Boosting methods

Emanuel Sommer

2021-04-16

Contents

Chapter 1

Prerequisites

The reader should have some basic knowledge of the following topics:

- Statistical learning
  - Train/ test split
  - Cross-validation
  - Overfitting
  - Gradient descent
  - Bias variance trade-off
  - Hyperparameters
- Regression trees and random forest models
- Basic R knowledge (only for the applied part)

Chapter 2

Introduction

Introduction to the whole setting (seminar topic/ intro to topic maybe mention kaggle wins)

> "Boosting is one of the most powerful learning ideas introduced in the last twenty years." (Hastie et al., 2009)

Chapter 3

Theory

3.1 The powerful idea of gradient boosting

As roughly mentioned in the introduction section 2 the main idea of boosting is to sequentially build weak learners that form a powerful ensemble model. With weak learners models with high bias and low variance are meant that perform at least a little better than guessing. This already shows that the sequential approach of gradient boosting with 'weak' learners stands in strong contrast to bagged ensembles like random forest. There many models with low bias and high variance are fitted in a parallel fashion and the variance is then reduced by averaging over the models.(Boehmke and Greenwell, 2019) It is not totally clear from which field boosting methods emerged but some claim that the work of Freund and Schapire with respect to PAC learning in the 1990s were instrumental for their growth.(Hastie et al., 2009) PAC learning can be considered one field within the broader field of learning theory that tries to find generalization bounds for algorithms that are probably approximately correct (PAC).(Wolf, 2020) This section will first cover the general setup of gradient boosting as the most prominent method to train forward stagewise additive models. Secondly tree-based gradient boosting and finally a very efficient and robust tree-based gradient boosting algorithm namely XGBoost will be discussed in detail.

3.1.1 Forward Stagewise Additive Modeling

In the setting of the dataset $\mathcal{D} = \{(y_i, x_i) \mid i \in [N]\}$ with predictors $x_i \in \mathbb{R}^m$ and target $y_i \in \mathbb{R}$ boosting is fitting the following additive, still quite general, model.

$$\hat{y_i} = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F} \tag{3.1}$$

Where $\mathcal{F}$ is the space of learning algorithms that will be narrowed down later on. Additive expansions like this are at the core of many other powerful machine learning algorithms like Neural Networks or Wavelets.(Hastie et al., 2009)

The formulation (3.1) leads to so called forward stagewise additive modeling which basically means that one sequentially adds $f \in \mathcal{F}$ to the current model ϕ_k without changing anything about the previous models.(Hastie et al., 2009) The algorithm is shown below.

Algorithm 1: Forward Stagewise Additive Modeling (Hastie et al., 2009)

---

1. Initialize $\phi_0(x) = 0$
2. For $k = 1$ to K do:
    - $(\beta_k, \gamma_k) = argmin_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, \phi_{k-1}(x_i) + \beta f(x_i, \gamma))$
    - $\phi_k(x) = \phi_{k-1}(x) + \beta_k f(x, \gamma_k)$

Where γ parameterizes the learner $f \in \mathcal{F}$ and the β_k are the expansion coefficients. L should be a differentiable loss function.

For example for the basic L_2 loss the expression to be minimized simplifies to the following:

$$L_2(y_i, \phi_{k-1}(x_i) + \beta f(x_i, \gamma)) = (y_i - \phi_{k-1}(x_i) - \beta f(x_i, \gamma))^2$$

As $y_i - \phi_{k-1}(x_i)$ is just the residual of the previous model, the next model that is added corresponds to the model that best approximates the residuals of the current model. Although the L_2 loss has many very nice properties like the above, it lacks robustness against outliers. Therefore two alternative losses for boosting in the regression setting are worth considering.

