Common techniques in competitive data-science

Alessandro Solbiati (alessandro.solbiati@polimi.it), 同济大学 Sino-Italian Double Degree Class

School of Electronics and Information Engineering,

ABSTRACT

The purpose of this internship research project is to explore the most common techniques used by professional in data science nowadays. The research will follow along what has been done during the internship project, describing in detail all the different stages of how to approach a data-science task. The internship project was conducted in Anyhelper Technologies Shanghai LTD, from September 2018 to December 2018, in the data mining team. Note that in this paper it will not be shared details of the project and data used to work in the team due to a confidentiality agreement. Instead, will be explored all the approached commonly used by the data mining team to deal with their common tasks of predictive analysis on big scale data set. This internship research paper will cover the following topics.

In the first section will be explained how to approach a Machine learning task, how to do feature engineering and data cleaning, how to handle different kind of features (categorical and numerical), how to handle missing values. An overview of the most common evaluation metrics, both for classification and regression tasks, and also some optimization approach for these metrics.

In the second section will be examined how to implement a Machine Learning model, how to set up a proper validation system that doesn't lead to overfitting, and will be also explored the big topic of Exploratory Data Analysis (EDA): is explained how to use visualization tools to plot features and relationships between different features.

Finally in the third section will be examined one of the newest techniques in Data Science: ensembling. It will be shown how the combination of multiple Machine Learning models can often lead to a better model, and it will be explored the most common approach to set up this ensembling: from the most classical one (averaging and bagging) to the latest one (Boosting and Stacking). There will also be some mentions to the later implementations available for these Machine Learning models like XGBoost and lightLGB.

Finally a word of thanks for this project to my company mentors, Kelvin and Leo. This thesis would have not been possible without the guidance and helps of my two amazing mentors from Anyhelper Technologies Shanghai LTD, who with their multi-year experience in data science tasks helped me to approach and discover this new field of research that I will be excited and grateful to pursue in my academic future.

数据科学中的常用技术

摘要

这个实习研究项目的目的是探索当今数据科学专业人士最常用的技术。该研究将遵循实习项目期间所做的工作，详细描述如何处理数据科学任务的所有不同阶段。实习项目于2018年9月至2018年12月在 **上海初人科技有限公司** 的数据挖掘团队进行。请注意，在本文中，由于保密协议，将不会共享项目的详细信息以及用于团队工作的数据。相反，将探索数据挖掘团队常用的所有方法，以处理他们在大规模数据集上的预测分析的常见任务。该实习研究论文将涵盖以下主题。

在第一部分将解释如何处理机器学习任务，如何进行特征工程和数据清理，如何处理不同类型的功能（分类和数字），如何处理缺失值。概述了最常见的评估指标，包括分类和回归任务，以及这些指标的一些优化方法。

在第二部分将研究如何实现机器学习模型，如何建立一个不会导致过度拟合的适当验证系统，并将探讨探索性数据分析（EDA）的大话题：解释如何使用可视化工具绘制不同特征之间的特征和关系。

最后在第三部分将研究数据科学的最新技术之一：集成。将展示多种机器学习模型的组合如何能够经常产生更好的模型，并且将探索设置此集成的最常用方法：从最经典的（平均和装袋）到最新的（提升和堆叠）。对于像XGBoost和lightLGB这些机器学习模型可用的后续实现也会有一些提及。

最后，我的公司导师Kelvin和Leo对这个项目表示感谢。如果没有来自 **上海初人科技有限公司** 的两位出色指导者的指导和帮助，这篇论文是不可能的，他们拥有多年的数据科学任务经验，帮助我接近并发现这个新的研究领域，我将很兴奋并感激我的学术未来。

Index of contents

1.1 Anatomy of a data-science task

We will briefly examine the components of a data-science task/machine-learning competition.

1.1.1. Data

Data is the raw material we have to start our modelling. We will use it in order to produce our solution. With the data, usually there is a description. It's useful to read it in order to understand what we'll work with and which feature can be extracted.

1.1.2 Model

The next concept is a model. This is exactly what we will build during the competition. It's better to think about model not as one specific algorithm, but something that transforms data into answers. The model should have two main properties. It should produce best possible prediction and be reproducible. In fact, it can be very complicated and contain a lot of algorithms, handcrafted features, use a variety of libraries.

1.1.3 Predictions

Usually the data-set is divided into two parts, training and testing. The first part is used to train the model and the second part to test it, to check the strength of our predictions. Therefore, a standard competition routine looks like this. You as the competition, you analyze the data, improve model, prepare submission, and validate them on testing set. You repeat this action several times.

1.2 Machine-learning approaches

Machine learning is a science of devising algorithms to get computers to learn from data automated

without being explicitly programmed [4]. Such algorithms (learning algorithms)

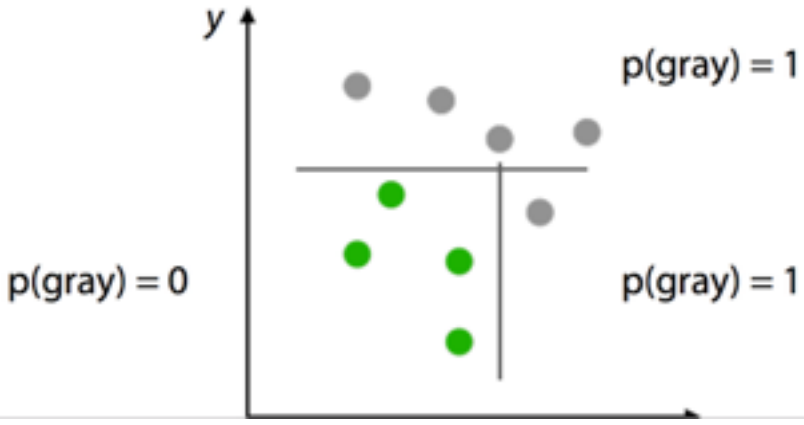
build a model after learning from the data and the model can be used to make predictions or decisions. The learning phase is also known as the training phase.

Here we will briefly go thourgh the most used machine learning algorithms for data-science tasks.

1.2.1. Linear Models

The term linear model implies that the model is specified as a linear combination of features. Based on training data, the learning process computes one weight for each feature to form a model that can predict or estimate the target value. You can find implementations of Linear Models in almost every machine learning library. Most known implementation in Scikit-Learn library. Take the figure above, where we have two sets of points, gray points belong to one class and green ones to another. It is very intuitive to separate them with a line. In this case, it's quite simple to do since we have only two dimensional points. But this approach can be generalized for a high dimensional space. This is the main idea behind Linear Models. They try to separate objects with a plane which divides space into two parts.

1.2.3 Tree Based Methods



Tree-Based Methods use decision tree as a basic block for building more complicated models. Let's consider an example of how decision tree works. Let’s consider the same set of point of the Linear Case, in the figure above. Let's separate one class from the other by a line parallel to the one of the axes. We use such restrictions as it significantly reduces the number of possible lines and allows us to describe the line in a simple way. After setting the split as shown at that picture, we will get two sub spaces, upper will have probability of gray=1, and lower will have probability of gray=0.2. Upper sub-space doesn't require any further splitting. Let's continue splitting for the lower sub-space. Now, we have zero probability on gray for the left sub-space and one for the right. This was a brief overview of how decision tree works.

1.2 Feature analysis and preprocessing

We will cover here a very useful topic of basic feature preprocessing and basic feature generation for different types of features. Namely, we will go through numeric features, categorical features, datetime features and coordinate features. Beside that, we also will discus dependence of preprocessing and generation on a model we're going to use.

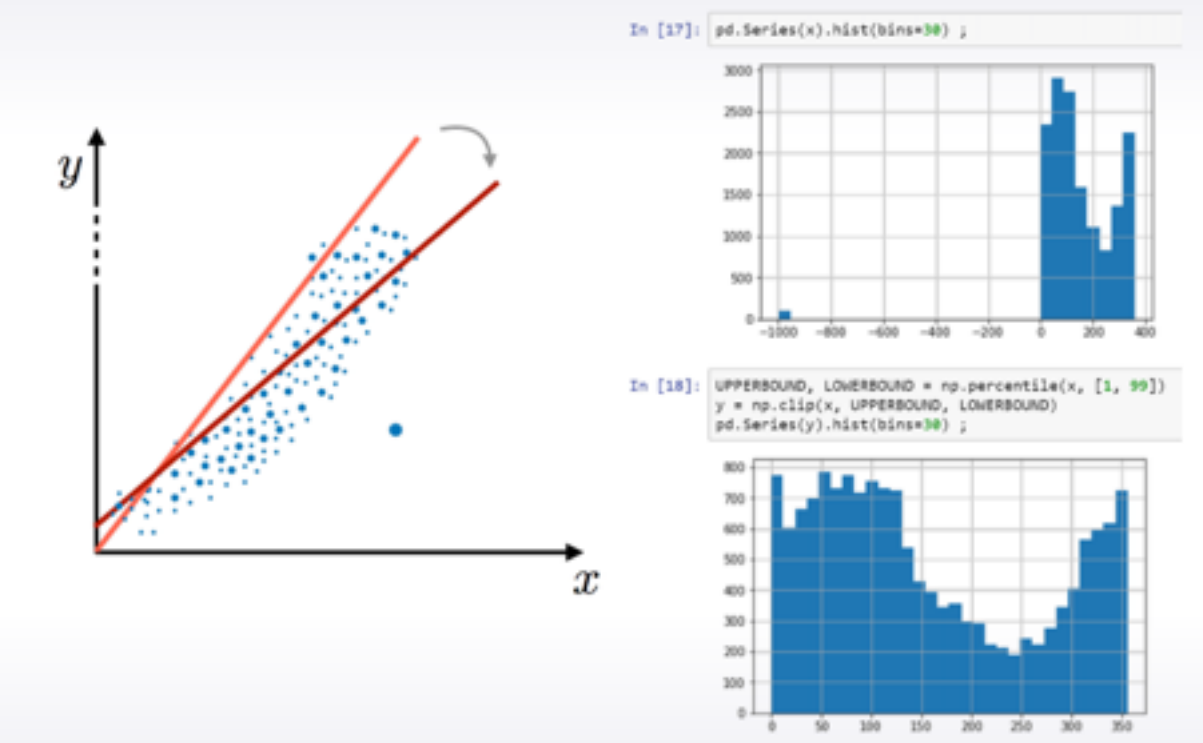
pclass	1	2	3
target	1	0	1

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

Feature preprocessing is highly dependent on the selection of the model. Let’s see the figure above. 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 will not get us to a good result here. So in order to improve a linear model's quality, we would want to **preprocess pclass feature in some wa**y. For example, with the so-called which will replace our feature with three, one for each of pclass values. The linear model will fit much better now than in the previous case. 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.

1.2.1 Numerical Features

When considering preprocessing of 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. It can multiply the feature 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 your nearest neighbors, linear models, and neural network.



