Class Activity 25

Your name here

May 28 2023

Group Activity 1: Address Classification using Random Forest in R

In this tutorial, we will use a Random Forest model to classify whether a given address is correctly formatted. The dataset consists of synthetic examples generated by chatgpt, with certain cases based on various features extracted from the address. The outcome variable is the Label, which indicates if the address is correctly formatted.

Splitting the Data

The first step is to split our data into a training set and a testing set. We will use 75% of the data for training and reserve 25% for testing.

```
# Set random seed for reproducibility
set.seed(314)

# Split data into training and testing set
db_split <- initial_split(street_address %>% select(-1) %>% janitor::clean_names(), prop = 0.75)
db_train <- training(db_split)
db_test <- testing(db_split)
```

Preprocessing

Our data contains categorical variables, so we need to convert them into numerical form using dummy variables. We define a recipe to do this preprocessing.

```
# Create a recipe for preprocessing
db_recipe <- recipe(label ~ ., data = db_train) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  prep()
```

Defining the Model

We will use a Random Forest model for our classification task. Random Forest is a powerful machine learning algorithm that can handle both regression and classification problems. It works well with both categorical and numerical data.

```
# Define the model specification
rf_model <- rand_forest(mtry = tune(),
                        trees = 1000,
                        min_n = tune()) %>%
  set_engine('ranger', importance = "impurity") %>%
  set_mode('classification')
```

We then combine our model and preprocessing recipe into a workflow.

```
# Combine the model and recipe into a workflow
rf_workflow <- workflow() %>%
  add_model(rf_model) %>%
  add_recipe(db_recipe)
```

Cross Validation

Next, we will perform cross-validation on our training data to avoid overfitting. This will also help us tune our model parameters.

```
# Create folds for cross validation on the training data set
db_folds <- vfold_cv(db_train, v = 5)
```

Hyperparameter Tuning

We define a grid of parameters for tuning our model. We will use a random search strategy for our grid, which is more efficient than an exhaustive grid search.

```
# Define the grid for tuning
rf_grid <- grid_random(
  mtry(range = c(1, 10)),
  min_n(range = c(1, 10)),
  size = 10
)
```

We then use this grid to tune our model.

```
# Tune the model
rf_tuning <- tune_grid(
  rf_workflow,
  resamples = db_folds,
  grid = rf_grid
)
```

After tuning, we select the best model based on accuracy, finalize the workflow with the best parameters, and then fit the model on the training data.

