Published in Towards Data Science

Bex T. Follow
Apr 7, 2021 · 8 min read ★ · Listen

Beginner's Guide to XGBoost for Classification Problems

Utilize the hottest ML library for state-of-the-art performance

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

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

Codeless Machine Learning Dictionary For Dummies
Edit description
towardsdatascience.com

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.

```python
import pandas as pd

rain = pd.read_csv("data/weatherAUS.csv")

>>> rain.head()
```

5001.py hosted with ❤ by GitHub view raw

```
>>> rain.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype  
---  ------         --------------   -----  
 0   Date           145460 non-null  object 
 1   Location       145460 non-null  object 
 2   MinTemp        143975 non-null  float64
 3   MaxTemp        144199 non-null  float64
 4   Rainfall       142199 non-null  float64
 5   Evaporation    82670 non-null   float64
 6   Sunshine       75625 non-null   float64
 7   WindGustDir    135134 non-null  object 
 8   WindGustSpeed  135197 non-null  float64
 9   WindDir9am     134894 non-null  object 
 10  WindDir3pm     141232 non-null  object 
 11  WindSpeed9am   143693 non-null  float64
 12  WindSpeed3pm   142398 non-null  float64
 13  Humidity9am    142806 non-null  float64
 14  Humidity3pm    140953 non-null  float64
 15  Pressure9am    130395 non-null  float64
 16  Pressure3pm    130432 non-null  float64
 17  Cloud9am       89572 non-null   float64
 18  Cloud3pm       86102 non-null   float64
 19  Temp9am        143693 non-null  float64
 20  Temp3pm        141851 non-null  float64
 21  RainToday      142199 non-null  object 
 22  RainTomorrow   142193 non-null  object 
dtypes: float64(16), object(7)
memory usage: 25.5+ MB
```

5002.py hosted with ❤ by GitHub view raw

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`.

```
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:

```
missing_props = rain.isna().mean(axis=0)

>>> missing_props

MinTemp          0.010209
MaxTemp          0.008669
Evaporation      0.431665
Sunshine         0.480098
WindGustDir      0.070989
WindGustSpeed    0.070555
WindDir9am       0.072639
WindDir3pm       0.029066
WindSpeed9am     0.012148
WindSpeed3pm     0.021050
Humidity9am      0.018246
Humidity3pm      0.030984
Pressure9am      0.103568
Pressure3pm      0.103314
Cloud9am         0.384216
Cloud3pm         0.408071
Temp9am          0.012148
Temp3pm          0.024811
RainToday        0.022419
dtype: float64
```

5003.py hosted with ❤ by GitHub view raw

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

```
Evaporation     0.431665
Sunshine        0.480098
Cloud3pm        0.408071
dtype: float64
```

5004.py hosted with ❤ by GitHub view raw

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

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

5005.py hosted with ❤ by GitHub view raw

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

```
X = rain.drop("RainToday", axis=1)
y = rain.RainToday
```

5006.py hosted with ❤ by GitHub view raw

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

> *The next code examples will heavily use Sklearn-Pipelines. If you are not familiar with them, check out my separate article for the complete guide on them.*

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

```
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
```

```
        ]
    )
```

5007.py hosted with ❤ by GitHub view raw

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:

```
from sklearn.preprocessing import StandardScaler

numeric_pipeline = Pipeline(
    steps=[("impute", SimpleImputer(strategy="mean")),
           ("scale", StandardScaler())]
)
```

5008.py hosted with ❤ by GitHub view raw

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:

```
cat_cols = X.select_dtypes(exclude="number").columns
num_cols = X.select_dtypes(include="number").columns
```

5009.py hosted with ❤ by GitHub view raw

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

```
from sklearn.compose import ColumnTransformer

full_processor = ColumnTransformer(
    transformers=[
        ("numeric", numeric_pipeline, num_cols),
        ("categorical", categorical_pipeline, cat_cols),
    ]
```

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:

```python
import xgboost as xgb

xgb_cl = xgb.XGBClassifier()

>>> print(type(xgb_cl))