When we work with linear models, there is a important part that influences model training results: **outliers.** For example, in this plot above, 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 zero and one. 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

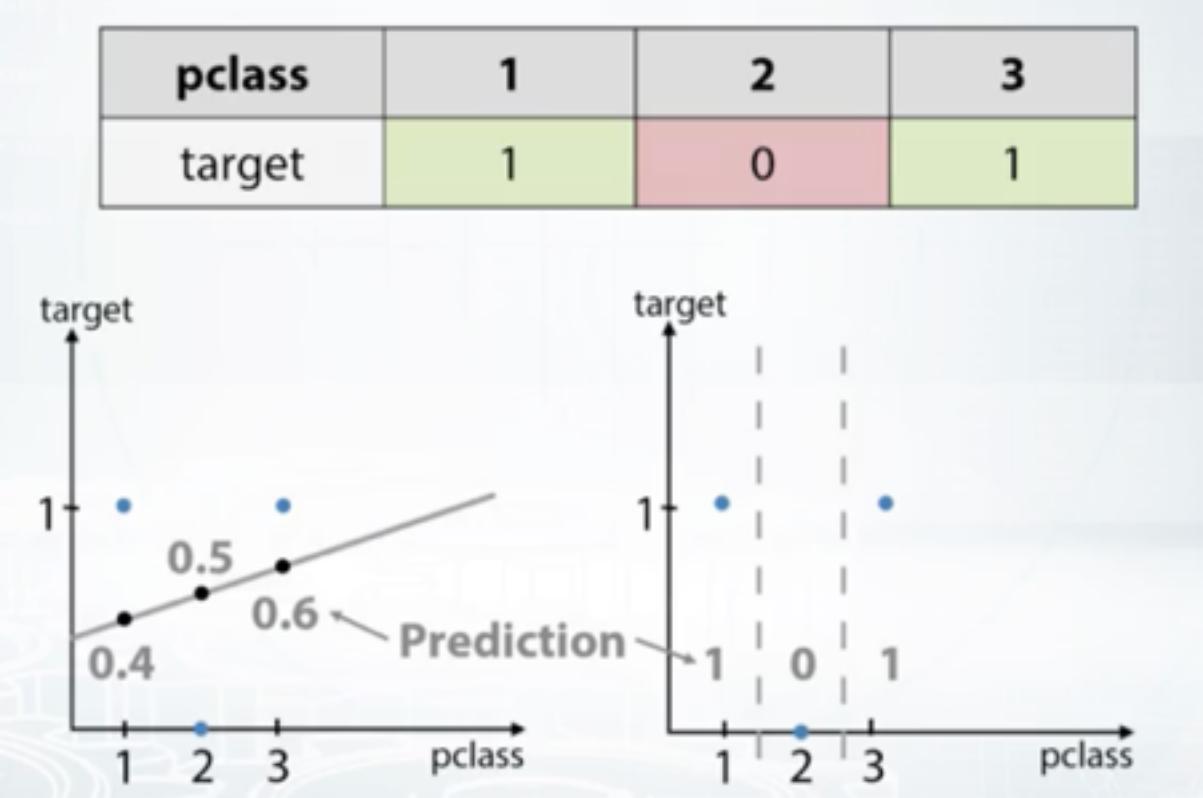
from outliers, we can clip features values between two chosen values of lower bound and upper bound. We can choose them as some percentiles of that feature. For example, first and 99s percentiles. This procedure of clipping is well-known in financial data and it is called winsorization. Let's take a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

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 not discuss feature generation with prior knowledge: we will thoroughly analyze and illustrate this skill in the next chapters on exploratory data analysis.

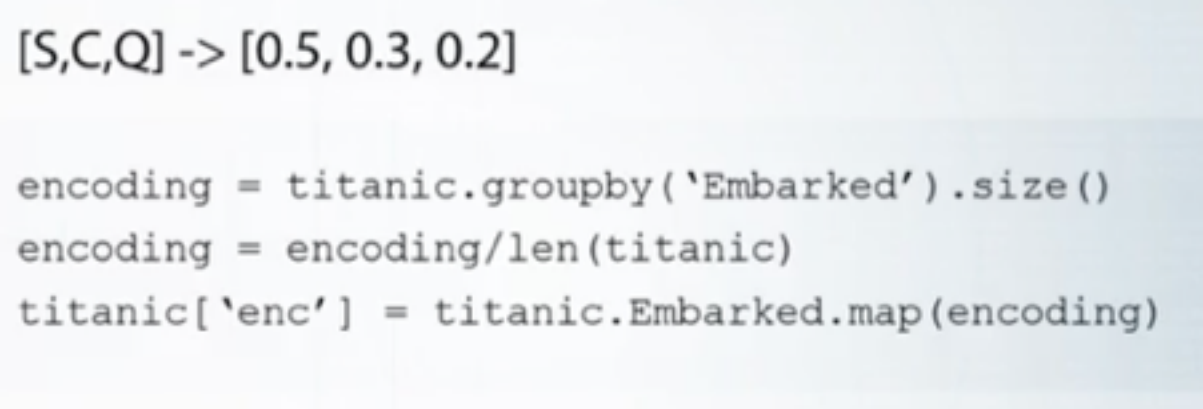
1.2.1 Categorical Features

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 two ways 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 one usually leads to the target of one, Pclass of two leads to zero, and Pclass of three leads to one. 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 set but three on the other side, we'll just make two splits select in each unique value and reaching it independently. Thus, this entries could achieve much better score here using these feature.

First, we can apply encoding in the **alphabetical** or sorted order. Unique way to solve of this feature namely S, C, Q. Thus, can be encoded as two,one, three. This is called label encoder from sklearn works by default. The second way is also labeling coding but slightly different. Here, we encode a categorical feature by **order of appearance**. For example, s will change to one because it was meant first in the data. Second then c, and we will change c to two. And the last is q, which will be changed to three. This can make sense if all were sorted in some meaningful way. This is the default behavior of *pandas.factorize* function.



The third method that I will tell you about is called **frequency encoding.** We can encode this feature via mapping values to their frequencies. Even 30 percent for 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 three models. first ones, can find this feature useful if value frequency is correlated to it's target value. While the second ones can help with less number of split 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 a apply or run categorization here in order to deal with such ties.

1.2.3 Missing Values

Often we have to deal with **missing values** in our data. They could look like not numbers, empty strings, or outliers like minus 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?

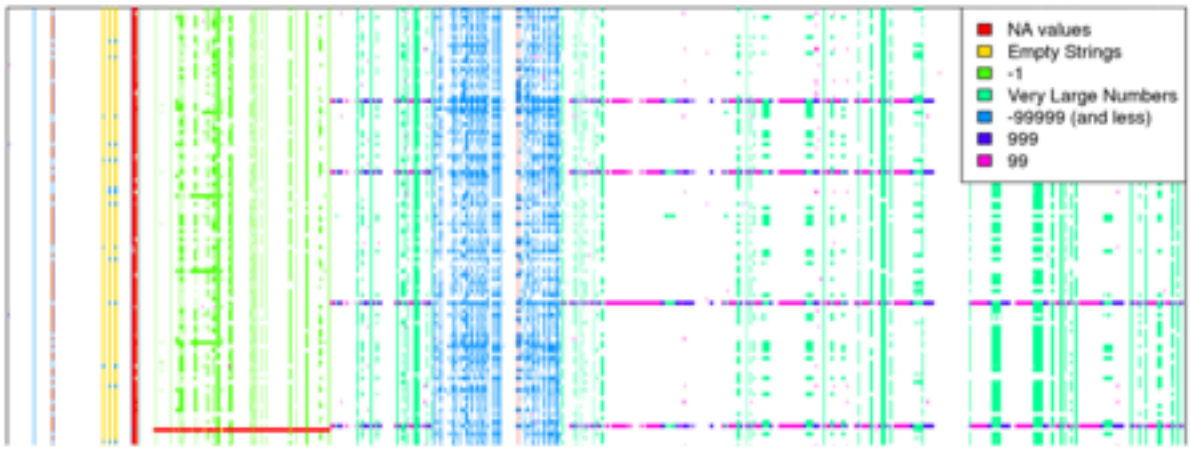
Let's take a look at missing values in the data-set above. This is metrics of samples and features. We reviewed each feature, and found missing values for each column. This latest could be not a number, empty string, minus 1, 99, and so on. For example, how can we found

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.

Let's talk about missing value importation. The most often examples are first, replacing not a number with some (1) **value outside fixed value range**. Second, replacing not a number with **(2) mean or median**. And third, trying to (3) **reconstruct** value somehow.

1) 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 networks can suffer.

2) Second method usually beneficial for simple linear models and neural networks. But again



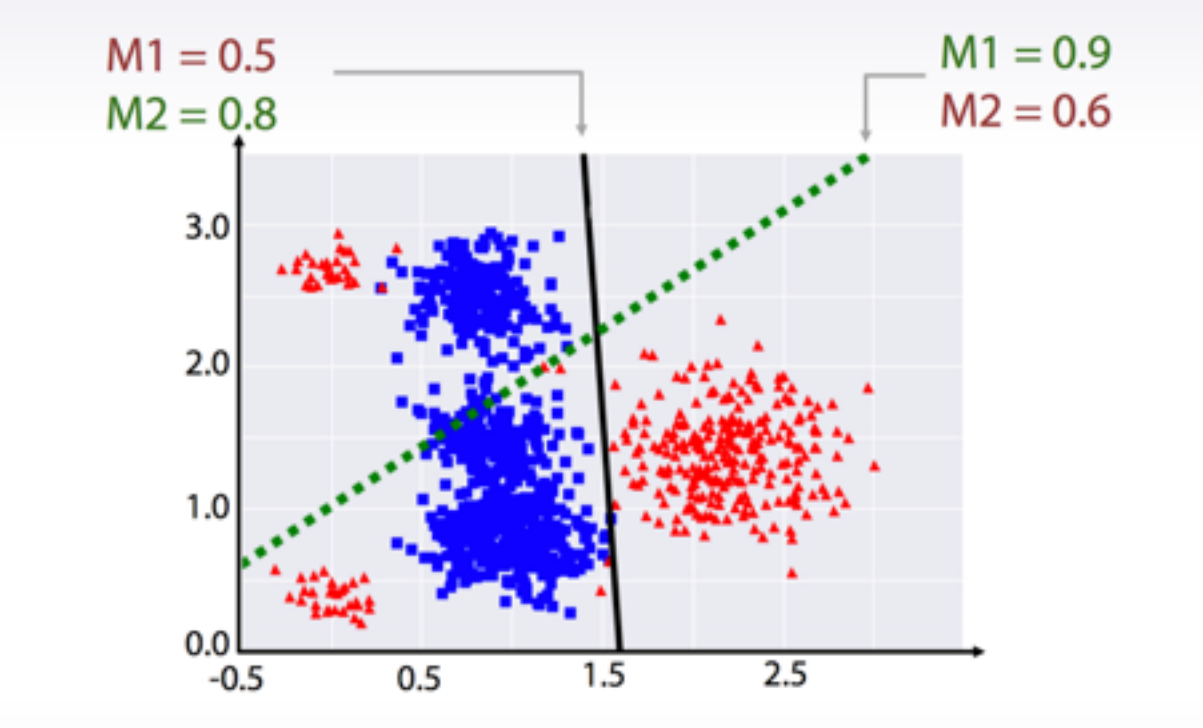
for trees it can be harder to select object which had missing values in the first place.

