

# Week 1

## Feature preprocessing

Each type of feature has its own ways to be preprocessed in order to improve quality of the model. In other words, choice of preprocessing matter, it depends on the model we're going to use. For example, let's suppose that target has nonlinear dependency on the pclass feature. Pclass linear of 1 usually leads to

target of 1, 2 leads to 0, and 3 leads to 1 again. Clearly, because this is not a linear dependency, linear model cannot get a good result here. So in order to improve a linear model's quality, we would want to preprocess pclass feature in some way. For example, with the so-called one-hot-encoding. The linear model will fit much better now than in the previous case.

pclass	1	2	3
target	1	0	1

pclass	pclass==1	pclass==2	pclass==3
1	1		
2		1	
3			1

However, random forest does not require this feature to be transformed at all. Random forest can easily put each pclass in separately and predict fine probabilities. So, that was an example of preprocessing. The second reason why we should be aware of different feature types is to ease generation of new features. Feature types differ in this and comprehends in common feature generation methods. While gaining an ability to improve your model through them. Also understanding of basics of feature generation will aid you greatly in upcoming advanced feature topics from our course.

As in the first point, understanding of a model here can help us to create useful features. Let me show you an example. Say, we have to predict the number of apples a shop will sell each day next week and we already have a couple of months sales history as training data.

Let's consider that we have an obvious linear trend through out the data and we want to inform the model about it. To provide you a visual example, we prepare the second table with last days from train and first days from test.

One way to help model to utilize linear trend is to add feature indicating the week number passed. With this feature, linear model can successfully find an existing linear and dependency.

Week	Number of apples
1	42
2	46
3	52
4	58
5	64

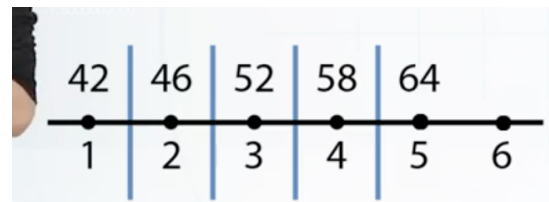
Day	Week	Number of apples
33	5	45
34	5	72
35	5	81
36	6	?
37	6	?

On the other hand, a gradient boosted decision tree will use this feature to calculate something like mean target value for each week. Here, I calculated mean values manually and placed them in the data frame. We're going to predict number of apples for the sixth week.

Note that we indeed have linear trend here. So let's plot how a gradient within the decision tree will split the week feature.

As we do not train Gradient Boosting decision tree on the sixth week, it will not put splits between the fifth and the sixth weeks, then, when we will predict the numbers for the 6th week, the model will end up using the value from the 5th week. As

we can see unfortunately, no users shall land their linear trend here. And vice versa, we can come up with an example of generated feature that will be beneficial for gradient boosting decisions tree and useless for linear model.



So this example shows us, **that our approach to feature generation should rely on understanding of employed model.** To summarize this video, first, **feature preprocessing** is necessary instrument you have to use to adapt data to your model. Second, **feature generation** is a very powerful technique which can aid you significantly. And at last, both feature preprocessing and feature generation **depend on the model** you are going to use. So these three topics, in connection to **feature types**, will be general theme of the next videos. We will thoroughly examine most frequent methods which you can be able to incorporate in your solutions.

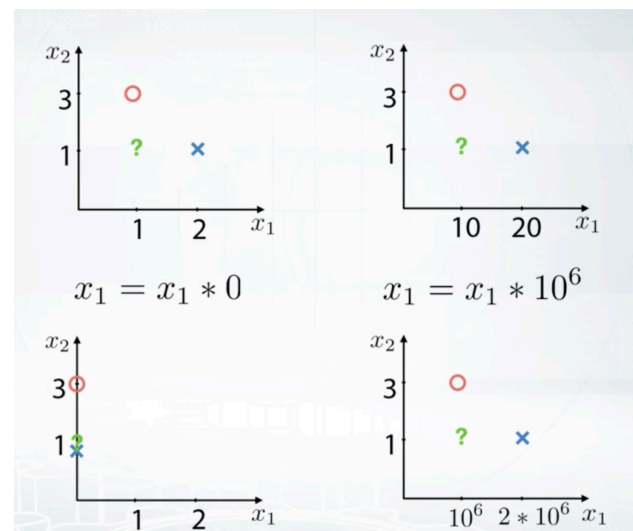
## Numeric Preprocessing

Here, we will cover basic approach as to feature preprocessing and feature generation for numeric features. We will understand how **model choice impacts feature preprocessing**. We will identify the **preprocessing methods** that are used most often, and we will discuss feature generation and go through several examples. Let's start with preprocessing.

First thing you need to know about handling numeric features is that **there are models which do and don't depend on feature scale**. For now, we will broadly divide all models into tree-based models and non-tree-based models. For example, decision trees classifier tries to find the most useful split for each feature, and it won't change its behavior and its predictions if we multiply the features by a constant and to retrain the model.

On the other side, there are **models which depend on these kind of transformations**. The model based on nearest neighbors, linear models, and neural network. Let's consider the following example. We have a binary classification task with two features. The object in the picture belong to different classes. The red circle to class zero, and the blue cross to class one, and finally, the class of the green object is unknown. Here, we will use a one nearest neighbor's model to predict the class of the green object. We will measure distance using square distance.

