1. Introduction

In the world of Pokémon academia, one name towers above any other – Professor Samuel Oak. While his colleague Professor Elm specializes in Pokémon evolution, Oak has dedicated his career to understanding the relationship between Pokémon and their human trainers. A former trainer himself, the professor has first-hand experience of how obstinate Pokémon can be – particularly when they hold legendary status.

For his latest research project, Professor Oak has decided to investigate the defining characteristics of legendary Pokémon to improve our understanding of their temperament. Hearing of our expertise in classification problems, he has enlisted us as the lead researchers.

Our journey begins at the professor's research lab in Pallet Town, Kanto. The first step is to open up the Pokédex, an encyclopaedic guide to 801 Pokémon from all seven generations.

Source: bagogames (https://www.flickr.com/photos/bagogames/) on Flickr

In [94]:
```
# Load the tidyverse
library(tidyverse)

# Import the dataset and convert variables
pokedex <- read_csv("datasets/pokedex.csv",
                    col_types = cols(name = col_factor(),
                                     type = col_factor(),
                                     is_legendary = col_factor()))

# Look at the first six rows
head(pokedex)

# Examine the structure
str(pokedex)
```

pokedex_number	name	attack	defense	height_m	hp	percentage_male	sp_attac
1	Bulbasaur	49	49	0.7	45	88.1	65
2	Ivysaur	62	63	1.0	60	88.1	80
3	Venusaur	100	123	2.0	80	88.1	122
4	Charmander	52	43	0.6	39	88.1	60
5	Charmeleon	64	58	1.1	58	88.1	80
6	Charizard	104	78	1.7	78	88.1	159

```
Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame':        801 obs. of
14 variables:
 $ pokedex_number : num  1 2 3 4 5 6 7 8 9 10 ...
 $ name           : Factor w/ 801 levels "Bulbasaur","Ivysaur",..: 1 2 3 4 5
6 7 8 9 10 ...
 $ attack         : num  49 62 100 52 64 104 48 63 103 30 ...
 $ defense        : num  49 63 123 43 58 78 65 80 120 35 ...
 $ height_m       : num  0.7 1 2 0.6 1.1 1.7 0.5 1 1.6 0.3 ...
 $ hp             : num  45 60 80 39 58 78 44 59 79 45 ...
 $ percentage_male: num  88.1 88.1 88.1 88.1 88.1 88.1 88.1 88.1 88.1 50 ...
 $ sp_attack      : num  65 80 122 60 80 159 50 65 135 20 ...
 $ sp_defense     : num  65 80 120 50 65 115 64 80 115 20 ...
 $ speed          : num  45 60 80 65 80 100 43 58 78 45 ...
 $ type           : Factor w/ 18 levels "grass","fire",..: 1 1 1 2 2 2 3 3 3
4 ...
 $ weight_kg      : num  6.9 13 100 8.5 19 90.5 9 22.5 85.5 2.9 ...
 $ generation     : num  1 1 1 1 1 1 1 1 1 1 ...
 $ is_legendary   : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 1 ...
 - attr(*, "spec")=
  .. cols(
  ..   pokedex_number = col_double(),
  ..   name = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE),
  ..   attack = col_double(),
  ..   defense = col_double(),
  ..   height_m = col_double(),
  ..   hp = col_double(),
  ..   percentage_male = col_double(),
  ..   sp_attack = col_double(),
  ..   sp_defense = col_double(),
  ..   speed = col_double(),
  ..   type = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE),
  ..   weight_kg = col_double(),
  ..   generation = col_double(),
  ..   is_legendary = col_factor(levels = NULL, ordered = FALSE, include_na =
FALSE)
  .. )
```

In [95]:
```
library(testthat)
library(IRkernel.testthat)

soln_pokedex <- read_csv("datasets/pokedex.csv",
                         col_types = cols(name = col_factor(),
                                          type = col_factor(),
                                          is_legendary = col_factor()))

run_tests({

    test_that("packages are loaded", {
        expect_true("tidyverse" %in% .packages(), info = "Did you load the `ti
dyverse` package?")
    })

    test_that("data is loaded and formatted correctly", {
        expect_equal(pokedex, soln_pokedex,
                         info = "Did you convert `type` and `is_legendary` to
 factors using `col_factor()`?")
    })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 19.323 0.28 594.199 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

2. How many Pokémon are legendary?

After browsing the Pokédex, we can see several variables that could feasibly explain what makes a Pokémon legendary. We have a series of numerical fighter stats – `attack`, `defense`, `speed` and so on – as well as a categorization of Pokemon `type` (bug, dark, dragon, etc.). `is_legendary` is the binary classification variable we will eventually be predicting, tagged `1` if a Pokémon is legendary and `0` if it is not.

Before we explore these variables in any depth, let's find out how many Pokémon are legendary out of the 801 total, using the handy `count()` function from the `dplyr` package.

In [96]:
```
# Prepare the data
legendary_pokemon <- pokedex %>%
  count(is_legendary) %>%
  mutate(prop = n / nrow(pokedex))

# Print the data frame
legendary_pokemon
```

is_legendary	n	prop
0	731	0.91260924
1	70	0.08739076

In [97]:
```
soln_legendary_pokemon <- soln_pokedex %>%
  count(is_legendary) %>%
  mutate(prop = n / nrow(soln_pokedex))

run_tests({

    test_that("data is prepared correctly", {

        expect_equal(colnames(legendary_pokemon)[1], colnames(soln_legendary_p
okemon)[1],
                         info = "Did you pass `is_legendary` to the `count()`
 function?")

        expect_equal(legendary_pokemon[3], soln_legendary_pokemon[3],
                     info = "Did you divide `n` by `nrow(pokedex)`?")

  })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 19.361 0.28 594.237 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

