Parameter Tuning: A KNN Example

Jaime Davila and Adam Loy

Last updated:

Intended Learning Outcomes

By the end of this activity, you will be able to:

- Tune the parameters of a KNN model using cross validation
- Explore the results of the tuning process
- Finalize a KNN model workflow with the selected parameter values

Overview

Now that you know how to fit KNN models and how to conduct cross validation to estimate their out-of-bag performance, it's time to explore how to tune the parameters of a KNN model using the functionality of the {tidymodels} ecosystem.

In this activity we'll revisit the Scooby Doo data set and use it to build a KNN model that predicts whether a monster is real or not. Our goal is to select the "best" value for the number of neighbors, `k`. To begin, let's load in the data and create our training and testing splits:

```r
scooby <- readr::read_csv("scooby.csv", col_types="ddfcd")
set.seed(1234)

scooby_split <- initial_split(scooby,
                              prop = 0.75,
                              strata = monster_real)
scooby_train <- training(scooby_split)
scooby_test <- testing(scooby_split)
```

Tuning a KNN Model

To tune a model, we must first define our model, which includes specifying the recipe and the model specification, and then combining these components into a workflow. There is one key change required, however, to perform parameter tuning: we need to use the `tune()` function to indicate which parameter we want to tune in our model specification. In the case of KNN, we set `neighbors = tune()` rather than setting it to a specific value.

```r
# Recipe definition
knn_recipe <- recipe(monster_real ~ imdb + year_aired, data = scooby_train) |>
  step_normalize(all_numeric_predictors()) |>
  step_naomit(all_predictors())
```

```r
# Model specification
knn_spec <- nearest_neighbor(neighbors = tune()) |>
  set_mode("classification") |>
  set_engine("kknn")

# Workflow definition
knn_wflow <- workflow() |>
  add_recipe(knn_recipe) |>
  add_model(knn_spec)
```

Next, we need to create our cross-validation splits. In this example, let's use 5-fold cross-validation.

```r
# Cross-validation splits
scooby_folds <- vfold_cv(scooby_train, v = 5, strata = monster_real)
```

We also need to create a grid of values for the `neighbors` parameter, for example from 1 to 21 (odds only), using the `grid_regular()` function. We then pass this grid to the `tune_grid()` function along with our workflow and cross-validation folds. When tuning a single parameter you could also create a tibble with the values you want to test, but using `grid_regular()` is a convenient way to generate a range of values and makes tuning multiple parameters easier.

```r
k_grid <- grid_regular(neighbors(range = c(1, 21)), levels = 10)
```

Note: {tidymodels} does have functionality to automatically generate candidate parameter grids for many models; however, it's better to manually specify the values when you are first learning to reinforce your understanding of the parameters and their impact on model performance.

With the grid defined, we can now perform the tuning process by passing in the workflow, the cross-validation folds, and the grid of values to the `tune_grid()` function. We also specify the metrics we will use to evaluate the model's performance, such as accuracy, sensitivity, and specificity. Recall that we use the `metric_set()` to combine multiple metrics into a single object that can be passed to the tuning function.

```r
scooby_tune <- tune_grid(
  knn_wflow,
  resamples = scooby_folds,
  grid = k_grid,
  metrics = metric_set(accuracy, sens, spec)
)
```

To collect the results of the tuning process, we use the `collect_metrics()` function. This returns a tibble with the performance metrics for each value of `k` that was explored.

```r
(scooby_tune_results <- collect_metrics(scooby_tune))
```

```
## # A tibble: 30 x 7
##    neighbors .metric  .estimator  mean     n std_err .config
##        <int> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
##  1         1 accuracy binary     0.835     5  0.0186 Preprocessor1_Model01
##  2         1 sens     binary     0.917     5  0.0253 Preprocessor1_Model01
##  3         1 spec     binary     0.549     5  0.0267 Preprocessor1_Model01
##  4         3 accuracy binary     0.827     5  0.0172 Preprocessor1_Model02
##  5         3 sens     binary     0.914     5  0.0225 Preprocessor1_Model02
##  6         3 spec     binary     0.525     5  0.0386 Preprocessor1_Model02
##  7         5 accuracy binary     0.843     5  0.0194 Preprocessor1_Model03
##  8         5 sens     binary     0.918     5  0.0199 Preprocessor1_Model03
##  9         5 spec     binary     0.585     5  0.0391 Preprocessor1_Model03
## 10         7 accuracy binary     0.875     5  0.0121 Preprocessor1_Model04
## # i 20 more rows
```

