Hyperparameter Tuning Strategies and Applications to Random Forests

Tony Ni *
Department of Mathematics and Statistics, Amherst College

December 8, 2020

Abstract

The performance of machine learning models heavily depend upon the correct specification of certain variables, known as hyperparameters, during their creation. How to best pick these hyperparameters in order to create the most optimum model is quite difficult due to the trial and error approach of creating ML models being quite time-consuming. As a meta-learning problem, several popular approaches have emerged to aid in the creation of optimal machine learning models those of which are further discussed in this report include: grid search, random search, and sequential model-based optimization methods. We also perform some application studies to random forest classification problems with certain datasets in order to compare and contrast their effectiveness and runtimes, alongside performing a case-study on language learning abilities.

Keywords: machine learning, classification, meta learning, model parameter, hyperparameter

*Thank you to Nick Horton for his help and guidance in the creation of this project.

1 Introduction

Machine Learning is a rapidly evolving subfield of artificial intelligence widely utilized by practitioners in fields ranging from health care, sciences, government. Its ubiquity is widely due to its practicality, especially in an era where data is more accessible and prevalent than ever. As its name suggests, many of the techniques in machine learning have to do with the training of a computer (machine) to learn from data and use it to execute a program to solve a problem at hand.

These problems can range from simple tasks such as trying to generate a "rule" to differentiate between groups such as apples and oranges (classification task), trying to find unknown clusters in a set of data (clustering), etc. It would be difficult to go into the details and specifics within the types of learning that machine learning encompasses, as there are far too many machine learning tasks to cover in a single report – but the most important takeaway would have to be the ability of machine learning algorithms to develop, learn, and train from data to create models.

While in the process of creating these models, it can often be challenging to find the model which best captures the information from the training data. Depending on what sort of machine learning task is in question (classification, clustering, reinforcement learning, etc.) each model takes in unique parameters known as 'hyperparameters,' which govern the training process itself.

For example, if we were looking at clustering as a technique (specifically K-Means clustering) – some hyperparameters to alter would be things such as: the number of clusters we want, the initial configuration of centroids, the max number of iterations we want the algorithm to run, etc. (Picco et al. 2016). Altering these hyperparameters (also known as model-specific properties) can often change how the model fits and adheres to the training data. Thus, it is important to choose hyperparameters that allows for the model to best adapt to the training data without underfitting or overfitting to the data.

How do we pick the correct configuration of parameters to use then? The process of creating these models itself is a sub-field known as "meta-learning," and many techniques have been founded and developed to tackle the challenges of finding the best model for a given task. One of the most notable methods of meta-learning tasks is known as "hyper-

parameter tuning," which will be the focus of this report. For those interested in a more in-depth exploration of other topics in meta learning, please refer to: (Hutter et al. 2014), (Vanschoren 2018), and (Reif et al. 2012).

Hyperparameter tuning is essentialy an optimization task, in which we are trying to look for the best combinations of hyperparameters to use to train our model. Methods such as grid search, random search, and Bayesian searches have come up in recent times which aid machine scientists to obtain the set of hyperparameters which builds the best performing model.

2 Question

The focus of my project specifically encompasses hyperparameter tuning and optimization with regards to ensemble-learning algorithms – specifically classification with random forests, and studying the effectiveness of three different hyperparameter tuning techniques: grid search, random search, and Bayesian hyperparameter optimization – with regards to large data.

3 Background

3.1 Model Parameters and Hyperparameters

An important conceptualization of a model parameter and hyperparameter is essential to understanding the underpinnings of model tuning and meta-learning. These are terms which are often confused for one another in machine learning. A *model parameter*, in terms of machine learning algorithms, is a variable whose value is estimated from the dataset, *not* by the user. For example, the coefficients/effects of the explanatory variables in a linear regression model are known as model parameters. Some other model parameters in machine learning include weights and biases which are important to the functioning of neural networks (Yang & Shami 2020).

A hyperparameter on the other hand is a variable whose value is set by the user before the model training begins. These values can freely be adjusted by the user in order to

obtain different models, and as such, it is important to set model parameters correctly in order to obtain models with optimal performance. An example of a model hyperparameter includes variables such as k (the number of clusters) in a clustering algorithm.

Misspecification of model hyperparameters are detrimental to machine learning models' performance. If we think about clustering as a technique we wish to run in order to differentiate between different types of fruits (say, apples and oranges), it is important to specify a logical number of clusters to find. Assuming that the dataset only has information on the most common fruits in the U.S., values between 10 or 20 seem to be reasonable. We don't really expect to find more than 20 different fruits in our dataset. Some unreasonable values one might set the number of clusters to find, k, such as 1000 would be unreasonable due to it not making sense in the context of the situation. If this were the case, the model clearly would not be able to find the correct clusters in the data due to misspecification of the number of clusters k.

