

# XGBoost Basics

Learn about the basics of using XGBoost.

## Chapter Goals:

- Learn about the XGBoost data matrix
- Train a **Booster** object in XGBoost

### A. Basic data structures

The basic data structure for XGBoost is the **DMatrix**, which represents a data matrix. The **DMatrix** can be constructed from NumPy arrays.

The code below creates **DMatrix** objects with and without labels.

```
data = np.array([
    [1.2, 3.3, 1.4],
    [5.1, 2.2, 6.6]])

import xgboost as xgb
dmat1 = xgb.DMatrix(data)

labels = np.array([0, 1])
dmat2 = xgb.DMatrix(data, label=labels)
```



The **DMatrix** object can be used for training and using a **Booster** object, which represents the gradient boosted decision tree. The **train** function in XGBoost lets us train a **Booster** with a specified set of parameters.

The code below trains a **Booster** object using a predefined labeled dataset.

```
# predefined data and labels
print('Data shape: {}'.format(data.shape))
print('Labels shape: {}'.format(labels.shape))
dtrain = xgb.DMatrix(data, label=labels)

# training parameters
```

```
params = {
    'max_depth': 0,
    'objective': 'binary:logistic'
}
print('Start training')
bst = xgb.train(params, dtrain) # booster
print('Finish training')
```



A list of the possible parameters and their values can be found [here](#). In the example above, we set the `'max_depth'` parameter to `0` (which means no limit on the tree depths, equivalent to `None` in scikit-learn). We also set the `'objective'` parameter (the objective function) to binary classification via logistic regression. For the remaining available parameters, we used their default settings (so we didn't include them in `params`).

## B. Using a **Booster**

After training a **Booster**, we can evaluate it and use it to make predictions.

```
# predefined evaluation data and labels
print('Data shape: {}'.format(eval_data.shape))
print('Labels shape: {}'.format(eval_labels.shape))
deval = xgb.DMatrix(eval_data, label=eval_labels)

# Trained bst from previous code
print(bst.eval(deval)) # evaluation

# new_data contains 2 new data observations
dpred = xgb.DMatrix(new_data)
# predictions represents probabilities
predictions = bst.predict(dpred)
print('{}\n'.format(predictions))
```



The evaluation metric used for binary classification (`eval-error`) represents the classification error, which is the default `'eval_metric'` parameter for binary classification **Booster** models.

Note that the model's predictions (from the `predict` function) are probabilities, rather than class labels. The actual label classifications are just the rounded probabilities. In the example above, the **Booster** predicts classes of 0 and 1, respectively.

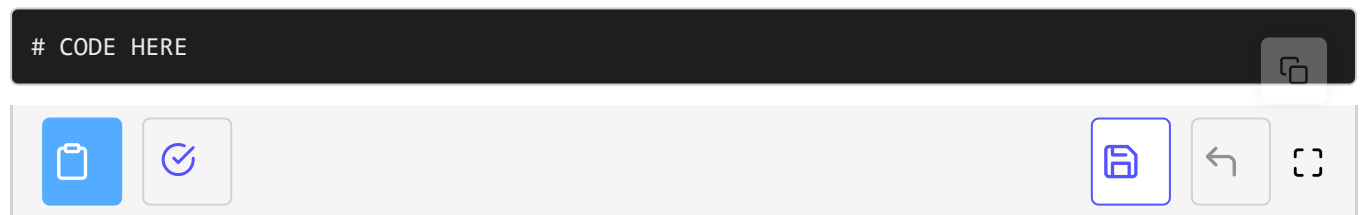
## Time to Code!

The coding exercise for this chapter will be to train a `Booster` object on input `data` and `labels` (predefined in the backend).

The first thing to do is set up a `DMatrix` for training.

Set `dtrain` equal to `xgb.DMatrix` initialized with `data` as the required argument and `labels` as the `label` keyword argument.

```
# CODE HERE
```

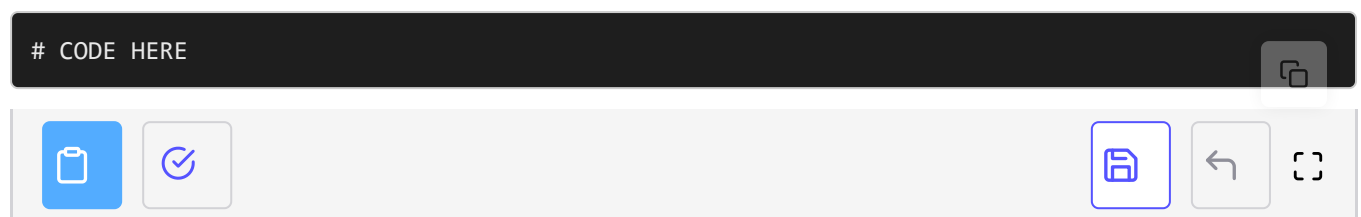


The input dataset contains 3 classes, so we'll perform multiclass classification with the `Booster`. The dataset is also relatively small, so we limit the decision tree's maximum depth to 2.

This means that the parameters for the `Booster` object will have `'max_depth'` set to 2, `'objective'` set to `'multi:softmax'`, and `'num_class'` set to 3.

Set `params` equal to a dictionary with the specified keys and values.

```
# CODE HERE
```



Using the data matrix and parameters, we'll train the `Booster`.

Set `bst` equal to `xgb.train` applied with `params` and `dtrain` as the required arguments.

```
# CODE HERE
```