3.1.2 Robust loss functions for regression

As the L_2 loss squares the residuals, observations with large absolute residuals are overly important in the minimization step. This effect can be reduced intuitively by just using the L_1 loss i.e. minimize over the sum over just the absolute residuals. To do this is indeed a valid approach and can reduce the influence of outliers greatly and thus make the final model more robust. Another good choice could be the **Huber** loss which tries to get the best of L_1 and L_2 loss.(Hastie et al., 2009)

$$L_{Huber}(y, f(x)) = \begin{cases} L_2(y, f(x)) & |y - f(x)| \leq \delta \\ 2\delta|y - f(x)| - \delta^2 & otherwise. \end{cases} \tag{3.2}$$

In Figure 3.1 is a comparison of the three different losses discussed so far.(Hastie et al., 2009)

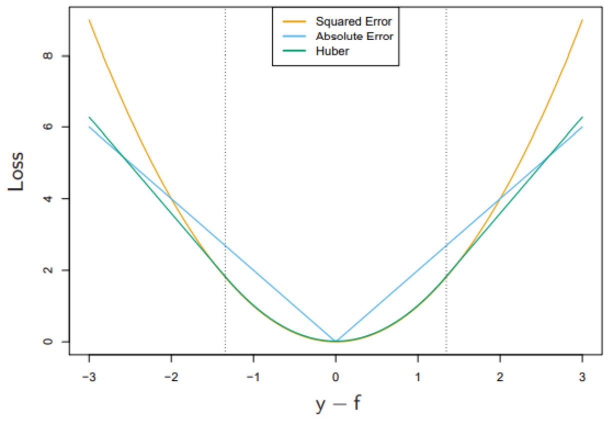

Figure 3.1: Comparison of different regression loss functions.

These alternative loss criteria are more robust but make the fitting i.e. the minimization much more complex as they do not yield such simplifications like the L_2 loss.(Hastie et al., 2009) The next step in the journey of exploring boosting is to narrow down the argument spaces of **Algorithm 1** (Forward Stagewise Additive Modeling) and to specify a subset of the general space of learning algorithms. This subset will be the space of Classification and Regression Tree (CART) models and in this case as the focus is on a regression task the space of regression trees. This choice is by no means arbitrary as in practice tree-based boosting algorithms have proven countless of times that they provide very robust and accurate models but still other learners might be chosen.(Hastie et al., 2009, Boehmke and Greenwell (2019)) The next subsection will explore how one can actually fit such a forward stagewise additive model when using regression trees as the learner class.

3.2 General gradient tree boosting

From now on there is the switch from the space of learning algorithms $\mathcal{F}$ to the space of regression trees $\mathcal{T}$. Such a regression tree can be formally expressed by:

$$t(x, \gamma, R) = \sum_{j=1}^{J} \gamma_j I(x \in R_j) \quad \text{for } t \in \mathcal{T} \tag{3.3}$$

With R_j being J distinct regions of the predictor space usually attained by recursive binary splitting. Moreover these regions correspond to the leafs of the tree and the number of leafs J or the depth of the trees are most often hyperparameters (not trained). The $\gamma_j \in \mathbb{R}$ are the predictions for a given x if x is contained in the region R_j. While it is quite easy to get the γ_j for the regions given, most often by computing $\gamma_j = \frac{1}{|\{x \in R_j\}|} \sum_{\{x \in R_j\}} x$, it is a much harder problem to get good distinct regions. The above mentioned recursive binary splitting is an approximation to do this and works in a top down greedy fashion.(Hastie et al., 2009) From now on we assume that we have an efficient way of fitting such trees to a metric outcome variable e.g. by recursive binary splitting.

A nice graphical example of an additive model based on trees is displayed in the figure 3.2 below.(Chen and Guestrin, 2016)

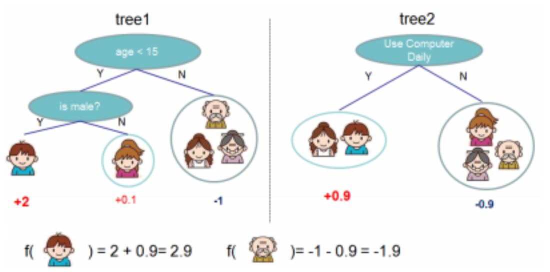

Figure 3.2: Example of an additive tree ensamble.

