RSBoost

a Really Simple XGBoost implementation

Presented by: **Ilaria Ritelli** and **Salvatore Salerno**

Academic Year 2024/2025

Outline

1. Understanding XGBoost
2. Greedy tree growing strategy
3. Developing RSBoost
4. Constructing efficiently a gradient boosted tree
5. Testing our implementation
6. Conclusions

Understanding XGBoost

XGBoost is a tree boosting system that uses a **regularized learning objective**.

- Given a dataset $D = \{x_i, y_i\}$, tree boosting calculates the prediction $\hat{y}_i$ of input sample x_i as

$$\hat{y}_i = \phi(x_i) = \sum_k \eta \cdot f_k(x_i)$$

  f_k is a function corresponding to a tree, η is the shrinkage factor ($\approx$ learning rate)

- The tree at step t is trained by minimizing the objective function:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

Loss function

Regularization term

Understanding XGBoost

- $\Omega(f_t) = \gamma T_t + \frac{1}{2}\lambda \|w_t\|^2$ is the complexity of the t-th tree

- The objective function is approximated as

Iterating over all leaves of the tree

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) = \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$

where g_i, h_i are the **gradient** and the **hessian** of the loss for sample x_i

- For a fixed tree structure q, the optimal value is found in

$$\mathcal{L}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_i g_i)^2}{\sum_i h_i + \lambda} + \gamma T$$

Understanding XGBoost

- In practice, tree is iteratively calculated
- Training starts at a single root node with all instances
- At each iteration step, the optimal split of node I is computed as the leaves I_L, I_R s.t.

$$\mathscr{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)}{\sum_{i\in I} h_i + \lambda}\right] - \gamma$$

(loss reduction, a.k.a. **similarity score** gain) is maximized.

Outline

1. Understanding XGBoost
2. Greedy tree growing strategy
3. Developing RSBoost
4. Constructing efficiently a gradient boosted tree
5. Testing our implementation
6. Conclusions

Greedy Tree growing strategy

- XGBoost offers two variants: ***exact*** and ***approximate***
- ***Exact*** iterates over all distinct values of all input features to find the one best split: accurate but takes longer to be calculated
- ***Approximate*** pre-selects a number of candidate splits from quantiles of input data: may be less accurate, but faster
- Quantiles are weighted by the hessians of the input points via a ranking function $r_k()$
- Goal: for feature k, find candidates $\{s_{k,1}, s_{k,2}, \ldots, s_{k,n}\}$ such that for some ϵ

$$| r_k(s_{k,j}) - r_k(s_{k,j+1}) | < \epsilon$$

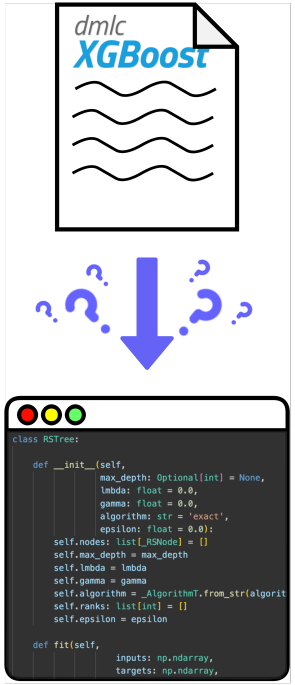
Outline

1. Understanding XGBoost
2. Greedy tree growing strategy
3. Developing RSBoost
4. Constructing efficiently a gradient boosted tree
5. Testing our implementation
6. Conclusions

From theory to implementation: RSBoost

The main goal of the project was to **understand** how to translate the paper's concepts into a real implementation.

- RSBoost has been developed in Python using the Numpy library.
- We focused our attention on how to efficiently implement the data structure and algorithm proposed while leveraging the capabilities of Python.
- To verify the correctness of out implementation we compared its results with those of the XGBoost library.



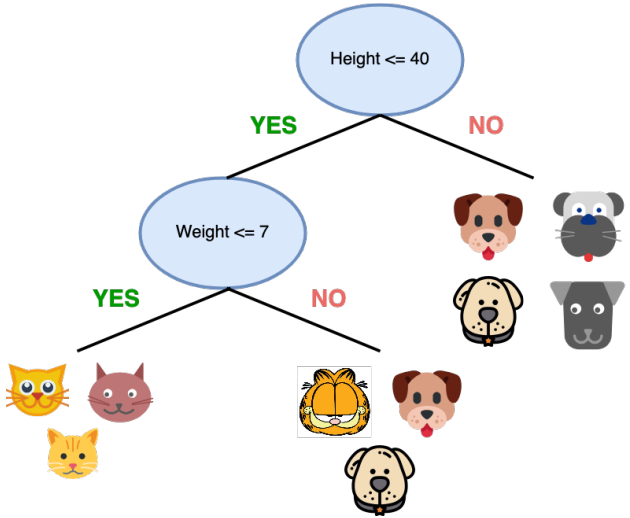
Outline

1. Understanding XGBoost
2. Greedy tree growing strategy
3. Developing RSBoost
4. Constructing efficiently a gradient boost tree
5. Testing our implementation
6. Conclusions

