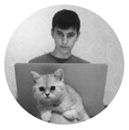


Yury Kashnitskiy Follow
Data Scientist at Mail.Ru Group
Mar 5 · 15 min read

# Open Machine Learning Course. Topic 5. Bagging and Random Forest

Source: https://music.yandex.ua/artist/3177683

In the previous articles, you saw different classification algorithms as well as techniques for how to properly validate and evaluate the quality of your models.

Now, suppose that you have chosen the best possible model for a particular problem and are struggling to further improve its accuracy. In this case, you would need to apply some more advanced machine learning techniques that are collectively referred to as *ensembles*.

An *ensemble* is a set of elements that collectively contribute to a whole. A familiar example is a musical ensemble, which blends the sounds of several musical instruments to create a beautiful harmony, or architectural ensembles, which are a set of buildings designed as a unit. In ensembles, the (whole) harmonious outcome is more important than the performance of any individual part.

## Article outline

1. Ensembles
2. Bootstraping
3. Bagging
4. Out-of-bag error
5. Random Forest
6. Feature importance
7. Assignment #5
8. Useful resources

## 1. Ensembles

Condorcet's jury theorem (1784) is about an ensemble in some sense. It states that, if each member of the jury makes an independent judgement and the probability of the correct decision by each juror is more than 0.5, then the probability of the correct decision by the whole jury increases with the total number of jurors and tends to one. On the other hand, if the probability of being right is less than 0.5 for each juror, then the probability of the correct decision by the whole jury decreases with the number of jurors and tends to zero.

One can find an analytic expression for this theorem in a Jupyter notebook accompanying this article.

Let's look at another example of ensembles: an observation known as Wisdom of the crowd. In 1906, Francis Galton visited a country fair in Plymouth where he saw a contest being held for farmers. 800 participants tried to estimate the weight of a slaughtered bull. The real weight of the bull was 1198 pounds. Although none of the farmers could guess the exact weight of the animal, the average of their predictions was 1197 pounds.

A similar idea for error reduction was adopted in the field of Machine Learning.

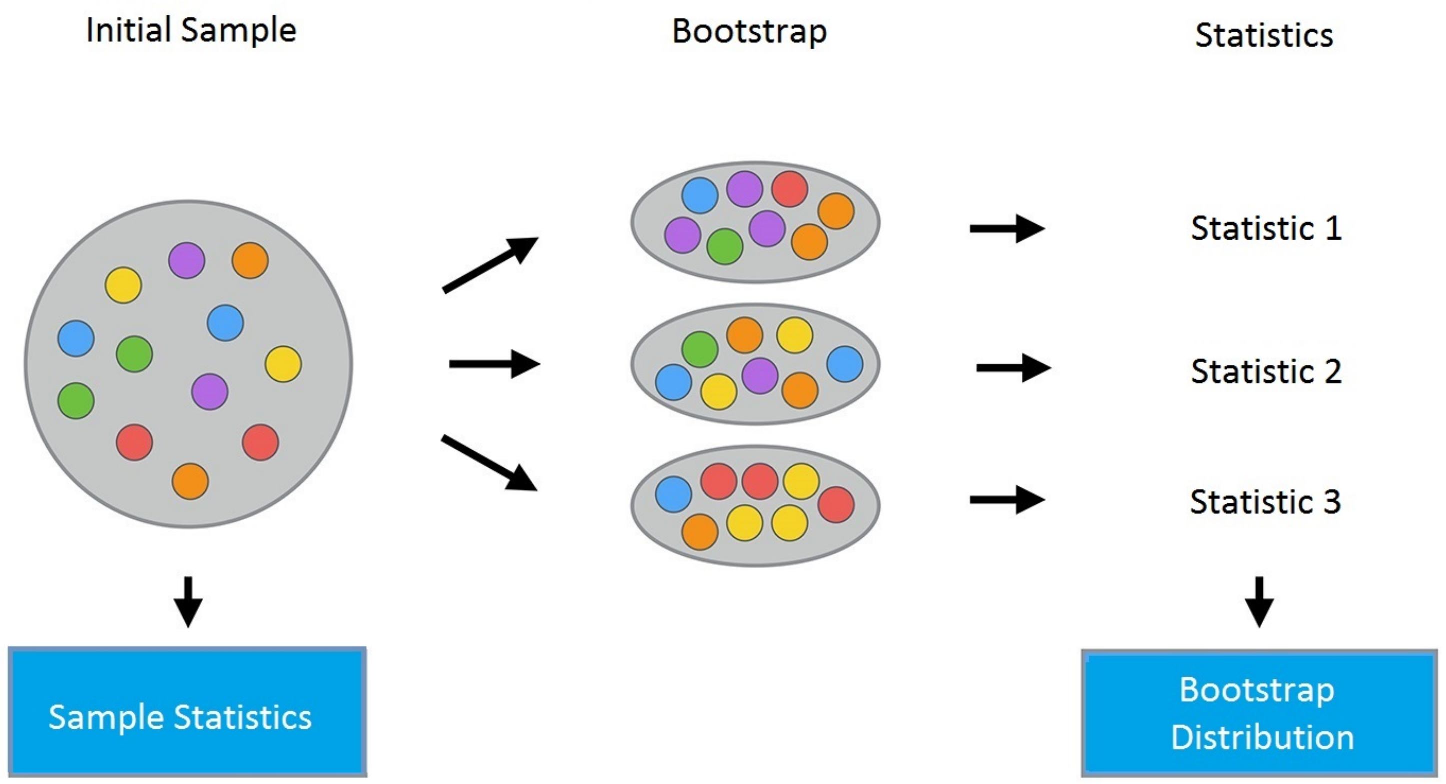
## 2. Bootstraping

*Bagging* (also known as Bootstrap aggregation) is one of the first and most basic ensemble techniques. It was proposed by Leo Breiman in 1994. Bagging is based on the statistical method of bootstraping, which makes the evaluation of many statistics of complex models feasible.