Having now the new space $\mathcal{T}$ for the general boosting model (3.1) one can write down the optimization problem that has to be solved in each step of the forward stagewise process of fitting the model.

$$(\gamma^{(k)}, R^{(k)}) = argmin_{\gamma,R} \sum_{i=1}^{N} L(y_i, \phi_{k-1}(x_i) + t(x_i, \gamma, R)) \tag{3.4}$$

This can be estimated fast and quite straight forward if there is a simplification like the one seen for the L_2 loss. But in the more general case of an arbitrary differentiable convex loss function like the Huber loss techniques from numerical optimization are needed to derive fast algorithms.(Hastie et al., 2009)

3.2.1 Numerical optimization

According to **Algorithm 1** the goal in order to fit the boosting model is to minimize over the full loss of the training data $\mathcal{D}$ which is the sum over all observation losses.

$$L(\phi) = \sum_{i=1}^{N} L(y_i, \phi(x_i))$$

And thus the $\hat{\phi}$ additive boosting model we try to get is the following.

$$\hat{\phi} = argmin_{\phi} L(\phi)$$

Now we basically follow the spirit of the general gradient descent algorithm for differentiable functions. In the general case we minimize a function $f(x)$ by stepping iteratively along the direction of the steepest descent i.e. the negative gradient. The step length can then either be a constant small scalar or be determined by a line search.

In the setting of the additive boosting model ϕ can be viewed as a vector of dimension N that contains the prediction according to ϕ of the corresponding observation i.e. $\phi = (\phi(x_1), ..., \phi(x_N))$. So the loss function $L(\phi)$ corresponds to the $f(x)$ in the general gradient descent algorithm. Numerical optimization, here gradient descent, then solves for $\hat{\phi}$ by a sum of vectors of the same dimension as the ϕ.(Hastie et al., 2009) The result of the sum ϕ_K ((3.5)) can be viewed as the current proposal for the optimal $\hat{\phi}$ after K optimization steps and each $h^{(k)}$ is the proposed improvement step.

$$\phi_K = \sum_{k=0}^{K} h^{(k)} \quad h^{(k)} \in \mathbb{R}^N \tag{3.5}$$

While $h^{(0)}$ is just an initial guess the subsequent $h^{(k)}$ are again the prediction vectors of the corresponding model out of $\mathcal{T}$ i.e. $h_i^{(k)} = t_k(x_i)$ with $t \in \mathcal{T}$. This means that each $\phi_k = \phi_{k-1} + h^{(k)}$. The $h^{(k)}$ are calculated via the gradient which finally comes into play. As mentioned above one minimizes the loss the most by going towards the direction of the steepest descent. For (3.5) follows from its additive formulation and defining $g^{(k)}$ as the gradient of $L(\phi_k)$ evaluated for ϕ_{k-1} the update $h^{(k)} = -\lambda_k g^{(k)}$. As we assumed the loss to be differentiable we see that $g^{(k)}$ is well defined. Here λ_k is the usual step length for gradient descent methods. It is the solution of the line search $\lambda_k = argmin_{\lambda} L(\phi_{k-1} - \lambda g^{(k)})$.

This λ_k almost exactly corresponds to the β_k in **Algorithm 1** although here the optimization is performed for every region of the tree separately.(Hastie et al., 2009)

With these insights it is clear that the tree predictions correspond to the negative gradient $-g^{(k)}$. Of course the predictions are not independent as the prediction is constant for each leaf of the tree. So the new optimization proposed by numerical optimization via gradient boosting is given in (3.6) below.

$$(\tilde{\gamma}^{(k)}, \tilde{R}^{(k)}) = argmin_{\gamma,R} \sum_{i=1}^{N} [-g_i^{(k)} - t(x_i, \gamma, R)]^2 \tag{3.6}$$

In words this just means fitting a regression tree by least squares to the negative gradients that were evaluated with the current predictions. The solution regions will not exactly match the ones from (3.4) but should be very similar.(Hastie et al., 2009) After having estimated the regions one estimates the parameters γ by solving the line search (3.7).

$$\tilde{\gamma}_j^{(k)} = argmin_{\gamma_j^{(k)}} \sum_{x \in R_j^{(k)}} L(y_i, \phi_{k-1}(x_i) + \gamma_j^{(k)}) \tag{3.7}$$

Putting all of this back together with **Algorithm 1** results in **Algorithm 2** that covers a general algorithm for tree-based gradient boosting.

