Data-intensive space engineering

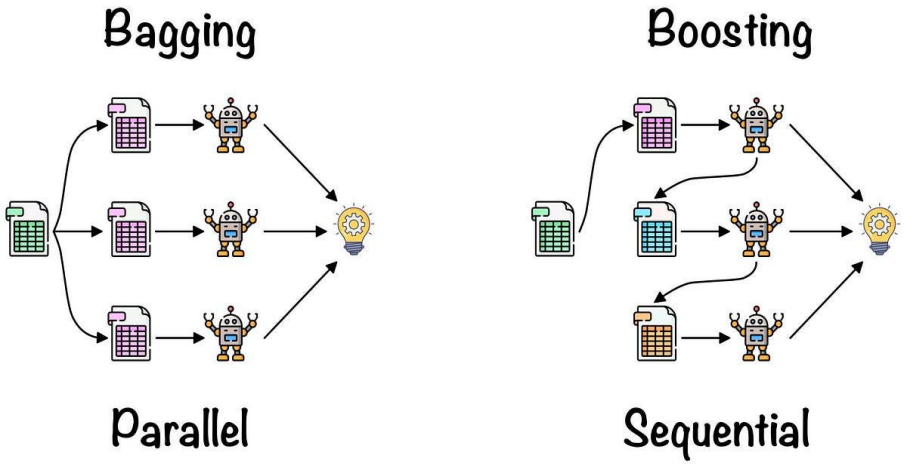
Lecture 5

Carlos Sanmiguel Vila

Introduction to Boosting

- **What is Boosting?**
  - Boosting is an ensemble technique where models are trained sequentially, and each model corrects the mistakes of the previous one.
  - The goal of Boosting is to focus on the hardest-to-classify instances, making the model progressively better.

- **How Boosting Differs from Bagging:**
  - Bagging (e.g., Random Forests) builds each model independently in parallel, using different subsets of the data.
  - Boosting builds models sequentially, where each subsequent model focuses on correcting the errors of the previous model.



AdaBoost

•**AdaBoost** (Adaptive Boosting):

- **How it works**: AdaBoost assigns more weight to misclassified instances, ensuring that subsequent models focus on getting those examples right.
- **Key Concept**: Each model in the sequence is trained with adjusted weights based on the errors of the previous model.

AdaBoost Algorithm:

1. Initialize the weights of all samples equally.
2. Train a weak learner (like a shallow Decision Tree).
3. Increase the weights of incorrectly classified samples.
4. Train another weak learner using the updated weights.
5. Repeat until the desired number of models is reached.

AdaBoost (Adaptive Boosting)

- In AdaBoost, the weak learners are usually simple models with low predictive power, such as shallow decision trees or stumps (trees with a single split). Each weak learner is trained sequentially, and at each iteration, the weights of misclassified instances are increased, forcing the model to focus on the difficult-to-classify examples.

- AdaBoost assigns a weight to each weak learner based on their performance, and the final prediction is made by combining the weighted predictions of all weak learners. Instances consistently misclassified by the ensemble receive higher weights, allowing subsequent weak learners to give more emphasis to these challenging cases.

- One of the significant advantages of boosting is its ability to handle complex relationships in the data and significantly improve weak learners' performance. Boosting often outperforms bagging when it comes to reducing both bias and variance. However, boosting is more sensitive to noisy data and outliers than bagging.

Gradient Boosting

What is Gradient Boosting?

- Gradient Boosting is a powerful Boosting method where models are built sequentially, each new model correcting the residual errors made by the previous model.
- Unlike AdaBoost, which adjusts weights, Gradient Boosting optimizes a loss function by adding models that reduce the residuals (the difference between predicted and actual values).

Key Concepts in Gradient Boosting:

1. **Residuals:** The difference between the true value and the predicted value.
2. **Additive Modeling:** New models are added to minimize the residual errors of the previous model.
3. **Gradient Descent:** The models are trained to minimize a specific loss function, and the "gradient" refers to the direction and step size taken to minimize the error.