The bootstrap method goes as follows. Let there be a sample **X** of size **N**. We can make a new sample from the original sample by drawing **N** elements from the latter randomly and uniformly, with replacement. In other words, we select a random element from the original sample of size **N** and do this **N** times. All elements are equally likely to be selected, thus each element is drawn with the equal probability **1/N**.

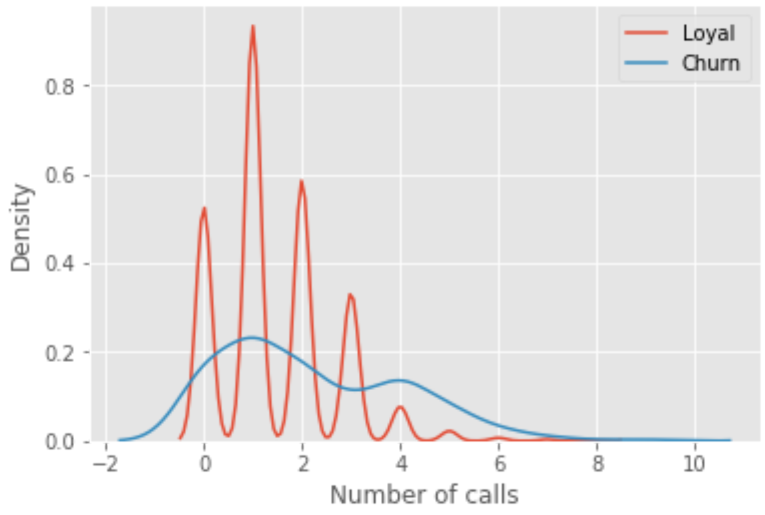
Let's say we are drawing balls from a bag one at a time. At each step, the selected ball is put back into the bag so that the next selection is made equiprobably i.e. from the same number of balls **N**. Note that, because we put the balls back, there may be duplicates in the new sample. Let's call this new sample **X1**.

By repeating this procedure **M** times, we create **M** *bootstrap samples* **X1**, …, **XM**. In the end, we have a sufficient number of samples and can compute various statistics of the original distribution.



For our example, we'll use the familiar `telecom_churn` dataset. Previously, when we discussed feature importance, we saw that one of the most important features in this dataset is the number of calls to customer service. Let's visualize the data and look at the distribution of this feature.

```
import pandas as pd
from matplotlib import pyplot as plt
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = 10, 6
import seaborn as sns
%matplotlib inline
telecom_data = pd.read_csv('../../data/telecom_churn.c
fig = sns.kdeplot(telecom_data[telecom_data['Churn'] =
                  label = 'Loyal')
```



As you can see, loyal customers make fewer calls to customer service than those who eventually left. Now, it might be a good idea to estimate the average number of customer service calls in each group. Since our dataset is small, we would not get a good estimate by simply calculating the mean of the original sample. We will be better off applying the bootstrap method. Let's generate 1000 new bootstrap samples from our original population and produce an interval estimate of the mean.

```python
import numpy as np
def get_bootstrap_samples(data, n_samples):
    """Generate bootstrap samples using the bootstrap
    indices = np.random.randint(0, len(data), (n_sampl
    samples = data[indices]
    return samples
def stat_intervals(stat, alpha):
    """Produce an interval estimate."""
    boundaries = np.percentile(stat, [100 * alpha / 2.
    return boundaries
# Save the data about the loyal and former customers t
loyal_calls = telecom_data[telecom_data['Churn']
                          == False]['Customer service
churn_calls= telecom_data[telecom_data['Churn']
                         == True]['Customer service c
# Set the seed for reproducibility of the results
np.random.seed(0)
# Generate the samples using bootstrapping and calcula
```

```
Service calls from loyal: mean interval [1.4077193
1.49473684] # Service calls from churn: mean interval
[2.0621118 2.39761905]
```

In the end, we see that, with 95% probability, the average number of customer service calls from loyal customers lies between 1.4 and 1.49 while the churned clients called 2.06 through 2.40 times on average. Also, note that the interval for the loyal customers is narrower, which is reasonable since they make fewer calls (0, 1 or 2) in comparison with the churned clients who called until they became fed up and switched providers.

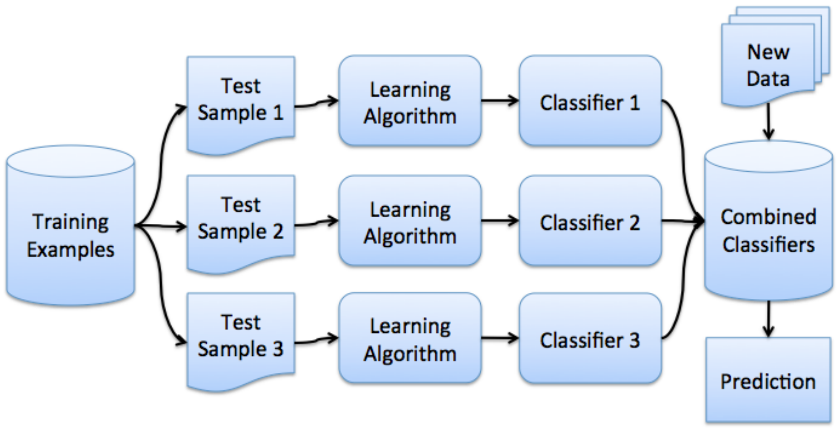
## 3. Bagging

Now that you've grasped the idea of bootstrapping, we can move on to *bagging*.

