Understanding Hyperparameters: The Key to Optimizing Machine Learning Models

Olawale Awe, AIMM 2024- IMECC, UNICAMP, Campinas, Brazil

A **hyperparameter** is a configuration or setting in a machine learning algorithm that is set before the learning process begins and cannot be learned from the data itself.

Hyperparameters control the overall behavior of the model and can influence how the model learns and performs.

Unlike parameters (which are learned from the training data during model fitting, such as the weights in a neural network), hyperparameters are **external configurations** that you must define **before the training process starts**.

Examples of Hyperparameters:

1. **Learning Rate** (in neural networks or gradient-based algorithms): Controls how much the model's parameters are adjusted with respect to the loss gradient.

2. **Number of Trees** (in Random Forests or Gradient Boosting): Determines how many decision trees are built in the ensemble.

3. **Depth of Tree** (in decision trees): Specifies the maximum depth of each decision tree, controlling how complex the tree can be.

4. **Number of Neighbors** (k) (in k-NN): Specifies how many neighbors to consider when classifying a new data point.

5. **Regularization Parameter** (in logistic regression or SVM): Controls the trade-off between fitting the training data and keeping the model as simple as possible (avoiding overfitting).

6. **Batch Size** (in neural networks): Determines the number of training samples used in one iteration of model training.

Why Are Hyperparameters Important?
- **Model Performance**: The choice of hyperparameters significantly affects the model's performance. Proper tuning can lead to better accuracy, precision, and generalization to new data.

- **Overfitting vs. Underfitting:** Hyperparameters like regularization and tree depth help balance the model between overfitting (where the model is too complex and fits the noise in the training data) and underfitting (where the model is too simple and fails to capture the underlying patterns).

How Are Hyperparameters Chosen?
- Grid Search: Systematically trying out a range of values for hyperparameters to find the best combination.

- **Random Search**: Randomly selecting combinations of hyperparameters from predefined ranges.

- **Bayesian Optimization**: A more sophisticated method that models the relationship between hyperparameters and model performance and selects hyperparameters based on this model.

Hyperparameter tuning is a critical part of building an effective machine learning model, as the right set of hyperparameters can significantly enhance model performance.

Key Points:
- `C` and `sigma` in SVM control the trade-off between margin maximization and error minimization, and the kernel width, respectively.
- `mtry` in Random Forest is the number of variables randomly sampled as candidates at each split.
- `k` in k-NN is the number of neighbors to consider.
- `size` in Neural Networks controls the number of neurons in the hidden layer, and `decay` is the regularization parameter.
- `fL`, `usekernel`, and `adjust` in Naive Bayes control Laplace correction, kernel density estimation, and bandwidth adjustment, respectively.
- `nIter` in Boosting is the number of boosting iterations.

These setups allow you to systematically search through a range of plausible values for hyperparameters, enabling the selection of the optimal model configuration.

Practical Hypeparameter Tuning of our Selected Models

Here's a breakdown of hyperparameter tuning setups for each of the specified models using the `caret` package in R:

1. **Support Vector Machine (SVM)**

```
tuneGrid_svm <- expand.grid(C = c(0.1, 1, 10, 100),
                            sigma = c(0.01, 0.1, 1))
```

2. **Random Forest**

```
tuneGrid_rf <- expand.grid(mtry = c(2, 4, 6, 8, 10))
```

3. **Naive Bayes**

```
tuneGrid_nb <- expand.grid(fL = c(0, 0.5, 1),
                           usekernel = c(TRUE, FALSE),
                           adjust = c(0, 0.5, 1))
```

4. **Logistic Regression (LR)**

Logistic regression **typically doesn't require extensive hyperparameter** tuning unless you are using regularization techniques (like Ridge or Lasso). For a regular logistic regression, you would focus on data preprocessing and feature selection rather than hyperparameter tuning.

For Ridge Logistic Regression

```
tuneGrid_lr <- expand.grid(lambda = seq(0, 1, length = 10))
```

5**. k-Nearest Neighbors (k-NN)**

```
tuneGrid_knn <- expand.grid(k = seq(1, 21, by = 2))
```

6. **Linear Discriminant Analysis (LDA)**

LDA has few hyperparameters, so tuning is not as common. The main focus is usually on data preprocessing. No typical tuning grid needed; LDA is parameter-free in terms of model complexity.

