# Pipeline Optimization

---

Finn Tomasula Martin
COSC-4557

---

## 1 Introduction

Machine learning often involves many steps, all of which may or may not improve a models performance. The sequence of these steps in order is often referred to as a machine learning pipeline. The pipeline will include every step taken from the point that the data is input to the point where the models performance is evaluated. Often times the exact steps that need to be taken differs depending on the data. So, it can often be helpful to automate the construction of a pipeline so you can simply input your data and it will output results. In this paper we will be taking a look at how we can build a machine learning pipeline that will give us optimal pipeline configurations for any dataset.

---

## 2 Dataset Description

In order to show that our pipeline works on very different datasets, we will be using two datasets with very different characteristics. The first one is called the winequality-red.csv. It contains 1599 observations on various red wines. The dataset aims to predict wine quality on a scale of 3 – 8 (3 being the worst, 8 being the best) and has 11 feature variables. The second dataset is called primary-tumor.csv. It contains observations about 309 patients with tumors. The dataset aims to predict the primary location of the patients tumor based on 17 feature variables. (The tumor dataset contains variables of type character. Before any processing is done we will change these all to facts as that will make it easier to work with).

---

## 3 Experimental Setup

To complete this evaluation we will be using the R programming language in conjunction with the mlr3verse library. The first thing we need to do for our pipeline is establish our tasks. Each task basically represents a new run through the pipeline with a new goal in mind. We will define two classification tasks, one for the wine dataset with the goal of predicting quality and the other for the tumor dataset with the goal of predicting class (tumor location). Now we can start defining our pipeline

operations (PipeOps). Mlr3 allows you to define different operations that you can later combine into a "graph" or whole pipeline. This allows us to fully define what will happen for each task as it goes through the pipeline. It also allows us to do all our optimization at the same time. The first set of PipeOps we define will involve the preprocessing of the data, or anything that we are doing before we run the model. In this case we will define three preprocessing operations. First we will deal with missing values by imputing the mode. Second we will tune whether or not to center all our features with the scale PipeOp. We can tune this by setting the robust parameter to: to_tune("TRUE", "FALSE"). Then when we go to tune the pipeline, it will try both options when picking configurations. Last we will also tune the encoding of our data to be one of the following: one-hot, treatment, helmert, poly, sum.

The next thing we will do is define our base pipelines so we have something to compare our optimized pipelines with. For this analysis we will be looking at three ML classification models: k-nearest neighbor, classification tree, and random forest. We have to do some preprocessing for each pipeline since some models will not work if there are missing values or factor features. So for each model we will impute missing values with mode, encode the features with default encoding and then run each model with default hyperparameters.

Next we will define models to be optimized. Below is a table containing each model along with the hyperparameters and value ranges to be tuned.

| **Model** | **Hyperparameters** |
|---|---|
| K-Nearest Neighbor | K = 1 - 20 |
| | Distance = 1 - 20 |
| Classification Tree | Max depth = 1 - 20 |
| | Min bucket = 1- 20 |
| Random Forest | Num trees = 50 - 200 |
| | Min node size = 1 - 10 |

We will simply define each model as a learner with all the hyperparameters listed above set to tune.

Now, we can define our whole pipeline to be optimized. For each model the following pipeline will be created: impute missing values with mode, tune the feature encoding, tune the scale, perform model stacking with a base k-nearest neighbor and random forest, tune hyperparameters of the model. The next thing we need to do is define the tuning for our pipelines. We will run each of our pipelines through mlr3s auto_tuner function. This will create a list of pipelines set to be tuned, so that when we go to evaluate all of our pipelines at the same time it will tune each one before picking the optimal configuration to be run and evaluated. There are several things we need to define for the tuner. First is the type of tuner that will be run, we will use Bayesian optimization. Next the inner resampling, we

will do 10-fold cross validation. Next, we need to define the measurement that will be used for tuning, we will pick classification error. Finally we need to specify when the optimization will terminate, we will set it to stop after 100 iterations. The tuner works by picking a hyperparameter configuration based on the Bayesian process during each iteration. It then uses the 10-fold cross validation to evaluate the performance of that specific configuration and picks a new configuration based on the results. The optimal configuration may differ between the folds but we are interested in finding the best configuration for the whole set so we simply use the configuration that produced the best average results between all folds.

