Machine Learning

| Gradient Boosting - XGBoost

Prof. Dr. Mostafa Elhosseini

https://youtube.com/ElhosseiniAcademy

Agenda

- Gradient Boosting
- XGBoost

Gradient Boosting

- Gradient boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor.

- However, **instead of tweaking the instance weights at every iteration** like AdaBoost does, this method tries to **fit the new predictor to the residual errors** made by the previous predictor

Gradient Boosting

- Simple regression example, using decision trees as the base predictors; this is called gradient tree boosting, or gradient boosted regression trees (GBRT).

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100)  # y = 3x² + Gaussian noise

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

Gradient Boosting

- Train a second DecisionTreeRegressor on the residual errors made by the first predictor.

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
tree_reg2.fit(X, y2)
```

- Train a third regressor on the residual errors made by the second predictor

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
```

Gradient Boosting

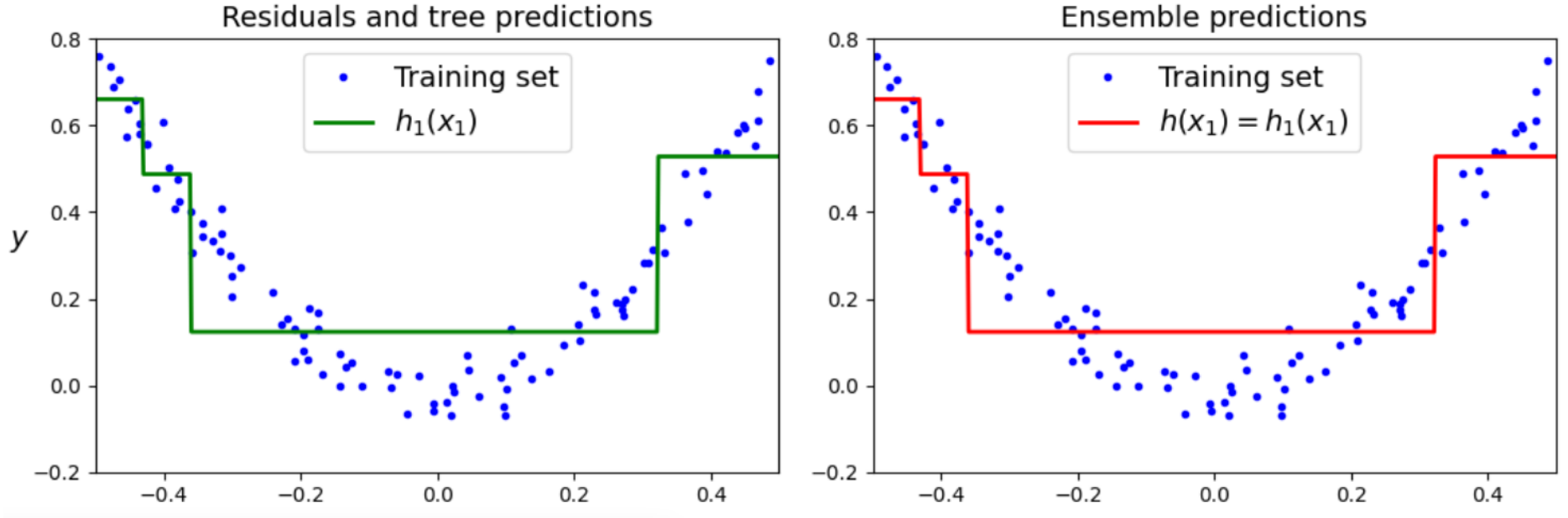
- Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees

```
X_new = np.array([[-0.4], [0.], [0.5]])
sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