3) 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.

1.3 Evaluation metrics

Metrics are an essential part of any competition. They are used to evaluate our submissions. Why do we have a different evolution metric on each competition?

That is because there are plenty of ways to measure equality of an algorithm and each company decides for themselves what is the most appropriate way for their particular problem. For example, let's say an online shop is trying to maximize effectiveness of their website. The thing is you need to formalize what is effectiveness. You need to define a metric how effectiveness is measured.



It is very important to understand how metric works and how to optimize it efficiently. I want to stress out that it is really important to optimize **exactly the metric we're given** in the competition and not any other metric.

Consider an example, blue and red lines represent objects of a class zero and one respectively. And say we decided to use a linear classifier,and came up with two matrix to optimize, M1 and M2. The question is, how much different the resulting classifiers would be?

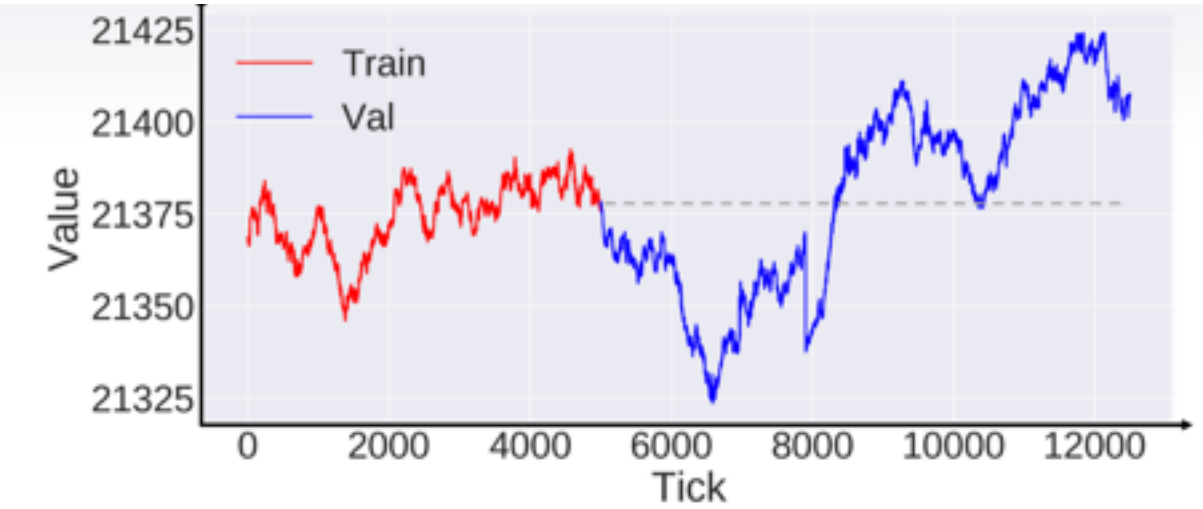
Actually by a lot. The two lines here, the solid and the dashed one show the best line your boundaries for the two cases. For the dashed, M1 score is the highest among all possible

hyperplanes. But M2 score for the hyperplane is low. And we have an opposite situation for the solid boundary. M2 score is the highest, whereas M1 score is low.

Now, if we know that in this particular competition, the ranking is based on M1 score, then we need to optimize M1 score and so we should submit the prediction. Predictions of the model with dash boundary. Once again, **if your model is scored with some metric, you get best results by optimizing exactly that metric.**

Now, the biggest problem many times is that some metrics cannot be optimized efficiently. That is there is no simple enough way to find the optimal hyperplane.

That is why sometimes we need to **train our model to optimize something different than competition metric.** But in this case we will need to apply various heuristics to improve competition metric score.



$$Loss(\hat{y}_i; y_i) = \begin{cases} |y_i - \hat{y}_i|, & \text{if trend predicted correctly} \\ (y_i - \hat{y}_i)^2, & \text{if trend predicted incorrectly} \end{cases}$$

Let's look at a time series example. In this example about time-series forcasting the metric was quite unusual actually, but it is intuitive: if a trend is guessed correctly, then the absolute difference between the prediction and the target is considered as an error.

If for instance, model predict end value in the prediction horizon to be higher than the last value from the train side but in reality it is lower, then the trend is predicted incorrectly, and the error was set to absolute difference squared.

So this metric carries a lot more about correct trend to be predicted than about actual value you predict. And **that is something it is possible to exploit:** there were several times series

was to forecast, the horizon to predict was wrong, and the model's predictions were unreliable. Moreover, **it is not possible to optimize this metric exactly**. The key is that itwould be much better to set all the predictions to either last value plus a very tiny constant, or last value minus very tiny constant. The same value for all the points in the time interval, we are to predict for each time series.

We should not forget to do kind of exploratory metric analysis along with exploratory data analysis. At least when the metric is an unusual one.

1.3.1 Regression metrics

We will start discussing metrics for regression tesk. First, let us clarify the notation we're going to use throughout the chapter. *N* will be the number of samples in our training data set, *y* is that the target, and *y-hat* is our model's predictions. And *y-hat and y with index i* are the predictions, and target value respectively for *i-th* object.

- N – number of objects
- $y \in \mathbb{R}^N$ – target values
  $\hat{y} \in \mathbb{R}^N$ – predictions
- $\hat{y}_i \in \mathbb{R}$ – prediction for i-th object
  $y_i \in \mathbb{R}$ – target for i-th object

1.3.1.1 MEAN SQUARE ERROR (MSE)

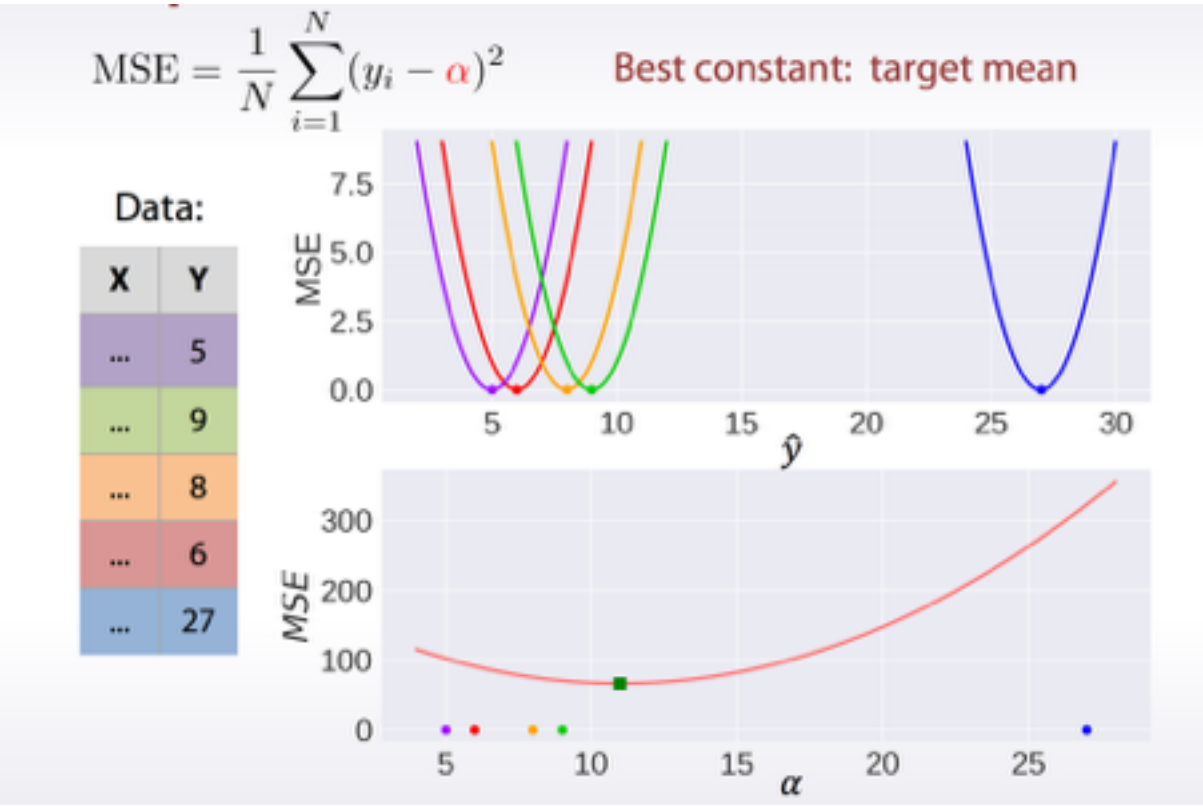
MSE basically measures **average squared error** of our predictions. For each point, we calculate square difference between the predictions of the target and then average those values over the objects. Let's introduce a simple data set now. Say, we have five objects, and each object has some features, X, and the target is shown in the column Y.

Let's ask ourselves a question. **How will the error change if we fix all the predictions but want to be perfect, and we'll derive the value of the remaining one?**

To answer this question, take a look at the plot below . On the horizontal line, we will first put points to the positions of the target values. The points are colored according to the