3. Legendary Pokémon by height and weight

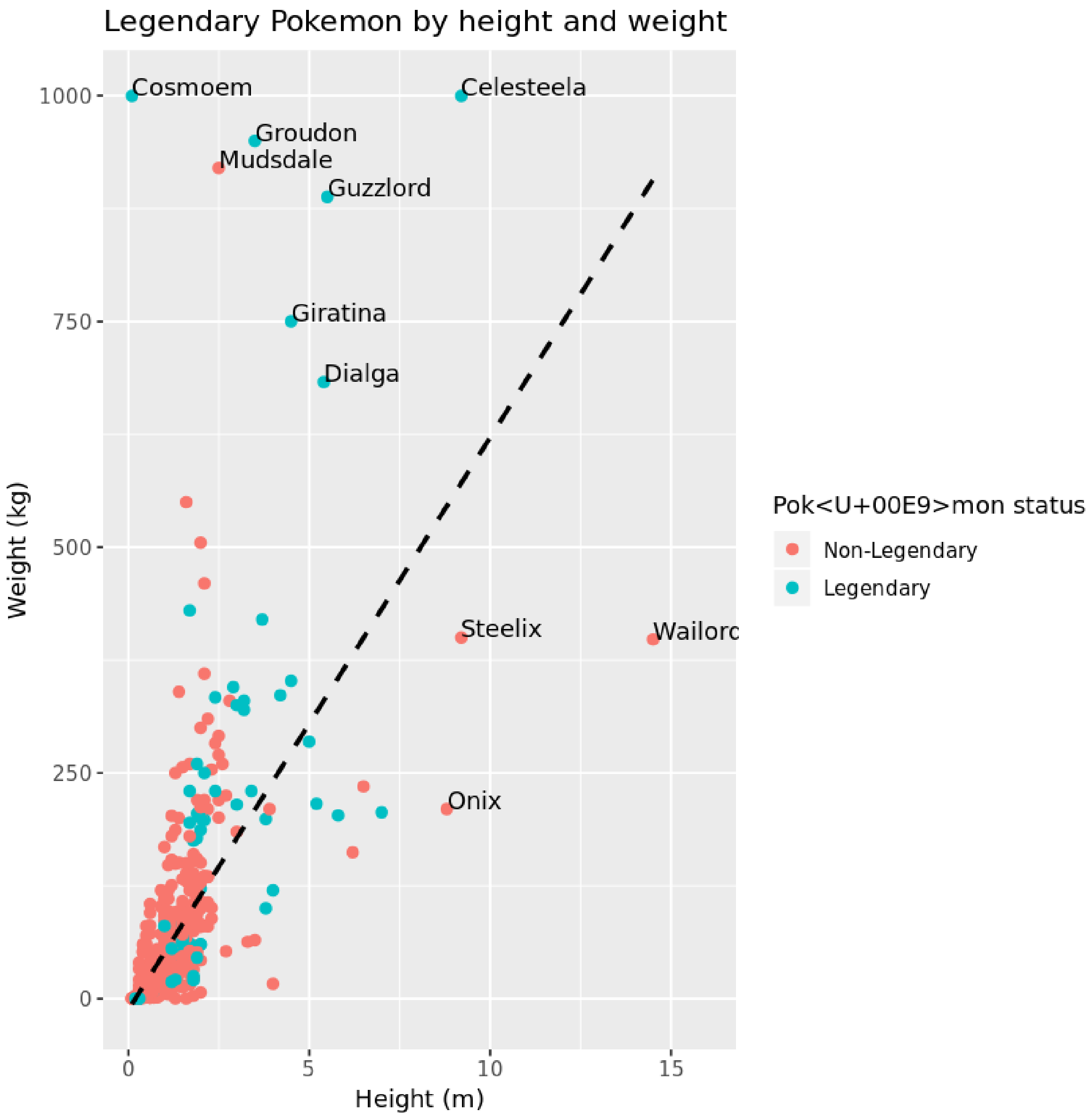
We now know that there are 70 legendary Pokémon – a sizable minority at 9% of the population! Let's start to explore some of their distinguishing characteristics.

First of all, we'll plot the relationship between `height_m` and `weight_kg` for all 801 Pokémon, highlighting those that are classified as legendary. We'll also add conditional labels to the plot, which will only print a Pokémon's name if it is taller than 7.5m or heavier than 600kg.

In [98]:
```
# Prepare the plot
legend_by_heightweight_plot <- pokedex %>%
  ggplot(aes(x= height_m,
            y = weight_kg)) +
  geom_point(aes(color = is_legendary), size = 2) +
  geom_text(aes(label = ifelse(height_m > 7.5 | weight_kg > 600, as.character(
name), '')),
            vjust = 0, hjust = 0) +
  geom_smooth(method = "lm", se = FALSE, col = "black", linetype = "dashed") +
  expand_limits(x = 16) +
  labs(title = "Legendary Pokemon by height and weight",
       x = "Height (m)",
       y = "Weight (kg)") +
  guides(color = guide_legend(title = "Pokémon status")) +
  scale_color_manual(labels = c("Non-Legendary", "Legendary"),
                     values = c("#F8766D", "#00BFC4"))

# Print the plot
legend_by_heightweight_plot
```

```
Warning message:
"Removed 20 rows containing non-finite values (stat_smooth)."Warning message:
"Removed 20 rows containing missing values (geom_point)."Warning message:
"Removed 20 rows containing missing values (geom_text)."
```



