VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

PATTERN RECOGNITION PROJECT

Lecturer: TS. Ngô Minh Mẫn

Group 3:

Đoàn Thị Trâm - 20C29016
Nguyễn Thị Thu Thảo - 20C29035
Phạm Thị Hồng Phụng - 20C29033
Nguyễn Thanh Việt Cường - 20C29018
Nguyễn Thị Kim Hoàng - 20C29022

TP. HỒ CHÍ MINH – 2021

CONTENTS

1. Introduction

In recent years, Machine learning (ML) has become more popular and applied to various fields in our lives. There are two common ways to classify machine learning algorithms. One is based on the learning style, and the other is based on the function. In this project, we will mention the learning style.

According to the learning method, ML includes two groups: Supervised and Unsupervised learning as depicted in Figure 1. Supervised learning is a group of algorithms that use labeled data to model the relationship between an input variable (x) and an output variable (y). The two primary groups of problems in supervised learning are classification and regression, in which the output variable of the classification problem has discrete values . In contrast, the output variable of the regression problem has discrete values or continuous values. With Supervised Learning, besides building strong models, collecting and labeling good and reasonable data also plays a key role in solving real-world problems. There are subproblems in regression problems, including Linear Polynomial, KNN, Penalized regression, SVR, and Tree-based Regression models. Some subproblems in classification problems include Logistic regression, KNN, SVC, and Tree-based Classification models. For Tree-based Classification models, one can use some algorithms such as Decision Tree (DTs), Bagging, Random Forests, and Boosting.

There are two basic groups for Unsupervised learning, including Dimensionality Reduction and Clustering. The principal component analysis (PCA) is widespread in dimensionality reduction problems. K-mean and hierarchical are two algorithms in clustering problems.

In this project, our group will present two problems:

- How to enhance imbalanced data in classification problems when using Bagging
- Use Monotonic constraint (partially) for Boosting.