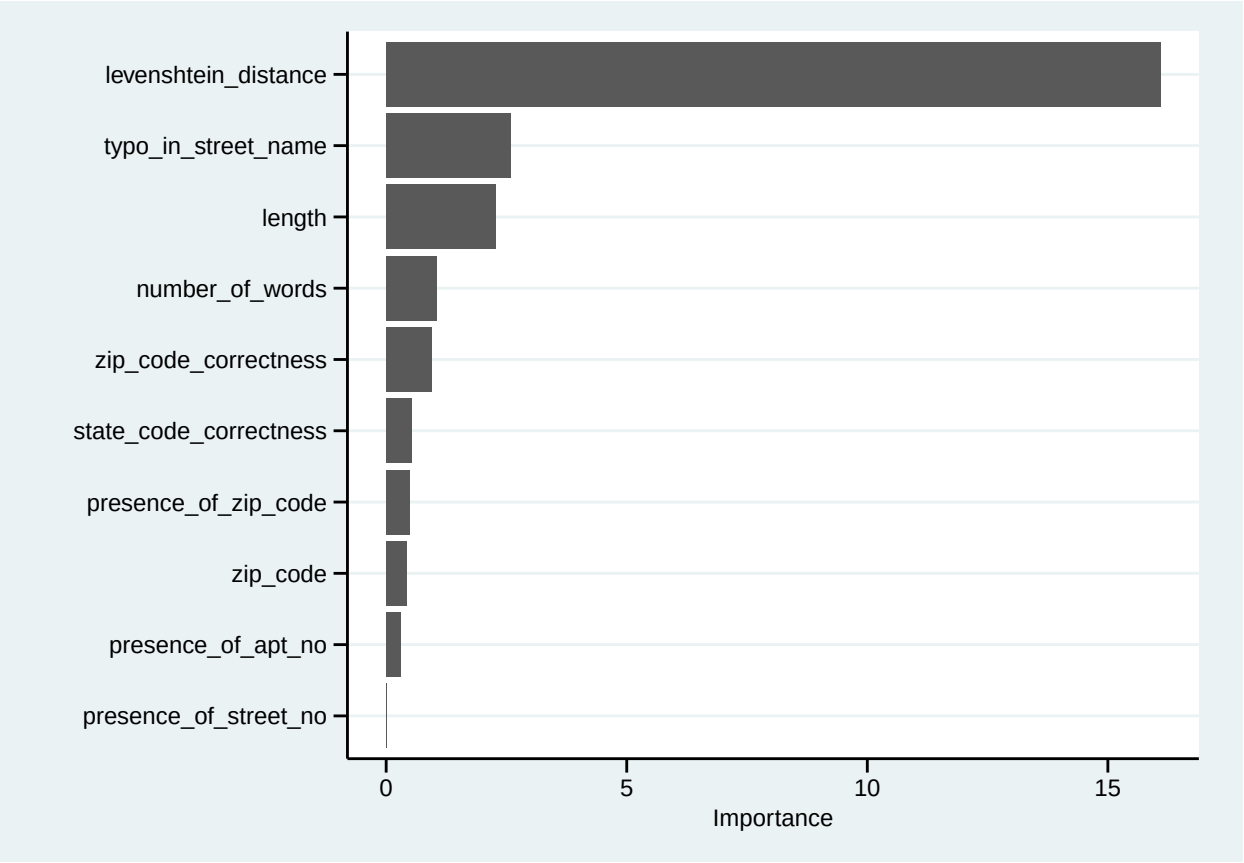
```r
# Select the best model based on accuracy
best_rf <- select_best(rf_tuning, metric = "accuracy")

# Finalize the workflow with the best parameters
final_rf_workflow <- finalize_workflow(rf_workflow, best_rf)

# Fit the model on the training data
rf_fit <- fit(final_rf_workflow, data = db_train)
```

We can also plot the importance of each feature in our model. This tells us which features have the greatest impact on the model's decision-making process.

```r
# Extract the fitted model and plot feature importance
library(vip)
rf_fit_parsnip <- extract_fit_parsnip(rf_fit)
vip(rf_fit_parsnip) + theme(axis.text.y = element_text(angle = 0))
```



Making Predictions

Now that our model is trained, we can use it to make predictions on our test data.

```r
# Make predictions on the test data
predictions <- predict(rf_fit, new_data = db_test)
```

```r
# Compare predictions to true outcomes
db_results <- db_test %>% bind_cols(predictions) %>% select(label, .pred_class)

# Print confusion matrix
conf_mat(db_results %>% mutate(label = factor(label)), truth = label, estimate = .pred_class)
```

```
           Truth
Prediction  Correct Incorrect
  Correct         3         0
  Incorrect       1        22
```

Feature Extraction

```r
# Define function to extract features
extract_features <- function(address) {

  # Split the address into components
  components <- str_split(address, ",\\s*")[[1]]

  # Extract specific components
  street_apt <- components[1]
  city <- components[2]
  state_zip <- components[3]

  # Further split specific components
  street_apt_split <- str_split(street_apt, "\\s+")[[1]]
  state_zip_split <- str_split(state_zip, "\\s+")[[1]]

  # Identify each component
  street_number <- str_extract(street_apt, "^\\d+")
  street_name <- str_replace(street_apt, "^\\d+\\s?", "") # remove the street number
  apt_number <- ifelse(str_detect(street_apt, "(?i)apt"), 1, 0)
  state <- state_zip_split[1]
  zip_code <- str_extract(address, "\\d+")

  # Calculate features
  length <- str_length(address)
  num_words <- str_count(address, "\\w+")
  state_code <- ifelse(state == "MN", 1, 0)
  zip_code_present <- ifelse(!is.na(zip_code), 1, 0)
  zip_code_correctness <- ifelse(zip_code == "55057", 1, 0)
  typo_in_street_name <- stringdist::stringdist("2nd st E", street_name)

  # Return as a data frame
  tibble(
    length = length,
    number_of_words = num_words,
    state_code = state_code,
    zip_code = zip_code_present,
    levenshtein_distance = typo_in_street_name,
    presence_of_apt_no = apt_number,
```

```
    presence_of_street_no = as.integer(!is.na(street_number)),
    presence_of_street_name = as.integer(!is.na(street_name)),
    presence_of_city = as.integer(!is.na(city)),
    presence_of_state = as.integer(!is.na(state)),
    presence_of_zip_code = as.integer(zip_code_present),
    state_code_correctness = state_code,
    zip_code_correctness = zip_code_correctness,
    typo_in_street_name = as.integer(typo_in_street_name > 0)
  )
}
```

