Classification using KNN

Jaime Davila and Adam Loy

Last updated:

Intended Learning Outcomes

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

- Fit a KNN classifier using the {tidymodels} framework
- Predict the class label for a new observation using a fitted KNN classifier
- Evaluate the performance of a KNN classifier using accuracy, sensitivity, and specificity

Data set

This activity is inspired by the following blog post from Julia Silge. We will be using a data set collected from the popular animated series, Scooby Doo. Specifically, we'll build a KNN model to predict whether a monster is real or not.

To begin, you can load the data

```
scooby <- read_csv("scooby.csv", col_types = "ddfcd")
```

and take a `glimpse()` at it:

```
glimpse(scooby)
```

```
## Rows: 501
## Columns: 5
## $ year_aired      <dbl> 1969, 1969, 1969, 1969, 1969, 1969, 1969, 1969, 1969, ~
## $ imdb            <dbl> 8.1, 8.1, 8.0, 7.8, 7.5, 8.4, 7.6, 8.2, 8.1, 8.0, 8.5,~
## $ monster_real    <fct> fake, fake, fake, fake, fake, fake, fake, fake, fake, ~
## $ title           <chr> "What a Night for a Knight", "A Clue for Scooby Doo", ~
## $ suspects_amount <dbl> 2, 2, 0, 2, 1, 2, 1, 2, 1, 1, 1, 1, 2, 2, 1, 3, 3, 2, ~
```

Teaching note: This is a "clean" version of the data set. If you want to see the original data set, you can load it from the Tidy Tuesday Github Repo. The original data set has some missing values, many logical columns that should be converted to more useful factor variables, and many columns not used in this analysis. If you want to focus on the full data analysis cycle, then you could start with the this raw data set.

Now that the data set is loaded, let's split our data set into a training and testing set. We'll use 75% of the data for training and 25% for testing. We will also stratify our data set by the `monster_real` column to ensure that the training and testing sets have a similar breakdown of real and fake monsters.

```r
# Be sure to run the setup chunk at the top of the document first!
set.seed(1234)
scooby_split <- initial_split(scooby, prop = 0.75, strata = monster_real)
scooby_train <- training(scooby_split)
scooby_test <- testing(scooby_split)
```

Classification and KNN

We are interested in determining whether the monster in the episode is real or fake. To do this we will use the year that the episode was aired (`year_aired`) and the rating the episode got on `imdb`.

Your turn 1: Create a scatterplot of `imdb_rating` vs. `year` and color the points by `real_monster`. What do you observe?

Your turn 2: A KNN classifier assigns a new observation a label based on the labels of the `k` closest observations using a *majority vote*. In the scatterplot below there are two new observations marked with a black `x`. For each of the new observations, determine whether a KNN classifier would label the observation as a real monster or not based on the following values of `k`:

i. `k = 1`
ii. `k = 3`
iii. `k = 5`

Reflection questions:

- Did you choose the closest neighbors based on the `year_aired` or the `imdb_rating` or a combination of the two?
- Were there any major difficulties in deciding on the closest observations?

Your turn 3: Quantitative variables should be standardized (normalized) before training (fitting) a KNN classifier. Explain in your own words why this is recommended.

A little more data wrangling

Before going any farther we should standardize the predictor variables. When making predictions we want to use the same data wrangling steps, and use the mean and standard deviations from the training set to standardize the predictor variables in the testing set to avoid "data leakage". To do this, we will create a **recipe** using the {recipes} package from the {tidymodels} framework. A recipe is a way to specify a sequence of data wrangling steps that can be applied to a data set. The recipe will be used to prepare the training data set, and then it will be "baked" to apply the same steps to the testing data set. This ensures that the same transformations are applied to both training and testing data sets, which is crucial for model performance.

To begin a recipe, specify a formula that describes the response variable and the predictor variables. The response variable is placed to the left of the `~` and the predictors are placed on the right separated by `+`. Additional data processing steps are added with `step_*()` functions. For example, the `step_naomit()` function allows us to remove rows with missing values and the `step_normalize()` function allows us to standardize the variables.

