Section C

Zixia Zeng

2024-11-18

1. Extreme Gradient Boosting (XGBoost)

In Section C, I will investigate the performance of Extreme Gradient Boosting (XGBoost) in classification.

XGBoost (Chen and Guestrin 2016) is a supervised learning method based on decision tree and is a highly efficient and scalable implementation of gradient boosting. It builds multiple decision trees sequentially, where each new tree focuses on correcting the errors of the previous ones.

1.1 Underlying Principles

Boosting: XGBoost uses the boosting method, which involves combining multiple weak learners (typically decision trees) to create a strong learner. Each new tree is trained to correct the errors of the previous trees.

Gradient Descent: XGBoost uses gradient descent to optimize a loss function, iteratively adding trees to minimize the loss.

Regularization: XGBoost applies L_1 (lasso) and L_2 (ridge) regularization functions to reduce over-fitting.

1.2 Assumptions

- XGBoost assumes that the samples in the training data are independent and identically distributed (iid).
- XGBoost assumes that the features can be divided into discrete categories to build decision trees.
- XGBoost assumes that the training data is of high quality, without missing values or noise.

1.3 Objective Function

The objective function of XGBoost consists of two parts: loss function and regularization term:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

Where:

- $l(y_i, \hat{y}_i)$: The loss function.
- $\Omega(f_k)$: A regularization term for each tree to control complexity:

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

  Where:

  - T: The number of leaves in the tree.
  - w_j: The prediction score (weight) of leaf j.
  - γ: Penalty for adding a new leaf.
  - λ: Regularization parameter for controlling leaf weights.

1.4 Training Algorithm with Mathematical Formula

Step 1: XGBoost starts with an initial prediction of all data points, but the formula of initialization depends on different questions it focuses on. In Section C, I will use XGBoost to deal with a classification problem.

Step 2: Building decision trees by iteration, and at the t-th iteration, the objective function can be written as:

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{t=1}^{T} \left(\gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \right)$$

Then, according to Taylor expansion:

$$f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2} f''(x)\Delta x^2$$

The loss function of the objective function can be expanded to an approximation:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

where Gradient g_i is the first derivative of the loss function and Hessian h_i is the second derivative of the loss function, and they represent the residuals.

Next, by using residuals g_i and h_i, I can fit a new decision tree. Firstly, for each possible split of a feature, calculate the split gain, which is the reduction in the loss function from splitting the data into two subsets (left subset is L and right subset is R).

The gain is given by:

$$\text{Gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma$$

Where:

- $G_L = \sum_{i \in L} g_i$, $G_R = \sum_{i \in R} g_i$: Sum of gradients for the left and right subsets.
- $H_L = \sum_{i \in L} h_i$, $H_R = \sum_{i \in R} h_i$: Sum of Hessians for the left and right subsets.
- λ: Regularization parameter.
- γ: Minimum gain required to make a split.

A split is accepted if the gain is positive, indicating an improvement in the objective function.

Secondly, after determining the structure of the tree, I can calculate the optimal weight for each leaf j:

$$w_j^* = -\frac{\sum_{i \in \text{leaf}_j} g_i}{\sum_{i \in \text{leaf}_j} h_i + \lambda} = -\frac{G_j}{H_j + \lambda}$$

Therefore, the regularization term will be optimized and applied to next training iteration:

$$\Omega(f_k)^* = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} -\frac{G_j}{H_j + \lambda}^2$$

Finally, the algorithm will add the regularization term to penalize complexity to avoid over fitting and update the predictions:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta f_t(x_i)$$

Where:

- η is learning rate, which scales the contribution of each tree to ensure gradual improvement.

Step 3: Repeat the above steps for a predefined number of iterations or until the model meets the stopping criteria.

Here is a diagram of XGBoost algorithm:

1.5 Make New Predictions

After training the model, for a test sample x, XGBoost aggregates the outputs of all trees:

$$\hat{y} = \sum_{k=1}^{K} f_k(x), \quad f_k \in \mathcal{F}$$

Where:

- $f_k(x)$: The prediction from the k-th decision tree.
- $\mathcal{F}$: The set of all possible decision trees.

For classification problem, the output is often transformed into probabilities using a sigmoid (binary) or softmax (multi-class) function.

1.6 Types of Problems Suitable for XGBoost

XGBoost is suitable for solving regression on predicting continuous values like stock price and classification problems like binary classification and multi-class classification.

In Section C, I decide to solve a multi-class classification problem using Car Evaluation (Bohanec 1988) data set provided by UC Irvine machine learning repository.

