize better, at least that's a way that has worked in practice.

I'll list a few software which you can use for stacking, one is StackNet which is the product of my research. You can give it a shot if you want. Another thing which you can try, is Stacked ensembles from H2O.

There is also a new software called, Xcessiv and Python where you can also create quite diverse, and large ensembles. A few more things to

know about StackNet if you want to use it, is that it now supports many of the common machine learning tools we use, like xgboost, lightgbm, H2O. So, you can pretty much have available all the great tools in order to build a strong ensemble. An interesting addition which we didn't have much chance to discuss but nevertheless, it is quite interesting, is that you can run classifiers in a regression problem, and vice versa. This means that instead of predicting AIDS, I could be predicting, will this person live more than 50 years old or not. And this tends to work quite well because it makes the model focus in certain areas, and meta-model will be able to utilize this information to make better predictions for AIDS. So, this tends to work quite well. I've found it useful quite many often. So, this is something you should explore. Generally, the software has already many top 10 solutions, and not just by me. So, it has been tested. And in the examples section, I think there is a really interesting example with a very popular kind of competition which was hosted by Amazon. And this example uses StackNet, and you can see how you can get with top 10. But in principle, this is a very nice competition. It doesn't have very big data, nor very small. You can try lots of transformations especially with the categorical data. And it is very good place to start. The other thing I wanted to say is that StackNet has also an educational flavor. I made it as such, I have this focus in mind. So, if you go to the parameter section where it leaves all the different tools, you can find sections where they tell you which parameters are the most important. This is based on my experience. For example, in xgboost, num\_round is important, eta is important. You can take, then, this information and use it even outside StackNet. For example, if you want to use this, they're from Python. Because the parameters are generally the same. So, if you don't know where to start, how to look here, see which these parameters are important. And focus on them in order to try and get a good resolution. Before we close this session, there are a few things I'd like to tell you. Go out there, apply what you've learned. There is no such thing as learning only theoretically. The best thing to learn is to bleed on the battlefield. Choose a competition. You may start with some which are named as tutorials. And then, you can go on into the real competitions. This is how you've learned. You need to get more practical experience, obviously. Don't be demoralized if you see there's still a gap with the top people, because it does take some time to adjust. You need to learn the dynamics. You need to understand how

you need to work, where to optimize your work, how maximize the intensity. So, it takes a bit of time. But you'll get there. My main point is don't get disappointed, you'll definitely get there. Something that has always helped me is to save my code. Let's say at the end of it it's a silence. That is good. Because I can then take home this code in the next competition, and try and improve it. So, this helps to gradually build a much stronger pipeline, and at the same time, it saves time. Something that has helped me personally is to seek collaborations.

These generally, I think, there are two elements into it. One is, you definitely improve your result, because every person seizes the problem from different angles, is able to extract different information. Therefore, when you join forces the score is better, but it's also more fun. And since you're doing it, you might as well enjoy it. And the other thing I need to highlight is generally, you need to be connected with forums, and codes, and kernels, because there might be tips, there might be some cutting-edge solutions that come out, and they can significantly sift a leader-board. So generally, you need to keep reading, and have that in mind. This is the last video in a series where we have examined and sample methods will look that simple and sampling. Then we went to bagging, boosting, stacking, multi-layer stacking. Hopefully, you found this useful.

- [Kaggle ensembling guide at MLWave.com \(overview of approaches\)](#)
- [StackNet – a computational, scalable and analytical meta modelling framework \(by KazAnova\)](#)
- [Heamy – a set of useful tools for competitive data science \(including ensembling\)](#)