Your turn 1: Explore the results of 5-fold cross validation for the various values of `k`. What is the best value of `k` based on the accuracy metric? How does this value compare to the sensitivity and specificity metrics?

It can be tedious to explore the table of results manually, particularly if you have many parameters to tune. To quickly visualize the results we use the `autoplot()` function from the {tune} package. This function will create a plot of the performance metrics for each value of our parameters, allowing us to quickly identify the best performing model.

Your turn 2: Use the `autoplot()` function to visualize the results of the tuning process. What does the plot tell you about the relationship between the number of neighbors and the model's performance?

If your goal is to select the "best" value of `k`, you can use the `select_best()` function to extract the best performing parameter value based on a specific metric. For example, if you want to select the best value of `k` based on accuracy, you can do the following:

```
best_k <- select_best(scooby_tune, metric = "accuracy")
```

This will return a tibble with the best value of `k` and its corresponding accuracy.

Your turn 3: What is the best value of `k` based on sensitivity? For specificity?

Once you have determined what value of `k` to use, you can **finalize** your model.

Other 1 SE for optimizing parameters

Often there are many values of `k` that yield similar performance. In these cases, it can be beneficial to select a model that is easier to interpret and less likely to overfit the training data. One common approach is the "one standard error" rule, which selects the simplest model whose performance is within one standard error of the best performing model. This can be done using the `select_by_one_std_err()` function, which takes the results of `tune_grid()`, the name of the performance metric (as a character string), and the parameter(s) to optimize (unquoted).

Your turn 4: What value of `k` is selected using the one standard error rule based on sensitivity? For specificity?

Your turn 5: Finalize the model using the value of `k` selected by the one standard error rule based on accuracy. How does this model perform on the test data compared to the model selected by the best accuracy?

{usemodels} for parameter optimization

It can be rather tedious to type out all the code required to tune a starting from the recipe. The {usemodels} package implements model skeletons for a few commonly used models, including KNN. Given a formula and data set it will create the recipe, model specification, and workflow for you. It also sets the framework for tuning the model with helpful reminders of the components you should think carefully about.

```r
library(usemodels)
use_kknn(monster_real ~ imdb + year_aired, data = scooby)
```

```
## kknn_recipe <-
##   recipe(formula = monster_real ~ imdb + year_aired, data = scooby) %>%
##   step_zv(all_predictors()) %>%
##   step_normalize(all_numeric_predictors())
##
## kknn_spec <-
##   nearest_neighbor(neighbors = tune(), weight_func = tune()) %>%
##   set_mode("classification") %>%
##   set_engine("kknn")
##
## kknn_workflow <-
##   workflow() %>%
##   add_recipe(kknn_recipe) %>%
##   add_model(kknn_spec)
##
## set.seed(68778)
## kknn_tune <-
##   tune_grid(kknn_workflow, resamples = stop("add your rsample object"), grid = stop("add number of ca
```

Your turn 6: What two objects do you need to define in order to run the tuning process using the {usemodels} package? What alterations to the code are needed?

Functions used

- `grid_regular()`: Generates a regular grid of parameter values for tuning.
- `tune_grid()`: Performs tuning of a model using cross-validation and a grid of parameter values.
- `autoplot()`: Visualizes the results of the tuning process, showing performance metrics for different parameter values.
- `select_best()`, `select_by_one_std_err()`: Functions to select the best parameter values based on a specific metric or the one standard error rule.
- `finalize_workflow()`: Finalizes a workflow with the best parameter values selected from tuning.