2. Introdction of Car Evaluation data set

Firstly, retrieving data using the UCI machine learning repository URL and download it. However, the file only contains the data without the columns' name. Therefore, I need to add the columns' name according to the variables table (Bohanec 1988) , here is a screen shot of the variables table:

```
# Download the Car Evaluation dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/car/car.data"
data = read.csv(url, header = FALSE, stringsAsFactors = TRUE)

# Add column names
colnames(data) <- c("buying", "maint", "doors", "persons", "lug_boot", "safety", "class")

# Check the structure of the data set
str(data)
```

```
## 'data.frame':    1728 obs. of  7 variables:
##  $ buying  : Factor w/ 4 levels "high","low","med",..: 4 4 4 4 4 4 4 4 4 4 ...
##  $ maint   : Factor w/ 4 levels "high","low","med",..: 4 4 4 4 4 4 4 4 4 4 ...
##  $ doors   : Factor w/ 4 levels "2","3","4","5more": 1 1 1 1 1 1 1 1 1 1 ...
##  $ persons : Factor w/ 3 levels "2","4","more": 1 1 1 1 1 1 1 1 1 2 ...
##  $ lug_boot: Factor w/ 3 levels "big","med","small": 3 3 3 2 2 2 1 1 1 3 ...
##  $ safety  : Factor w/ 3 levels "high","low","med": 2 3 1 2 3 1 2 3 1 2 ...
##  $ class   : Factor w/ 4 levels "acc","good","unacc",..: 3 3 3 3 3 3 3 3 3 3 ...
```

```
table(data$class)
```

```
##
##   acc  good unacc vgood
##   384    69  1210    65
```

From Figure 2 and str() function, data set contains 1,728 examples and six categorical features and one categorical target with no missing values. Moreover, there are four classes in target value “class”, “unacc” means unacceptable, “acc” mean acceptable, “good” means the car is better than acceptable, and “vgood” means very good. Therefore, multi-classification is well-suited for this data set.

However, by using table() function, most examples are concentrated in one class, while the other classes have fewer instances. This imbalance can lead to issues where the model performs well on the majority class while poorly on the minority classes.

The goal of modeling Car Evaluation data set is using the supervised learning method, which is XGBoost, to build a model to classify the cars into four classes (unacceptable, acceptable, good and very good) based on the six features. Since the data set only has 1,728 observations, XGBoost is well-suited for this small data set, which means the accuracy of the model would be higher.

In next part, I will apply XGBoost by R.

3. Apply XGBoost

3.1 Data Preprocessing

Firstly, load packages needed.

```
# Load packages and do not show any messages
# XGBoost package
suppressMessages(library(xgboost))
```

```
## Warning: 程序包'xgboost'是用R版本4.4.2 来建造的
```

```
# machine learning package
suppressMessages(library(caret))
```

```
## Warning: 程序包'caret'是用R版本4.4.2 来建造的
```

```
# data wrangling
suppressMessages(library(tidyverse))

# data visualization
suppressMessages(library(ggplot2))
```

Then, preprocessing the raw data for later model training.

```
# Convert the target variable 'class' to numeric labels
# XGBoost requires 0-based labels
data$class = as.numeric(factor(data$class)) - 1
```

Since XGBoost can not handle the raw categorical variables, I need to transfer the categorical variables to dummy variables (dummy coding).

Dummy Coding is a method of representing categorical variables as binary (0 or 1) variables in statistical models. It uses one category as the reference level, and this reference category will not have its own variable to avoid multicollinearity. If a dummy variable is created for every category, this means that the sum of all dummy variables for a single feature would always equal 1, leading to redundancy in the data.

```
# Transform all raw categorical variables into dummy variables
# Build full-rank dummy variables to avoid multicollinearity issues
dummies = dummyVars(~ ., data = data, fullRank = TRUE)
# Build a new data frame that can be used for XGBoost
data_transformed = data.frame(predict(dummies, newdata = data))
# Check the structure of dummy variables
str(data_transformed)
```

```
## 'data.frame':    1728 obs. of  16 variables:
##  $ buying.low    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ buying.med    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ buying.vhigh  : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ maint.low     : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ maint.med     : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ maint.vhigh   : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ doors.3       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ doors.4       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ doors.5more   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ persons.4     : num  0 0 0 0 0 0 0 0 0 1 ...
##  $ persons.more  : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ lug_boot.med  : num  0 0 0 1 1 1 0 0 0 0 ...
##  $ lug_boot.small: num  1 1 1 0 0 0 0 0 0 1 ...
##  $ safety.low    : num  1 0 0 1 0 0 1 0 0 1 ...
##  $ safety.med    : num  0 1 0 0 1 0 0 1 0 0 ...
##  $ class         : num  2 2 2 2 2 2 2 2 2 2 ...
```

Next, I will set the hyper-parameters for the model. Hyper-parameters are set before training, they control the model's learning process and performance. Meanwhile, hyper-parameters are not learned from the data but are instead set by the user and often tuned to optimize the model's performance.

Here, I will set the hyper-parameters of XGBoost randomly, and later I will try to use tuning techniques to find the best combination of hyper-parameters.

```
# Set hyper-parameters for the XGBoost model
params = list(
  # This is a multi-classification problem
  objective = "multi:softmax",
  # Number of classes
  num_class = length(unique(data_transformed$class)),
  # Learning rate
  eta = 0.1,
  # Maximum depth of trees
  max_depth = 6,
  # Subsample ratio
  subsample = 0.8,
  # Feature subsampling ratio
  colsample_bytree = 0.8,
  # Evaluation metric: classification error
  eval_metric = "merror"
)
```

3.2 Find Appropriate Performance Metric

Before starting model training, I still need to decide what performance metric to use when evaluating the model.