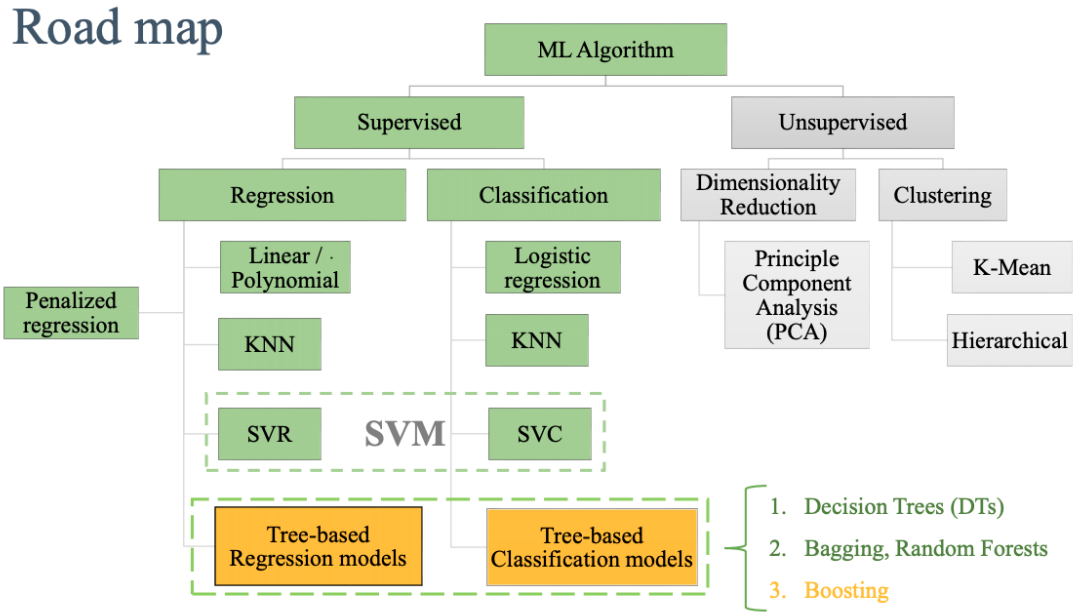

Figure 1. The roadmap of Machine Learning Algorithm

2. Content

2.1. How to enhance imbalanced data in classification problems when using Bagging

2.1.1. The issue

Learning from imbalanced data is an important problem in data mining research. Much research has addressed the problem of imbalanced data by using sampling methods to generate an equally balanced training set to improve the performance of the prediction models, but it is unclear what ratio of class distribution is best for training a prediction model. Bagging is one of the most popular and effective ensemble learning methods for improving the performance of prediction models. In the Bagging method, diversity of classifiers, the performance of classifiers, the appropriate number of bags (classifiers), and balanced training sets to train the classifiers are important factors in the success of Bagging so as to deal with imbalanced problems.

Because of the mechanism of bagging approaches, there is a chance that observations from minority classes be missed in generating bootstrapped datasets, especially when we deal with imbalanced data.

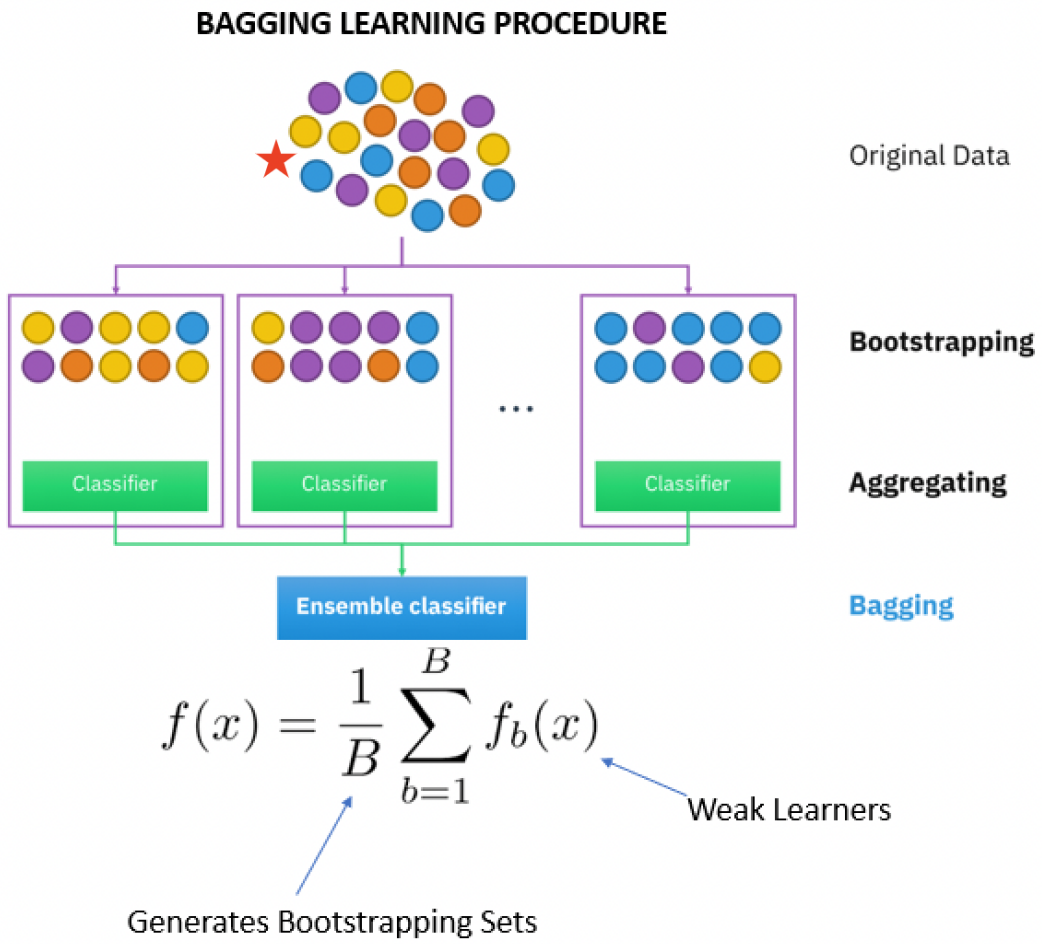

Figure 2. Bagging learning procedure

2.1.2. Approaches

To deal with the issue, there are several methods we can apply:

a. Resampling approach

In this approach, we will change the original dataset distribution before applying the bagging algorithm. The target of resampling approaches is to make sure that

the sample sizes of classification classes are less imbalanced. There are 03 primary ways to resample:

i. Under-sampling

By under-sampling, we reduce the size of the majority class with several techniques.