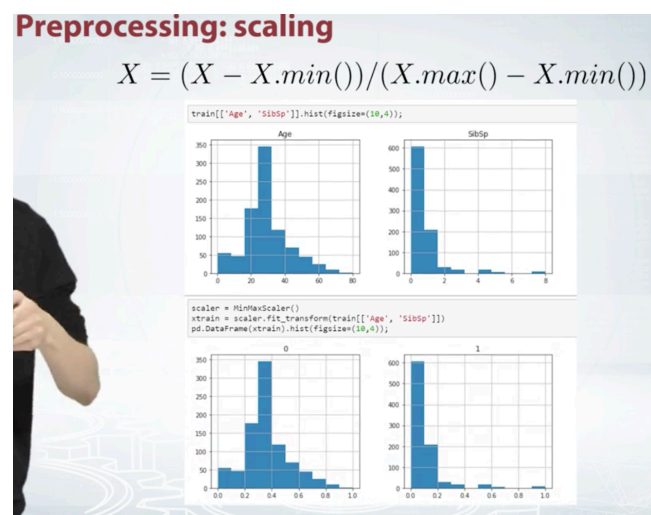
Now, if we calculate distances to the red circle and to the blue cross, we will see that our model will predict class one for the green object because the blue cross of class one is much closer than the red circle. But if we multiply the first feature by 10, the red circle will became the closest object, and we will get an opposite prediction. Let's now consider two extreme cases.



What will happen if we multiply the first feature by zero and by one million? If the feature is multiplied by zero, then every object will have feature relay of zero, which results in KNN ignoring that feature. On the opposite, if the feature is multiplied by one million, slightest differences in that features values will impact prediction, and this will result in KNN **favoring that feature over all others**.

Great, but what about other models? Linear models are also experience difficulties with differently scaled features. First, we want regularization to be applied to linear models coefficients for features in equal amount. But in fact, regularization impact turns out to be proportional to feature scale. And second, gradient descent methods can go crazy without a proper scaling. Due to the same reasons, neural networks are similar to linear models in the requirements for feature preprocessing.

**It is important to understand that different features scalings result in different models quality.** In this sense, it is just another hyper parameter you need to optimize. The easiest way to do this is to rescale all features to the same scale. For example, to make the minimum of a feature equal to 0 and the maximum equal to 1, you can achieve this in two steps. First, we subtract the minimum value. And then, we divide the difference by the range. It can be done with MinMaxScaler from sklearn. Let's illustrate this with an example. We apply the so-called MinMaxScaler to two features from the detaining dataset, Age and SibSp. Looking at histograms, we see that the features have different scale, age is between 0 and 80, while SibSp is between 0 and 8. Let's apply MinMaxScaling and see what it will do. Indeed, we see that after this transformation, both age and SibSp features were successfully converted to the same range of (0,1).



Note that distributions of values which we observe from the histograms didn't change. To give you

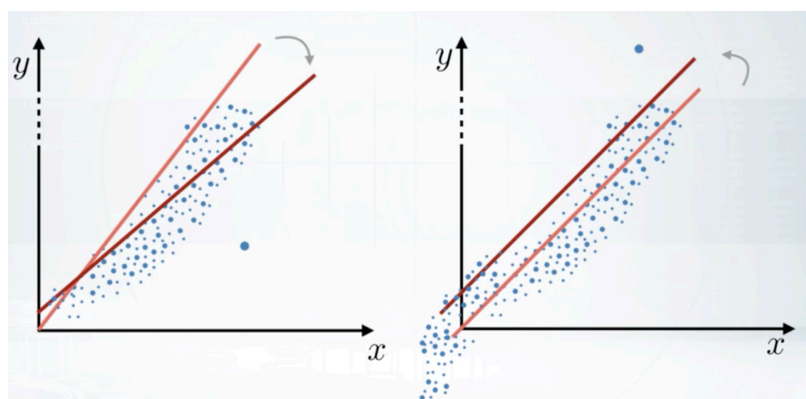
another example, we can apply a scalar named StandardScaler in sklearn, which basically first subtracts mean value from the feature, and then divides the result by feature standard deviation. In this way, we'll get standardized distribution, with a mean of 0 and standard deviation of 1. After either of MinMaxScaling or StandardScaling transformations, features impacts on non-tree-based models will be roughly similar.

```
Preprocessing: scaling  
  
1. To [0,1]  
sklearn.preprocessing.MinMaxScaler  
X = (X - X.min())/(X.max() - X.min())  
  
2. To mean=0, std=1  
sklearn.preprocessing.StandardScaler  
X = (X - X.mean())/X.std()
```

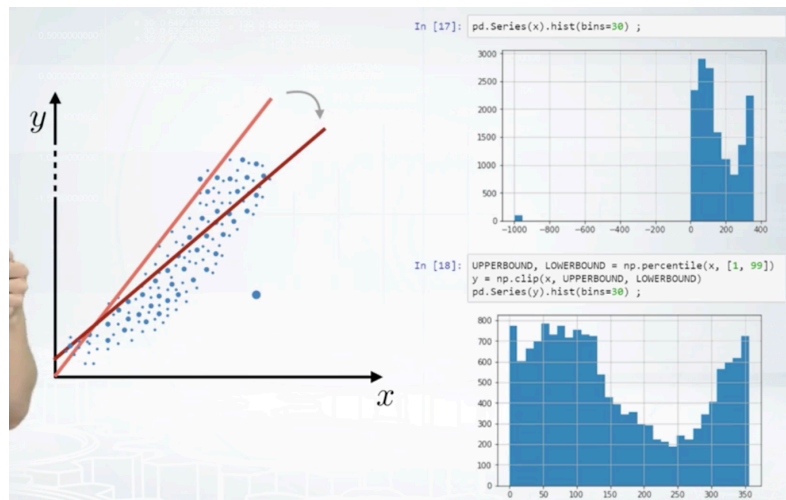
Even more, if you want to use KNN, we can go one step further and recall that the bigger the feature is, the more important it will be for KNN. So, we can optimize scaling parameter to boost features which seems to be more important for us and see if this helps.