As mentioned before, the data set is imbalanced, so using confusion matrix and calculate the accuracy of this model is not appropriate enough. The mathematical formula of accuracy is:

$$\text{Accuracy} = \frac{Number\,of\,Correct\,Prediction}{Total\,Prediction}$$

This metric only considers the proportion of correct predictions but ignores the fact that there are more examples of “unacc,” which means the model will perform better for this majority class. However, for the minority class, such as “vgood,” the performance will not remain the same. Therefore, the accuracy will looks higher when using this imbalanced data set to train the model, but it can not represent the true performance of the model.

However, the Weighted F1-Score can show the true performance of the model better theoretically, because it evaluates precision and recall for all classes, emphasizing correct classification of minority classes.

The formula of F1-Score for one class i is:

$$\text{F1-Score}_i = \frac{2 \cdot Precision_i \cdot Recall_i}{Precision_i + Recall_i}$$

Where: - Precision for class i is:

$$\text{Precision}_i = \frac{TruePositives_i}{TruePositives_i + FalsePositives_i}$$

- Recall for class i is:

$$\text{Recall}_i = \frac{TruePositives_i}{TruePositives_i + FalseNegatives_i}$$

The Weighted F1-Score (Hinojosa Lee, Braet, and Springael 2024) is an improvement based on F1-Score, which adjusts the F1-Score for each class by adding a weight that is represented by the number

of example in each class i.

$$\text{Weighted F1-Score} = \frac{\sum_{i=1}^{C} n_i \cdot \text{F1-Score}_i}{\sum_{i=1}^{C} n_i}$$

Where:

- C is the total number of classes.
- n_i is the number of example in class i.

According to the mathematical formula, I write a R function to calculate Weighted F1-Score:

```
# Function to calculate weighted F1-Score
weightedF1 = function(true, pred) {
  cm = confusionMatrix(as.factor(pred), as.factor(true))
  # Calculate precision according to confusion matrix
  precision = diag(cm$table) / rowSums(cm$table)

  # Calculate Recall according to confusion matrix
  recall = diag(cm$table) / colSums(cm$table)

  # Calculate F1-Score for each class
  f1 = 2 * (precision * recall) / (precision + recall)

  # Combine the weight (Number of example in each class)
  weighted_f1 = sum(f1 * colSums(cm$table) / sum(cm$table), na.rm = TRUE)
  return(weighted_f1)
}
```

Now, starting training process and calculate accuracy and Weighted F1-Score then making comparison. Meanwhile, training data size will varies and I will compare the performance of the model across different sizes of training data. In the R code, the proportion of training set starts from 0.2 to 0.9, increased by 0.05.

```
# Initialize results storage
results = data.frame(T_Size = numeric(),
                     T_W_F1 = numeric(),
                     Val_W_F1 = numeric(),
                     T_Accuracy = numeric(),
                     Val_Accuracy = numeric())

# Create a sequence to record different training size
training_sizes = seq(0.2, 0.9, by = 0.05)

# Train model using different training size
for(size in training_sizes){
  # Create subset for training data
  subset_index = sample(1:nrow(data_transformed), size = floor(size * nrow(data_transformed)))
  subset_train_data = data_transformed[subset_index, ]
  subset_validation_data = data_transformed[-subset_index, ]

  # Separate features and target for training data
  X_train_subset = as.matrix(subset_train_data[, -ncol(subset_train_data)])
  y_train_subset = subset_train_data$class

  # Separate features and target for validation data
  X_validation_subset = as.matrix(subset_validation_data[, -ncol(subset_validation_data)])
  y_validation_subset = subset_validation_data$class

  # Convert data to DMatrix for XGBoost
  dtrain_subset = xgb.DMatrix(data = X_train_subset, label = y_train_subset)
  dvalidation_subset = xgb.DMatrix(data = X_validation_subset, label = y_validation_subset)

  # Train XGBoost model
  model = xgb.train(params = params,
                    data = dtrain_subset,
                    nrounds = 50,
                    watchlist = list(train = dtrain_subset, val = dvalidation_subset),
                    verbose = 0)

  # Predict on training data
  train_preds = predict(model, X_train_subset)
  # Convert to class labels
  train_preds = as.integer(train_preds)

  # Predict on validation data
  validation_preds = predict(model, X_validation_subset)
  # Convert to class labels
  validation_preds = as.integer(validation_preds)

  # Calculate Weighted F1-Scores
  train_f1 = weightedF1(y_train_subset, train_preds)
  validation_f1 = weightedF1(y_validation_subset, validation_preds)

  # Calculate Accuracy
  train_accuracy = mean(train_preds == y_train_subset)
  validation_accuracy = mean(validation_preds == y_validation_subset)

  # Store results
```

```
  results = rbind(results, data.frame(T_Size = size,
                                      T_W_F1 = train_f1,
                                      Val_W_F1 = validation_f1,
                                      T_Accuracy = train_accuracy,
                                      Val_Accuracy = validation_accuracy))
}

print(results)
```

```
##    T_Size    T_W_F1  Val_W_F1 T_Accuracy Val_Accuracy
## 1    0.20 0.9796866 0.8702214  0.9797101    0.8778019
## 2    0.25 0.9859854 0.8788179  0.9861111    0.8788580
## 3    0.30 0.9843952 0.8825513  0.9845560    0.8958678
## 4    0.35 0.9848840 0.8856592  0.9850993    0.8950178
## 5    0.40 0.9856412 0.9099314  0.9855282    0.9132112
## 6    0.45 0.9792756 0.8844100  0.9794080    0.8906414
## 7    0.50 0.9872357 0.9040919  0.9872685    0.9120370
## 8    0.55 0.9884374 0.9046614  0.9884211    0.9074550
## 9    0.60 0.9837832 0.9266493  0.9835907    0.9291908
## 10   0.65 0.9811646 0.9044488  0.9813001    0.9090909
## 11   0.70 0.9843256 0.9155710  0.9842845    0.9210019
## 12   0.75 0.9808012 0.9368301  0.9807099    0.9375000
## 13   0.80 0.9812877 0.9401106  0.9811867    0.9364162
## 14   0.85 0.9810755 0.9236160  0.9809264    0.9230769
## 15   0.90 0.9821175 0.9261052  0.9819936    0.9190751
```