Algorithm 2: Tree-based gradient boosting (Hastie et al., 2009)

---

1. Initialize $\phi_0(x)$ as a singular node tree.
2. For $k = 1$ to K do:
   - For $i = 1$ to N compute:
     $g_i^{(k)} = \left[\frac{\partial L(y_i, \phi(x_i))}{\partial \phi(x_i)}\right]_{\phi=\phi_{k-1}}$
   - Fit a regression tree by least squares to the outcome vector $-g^{(k)}$ in order to get the $J^{(k)}$ distinct regions $\tilde{R}_j^{(k)}$.
   - For each of these $J^{(k)}$ regions perform a line search in order to compute the leaf predictions $\tilde{\gamma}_j^{(k)}$ exactly like in (3.7).
   - Set $\phi_k(x) = \phi_{k-1}(x) + t(x, \tilde{\gamma}_j^{(k)}, \tilde{R}_j^{(k)})$ with $t \in \mathcal{T}$

The only unknowns in this algorithm are now the differentiable loss function and the hyperparameters like the $J^{(k)}$ and the K. While choices for the hyperparameters are discussed further below, the following table 3.1 displays the gradients for the losses discussed so far.

Table 3.1: Gradients of the discussed losses (Hastie et al., 2009)

Loss	Gradient
L_2: $(y_i - \phi(x_i))^2$	$2(y_i - \phi(x_i))$
L_1: $\|y_i - \phi(x_i)\|$	$sign(y_i - \phi(x_i))$
Huber loss (3.2)	$y_i - \phi(x_i)$ for $\|y_i - \phi(x_i)\| \leq \delta$
	$\delta sign(y_i - \phi(x_i))$ otherwise, with δ quantile of $\|y_i - \phi(x_i)\|$

3.2.2 Single tree depth

The question which $J^{(k)}$ should be used at each iteration is now shortly discussed. Basically using the tree depth two means allowing only the main effects and no interactions. A value of 3 already allows all two way interaction effects and so on. As one stacks these models additive and does not restrict oneself to a single one, the building of a large tree and then pruning it back at each iteration as the regular CART algorithms do would be a computational overkill. Instead it has proven to be sufficient and efficient in practice to set the values $J^{(k)}$ to a constant $J \approx 6$.(Hastie et al., 2009) Also decision stumps ($J = 2$) could be used but may require a lot more iterations.

3.2.3 Combat overfitting

No learning algorithm could be fully covered without treating the good old problem of overfitting. In the setting of boosting the experienced eye could have spotted the problem of overfitting already in the definition of the additive model (3.1). There the number K of models that form the ensemble was introduced but was not discussed further till now. Of course one can arbitrarily fit or better say remember some given training data in order to minimize the loss with such an additive model by letting K be arbitrarily large. This comes from the fact that the loss reduces usually after each iteration over K.(Hastie et al., 2009) The easiest way to prevent overfitting is to have a validation set at hand which is disjoint from the training data. Having such a validation set at hand can be used to monitor the loss on the unseen data. In the case the loss would rise again on the validation data one can consider to stop the algorithm and to use the current iteration number for the final parameter K. This approach is often called early stopping. Besides that there are methods that regularize the individual trees that are fitted in each iteration. Two of those will be discussed now.