When we work with linear models, there is another important moment that influences model's results. I'm talking about outliers. For example, in this plot, we have one feature, X, and a target variable, Y. If you fit a simple linear model, its predictions can look just like the red line. But if you do have one outlier with X feature equal to some huge value, predictions of the linear model will look more like the purple line. The same holds, not only for features values, but also for target values. For example, let's imagine we have a model trained on the data with target values between 0 and 1. Let's think what happens if we add a new sample in the training data with a target value of 1,000. When we retrain the model, the model will predict abnormally high values. Obviously, we have to fix this somehow. To protect linear models from outliers, we can clip features values

between two chosen values of lower bound and upper bound. We can choose them as some percentiles of that feature. For example, first and 99s percentiles. This procedure of clipping is well-known in financial data and it is called winsorization.



Let's take a look at this histogram for an example. We see that the majority of feature values are between zero and 400. But there is a number of outliers with values around -1,000. They can make life a lot harder for our nice and simple linear model. Let's clip this feature's value range, and to do so, first, we will calculate lower bound and upper bound values as features values at first and 99th percentiles. After we clip the features values, we can see that features distribution looks fine, and we hope now this feature will be more useful for our model.



Another effective preprocessing for numeric features is the **rank transformation**. Basically, it sets spaces between proper assorted values to be equal. This transformation, for example, can be a better option than MinMaxScaler if we have outliers, because rank transformation will move the outliers closer to other objects. Let's understand rank using this example. If we apply a rank to the sorted of array, it will just change values to their indices. Now, if we apply a rank to the not-sorted array, it will sort this array, define mapping between values and indices in this sorted of array, and apply this mapping to the initial array. **Linear models, KNN, and neural networks can benefit from this kind of transformation if we have no time to handle outliers manually.** Rank can be imported as a random data function from scipy. One more important note about the rank transformation is that to apply it to the test data, you need to store the created mapping from features values to their rank values. Or alternatively, you can concatenate, train, and test data before applying the rank transformation.

- `rank([-100, 0, 1e5]) == [0, 1, 2]`
- `rank([1000, 1, 10]) = [2, 0, 1]`

There is one more example of numeric features preprocessing which often helps non-tree-based models and especially **neural networks**. You can apply **log transformation** through your data, or there's another possibility. You can extract a square root of the data. Both these transformations can be useful because they drive too big values closer to the features' average value. Along with this, the values near 0 are becoming a bit more distinguishable. Despite the simplicity, one of these transformations can improve your neural network's results significantly.

```
1.Log transform:
np.log(1 + x)

2.Raising to the power < 1:
np.sqrt(x + 2/3)
```

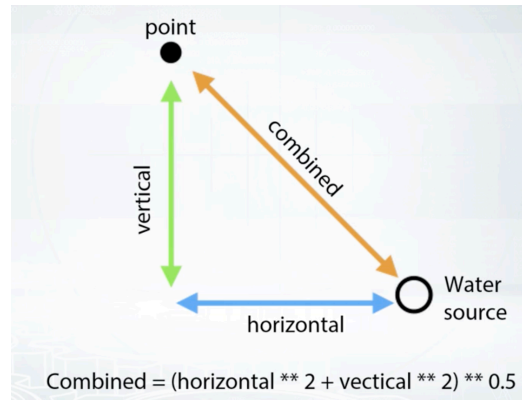
Another important moment which holds true for all preprocessing is that sometimes it is beneficial to train a model on **concatenated data frames produced by different preprocessing**, or to **mix models trained on differently-processed data**. Again, **linear models, KNN, and neural networks** can benefit hugely from this.

To this end, we have discussed numeric feature preprocessing, how model choice impacts feature preprocessing, and what are the most commonly used preprocessing methods.

Let's now move on to feature generation. Feature generation is a process of creating new features using knowledge about the features and the task. It helps us by making model training more simple and effective. Sometimes, we can engineer these features using prior knowledge and logic. Sometimes we have to dig into the data, create and check hypothesis, and use this derived knowledge and our intuition to derive new features. Here, we will discuss feature generation with prior knowledge, but as it turns out, an ability to dig into the data and derive insights is what makes a good competitor a great one. We will thoroughly analyze and illustrate this skill in the next lessons on exploratory data analysis. For now, let's discuss examples of feature generation for numeric features. First, let's start with a simple one.

If you have columns: Real Estate price and Real Estate squared area in the dataset, we can quickly add one more feature, price per meter square. Easy, and this seems quite reasonable. Or, let me give you

another quick example from the Forest Cover Type Prediction dataset. If we have a horizontal distance to a water source and the vertical difference in heights within the point and the water source, we as well may add combined feature indicating the direct distance to the water from this point. Among other things, it is useful to know that adding, multiplications, divisions, and other features interactions can be of help not only for linear models. For example, although gradient boosting decision tree is a very powerful model, it still experiences difficulties with approximation of multiplications and divisions. And adding size features explicitly can lead to a more robust model with less amount of trees. The third example of feature generation for numeric features is also very interesting. Sometimes, if we have prices of products as a feature, we can add new feature indicating fractional part of these prices. For example, if some product costs 2.49, the fractional part of its price is 0.49. This feature can help the model utilize the differences in people's perception of these prices.



price	fractional_part
0.99	0.99
2.49	0.49
1.0	0.0
9.99	0.99

Also, we can find similar patterns in tasks which require distinguishing between a human and a robot. For example, if we will have some kind of financial data like auctions, we could observe that people tend to set round numbers as prices, and there are something like 0.935, blah, blah, blah, very long number here. Or, if we are trying to find spambots on social networks, we can be sure that no human ever write messages with an exact interval of one second.

Great, these three examples should have provided you an idea that creativity and data understanding are the keys to productive feature generation. Let's summarize this up. In this video, we have discussed numeric features.