From the result, both training Weighted F1-Score and training Accuracy are very high (around 0.98), these show the XGBoost is a powerful superivsed learning algorithm, and during training process, it can achieve nearly perfect predictions for the training data.

However, the validation Accuracy and validation Weighted F1-Score are not stable, varying dramatically from around 0.85 to around 0.93, and they are much lower than the training metrics. This means that the model performs well on the training data set but fails to generalize to testing data, indicating that the model exists overfitting.

For a single columns, as the training data size grows, both the accuracy and Weighted F1-Score will increase and the digits will reach the highest point at around 0.8 to 1.0, which means it is suitable to set around 80% of the whole data set as training data.

Then I will create a plot for the result to help finding the trend.

```
# Create a plot to show how F1-Score varies as the training data set increase
ggplot(results, aes(x = T_Size)) +
  geom_line(aes(y = T_W_F1, color = "Training Weighted F1")) +
  geom_line(aes(y = Val_W_F1, color = "Validation Weighted F1")) +
  geom_line(aes(y = T_Accuracy, color = "Training Accuracy")) +
  geom_line(aes(y = Val_Accuracy, color = "Validation Accuracy")) +
  labs(title = "Figure 3: Performance Metrics vs Training Size",
       x = "Training Data Size (Proportion)",
       y = "Metrics") +
  scale_color_manual(name = "Metrics",
                     values = c("Training Weighted F1" = "blue",
                                "Validation Weighted F1" = "red",
                                "Training Accuracy" = "green",
                                "Validation Accuracy" = "orange"))  +
  theme_minimal()
```

From Figure 3, when training data size's proportion is lower than 0.7, the gap between Training metrics and Validation metrics are very large, which means too small training size leads to overfitting. When there is not enough training data, the model will just memorize the results of training data but can not make predictions on unseen data well, this can lead to the large gap between Training and Validation metrics.

However, when the proportion of training data exceed 0.7, the Validation metrics seems to reach the optimal region, which means the model performance is optimal when training size is around 0.7 to 0.9. Unfortunately, there still exists large gap between Training and Validation metrics as the training size increases. To enhance the model prediction performance, I will explore the impact of hyper-parameters later.

Moreover, Training Accuracy and Training Weighted F1-Score are almost the same (close to 1.0), maybe because the model performs high precision and recalls for all classes on the training data. However, the Validation Weighted F1-Score is slightly lower than the Validation Accuracy when training data size is small,

showing that the model is not predicting minority classes well on new data. This is because Validation Accuracy may overestimate performance because it doesn't penalize the model for poor recall on minority classes as strongly as Weighted F1-Score does.

Although the two metrics do not seem to have significant difference when dealing with sufficient training sizes (over 0.7 proportion), Weighted F1-Score still can be more reasonable for this imbalanced data set, especially when dealing with small training sets. Therefore, **Weighted F1-Score** is an appropriate metric for evaluating the performance of XGBoost.

3.3 Impact of Different Training Set Sizes

Next, I will explain the impact of different training data size. From Figure 3, I observe that the Training Weighted F1-Score remains high (close to 1) across all training size. This suggests that the XGBoost performs well on training data, even with small training data sizes.

However, Validation Weighted F1-Score starts lower for smaller training sizes and increases as the training size grows, which means the insufficient training size has negative impact on the model's generalization. As training size increases, the gap between training and validation performance decreases, meaning the model performs better with larger training size.

Additionally, at around 70% to 90% training size, the Validation Weighted F1-Score converges and reaches a satisfactory level. Therefore, a training size of approximately 80% seems to provide the optimal balance between performance and efficiency, as adding more data beyond this point provides minimal improvement or even negative impact. Therefore, in later model adjustment, I will keep using 80% of the whole data set as the training size.

4. Impact on varying learning rate η

This part will explore how the performance of XGBoost model varies on both the training data and the validation data as as the learning rate η (a hyper-parameter) is varied.

Learning rate, as a hyper-parameter, its value is between 0 to 1 and it determines the step size during optimization. Smaller values lead to better generalization but may require more boosting rounds, increasing cost of computation. In contrast, larger values make the process faster, but potentially increase the risk of overfitting.

