What is XGBoost and why is it so popular?

Let me introduce you to the hottest Machine Learning library in the ML community — XGBoost. In recent years, it has been the main driving force behind the algorithms that win massive ML competitions. Its speed and performance are unparalleled and it consistently outperforms any other algorithms aimed at supervised learning tasks.

The library is parallelizable which means the core algorithm can run on clusters of GPUs or even across a network of computers. This makes it feasible to solve ML tasks by training on hundreds of millions of training examples with high performance.

Originally, it was written in C++ as a command-line application. After winning a huge competition in the field of physics, it started being widely adopted by the ML community. As a result, now the library has its APIs in several other languages including Python, R, and Julia.

In this post, you will learn the fundamentals of XGBoost to solve classification tasks, an overview of the massive list of XGBoost's hyperparameters and how to tune them.

Refresher on Terminology

Before we move on to code examples of XGBoost, let's refresh on some of the terms we will be using throughout the post.

Classification task: a supervised machine learning task in which one should predict if an instance is in some category by studying the instance's features. For example, by looking at the body measurements, patient history, and glucose levels of a person, you can predict whether a person belongs to the 'Has diabetes' or 'Does not have diabetes' group.

Binary classification: One type of classification where the target instance can only belong to either one of two classes. For example, predicting whether an email is a spam or not, whether a customer purchases some product or not, etc.

Multi-class classification: Another type of classification problem where the target can belong to one of many categories. For example, predicting the species of a bird, guessing someone's bloody type, etc.

If you find yourself confused by other terminology, I have written a small ML dictionary for beginners:

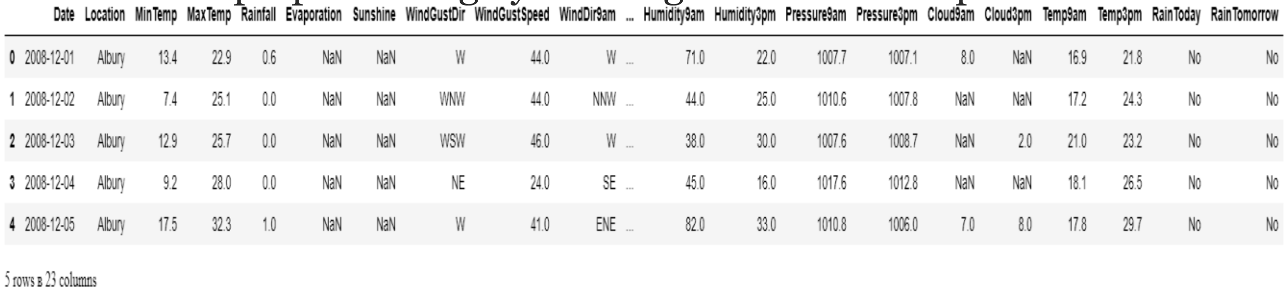
How to preprocess your datasets for XGBoost

Apart from basic data cleaning operations, there are some requirements for XGBoost to achieve top performance. Mainly:

- Numeric features should be scaled

- Categorical features should be encoded

To show how these steps are done, we will be using the Rain in Australia dataset from Kaggle where we will predict whether it will rain today or not based on some weather measurements. In this section, we will focus on preprocessing by utilizing Scikit-Learn Pipelines.



	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am	...	Humidity9am	Humidity3pm	Pressure9am	Pressure3pm	Cloud9am	Cloud3pm	Temp9am	Temp3pm	RainToday	RainTomorrow
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	W	...	71.0	22.0	1007.7	1007.1	8.0	NaN	16.9	21.8	No	No
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	NNW	...	44.0	25.0	1010.6	1007.8	NaN	NaN	17.2	24.3	No	No
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0	W	...	38.0	30.0	1007.6	1008.7	NaN	2.0	21.0	23.2	No	No
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0	SE	...	45.0	16.0	1017.6	1012.8	NaN	NaN	18.1	26.5	No	No
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0	ENE	...	82.0	33.0	1010.8	1006.0	7.0	8.0	17.8	29.7	No	No

5 rows в 23 columns

The dataset contains weather measures of 10 years from multiple weather stations in Australia. You can either predict whether it will rain tomorrow or today, so there are two targets in the dataset named `RainToday`, `RainTomorrow`.