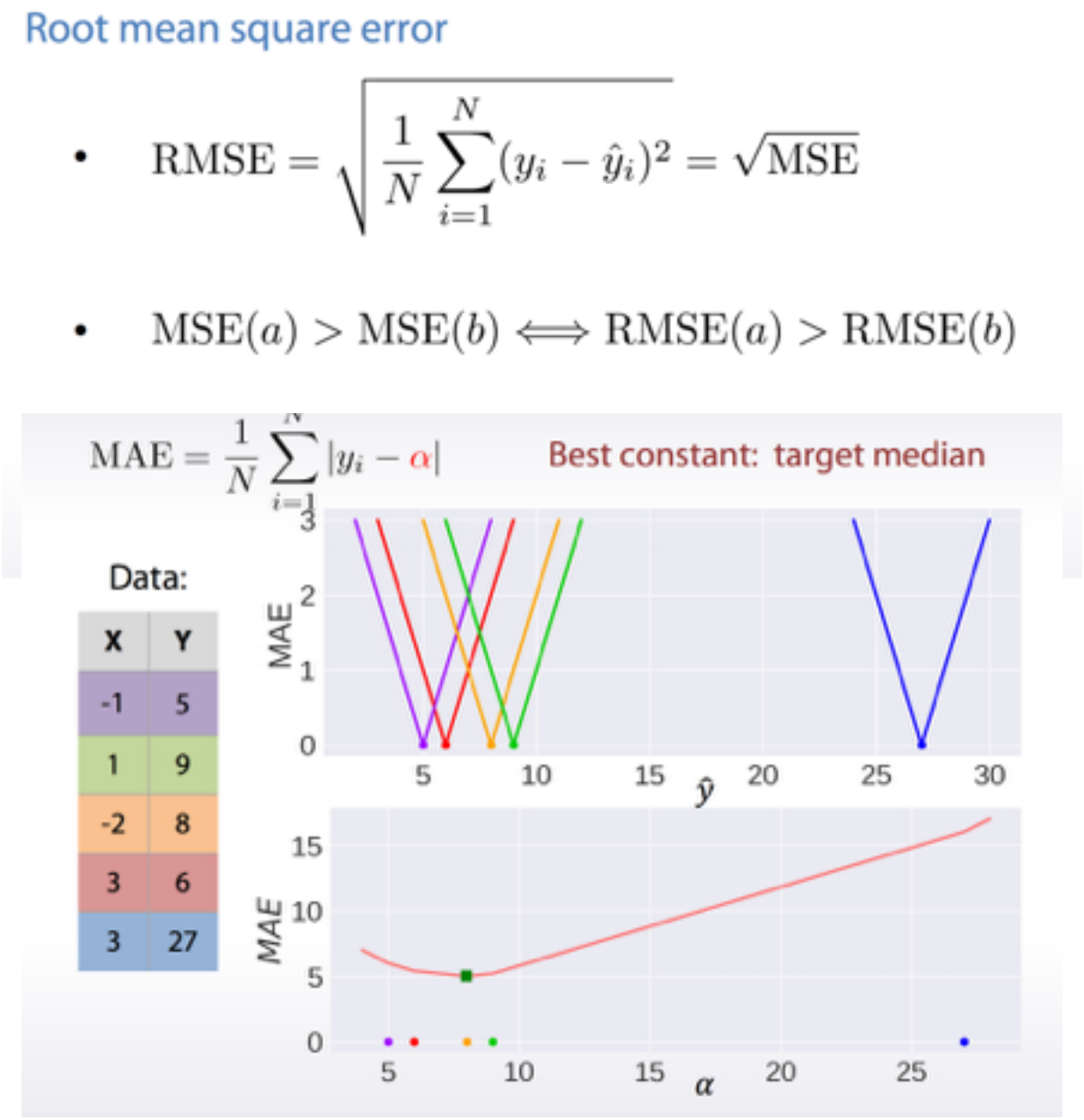
corresponding rows in our data table. And on the Y-axis, we will show the mean square error. So, let's now assume that our predictions for the first four objects are perfect, and let's draw a curve. How the metric value will change if we change the prediction for the last object?

For MSE metric, it looks like that. In fact, if we predict 25, the error is zero, and if we predict



something else, then it is greater than zero. And the error curve looks like parabola. Let's now draw analogous curves for other objects. Well, right now it's hard to make any conclusions but we will build the same kind of plot for every metric and we will note the difference between them. Now, let's build the simplest baseline model. We'll not use the features X at all and we will always predict a constant value Alpha. B**ut, what is the optimal constant?** What constant minimizes the mean square error for our data set? In fact, it is easier to set the derivative of our total error with respect to that constant to zero, and find it from this equation. What we'll find is that the best constant is the **mean value of the target column.**

1.3.1.2 ROOT MEAN SQUARE ERROR (RMSE)



RMSE, Root Mean Square Error, is a **very similar metric to MSE.** In fact, it is calculated in two steps. First, we calculate regular mean square error and then, we take a square root of it. The square root is introduced to make scale of the errors to be the same as the scale of the targets. For MSE, the error is squared, so taking a root out of it makes total error a little bit easier to comprehend because it is linear now. Now, it is very important to understand in what sense RMSE is similar to MSE, and what is the difference. First, they are similar in terms of their minimizers. **Every minimizer of MSE is a minimizer of RMSE** and vice versa. But generally, if we have two sets of predictions, A and B, and say MSE of A is greater than MSE

of B, then we can be sure that RMSE of A is greater RMSE of B. And it also works in the opposite direction. This is actually true only because square root function is non-decreasing.

What does it mean for us? It means that, if our target the metric is RMSE, we still can compare our models using MSE, since MSE will order the models in the same way as RMSE. And we can optimize MSE instead of RMSE. In fact, **MSE is a little bit easier to work with, so everybody uses MSE instead of RMSE**. But there is a little bit of difference between the two for gradient-based models. Take a look at the gradient of RMSE with respect to i-th prediction. It is basically equal to gradient of MSE multiplied by some value. The value doesn't depend on the index I. It means that travelling along MSE gradient is equivalent to traveling along RMSE gradient but with a different flowing rate and the flowing rate depends on MSE score itself. So, it is kind of dynamic. So even though RMSE and MSE are really similar in terms of models scoring, **they can be not immediately interchangeable for gradient based methods.**

1.3.1.3 R_SQUARED

R-squared:

$$R^2 = 1 - \frac{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2} = 1 - \frac{MSE}{\frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

$$\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

Now, what if I told you that MSE for my models predictions is 32? Should I improve my model or is it good enough? Or what if my MSE was 0.4? Actually, it's hard to realize if our model is good or not by looking at the absolute values of MSE or RMSE. It really depends on the properties of the dataset and their target vector. How much variation is there in the target vector. We would probably want to measure how much our model is better than the constant baseline. And say, the desired metrics should give us zero if we are no better than the baseline and one if the predictions are perfect.

For that purpose, R_squared metric is usually used. Take a look. When MSE of our predictions is zero, the R_squared is 1, and when our MSE is equal to MSE over constant model, then R_squared is zero. Well, because the values in numerator and denominator are the same. And all reasonable models will score between 0 and 1. The most important thing for us is that to optimize R_squared, we can optimize MSE. It will be absolutely equivalent since R_squared is basically MSE score divided by a constant and subtracted from another constant. These constants doesn't matter for optimization.

1.3.1.4 MEAN ABSOLUTE ERROR (MAE)

Lets move on and discuss another metric called **Mean Absolute Error, or MAE in short.** The error is calculated as an average of absolute differences between the target values and the predictions. What is important about this metric is that **it penalizes huge errors that not as that badly as MSE does.** Thus, it's not that sensitive to outliers as mean square error.

It also has a little bit different applications than MSE. MAE is widely used in finance, where $10 error is usually exactly two times worse than $5 error. On the other hand, MSE metric thinks that $10 error is four times worse than $5 error. MAE is easier to justify. And if you used RMSE, it would become really hard to explain to your boss how you evaluated your model.

What constant is optimal for MAE? It's quite easy to find that its a median of the target values. In this case, it is eight. Just to verify that everything is correct, we again can try to Greek search for an optimal value with a simple loop. And in fact, the value we found is 7.98, which indicates we were right. Here, we see that MAE is more robust than MSE, that is, it is not that influenced by the outliers.

In fact, recall that the optimal constant for MSE was about 11 while for MAE it is eight. And eight looks like a much better prediction for the points on the left side. If we assume that point with a target 27 is an outlier and we should not care about the prediction for it.

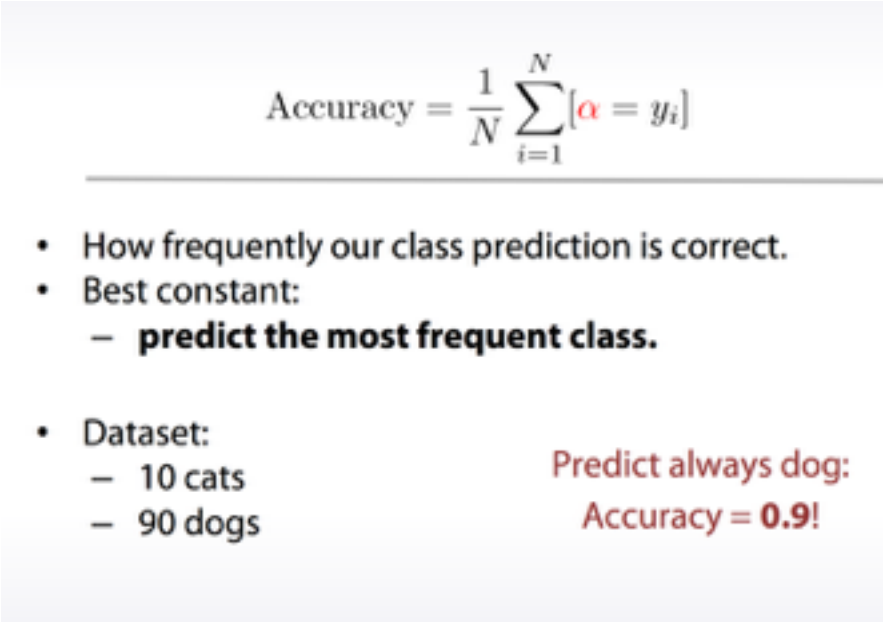
1.3.2 Classification Metrics

1.3.2.1 ACCURACY

Accuracy is the most straightforward measure of classifiers quality.

It's a value between 0 and 1. The higher, the better. And it is equal to the fraction of correctly classified objects. To compute accuracy, we need hard predictions. We need to assign each object a specific table. Now, what is the best constant to predict in case of accuracy? Actually, there are a small number of constants to try. We can only assign a class label to all the objects at once. So what class should we assign? **Obviously, the most frequent one.**

Then the number of correctly guessed objects will be the highest. But exactly because of that reason, there is a caveat in interpreting the values of the accuracy score.



Take a look at this example. Say we have 10 cats and 90 dogs in our train set. If we always predicted dog for every object, then the accuracy would be already 0.9. And imagine you tell someone that your classifier is correct 9 times out of 10.

The person would probably think you have a nice model. But in fact, your model just predicts dog class no matter what input is. So the problem is, that the base line accuracy can be very high for a data set, even 99%, and that makes it hard to interpret the results. Although accuracy score is very clean and intuitive, it turns out to be quite hard to optimize.

Accuracy also doesn't care how confident the classifier is in the predictions, and what soft predictions are. It cares only about arg max of soft predictions. And thus, people sometimes prefer to use different metrics that are first, easier to optimize. And second, these metrics work with soft predictions, not hard ones.

1.3.2.2. LOG-LOSS

One of such metrics is logarithmic loss. It tries to make the classifier to output two posterior probabilities for their objects to be of a certain kind, of a certain class. A log loss is usually the reason a little bit differently for binary and multi class tasks. For binary, it is assumed that y hat is a number from 01 range, and it is a probability of an object to belong to class one. So 1 minus y hat is the probability for this object to be of class 0. For multiclass tasks, LogLoss is written in this form.