In [99]:
```
soln_legend_by_heightweight_plot <- soln_pokedex %>%
  ggplot(aes(x = height_m, y = weight_kg)) +
  geom_point(aes(color = is_legendary), size = 2) +
  geom_text(aes(label = ifelse(height_m > 7.5|weight_kg > 600, as.character(na
me), '')), vjust = 0, hjust = 0) +
  geom_smooth(method = "lm", se = FALSE, col = "black", linetype = "dashed") +
  expand_limits(x = 16) +
  labs(title = "Legendary Pokemon by height and weight",
       x = "Height (m)",
       y = "Weight (kg)") +
  guides(color = guide_legend(title = "Pokemon status")) +
  scale_color_manual(labels = c("Non-Legendary", "Legendary"),
                     values = c("#F8766D", "#00BFC4"))
run_tests({

  test_that("the mappings are correct", {

        expect_equal(deparse(legend_by_heightweight_plot$mapping$x),
                         deparse(soln_legend_by_heightweight_plot$mapping$x),
                         info = 'The `x` aesthetic is incorrect. Did you map i
t to `height_m`?')

        expect_equal(deparse(legend_by_heightweight_plot$mapping$y),
                         deparse(soln_legend_by_heightweight_plot$mapping$y),
                         info = 'The `y` aesthetic is incorrect. Did you map i
t to `weight_kg`?')

        expect_equal(legend_by_heightweight_plot$layers[[1]]$mapping,
                         soln_legend_by_heightweight_plot$layers[[1]]$mapping,
                         info = "The `col` aesthetic in `geom_point()` is inco
rrect. Did you map it to `is_legendary`?")

  })

  test_that("the conditional labels are correct", {

        expect_true(str_detect(legend_by_heightweight_plot$labels$label, "heig
ht_m > 7.5"),
                     info = "The conditional labels are incorrect. Did you spe
cify that `height_m` should be greater than `7.5` in the first argument of the
`ifelse()` function?")

        expect_true(str_detect(legend_by_heightweight_plot$labels$label, "weig
ht_kg > 600"),
                    info = "The conditional labels are incorrect. Did you spec
ify that `weight_kg` should be greater than `600` in the first argument of the
`ifelse()` function?")

        expect_true(str_detect(legend_by_heightweight_plot$labels$label, "\\|"
),
                    info = "The conditional labels are incorrect. Did you reme
mber to use the conditional 'OR' operator (|)?")

  })

    test_that("the plot limits are correct", {
        expect_equal(ggplot_build(legend_by_heightweight_plot)$data[[4]],
                     ggplot_build(soln_legend_by_heightweight_plot)$data[[
4]],
                     info = "The plot limits are incorrect. Did you expand
the limit of the `x`-axis to 16?")

    })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 19.76 0.288 594.642 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

4. Legendary Pokémon by type

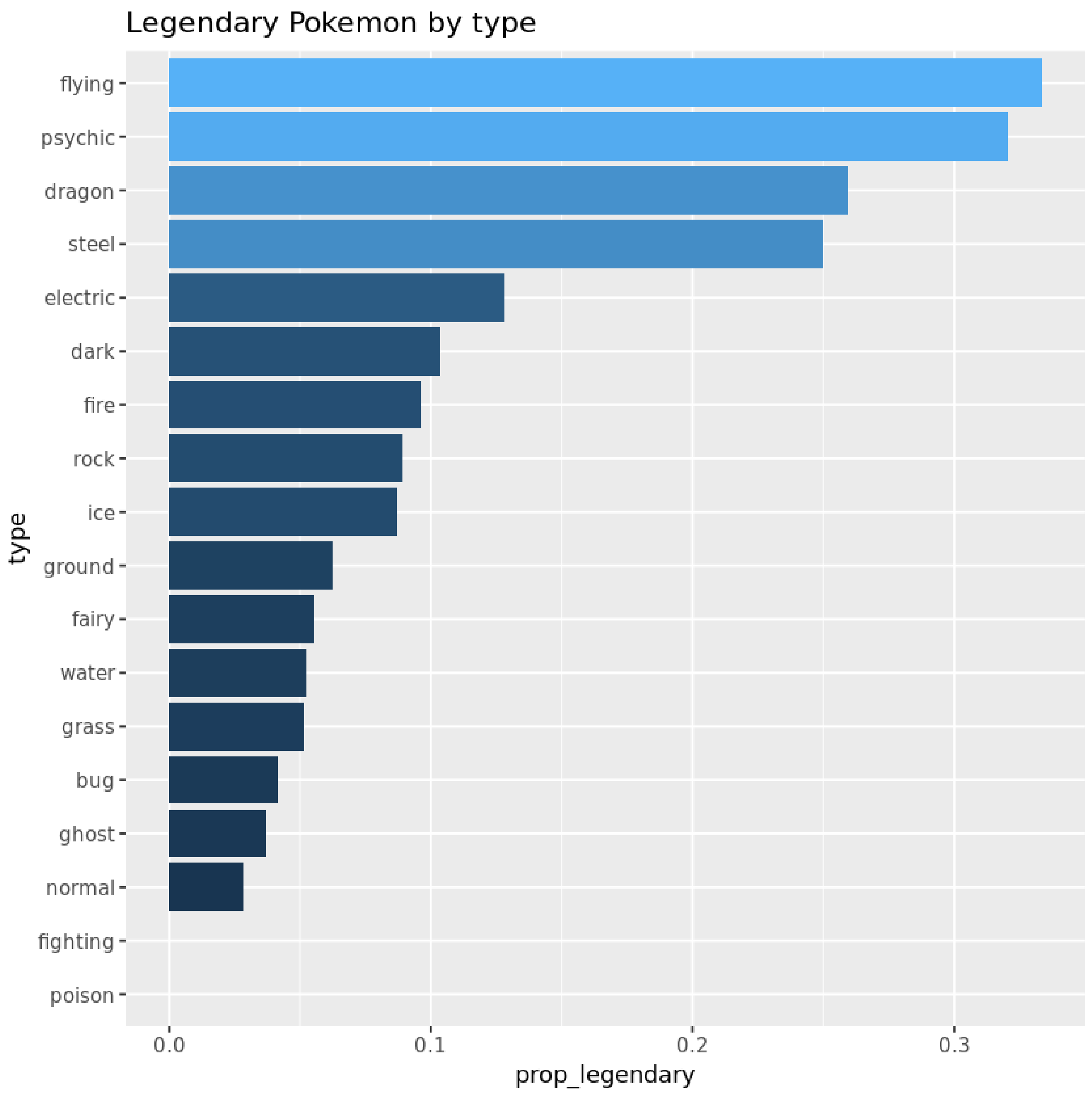
It seems that legendary Pokémon are generally heavier and taller, but with many exceptions. For example, Onix (Gen 1), Steelix (Gen 2) and Wailord (Gen 3) are all extremely tall, but none of them have legendary status. There must be other factors at play.

We will now look at the effect of a Pokémon's `type` on its legendary/non-legendary classification. There are 18 possible types, ranging from the common (Grass / Normal / Water) to the rare (Fairy / Flying / Ice). We will calculate the proportion of legendary Pokémon within each category, and then plot these proportions using a simple bar chart.

In [100]:
```r
# Prepare the data
legend_by_type <- pokedex %>%
    group_by(type) %>%
    mutate(is_legendary = as.numeric(is_legendary) - 1) %>%
    summarise(prop_legendary = mean(is_legendary)) %>%
    ungroup() %>%
    mutate(type = fct_reorder(type,prop_legendary))

# Prepare the plot
legend_by_type_plot <- legend_by_type %>%
    ggplot(aes(x = type, y = prop_legendary, fill = prop_legendary)) +
    geom_col() +
    labs(title = "Legendary Pokemon by type") +
    coord_flip() +
    guides(fill = FALSE)

# Print the plot
legend_by_type_plot
```



In [101]:
```
soln_legend_by_type <- soln_pokedex %>%
    group_by(type) %>%
    mutate(is_legendary = as.numeric(is_legendary) - 1) %>%
    summarise(prop_legendary = mean(is_legendary)) %>%
    ungroup() %>%
    mutate(type = fct_reorder(type, prop_legendary))

soln_legend_by_type_plot <- soln_legend_by_type %>%
    ggplot(aes(x = type, y = prop_legendary, fill = prop_legendary)) +
    geom_col() +
    labs(title = "Legendary Pokemon by type") +
    coord_flip() +
    guides(fill = FALSE)

run_tests({

  test_that("the data preparation is correct", {

      expect_equal(group_vars(legend_by_type), group_vars(soln_legend_by_type
),
                   info = "Did you group by `type`?")

      expect_equal(legend_by_type$prop_legendary, soln_legend_by_type$prop_leg
endary,
                   info = "Did you set `prop_legendary` equal to the mean of `
is_legendary`?")

      expect_equal(legend_by_type$type, soln_legend_by_type$type,
                  info = "Did you the `fct_reorder()` function to order `type`
by `prop_legendary`?")


  })

    test_that("the plot preparation is correct", {

      expect_equal(deparse(legend_by_type_plot$mapping$y),
                        deparse(soln_legend_by_type_plot$mapping$y),
                        info = 'The `y` aesthetic is incorrect. Did you map it
 to `prop_legendary`?')

      expect_equal(deparse(legend_by_type_plot$mapping$fill),
                        deparse(soln_legend_by_type_plot$mapping$fill),
                        info = 'The `fill` aesthetic is incorrect. Did you map
 it to `prop_legendary`?')