Since we will only be predicting for `RainToday`, we will drop the other one along with some other features that won't be necessary:

```
cols_to_drop = ["Date", "Location", "RainTomorrow", "Rainfall"]

rain.drop(cols_to_drop, axis=1, inplace=True)
```

Dropping the `Rainfall` column is a must because it records the amount of rain in millimeters.

Next, let's deal with missing values starting by looking at their proportions in each column:

If the proportion is higher than 40% we will drop the column:

Three columns contain more than 40% missing values. We will drop them:

Now, before we move on to pipelines, let's divide the data into feature and target arrays beforehand:

Next, there are both categorical and numeric features. We will build two separate pipelines and combine them later.

For the categorical features, we will impute the missing values with the mode of the column and encode them with One-Hot encoding:

For the numeric features, I will choose the mean as an imputer and `StandardScaler` so that the features have 0 mean and a variance of 1:

Finally, we will combine the two pipelines with a column transformer. To specify which columns the pipelines are designed for, we should first isolate the categorical and numeric feature names:

Next, we will input these along with their corresponding pipelines into a `ColumnTransFormer` instance:

The full pipeline is finally ready. The only thing missing is the XGBoost classifier, which we will add in the next section.

An Example of XGBoost For a Classification Problem

To get started with `xgboost`, just install it either with `pip` or `conda`:

```
# pip
pip install xgboost

# conda
conda install -c conda-forge xgboost
```

After installation, you can import it under its standard alias — `xgb`. For classification problems, the library provides `XGBClassifier` class:

Fortunately, the classifier follows the familiar fit-predict pattern of `sklearn` meaning we can freely use it as any `sklearn` model.

Before we train the classifier, let’s preprocess the data and divide it into train and test sets:

Since the target contains `NaN`, I imputed it by hand. Also, it is important to pass `y_processed` to `stratify` so that the split contains the same proportion of categories in both sets.

Now, we fit the classifier with default parameters and evaluate its performance:

Even with default parameters, we got an 85% accuracy which is reasonably good. In the next sections, we will try to improve the model even further by using `GridSearchCV` offered by Scikit-learn.

What Powers XGBoost Under the Hood

Unlike many other algorithms, XGBoost is an ensemble learning algorithm meaning that it combines the results of many models, called **base learners** to make a prediction.

Just like in Random Forests, XGBoost uses Decision Trees as base learners:

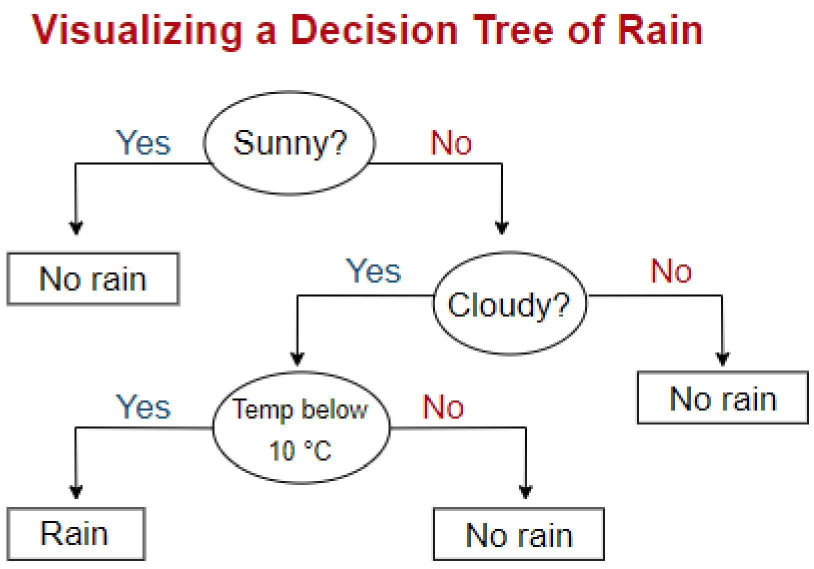


Image by the author. Decision tree to predict rain

An example of a decision tree can be seen above. In each decision node (circles), there is a single question that is being asked with only two possible answers. At the bottom of each tree, there is a single decision (rectangles). In the above tree, the first question is whether it is sunny or not. If yes, you immediately decide that it is not going to rain. If otherwise, you continue to ask more binary (yes/no) questions that ultimately will lead to some decision at the last “leaf” (rectangle).