Suppose that we have a training set **X**. Using bootstrapping, we generate samples **X1**, …, **XM**. Now, for each bootstrap sample, we train its own classifier **ai(x)**. The final classifier will average the outputs from all these individual classifiers. In the case of classification, this technique corresponds to voting:

$$a(x) = \frac{1}{M}\sum_{i=1}^{M} a_i(x).$$

The picture below illustrates this algorithm:



In a regression problem, by averaging individual answers, bagging reduces the mean squared error by a factor of **M**, the number of regressors. See the mathematical proof in our Jupyter notebook.

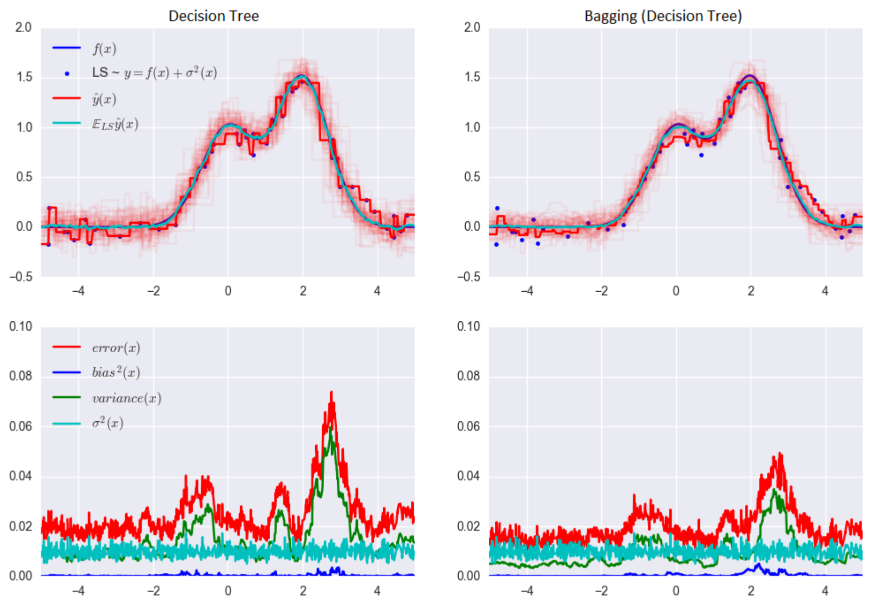
From our previous lesson, let’s recall the components that make up the total out-of-sample error:

$$\begin{aligned}\text{Err}(\vec{x}) &= \mathbb{E}\left[\left(y - \hat{f}\,(\vec{x})\right)^2\right] \\ &= \sigma^2 + f^2 + \text{Var}\left(\hat{f}\right) + \mathbb{E}\left[\hat{f}\right]^2 - 2f\,\mathbb{E}\left[\hat{f}\right] \\ &= \left(f - \mathbb{E}\left[\hat{f}\right]\right)^2 + \text{Var}\left(\hat{f}\right) + \sigma^2 \\ &= \text{Bias}\left(\hat{f}\right)^2 + \text{Var}\left(\hat{f}\right) + \sigma^2\end{aligned}$$

Bagging reduces the variance of a classifier by decreasing the difference in error when we train the model on different datasets. In other words, bagging prevents overfitting. The efficacy of bagging comes from the fact that individual models are quite different due to the different training data and their errors cancel out during voting. Additionally, outliers are likely omitted in some of the training bootstap samples.

The `scikit-learn` library supports bagging with meta-estimators `BaggingRegressor` and `BaggingClassifier`. You can use most of the algorithms as a base.

Let's examine how bagging works in practice and compare it with the decision tree. For this, we will use an example from sklearn's documentation.



The error for the decision tree:

0.0255 = 0.0003 *(bias²)* + 0.0152 *(variance)* + 0.0098 *(σ²)*

The error when using bagging:

0.0196 = 0.0004 *(bias²)* + 0.0092 *(variance)* + 0.0098 (σ²)

As you can see from the graph above, the variance in the error is much lower for bagging. Remember that we have already proved this theoretically.

Bagging is effective on small datasets. Dropping even a small part of training data leads to constructing substantially different base classifiers. If you have a large dataset, you would generate bootstrap samples of a much smaller size.

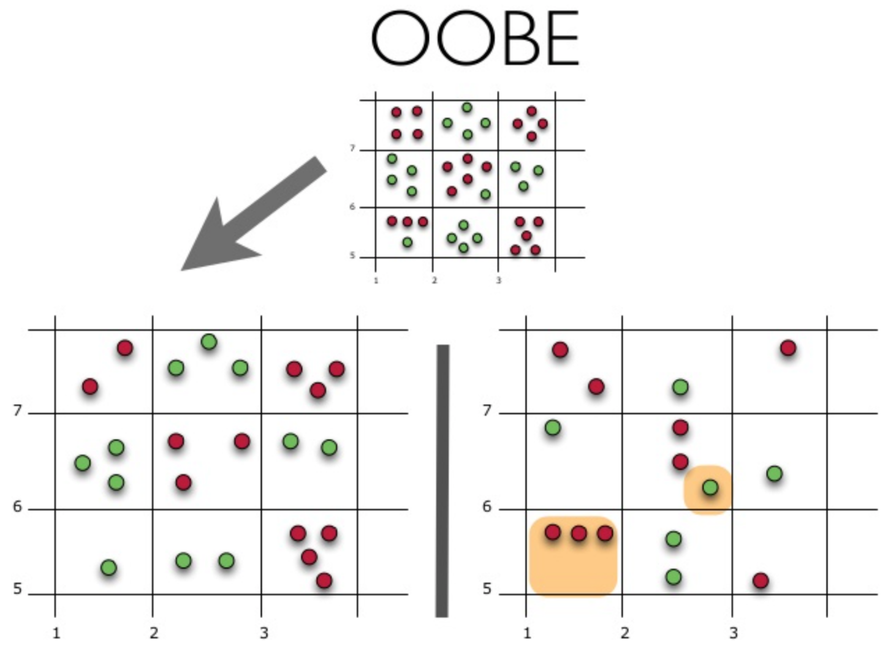
The example above is unlikely to be applicable to any real work. This is because we made a strong assumption that our individual errors are uncorrelated. More often than not, this is way too optimistic for real-world applications. When this assumption is false, the reduction in error will not be as significant. In the following lectures, we will discuss some more sophisticated ensemble methods, which enable more accurate predictions in real-world problems.