Firstly, I will train the model with different learning rate and record the Weighted F1-Score.

```
# Learning rate sequence
learning_rates = seq(0.01, 1, by=0.05)

# Create result for different learning rate
results_rl = data.frame(learning_rate = numeric(),
                        T_W_F1 = numeric(),
                        Val_W_F1 = numeric())

# Set params without learning rate, adjust learning rate in each iteration
params = list(
        objective = "multi:softmax",
        num_class = length(unique(data_transformed$class)),
        eval_metric = "merror",
        max_depth = 6,
        subsample = 0.8,
        colsample_bytree = 0.8
    )

# Set 80% training and 20% validation data
train_index = createDataPartition(data_transformed$class, p = 0.8, list = FALSE)
train_data = data_transformed[train_index, ]
validation_data = data_transformed[-train_index, ]

# Separate features and target for training and validation data
X_train = as.matrix(train_data[, -ncol(train_data)])
y_train = train_data$class
X_validation = as.matrix(validation_data[, -ncol(validation_data)])
y_validation = validation_data$class

# Convert training and validation data to DMatrix format for XGBoost
dtrain = xgb.DMatrix(data = X_train, label = y_train)
dvalidation = xgb.DMatrix(data = X_validation, label = y_validation)

for (lr in learning_rates) {

  # Update the learning rate in the parameters
  params$eta = lr

  # Train the XGBoost model
  model <- xgb.train(params = params,
                     data = dtrain,
                     nrounds = 50,
                     watchlist = list(train = dtrain, val = dvalidation),
                     verbose = 0)

  # Predictions on training and validation data
  train_preds = predict(model, X_train)
  train_preds = as.integer(train_preds)
  validation_preds = predict(model, X_validation)
  validation_preds = as.integer(validation_preds)

  # Calculate Weighted F1-Score for both training and validation sets
  train_f1 = weightedF1(y_train, train_preds)
  validation_f1 = weightedF1(y_validation, validation_preds)
```

```
  # Store the results
  results_rl = rbind(results_rl, data.frame(learning_rate = lr,
                                     train_W_F1 = train_f1,
                                     Val_W_F1 = validation_f1))
}
head(results_rl)
```

	learning_rate <dbl>	train_W_F1 <dbl>	Val_W_F1 <dbl>
1	0.01	0.9271418	0.8740641
2	0.06	0.9679740	0.8875331
3	0.11	0.9849369	0.9367814
4	0.16	0.9963995	0.9484454
5	0.21	0.9978425	0.9538619
6	0.26	0.9992798	0.9677961

6 rows

```
# Plot the results
ggplot(results_rl, aes(x = learning_rate)) +
  geom_smooth(aes(y = train_W_F1, color = "Training Weighted F1-Score")) +
  geom_smooth(aes(y = Val_W_F1, color = "Validation Weighted F1-Score")) +
  scale_color_manual(values = c("Training Weighted F1-Score" = "cyan",
                                "Validation Weighted F1-Score" = "brown")) +
  labs(title = "Figure 4 Weighted F1-Score vs Learning Rate",
       x = "Learning Rate", y = "Weighted F1-Score") +
  theme_minimal()
```

```
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

From Figure 4, it is obvious that the Validation Weighted F1-Score increases sharply (from around 0.93 to around 0.97) as the learning rate increase from 0.01 to 0.25. However, when the learning rate exceeds 0.25, both the training and validation Weighted F1-Scores' curves converge to nearly a straight line. These observations indicate that the best learning rate for this model is around 0.25. Moreover, if using a larger learning rate than 0.25, it will lead to a higher risk of overfitting.

5 Find optiaml learning rate using cross-validation

5.1 Introduce Cross-validation

To validate the observation from Figure 4, cross-validation will be used to accurately determine the optimal learning rate for this model (Arlot and Celisse 2010).

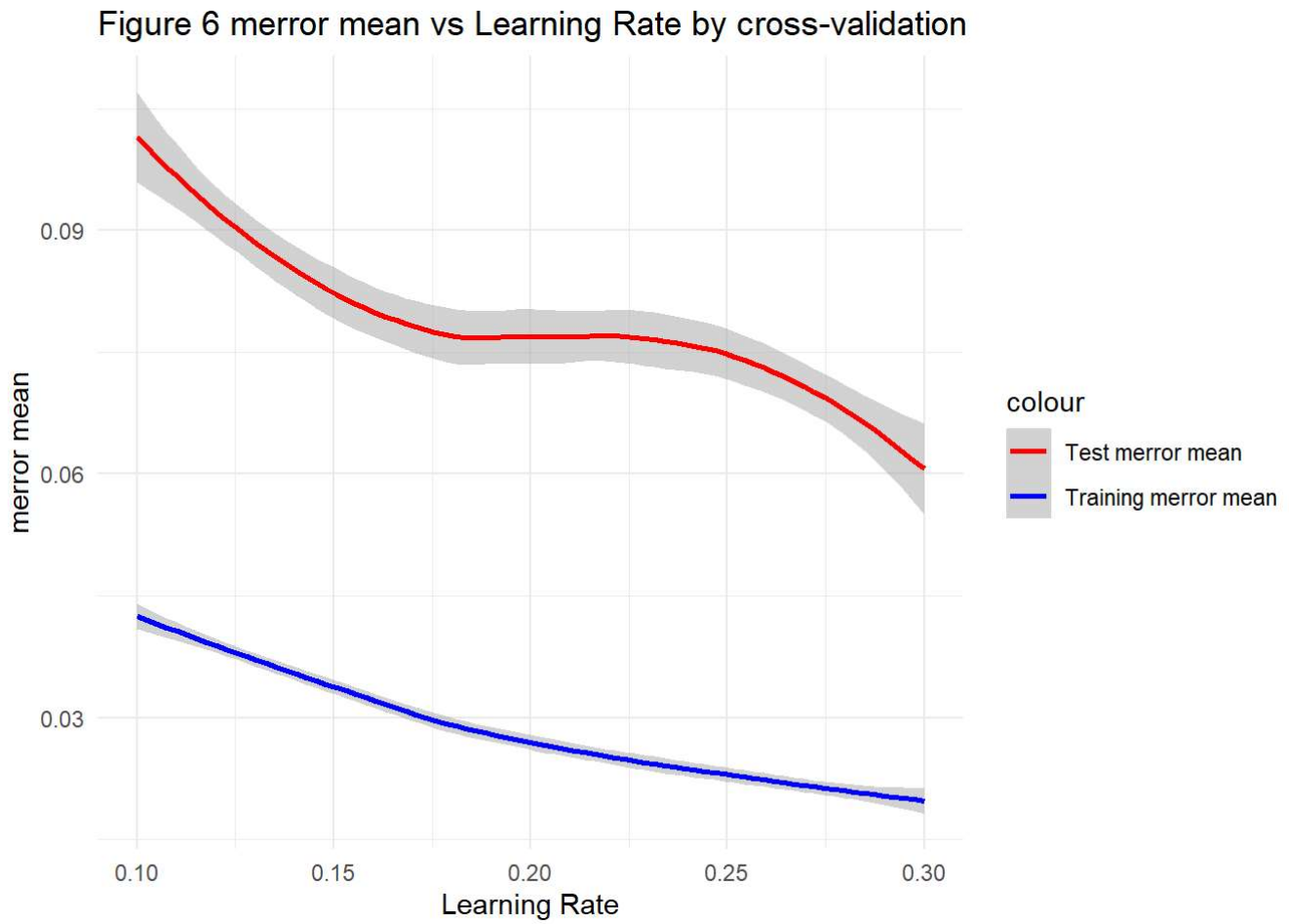
Cross-validation (CV) splits data set into several training and validation folds to help evaluate model performance. It reduces the risk of overfitting or underfitting compared to a single train-validation split. Moreover, CV is useful to help finding optimal hyper-parameters before training the final model.

Here is a diagram of 5-fold cross-validation:

Now apply CV to find the optimal learning rate.