  })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 20.093 0.288 594.975 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

5. Legendary Pokémon by fighter stats

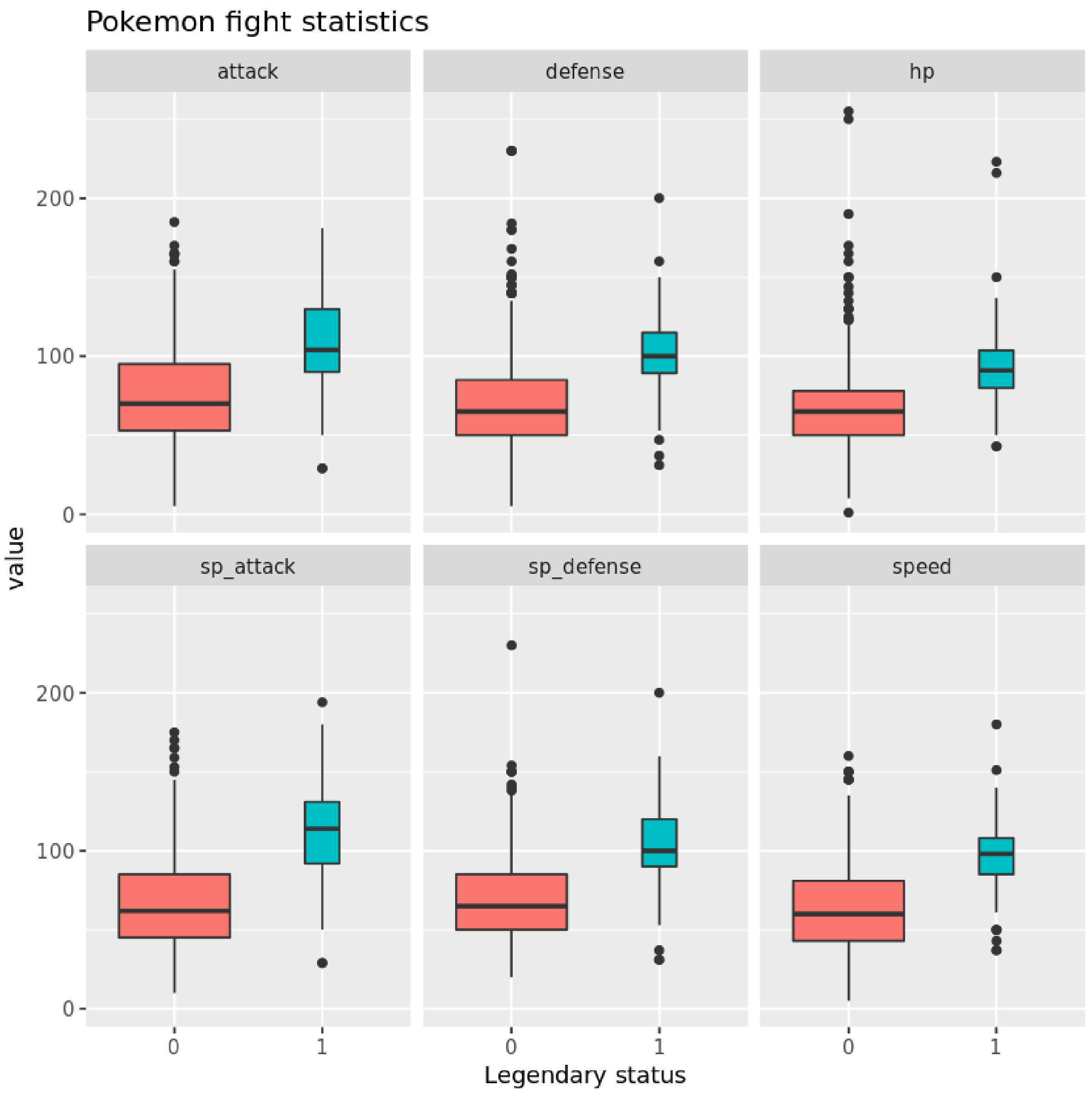
There are clear differences between Pokémon types in their relation to legendary status. While more than 30% of flying and psychic Pokémon are legendary, there is no such thing as a legendary poison or fighting Pokémon!

Before fitting the model, we will consider the influence of a Pokémon's fighter stats (`attack`, `defense`, etc.) on its status. Rather than considering each stat in isolation, we will produce a boxplot for all of them simultaneously using the `facet_wrap()` function.

In [102]:
```
# Prepare the data
legend_by_stats <- pokedex  %>%
  select(is_legendary, attack, sp_attack, defense, sp_defense, hp, speed)  %>%
  gather(key = fght_stats , value = value , -is_legendary)

# Prepare the plot
legend_by_stats_plot <- legend_by_stats %>%
 ggplot(aes(x = is_legendary, y = value, fill = is_legendary)) +
 geom_boxplot(varwidth = TRUE) +
 facet_wrap(~fght_stats) +
 labs(title = "Pokemon fight statistics",
        x = "Legendary status") +
 guides(fill = F)

# Print the plot
legend_by_stats_plot
```



In [103]:
```
soln_legend_by_stats <- soln_pokedex  %>%
  select(is_legendary, attack, sp_attack, defense, sp_defense, hp, speed)  %>%
  gather(key = "fght_stats", value = "value", -is_legendary)

soln_legend_by_stats_plot <- soln_legend_by_stats %>%
 ggplot(aes(x = is_legendary, y = value, fill = is_legendary)) +
 geom_boxplot(varwidth = TRUE) +
 facet_wrap(~fght_stats) +
 labs(title = "Pokemon fight statistics",
        x = "Legendary status") +
 guides(fill = FALSE)

run_tests({

  test_that("the data preparation is correct", {

      expect_equal(colnames(legend_by_stats)[1], colnames(soln_legend_by_stats
)[1],
                       info = "Did you exclude `is_legendary` from the `gather
()` process by placing it after the `-` sign?")

      expect_equal(colnames(legend_by_stats)[2:3], colnames(soln_legend_by_sta
ts)[2:3],
                      info = "Did you set `\"fght_stats\"` as the `key` and `
\"value\"` as the `value` when using `gather()`?")

  })

    test_that("the plot preparation is correct", {

      expect_equal(legend_by_stats_plot$facet$params$facets, soln_legend_by_st
ats_plot$facet$params$facets,
                        info = "Did you use the `facet_wrap()` function to fac
et by `fght_stats?`")


  })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 20.769 0.292 595.654 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

6. Create a training/test split

As we might expect, legendary Pokémon outshine their ordinary counterparts in all fighter stats. Although we haven't formally tested a difference in means, the boxplots suggest a significant difference with respect to all six variables. Nonetheless, there are a number of outliers in each case, meaning that some legendary Pokémon are anomalously weak.

We have now explored all of the predictor variables we will use to explain what makes a Pokémon legendary. Before fitting our model, we will split the `pokedex` into a training set (`pokedex_train`) and a test set (`pokedex_test`). This will allow us to test the model on unseen data.

In [104]:
```
# Set seed for reproducibility
set.seed(1234)
# Save number of rows in dataset
n = nrow(pokedex)

# Generate 60% sample of rows
sample_rows <- sample(n,n*0.6)

# Create training set
pokedex_train <- pokedex  %>%
  filter(row_number() %in% sample_rows)

# Create test set
pokedex_test <- pokedex  %>%
  filter(!row_number() %in% sample_rows)
```

In [105]:
```
set.seed(1234)

soln_n <- nrow(soln_pokedex)
soln_sample_rows <- sample(soln_n, 0.6 * soln_n)

soln_pokedex_train <- soln_pokedex  %>%
  filter(row_number() %in% soln_sample_rows)

soln_pokedex_test <- soln_pokedex  %>%
  filter(!row_number() %in% soln_sample_rows)

run_tests({

  test_that("the seed is correct", {
      expect_equal(sample_rows, soln_sample_rows,
                       info = "Did you set the seed to `1234`?")
  })

    test_that("the value for `n` is correct", {
        expect_equal(n, soln_n,
                        info = "The value for `n` is incorrect. Did you save i
t as the number of rows in `pokedex`?")
  })

    test_that("the sample is correct", {
        expect_equal(length(sample_rows), length(soln_sample_rows),
                        info = "Did you generate a 60% sample of rows?")
    })

    test_that("the training/test sets are correct", {
        expect_equal(pokedex_train, soln_pokedex_train,
                         info = "The training set is incorrect. Did you filter
the `pokedex` using `sample_rows`?")

        expect_equal(pokedex_test, soln_pokedex_test,
                         info = "The test set is incorrect. Did you filter the
`pokedex` using `sample_rows`?")
    })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 20.819 0.292 595.704 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

7. Fit a decision tree

Now we have our training and test sets, we can go about building our classifier. But before we fit a random forest, we will fit a simple **classification decision tree**. This will give us a baseline fit against which to compare the results of the random forest, as well as an informative graphical representation of the model.