Constructing efficiently a gradient boosted tree

The main data structure of gradient boosting is the gradient boosted tree, as it represent the models that make up the ensemble.

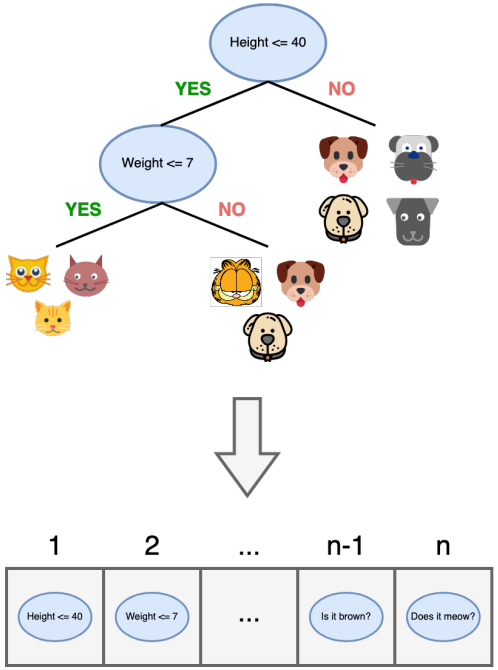
- A gradient boosted tree is a **binary** tree where internal nodes encode **decision rules** that guide traversal, while leaf nodes store the final **predictions**.
- A straightforward approach to representing the tree is as a set of nodes connected via pointers.
- Although simple, this is suboptimal in both space and time complexity, especially in real applications that involve large datasets.



Optimisation: array-based tree representation

A tree can be represented as an array whose entries contain nodes. The structure of the tree is preserved by moving using the **indices** of the array.

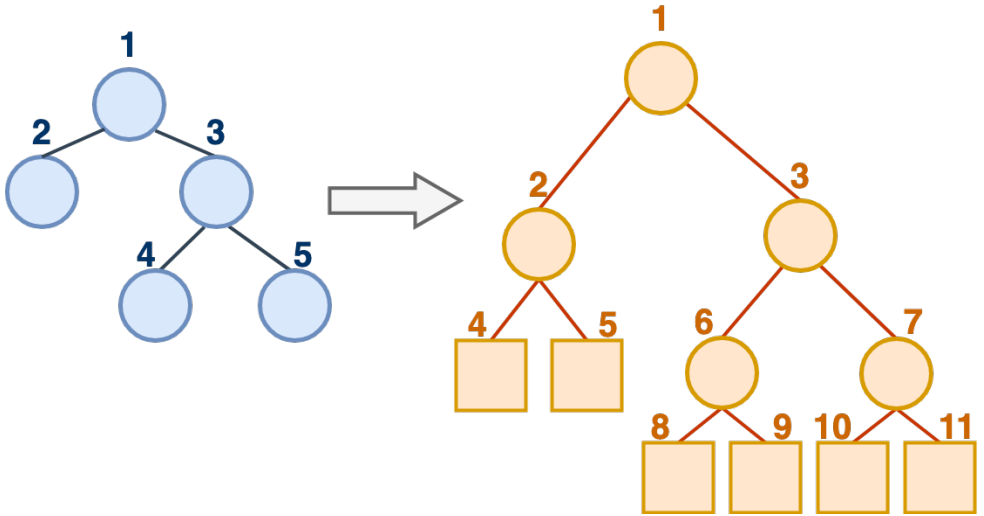
- Explicit pointers node are not needed anymore, reducing space overhead.
- However, its still necessary to store in each node its children's indices.
- There should be a better way to percolate the tree without needing explicit child information in the nodes.