- Binary:

$$\text{LogLoss} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

$$y_i \in \mathbb{R}, \quad \hat{y}_i \in \mathbb{R}$$

- Multiclass:

$$\text{LogLoss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{L} y_{il} \log(\hat{y}_{il})$$

$$y_i \in \mathbb{R}^L, \quad \hat{y}_i \in \mathbb{R}^L$$

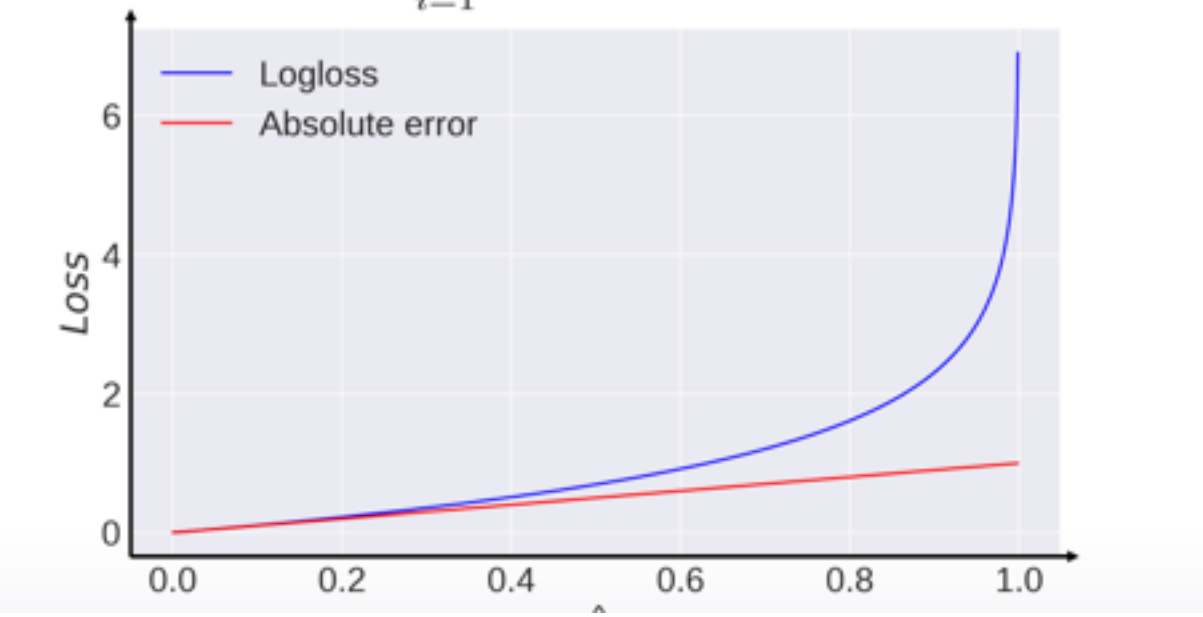
- In practice:

$$\text{LogLoss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{L} y_{il} \log(\min(\max(\hat{y}_{il}, 10^{-15}), 1 - 10^{-15}))$$

Here y hat ith is a vector of size L, and its sum is exactly 1. The elements are the probabilities to belong to each of the classes. Try to write this formula down for L equals 2, and you will see it is exactly binary loss from above. And finally, it should be mentioned that to avoid in practice, predictions are clipped to be not from 0 to 1, but from some small positive number to 1 minus some small positive number.

Okay, now let us analyze it a little bit. Assume a target for an object is 0, and here on the plot, we see how the error will change if we change our predictions from 0 to 1. For comparison, we'll plot absolute error with another color. **Logloss usually penalizes completely wrong answers and prefers to make a lot of small mistakes to one but severer mistake.** Now, what is the best constant for logarithmic loss? It turns out that you need to set predictions to the frequencies of each class in the data set. In our case, the frequencies for the cat class is 0.1, and it is 0.9 for class dog. Then the best constant is vector of those two values.

$$\text{LogLoss} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$



1.3.2 Metrics Optimization

Let's start with a comparison between two notions, **loss** and **metric**. The metric or target metric is a function which we want to use to evaluate the quality of our model. For example, for a classification task, we may want to maximize accuracy of our predictions, how frequently the model outputs the correct label. But the problem is that no one really knows how to optimize accuracy efficiently. Instead, people come up with the proxy loss functions. They are such evaluation functions that are easy to optimize for a given model. For example, logarithmic loss is widely used as an optimization loss, while the accuracy score is how the solution is eventually evaluated. So, once again, the loss function is a function that our model optimizes and uses to evaluate the solution, and the target metric is how we want the solution to be evaluated.

Now, let's overview the approaches to target metrics optimization in general. The approaches can be broadly divided into several categories, depending on the metric we need to optimize. Some **metrics can be optimized directly.** That is, we should just find a model that optimizes this metric and run it. In fact, all we need to do is to set the model's loss function to these metric. The most common metrics like MSE, Logloss are implemented as loss functions in almost every library. For some of the metrics that cannot be optimized directly, we can somehow **pre-process the train set and use a model with a metric or loss function which is easy to optimize.** For example, while MSPE metric cannot be optimized directly with XGBoost, we will see later that we can resample the train set and optimize MSE loss instead,

which XGBoost can optimize. Sometimes, **we'll optimize incorrect metric, but we'll post-process the predictions to fit classification, to fit the communication metric better**. For some models and frameworks, it's possible to define a custom loss function, and sometimes it's possible to implement a loss function which will serve as a nice proxy for the desired metric.

- **Define an 'objective':**
  - function that computes *first and second order derivatives* w.r.t. predictions.

```
def logregobj(preds, dtrain):
    labels = dtrain.get_label()
    preds = 1.0 / (1.0 + np.exp(-preds))
    grad = preds - labels
    hess = preds * (1.0-preds)
    return grad, hess
```

It's actually quite easy to define a custom loss function for XGBoost. We only need to implement a single function that takes predictions and the target values and **computes first and second-order derivatives of the loss function with respect to the model's predictions.** For example, here you see one for the Logloss. Of course, the loss function should be smooth enough and have well-behaved derivatives, otherwise XGBoost will drive crazy. In this course, we consider only a small set of metrics, but there are plenty of them in fact. And for some of them, it is really hard to come up with a neat optimization procedure or write a custom loss function.

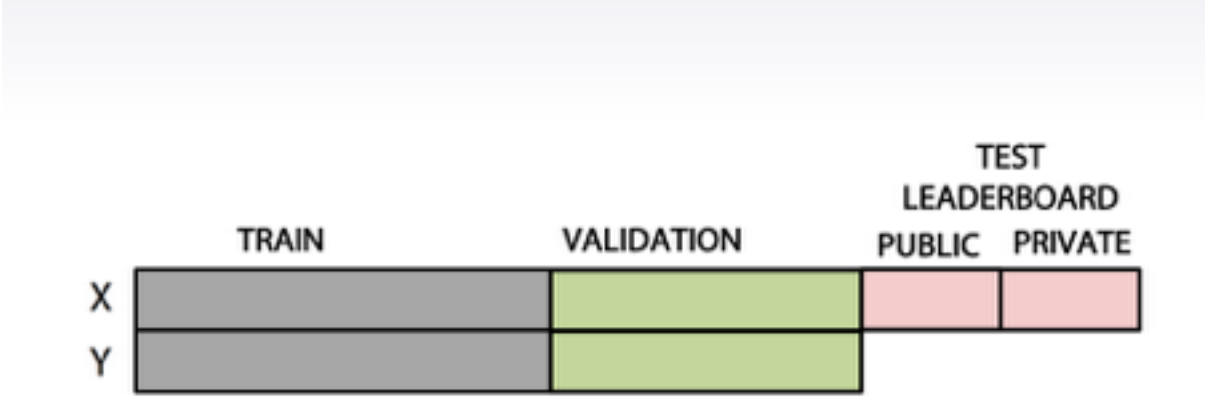
2 Implementing your models

2.1 Validation

Validation is one of the most common issues in a data-science task. What is it about?

In the nutshell, we want to check if the model gives expected results on the unseen data. For example, if you've worked in a healthcare company which goal is to improve life of patients, we could be given the task of predicting if a patient will be diagnosed a particular disease in the near future. Here, we need to be sure that the model we train will be applicable in the future. And not just applicable, we need to be sure about what quality this model will have depending on the number of mistakes the model make. And on the predictive probability of a patient having this particular disease, we may want to decide to run special medical tests for the patient to clarify the diagnosis.

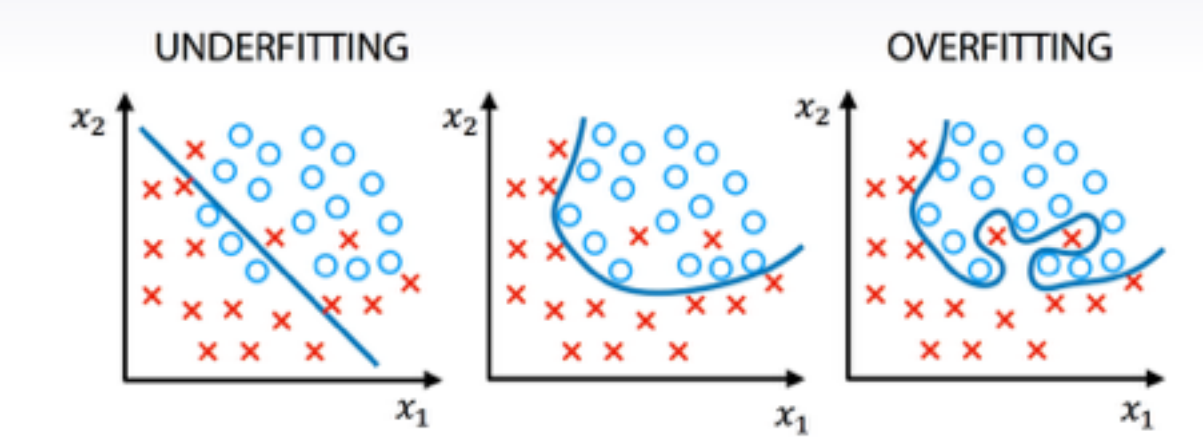
So, we need to correctly understand the quality of our model. But, this quality can differ on train data from the past and on the unseen test data from the future. The model could just memorize all patients from the train data and be completely useless on the test data because we don't want this to happen. We need to check the quality of the model with the data we have and these checks are the validation.



The common approach is the follwing:, we divide data we have into two parts, train part and validation part. We fit our model on the train part and check its quality on the validation part. Beside that, in the last example, our model will be checked against the unseen data in the future and actually these data can differ from the data we have.

One of the most important concept regarding validation is **underfitting and overfitting.** Let's understand this concept on a very simple example of a binary classification test. We will be using simple models defined by formulas under the pictures and visualize the results of model's predictions. Here on the left picture, we can see that if the model is too simple, it can't capture underlined relationship and we will get poor results. This is called underfitting. Then, if we want our results to improve, we can increase the complexity of the model and that will probably find that quality on the training data is going down. But on the other hand, if we make too complicated model like on the right picture, it will start describing noise in

the train data that doesn't generalize the test data. And this will lead to a decrease of model's



quality. This is called overfitting.

2.1 Validation Strategies

To use validation, we first need to split the data with given labels, in train and validation parts. In this section, we will discuss different validation strategies. And answer the

questions: how many splits should we make and what are the most often used methods to perform such splits?Loosely speaking, the main difference between these validation strategies is the number of splits being done. Here I will discuss three of them.

2.1.1 Holdout

Let's start with holdout. It's a simple data split which divides data into two parts, training data frame, and validation data frame. And the important note here is that in any method, holdout included, one sample can go either to train or to validation. So the samples between train and the validation do not overlap, if they do, we just can't trust our validation.