However, how would the user know what values are the most optimal to pick in order to achieve the most optimum level? This is where hyperparameter tuning can assist us.

3.2 Hyperparameter Tuning

Hyperparameter tuning works in a variety of ways but it generally involves running an algorithm in order to find the combination of optimum combinations of hyperparameters and the methods to do have are varied and differ in their approaches. The core of all tuning algorithms involves running multiple trials with a variety of combinations of hyperparameters and identifying which combination minimizes (or maximizes) the loss function metric the user specifies (accuracy, misclassification rate, etc.).

To get a better sense of things (i.e, why hyperparameter tuning is important/helpful), the following section will discuss how hyperparameter tuning with regards to classification trees. In classification, our goal is of course, to create a model which is able to predict which class a certain observation belongs to. An example classification task will be performed with the `palmerpenguins` dataset which contains observations regarding body measurements and other supplementary information for 344 penguins collected from the Palmer Archipelago. The classification task at hand will be to create a classification tree

model which is able to predict the `species` from the following "Adelie," "Gentoo," and "Chinstrap" penguins which comprise the dataset.

The difficulty in creating classification tree models which are meaningful lies in the splitting and pruning the trees itself. First and foremost, trees generally perform splits on variables in order to differentiate between classes. For example in our `palmerpenguins` dataset, we may want to create a split on the variable `bill_length_mm` and claim that a `bill_length_mm` of < 40 usually means that the penguin is an "Adelie" penguin, and those with `bill_length_mm` $>= 40$ usually are "Gentoo" and "Chinstrap" penguins. In general, splits in which the groups are mostly pure are the most ideal – if a node is 100% pure, it means that all observations in that node belong to the same class.

In creating a classification tree model, it is possible to specify `minsplit`, the minimum number of observations which must exist in a node in order for a split to be attempted, `minbucket`, the minimum number of observations in any terminal/leaf node, etc. These are the hyperparameters for creating a decision tree.

We will create and feature two unique decision tree models for the `palmerpenguins` data set using distinct combinations for `minsplit` and `minbucket` to show how the model output can vary with a slight tweak to these values.

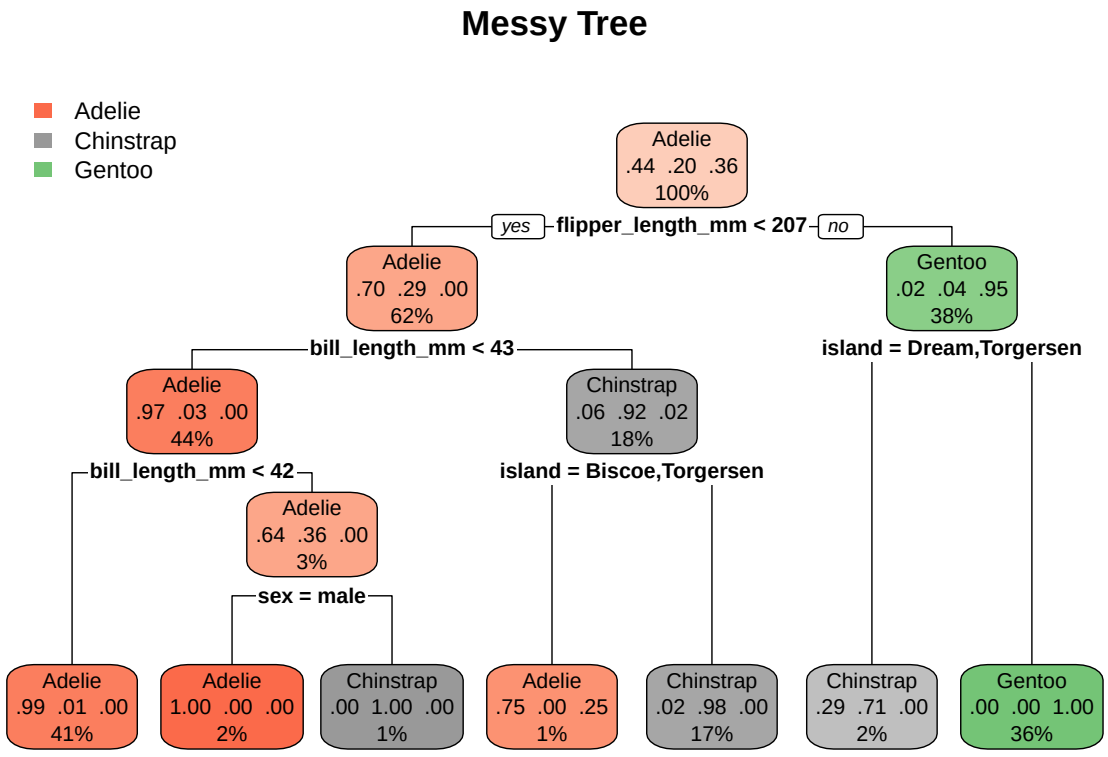

Figure 1: An classification tree created using the hyperparameters of `minsplit` = 2 and `minbucket` = 2