Optimisation: succinct representation of binary trees

Given a **binary** tree T it can be represented as an array of nodes, in which traversal does not need to keep child information explicitly, via three phases:

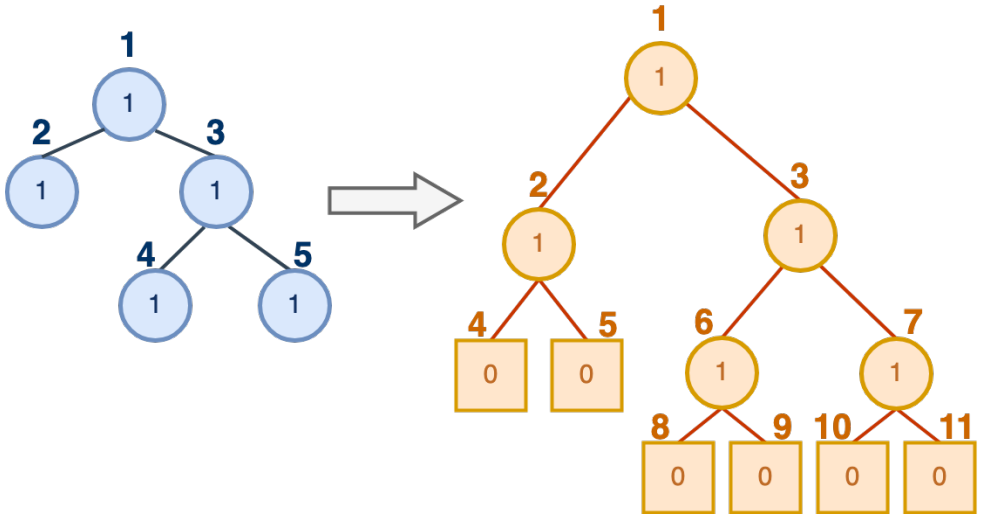
- **Expand:** for each node missing one or both children, insert special **dummy** leaves, so that each **real** node has always two children.
- This generates the expanded tree T' composed of $2 \times n + 1$ nodes.



Optimisation: succinct representation of binary trees

Given a **binary** tree T it can be represented as an array of nodes, in which traversal does not need to keep child information explicitly, via three phases:

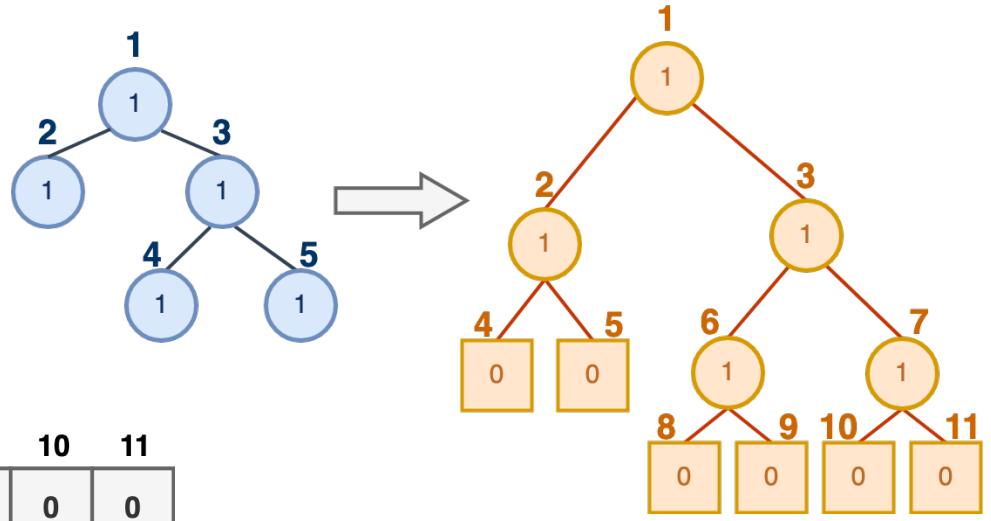
- **Label:** assign a binary label to each node, distinguishing nodes coming from the original tree to dummy ones inserted in the expanded tree.



Optimisation: succinct representation of binary trees

Given a **binary** tree T it can be represented as an array of nodes, without needing to store child information explicitly, via three phases:

- **Serialise:** perform a breadth-first traversal of the expanded tree.
- While visiting the tree level by level construct a binary array B that stores the binary labels of the nodes.



B=

1	2	3	4	5	6	7	8	9	10	11
1	1	1	0	0	1	0	0	0	0	0

Moving in a binary tree using its succinct representation

Each **real** node has two indices, one in the original tree i and one in the expanded tree i'.
It is possible to move from one to the other through the following operations:

$$i = B\,.\,RANK_1(i') \text{ and } i' = B\,.\,SELECT_1(i)$$

By exploiting this relationship we can move in T using only its index i and B:

- The left child is computed as: `left_child(`i`)` $= B\,.\,RANK_1(2 \times i)$
- The right child is computed as: `right_child(`i`)` $= B\,.\,RANK_1(2 \times i + 1)$
- The parent is computed as: `parent(`i`)` $= \lfloor B\,.\,SELECT_1(i) \div 2 \rfloor$

Implementing the succinct representation

The labelling is handled implicitly through types: a node is represented by one of the classes `_Internal( )`, `_Leaf( )` or `_Dummy( )`, all of which inherit from the base class `_RSNode( )`.