## 4. Out-of-bag error

Looking ahead, in case of Random Forest, there is no need to use cross-validation or hold-out samples in order to get an unbiased error estimation. Why? Because, in ensemble techniques, the error estimation takes place internally.

Random trees are constructed using different bootstrap samples of the original dataset. Approximately 37% of inputs are left out of a particular bootstrap sample and are not used in the construction of the **K**-th tree. The proof is easy and elegant, it's done in our Jupyter notebook.

Let's visualize how **O**ut-**o**f-**B**ag **E**rror (or OOBE) estimation works:



The top part of the figure above represents our original dataset. We split it into the training (left) and test (right) sets. In the left image, we draw a grid that perfectly divides our dataset according to classes. Now, we use the same grid to estimate the share of the correct answers on our test set. We can see that our classifier gave incorrect answers in those 4 cases that have not been used during training (on the left). Hence, the accuracy of our classifier is 11/15*100% = 73.33%.

To sum up, each base algorithm is trained on ~ 63%$ of the original examples. It can be validated on the remaining ~37%. The Out-of-Bag estimate is nothing more than the mean estimate of the base algorithms on those ~37% of inputs that were left out of training.

## 5. Random Forest

Leo Breiman managed to apply bootstrapping not only in statistics but also in machine learning. He, along with Adel Cutler, extended and improved the Random Forest algorithm proposed by Tin Kam Ho. They combined the construction of uncorrelated trees using CART, bagging, and the random subspace method.

Decision trees are a good choice for the base classifier in bagging because they are quite sophisticated and can achieve zero classification error on any sample. The random subspace method reduces the correlation between the trees and thus prevents overfitting. With bagging, the base algorithms are trained on different random subsets of the original feature set.

The following algorithm constructs an ensemble of models using the random subspace method:

1. Let the number of instance be equal to **n**, and the number of feature dimensions be equal to **d**.
2. Choose **M** as the number of individual models in the ensemble.
3. For each model **m**, choose the number of features **dm < d**. As a rule, the same value of **dm** is used for all the models.
4. For each model **m**, create a training set by selecting **dm** features at random from the whole set of **d** features.
5. Train each model.
6. Apply the resulting ensemble model to a new input by combining the results from all the models in **M**. You can use either majority voting or aggregation of the posterior probabilities.

## Outline of part 5

1. Algorithm
2. Comparison with Decision Trees and Bagging
3. Parameters
4. Extremely Randomized Trees
5. Transformation of a dataset into a high-dimensional representation
6. Pros and cons of Random Forests

## 5.1. Algorithm

The algorithm for constructing a random forest of **N** trees goes as follows:

- For each $k = 1, \dots, N$:
  - Generate a bootstrap sample $X_k$.
  - Build a decision tree $b_k$ on the sample $X_k$:
    - Pick the best feature dimension according to the given criteria. Split the sample by this feature to create a new tree level. Repeat this procedure until the sample is exhausted.
    - Building the tree until any of its leaves contains no more than $n_{\min}$ instances or until a certain depth is reached.
    - For each split, we first randomly pick $m$ features from the $d$ original ones and then search for the next best split only among the subset.

The final classifier is defined by:

$$a(x) = \frac{1}{N} \sum_{k=1}^{N} b_k(x)$$

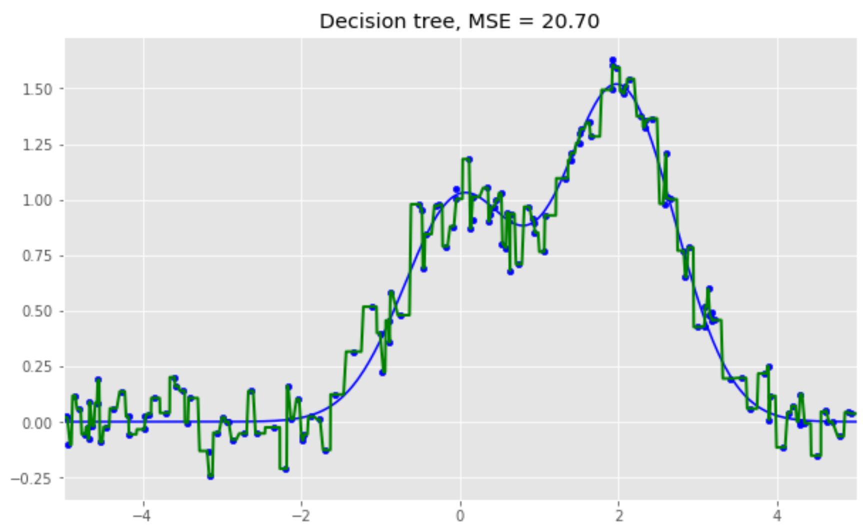
We use the majority voting for classification and the mean for regression.