The first classification tree shown in figure 1 was created used a `minsplit` = 10 and `minbucket` = 2. The following rules set by these hyperparameter dictates the following: while creating the tree, the algorithm only splits a node if it has more than 10 observations in it, and if a split would create a new node with less than 2 observations. Due to the `minsplit` value being rather small, the algorithm will be quite liberal with splitting nodes (it keeps splitting nodes if they have more than 10 observations), and also is quite generous with what sort of nodes are being created (nodes of size 2 or larger are deemed fine to make). This is obviously quite troublesome as the values that we set for these hyperparameters yield which is rather descriptive and difficult to interpret as it seems to overfit the data tremendously.

To give a expository explanation as to how to interpret this classification tree for those

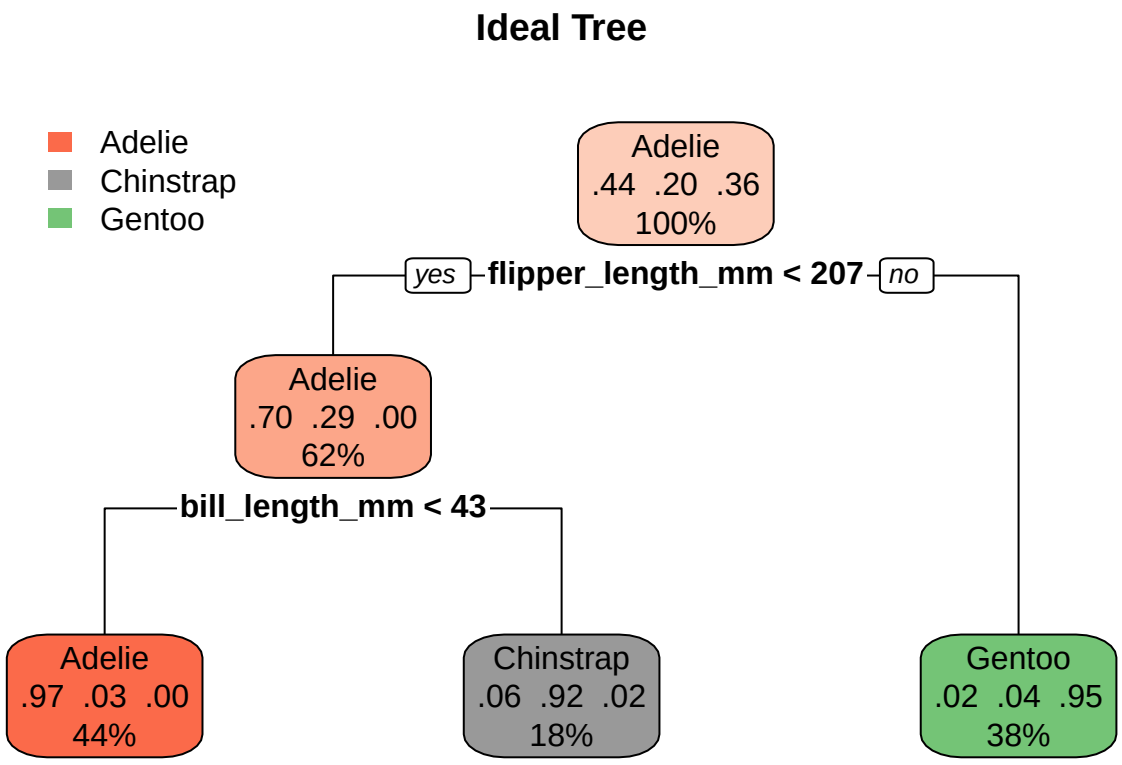

Figure 2: An classification tree created using the hyperparameters of `minsplit` = 30 and `minbucket` = 15

who may be unfamiliar with it: starting from the the root (top-most) node, once traverses down the down while asking the question at each split, and then going left or right depending on the answer. For example, "Is `flipper_length_mm` < 207?" If yes, we travel down the left of the split and proceed to the next question. The numbers inside each node specify the proportion of each type of penguin which fell into this node depending on the specifications of the model. For example, we can see that 70% of Adelie penguins, 29% of Chinstrap penguins, and 0% of Gentoo penguins have a `flipper_length_mm` < 207.