```
# Based on Figure 4, we just need to validate several typical learning rate
learning_rates = seq(0.1, 0.3, by = 0.01)

# Storage for cross-validation results
cv_results = data.frame(learning_rate = numeric(),
                        train_merror_mean = numeric(),
                        test_merror_mean = numeric())

params = list(
  objective = "multi:softmax",
  num_class = length(unique(data_transformed$class)),
  eta = lr,
  max_depth = 6,
  subsample = 0.8,
  colsample_bytree = 0.8,
  eval_metric = "merror"
)
# Perform Cross-Validation for Each Learning Rate
set.seed(333)
for (lr in learning_rates) {
  # Update the learning rate in parameters
  params$eta = lr

  # Try 5-Fold Cross-Validation
  cv = xgb.cv(
    params = params,
    data = dtrain,
    nfold = 5,
    nrounds = 50,
    verbose = 0,
    stratified = TRUE
  )

  # Store Average Train and Validation F1-Score
  train_merror_mean = mean(cv$evaluation_log$train_merror_mean)
  test_merror_mean = mean(cv$evaluation_log$test_merror_mean)

  cv_results = rbind(cv_results, data.frame(learning_rate = lr,
                                            train_merror_mean = train_merror_mean,
                                            test_merror_mean = test_merror_mean))
}

ggplot(cv_results, aes(x = learning_rate)) +
  geom_smooth(aes(y = train_merror_mean, color = "Training merror mean")) +
  geom_smooth(aes(y = test_merror_mean, color = "Test merror mean")) +
  scale_color_manual(values = c("Training merror mean" = "blue",
                                "Test merror mean" = "red")) +
  labs(title = "Figure 6 merror mean vs Learning Rate by cross-validation",
       x = "Learning Rate", y = "merror mean") +
  theme_minimal()
```

```
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



From Figure 6, it is evident that the merror mean decreases sharply as the learning rate increases from 0.1 to around 0.25. After that, the decrease slows down, which indicates that the best learning rate should be around 0.25 to 0.3.

However, the graph is still not direct for choosing the best learning rate, because the optimal learning rate should ensure that the test error is low while keeping the training error relatively close. Therefore, I can search the minimal test_merror_mean in results and this point's learning rate can be considered as best learning rate. This test_merror_mean is the average misclassification error across all cross-validation folds, and misclassification error is a metric represents the proportion of incorrectly classified instances out of the total instances.