AdaBoost vs. Gradient Boosting

Aspect	AdaBoost	Gradient Boosting
Boosting Mechanism & Model Focus	Focuses on **adjusting weights** of misclassified instances at each iteration. Each subsequent model pays more attention to the **samples that were misclassified** by the previous model.	Focuses on **minimizing the residual errors** of previous predictions by fitting subsequent models to the residuals. Each subsequent model attempts to **correct the residual errors** of the previous model using gradient descent.
Loss Function	Implicit loss function (typically based on misclassification error).	Explicit loss function (commonly squared error for regression or log loss for classification).
Weak Learners	Typically **shallow decision trees (stumps)** (depth = 1).	Typically **shallow decision trees**, but can use deeper trees.
Error Handling	Errors in classification lead to an increase in the **weights** of misclassified samples.	Errors are corrected by fitting subsequent models to the **residuals** of the previous model.
Performance in Noisy Data	Can be **sensitive to noisy data** since the algorithm focuses on misclassified samples, potentially overfitting noisy instances.	More **robust** to noisy data due to its gradient descent-based approach, but can still overfit if not regularized properly.

AdaBoost vs. Gradient Boosting

Aspect	AdaBoost	Gradient Boosting
Hyperparameters	Fewer hyperparameters: **n_estimators, learning_rate**.	More hyperparameters: **n_estimators, learning_rate, max_depth, min_samples_split, subsample, etc.**
Speed	Generally **faster** since it deals with fewer parameters and focuses on sample weights.	Generally **slower** due to the iterative fitting of residuals and more complex hyperparameters to tune.
Regularization	Basic regularization via the number of iterations and learning rate.	More built-in regularization options, including tree depth, learning rate, and subsampling, which can help prevent overfitting.
Parallelization	Harder to parallelize due to the sequential nature of the algorithm (as it relies on adjusting sample weights).	Can be **parallelized** at the level of each tree or within each boosting iteration (e.g., **XGBoost**).
Common Use Cases	Often used when fast and simple boosting is required, such as in cases with **low noise** and **simple weak learners**.	Preferred for more complex scenarios requiring **robust error correction** and better performance with **hyperparameter tuning**.
Robustness to Outliers	Less robust to outliers because the weight of misclassified points increases.	More robust because outliers can be treated as large residuals, and the model will not overfit them as easily.

XGBoost: An Advanced Gradient Boosting Variant

XGBoost (Extreme Gradient Boosting) is a highly optimized version of Gradient Boosting.

Why is XGBoost Popular?

- **Speed and Efficiency:** XGBoost is optimized for computational speed and memory efficiency. It uses advanced techniques like parallel computing, tree pruning, and cache awareness to train models faster than regular Gradient Boosting. Implements out-of-core computation for large datasets that don't fit into memory.
- **Regularization:** XGBoost includes built-in L1 and L2 regularization, which helps control overfitting and makes the model more robust to noise.
- **Handling Missing Data:** XGBoost can automatically handle missing data by learning which path to take for missing values, making it easier to work with real-world data.
- **Custom Objective Functions:** You can use custom loss functions in XGBoost, making it highly flexible for different types of problems (e.g., ranking, classification, regression).
- **Feature Importance:** XGBoost provides easy access to feature importance, allowing data scientists to interpret which features have the highest impact on model predictions.
- **Scalability:** XGBoost is scalable to large datasets and is often used in machine learning competitions (e.g., Kaggle) and industry.

XGBoost: An Advanced Gradient Boosting Variant

When to Use XGBoost?

- Tabular data with structured rows and columns.
- When you have a large dataset and need a fast and scalable solution.
- When overfitting is a concern, as XGBoost's regularization helps mitigate it.
- When you need a powerful and flexible algorithm for complex problems.