Altering the hyperparameters to have `minsplit` = 30, and `minbucket` = 15, to obtain a new tree then, will yield:

In the tree shown in 2tree, more reasonable values of `minsplit` and `minbucket` are chosen, in which a split is considered if a node has observations ≥ 30, and if a split would

make a node that has less than 15 observations, it won't split the node. The second set of hyperparameters yields a tree which isn't as overfitted as seen in the first tree, as a result of picking good hyperparameter values.

Thus from this in-depth study on classification trees and the hyperparameters which dictate the algorithm's behavior in creating a decision tree model, it can be seen that picking good hyperparamter values is essential – not just for decision tree model making, but all types of machine learning models also. It can be quit troublesome to determine what sorts of hyperparameter values to pick, and that fact compounded by the myriad of machine learning techniques out there makes it quite bothersome to have to go through multiple combinations of hyperparameter values in a model just to see which one is the most optimum, which is why hyperparameter optimization/tuning is essential.

A more in-depth discussion on how the hyperparameter methods perform and work will be discussed in the methods section 4.

4 Methods

Grid search (which will be implemented with the `caret` package in R) is a brute-force/exhaustive algorithm where the user specifies a list of values for hyperparameters and the computer evaluates the performance of each combination, which allows us to observe the most optimum combination of values. Although this method guarantees that the most optimal configuration of hyperparameters will be found at the search's conclusion, the issue lies in the runtime. As a brute force approach whose runtime is dependent on the number of hyperparameters given, we can expect the time-complexity of this approach to be quite abysmal. Grid search has been found to be reliable in low dimensional spaces however, and is simple to implement in cases when the dimensionality of a dataset is low (Bergstra & Bengio 2012).

The idea of random search is quite similar to grid search, but instead of evaluating each combination of hyperparameters like grid search does, random search evaluates random combinations of a range hyperparameter values until a criteria is met (like the maximum number of iterations being reached). Random search will also be implemented using the `caret` package in R. While random search sacrifices the comprehensibility (looking every possible combination of hyperparameters) of grid search, its runtime is far more feasible. Where it falls short however is that due to the combinations being selected by random chance, it can often yield high variance during computing. Everything is being left to luck, which is quite risky (Bergstra & Bengio 2012).

The most recent advances in hyperparameter tuning is Bayesian hyperparameter optimization which as its name suggests, utilizes Bayesian logic in order to select the most promising hyperparameters. The specific type of Bayesian hyperparameter optimziation technique we will be observing is the Sequential Model-based Optimization (with the `tuneranger` package in R) which creates and evaluates models based on previous iterations and chooses future combinations while considering for these results. Essentially, with the line of Bayesian reasoning, with more runs and combinations – the model will eventually become "less wrong," and will settle on the best combination of hyperparameters in fewer iterations that both grid search and random search (Probst et al. 2019).

Sequential Model-Based Optimization essentially consists of the following steps:

1) Specify an evaluation metric to use (RMSE, accuracy), evaluation strategy to test the model on (bagging, CV), and hyperparameters you wish to use in creating the model in question and create your initial model.

2) Based on the output from step 1, fit a regression model (also known as a surrogate model) on all previously tested hyperparameters with the y-axis as the evaluation measure and hyperparameters as the x-axis

3) Based on the regression model in step 2, choose a new combination of hyperparameters with good performance metrics to create an updated model, evaluate the updated model, and then add it to the existing hyperparameter space.

4) Repeat steps 2-3 until a desired model is reached (Probst et al. 2019).

The focus on the differences between these methods thus lie in their runtimes. Each method will be able to find a combination of hyperparameters which yield the most optimum model in accordance with their principles, but the problem lies in how quickly these combinations can be obtained alongside their ease of implementation and effectiveness (Probst et al. 2019)

5 Example: Cells

To begin our investigation on the implementations of the tuning algorithms, we will introduce `cells` a dataset which is featured in the package `modeldata`, which contains information on cell segmentation imaging – with observations belonging to two classes `WS`, well-segmented and `PS` poorly-segmented.

Each of the three aforementioned techniques will be implemented with regards to creating a random forest classification model of the `cells` dataset. A careful consideration of the runtimes and effectiveness of each of the three methods will further be discussed as well. In each implementation of the methods discussed, the length it will take for the tuning algorithm to finish will be measured, to give a sense of the feasbility of implementation of each respective method towards a large-scale application to big data.