- Holdout: ngroups = 1
  `sklearn.model_selection.ShuffleSplit`

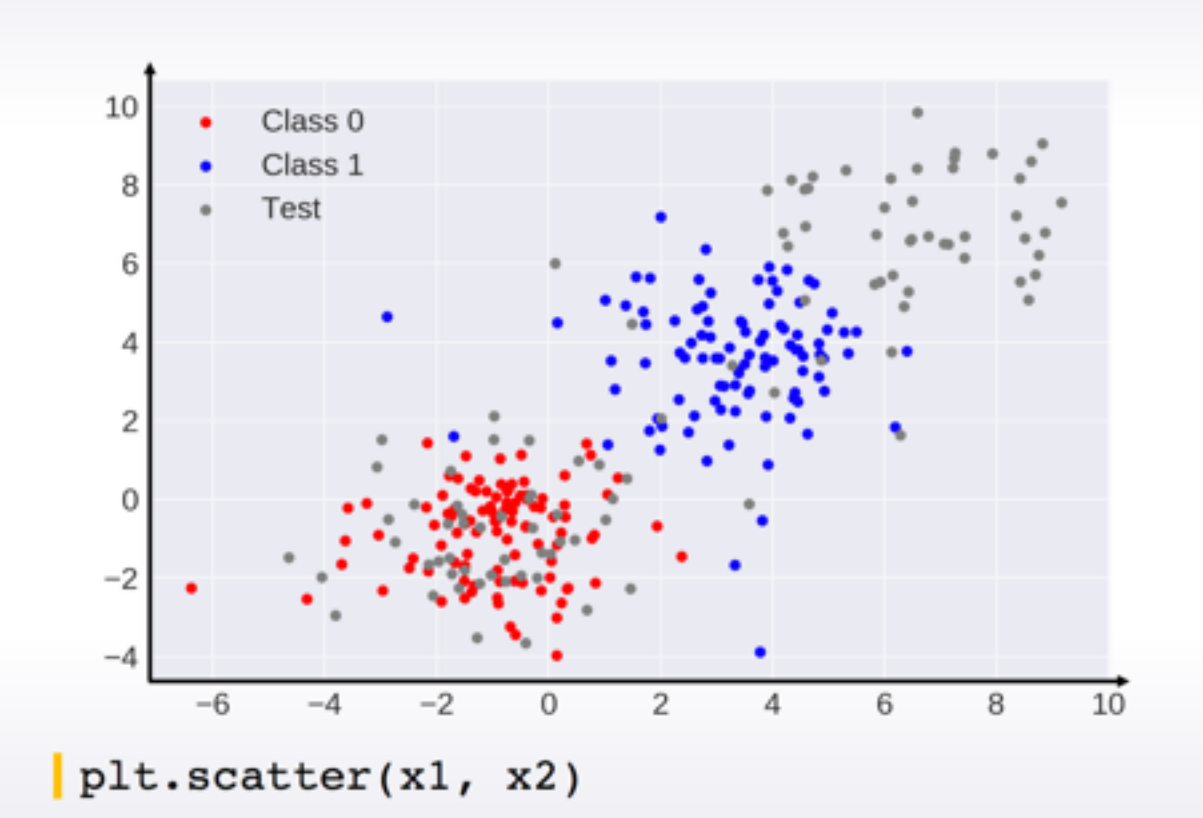
2.1.2 K-FOLD

K-fold can be viewed as a repeated holdout, because we split our data into key parts and iterate through them, using every part as a validation set only once. After this procedure, we average scores over these K-folds. Here it is important to understand the difference between K-fold and usual holdout or bits of K-times. While it is possible to average scores they receive after K different holdouts. In this case, some samples may never get invalidation, while others can be there multiple times. On the other side, the core idea of K-fold is that we want to use every sample for validation only once. This method is a good choice when we have a minimum amount of data, and we can get either a sufficiently big difference in quality, or different optimal parameters between folds

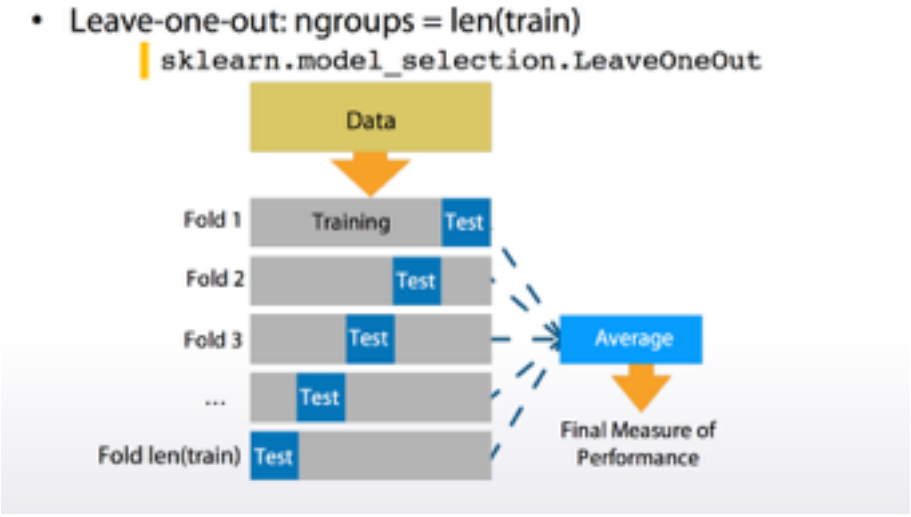
2.1.3 Leave-one-out

It is called leave-one-out. And basically it is a special case of Kfold when K is equal to the number of samples in our data. This means that it will iterate through every sample in our

data. Each time usion came in a slot minus one object is a train subset and one object left is a test subset. This method can be helpful if we have too little data and just enough model to



entrain.



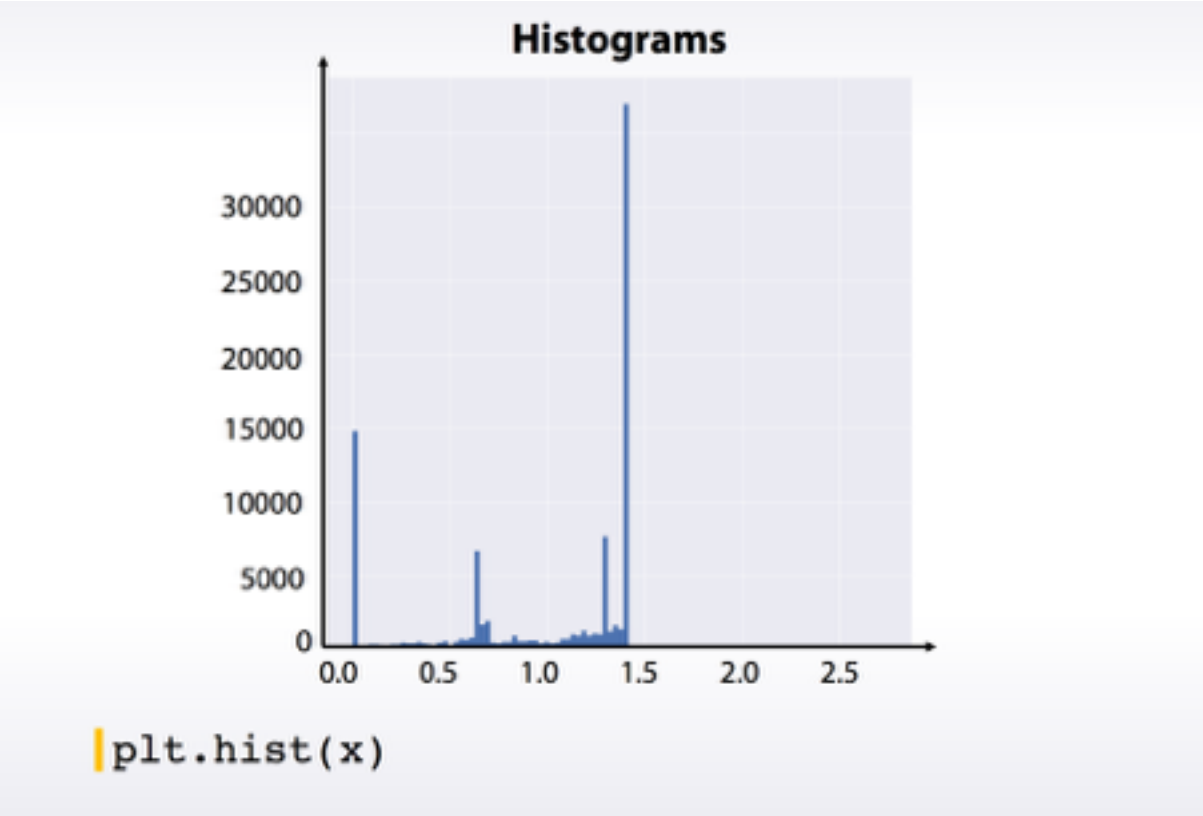
2.2 Exploratory Data Analysis

In this section we will talk about the very first steps a good data scientist takes when he is given a new data set. Mainly, exploratory data analysis or EDA in short.

What is Exploratory Data Analysis? It's basically a process of looking into the data, understanding it and getting comfortable with it.

To solve a problem, you need to understand a problem, and to know what you are given to solve it. In data science, complete data understanding is required to generate powerful features and to build accurate models. In fact while you explore the data, you build an intuition about it. And when the data is intuitive for you, you can generate hypothesis about possible new features and eventually find some insights in the data which in turn can lead to a better score.

2.2.1 VISUALIZATION STRATEGIES



The first, we can build histograms. Histograms split feature edge into bins and show how many points fall into each bin. Note that histograms may be misleading in some cases, so try

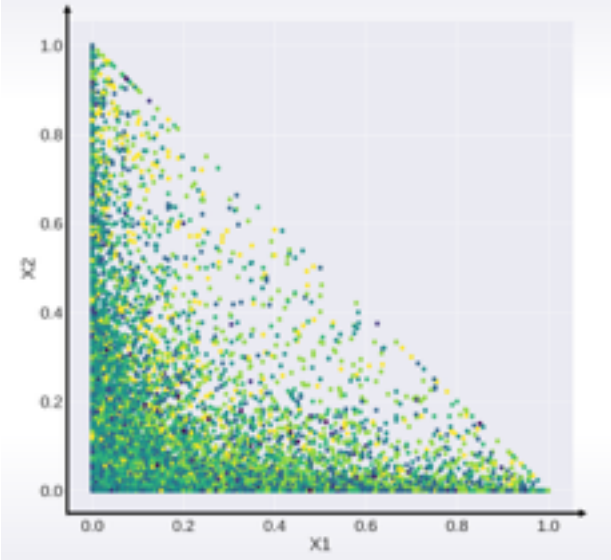
to overwrite its number of bins when using it. Also, know that it aggregates in the data, so we cannot see, for example, if all the values are unique or there are a lot of repeated values.

We can also build the plot where on X axis, we have a row index, and on the Y axis, we have feature values. It is convenient not to connect points with line segments but only draw them with circles. Now, if we observe horizontal lines on this kind of plot, we understand there are a lot of repeated values in this feature. Also, note the randomness over the indices. That is, we see some horizontal patterns but no vertical ones. It means that the data is properly shuffled. We can also color code the points according to their labels. Here, we see that the feature is quite good as it presumably gives a nice class separation.