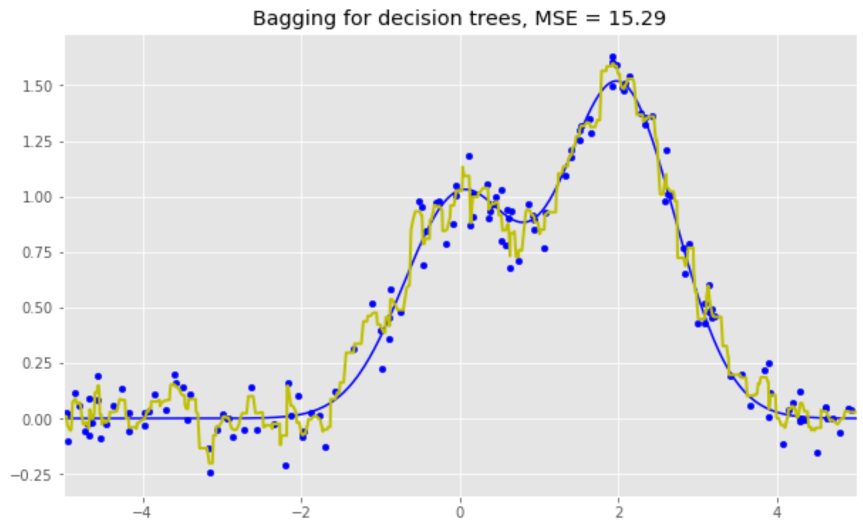
For classification problems, it is advisable to set **m** to be equal the square root of **d**. For regression problems, we usually take **m = d/3**, where **d** is the number of features. It is recommended to build each tree until all of its leaves contain only 1 instance for classification and 5 instances for regression.

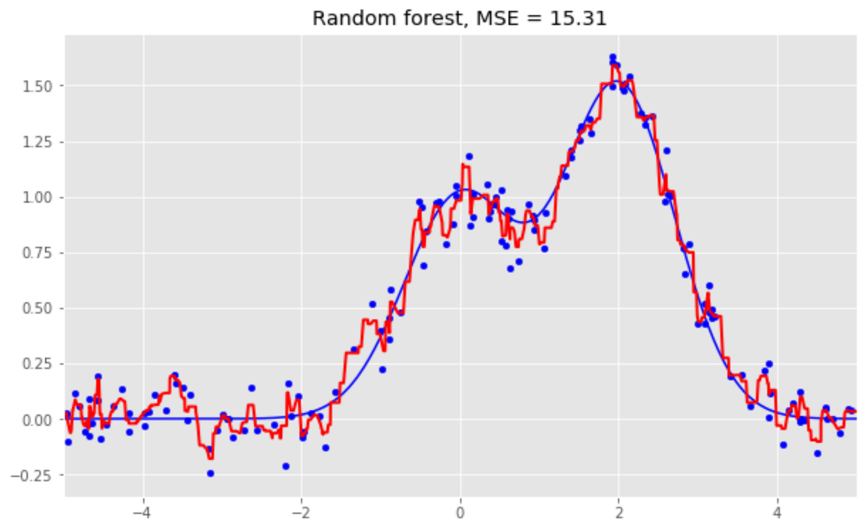
You can see Random Forest as bagging of decision trees with the modification of selecting a random subset of features at each split.

## 5.2. Comparison with Decision Trees and Bagging

*See the code in this Jupyter notebook*


Decision tree, MSE = 20.70
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−4
−2
0
2
4

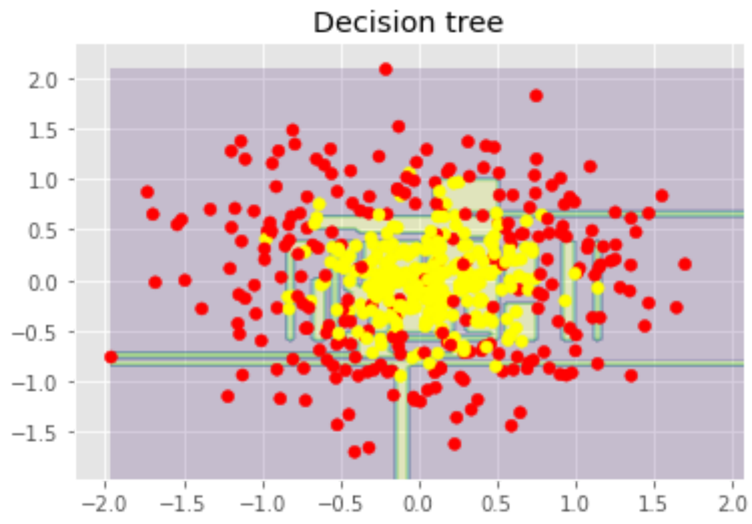

Bagging for decision trees, MSE = 15.29
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−4
−2
0
2
4

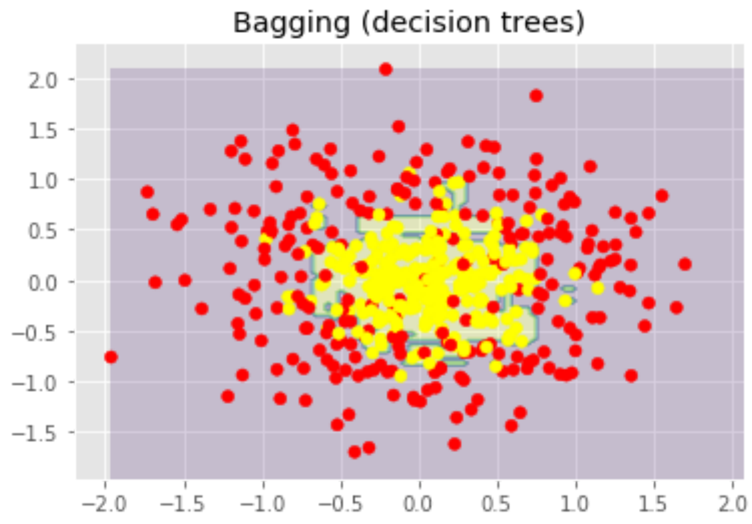

Random forest, MSE = 15.31
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−4
−2
0
2
4