```
array([0.49484029, 0.04021166, 0.75026781])
```

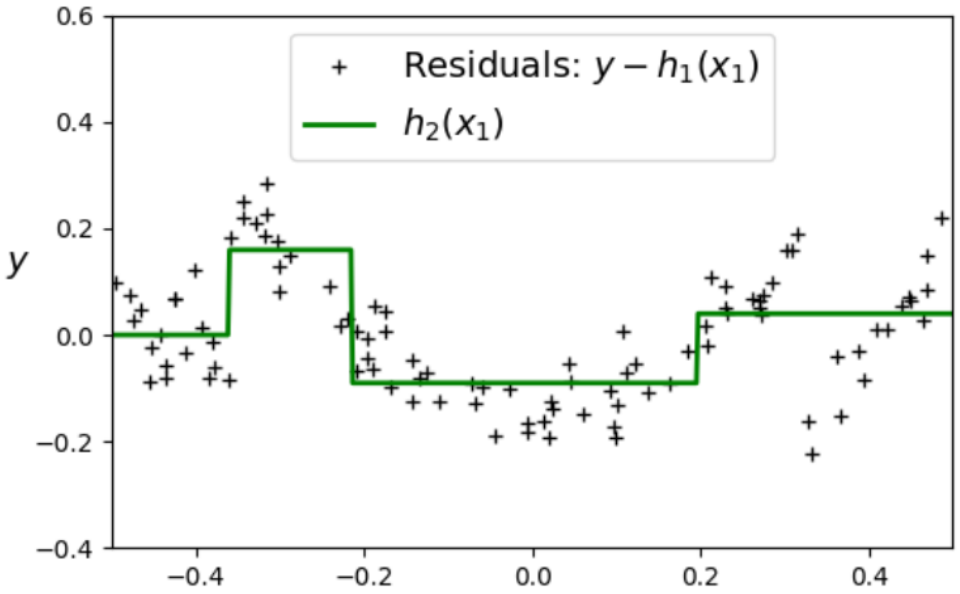
Gradient Boosting

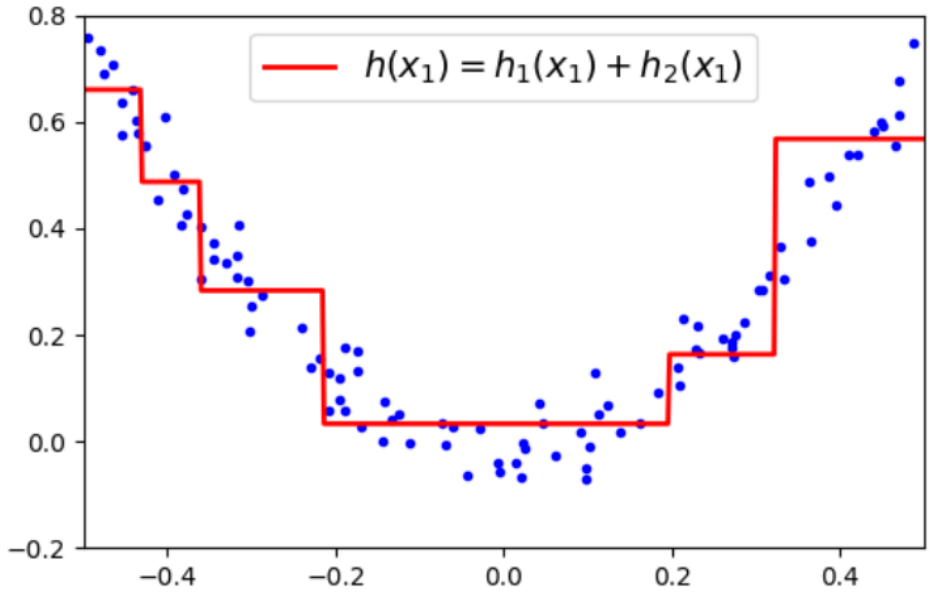
- Figure below represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column
- In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions



Gradient Boosting

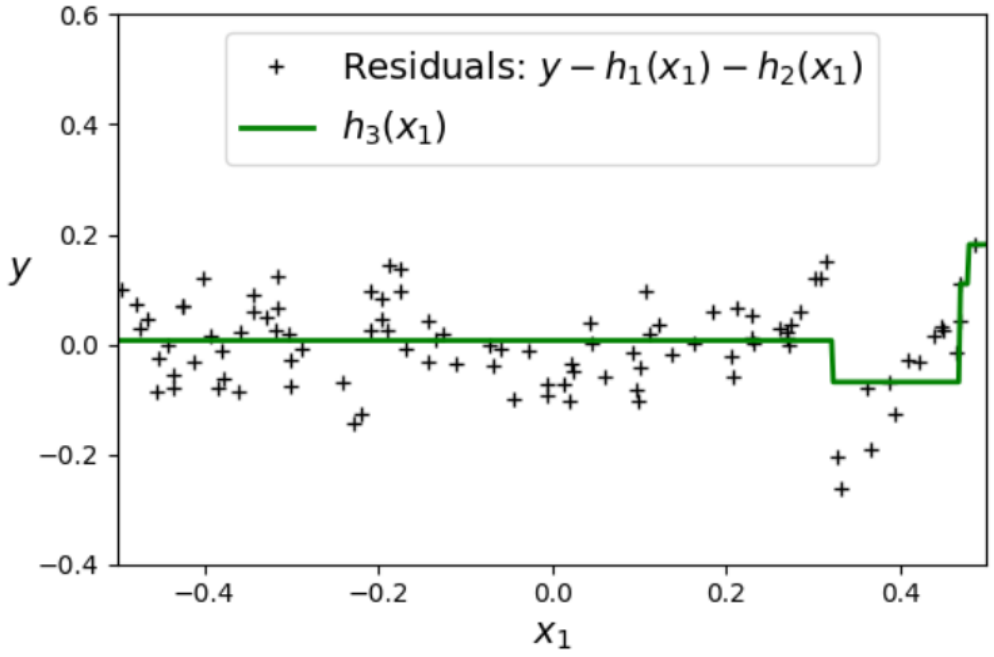
- A new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees

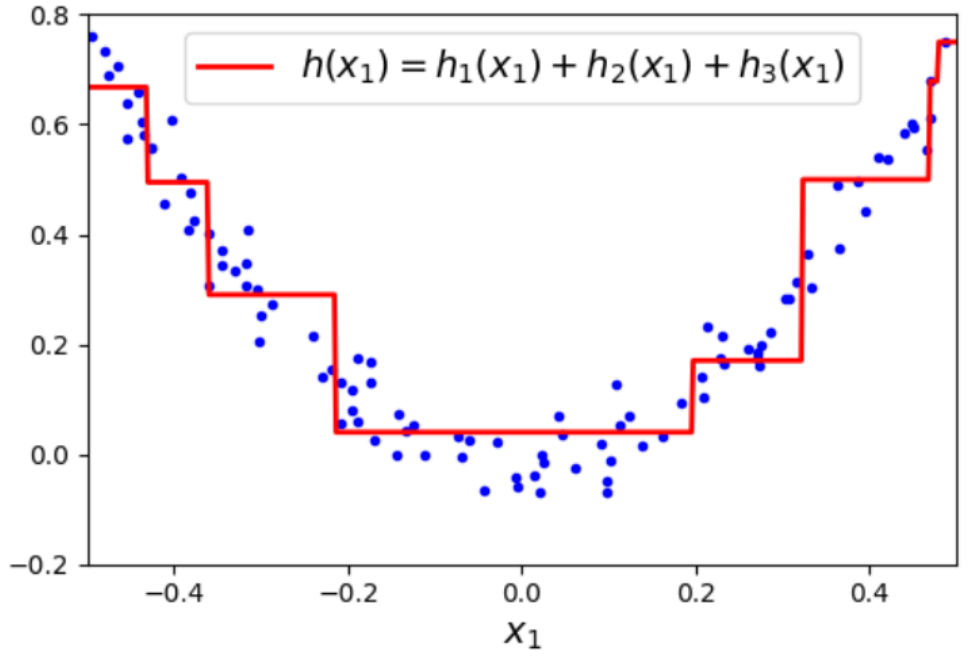




Gradient Boosting

- Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.





Gradient Boosting

- You can use Scikit-Learn's GradientBoostingRegressor class to train GBRT ensembles more easily (there's also a GradientBoostingClassifier class for classification). Much like the RandomForestRegressor