<class 'xgboost.sklearn.XGBClassifier'>
```

5011.py hosted with ❤ by GitHub view raw

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:

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X_processed, y_processed, stratify=y_processed, random_state=1121218
)
```

5012.py hosted with ❤ by GitHub view raw

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:

```python
from sklearn.metrics import accuracy_score

# Init classifier
xgb_cl = xgb.XGBClassifier()

# Fit
xgb_cl.fit(X_train, y_train)

# Predict
preds = xgb_cl.predict(X_test)

# Score
>>> accuracy_score(y_test, preds)
0.8507080984463082
```

5013.py hosted with ❤ by GitHub view raw

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.

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:

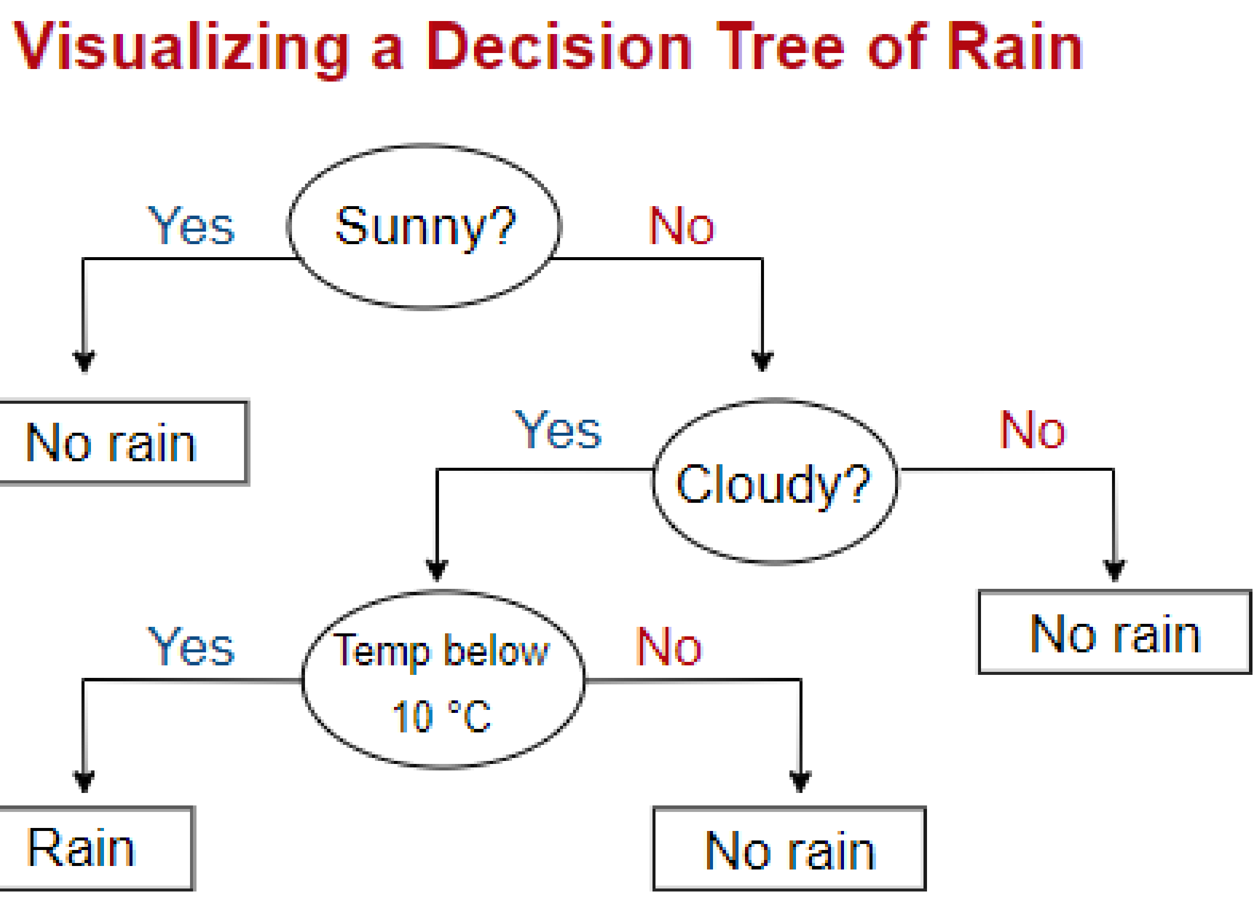


Image by the author. Decision tree to predict rain

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).