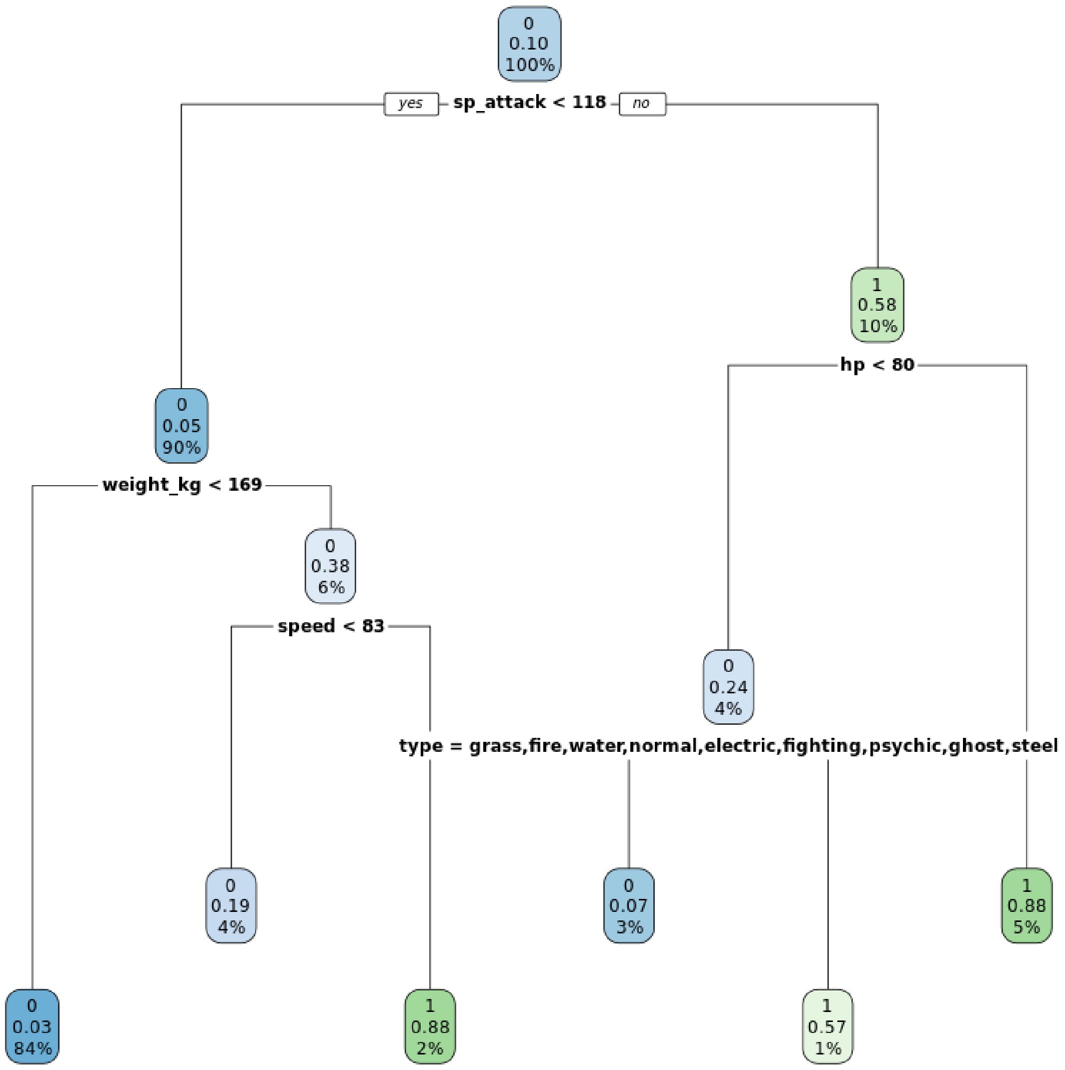
Here, and also in the random forest, we will omit incomplete observations by setting the `na.action` argument to `na.omit`. This will remove a few Pokémon with missing values for `height_m` and `weight_kg` from the training set. Remember the warning messages when we made our height/weight plot in Task 3? These are the Pokémon to blame!

In [106]:
```
# Load packages and set seed
library(rpart)
library(rpart.plot)

set.seed(1234)

# Fit decision tree
model_tree <- rpart(is_legendary ~ attack + defense + height_m +
                    hp + sp_attack + sp_defense + speed + type + weight_kg,
                      data = pokedex_train,
                      method = "class",
                      na.action = na.omit)

# Plot decision tree
rpart.plot(model_tree)
```



In [107]:
```
set.seed(1234)

soln_model_tree <- rpart(is_legendary ~ attack + defense + height_m +
                    hp + sp_attack + sp_defense + speed + type + weight_kg,
                       data = soln_pokedex_train,
                       method = "class",
                       na.action = na.omit)

run_tests({

  test_that("packages are loaded", {
      expect_true("rpart" %in% .packages(), info = "Did you load the `rpart` p
ackage?")
      expect_true("rpart.plot" %in% .packages(), info = "Did you load the `rpa
rt` package?")
  })

    test_that("the decision tree is correct", {

        expect_true(model_tree[["call"]][["data"]] == "pokedex_train",
                         info = "Did you train the model on `pokedex_train`?")

        expect_equal(model_tree[["call"]][["na.action"]], soln_model_tree[["ca
ll"]][["na.action"]],
                         info = "Did you set `na.action` to `na.omit`?")

    })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 21.058 0.292 595.943 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

8. Fit a random forest

Each node of the tree shows the predicted class, the probability of being legendary, and the percentage of Pokémon in that node. The bottom-left node, for example – for those with `sp_attack < 118` and `weight_kg < 169` – represents 84% of Pokémon in the training set, predicting that each only has a 3% chance of being legendary.

Decision trees place the most important variables at the top and exclude any they don't find to be useful. In this case, `sp_attack` occupies node 1 while `attack`, `defense`, `sp_defense` and `height_m` are all excluded.

However, decision trees are unstable and sensitive to small variations in the data. It therefore makes sense to fit a **random forest** – an ensemble method that averages over several decision trees all at once. This should give us a more robust model that classifies Pokémon with greater accuracy.

In [108]:
```
# Load package and set seed
library(randomForest)
set.seed(1234)