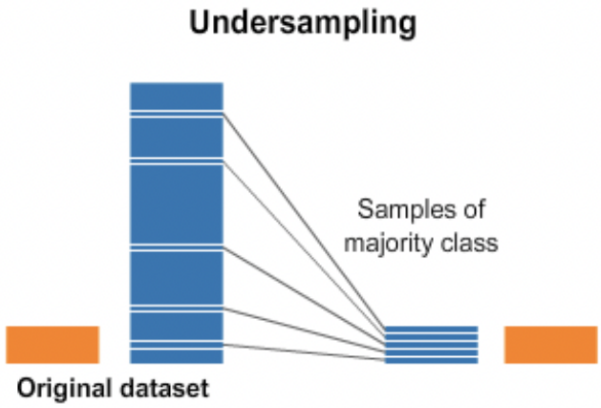

Figure 3. Under-sampling

The most simple way is random undersampling in which we just remove random records from the majority class.

Another popular method is Tomek's Links. This approach tries to detect pairs of observations that belong to different classes but are the nearest neighbors of each other and remove them in order to make the dataset more separated. In the default setting, all the pairs will be excluded but to deal with our concern, only observations from the majority class to be removed.

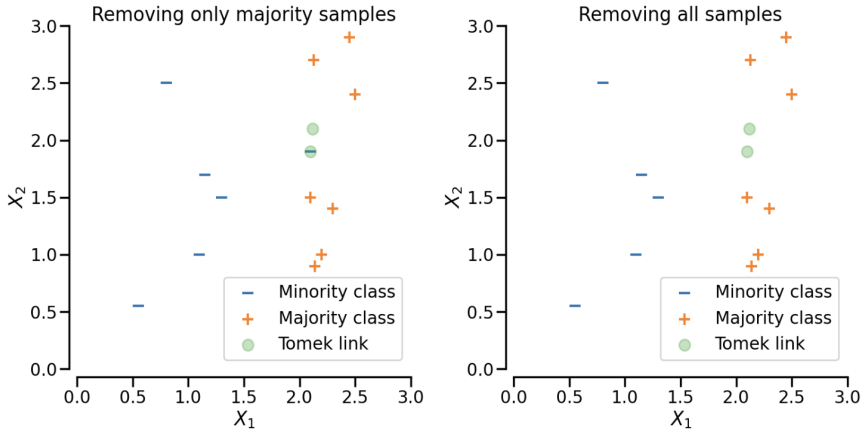

Figure 4. Tomek's Links

There are other methods such as Near Miss algorithms or Edited Nearest Neighbors, but in practice, we need to be careful when using Under Sampling approaches since they could lead to significant information loss.

ii. Over-sampling (Duplicating)

In over-sampling approaches, we enrich the minority group with various techniques.

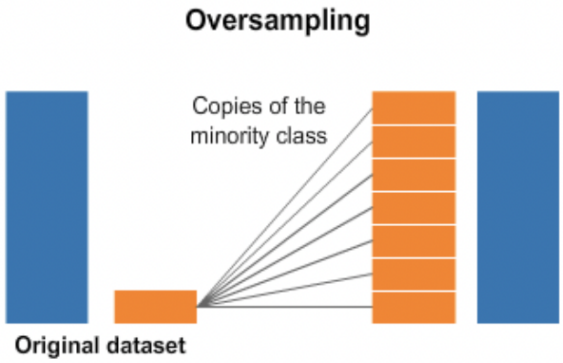

Figure 5. Oversampling

Similar to under-sampling, we can apply random sampling to increase the size of the minority class. It makes the data duplicated.