**First**, the impact of feature preprocessing is different for different models. Tree-based models don't depend on scaling, while non-tree-



based models usually depend on them. **Second**, we can treat scaling as an important hyper parameter in cases when the choice of scaling impacts predictions quality. And at **last**, we should remember that feature generation is powered by an understanding of the data.

## Conclusion

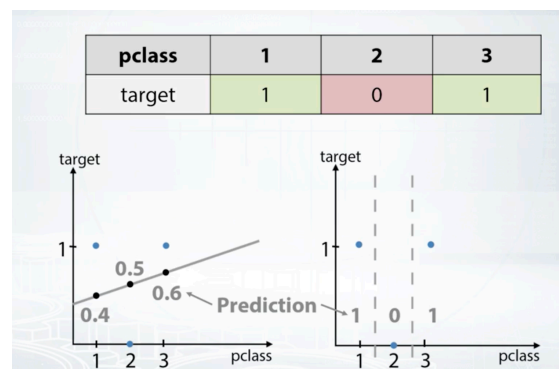
1. Scaling and Rank for numeric features:
  - a. Tree-based models doesn't depend on them
  - b. Non-tree-based models hugely depend on them
2. Most often used preprocessings are:
  - a. MinMaxScaler - to [0,1]
  - b. StandardScaler - to mean==0, std==1
  - c. Rank - sets spaces between sorted values to be equal
  - d.  $\text{np.log}(1+x)$  and  $\text{np.sqrt}(1+x)$
3. Feature generation is powered by:
  - a. Prior knowledge
  - b. Exploratory data analysis

## Ordinal And Categorical

In particular, what kind of pre-processing will be used for each model type of them? What is the difference between categorical and ordinal features and how we can generate new features from them?

First, let's look at several rows from the Titanic dataset and find categorical features here. Their names are: Sex, Cabin and Embarked. These are usual categorical features but there is one more special, the Pclass feature. Pclass stands for ticket class, and has three unique values: 1, 2, and 3. It is ordinal, in other words, ordered categorical feature. This basically means that it is ordered in some meaningful way. For example, if the first class was more expensive than the second, or the more the first should be more expensive than the third.

We should make an important note here about differences between ordinal and numeric features. If Pclass would have been a numeric feature, we could say that the difference between first, and the second class is equal to the difference between second and the third class, but because Pclass is ordinal, we don't know which difference is bigger. As with numeric features, we can't sort and integrate(?) an ordinal feature the other way, and expect to get similar performance. Another example for ordinal feature is a driver's license type. It's either A, B, C, or D. Or another example, level of education, kindergarten, school, undergraduate, bachelor, master, and doctoral. These categories are sorted in increasingly complex order, which can prove to be useful. The simplest way to encode a categorical feature is to map it's unique values to different numbers. Usually, people referred to this procedure as **label encoding**. **This method works fine with trees** because tree-methods can split feature, and extract most of the useful values in categories on its own. Non-tree-based-models, on the other side, usually can't use this feature effectively. And if you want to train linear model kNN on neural network, you need to treat a categorical feature differently. To illustrate this, let's remember example we had in the beginning of this topic. What if Pclass of 1 usually leads to the target of 1, Pclass of 2 leads to 0, and Pclass of 3 leads to 1. This dependence is not linear, and linear model will be confused. And indeed, here, we can put linear models predictions, and see they all are around 0.5. This looks kind of sad but trees on the other side, will just make two splits selecting each unique value and reaching it independently. Thus, decision trees could achieve much better score here using these feature. Let's take another categorical feature and again, apply label encoding. Let this be the feature Embarked. Although, we didn't have to encode the previous feature Pclass before using it in the model. Here, we definitely need to do this with embarked. It can be achieved in several ways. **First**, we can apply encoding in the alphabetical or sorted order. Unique way to solve of this feature namely S, C, Q, can be encoded as 2, 1, 3. This is called label encoder from sklearn works by default. The **second** way is also label encoding but slightly different. Here, we encode a categorical



target of 1, Pclass of 2 leads to 0, and Pclass of 3 leads to 1. This dependence is not linear, and linear model will be confused. And indeed, here, we can put linear models predictions, and see they all are around 0.5. This looks kind of sad but trees on the other side, will just make two splits selecting each unique value and reaching it independently. Thus, decision trees could achieve much better score here using these feature. Let's take another categorical feature and again, apply label encoding. Let this be the feature Embarked. Although, we didn't have to encode the previous feature Pclass before using it in the model. Here, we definitely need to do this with embarked. It can be achieved in several ways. **First**, we can apply encoding in the alphabetical or sorted order. Unique way to solve of this feature namely S, C, Q, can be encoded as 2, 1, 3. This is called label encoder from sklearn works by default. The **second** way is also label encoding but slightly different. Here, we encode a categorical

feature by order of appearance. For example, s will change to 1 because it was met first in the data. Second then c, and we will change c to 2. And the last is q, which will be changed to 3. This can make sense if all were sorted in some meaningful way. This is the default behavior of `pandas.factorize` function.

K	
embarked	1. Alphabetical (sorted)
S	[S,C,Q] -> [2, 1, 3]
C	
S	<code>sklearn.preprocessing.LabelEncoder</code>
S	
S	2. Order of appearance
Q	[S,C,Q] -> [1, 2, 3]
S	
S	<code>Pandas.factorize</code>
S	
C	
S	
S	

The **third** method that I will tell you about is called **frequency encoding**. We can encode this feature via mapping values to their frequencies. If 30 percent for

[S,C,Q] -> [0.5, 0.3, 0.2]

```
encoding = titanic.groupby('Embarked').size()
encoding = encoding/len(titanic)
titanic['enc'] = titanic.Embarked.map(encoding)

from scipy.stats import rankdata
```

us embarked is equal to c and 50 to s and the rest 20 is equal to q. We can change this values accordingly: c to 0.3, s to 0.5, and q to 0.2. This will preserve some information about values distribution, and can help both linear and tree models. The former methods, can find this feature useful if value frequency is correlated with target value. While the latter models can help with less number of splits because of the same reason.