7**. Neural Networks (nnet)**

```
tuneGrid_nnet <- expand.grid(size = c(1, 3, 5, 7),
                             decay = c(0, 0.01, 0.1))
```

8**. Learning Vector Quantization (LVQ)**

```
tuneGrid_lvq <- expand.grid(size = c(5, 10, 15),
                            k = c(1, 3, 5),
                            learning_rate = c(0.01, 0.05, 0.1))
```

9. **Bagging (using `treebag` method)**

```
tuneGrid_bagging <- expand.grid(.cp = seq(0, 0.1, length = 10))
```

10. Boosting (AdaBoost using `adaboost` method)

```
tuneGrid_boosting <- expand.grid(nIter = c(50, 100, 150),
                                 method = "Adaboost.M1")
```

Example Setup in `caret`

Here’s an example of how you can use these tuning grids in the `caret` package:

```
set.seed(123)
control <- trainControl(method = "cv", number = 10)
```

Example for SVM

```
svmModel <- train(factor(severe_maleria) ~ ., data = train,
                  method = "svmRadial",
                  trControl = control,
                  tuneGrid = tuneGrid_svm)
```

Example for Random Forest

```
rfModel <- train(factor(severe_maleria) ~ ., data = train,
                 method = "rf",
                 trControl = control, sampling= ‘up’,
                 tuneGrid = tuneGrid_rf)
```

Evaluate the results

```
print(svmModel)
plot(svmModel)

print(rfModel)
plot(rfModel)
```

You can go ahead to evaluate the models as usual.

Hyperparameter Tuning tor Decision Trees

Let us demonstrate a complete setup for training and tuning a Decision Tree model using the `caret` package in R.

This includes data preparation, model training, hyperparameter tuning, and evaluation.

Step 1: Install and Load Necessary Packages

Make sure you have the necessary packages installed and loaded.

Install caret if you haven't already

```
install.packages("caret")
```

Load the caret package

```
library(caret)
```

Step 2: Prepare the Data

For demonstration purposes, we'll use the Malaria dataset, which is included in R.

Split the data into training and testing sets

```
set.seed(123)  # For reproducibility
trainIndex <- createDataPartition(mdata$severe_maleria, p = .75,
                                  list = FALSE,
                                  times = 1)
train <- mdata[trainIndex,]
test  <- mdata[-trainIndex,]
```

S**tep 3: Define the Model and Hyperparameter Grid**

For Decision Trees, common hyperparameters include `cp` (complexity parameter), `maxdepth` (maximum depth of the tree), and `minsplit` (minimum number of observations that must exist in a node for a split to be attempted).

```
#  Define the hyperparameter grid
tuneGrid_dt <- expand.grid(cp = c(0.001, 0.01, 0.05, 0.1),
                           maxdepth = c(5, 10, 15, 20),
                           minsplit = c(10, 20, 30, 40))
```

Step 4: **Train the Model with Cross-Validation**
Use cross-validation to evaluate different combinations of hyperparameters.

```
# Set up cross-validation
control <- trainControl(method = "cv", number = 10)

 Train the Decision Tree model
set.seed(123)
dtModel <- train(Species ~ ., data = train,
                 method = "rpart",
                 trControl = control,
                 tuneGrid = tuneGrid_dt)
```

Step 5: Evaluate the Model
Once the model is trained, you can evaluate the results and make predictions on the test set.

```
# Print the results of the trained model
print(dtModel)

# Plot the results to visualize the performance across different hyperparameter values
plot(dtModel)

# Make predictions on the test set
predictions <- predict(dtModel, newdata = test)

# Evaluate the performance using a confusion matrix
confMatrix <- confusionMatrix(predictions, testt$severe_maleria)
print(confMatrix
```

Step 6: Visualize the Decision Tree

You can visualize the final decision tree to understand how it makes decisions.

```
# Load the rpart.plot package for tree visualization
install.packages("rpart.plot")
library(rpart.plot)

 #Plot the decision tree
rpart.plot(dtModel$finalModel)
```

Summary of the Steps:
1. Data Preparation: Split the data into training and testing sets.
2. Model Setup: Define the Decision Tree model and the hyperparameters to tune.
3. Cross-Validation: Use 10-fold cross-validation to train and evaluate the model.
4. Evaluation: Assess model performance using a confusion matrix.
5. Visualization: Plot the final decision tree for interpretation.

Notes:
- `cp` (Complexity Parameter): Prunes the tree by penalizing the complexity of the model. A smaller `cp` allows for a more complex tree.
- `rpart` Method: In `caret`, the `rpart` method is used to fit Decision Trees.
- `tuneGrid`: This grid specifies the range of `cp` values to explore during the tuning process.

This setup provides a comprehensive approach to building and evaluating a Decision Tree model using the `caret` package in R.