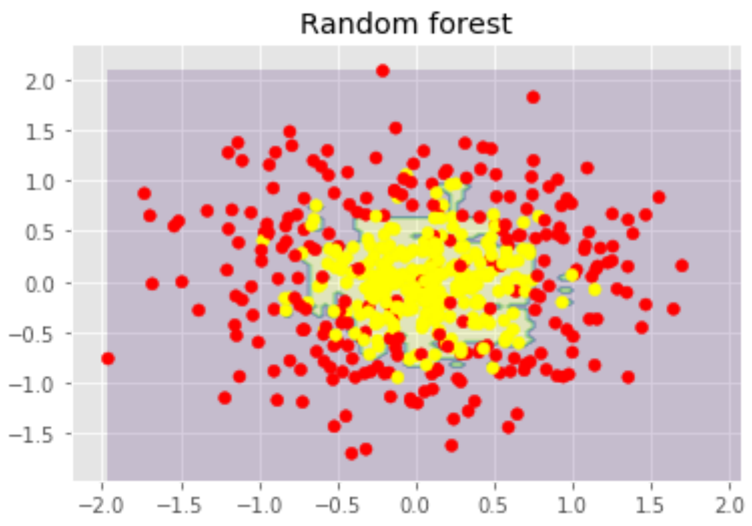
As we can see from our graphs and the MSE values above, a Random Forest of 10 trees achieves a better result than a single decision tree or bagging with 10 trees. The main difference between Random Forests and bagging is that, in a Random Forest, the best feature for a split is selected from a random subset of the available features while, in bagging, all features are considered for the next best split.

We can also look at the advantages of Random Forests and bagging in classification problems.

*See the code in this Jupyter notebook*







The figures above show that the decision boundary of the decision tree is quite jagged and has a lot of acute angles that suggest overfitting and a weak ability to generalize. We would have trouble making reliable predictions on new test data. In contrast, the bagging algorithm has a rather smooth boundary and has no obvious signs of overfitting.

Now, let's investigate some parameters which can help us increase the model accuracy.

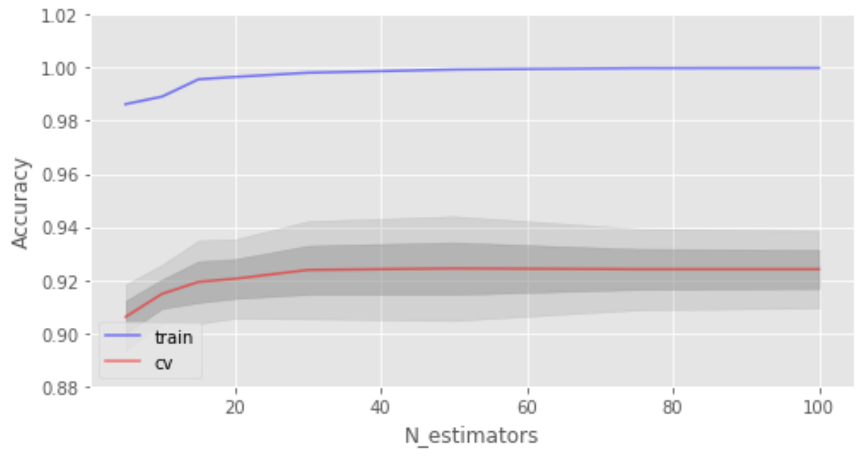
## 5.3. Parameters

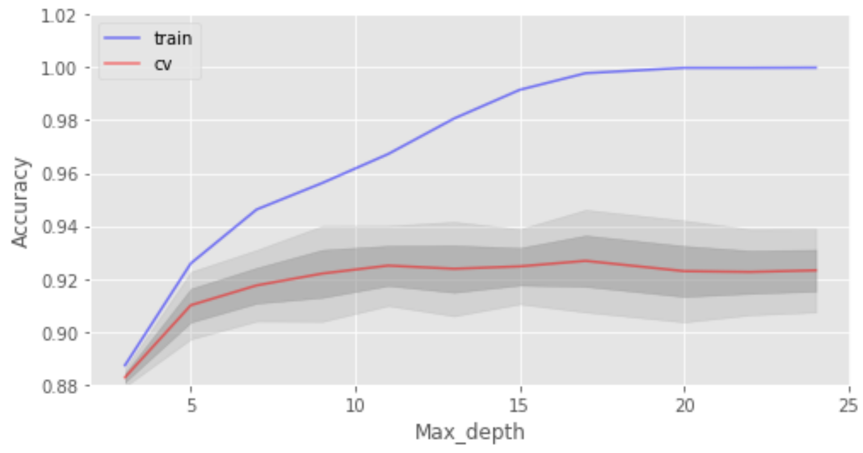
The scikit-learn library implements random forests by providing two estimators: `RandomForestClassifier` and `RandomForestRegressor`.