# Fit random forest
model_forest <- randomForest(is_legendary ~ attack + defense + height_m +
                         hp + sp_attack + sp_defense + speed + type + weight_k
g,
                         data = pokedex_train,
                         importance = TRUE,
                         na.action = na.omit)

# Print model output
model_forest
```

```
Call:
 randomForest(formula = is_legendary ~ attack + defense + height_m +      hp
+ sp_attack + sp_defense + speed + type + weight_kg, data = pokedex_train,
importance = TRUE, na.action = na.omit)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 7.05%
Confusion matrix:
    0  1 class.error
0 411  9  0.02142857
1  24 24  0.50000000
```

In [109]:
```
random_state <- .Random.seed
set.seed(1234)
soln_random_state <- .Random.seed

soln_model_forest <- randomForest(is_legendary ~ attack + defense + height_m +
                         hp + sp_attack + sp_defense + speed + type + weight_k
g,
                         data = soln_pokedex_train,
                         importance = TRUE,
                         na.action = na.omit)

run_tests({

  test_that("packages are loaded", {
      expect_true("randomForest" %in% .packages(), info = "Did you load the `r
andomForest` package?")
  })

    test_that("the random forest is correct", {

        expect_true(model_forest[["call"]][["data"]] == "pokedex_train",
                         info = "Did you train the model on `pokedex_train`?")

        expect_equal(model_forest[["call"]][["na.action"]], soln_model_forest
[["call"]][["na.action"]],
                         info = "Did you set `na.action` to `na.omit`?")

    })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 21.425 0.304 596.321 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

9. Assess model fit

Looking at the model output, we can see that the random forest has an out-of-bag (OOB) error of 7.48%, which isn't bad by most accounts. However, since there are **24 true positives** and **24 false negatives**, the model only has a recall of 50%, which means that it struggles to successfully retrieve every legendary Pokémon in the dataset.

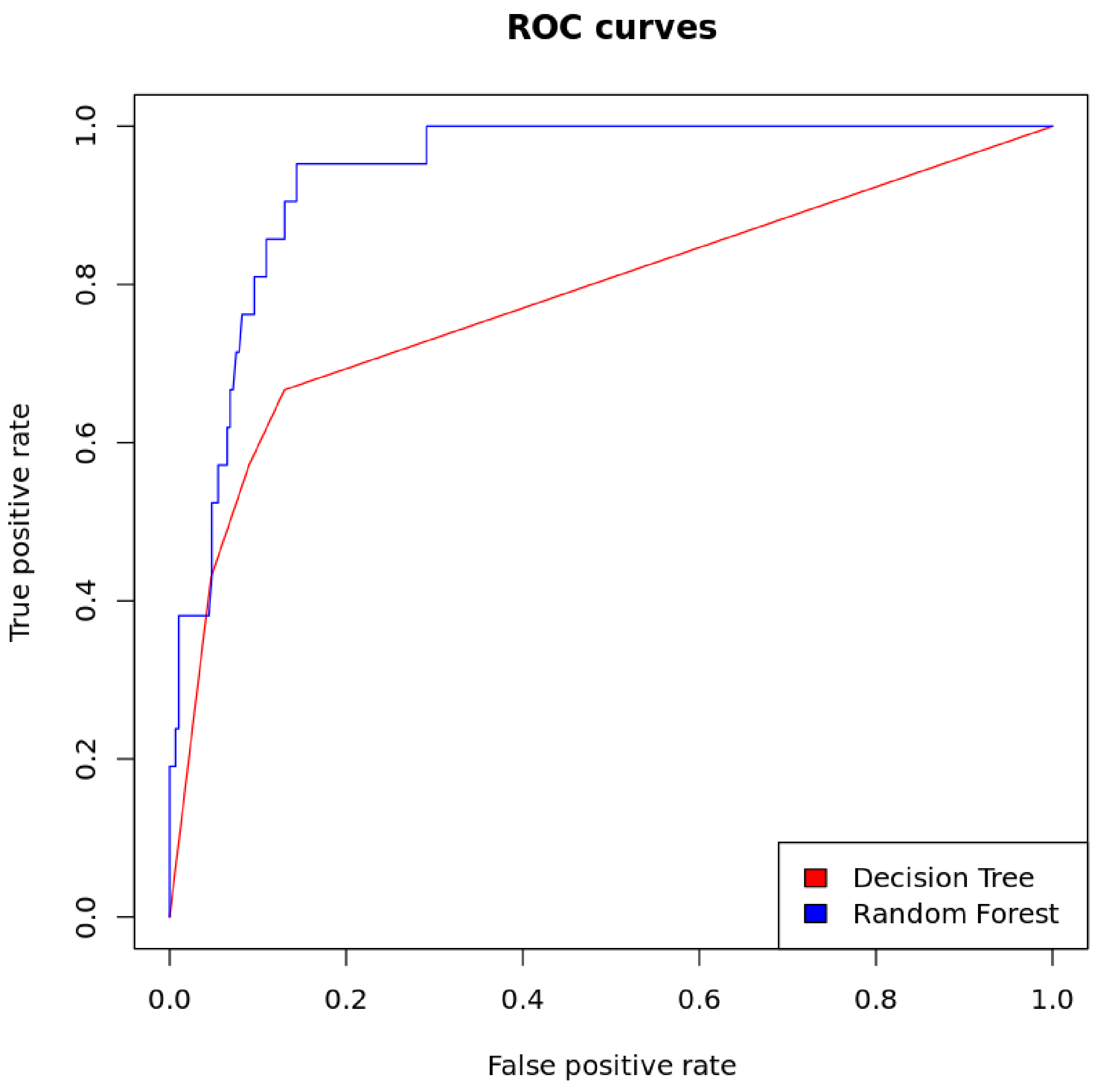
In order to allow direct comparison with the decision tree, we will plot the **ROC curves** for both models using the `ROCR` package, which will visualize their true positive rate (TPR) and false positive rate (FPR) respectively. The closer the curve is to the top left of the plot, the higher the area under the curve (AUC) and the better the model.