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

To view the results, we need to `prep()` and `bake()` the recipe. The `prep()` function prepares the recipe by calculating the necessary statistics (e.g., means and standard deviations) from the training data set, and the `bake()` function applies the same transformations to the training and testing data sets.

```
prep(knn_recipe, training = scooby_train) |>
  bake(new_data = scooby_train)
```

```
## # A tibble: 375 x 3
##     imdb year_aired monster_real
##    <dbl>      <dbl> <fct>
##  1 1.04       -1.57 fake
##  2 0.636      -1.57 fake
##  3 1.44       -1.57 fake
##  4 0.369      -1.57 fake
##  5 0.904      -1.57 fake
##  6 1.17       -1.57 fake
##  7 1.44       -1.57 fake
##  8 1.04       -1.57 fake
##  9 1.44       -1.57 fake
## 10 0.904      -1.51 fake
## # i 365 more rows
```

Your turn 4: Create a scatterplot of `imdb_rating` vs `year` and color the points by `real_monster` using the updated (normalized) `scooby_train` data set.

Fitting a KNN classifier in the {tidymodels} framework

Now that we have our data set ready, and we are familiar with the intuition behind KNN for classification, we can fit a KNN classifier using the {tidymodels} framework. For now, let's assume we are interested in `k = 3` nearest neighbors.

As in a KNN regression model we create a **model specification** using the `nearest_neighbor()` function. The key difference in the specification for a classification model is that we set the mode to `"classification"` instead of `"regression"`.

```
# Model specification
knn_spec <- nearest_neighbor(neighbors = 3) |>
  set_mode("classification") |>
  set_engine("kknn")
```

As with KNN regression we next create a **workflow**, which combines a model specification and a recipe into a single object.

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

Finally, we fit our model to the training set using the `fit()` function.

```
# Model training
(knn_model <- fit(knn_wflow, data = scooby_train))
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## -- Preprocessor ----------------------------------------------------------------
## 2 Recipe Steps
##
## * step_normalize()
## * step_naomit()
##
## -- Model ----------------------------------------------------------------------
##
## Call:
## kknn::train.kknn(formula = ..y ~ ., data = data, ks = min_rows(3,     data, 5))
##
## Type of response variable: nominal
## Minimal misclassification: 0.1333333
## Best kernel: optimal
## Best k: 3
```

Making predictions and evaluating performance

To obtain the predicted class labels we can use the `augment()` function, which adds columns for the predicted class labels (`.pred_class`), the predicted probabilities of an observation being in that class (`.pred_fake` and `.pred_real` here) to the specified data set.

```
augment(knn_model, new_data = scooby_test)
```

```
## # A tibble: 126 x 8
##    .pred_class .pred_fake .pred_real year_aired  imdb monster_real title
##    <fct>            <dbl>      <dbl>      <dbl> <dbl> <fct>        <chr>
##  1 fake                 1          0       1969   8.1 fake         What a Night~
##  2 fake                 1          0       1969   8   fake         Hassle in th~
##  3 fake                 1          0       1969   7.5 fake         Decoy for a ~
##  4 fake                 1          0       1969   8.2 fake         Foul Play in~
##  5 fake                 1          0       1969   8.1 fake         The Backstag~
##  6 fake                 1          0       1969   8.5 fake         A Gaggle of ~
##  7 fake                 1          0       1970   8.7 fake         A Night of F~
##  8 fake                 1          0       1970   8.3 fake         Nowhere to H~
##  9 fake                 1          0       1970   8.5 fake         Jeepers, It'~
## 10 fake                 1          0       1970   7.9 fake         A Tiki Scare~
## # i 116 more rows
## # i 1 more variable: suspects_amount <dbl>
```

Your turn 5: Use the `augment()` function and fill in the plotting template to create a jittered scatterplot of the predicted probabilities of being a real monster (`.pred_real`) vs. the predicted probabilities of being a fake monster (`.pred_fake`). Color the points by `monster_real`. What do you observe?