3.2.3.1 Shrinkage

Shrinkage basically refers to just introducing a learning rate η. This learning rate η scales the contribution of the new model. In general the learning rate should be in $(0, 1]$ but in practice it has been shown that rather small values like $\eta < 0.1$ work very good.(Hastie et al., 2009) As almost everything in life this does not come for free. A smaller learning rate usually comes with a computational cost as with lower η a larger K is required. Those two hyperparameters represent a trade-off in a way. In practice it is advisable to use a small learning rate η and adjust the K accordingly which in this case would mean to make the K large enough until one reaches an early stopping point. Still it is good to keep in mind that a η that is too small can lead to an immense and unnecessary computational effort. All in all using shrinkage the update step in **Algorithm 2** changes to the following.

$$\phi_k(x) = \phi_{k-1}(x) + \eta * t(x, \tilde{\gamma}_j^{(k)}, \tilde{R}_j^{(k)}) \tag{3.8}$$

3.2.3.2 Subsampling

The other method to regularize the trees will be subsampling. There are two different kinds of subsampling. The first one is row-subsampling which basically means just using a random fraction of the training data in each iteration when fitting the new tree. Common values are $\frac{1}{2}$ but for a larger training set the value can be chosen to be smaller. This does not only help to prevent overfitting and thus achieving a better predictive performance but reduces also the computational effort in each iteration.(Hastie et al., 2009) The approach is very similar to the dropout technique in the setting of deep neural networks.

Moreover one can apply column-subsampling or feature-subsampling. Here only a fraction of the features or an explicit number of the features are used in each iteration. This is the exact same method and intention as the one applied in random forest models. Again the technique not only boosts performance on unseen data but also reduces the computational time.(Chen and Guestrin, 2016) Still it must be noted that by using one or both subsampling methods one introduces one or two additional hyperparameters to the model that have to be tuned.

Having the **Algorithm 2** for tree-based gradient boosting alongside some regularization techniques it is time to look at a very popular, efficient and powerful open source implementation.

3.3 XGBoost a highly efficient implementation

The XGBoost algorithm is a highly scalable, efficient and successful implementation and optimization of **Algorithm 2** introduced by Chen and Guestrin in 2016.(Chen and Guestrin, 2016, Boehmke and Greenwell (2019)) It has gained a lot of spotlight when it was integral for many winning submissions at kaggle machine learning challenges. In the following the most important tweaks that are proposed to **Algorithm 2** are covered.

3.3.1 Regularized loss

Instead of just minimizing a convex and differentiable loss L XGBoost minimizes the following regularized loss.(Chen and Guestrin, 2016)

$$\begin{aligned} \mathcal{L}(\phi) &= L(\phi) + \sum_{k=1}^{K} \Omega(t_k) \\ \text{where } \Omega(t) &= \nu J + \frac{1}{2}\lambda||\gamma||^2 \quad t \in \mathcal{T} \end{aligned} \tag{3.9}$$

Here J and γ again parameterize the regression trees $t \in \mathcal{T}$ as defined in (3.3). As evident from the formula both hyperparameters ν and λ favor when increased less complex models at each iteration. This is again a measure against overfitting.

This regularized objective leads to a slightly modified version of the minimization problem (3.4) that has to be solved in each iteration.

$$(\gamma^{(k)}, R^{(k)}) = argmin_{\gamma,R} \sum_{i=1}^{N} L(y_i, \phi_{k-1}(x_i) + t(x_i, \gamma, R)) + \Omega(t(x_i, \gamma, R)) \tag{3.10}$$

Instead of the first order approach, that has been introduced in 3.2.1, XGBoost uses a second order approximation to further simplify the objective. This means that besides the first order gradient $g^{(k)}$ of the loss function evaluated at ϕ_{k-1} also the second order gradient $z_i^{(k)} = \left[\frac{\partial^2 L(y_i, \phi(x_i))}{\partial^2 \phi(x_i)}\right]_{\phi=\phi_{k-1}}$ is used. By neglecting constants one reaches the approximate new objective below.(Chen and Guestrin, 2016)

$$\begin{aligned}\tilde{\mathcal{L}}^{(k)} &= \sum_{i=1}^{N}[g_i^{(k)}t^{(k)}(x_i) + \frac{1}{2}t^{(k)}(x_i)^2] + \Omega(t^{(k)}) \\ &= \sum_{j=1}^{J}[(\sum_{\{i|x_i \in R_j^{(k)}\}} g_i^{(k)})\gamma_j^{(k)} + \frac{1}{2}(\sum_{\{i|x_i \in R_j^{(k)}\}} z_i^{(k)} + \lambda)(\gamma_j^{(k)})^2] + \nu J\end{aligned} \tag{3.11}$$

This representation is used as it can be decomposed by the J leafs of the tree. One can then define an scoring function for split finding that only depends on $g^{(k)}$ and $z^{(k)}$.