class, it has hyperparameters to control the growth of decision trees (e.g., max_depth, min_samples_leaf), as well as hyperparameters to control the ensemble training, such as the number of trees (n_estimators).

Gradient Boosting

- The following code creates the same ensemble as the previous one

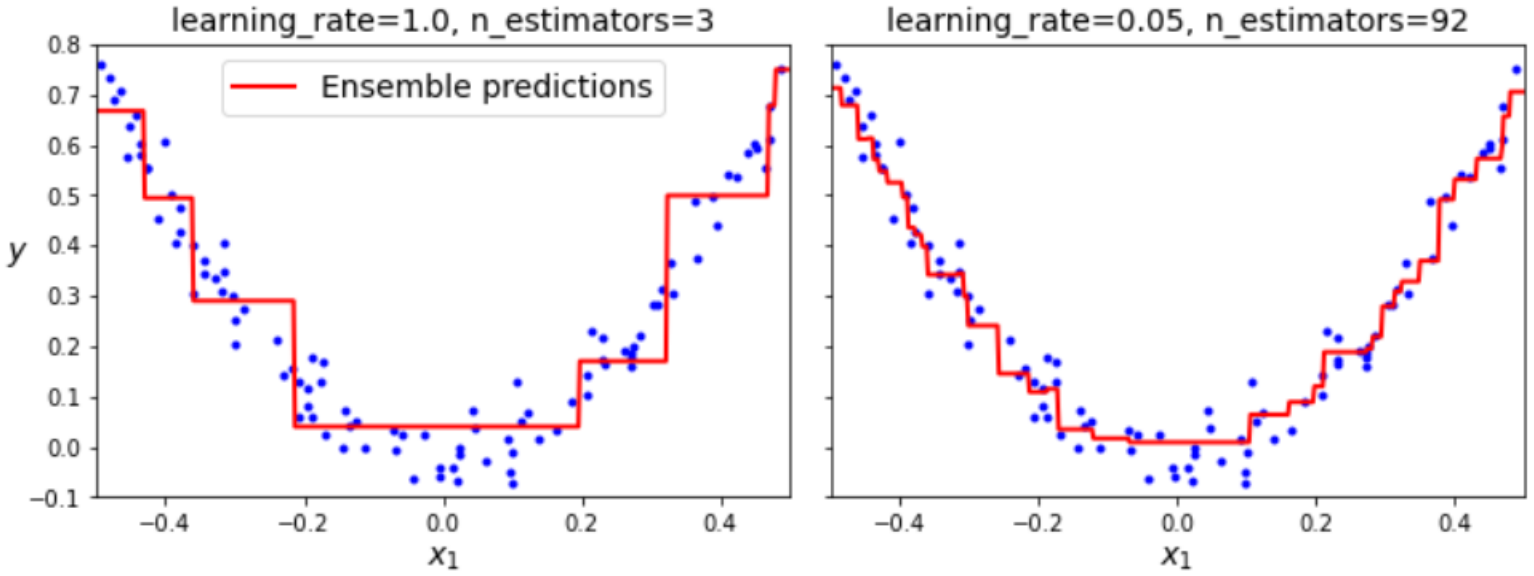
```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                 learning_rate=1.0, random_state=42)
gbrt.fit(X, y)
```

- The learning_rate hyperparameter scales the contribution of each tree. If you set it to a low value, such as 0.05, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called **shrinkage**

Gradient Boosting

- Figure below shows two GBRT ensembles trained with different hyperparameters: the one on the left does not have enough trees to fit the training set, while the one on the right has about the right amount.



Gradient Boosting

- If we added more trees, the GBRT would start to overfit the training set.
- To find the optimal number of trees, you could perform cross-validation using GridSearchCV or RandomizedSearchCV, as usual, but there's a simpler way if you set the n_iter_no_change hyperparameter to an integer value, say 10, then the GradientBoostingRegressor will automatically stop adding more trees during training if it sees that the last 10 trees didn't help.

Gradient Boosting

- This is simply early stopping, but with a little bit of patience: it tolerates having no progress for a few iterations before it stops.