Your turn 6: The below code creates a table of the predicted class labels and the actual class labels. What is the accuracy of your model? Report both the numeric value and give an interpretation of what accuracy means.

```
augment(knn_model, new_data = scooby_test) |>
  count(monster_real, .pred_class)
```

```
## # A tibble: 4 x 3
##   monster_real .pred_class     n
##   <fct>        <fct>       <int>
## 1 fake         fake           92
## 2 fake         real            6
## 3 real         fake           15
## 4 real         real           13
```

```
augment(knn_model, new_data = scooby_test) |>
  conf_mat(truth = monster_real, estimate = .pred_class)
```

```
##           Truth
## Prediction fake real
##       fake   92   15
##       real    6   13
```

Your turn 7: Sensitivity is the proportion of true positives that are correctly identified (the true positive rate).

- What is a positive prediction in this KNN model?
- How do we interpret sensitivity in this context?
- Can you think of an example where sensitivity is more important than accuracy?
- Use the above table to calculate the sensitivity of your model.

Your turn 8: Specificity is the proportion of true negatives that are correctly identified (the true negative rate).

- What is a negative prediction in this KNN model?
- How do we interpret specificity in this context?
- Can you think of an example where specificity is more important than accuracy?
- Use the above table to calculate the specificity of your model.

While you should be able to calculate accuracy, sensitivity, and specificity "by hand" from a summary table, the {yardstick} package (loaded with {tidymodels}) provides a number of functions to calculate these metrics. For example, the `accuracy()` function takes a tibble with the predicted class labels and the actual class labels as arguments and returns the accuracy of the model. Similarly, the `sensitivity()` (also `sens()`) and `specificity()` (`spec()`) functions take a tibble with the predicted class labels and the actual class labels as arguments and return the sensitivity and specificity of the model, respectively.

Run the below code to check your calculations from above.

```
scooby_test_aug <- augment(knn_model, new_data = scooby_test)
scooby_test_aug |> accuracy(truth = monster_real, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.833
```

```
scooby_test_aug |> sensitivity(truth = monster_real, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 sensitivity binary         0.939
```

```
scooby_test_aug |> specificity(truth = monster_real, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 specificity binary         0.464
```

Note: We can also make predictions for all observations in the testing data set using the `predict()` function. The `predict()` function takes a fitted model object and a new data set as arguments, and it returns a tibble with the predicted class labels as the only column. While we will typically use `augment()` to get the predicted class labels along with the original data set, `predict()` is useful when we **only** want the predicted class labels.

```
predict(knn_model, new_data = scooby_test)
```

```
## # A tibble: 126 x 1
##    .pred_class
##    <fct>
##  1 fake
##  2 fake
##  3 fake
##  4 fake
##  5 fake
##  6 fake
##  7 fake
##  8 fake
##  9 fake
## 10 fake
## # i 116 more rows
```

Review exercise

You now know the basics of KNN models for classification and how to fit them using the {tidymodels} framework. You also know how to make predictions and evaluate the model performance using accuracy, sensitivity, and specificity. To see if this make sense, complete the following exercise:

Fit a KNN model with `k = 5` using `imdb`, `year`, and `suspects_amount` to predict whether a monster is real (`monster_real`). Calculate the model's accuracy, sensitivity, and specificity on the test set and write a brief summary of your findings. * How does the model perform compared to the model with `k = 3`? * What do you think about the number of snacks as a predictor variable? * Do you think KNN is a good model for this data set? Why or why not?

Functions introduced

- `initial_split()`: Creates a training/testing split of a data set
- `training()`, `testing()`: Extracts the training and testing data sets from a split
- `step_normalize(all_numeric_predictors())`: Standardizes all numeric predictors in a recipe
- `step_naomit(all_predictors())`: Removes rows with missing values in a recipe
- `prep()`, `bake()`: Prepares a recipe and applies it to a data set
- `conf_mat()`, `accuracy()`, `sensitivity()`, `specificity()`: Functions to calculate confusion matrix, accuracy, sensitivity, and specificity