At the start an array of nodes is initialised. The expansion phase is handled during the construction of the gradient boosted tree. At each iteration the tree's depth and the split computation decides whether or not the current node should be a leaf or not:

- **If it is an internal node:** the algorithm must recurse to construct its two children, meaning it needs no expansion.

- **If is a leaf:** the node misses both children, so it is immediately expanded by adding its dummy leaves.

Implementing the succinct representation

The model only needs to handle traversal from root to leaf, meaning it requires only $RANK_1$ operations over the label array B, where:

$$B\,.\,RANK_1(i) = \sum_{j=1}^{i} B[j]$$

Instead of constructing B explicitly, needing us to scan all elements in 1 to i each time, a precomputed prefix array **rank** is constructed where:

$$\forall i \in [1,n]\,.\,rank[i] = B\,.\,RANK_1(i)$$

Here we assumed indices are in [1, n], in the real implementation indices are in [0, n-1]

Implementing the succinct representation

The constructed node array currently refers to the expanded tree. To recover the original tree structure and construct the $rank$ array, we perform a breadth-first traversal on the expanded tree, updating a counter variable containing the number of real nodes encountered so far. At each iteration i:

- **If the visited node is a real one:** we keep it in the node array and increment the counter by one.
- **if the visited node is a dummy one:** we remove it from the node array and do not increment the counter.

After this checks the counter value contains $rank[i]$ for the corresponding node in the original tree.

Building the ensemble

- Module **`rsboosting.py`** implements the ensemble
- Offers interfaces **`RSBoostClassifier`** and **`RSBoostRegressor`**
- Method **`fit()`** trains the trees iteratively

```
def fit(X, y):
   pred = np.array([self.starting_value] * X.shape[0])

   for _ in range(self.n_estimators):
      residuals = y - pred

      tree = Tree(*params)
      tree.fit(X, residuals, self.gradient(y, pred), self.hessian(y, pred))

      pred += self.eta * tree.predict(X)
      self.ensemble.append(tree)
```

Building the ensemble

- Method `predict()` returns the output of the entire model by adding up single tree outputs

```
def predict(X):
  # Output is starting value + sum of predictions of each tree
   pred = np.array([self.starting_value] * X.shape[0])

   for tree in self.ensemble:
      pred += self.eta * tree.predict(X)
   return pred
```

Outline

1. Understanding XGBoost
2. Greedy tree growing strategy
3. Developing RSBoost
4. Constructing efficiently a gradient boosted tree
5. Testing our implementation
6. Conclusions

Testing our implementation : Criteo

- Criteo click log dataset; binary classification task
- Contains several categorical features, which were one-hot encoded
- 6K samples, 30K features
- 80%/20% train/test split

Implementation	Algo. type	TR accuracy	TS accuracy
XGBoost	exact	0.9370	0.78
	approx.	0.9370	0.78
RSBoost	exact	0.8837	0.75
	approx.	0.8425	0.76

Testing our implementation : Higgs

- Higgs boson dataset, binary classification task
- 10K samples, 29 features
- 80%/20% train/test split

Implementation	Algo. type	TR accuracy	TS accuracy
XGBoost	exact	0.8893	0.70
	approx.	0.8893	0.70
RSBoost	exact	0.9276	0.67
	approx.	0.9380	0.67

Outline

1. Understanding XGBoost
2. Greedy tree growing strategy
3. Developing RSBoost
4. Main data structures of RSBoost
5. Constructing efficiently a gradient boosted tree
6. Conclusions

Thank you for your attention!

Conclusions

- We partially reimplemented the XGBoost system, focusing on the split finding algorithms
- We obtained performance measures comparable to that of the original library
- We got to see how theoretical concepts actually work in practice, producing real results

References

- Chen, Tianqi, and Carlos Guestrin. "*Xgboost: A scalable tree boosting system.*" Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016.

- P. Miklos, "Succinct representation of binary trees," 2008 6th International Symposium on Intelligent Systems and Informatics, Subotica, Serbia, 2008.

- https://archive.ics.uci.edu/dataset/280/higgs - Higgs Dataset

- https://www.kaggle.com/c/criteo-display-ad-challenge - Criteo Dataset