We can enrich the minority class with synthetic data using techniques like SMOTE. Some variants of SMOTE such as SMOTE-NC can deal with categorical data as well.

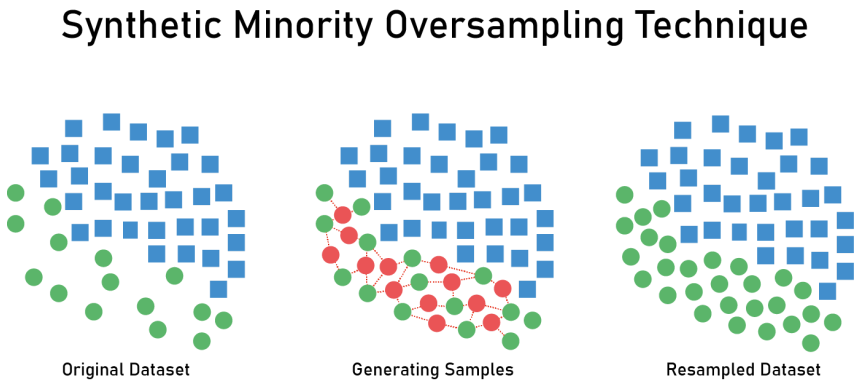

Figure 6. Synthetic Minority Oversampling Technique

We have a similar technique ADASYN which decides the appropriate number of synthetic data points based on the difficulty level of the original datasets.

iii. Combination Over-Sampling and Under-Sampling

We also can combine both approaches when dealing with imbalanced datasets. For example, we can SMOTE+TOMEK Links by generating synthetic data using SMOTE and then removing the resulting Tomek links.

b. Stratified bootstrapping and Balance Bagging approaches

This is a set of approaches that tackle directly to our concerns:

i. Original approach: generating datasets by random sampling from the original dataset (which may lead to the issue).

ii. Adjusted approach:

1. Apply stratified random sampling and get the output data with an unchanged ratio to make sure it contains the original dataset's distribution.
2. Instead of keeping the original distribution, the bootstrap process could be adjusted in order to output balanced datasets when training base learners, for example, Balanced Random Forest.

Figure 7.

c. Cost-Sensitive Learning

There are other approaches to deal with imbalanced datasets such as adjusting the cost function. The idea is to penalize the false majority prediction higher than the false minority prediction. For example in a dataset with majority class 0 and minority class 1

We have the original cost function: Loss = -ylog(p) -(1-y)log(1-p)

But if we want to put more penalization on false majority prediction to avoid the model predicting too much observation as 1, a new loss function will be set as: NewLoss = **-10**ylog(p) -(1-y)log(1-p).

This could be an efficient approach, however, it will not tackle directly our concern then we list it here as a way to improve the model results after dealing with the bootstrap issue.

2.2. Use Monotonic constraint (partially) for Boosting.

2.2.1. Definition

a. Monotonicity:

A non increasing or non decreasing function is said to be monotonic i.e. when x < y and f(x) < f(y) (monotonically increasing) or when x < y and f(x) > f(y) (monotonically decreasing)