Individual decision trees are low-bias, high-variance models. They are incredibly good at finding the relationships in any type of training data but struggle to generalize well on unseen data.

However, the trees used by XGBoost are a bit different than traditional decision trees. They are called CART trees (Classification and Regression trees) and instead of containing a single decision in each "leaf" node, they contain real-value scores of whether an instance belongs to a group. After the tree reaches max depth, the decision can be made by converting the scores into categories using a certain threshold.

I am in no way an expert when it comes to the internals of XGBoost. That's why I recommend you to check out this awesome YouTube playlist entirely on XGBoost and another one solely aimed at Gradient Boosting which I did not mention at all.

Overview of XGBoost Classifier Hyperparameters

So far, we have been using only the default hyperparameters of the XGBoost Classifier:

Terminology refresher: hyperparameters *of a model are the settings of that model which should be provided by the user. The model itself cannot learn these from the given training data.*

It has quite a few as you can see. Even though, we achieved reasonably good results with the defaults, tuning the above parameters might result in a significant increase in performance. But before we get to tuning, let’s look at the overview of the most frequently tuned hyperparameters:

1. `learning_rate`: also called *eta*, it specifies how quickly the model fits the residual errors by using additional base learners.

- typical values: 0.01–0.2

2. `gamma, reg_alpha`, `reg_lambda`: these 3 parameters specify the values for 3 types of regularization done by XGBoost - minimum loss reduction to create a new split, L1 reg on leaf weights, L2 reg leaf weights respectively

- typical values for `gamma`: 0 - 0.5 but highly dependent on the data
- typical values for `reg_alpha` and `reg_lambda`: 0 - 1 is a good starting point but again, depends on the data

3. `max_depth` - how deep the tree's decision nodes can go. Must be a positive integer

- typical values: 1–10

4. `subsample` - fraction of the training set that can be used to train each tree. If this value is low, it may lead to underfitting or if it is too high, it may lead to overfitting

- typical values: 0.5–0.9

5. `colsample_bytree`- fraction of the features that can be used to train each tree. A large value means almost all features can be used to build the decision tree

- typical values: 0.5–0.9

The above are the main hyperparameters people often tune. It is perfectly OK if you don't understand them all completely (like me) but you can refer to this post which gives a thorough overview of how each of the above parameters works and how to tune them.

Hyperparameter Tuning of XGBoost with GridSearchCV

Finally, it is time to super-charge our XGBoost classifier. We will be using the `GridSearchCV` class from Scikit-learn which accepts possible values for desired hyperparameters and fits separate models on the given data for each combination of hyperparameters. I won't go into detail about how GridSearch works but you can check out my separate comprehensive article on the topic:

We will be tuning only a few of the parameters in two rounds because of how tuning is both computationally and time-expensive. Let’s create the parameter grid for the first round:

In the grid, I fixed `subsample` and `colsample_bytree` to recommended values to speed things up and prevent overfitting.

We will import `GridSearchCV` from `sklearn.model_selection`, instantiate and fit it to our preprocessed data:

After an excruciatingly long time, we finally got the best params and best score:

This time, I chose `roc_auc` metric which calculates the area under the ROC (receiver operating characteristic) curve. It is one of the most popular and robust evaluation metrics for unbalanced classification problems. You can learn more about it here. Let's see the best params:

As you can see, only `scale_pos_weight` is in the middle of its provided range. The other parameters are at the end of their ranges meaning that we have to keep exploring:

We will fit a new GridSearch object to the data with the updated param grid and see if we got an improvement on the best score:

Looks like the second round of tuning resulted in a slight decrease in performance. We have got no choice but to stick with the first set of parameters which were:

Let’s create a final classifier with the above parameters:

Finally, make predictions on the test set:

Conclusion

We have made it to the end of this introductory guide on XGBoost for classification problems. Even though we covered a lot, there are still many topics to explore in terms of XGBoost itself and on the topic of classification.