Project 1

# Lasso

# Gradient Boosting

In Gradient Boosting Regression (GBR), feature importance scores are typically calculated based on the number of times a feature is used to split the data across all trees, and the improvement in the performance measure brought about by the splits on this feature.

1. \*\*Frequency-Based Importance (Count)\*\*: This is the simplest way to compute feature importance. It calculates how many times each feature is used to split the data across all trees in the ensemble. The idea is that if a feature is often used to make decisions, it must be important. However, this method can be biased towards features with more categories.

2. \*\*Weighted Average Importance (Gain)\*\*: This method calculates the average improvement in accuracy brought by the splits on each feature. Each split in each tree is assigned a score based on the improvement it brings (e.g., reduction in Gini impurity for classification, reduction in variance for regression). The score is then averaged across all trees in the ensemble. This method gives a more balanced importance measure, as it takes into account not just frequency, but also the effectiveness of each split.

3. \*\*Cover\*\*: This method calculates the average coverage of each feature, where coverage is defined as the number of samples affected by the splits on this feature. Like the Gain method, it also averages the scores across all trees in the ensemble.

In the XGBoost library, the `feature\_importances\_` attribute returns the Gain importance scores by default, but this can be changed using the `importance\_type` parameter when creating the model.

It's important to note that feature importance scores are a heuristic and can be misleading in some cases, especially when dealing with correlated features. They should be used as a guide to understand the model, but not as a definitive measure of feature importance.

In Gradient Boosting Regression, the `max\_depth` parameter refers to the maximum depth of the individual regression estimators. This parameter controls the size of the decision tree ensembles that are used as base learners.

Deep trees can model complex interactions because they can have many branches, and therefore fit the data very closely. However, they are also more likely to overfit to the training data, capturing noise in the data and making the model perform poorly on unseen data.

On the other hand, shallow trees (i.e., with a small `max\_depth`) might not capture important information, leading to underfitting.

Therefore, tuning the `max\_depth` parameter is important to find the right balance between underfitting and overfitting. In the code snippet you provided, `max\_depth` is set to vary between 3 and 7, inclusive. This means the model will be trained multiple times with different maximum depths to find the optimal depth that produces the best performance on the validation set.

In Gradient Boosting Regression, the `learning\_rate` parameter, also known as the shrinkage parameter, is a hyperparameter that determines the impact of each tree on the final outcome.

The `learning\_rate` scales the contribution of each tree added to the ensemble. If you set it to a low value, such as 0.1, each tree that you add to the ensemble helps us to correct the mistake of the previous set of trees in the ensemble by a small amount. This means you need to add many trees to the ensemble to effectively fit the training set. The advantage of this approach is that it can make the model less likely to overfit to the training data.

On the other hand, if you set the `learning\_rate` to a high value, such as 0.9, each tree's corrections can potentially correct the mistakes of the previous set of trees in the ensemble by a large amount. This means you don't need to add as many trees to the ensemble to fit the training set, but the model is more likely to overfit to the training data.

In Gradient Boosting Regression (GBR), the `n\_estimators` parameter refers to the number of boosting stages or the number of trees to be used in the ensemble.

Each "stage" in gradient boosting involves building a tree and adding it to the ensemble. Therefore, `n\_estimators` essentially controls the complexity of the model.

A larger number of trees can result in a more complex model, which can potentially capture more intricate patterns in the data. However, adding too many trees can lead to overfitting, where the model starts to learn not only the underlying patterns but also the noise in the training data. This can result in poor performance when the model is used to make predictions on new, unseen data.

On the other hand, if `n\_estimators` is set too low, the model might be too simple to capture important patterns in the data, leading to underfitting.

Therefore, `n\_estimators` is a crucial hyperparameter to tune in order to find the right balance between underfitting and overfitting. In practice, it's often used in combination with other parameters like `learning\_rate` and `max\_depth` to control the model's complexity and prevent overfitting.

The `min\_child\_weight` parameter in Gradient Boosting Regression (GBR) is a regularization parameter that can help prevent overfitting. It specifies the minimum sum of instance weights (hessian) needed in a child (or leaf) node.

In the context of GBR, trees are added to the model sequentially in an attempt to correct the residuals (errors) from the previous trees. Each tree is a simple decision tree, and each node in the tree represents a decision rule that splits the data. A node's "child" nodes are the nodes that result from the split.

The `min\_child\_weight` parameter imposes a constraint on the sum of instance weights in each child node. If a potential split results in a child node with a sum of instance weights less than `min\_child\_weight`, then the split is not made and the node is made a leaf node.

This can have the effect of smoothing the model by preventing splits that would result in leaf nodes with very few instances, which can help to prevent overfitting.

For example, consider a case where `min\_child\_weight` is set to 10. If a potential split in the data would result in a child node with only 5 instances, then this split would not be made, regardless of how much it might decrease the overall loss of the model. This can prevent the model from learning overly complex patterns in the training data that might not generalize well to new data.

However, setting `min\_child\_weight` too high can prevent the model from learning necessary splits, leading to underfitting. Therefore, it's important to find a good balance for this parameter, often through a process like cross-validation.

The choice between the `reg:tweedie` and `reg:squarederror` objective functions in Gradient Boosting Regression (GBR) depends on the nature of the target variable in your dataset.

The `reg:squarederror` objective function, also known as Mean Squared Error (MSE), is a common choice for regression problems. It works well when the errors are normally distributed and the variance of the errors is constant across different levels of the predicted values (homoscedasticity).

On the other hand, the `reg:tweedie` objective function is designed for positive continuous variables that can have a large number of exact zeros, as well as for cases where the variance of the errors may not be constant (heteroscedasticity). It's often used in insurance claim modeling where the target variable (e.g., claim amount) can be exactly zero for a large number of instances, but can take on large positive values otherwise.

If the target variable in your dataset has these characteristics, then `reg:tweedie` might perform better than `reg:squarederror`. However, the best way to determine which objective function performs better is to try both and compare their performance on a validation set or through cross-validation.

1. `reg:squarederror`: This is the standard Mean Squared Error (MSE) used in many regression problems. It calculates the average of the squared differences between the predicted and actual values. Mathematically, it's represented as:

MSE = (1/n) \* Σ (y\_i - ŷ\_i)^2

where:

- n is the number of observations,

- y\_i is the actual value for the i-th observation,

- ŷ\_i is the predicted value for the i-th observation.

2. `reg:tweedie`: This objective function is based on the Tweedie distribution, which is a type of compound Poisson distribution often used in insurance claim modeling. The exact form of the Tweedie loss function depends on a power parameter `p`. However, in general, it's represented as:

Tweedie Loss = Σ (y\_i \* exp(-ŷ\_i) - ŷ\_i^(2-p) / (2-p))

where:

- y\_i is the actual value for the i-th observation,

- ŷ\_i is the predicted value for the i-th observation,

- p is the power parameter of the Tweedie distribution.

The key difference between these two objective functions is that the `reg:squarederror` objective function assumes that the errors are normally distributed and have constant variance, while the `reg:tweedie` objective function can handle cases where the target variable is positive and zero-inflated, and the variance of the errors is not constant.

# Alternatives

## Model

* Linear regression
* Poisson regression
* Decision tree regression: Inferior as the tree fitted is heavily dependent on the training data, making it not as generalizable.
* Random forest regression: Better than DTR as it uses bagging and feature selection to compute each tree, but very computationally demanding and still slightly more variable due to stochastic nature.