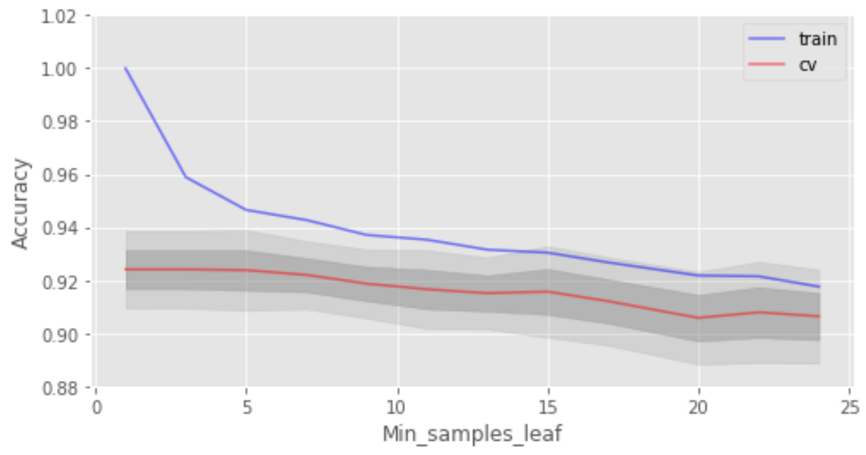
Below are the parameters which we need to pay attention to when we are building a new model:

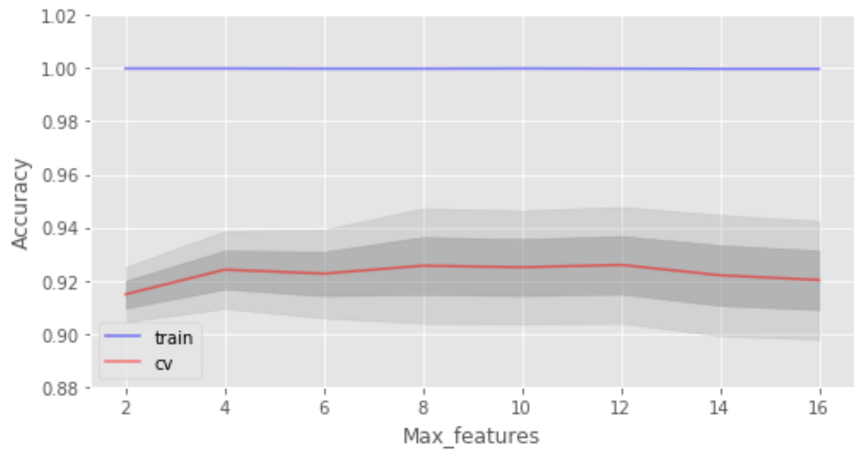
- `n_estimators` is the number of trees in the forest;
- `criterion` is the function used to measure the quality of a split;
- `max_features` is the number of features to consider when looking for the best split;
- `min_samples_leaf` is the minimum number of samples required to be at a leaf node;
- `max_depth` is the maximum depth of the tree.

Please refer to this Jupyter notebook to find an example of Random Forest tuning in a telecom churn problem. We get the following validation curves.









The most important fact about Random Forests is that it's accuracy doesn't decrease when we add trees, so the number of trees is not a *compexity* hyperparameter as opposed to `max_depth` and `min_samples_leaf.` This means that you can tune hyperparameters with, say, 10 trees, and then increase the number of trees till 500 and be sure that accuracy will only get better.

In this notebook you'll also find more analysis of bias and variance for Random Forests and it's analogies with k Nearset Neighbors.

## 5.4. Extremely Randomized Trees

Extremely Randomized Trees employ a greater degree of randomization at the cut-point choice when splitting a tree node. As in Random Forests, a random subset of features is used. But, instead of the search for optimal thresholds, their values are selected at random for each possible feature, and the best one among these randomly generated thresholds is used as the best rule to split the node. This usually trades off a slight reduction in the model variance with a small increase of bias.

In the `scikit-learn` library, there are 2 implementations of Extremely Randomized Trees: ExtraTreesClassifier and ExtraTreesRegressor.

This method should be used if you have greatly overfit with Random Forests or gradient boosting.

## 5.5. Transformation of a dataset into a high-dimensional representation

Random forests are mostly used in supervised learning, but there is a way to apply them in the unsupervised setting.

Using the `scikit-learn` method RandomTreesEmbedding, we can transform our dataset into a high-dimensional, sparse representation. We first build extremely randomized trees and then use the index of the leaf containing the example as a new feature.

For example, if the input appears in the first leaf, we assign 1 as the feature value; if not, we assign 0. This is a so-called *binary coding*. We can control the number of features and the sparseness of data by increasing or decreasing the number of trees and their depth. Because nearby data points are likely to fall into the same leaf, this transformation provides an implicit nonparametric estimate of their density.

## 5.6. Pros and cons of Random Forests

### Pros:

- High prediction accuracy; will perform better than linear algorithms in most problems; the accuracy is comparable with that of boosting;
- Robust to outliers, thanks to random sampling;
- Insensitive to feature scaling as well as any other monotonic transformations due to the random subspace selection;
- Doesn't require fine-grained parameter tuning, works quite well out-of-the-box. With tuning, it is possible to achieve a 0.5–3% gain in accuracy, depending on the problem setting and data;
- Efficient for datasets with a large number of features and classes;
- Handles both continuous and discrete variables equally well;
- Rarely overfits. In practice, an increase in the tree number almost always improves the composition. But, after reaching a certain number of trees, the learning curve is very close to the asymptote;
- There are developed methods to estimate feature importance;

- Works well with missing data and maintains good accuracy levels even when a large part of data is missing;
- Provides means to weight classes on the whole dataset as well as for each tree sample;
- Under the hood, calculates proximities between pairs of instamces that can subsequantly be used in clustering, outlier detection, or interesting data representations;
- The above functionality and properties may be extended to unlabeled data to enable unsupervised clustering, data visualization, and outlier detection;
- Easily parallelized and highly scalable.