3.3.2 Shrinkage and subsampling

As discussed above in 3.2.3 shrinkage and subsampling are great ways to regularize the model and prevent overfitting. XGBoost implements both shrinkage and subsampling. XGBoost enables the user to use column as well as row subsampling. The authors claim that in practice column subsampling has prevented overfitting more than row subsampling.(Chen and Guestrin, 2016)

3.3.3 Even more tweaks

Besides the changes above the authors also implemented an effective approximate algorithm for split finding for the tree building step. A very nice attribute of this approximate procedure is that the algorithm is sparsity aware i.e. the processing of missing values or 0 values is very efficient. This is done via a learned default direction in each split. Moreover they used very efficient data structures to allow a lot of parallel processing.(Chen and Guestrin, 2016) It is written in C++ but has APIs in various languages like in R via the `xgboost` package.(Chen et al., 2021)

3.3.4 Hyperparameters overview

As the **tidymodels** framework will be used in the practical part, the according names from the parsnip interface for tree-based boosting i.e. `boost_tree` are discussed below.(Chen and Guestrin, 2016, Chen et al. (2021), Kuhn and Wickham (2020))

Table 3.2: XGBoost hyperparameters overview.

Hyperparameter	Notation above	Meaning
`trees`	K	Number of trees in the ensemble.
`tree_depth`	$max(J^{(k)})$	Maximum depth of the trees.
`learn_rate`	η	Learning rate (shrinkage).
`min_n`		Minimum number of observations in terminal region $R_j^{(k)}$.
`lambda`	λ	L_2 regularization parameter on γ as in (3.11) (default = 0).
`alpha`		L_1 regularization parameter on γ (default = 0).
`loss_reduction`	ν	Minimal loss reduction for each partition.
`mtry`		Proportion or number of columns for column subsampling.
`sample_size`		Proportion of observations for row subsampling.
`stop_iter`		Allowed number of rounds without improvement before early stopping.

This closes the chapter on the theory of boosting methods with a focus on tree-based gradient boosting. The discussed tools will be put to the test in the subsequent sections where real world data will be analyzed with them.

Chapter 4

Expore the data

We use the following packages.(Wickham et al., 2019; Kuhn and Wickham, 2020) and use the mantra of tidyverse as well as tidy modeling with R

```
library(tidyverse)
library(tidymodels)
```

4.1 Train-test split

```
set.seed(2)
# ames_split <- initial_split(ames, prop = 0.80, strata = Sale_Price)
# ames_train <- training(ames_split)
# ames_test  <-  testing(ames_split)
```

4.2 Visualize the data

- pair plots
  - indiviual main effects
  - correlations
- correlation plot
- individual distributions
- identify categorical features
- feature engineering

4.3 Create recipe

one for lm (dummify categorical features) and one for the tree based approaches

```
# ames_rec <-
#   recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
#            Latitude + Longitude, data = ames_train) %>%
#   step_log(Gr_Liv_Area, base = 10) %>%
#   step_other(Neighborhood, threshold = 0.01) %>%
#   step_dummy(all_nominal()) %>%
#   step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
#   step_ns(Latitude, Longitude, deg_free = 20)
```

Chapter 5

Let's boost the models

boooooooooooooooooooooooost them (gif here?)