First, we will be implementing grid search as a means to find optimal hyperparameters for our random forest.

```
## Random Forest
##
## 1414 samples
##   56 predictor
##    2 classes: 'PS', 'WS'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 1273, 1272, 1272, 1273, 1273, 1272, ...
## Resampling results across tuning parameters:
##
##   mtry  min.node.size  splitrule   Accuracy   Kappa
##   2     1              gini        0.8208138  0.6057310
##   2     1              extratrees  0.8184497  0.5989964
##   2     2              gini        0.8243499  0.6139725
##   2     2              extratrees  0.8160840  0.5930653
##   2     3              gini        0.8234009  0.6121283
```

```
##   2     3              extratrees  0.8172727  0.5944846
##   2     4              gini        0.8219908  0.6082599
##   2     4              extratrees  0.8210568  0.6034409
##   2     5              gini        0.8212799  0.6067697
##   2     5              extratrees  0.8151483  0.5898296
##   3     1              gini        0.8245713  0.6149485
##   3     1              extratrees  0.8234026  0.6105288
##   3     2              gini        0.8243449  0.6147515
##   3     2              extratrees  0.8207988  0.6055935
##   3     3              gini        0.8248127  0.6159866
##   3     3              extratrees  0.8231662  0.6093750
##   3     4              gini        0.8236373  0.6121643
##   3     4              extratrees  0.8252905  0.6149139
##   3     5              gini        0.8236423  0.6131679
##   3     5              extratrees  0.8184480  0.5993792
##   4     1              gini        0.8257550  0.6186003
##   4     1              extratrees  0.8219842  0.6078107
##   4     2              gini        0.8241052  0.6147956
##   4     2              extratrees  0.8233909  0.6120347
##   4     3              gini        0.8267040  0.6206822
##   4     3              extratrees  0.8219825  0.6075990
##   4     4              gini        0.8238737  0.6138257
##   4     4              extratrees  0.8257750  0.6166705
##   4     5              gini        0.8241035  0.6147879
##   4     5              extratrees  0.8236457  0.6111562
##   5     1              gini        0.8262311  0.6201798
##   5     1              extratrees  0.8259964  0.6181764
##   5     2              gini        0.8262311  0.6194586
##   5     2              extratrees  0.8255269  0.6171010
##   5     3              gini        0.8292878  0.6264271
```

```
##   5     3             extratrees  0.8271651  0.6205325
##   5     4             gini        0.8245763  0.6167361
##   5     4             extratrees  0.8259897  0.6181006
##   5     5             gini        0.8245763  0.6167174
##   5     5             extratrees  0.8243449  0.6138758
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 5, splitrule = gini
##  and min.node.size = 3.

## Time difference of 2.995551 mins
```

Our grid search tuning algorithm took around 3 minutes to run, and it suggested the following hyperparameters to use in order to achieve the most optimal model (with accuracy and kappa metrics: mtry = 5, splitrule = extratrees and min.node.size = 3. Although it is known that the algorithm went over every possible combination of hyperparameters, and thus, this is indeed most likely the best one, the amount of time this method took is a bit concerning. In this example of grid search, the tuning algorithm was offered 4 possible values for `mtry`, 5 for `min.node.size`, and "gini" for `splitrule`, leading to a total of 4 $*$ 5 $*$ 2 = 40 different models to create. The more hyperparameters one has to tune, the higher the runtime it will take which is an important consideration to take into account.

Creating a our first model using the hyperparamters suggested by grid search:

```
set.seed(7271999)

#grid search random forest
rg.cells.grid <- ranger(class ~ ., data = cells_training,
                        mtry = 5, min.node.size = 3,
                        splitrule = "gini")

pred.cells.grid <- predict(rg.cells.grid, data = cells_testing)
table(cells_testing$class, pred.cells.grid$predictions)
```

```
##
##       PS  WS
##   PS 344  46
##   WS  55 160
```

```
(346+151)/(326+44+64+151)
```

```
## [1] 0.8495726
```

The model using grid search suggested hyperparameters, yields an accuracy of 0.849. Next, we implement random search:

```
## Random Forest
##
## 1414 samples
##   56 predictor
##    2 classes: 'PS', 'WS'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 1273, 1272, 1272, 1273, 1273, 1272, ...
## Resampling results across tuning parameters:
##
##   min.node.size  mtry  splitrule   Accuracy   Kappa
##    2             52    extratrees  0.8255203  0.6220294
##    3             13    gini        0.8290564  0.6282881
##   14             51    extratrees  0.8278760  0.6258603
##   15             28    extratrees  0.8248077  0.6182747
##   17             29    gini        0.8281157  0.6277820
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 13, splitrule = gini
##  and min.node.size = 3.
```

```
## Time difference of 1.29313 mins
```

Our random search tuning algorithm took around 1.3 minutes to run, which is a marked improvement over the grid search implementation. The output from this code suggested to pick the following values to obtain the highest accuracy: mtry = 28, splitrule = extratrees and min.node.size = 15. This method uses a small search space which is why it runs much faster than the previous grid search. However, it is important to note that random search yields high variance during computing due to its 'random' nature, and as such it may not also find the most optimal configuration of hyperparameters as it leaves it up to chance.