In [110]:
```
# Load the ROCR package
library(ROCR)

# Create prediction and performance objects for the decision tree
probs_tree <- predict(model_tree, pokedex_test, type = "prob")
pred_tree <- prediction(probs_tree[,2], pokedex_test$is_legendary)
perf_tree <- performance(pred_tree, "tpr", "fpr")

# Create prediction and performance objects for the random forest
probs_forest <- predict(model_forest, pokedex_test, type = "prob")
pred_forest <- prediction(probs_forest[,2], pokedex_test$is_legendary)
perf_forest <- performance(pred_forest, "tpr", "fpr")

# Plot the ROC curves: first for the decision tree, then for the random forest
plot(perf_tree, col = "red", main = "ROC curves")
plot(perf_forest, add = TRUE, col = "blue")
legend(x = "bottomright",  legend = c("Decision Tree", "Random Forest"), fill
= c("red", "blue"))
```



In [111]:

```
soln_probs_forest <- predict(soln_model_forest, soln_pokedex_test, type = "pro
b")
soln_pred_forest <- prediction(soln_probs_forest[,2], soln_pokedex_test$is_leg
endary)
soln_perf_forest <- performance(soln_pred_forest, "tpr", "fpr")

run_tests({

  test_that("packages are loaded", {
      expect_true("ROCR" %in% .packages(), info = "Did you load the `ROCR` pac
kage?")
  })

    test_that("the prediction and performance objects are correct", {

        expect_equal(probs_forest, soln_probs_forest,
                         info = "The `probs_forest` object is incorrect. Did y
ou remember to use `model_forest` rather than `model_tree`?")

        expect_equal(pred_forest, soln_pred_forest,
                         info = "The `pred_forest` object is incorrect. Did yo
u remember to use `probs_forest` rather than `probs_tree`?")

        expect_equal(perf_forest, soln_perf_forest,
                         info = "The `perf_forest` object is incorrect. Did yo
u remember to use `pred_forest` rather than `pred_tree`?")

  })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 21.548 0.304 596.443 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```

10. Analyze variable importance

It's clear from the ROC curves that the random forest is a substantially better model, boasting an AUC (not calculated above) of 91% versus the decision tree's 78%. When calculating variable importance, it makes sense to do so with the best model available, so we'll use the random forest for the final part of our analysis.

Note that a random forest returns two measures of variable importance:

- `MeanDecreaseAccuracy` – how much the model accuracy suffers if you leave out a particular variable
- `MeanDecreaseGini` – the degree to which a variable improves the probability of an observation being classified one way or another (i.e. 'node purity').

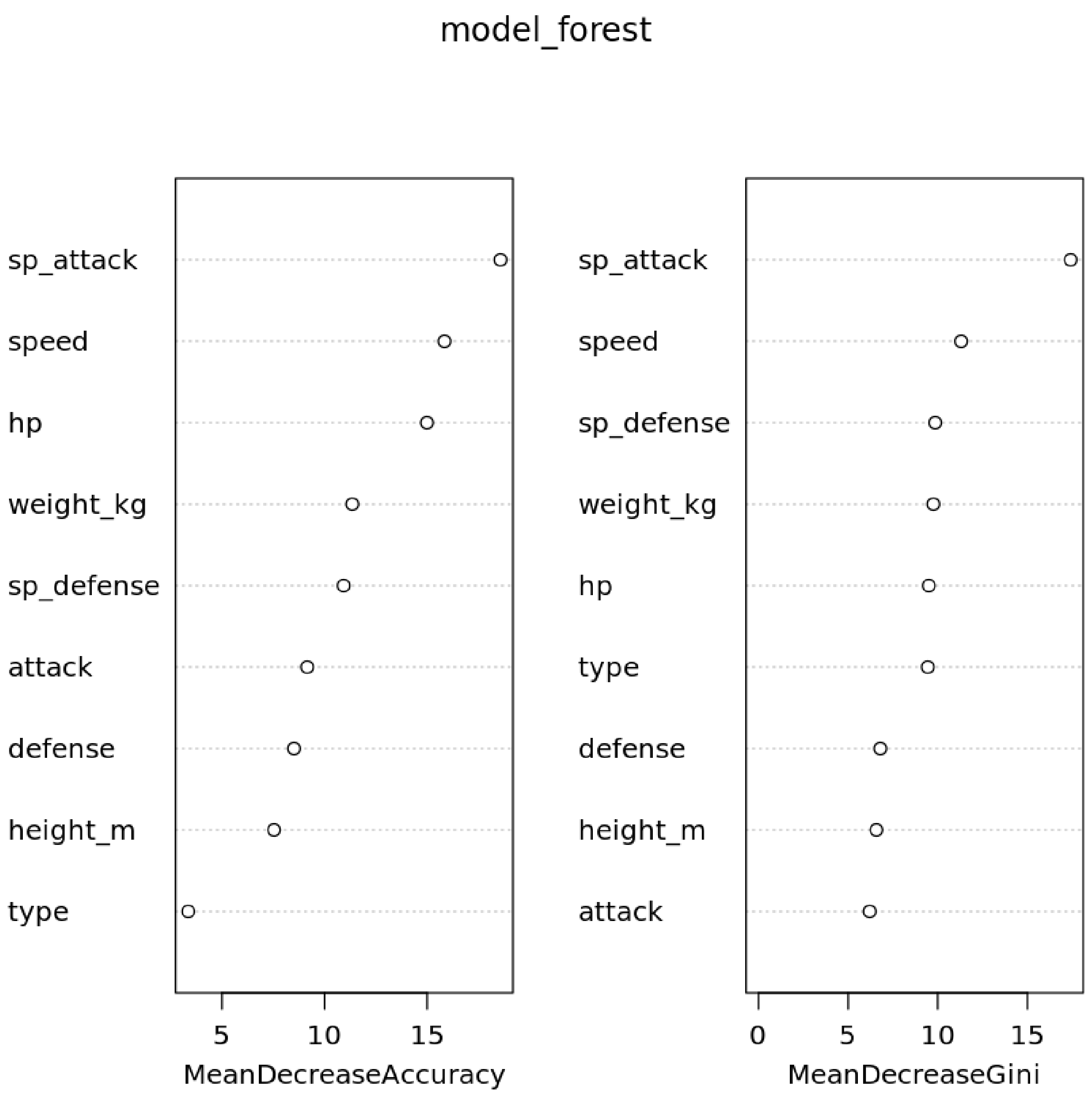
Together, these two measures will allow us to answer our original research question – what makes a Pokémon legendary?

In [112]:
```
# Print variable importance measures
importance_forest <- importance(model_forest)
importance_forest