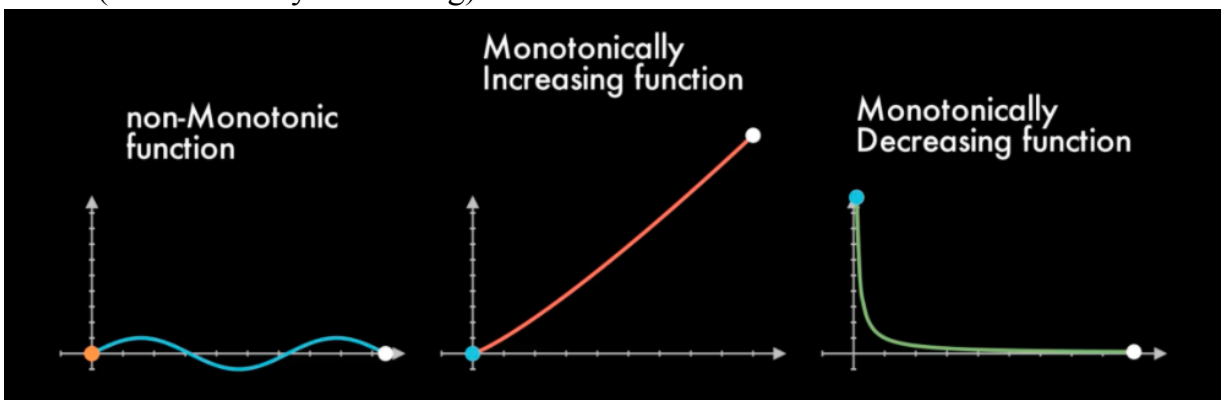

Figure 8.

b. Partial Dependence:

The partial dependency describes when other variables are fixed, and how the average response depends on a predictor. Therefore, partial dependence is used to view the global, average behavior of a variable under the monotonic model.

c. Boosting

- Boosting consists of using the "original" training data and iteratively creating multiple models by using a weak learner. Each new model would be different from the previous ones in the sense that the weak learner, by building each new model, tries to "fix" the errors which previous models make.

- In boosting, the trees are grown sequentially, which also means each tree is grown using information from the previous tree.
- It does not involve bootstrap sampling as bagging and being a family of weak learners should have a minimum correlation between them.
- Two main types:
    - Adaptive Boosting: Reweight datapoints based on the performance of the last weak learner. Focuses on points where the previous learner had trouble. Example: AdaBoost.
    - Gradient Boosting: Train new learners on residuals of the overall model. Constitutes gradient boosting because approximating the residual and adding to the previous result is essentially a form of gradient descent. Example: XGBoost.

2.2.2. The issue

In practice, there are some problems we need to solve following business sense, e.g., if other features keep unchanged, a larger house could lead to a higher price. But sometimes, the historical data does not align entirely with our expectations, and that behavior is not reflected in the model. Then we may need to apply some monotonic constraints to make the output results more intuitive. We want to create our house value model as an increasing function to feature house_area in the example; then, we could use the monotone_constraint hyper-parameter in boosting models.

1. Outliers + Sparse Areas

- When using the popular non-linear models such as random forests, boosted tree or neural networks, counter-intuitive scenarios, e.g. larger square footed associate with lower house price or lower median income of a neighborhood but have higher median house value. This mostly happens in the areas where data are spare or have a lot of noise or outliers.
- Hence, the application of monotonic constraint can provide better results in model performance on test data, meaning that the constrained models may generalize better and reduce overfitting, which can be viewed as a type of regularization
- In addition, high-level of outliers that contradict the domain knowledge so monotonic constraint can force the model to comply with the domain knowledge and help improve fairness.

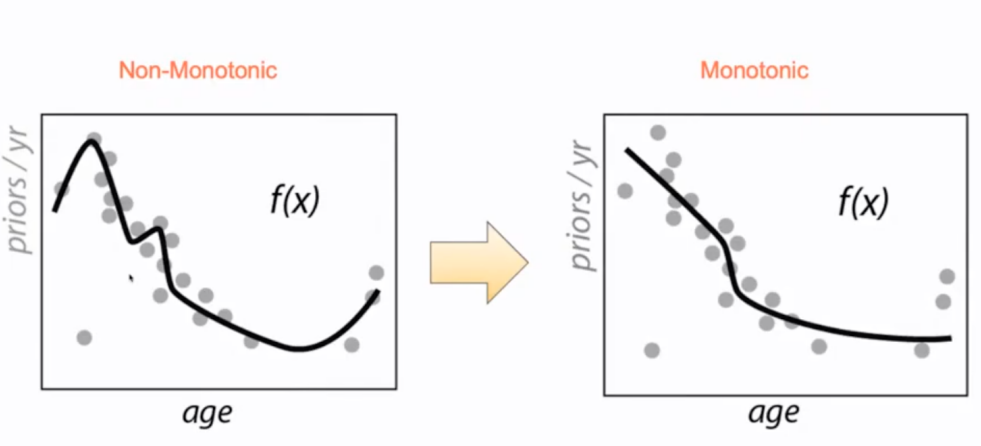

Figure 9. Non-Monotonic to Monotonic

2. Under-represented Groups :

Lack of data for one group vs others will create opportunities for the model to learn inconsistency between features and outcomes.