The formula of merror_mean is:

$$\text{Mislcassification\_Error\_Mean} = \frac{1}{K}\sum_{k=1}^{K}\frac{1}{N_k}\sum_{i=1}^{N_k}\frac{Number\ of\ Incorrect\ Predictions}{Total\ Predictions}$$

Where: - K: The total number of folds or iterations in the cross-validation process. - N_k: The number of instances (samples) in the k-th fold.

When merror_mean reach its minimal, this means the model's performance is optimal at this learning rate.

```
# Find the minimal test merror mean, this point can be the best learning rate
best_lr = cv_results[which.min(cv_results$test_merror_mean), "learning_rate"]
print(paste("Best Learning Rate:", best_lr))
```

```
## [1] "Best Learning Rate: 0.3"
```

5.2 Fit New Model

Then we can apply the best learning rate to build the adjusted model and create a test data to evaluate the performance of it.

```
# Train Final Model with Best Learning Rate
params$eta = best_lr
adjusted_model = xgb.train(
  params = params,
  data = dtrain,
  nrounds = 50
)
test_data = data_transformed[-train_index,]
# Predict on Test Data
test_preds = predict(adjusted_model, as.matrix(test_data[, -ncol(test_data)]))
test_preds = as.integer(test_preds)

# Calculate Weighted F1-Score on Test Data
test_f1 = weightedF1(test_data$class, test_preds)

original_f1 = results[7,]$Val_W_F1
cat("Test Weighted F1-Score of Adjusted Model:", test_f1,"\n")
```

```
## Test Weighted F1-Score of Adjusted Model: 0.9538619
```

```
cat("Test Weighted F1-Score of Original Model:", original_f1, "\n")
```

```
## Test Weighted F1-Score of Original Model: 0.9040919
```

From the comparison of the Test Weighted F1-Score between the adjusted model and the original model, it is obvious that the Weighted F1-Score of adjusted model is larger than that of the original model, indicating the cross-validation is useful for finding the optimal hyper-parameters for fitting a better model.

Therefore, cross-validation is still a useful method to guide the search for optimal hyper-parameters and to fit a more well-performing model.

6 Grid Search to find optimal hyper-parameters

6.1 Introduce Grid Search

After finding an optimal learning rate, the adjusted model's Weighted F1-Score is much higher than the origianl model. However, I did not consider the correlation between hyper-parameters, and this "optimal" learning rate may not be optimal when changing other hyper-parameters. Therefore, to enhance the model further, I need to find a set of hyper-parameters that can make the model perform better than previous. In this part, I will use Grid Search.

Grid Search is a method used for hyper-parameter optimization in machine learning. (Bergstra et al. 2011) It uses exhaustively searching through a predefined set of hyper-parameters to find a combination that produces the best performance for a model.

To apply grid search, firstly, I need to select a set of hyper-parameters of the model I want to tune, and for each hyper-parameter, I need to provide a list of possible values. Then, generating all possible combinations of the hyperparameter values as a grid. The last step is training and evaluating the model using cross-validation to

find the best combination.

```
# Grid search for hyper-parameter tuning

# Prepare possible hyper-parameters sequence for grid search
tune_grid = expand.grid(
  # The number of boosting rounds
  nrounds = c(50, 100, 150),

  # The maximum depth of each decision tree
  max_depth = c(4, 6, 8),

  # The learning rate
  eta = c(0.01, 0.1, best_lr),

  # Regularization term
  gamma = c(0, 0.1, 0.2),

  # The fraction of features to sample randomly for each tree.
  colsample_bytree = c(0.8, 1.0),

  # The minimum sum of instance weights (hessian) required in a child node
  min_child_weight = c(1, 3),

  # The fraction of training data to sample randomly for building each tree.
  subsample = c(0.8, 1.0)
)

# Function to evaluate a set of parameters
eval_params = function(nrounds, max_depth, eta, gamma,
                       colsample_bytree, min_child_weight, subsample) {
  # Set hyper-parameters
  params = list(
    objective = "multi:softmax",
    num_class = length(unique(data_transformed$class)),
    max_depth = max_depth,
    eta = eta,
    gamma = gamma,
    colsample_bytree = colsample_bytree,
    min_child_weight = min_child_weight,
    subsample = subsample
  )

  # Use cross-validation to find the optimal_params
  cv = xgb.cv(
    params = params,
    data = dtrain,
    nrounds = nrounds,
    nfold = 5,
    early_stopping_rounds = 10,
    metrics = "merror",
    verbose = FALSE
  )

  return(min(cv$evaluation_log$test_merror_mean))
}
```

```
# Perform grid search
results_gs = apply(tune_grid, 1, function(x) {
  error = eval_params(
    nrounds = x["nrounds"],
    max_depth = x["max_depth"],
    eta = x["eta"],
    gamma = x["gamma"],
    colsample_bytree = x["colsample_bytree"],
    min_child_weight = x["min_child_weight"],
    subsample = x["subsample"]
  )
  return(error)
})

# Find best parameters
optimal_params_idx = which.min(results_gs)
optimal_params = tune_grid[optimal_params_idx,]
cat("Optimal hyper-parameters:\n")
```

```
## Optimal hyper-parameters:
```

```
print(optimal_params)
```

```
##     nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 453     150         4 0.3   0.1                1                1         1
```