Next, we use the hyperparameters suggested by our random search algorithm to create a model:

```
set.seed(7271999)

#random search random forest
rg.cells.rand <- ranger(class ~ ., data = cells_training,
                        mtry = 28, min.node.size = 15,
                        splitrule = "extratrees")

pred.cells.rand <- predict(rg.cells.rand, data = cells_testing)
table(cells_testing$class, pred.cells.rand$predictions)
```

```
##
##        PS  WS
##   PS 341  49
##   WS  52 163
```

```
(348+152)/(348+42+63+152)
```

```
## [1] 0.8264463
```

The model using random search suggested hyperparameters, yields an accuracy of 0.826, only slightly less than our initial grid search model.

Finally, our implementation of SMBO tuning:

```
## Time difference of 1.491638 mins
```

`tuneRanger` is a specialized package in R useful for automatic tuning of random forests. The tuning strategy used in `tuneRanger` is of a sequential model-based optimization which essential iterates between fitted models and uses past model fits as knowledge in the creation of new ones – akin to Bayesian methods in statistics. This run took around 1.5 minutes to run which is still a marked improvement over grid search and takes only slightly longer than random search. The output obtained from running SMBO is omitted due to its excessive length, but the code can be found in section 8. Our SMBO algorithm suggests the usage of `mtry` = 10, `min.node.size` = 6, and `sample.fraction` = 0.8423.

From the tuning algorithms, it can clearly be seen that even with the rather small size of the `cells` dataset – the hyperparameter tuning algorithms took a slightly long time to run with a time of 1.88 minutes.

Using the hyperparameters suggested by our SMBO algorithm to create a model:

```
set.seed(7271999)

#SMBO random forest
rg.cells.smbo <- ranger(class ~ ., data = cells_training,
                        mtry = 10, min.node.size = 6,
                        sample.fraction = 0.8423)

pred.cells.smbo <- predict(rg.cells.smbo, data = cells_testing)
table(cells_testing$class, pred.cells.smbo$predictions)
```

```
##
##        PS  WS
##    PS 345  45
##    WS  53 162
```

```
(344+154)/(344+46+61+154)
```

```
## [1] 0.8231405
```

We obtain an accuracy of 0.823 which is comparable to our random search model.

Looking at the runtimes and performance of all three models then, we can see that grid search took the longest to run (3 minutes), SMBO is in the middle (1.88 minutes), and random search was the shortest (1.3 minutes). Each model seemed to have comparable accuracies with grid search being only slightly better than the other two (by a margin of 0.02). It is important to note that these runtimes are not representative for all datasets. The `cells` data set used in this example was rather small, and these tuning methods' runtimes may in fact be rather drastically different when utilized on a significantly larger data set, as we will see in section 6.

6 Real-World Example

Using knowledge gained from the previous section where we investigated the effectiveness and capabilities of each of the three tuning methods, we will begin a deep-dive into a large scale investigation on real-world data, specifically investing the language learning capabilities of individuals.

There is a commonly known belief that children learn languages much better than adults, but there is a lack of empirical evidence to support this which is why several researchers have conducted a large study with over 600,000 English learners (with participants from all facets of life) and conducted an English grammar test for all participants to assess these beliefs. Information in the dataset consist of demographic variables of the study partcipants alongside the proportion of correct answers in their grammar test.

We will be creating a ML classification task, specifically a random forest model, to aid in identifying individuals based on their skill levels (which we will assume to be connected directly to their English grammar test). Individuals are grouped into 4 distinct groups based on the proportion of answers the got correct on their test: "< 0.85", "0.850-0.899", "0.900-0.949", and "> 0.95".

Grid search and random search will not be implemented in this analysis as it simply takes up too much computational time and is unfeasible to utilize with the scale of our data (left machine to run grid search and random search on language learning data, and it took over a day/24 hours, and it still did not complete). As a result of this, model tuning will be handled with SMBO.