There is another important moment about frequency encoding. If you have multiple categories with the same frequency, they won't be distinguishable in this new feature. We might apply **rank operation here in order to deal with such ties**. It is possible to do like this. There are other ways to do label encoding, and I definitely encourage you to be creative in constructing them.

We just discussed label encoding, frequency encoding, and why this works fine for tree-based-methods. But we also have seen that linear models can struggle with label encoded feature. The way to identify categorical features to non-tree-based-models is also quite straightforward. We need to make new code for each unique value in the future, and put one in the appropriate place. Everything else will be zeroes. This method is called, **one-hot encoding**. Let's see how it

works on this quick example. So here, for each unique value of Pclass feature, we just created a new column. As I said, this works well for **linear methods, kNN, or neural networks**. Furthermore, one-hot encoded feature is already scaled because minimum this feature is 0, and maximum is 1.

**Note that** if you have a few important numeric features, and hundreds of binary features are used by one-hot encoding, it could become difficult for tree-methods to use first ones efficiently. More precisely, tree-methods will slow down, not always improving their results. Also, it's easy to imply that if

One-hot encoding

pclass	pclass==1	pclass==2	pclass==3
1	1		
2		1	
1	1		
3			1

`pd.get_dummies`, `sklearn.preprocessing.OneHotEncoder`

categorical feature has too many unique values, we will add too many new columns with a few non-zero values. To store these new array efficiently, we must know about sparse matrices. In a nutshell, instead of allocating space in RAM for every element of an array, we can store only non-zero elements and thus, save a lot of memory. Going with sparse matrices makes sense if number of non-zero values is far less than half of all the values. Sparse matrices are often useful when they work with categorical features or text data. Most of the popular libraries can work with these sparse matrices directly namely, XGBoost, LightGBM, sklearn, and others.

After figuring out how to pre-processed categorical features for tree based and non-tree based models, we can take a quick look at feature generation.

One of the most useful examples of **feature generation is feature interaction between several categorical features**. This is usually useful for **non tree based models** namely, linear model, kNN. For example, let's hypothesize that target depends on both Pclass feature, and sex feature. If this is true, linear model could adjust its predictions for every possible combination of these two features, and get a better result. How can we make this happen? Let's add this interaction by simply concatenating strings from both columns and one-hot encoding it. Now linear model can find optimal coefficient for every interaction and improve. Simple and effective.

More on features interactions will come in the following weeks especially, in advanced features topic.

Now, let's summarize this features.

**First**, ordinal is a special case of categorical feature but with values sorted in some meaningful order.

**Second**, label encoding basically replaces unique values of categorical features with numbers.

**Third**, **frequency encoding**, maps unique values to their frequencies.

**Fourth**, label encoding and frequency encoding are often used for tree-based methods.

**Fifth**, One-hot encoding is often used for non-tree-based-methods.

And **finally**, applying One-hot encoding to combinations of categorical features allows non-tree-based-models to take into consideration interactions between features, and improve.

We just sorted out it feature pre-process for categorical features, and took a quick look on feature generation. Now, you will be able to apply these concepts in your next competition and get better results.

### Categorical features

pclass	sex	pclass_sex
3	male	3male
1	female	1female
3	female	3female
1	female	1female

↓

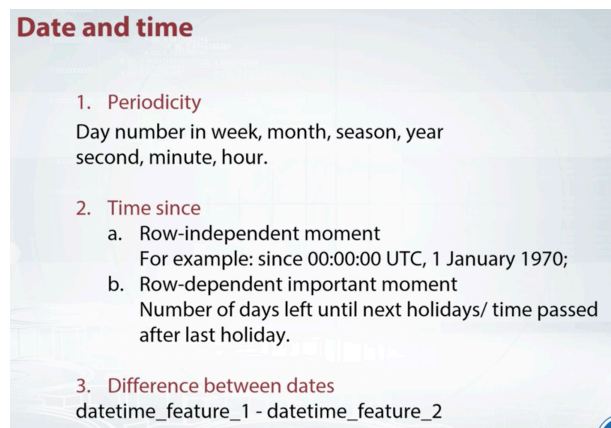
Pclass_sex==					
1male	1female	2male	2female	3male	3female
				1	
	1				
					1
	1				

1. Values in ordinal features are sorted in some meaningful order
2. Label encoding maps categories to numbers
3. Frequency encoding maps categories to their frequencies
4. Label and Frequency encodings are often used for tree-based models
5. One-hot encoding is often used for non-tree-based models
6. Interactions of categorical features can help linear models and KNN

## DateTime and Coordinate

We will discuss basic feature generation approaches for datetime and coordinate features. They both differ significantly from numeric and categorical features. Because we can interpret the meaning of datetime and coordinates, we can come up with specific ideas about future generation which we'll discuss here. Let's start with date-time.

Date-time is quite an interesting feature because it isn't on the linear nature, it also has several different parts like year, day or week. Most new features generated from date-time can be divided into two categories. The first one, time moments in a period, and the second one, time passed since particular event. First one is very simple. We can add features like second, minute, hour, day in a week, in a month, on the year and so on and so forth.



**Date and time**

- 1. Periodicity**  
Day number in week, month, season, year  
second, minute, hour.
- 2. Time since**
  - a. Row-independent moment  
For example: since 00:00:00 UTC, 1 January 1970;
  - b. Row-dependent important moment  
Number of days left until next holidays/ time passed after last holiday.
- 3. Difference between dates**  
`datetime_feature_1 - datetime_feature_2`

This is useful to capture repetitive patterns in the data. If we know about some non-common periods which influence the data, we can add them as well. For example, if we are to predict efficiency of medication, but patients receive pills one time every three days, we can consider this as a special time period.