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:

```
>>> xgb_cl

XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=4, num_parallel_tree=1, random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

5014.py hosted with ❤ by GitHub view raw

> *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.*

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.

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:

Automatic Hyperparameter Tuning with Sklearn GridSearchCV and RandomizedSearchCV

Edit description

towardsdatascience.com

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:

```python
param_grid = {
    "max_depth": [3, 4, 5, 7],
    "learning_rate": [0.1, 0.01, 0.05],
    "gamma": [0, 0.25, 1],
    "reg_lambda": [0, 1, 10],
    "scale_pos_weight": [1, 3, 5],
    "subsample": [0.8],
    "colsample_bytree": [0.5],
}
```

5015.py hosted with ❤ by GitHub view raw

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:

```python
from sklearn.model_selection import GridSearchCV
```

```
grid_cv = GridSearchCV(xgb_cl, param_grid, n_jobs=-1, cv=3, scoring="roc_auc")

# Fit
_ = grid_cv.fit(X_processed, y_processed)
```

5016.py hosted with ❤ by GitHub view raw

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

```
>>> grid_cv.best_score_
0.853810061069393
```

5017.py hosted with ❤ by GitHub view raw

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:

```
>>> grid_cv.best_params_

{'gamma': 1,
 'learning_rate': 0.1,
 'max_depth': 7,
 'reg_lambda': 10,
 'scale_pos_weight': 3}
```

5018.py hosted with ❤ by GitHub view raw

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:

```
# Insert the new fixed values to the grid
param_grid["scale_pos_weight"] = [3]
param_grid["subsample"] = [0.8]
param_grid["colsample_bytree"] = [0.5]
```

```
param_grid["learning_rate"] = [0.3, 0.5, 0.7, 1]
```

5019.py hosted with ❤ by GitHub view raw

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:

```
grid_cv_2 = GridSearchCV(xgb_cl, param_grid,
                         cv=3, scoring="roc_auc", n_jobs=-1)

_ = grid_cv_2.fit(X_processed, y_processed)

>>> grid_cv_2.best_score_
0.8501002182976786
```

5020.py hosted with ❤ by GitHub view raw

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:

```
>>> grid_cv.best_params_

{'gamma': 1,
 'learning_rate': 0.1,
 'max_depth': 7,
 'reg_lambda': 10,
 'scale_pos_weight': 3}
```

5021.py hosted with ❤ by GitHub view raw

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

```
final_cl = xgb.XGBClassifier(
    **grid_cv.best_params_,
    objective="binary:logistic",
    colsample_bytree=0.5,
    subsample=0.8
```

Finally, make predictions on the test set:

```python
from sklearn.metrics import roc_auc_score

_ = final_cl.fit(X_train, y_train)

preds = final_cl.predict(X_test)
```

5023.py hosted with ❤ by GitHub view raw

Conclusion

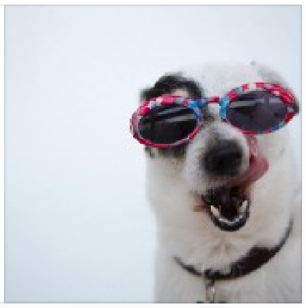
We have made it to the end of this introductory guide on XGBoost for classification problems. Even though we covered a lot, there are still many topics to explore in terms of XGBoost itself and on the topic of classification.

I strongly recommend you to check out the links I provided as additional sources to learn XGBoost and suggest reading more on how to tackle classification problems.

How To Touch Into the Heart of Matplotlib And Create Amazing Plots

Use Matplotlib like an absolute boss!

towardsdatascience.com



159 2