# Create a dotchart of variable importance
varImpPlot_forest <- varImpPlot(model_forest)
varImpPlot_forest
```

	0	1	MeanDecreaseAccuracy	MeanDecreaseGini
attack	5.5880374	8.5275682	9.145018	6.203285
defense	0.6724663	12.2083778	8.506461	6.793236
height_m	4.1403952	6.7509476	7.533695	6.579863
hp	2.7180639	19.2406506	14.980599	9.499892
sp_attack	6.3947972	22.5417527	18.567857	17.415111
sp_defense	-0.8501697	15.1682650	10.919369	9.857089
speed	2.2799561	21.0596369	15.838444	11.308568
type	3.3473633	0.7726262	3.347259	9.440971
weight_kg	8.8857876	6.6917782	11.343187	9.760121

	MeanDecreaseAccuracy	MeanDecreaseGini
attack	9.145018	6.203285
defense	8.506461	6.793236
height_m	7.533695	6.579863
hp	14.980599	9.499892
sp_attack	18.567857	17.415111
sp_defense	10.919369	9.857089
speed	15.838444	11.308568
type	3.347259	9.440971
weight_kg	11.343187	9.760121


model_forest
sp_attack
speed
hp
weight_kg
sp_defense
attack
defense
height_m
type
5
10
15
MeanDecreaseAccuracy
sp_attack
speed
sp_defense
weight_kg
hp
type
defense
height_m
attack
0
5
10
15
MeanDecreaseGini

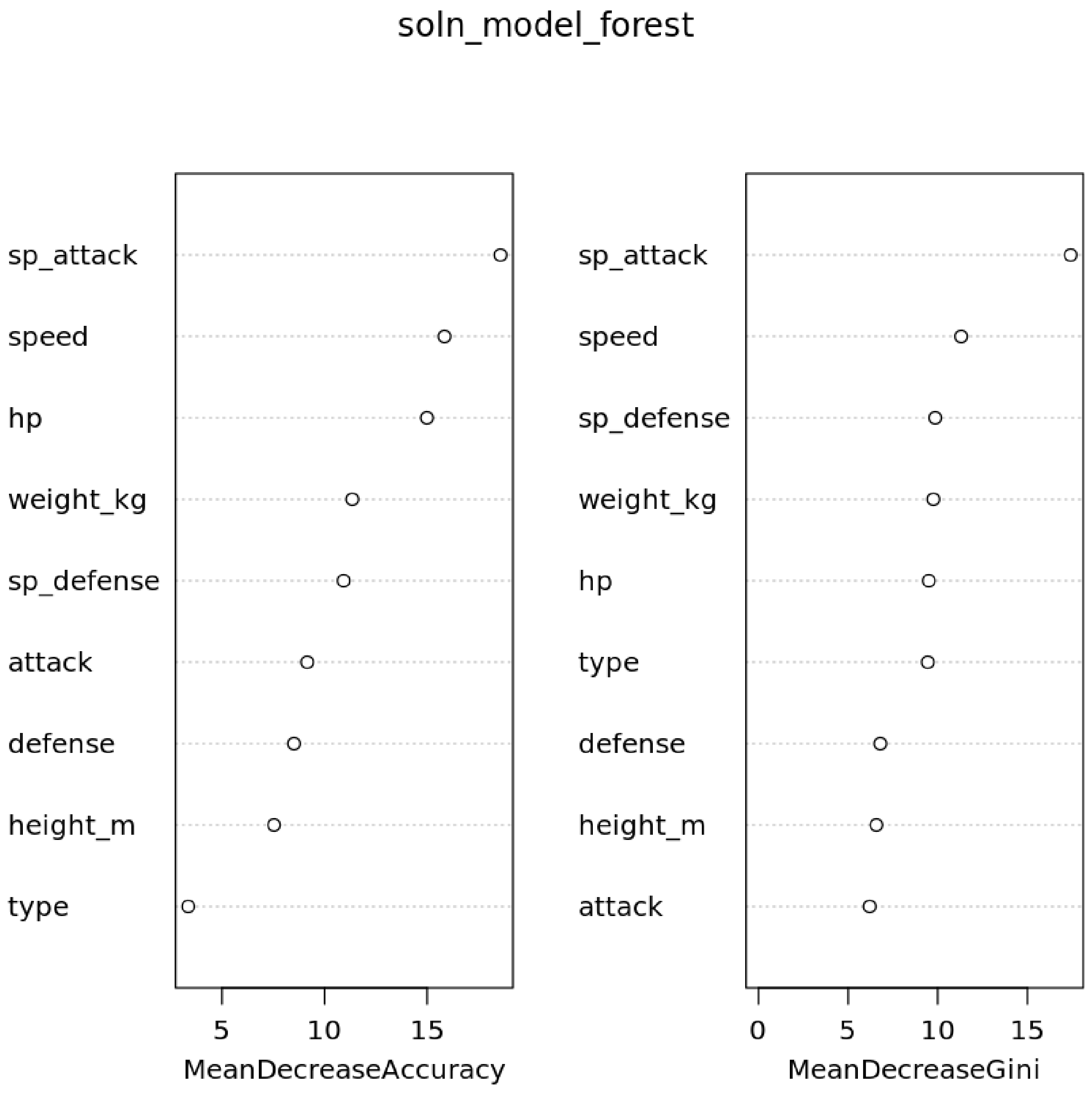
In [113]:
```
soln_importance_forest <- importance(soln_model_forest)
soln_varImpPlot_forest <- varImpPlot(soln_model_forest)

run_tests({

  test_that("the `importance` and `varImpPlot` objects are correct", {
      expect_equal(importance_forest, soln_importance_forest,
                       info = "Did you pass the random forest model to the `im
portance()` function?")
      expect_equal(varImpPlot_forest, soln_varImpPlot_forest,
                       info = "Did you pass the random forest model to the `va
rImpPlot()` function?")
  })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 21.665 0.304 596.559 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```



11. Conclusion

According to the variable importance plot, `sp_attack` is the most important factor in determining whether or not a Pokémon is legendary, followed by `speed`. The plot doesn't tell us whether the variables have a positive or a negative effect, but we know from our exploratory analysis that the relationship is generally positive. We therefore conclude that legendary Pokémon are characterized primarily by the power of their special attacks and secondarily by their speediness, while also exhibiting higher fighting abilities across the board.

Congratulations on completing your research into legendary Pokémon – Professor Oak is excited to share the findings! To finish, we'll answer a few of his questions about the variable importance results.

In [114]:
```
# According to the MeanDecreaseAccuracy plot:

# Q1. Is the `attack` or `defense` variable more important?
answer1 <- "attack"

# Q2. Is the `weight_kg` or `height_m` variable more important?
answer2 <- "weight_kg"

# According to the MeanDecreaseGini plot:

# Q3. Is the `attack` or `defense` variable more important?
answer3 <- "defense"

# Q4. Is the `weight_kg` or `height_m` variable more important?
answer4 <- "weight_kg"
```

In [115]:
```
soln_answer1 <- "attack"
soln_answer2 <- "weight_kg"
soln_answer3 <- "defense"
soln_answer4 <- "weight_kg"

run_tests({

  test_that("the answers are correct", {

      expect_equal(answer1, soln_answer1,
                      info = "The answer to Q1 is wrong. Did you check which v
ariable was higher in the `MeanDecreaseAccuracy` plot?")

      expect_equal(answer2, soln_answer2,
                      info = "The answer to Q2 is wrong. Did you check which v
ariable was higher in the `MeanDecreaseAccuracy` plot?")

      expect_equal(answer3, soln_answer3,
                      info = "The answer to Q3 is wrong. Did you check which v
ariable was higher in the `MeanDecreaseGini` plot?")

      expect_equal(answer4, soln_answer4,
                      info = "The answer to Q4 is wrong. Did you check which v
ariable was higher in the `MeanDecreaseGini` plot?")

  })

})
```

```
<ProjectReporter>
  Inherits from: <ListReporter>
  Public:
    .context: NULL
    .end_context: function (context)
    .start_context: function (context)
    add_result: function (context, test, result)
    all_tests: environment
    cat_line: function (...)
    cat_tight: function (...)
    clone: function (deep = FALSE)
    current_expectations: environment
    current_file: some name
    current_start_time: 21.761 0.304 596.656 0.004 0.001
    dump_test: function (test)
    end_context: function (context)
    end_reporter: function ()
    end_test: function (context, test)
    get_results: function ()
    initialize: function (...)
    is_full: function ()
    out: 3
    results: environment
    rule: function (...)
    start_context: function (context)
    start_file: function (name)
    start_reporter: function ()
    start_test: function (context, test)
```