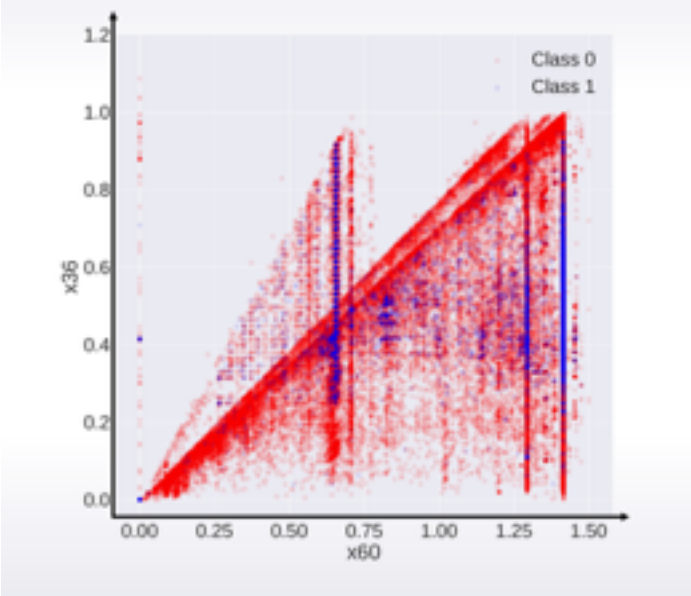
2.2.2 EXPLORATION OF FEATURE RELATIONS

Examining pairs of feature, we can also draw a scatter plot in this case. With it, we can draw one sequence of values versus another one. And usually, we plot one feature versus another feature. So each point on the figure correspond to an object with the feature values shown by points position. If it's a classification task, it's convenient to color code the points with their labels like on this picture. The color indicates the class of the object. For regression, the heat map light coloring can be used, too. Or alternatively, the target value can be visualized by point size. We can effectively use scatter plots to check if the data distribution in the train and test sets are the same. In this example, the red points correspond to class zero, and the blue points to class one. And on top of red and blue points, we see gray points. They correspond to test set. We don't have labels for the test set, that is why they are gray. And we clearly see that the red points are mixed with part of the gray ones, and that that is good actually. But other gray points are located in the region where we don't have any training data, and that is bad.

Let’s look at the example above. Say, we plot feature X1 versus feature X2. What can we say about their relation? The right answer is X2 is less or equal than one_minus_X1. Just realize that the equation for the diagonal line is X1 + X2 = 1, and for all the points below the line, X2 is less or equal than one_minus_X1. So, suppose we found this relation between two features, how do we use this fact? Of course, it depends, but at least there are some obvious

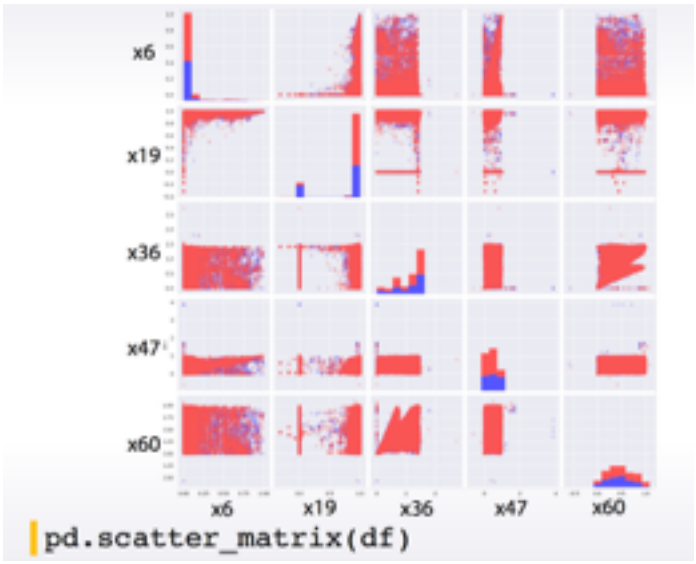


features to generate. For tree-based models, we can create a new features like the difference or ratio between X1 and X2.

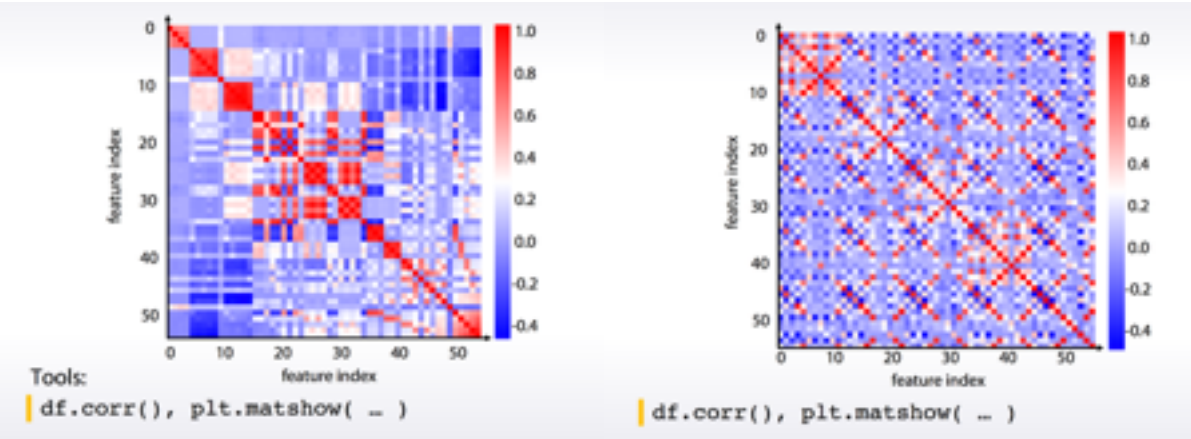


Another example above: this time it's harder to say what is the true relation between the features, but after all, our goal is not to decode the data here but to generate new features and get a better score. And this plot gives us an idea how to generate the features out of these two features. We see several triangles on the picture, so we could probably make a feature to each triangle a given point belongs, and hope that this feature will help.

When you have a small number of features, you can plot all the pairwise scatter plots at once using scatter metrics function from Pandas. It's pretty handy. It's also nice to have histogram and scatter plot before the eyes at the same time as scatter plot gives you very vague information about densities, while histograms do not show feature interactions.

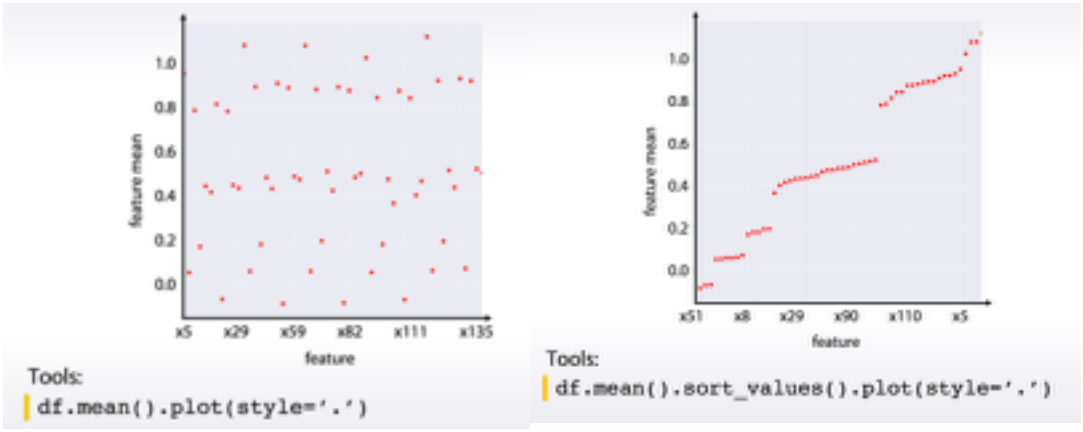


We can also compute some kind of **distance between the columns of our feature** table and store them into a matrix of size number of features by a number of features. For example, we can compute correlation between the counts. It's the most common type of matrices people build, correlation metric. But we can compute other things than correlation. For example, how many times one feature is larger than the other? I mean, how many rows are there such that the value of the first feature is larger than the value of the second one? Or another example, we can compute how many distinct combinations the features have in the dataset. With such custom functions, we should build the metrics manually, and we can use matshow function from Matplotlib to visualize it like on the slide you see in the right picture below. If the metrics looks like a total mess like in here, we can run some kind of clustering like K-means clustering on the rows and columns of this matrix and reorder the features. That is in the left picture below.



Another visualization that helps to find feature groups is the following: **We calculate the statistics of each feature, for example, mean value, and then plot it against column index**. This plot can look quite random if the columns are shuffled. So, what if we sorted the

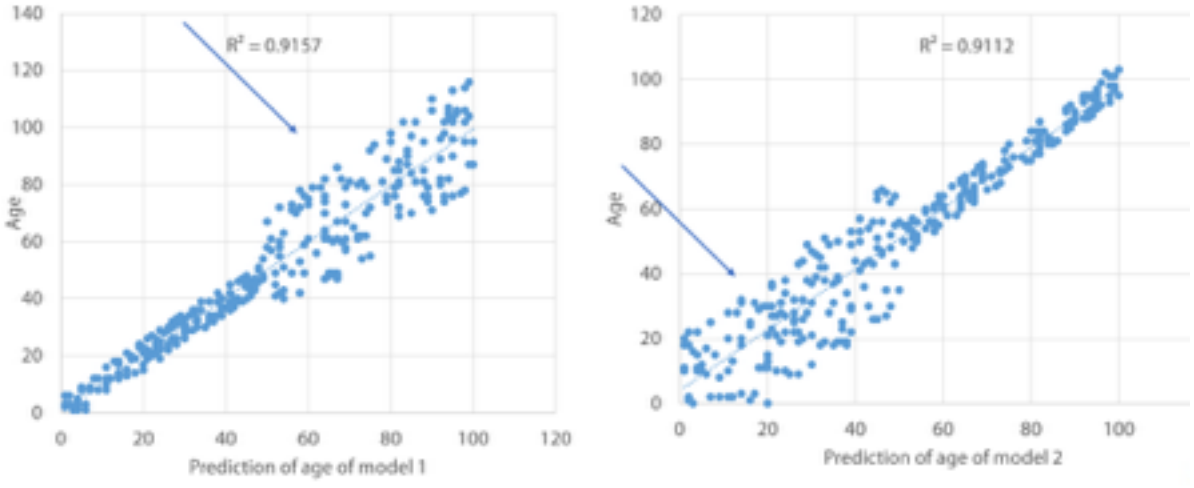
columns based on this statistic? Feature and mean, in this case. It looks like it worked out. We clearly see the groups here. So, now we can take a closer look to each group and use the imagination to generate new features.