Before running our SMBO tuning algorithm, we will create a model using some arbitrary random hyperparameter values to see how a model will run without any sort of tuning methods.

```
rg.lang2 <- ranger(skill_group ~ ., data = language_training,
                       mtry = 5, min.node.size = 1000,
                       sample.fraction = 0.5)
pred.lang2 <- predict(rg.lang2, data = language_testing)
table(language_testing$skill_group, pred.lang2$predictions)
```

```
##
##                < 0.85 0.850-0.899 0.900-0.949 > 0.95
##    < 0.85       10440         159         153      3
##    0.850-0.899     27       10932         109      4
##    0.900-0.949      1           0       21414      0
##    > 0.95           0           0           0  20332
```

```
(10440+10932+21414+20332)/(63574)
```

```
## [1] 0.9928273
```

We obtain an accuracy of 0.9928, which is quite good already.

Will SMBO tuning help give some optimum hyperparameters for us to use to improve this? Let us run our SMBO algorithm and see how the hyperparameters it suggests – performs:

The SMBO method took 48.72681 minutes to complete. From the output, it is suggested to use the following hyperparameters: `mtry` = 20, `min.node.size` = 3558, and `sample.fraction` = 0.5818905.

Creating this model then:

```
set.seed(7271999)
#random forest creation
rg.lang <- ranger(skill_group ~ ., data = language_training,
                      mtry = 20, min.node.size = 3558,
                      sample.fraction = 0.5818905)

pred.lang <- predict(rg.lang, data = language_testing)
table(language_testing$skill_group, pred.lang$predictions)
```

```
##
##                < 0.85 0.850-0.899 0.900-0.949 > 0.95
##    < 0.85       10755           0           0      0
##    0.850-0.899      0       11072           0      0
##    0.900-0.949      0           0       21415      0
##    > 0.95           0           0           0  20332
```

We can see in the confusion matrix that every single observation in the testing set seems to be correctly classified. This is very impressive. In this case, hyperparameter tuning was not essential to the creation of an accurate model, since our base model already had an accuracy of 0.99. However the importance and function that tuning algorithms provides, are not to be overlooked. In the case that it is essential to get every observation in the testing set correct, this could be a great strategy as how to do so.

7 Conclusion

Hyperparameter tuning is still a field which needs to be delved into more and has the potential to be improved upon. Although helpful in creating optimum ML models for any sort of ML-related task, we explored in this project applications of tuning algorithms ranging from grid search, random search, and sequential model-based optimization techniques as well, each with their own distinct advantages and downsides. Although helpful in certain cases when accuracy and misclassification rates is of utmost importance, it is important to consider whether or not if hyperparameter tuning is worth running for a specific task. The amount of time that it takes to optimize a model with tuning algorithms might not be worth the small amounts of improvements that the most optimum hyperparameters may give it.

8 Appendix

8.1 Data

```
penguins <- palmerpenguins::penguins
library(modeldata)
data(cells, package = "modeldata") #cells data

cells <- cells %>%
  select(c(3:58, 2))

#splitting training and testing
cells_partition <- createDataPartition(cells$class,
                                       p = 0.7, list = FALSE)
cells_training <- cells[cells_partition, ]
cells_testing <- cells[-cells_partition, ]

#defining variables and target
cells_vars <- cells[,1:56]
cells_target <- cells[,57]
## setting working directory
language <- read_csv("data/language_learning_cleaned.csv")

language <- language %>%
  janitor::clean_names() %>%
  mutate(age_group = case_when(
    age < 18 ~ "< 18",
    age >= 18 & age < 30 ~ "18-29",
    age >= 30 & age < 45 ~ "30-44",
    age >= 45 & age < 61 ~ "45-60",
    age > 60 ~ "> 60")) %>%
```

```
  mutate(age_group = factor(age_group,
                            levels = c("< 18", "18-29",
                                       "30-44", "45-60",
                                       "> 60"))) %>%
  mutate(skill_group = case_when(
    correct < 0.85 ~ "< 0.85",
    correct >= 0.85 & correct < 0.90 ~ "0.850-0.899",
    correct >= 0.90 & correct < 0.95 ~ "0.900-0.949",
    correct >= 0.95 ~ "> 0.95")) %>%
  mutate(skill_group = factor(skill_group,
                            levels = c("< 0.85", "0.850-0.899",
                                       "0.900-0.949",
                                       "> 0.95"))) %>%
  mutate(gender = factor(gender,
                         levels = c("female", "male", "other"))) %>%
  select(3:5, 7:24)

col_names <- c("natlangs", "education", "currcountry", "ebonics",
               "speaker_cat", "type", "eng_little")
language[col_names] <- lapply(language[col_names] , factor)

#splitting training and testing
language_partition <- createDataPartition(language$age_group,
                                          p = 0.7, list = FALSE)
language_training <- language[language_partition, ]
language_testing <- language[-language_partition, ]
```