### Cons:

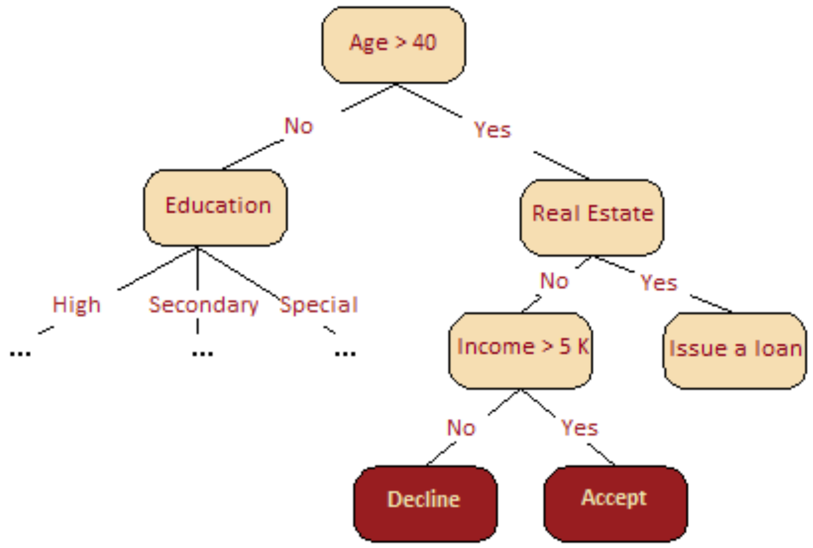
- In comparison with a single decision tree, Random Forest's output is more difficult to interpret.
- There are no formal p-values for feature importance estimation.
- Performs worse than linear methods in the case of sparse data: text inputs, bag of words, etc.;
- Unlike linear regression, Random Forest is unable to extrapolate. But, this can be also regarded as an advantage because outliers do not cause extreme values in Random Forests;
- Prone to overfitting in some problems, especially, when dealing with noisy data;
- In the case of categorical variables with varying level numbers, Random Forests favor variables with a greater number of levels. The tree will fit more towards a feature with many levels because this gains greater accuracy;
- If a dataset contains groups of correlated features with similar importance for predicted classes, then the preference will be given to smaller groups;
- The resulting model is large and requires a lot of RAM.

## 6. Feature importance

Pretty often, you would like to make out the exact reasons why the algorithm outputted a particular answer. Or, not being able to understand the algorithm completely, then st lest we'd like to find out which input features contributed the most to the result. With Random Forest, you can obtain such information quite easily.

## 6.1. Essence of the method

From the picture below, it is intuitively clear that, in our credit scoring problem, *Age* is much more important than *Income*. This can be formally explained using the concept of *information gain*.



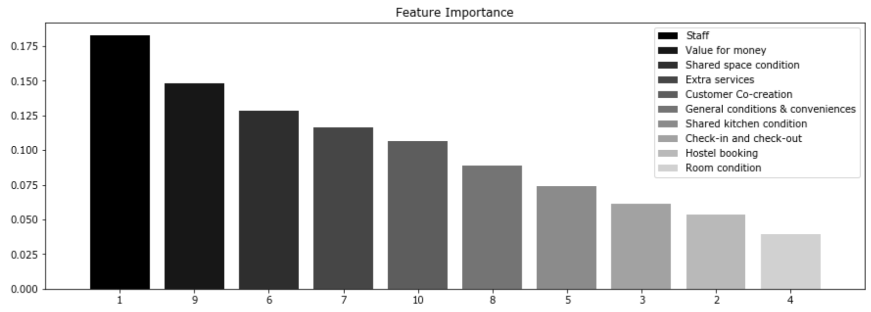
In the case of many decision trees (i.e. a Random Forest), the closer the mean position of a feature over all trees to the root, the more important it is for a given classification or regression problem. Gains in the splitting criterion, such as the *Gini impurity*, obtained at each optimal split in every tree, is a measure of importance that is directly associated with the splitting feature. The value of this score is distinct for each feature and accumulates over all the trees.

We go a little deeper into details in this notebook.

## 6.2. Practical example

*The code can also be found in a notebook.*

Let’s consider the results of a survey given to visitors of hostels listed on Booking.com and TripAdvisor.com. Our features here are the average ratings for different categories include service quality, room condition, value for money, etc. Our target variable is the hostel’s overall rating on the website.



The picture above shows that, more often than not, customers pay great attention to staff and the price-quality ratio. This couple of factors affects the resulting overall rating the most. But, the difference between these two features and other festures is not very large. We can therefore conclude that exclusion of any of these features will lead to a reduction in model accuracy. Based on our analysis, we csn recommend hostel owners to focus primarily on staff training and price-to-quality ratio.

## 7. Assignment #5

In the fifth assignment, you’ll compare Random Forest and logistic regression in the credit scoring task. You’ll understand pros and cons of both approaches. There are also some questions to strengthen your understanding of the material of this article.

We suggest that you complete the tasks in the Jupyter notebook, and then answer 12 questions in this form. You can edit your responses even after submitting the form.

**Hard deadline**: March 18, 23:59 CET.

## 8. Useful resources

- Chapter 15 of the Elements of Statistical Learning by Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie.

- More about practical applications of random forests and other algorithms can be found in the official documentation of `scikit-learn`.
- For more in-depth discussion of variance and decorrelation of random forests, see the original paper.

. . .

*Authors: Yury Kashnitsky, Data Scientist at Mail.Ru Group, and Vitaliy Radchenko, Data Scientist at You Scan. Translated and edited by Christina Butsko, Anna Shirshova, Anastasia Manokhina, Egor Polusmak, and Yuanyuan Pao.*