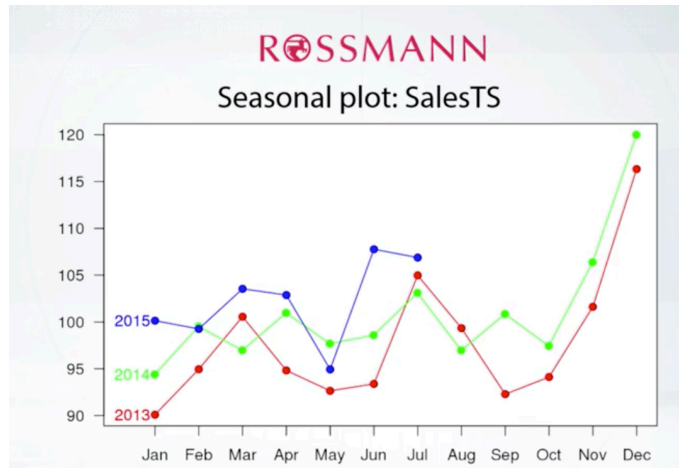
Now, Time since particular event.

This event can be either row-independent or row-dependent. In the first case, we just calculate time passed from one general moment for all data. For example, from here to thousand. Here, all samples will become comparable between each other on one time scale.

As the second variant of `time_since_particular_event`, that date will depend on the sample we are calculating this for. For example, if we are to predict sales in a shop, like in the ROSSMANN's store sales competition. We can add the number of days passed since the last holiday, weekend or since the last sales campaign, or maybe the

number of days left to these events. So, after adding these features, our data\_frame can look like this.

Date is obviously a date, and sales are the target of this task, while other columns are generated features. Week day feature indicates which day in the week is this, daynumber\_since\_year\_2014 indicates how many days have passed since January 1st, 2014. is\_holiday is a binary feature indicating whether this day is a holiday and days\_till\_holidays indicate how many days are left before the closest holiday. Sometimes we have several date-time columns in our data. The most straightforward idea here is to subtract one feature from another. Or perhaps subtract the generated features, like the one, we just have discussed.



Periodicity. «Time since»

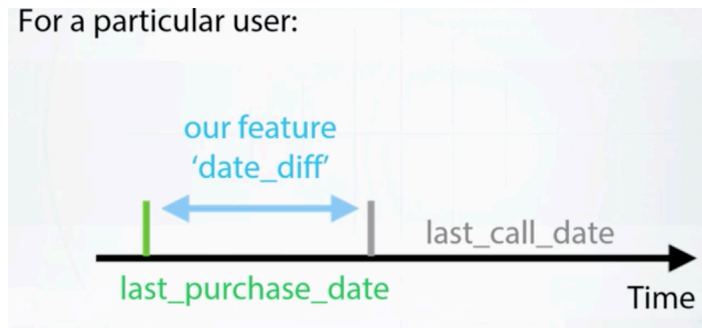
Date	week day	daynumber_since_year_2014	is_holiday	days_till_holidays	sales
01.01.14	5	0	True	0	1213
02.01.14	6	1	False	3	938
03.01.14	0	2	False	2	2448
04.01.14	1	3	False	1	1744
05.01.14	2	4	True	0	1732
06.01.14	3	5	False	9	1022

Time moment inside the period or time passed in row dependent events. One simple example of such generation can be found in churn prediction task. Basically churn prediction is about estimating the likelihood that customers will churn.

We may receive a valuable feature here by subtracting user registration date from the date of some action of his, like purchasing a product, or calling to the customer service. We can see how this works on this data\_frame. For every user, we know last\_purchase\_date and last\_call\_date. Here we add the difference between them as new feature named

user_id	registration_date	last_purchase_date	last_call_date	date_diff	churn
14	10.02.2016	21.04.2016	26.04.2016	5	0
15	10.02.2016	03.06.2016	01.06.2016	-2	1
16	11.02.2016	11.01.2017	11.01.2017	1	1
20	12.02.2016	06.11.2016	08.02.2017	94	0

date\_diff. For clarity, let's take a look at this figure. For every user, we have his last\_purchase\_date and his last\_call\_date. Thus, we can add date\_diff feature which indicates number of days between these events. Note that after generation features from date-time, you usually will get either numeric features like time passed since the year 2000, or categorical features like day of week. And these features now need to be treated accordingly with necessary pre-processing we have discussed earlier.



Now having discussed feature generation for date-time, let's move on to feature generation for coordinates. Let's imagine that we're trying to estimate the Real Estate price. Like in the Deloitte competition named Western Australia Rental Prices, or in the Sberbank Russian Housing Market competition.

1. **Periodicity**  
Day number in week, month, season, year  
second, minute, hour.
2. **Time since**
  - a. Row-independent moment  
For example: since 00:00:00 UTC, 1 January 1970;
  - b. Row-dependent important moment  
Number of days left until next holidays/ time passed after last holiday.
3. **Difference between dates**  
`datetime_feature_1 - datetime_feature_2`

Generally, you can calculate distances to important points on the map.

Keep this wonderful map. If you have additional data with infrastructural buildings, you can add as a feature distance to the nearest shop to the second by distance hospital, to the best school in the neighborhood and so on.

If you do not have such data, you can extract interesting points on the map from your trained test data. For example, you can do a new map to squares, with a grid, and within each square, find the most expensive flat, and for every other object in this square, add the distance to that flat. Or you can organize your data points into clusters, and then use centers of clusters as such important points.

Or again, another way. You can find some special areas, like the area with very old buildings and add distance to this one.