Practical Guide: XGBoost for Beginners

Key Hyperparameters in XGBoost:

- **n_estimators:** The number of trees to be built. Increasing this value can improve model accuracy but also increase training time.
- **learning_rate:** Controls the contribution of each tree. Lower values result in better performance but require more trees.
- **max_depth:** The maximum depth of each tree. Deeper trees can model more complex relationships but are prone to overfitting.
- **subsample:** The fraction of the training data used for building each tree. Lowering this value can prevent overfitting.
- **colsample_bytree:** The fraction of features to consider when building each tree. Reducing this helps prevent overfitting.
- **reg_alpha and reg_lambda:** L1 and L2 regularization terms to control overfitting.

Practical Guide: XGBoost for Beginners

XGBoost Tips for Beginners:

- Start with the default parameters, then gradually tune n_estimators, max_depth, and learning_rate.
- Use cross-validation to avoid overfitting, especially when tuning hyperparameters.
- Regularization is key! Use reg_alpha (L1 regularization) and reg_lambda (L2 regularization) to control overfitting.
- Use early_stopping_rounds to stop training once performance stops improving on a validation set.

CatBoost (Categorical Boosting)

CatBoost (Categorical Boosting) is a gradient boosting algorithm specifically optimized for handling categorical features. Unlike XGBoost, CatBoost can handle categorical data without the need for manual encoding like one-hot encoding or label encoding.

Why is CatBoost Popular?

1. **Native Handling of Categorical Features**:
    1. CatBoost automatically handles categorical features without preprocessing, which is a huge advantage for datasets with high-cardinality categorical variables.
2. **Faster Training**:
    1. CatBoost uses **ordered boosting**, which reduces prediction shift, making the model less prone to overfitting and improving training speed.
3. **Robust to Overfitting**:
    1. Includes several built-in regularization techniques, making it less likely to overfit, especially on small datasets.
4. **High Accuracy with Minimal Tuning**:
    1. CatBoost often requires fewer hyperparameter tuning efforts to achieve competitive accuracy.

CatBoost (Categorical Boosting)

When to Use CatBoost?

- **Datasets with many categorical features**
- When you want to reduce the preprocessing burden and skip manual encoding steps.
- When you are looking for an algorithm that works well out of the box with minimal tuning.
- Particularly useful for **tabular data**, mainly when it contains a mix of categorical and numerical variables.

Practical Guide: CatBoost for Beginners

Key Hyperparameters in CatBoost:

- **iterations**: Number of boosting rounds (similar to n_estimators in XGBoost).
- **depth**: Maximum depth of trees. Shallow trees are faster and less prone to overfitting.
- **learning_rate**: The step size for updates; smaller values often improve accuracy.
- **l2_leaf_reg**: L2 regularization term on the leaf weights to prevent overfitting.
- **subsample**: Fraction of data to use for each tree. Lowering this can reduce overfitting.

Practical Guide: CatBoost for Beginners

CatBoost Tips for Beginners:

- CatBoost often performs well out of the box, but you can start tuning iterations and depth for better accuracy.
- Use verbose=0 to suppress the training output unless you want detailed logs during training.
- CatBoost is particularly effective when working with datasets that have many categorical features.
- Like XGBoost, cross-validation is useful for evaluating model performance and avoiding overfitting.

XGBoost vs CatBoost

- XGBoost is widely used because of its speed, scalability, and flexibility. It performs exceptionally well in large datasets and allows for fine-grained control with its many hyperparameters, making it a go-to for machine learning competitions and production models.

- CatBoost is particularly effective when dealing with categorical features and mixed datasets. It simplifies preprocessing (e.g., no need for manual encoding), and its ordered boosting and automatic handling of categorical variables make it beginner-friendly and robust for business applications without extensive tuning.

- If you're working with tabular data that is mostly numerical, and you need maximum control over model tuning for high performance, start with XGBoost.

- If your dataset contains many categorical variables and you want to avoid the hassle of encoding them, or if you need a model that works well with minimal tuning, CatBoost is a great choice.