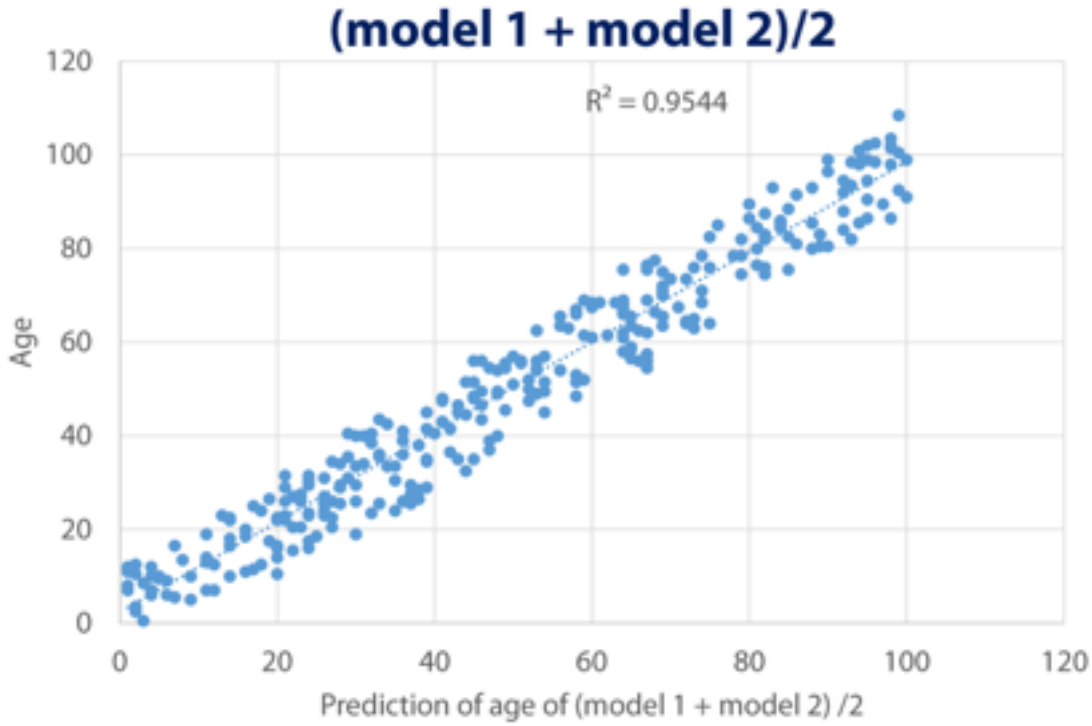
3 Ensemble modelling

What is ensemble modelling? We this term we refer to **combining many different machine learning models** in order to get a more powerful prediction. And later on we will see examples that this happens, that we combine different models and we do get better predictions. There are various ensemble methods. Here we'll discuss a few, those that we encounter quite often, in predictive modelling competitions, and they tend to be, in general, quite competitive. We will start with simple averaging methods, then we'll go to weighted averaging methods, and we will also examine conditional averaging. And then we will move to some more typical ones like bagging, or the very, very popular, boosting, then stacking and StackNet.

3.1 Averaging

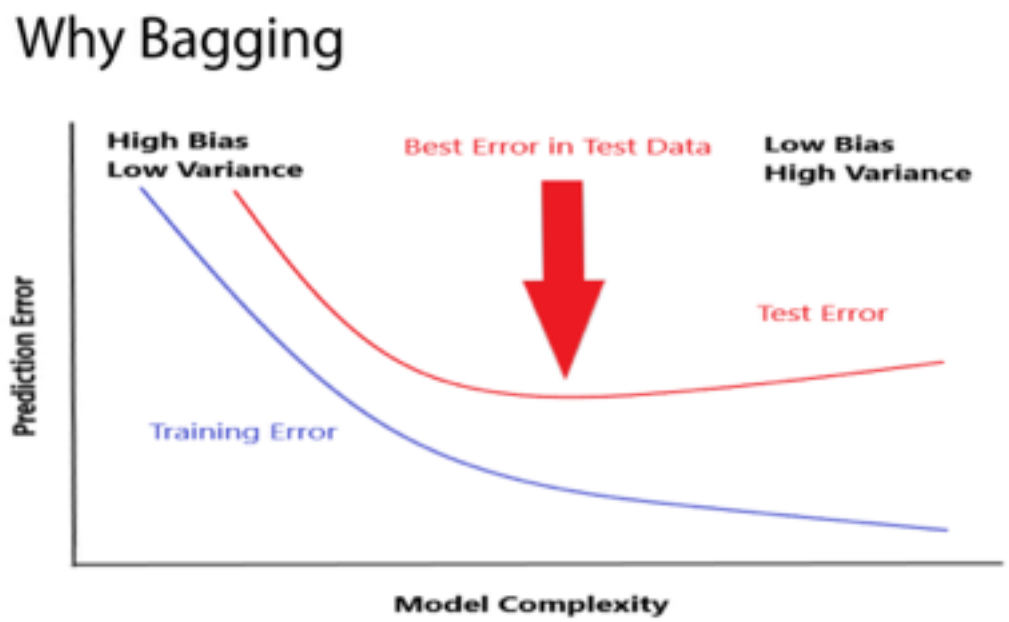


So, in order to help you understand a bit more about the averaging methods, let's take an example. Let's say we have a variable called age, as in age years, and we try to predict this. We have a model that yields prediction for age. Let's assume that the relationship between the two, the actual age in our prediction, looks like in the graph, as in the graph. So you can see that the model boasts quite a higher square of a value of 0.91, but it doesn't do so well in the whole range of values. So when age is less than 50, the model actually does quite well. But when age is more than 50, you can see that the average error is higher. Now let's take another example. Let's assume we have a second model that also tries to predict age, but this one looks like that. As you can see, this model does quite well when age is higher than 50, but not so well when age is less than 50, nevertheless, it scores again 0.91. So we have two models that 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 (model 1 + model two) / 2, so a simple averaging method. The end result will look as in the new graph below.



3.2 Bagging

What is 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 underfitting, and errors due to variance often referred to as overfitting.

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 it's 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.

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.** 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 dating a random sample of columns so bid models on different features or different variables of the data. Then you have **model-specific 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.**

3.3 Boosting

What is 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. So, every model we add sequentially to the ensemble, it takes into account how well the previous models have done in order to make better predictions. 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.

3.3.1 WEIGHT BASED BOOSTING

Rownum	x0	x1	x2	x3	y	pred	abs.error	weight
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

Let's say we have a tabular data set, with four features. Let's call them x0, x1, x2, and x3, and we want to use these features to predict a target variable, y. What we are going to do in weight boosting is, we are going to fit a model, and we will generate predictions. Let's call them pred. These predictions have a certain margin of error. We can calculate these absolute error, and when I say absolute error, is absolute of y minus our prediction. You can see there

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 a new 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.

3.3.2 RESIDUAL BASED BOOSTING

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

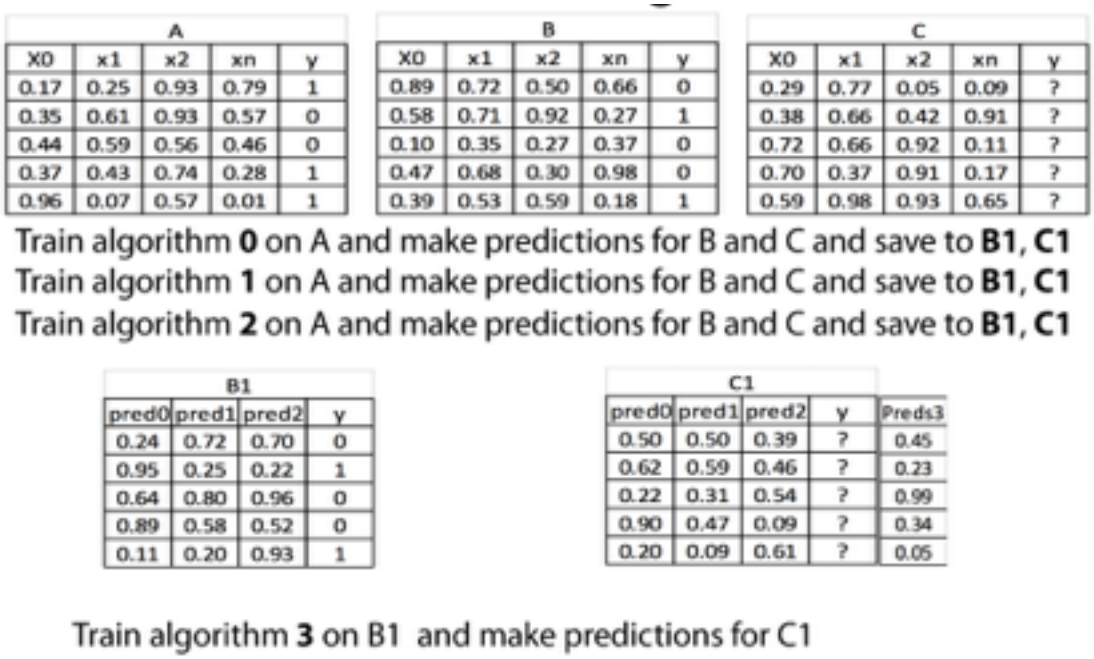
Let's say we have again the same dataset, same features, again when trying to predict a y variable, we fit a model, we make predictions. What we do next, is we'll **calculate the error of these predictions but this time, not in absolute terms because we're interested about the direction of the error.** 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.

3.3.3 IMPLEMENTATIONS

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.

3.4 Stacking

Going through the definition of stacking, it essentially means making several predictions with hold-out data sets. And then collecting or stacking these predictions to form a new data set, where you can fit a new model on it, on this newly-formed data set from predictions.



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

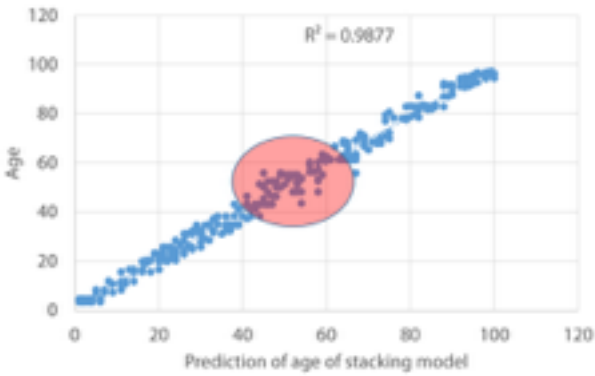
C1

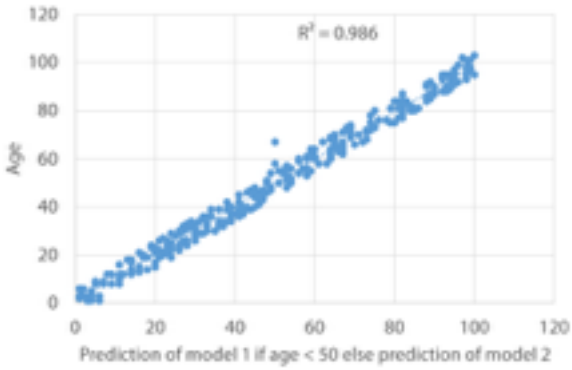
pred0	pred1	pred2	y	Preds3
0.50	0.50	0.39	?	0.45
0.62	0.59	0.46	?	0.23
0.22	0.31	0.54	?	0.99
0.90	0.47	0.09	?	0.34
0.20	0.09	0.61	?	0.05

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

So, let's go more into the methodology of stacking: stacking was introduced 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.

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.