Another major approach to use coordinates is to calculate aggregated statistics for objects surrounding area. This can include number of lets around this particular point, which can then be interpreted as areas or polarity. Or we can add mean realty price, which will indicate how expensive area around selected point is.

Both distances and aggregate statistics are often useful in tasks with coordinates.

One more trick you need to know about coordinates, that if you train decision trees from them, you can add slightly rotated coordinates is new features. And this will help a model make more precise selections on the map.

It can be hard to know what exact rotation we should make, so we may want to add all rotations to 45 or 22.5 degrees.

Let's look at the next example of a relative price prediction.

Here the street is dividing an area in two parts. The high priced district above the street, and the low priced district below it. If the street is slightly rotated, trees will try to make a lot of space here. But if we will add new coordinates in which these two districts can be divided by a single split, this will hugely facilitate the rebuilding process.

Great, we just summarize the most frequent methods used for future generation from date-time and coordinates. For date-time, these are applying periodicity, calculates in time passed since particular event, and engine differences between two date-time features.

For coordinates, we should recall extracting interesting samples from trained test data, using places from additional data, calculating distances to centers of clusters, and adding aggregated statistics for surrounding area.

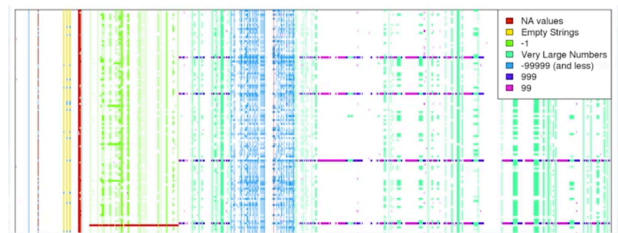
Knowing how to effectively handle date-time and coordinates, as well as numeric and categorical features, will provide you reliable way to improve your score. And to help you devise that specific part of solution which is often required to beat very top scores.

## Handling Missing Values

Often we have to deal with missing values in our data. They could look like not numbers, empty strings, or outliers like -999. Sometimes they can contain useful information by themselves, like what was the reason of missing value occurring here? How to use them effectively? How to engineer new features from them?

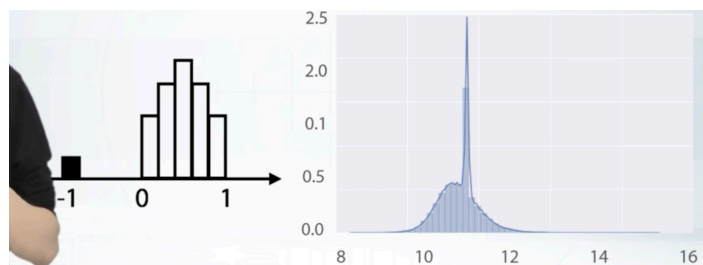
So what kind of information missing values might contain? How can they look like? Let's take a look at missing values in the Springfield competition.

This is matrix of samples and features. People manually reviewed each feature, and found missing values for each column. This values could be not a number, empty string, -1, 99, and so on. For example, how can we



found out that -1 can be the missing value? We could draw a histogram and see this variable has uniform distribution between 0 and 1. And that it has small peak of -1 values. So if there are no not numbers there, we can assume that they were replaced by -1. Or the feature distribution plot can look like the second figure.

Note that x-axis has log scale. In this case, not a numbers probably were filled by features mean value. You can easily generalize this logic to apply to other cases.



Okay on this example we just learned this, missing values can be hidden from us. And by hidden I mean replaced by some other value beside NaN.

Great, let's talk about missing value imputation. The most often examples are **first**, replacing NaN with some value outside feature

value range. **Second**, replacing NaN with mean or median. And **third**, trying to reconstruct value somehow.

**First** method is useful in a way that it gives three possibility to take missing value into separate category. The downside of this is that performance of linear models and NNs can suffer.

**Second** method usually beneficial for simple linear models and neural networks. But again for trees it can be harder to select object which had missing values in the first place.

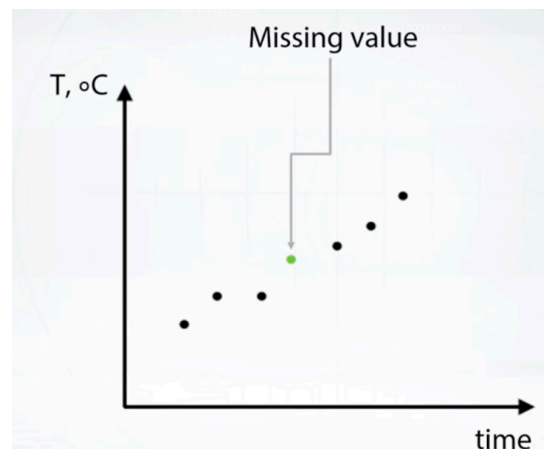
Let's keep the feature value reconstruction for now, and turn to feature generation for a moment.

The concern we just have discussed can be addressed by adding new feature `isnull` indicating which rows have missing values for this feature.

feature	isnull
0.1	False
0.95	False
NaN	True
-3	False
NaN	True

This can solve problems with trees and neural networks while imputing mean or median. But the downside of this is that we will double number of columns in the data set.

Now back to missing values imputation methods. The **third** one, and the last one we will discuss here, is to reconstruct each value if possible. One example of such possibility is having missing values in time series. For example, we could have everyday temperature for a month but several values in the middle of months are missing. Well of course, we can approximate them using nearby observations. But obviously, this kind of opportunity is rarely the case. In most typical scenario rows of our data set are independent. And we usually will not find any proper logic to reconstruct them.