Here's how we can use this function:

```
# Use the function
new_address1 <- "514 2nd st E, Apt 1w, Northfield, MN, 55057"
new_address_features1 <- extract_features(new_address1)

new_address2 <- "514 2nd street E, Northfield, MN, 5505"
new_address_features2 <- extract_features(new_address2)
```

Now we have extracted the features from the new addresses, let's use our trained Random Forest model to predict whether the address formatting is correct or not.

```
# Predict the class for the new data
new_data_predictions1 <- predict(rf_fit, new_data = new_address_features1)
new_data_predictions2 <- predict(rf_fit, new_data = new_address_features2)

# View the predictions
new_data_predictions1
# A tibble: 1 x 1
  .pred_class
  <fct>
1 Correct
new_data_predictions2
# A tibble: 1 x 1
  .pred_class
  <fct>
1 Incorrect
```

This will give us the predicted class (correctly formatted or not) for each of the new addresses.

In this tutorial, we walked through the steps of splitting data into training and testing sets, pre-processing the data, defining the model specification, creating folds for cross validation, tuning the model, selecting the best model, fitting the model on the training data, extracting feature importance, making predictions on the test data and finally, making predictions on new data.

We hope this tutorial helped you understand how to build a Random Forest model for address formatting correctness prediction. Happy coding!

Group Activity 2

Gen Z or Millenial?

```
# Gen Z slang words
genz_slang <- c("lit", "dead", "cap", "sksksk", "oop", "fleek", "low key", "high key", "gucci", "clout"

# Millennial slang words
millennial_slang <- c("on point", "fomo", "yolo", "can't even", "adulting", "slay", "squad", "bae", "ne

slang <- str_to_lower(c(genz_slang, millennial_slang))
```

```
library(tidyverse)
library(stringr)
library(tidytext)

# Create a function to count slang words
count_slang <- function(text, slang) {
  text_df <- tibble(text = text)

# Tokenize the text for each n-gram length
tokens <- bind_rows(
  text_df %>% unnest_tokens(output = word, input = text, token = "ngrams", n = 1),
  text_df %>% unnest_tokens(output = word, input = text, token = "ngrams", n = 2),
  text_df %>% unnest_tokens(output = word, input = text, token = "ngrams", n = 3)
)

# Count the slang
slang_count <- tokens %>%
  filter(word %in% slang) %>%
  count(word) %>% summarize(sum = sum(n))

  return(slang_count)
}

# Example usage
text <- "It's lit! Totally vibing with this new song blow up beat "
count_slang(text, slang)
# A tibble: 1 x 1
    sum
  <int>
1     2
```

```
# Create a function to compute text features
compute_text_features <- function(text, genz_slang, millennial_slang) {
  num_slang_words <- count_slang(text, c(genz_slang, millennial_slang))

  text_df <- tibble(text = text) %>%
    unnest_tokens(word, text, token = "characters")

  num_punctuation <- text_df %>%
    filter(word %in% str_extract_all(text, "[[:punct:]]")[[1]]) %>%
```

```
    nrow()



  return(data.frame(Num_Slang_Words = num_slang_words, Num_Punctuation = num_punctuation))
}
```

```
# Example usage
text <- "I can't even"
features <- compute_text_features(text, genz_slang, millennial_slang)
print(features)
  sum Num_Punctuation
1   2               0
```