=> Monotonic constraint ensure that the relationship between features and outcome consistent.

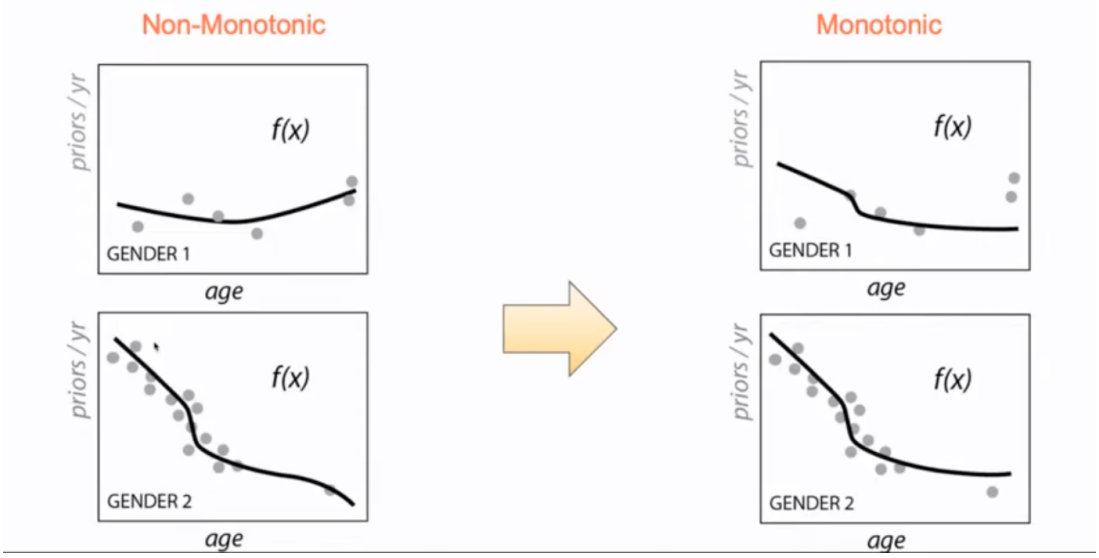

Figure 10.

3. Outside of the bounds of the training feature range:

When applying monotonic constraints on data outside the range of training feature range, a non-monotonic model will not have a consistent relationship between the predictor and feature. But a monotonic model will continue on-trend and be more reliable.

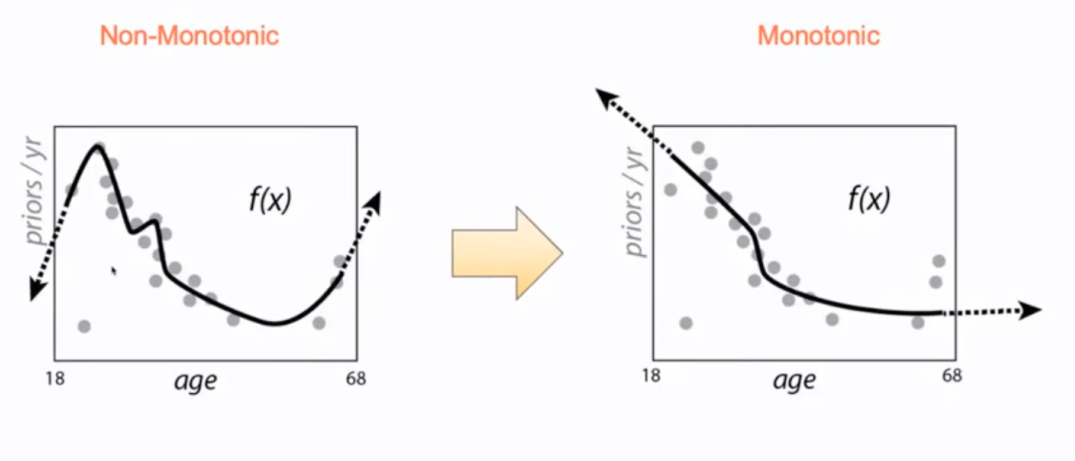

Figure 11.