Cite additional model packages.(Chen et al., 2021, Wright and Ziegler (2017))

```
library(xgboost)
library(ranger)
```

To do

1. register parallel backend
2. create models and workflows
3. create resampling objects
4. choose tuning method (racing/iterative or grid search)
5. tune models and select best ones
6. compare final models to test set
7. save final model

cite (Corporation and Weston, 2019)

```
library(doParallel)

# Create a cluster object and then register:
cl <- makePSOCKcluster(2)
registerDoParallel(cl)

# Put at the end:
stopCluster(cl)
```

for 2. set engine specific arguments in the set_engine() function use tune() for parameters that should be tuned (optional tune("id_name"))

```
# Models:
lm_model <- linear_reg() %>%
  set_engine("lm")

rf_model <- rand_forest(trees = 1000) %>%
  set_engine("ranger") %>%
  set_mode("regression")

boost_model <- boost_tree() %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

```
# Workflows:

# lm_wflow <-
#   workflow() %>%
#   add_model(lm_model) %>%
#   add_recipe()
```

for 3.

```
# Create Resampling objects
# set.seed(2)
# ames_folds <- vfold_cv(ames_train, v = 10)
```

for fitting resampling objects without tuning:

```
# set.seed(2)
# rf_res <- rf_wflow %>%
#   fit_resamples(resamples = ames_folds)
# # collect the metrics during resampling with
# collect_metrics()
```

for 4. and following

```
# show the parameters to be tuned with the range
# dials::parameters()
# get and modify the parameters to be tuned with:
# name()
# wflow_param %>% pull_dials_object("threshold")
# parameters(ames_rec) %>%
#  update(threshold = threshold(c(0.8, 1.0)))

# finalize for data dependent params
# updated_param <-
#   workflow() %>%
#   add_model(rf_spec) %>%
#   add_recipe(pca_rec) %>%
#   parameters() %>%
#   finalize(ames_train)
```

grids: grid_regular(levels = c(hidden_units = 3, penalty = 2, epochs = 2)) space filling: grid_latin_hypercube(size = 15, original = FALSE) then tune_grid() function instead of fit_resamples

```
roc_res <- metric_set(roc_auc)
set.seed(99)
mlp_reg_tune <-
  mlp_wflow %>%
  tune_grid(
    cell_folds,
    grid = mlp_param %>% grid_regular(levels = 3),
    metrics = roc_res
  )

autoplot(mlp_reg_tune) + theme(legend.position = "top")

show_best(mlp_reg_tune)

# for spacefilling
mlp_sfd_tune <-
  mlp_wflow %>%
  tune_grid(
    cell_folds,
    grid = 20,
    # Pass in the parameter object to use the appropriate range:
    param_info = mlp_param,
    metrics = roc_res
  )

select_best()
```

```
logistic_param <-
  tibble(
    num_comp = 0,
    epochs = 125,
    hidden_units = 1,
    penalty = 1
  )

final_mlp_wflow <-
  mlp_wflow %>%
  finalize_workflow(logistic_param)

final_mlp_fit <-
  final_mlp_wflow %>%
  fit(cells)
```

Racing:

```
library(finetune) # if used to be cited

set.seed(99)
mlp_sfd_race <-
  mlp_wflow %>%
  tune_race_anova(
    cell_folds,
    grid = 20,
    param_info = mlp_param,
    metrics = roc_res,
    control = control_race(verbose_elim = TRUE)
  )
```

Here no iterative search as big parameter space, could be done after grid search or for single parameters. (If then Simulated Annealing)

Chapter 6

Conclusion

Let's wrap it up!

Chapter 7

References

Bibliography

Boehmke, B. and Greenwell, B. (2019). *Hands-On Machine Learning with R.*

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754.

Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., Mitchell, R., Cano, I., Zhou, T., Li, M., Xie, J., Lin, M., Geng, Y., and Li, Y. (2021). *xgboost: Extreme Gradient Boosting.* R package version 1.3.2.1.

Corporation, M. and Weston, S. (2019). *doParallel: Foreach Parallel Adaptor for the 'parallel' Package.* R package version 1.0.15.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning (12th printing).* Springer New York.

Kuhn, M. and Wickham, H. (2020). *Tidymodels: a collection of packages for modeling and machine learning using tidyverse principles.*

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wolf, M. M. (2020). Lecture notes in mathematical foundations of supervised learning.

Wright, M. N. and Ziegler, A. (2017). ranger: A fast implementation of random forests for high dimensional data in C++ and R. *Journal of Statistical Software*, 77(1):1–17.