Now, our pipelines are complete and we can run and evaluate them. To do this we will be using a benchmark design. Mlr3 allows you to design a benchmark grid that takes a list of tasks, a list of learners and a resampling technique. For each task it will run each learner and evaluate them according to the resampling. For our purposes, the list of tasks will be the tasks we defined at the beginning, the learners will be our base pipelines as well as our optimized pipelines, and we will use “holdout” resampling, which just splits the data into train test sets (train: 2/3, test 1/3). Once this is run we will be able to see what the optimal configuration was for each pipeline and also which tuned model was the best for each of our tasks.

---

**4 Results**

After tuning the optimal configuration for each model task combination is given in the table below:

| Task | Model | Configuration |
|---|---|---|
| Wine | K-Nearest Neighbor | Scale: False<br>encode: helmert<br>k: 16<br>distance: 14.815 |
| Tumor | K-Nearest Neighbor | Scale: False<br>encode: treatment<br>k: 12<br>distance: 12.021 |
| Wine | Classification Tree | Scale: False<br>encode: poly<br>Max depth: 18<br>Min bucket: 15 |
| Tumor | Classification Tree | Scale: False<br>encode: sum<br>Max depth: 13<br>Min bucket: 5 |
| Wine | Random Forest | Scale: False |

| | | |
|---|---|---|
| Tumor | Random Forest | encode: helmert<br>Num trees: 146<br>Min node size: 4<br>Scale: True<br>encode: sum<br>Num trees: 177<br>Min node size: 10 |

Using those configurations here is a table comparing the performance of the optimized pipelines to base pipelines with respect to classification error:

| Task | Model | Classification Error |
|---|---|---|
| Wine | Base K-Nearest Neighbor | 0.381 |
| Wine | Optimized K-Nearest Neighbor | 0.360 |
| Wine | Base Classification Tree | 0.427 |
| Wine | Optimized Classification Tree | 0.436 |
| Wine | Base Random Forest | 0.289 |
| Wine | Optimized Random Forest | 0.287 |
| Tumor | Base K-Nearest Neighbor | 0.718 |
| Tumor | Optimized K-Nearest Neighbor | 0.660 |
| Tumor | Base Classification Tree | 0.611 |
| Tumor | Optimized Classification Tree | 0.602 |
| Tumor | Base Random Forest | 0.635 |
| Tumor | Optimized Random Forest | 0.579 |

In most cases, optimizing the pipeline led to an improvement in accuracy to varying degrees. But, it did not actually improve all of our models, as you can see the base classification tree was actually a little better than the optimized one for the wine task. This could be a result of several things including some nuisance in the dataset we missed or unlucky tuning. A likely cause may be our relatively small search space as we are only optimizing two parameters with 20 options each. Some further investigation reveals that the default values for a classification tree are, max depth = 30 and min bucket = 20, which means that the base configuration doesn't even exist in our search space which may be evidence that the optimal configuration does not either. Overall, for both tasks, it would appear that using an optimized random forest is the best choice based on this analysis.

---

**5 code**

See PipelineOpt.R for all code.

https://github.com/COSC5557/pipeline-optimization-ftomasul/blob/main/PipelineOpt.R

---

**Sources**

Casalicchio G, Burk L. (2024). Evaluation and Benchmarking. In Bischl B, Sonabend R, Kotthoff L, Lang M, (Eds.), *Applied Machine Learning Using mlr3 in R*. CRC Press. https://mlr3book.mlr-org.com/evaluation_and_benchmarking.html.

Becker M, Schneider L, Fischer S. (2024). Hyperparameter Optimization. In Bischl B, Sonabend R, Kotthoff L, Lang M, (Eds.), *Applied Machine Learning Using mlr3 in R*. CRC Press. https://mlr3book.mlr-org.com/hyperparameter_optimization.html.

Thomas J. (2024). Preprocessing. In Bischl B, Sonabend R, Kotthoff L, Lang M, (Eds.), *Applied Machine Learning Using mlr3 in R*. CRC Press. https://mlr3book.mlr-org.com/preprocessing.html.

Binder, Martin, and Florian Pfisterer. “German Credit Series - Pipelines.” *Mlr*, 11 Mar. 2020, mlr-org.com/gallery/basic/2020-03-11-mlr3pipelines-tutorial-german-credit/index.html.