8.2 Creation of Trees

```
#messy tree
messytree.control <- rpart.control(minsplit = 10, minbucket = 2,
                                           xval = 5)
messytree <- rpart(species ~ ., data = penguins, method = "class",
                         control = messytree.control)
printcp(messytree)
#plotting and output
rpart.plot(messytree, main= "Messy Tree")
#ideal tree
idealtree.control <- rpart.control(minsplit = 30, minbucket = 15,
                                       xval = 5)
idealtree <- rpart(species ~ ., data = penguins, method = "class",
                   control = idealtree.control)

printcp(idealtree)
#plotting and output
rpart.plot(idealtree, main="Ideal Tree")
```

8.3 Grid Search

```
#grid search: grid search w/ specifications
set.seed(7271999)

grid_start <- Sys.time()

#fit rf model
control <- trainControl(method = "repeatedcv",
                         number = 10, repeats = 3, search = "grid")
```

```
tunegrid <- expand.grid(mtry = c(2:5),
                        min.node.size = c(1:5),
                        splitrule = c("gini", "extratrees"))

rf_cells_fit <- caret::train(class ~ .,
                     data = cells_training,
                     method = "ranger",
                     trControl = control,
                     tuneGrid = tunegrid)
rf_cells_fit

grid_end = Sys.time()

grid_time = grid_end - grid_start
grid_time
```

8.4 Random Search

```
#random search
set.seed(7271999)

random_start <- Sys.time()

#fit rf model
control2 <- trainControl(method = "repeatedcv",
                         number = 10, repeats = 3, search = "random")

rf_cells_fit2 <- caret::train(class ~ .,
                     data = cells_training,
                     method = "ranger",
```

```
                      tuneLength = 5,
                      trControl = control2)
rf_cells_fit2

random_end = Sys.time()

random_time = random_end - random_start
random_time
```

8.5 Sequential Model-Based Optimization

```
#sequential tuning with tuneranger
seq_start <- Sys.time()

cells.task = makeClassifTask(data = cells_training, target = "class")
res = tuneRanger(cells.task, measure = list(multiclass.brier))
res

seq_end <- Sys.time()

seq_time = seq_end - seq_start
seq_time
```

8.6 Real-World Case Study

```
#sequential tuning with tuneranger
set.seed(7271999)

seq_lang_start <- Sys.time()
```

```
lang.task = makeClassifTask(data = language_training,
                            target = "skill_group")
lang.res = tuneRanger(lang.task, measure = list(multiclass.brier))
lang.res

seq_lang_end <- Sys.time()

seq_lang_time = seq_lang_end - seq_lang_start
seq_lang_time
set.seed(7271999)
#random forest creation
rg.lang <- ranger(skill_group ~ ., data = language_training,
                      mtry = 20, min.node.size = 3558,
                      sample.fraction = 0.5818905)

pred.lang <- predict(rg.lang, data = language_testing)
table(language_testing$skill_group, pred.lang$predictions)
rg.lang2 <- ranger(skill_group ~ ., data = language_training,
                      mtry = 5, min.node.size = 1000,
                      sample.fraction = 0.5)
pred.lang2 <- predict(rg.lang2, data = language_testing)
table(language_testing$skill_group, pred.lang2$predictions)
(10440+10932+21414+20332)/(63574)
```

References

Bergstra, J. & Bengio, Y. (2012), 'Random search for hyper-parameter optimization', *Journal of Machine Learning Research* **13**, 281–305.

Hutter, F., Kotthoff, L. & Vanschoren, J. (2014), *Automated Machine Learning: Methods, Systems, Challenges*, Vol. 498.

Picco, S., Villegas, L., Tonelli, F., Merlo, M., Rigau, J., Diaz, D. & Masuelli, M. (2016), 'The K-Means Algorithm Evolution', *IntechOpen* .
URL: *https://www.intechopen.com/books/advanced-biometric-technologies/liveness-detection-in-biometrics*

Probst, P., Wright, M. N. & Boulesteix, A. L. (2019), 'Hyperparameters and tuning strategies for random forest', *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* **9**(3), 1–15.

Reif, M., Shafait, F. & Dengel, A. (2012), 'Meta-learning for evolutionary parameter optimization of classifiers', *Machine Learning* **87**(3), 357–380.

Vanschoren, J. (2018), 'Meta-Learning: A Survey', *arXiv* pp. 1–29.

Yang, L. & Shami, A. (2020), 'On hyperparameter optimization of machine learning algorithms: Theory and practice', *Neurocomputing* **415**, 295–316.
URL: *https://doi.org/10.1016/j.neucom.2020.07.061*