```
gbrt_best = GradientBoostingRegressor(
    max_depth=2, learning_rate=0.05, n_estimators=500,
    n_iter_no_change=10, random_state=42)
gbrt_best.fit(X, y)
```

```
gbrt_best.n_estimators_
```

```
92
```

Gradient Boosting

- If you set **n_iter_no_change** too low, training may stop too early and the model will underfit. But if you set it too high, it will overfit instead.
- We also set a fairly small learning rate and a high number of estimators, but the actual number of estimators in the trained ensemble is much lower, thanks to early stopping

Gradient Boosting

- When n_iter_no_change is set, the fit() method automatically splits the training set into a smaller training set and a validation set: this allows it to evaluate the model's performance each time it adds a new tree. The size of the validation set is controlled by the validation_fraction hyperparameter, which is 10% by default.

- The tol hyperparameter determines the maximum performance improvement that still counts as negligible. It defaults to 0.0001.

Gradient Boosting

- The GradientBoostingRegressor class also supports a subsample hyperparameter, which specifies the fraction of training instances to be used for training each tree. For example, if subsample=0.25, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this technique trades a higher bias for a lower variance. It also speeds up training considerably. This is called **stochastic gradient boosting**.

Histogram-Based Gradient Boosting

- Scikit-Learn also provides another GBRT implementation, optimized for large datasets: histogram-based gradient boosting (HGB).
- It works by binning the input features, replacing them with integers. The number of bins is controlled by the max_bins hyperparameter, which defaults to 255 and cannot be set any higher than this.
- Binning can greatly reduce the number of possible thresholds that the training algorithm needs to evaluate. Moreover, working with integers makes it possible to use faster and more memory efficient data structures. And the way the bins are built removes the need for sorting the features when training each tree

Histogram-Based Gradient Boosting

- As a result, this implementation has a computational complexity of $O(b \times m)$ instead of $O(n \times m \times \log(m))$, where b is the number of bins, m is the number of training instances, and n is the number of features. In practice, this means that HGB can train hundreds of times faster than regular GBRT on large datasets.
- However, binning causes a precision loss, which acts as a regularizer: depending on the dataset, this may help reduce overfitting, or it may cause underfitting.

Histogram-Based Gradient Boosting

- Scikit-Learn provides two classes for HGB: **HistGradientBoostingRegressor** and **HistGradientBoostingClassifier**.
- They're similar to GradientBoostingRegressor and GradientBoostingClassifier, with a few notable differences:
  - Early stopping is automatically activated if the number of instances is greater than 10,000. You can turn early stopping always on or always off by setting the early_stopping hyperparameter to True or False.
  - Subsampling is not supported.
  - n_estimators is renamed to max_iter.
  - The only decision tree hyperparameters that can be tweaked are max_leaf_nodes, min_samples_leaf, and max_depth.

Histogram-Based Gradient Boosting

- The HGB classes also have two nice features: they support both categorical features and missing values. This simplifies preprocessing quite a bit. However, the categorical features must be represented as integers ranging from 0 to a number lower than max_bins. You can use an OrdinalEncoder for this

Histogram-Based Gradient Boosting

- complete pipeline for the California housing dataset introduced in Chapter 2

```
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder

hgb_reg = make_pipeline(
    make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
                            remainder="passthrough"),
    HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
)
hgb_reg.fit(housing, housing_labels)
```

Histogram-Based Gradient Boosting

- The whole pipeline is just as short as the imports! No need for an imputer, scaler, or a one-hot encoder, so it’s really convenient. Note that categorical_features must be set to the categorical column indices (or a Boolean array). Without any hyperparameter tuning, this model yields an RMSE of about 47,600, which is not too bad.

Histogram-Based Gradient Boosting

- Several other optimized implementations of gradient boosting are available in the Python ML ecosystem: in particular, XGBoost, CatBoost, and LightGBM. These libraries have been around for several years. They are all specialized for gradient boosting, their APIs are very similar to Scikit-Learn's, and they provide many additional features, including GPU acceleration; you should definitely check them out! Moreover, the TensorFlow Random Forests library provides optimized implementations of a variety of random forest algorithms, including plain random forests, extra-trees, GBRT, and several more.