By applying grid search, I have found the overall optimal hyper-parameters for fitting this model. Then, I will use the optimal hyper-parameters to fit a final model.

```
# Train final model with best parameters
final_params = list(
  objective = "multi:softmax",
  num_class = length(unique(data_transformed$class)),
  max_depth = optimal_params$max_depth,
  eta = optimal_params$eta,
  gamma = optimal_params$gamma,
  colsample_bytree = optimal_params$colsample_bytree,
  min_child_weight = optimal_params$min_child_weight,
  subsample = optimal_params$subsample
)

# Use best params to fit a final model
final_model = xgb.train(
  params = final_params,
  data = dtrain,
  nrounds = optimal_params$nrounds,
  watchlist = list(train = dtrain, test = dvalidation),
  verbose = 0
)

# Final predictions with best model
final_pred_test = predict(final_model, dvalidation)
# Calculate Weighted F1-Score
final_W_F1 = weightedF1(y_validation, final_pred_test)
cat("Final model test Weighted F1-Score:", round(final_W_F1, 4), "\n")
```

```
## Final model test Weighted F1-Score: 0.9912
```

```
cat("Adjusted model test Weighted F1-Score:",test_f1,"\n")
```

```
## Adjusted model test Weighted F1-Score: 0.9538619
```

```
cat("Test Weighted F1-Score of Original Model:",original_f1, "\n")
```

```
## Test Weighted F1-Score of Original Model: 0.9040919
```

From the final model's Weighted F1-Score, there is a huge improvement compared with the adjusted model, indicating the grid search has large positive influence on model fitting. Moreover, grid search is simple to set up and understand so the interpretability is very strong.

However, grid search is time-consuming, which means the computation cost will be extremely high as the number of hyper-parameters and their possible values increase.

6.2 Multicalss Log Loss for evaluation

For further evaluation of new model performance, I will use multiclass log loss.

Multiclass Log Loss is a metric used to evaluate the quality of predictions from a classification model. It measures the difference between the predicted probability distribution and the true class labels, providing a numerical assessment of how well the model's prediction align with the actual outcomes.

For a dataset with N samples and K classes, the Multiclass Log Loss is defined as:

$$\text{Multiclass Log Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{i,k}\log(p_{i,k}),$$

where: - $y_{i,k}$ is an indicator variable (1 if sample i belongs to class k, 0 otherwise). - $p_{i,k}$ is the predicted probability for sample i belonging to class k. - N is the total number of samples.

Here, I will draw a multicalss log loss graph for training data and testing data.

```
# Create visualization of learning curves
# Get training Multiclass Log Loss
train_logloss = data.frame(
  Iteration = 1:length(final_model$evaluation_log$train_mlogloss),
  Logloss = final_model$evaluation_log$train_mlogloss,
  Dataset = "Training Multiclass Log Loss Vs Iteration"
)

# Get testing Multiclass Log Loss
test_logloss = data.frame(
  Iteration = 1:length(final_model$evaluation_log$test_mlogloss),
  Logloss = final_model$evaluation_log$test_mlogloss,
  Dataset = "Testing Multiclass Log Loss"
)

# Create logloss_data for plot
logloss_data = rbind(train_logloss, test_logloss)

# Plot learning curves
library(ggplot2)
ggplot(logloss_data, aes(x = Iteration, y = Logloss, color = Dataset)) +
  geom_line() +
  theme_minimal() +
  labs(title = "Figure 8 Training and Testing Multiclass Log Loss Vs Iteration",
       x = "Iteration",
       y = "Multiclass Log Loss")
```

Figure 8 illustrates the Multiclass Log Loss for both the training and testing datasets across iterations during model training. Initially, both the training and testing log-loss values are extremely high, indicating poor predictions.

As the training process (iteration grows up), log-loss decreases sharply, showing that the model's performance is getting better. The training log-loss stays below the testing log-loss is normal, because the model perform better on training data obviously.

Moreover, when iteration grows to over 75, the model is approaching its optimal performance, because both training and testing curves converge. Meanwhile, the gap between Training and Testing multiclass log loss remain small, and this demonstrates good generalization to the testing data.

In conclusion, by applying grid search, I fit a new model that achieves higher Test Weighted F1-Score than the original model, which means the new model can perform better when classifying unseen data.

Reference

Arlot, Sylvain, and Alain Celisse. 2010. "A Survey of Cross-Validation Procedures for Model Selection." *Statistics Surveys* 4: 40–79. https://doi.org/10.1214/09-SS054 (https://doi.org/10.1214/09-SS054).

Bergstra, James, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. "Algorithms for Hyper-Parameter Optimization." In *Advances in Neural Information Processing Systems*, edited by J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger. Vol. 24. https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf (https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf).

Bohanec, Marko. 1988. "Car Evaluation." UCI Machine Learning Repository.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–94. ACM.

Hinojosa Lee, Maria Cristina, Johan Braet, and Johan Springael. 2024. “Performance Metrics for Multilabel Emotion Classification: Comparing Micro, Macro, and Weighted F1-Scores.” *Applied Sciences* 14 (21). https://doi.org/10.3390/app14219863 (https://doi.org/10.3390/app14219863).