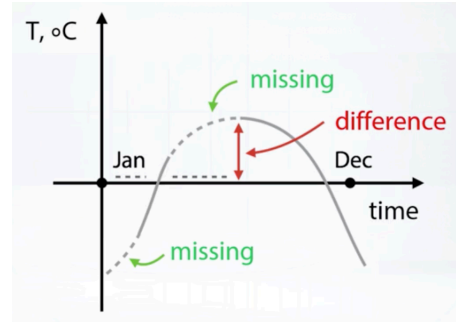


Great, to this moment we already learned that we can construct new feature, isnull indicating which rows contains NaN.

What are other important moments about feature generation we should know?

Well there's one general concern about generating new features from one with missing values. That is, if we do this, we should be very careful with replacing missing values before our feature generation. To illustrate this, let's imagine we have a year long data set with two features. Daytime feature and temperature which had missing values. We can see all of this on the figure.

Now we fill missing values with some value, for example with median. If you have data over the whole year median probably will be near zero so it should look like that. Now we want to add feature like difference between temperature today and yesterday, let's do this.



As we can see, near the missing values this difference usually will be abnormally huge. And this can be misleading our model. But hey, we already know that we can approximate missing values in time series by interpolation nearby points, great. But unfortunately, we usually don't have enough time to be so careful here. And more importantly, these problems can occur in cases when we can't come up with such specific solution.

Let's review **another method** of missing value imputation. Which will be substantially discussed later in advanced feature engineering topic.

Here we have a data set with independent rows. And we want to encode the categorical feature with the numeric feature. To achieve that we calculate mean value of numeric feature for every category, and replace categories with these mean values.

categorical_feature	numeric_feature
A	1
A	4
A	2
A	-1
B	9
B	NaN

What happens if we fill NaN in the numeric feature, with some value outside of feature range like -999.

categoryal _feature	numeric _feature	numeric_ feature_filled	categoryal _encoded
A	1	1	1.5
A	4	4	1.5
A	2	2	1.5
A	-1	-1	1.5
B	9	9	-495
B	NaN	-999	-495

As we can see, all values we will be doing them closer to -999. And the more the row's corresponding to particular category will have missing values, the closer mean value will be to -999. The same is true if we fill missing values with mean or median of the feature. This kind of missing value imputation definitely can screw up the feature we are constructing. The way to handle this particular case is to simply ignore missing values while calculating means for each category.

Again let me repeat the idea of these two examples.

You should be very careful with early NaN imputation if you want to generate new features. There's one more interesting thing about missing values.

XGBoost can handle NaNs and sometimes using this approach can change score drastically.

Besides common approaches we have discussed, sometimes we can treat outliers as missing values. For example, if we have some easy classification task with songs which are thought to be composed even before ancient Rome, or maybe the year 2025. We can try to treat these outliers as missing values.

If you have categorical features, sometimes it can be beneficial to change the missing values or categories which is present in the test data but is not present in the train data. The intention for doing so appeals to the fact that the model which didn't have that category in the train data will eventually treat it randomly. Here a supervised encoding of categorical features can be of help. As we already discussed, we can change categories to its frequencies and thus treat categories the same as before based on their frequency.

Let's walk through the example on the slide. There you see from the categorical feature, they not appear in the train. Let's generate new feature indicating number of where the occurrence is in the data.

We will name this feature `categorical_encoded`. Value A has six occurrences in both train and test, and that's value of new feature related to A will be equal to 6. The same works for values B, D, or C. But now new features various related to D and C are equal to each other. And if there is some dependence in between target and number of occurrences for each category, our model will be able to successfully visualize that.

Train:		Test:	
categorical_feature	target	categorical_feature	target
A	0	A	?
A	1	A	?
A	1	B	?
A	1	C	?
B	0		
B	0		
D	1		

Train:			Test:		
categorical_feature	categorical_encoded	target	categorical_feature	categorical_encoded	target
A	6	0	A	6	?
A	6	1	A	6	?
A	6	1	B	3	?
A	6	1	C	1	?
B	3	0			
B	3	0			
D	1	1			

To conclude this video, let's overview main points we have discussed.

The choice of method to fill NaN depends on the situation. Sometimes, you can reconstruct missing values. But usually, it is easier to replace them with value outside of feature range, like -999 or to replace them with mean or median.

Also missing values already can be replaced with something by organizers.

In this case if you want know exact rows which have missing values you can investigate this by plotting histograms. More, the model can improve its results using binary feature `isnull` which indicates what roles have missing values.

In general, avoid replacing missing values before feature generation, because it can decrease usefulness of the features. And in the end,

Xgboost can handle NaN directly, which sometimes can change the score for the better.

Using knowledge you have derived from our discussion, now you should be able to identify missing values. Describe main methods to handle them, and apply this knowledge to gain an edge in your next computation. Try these methods in different scenarios and for sure, you will succeed

### Treating values which do not present in train data

1. The choice of method to fill NaN depends on the situation
2. Usual way to deal with missing values is to replace them with -999, mean or median
3. Missing values already can be replaced with something by organizers
4. Binary feature "isnull" can be beneficial
5. In general, avoid filling nans before feature generation
6. Xgboost can handle NaN

### Overview of methods

- [Scikit-Learn \(or sklearn\) library](#)
- [Overview of k-NN \(sklearn's documentation\)](#)
- [Overview of Linear Models \(sklearn's documentation\)](#)
- [Overview of Decision Trees \(sklearn's documentation\)](#)
- [Overview of algorithms and parameters in H2O documentation](#)

### Additional Tools

- [Vowpal Wabbit repository](#)
- [XGBoost repository](#)

- [LightGBM repository](#)
- [Interactive demo of simple feed-forward Neural Net](#)
- [Frameworks for Neural Nets: Keras, PyTorch, TensorFlow, MXNet, Lasagne](#)
- [Example from sklearn with different decision surfaces](#)
- [Arbitrary order factorization machines](#)

## **Feature Extraction from Text and Images**