To sum up, monotonic constraints can help the model improve fairness, consistency & reliability.

2.2.3. XGBoost Monotonic Constraint Implementation

- XGBoost is a refined and customized version of a gradient boosting decision tree system, created with performance and speed in mind.
- When growing a tree within an XGBoost estimator, the algorithm will abandon a candidate split if it causes a non-monotonic relationship. When specifying a positive monotonic constraint on a particular feature f , and at a certain internal node, f gets picked and a split at value v will result in the best gain. Because the constraint is positive, it is expected that the weight assigned to the right child to be higher than the left child. The algorithm checks if that is the case, if yes it will go on and split the node at value v , if not it will give up on v and will try to find the next best split until the weight of the right child is higher than that of the left child.

- The positive monotonic condition is specified if constraint > 0, then when the weight of the right child is higher than the left child. If the constraint is met, then the gain is retained, else the gain is replaced by negative infinity and hence the algorithm will abandon the split. This corresponds to rows 432 and 433 in the XGBoost code snippet below.
- Similarly, the negative monotonic condition is specified if constraint < 0, then the weight of the left child needs to be higher than that of the right child (row 434 and 435).

```
421   template <typename ParamT>
422     XGBOOST_DEVICE inline double CalcSplitGain(const ParamT &param, int constraint,
423                                  GradStats left, GradStats right) const {
424       const double negative_infinity = -std::numeric_limits<double>::infinity();
425       double wleft = CalcWeight(param, left);
426       double wright = CalcWeight(param, right);
427       double gain =
428           CalcGainGivenWeight(param, left.sum_grad, left.sum_hess, wleft) +
429           CalcGainGivenWeight(param, right.sum_grad, right.sum_hess, wright);
430       if (constraint == 0) {
431         return gain;
432       } else if (constraint > 0) {
433         return wleft <= wright ? gain : negative_infinity;
434       } else {
435         return wleft >= wright ? gain : negative_infinity;
436       }
437     }
438
439     inline void SetChild(const TrainParam &param, bst_uint split_index,
440                          GradStats left, GradStats right, ValueConstraint *cleft,
441                          ValueConstraint *cright) {
442       int c = param.monotone_constraints.at(split_index);
443       *cleft = *this;
444       *cright = *this;
445       if (c == 0) {
446         return;
447   }
448       double wleft = CalcWeight(param, left);
449       double wright = CalcWeight(param, right);
450       double mid = (wleft + wright) / 2;
451       CHECK(!std::isnan(mid));
452       if (c < 0) {
453         cleft->lower_bound = mid;
454         cright->upper_bound = mid;
455       } else {
456         cleft->upper_bound = mid;
457         cright->lower_bound = mid;
458       }
459     }
460   };
```

- The above meeting the constraint mechanism is of good use when the feature is picked one, however, if the tree goes on and does the same

thing to the left child (L) which happens to pick the same feature again, we will get a higher weight assigned to the right child of the left child (LR) because of the positive monotonic constraint we enforced. That weight, however, maybe even higher than that of the right child (R) we previously obtained. In other words, in order to maintain a monotonic positive relationship, not only should the weight assigned to the right child (R) be higher than the left child (L), but it should also be higher than its descendant that decides to use the same feature (LL, LR, LLL, LLR, LRL, LRR, etc.).

- To achieve a consistent monotonic relationship, when splitting on a node, the weight of the children node is bounded by the mean of their parent and uncle node's weight (uncle refers to the same level as the node's direct parent). This ensures that if a feature is picked multiple times in a tree, the weights assigned to the descendant nodes from the latest split will not go higher or lower than the ancestor nodes in a way that violates the monotonic constraint.
- To sum up, for each split candidate, the steps are:
  - Check the values of both leaves against the monotonicity constraints propagated from predecessors.
  - Check the monotonicity between two leaves.
  - Reject the split if the monotonicity is broken.