Group 006-26 Project Report

Linda Han, Shaqed Orr, Eric Zhang, Prabhjot Singh

Introduction:

The Iris flower got its name from the goddess of rainbows in Acient Greek mythology. As suggested by this name origin, there are over 260 Iris species which can be found in diverse colours, shapes and sizes. For the purpose of our project, however, we are only interested in 3 specific classes of these perennial plants: Versicolor, Setosa, and Virginica.

In this report, we try to answer the question: given an Iris flower belonging to one of these three classes, how do we predict which class they are in?

We will be using the `Iris` data set found on https://archive.ics.uci.edu/ml/datasets/iris to build a classifier.

This data set contains 150 samples of data with 3 different classes. Each class contains 50 samples, and four features: sepal length, sepal width, petal length, and petal width. Ultimately, we are trying to predict the class of Iris flowers by these four distinct features using the K-Nearest Neighbor classification method.

Source: https://www.almanac.com/plant/irises

Methods and Results:

Preliminary data analysis:

First, we load in all the necessary libraries.

In [2]:
```r
library(tidyverse)
library(repr)
library(tidymodels)
library(ggplot2)

options(repr.matrix.max.rows = 6) # this lists only 6 rows when we try to display the dataset
```

```
── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──

✓ ggplot2 3.3.2     ✓ purrr   0.3.4
✓ tibble  3.0.3     ✓ dplyr   1.0.2
✓ tidyr   1.1.2     ✓ stringr 1.4.0
✓ readr   1.3.1     ✓ forcats 0.5.0

Warning message:
“package ‘ggplot2’ was built under R version 4.0.1”
Warning message:
“package ‘tibble’ was built under R version 4.0.2”
Warning message:
“package ‘tidyr’ was built under R version 4.0.2”
```

```
Warning message:
“package ‘dplyr’ was built under R version 4.0.2”
── Conflicts ──────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()

Warning message:
“package ‘tidymodels’ was built under R version 4.0.2”
── Attaching packages ────────────────────────────────────── tidymodels 0.1.1 ──

✓ broom     0.7.0      ✓ recipes   0.1.13
✓ dials     0.0.9      ✓ rsample   0.0.7
✓ infer     0.5.4      ✓ tune      0.1.1
✓ modeldata 0.0.2      ✓ workflows 0.2.0
✓ parsnip   0.1.3      ✓ yardstick 0.0.7

Warning message:
“package ‘broom’ was built under R version 4.0.2”
Warning message:
“package ‘dials’ was built under R version 4.0.2”
Warning message:
“package ‘infer’ was built under R version 4.0.3”
Warning message:
“package ‘modeldata’ was built under R version 4.0.1”
Warning message:
“package ‘parsnip’ was built under R version 4.0.2”
Warning message:
“package ‘recipes’ was built under R version 4.0.1”
Warning message:
“package ‘tune’ was built under R version 4.0.2”
Warning message:
“package ‘workflows’ was built under R version 4.0.2”
Warning message:
“package ‘yardstick’ was built under R version 4.0.2”
── Conflicts ─────────────────────────────────────────── tidymodels_conflicts() ──
✖ scales::discard() masks purrr::discard()
✖ dplyr::filter()   masks stats::filter()
✖ recipes::fixed()  masks stringr::fixed()
✖ dplyr::lag()      masks stats::lag()
✖ yardstick::spec() masks readr::spec()
✖ recipes::step()   masks stats::step()
```

Reading and cleaning the data

1) We read the `Iris` dataset from the original source on the web (UCI Machine Learning Repository) using the `read_csv` function.

2) We added column names to reflect each of the attributes and mutated the `class` column such that it becomes a factor using `as_factor()`.

We can now say that our dataset is **tidy** as there is a single Iris observation on every row, each column is a single variable (either a measurement of the Iris flower or the class it belongs to), and each cell holds a single value.

In [3]:
```
iris_col <- c("sepal_length_cm", "sepal_width_cm", "petal_length_cm", "petal_width_cm", "class")
iris <- read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data", col_names= iris_col) %>%
        mutate(class = as_factor(class))

iris
```

```
Parsed with column specification:
cols(
  sepal_length_cm = col_double(),
  sepal_width_cm = col_double(),
  petal_length_cm = col_double(),
  petal_width_cm = col_double(),
  class = col_character()
)
```

A spec_tbl_df: 150 × 5

sepal_length_cm	sepal_width_cm	petal_length_cm	petal_width_cm	class
<dbl>	<dbl>	<dbl>	<dbl>	<fct>
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa

sepal_length_cm	sepal_width_cm	petal_length_cm	petal_width_cm	class
<dbl>	<dbl>	<dbl>	<dbl>	<fct>
⋮	⋮	⋮	⋮	⋮
6.5	3.0	5.2	2.0	Iris-virginica
6.2	3.4	5.4	2.3	Iris-virginica
5.9	3.0	5.1	1.8	Iris-virginica

Splitting the Data

In order to ensure the reproducibility of our results, we use the `set.seed()` function with a randomly chosen initial value.

There are 150 observations in total and we see from `Figure 0.1` below that there are 50 observations from each Iris class.

In [4]:
```
# Summarizes the number of observations in each class
# (Iris-setosa, Iris-versicolor, or Iris-virginica)
count <- iris %>%
        count(class)

'Figure 0.1'
count
```

'Figure 0.1'

A spec_tbl_df: 3 × 2

class	n
<fct>	<int>
Iris-setosa	50
Iris-versicolor	50
Iris-virginica	50

Our goal is to have enough training data points for building an accurate classifier and enough testing data points for making an accurate assessment of the model's performance. Since the proportion of each class is equally represented and the data size isn't large, we choose to partition the data into a 80% training and 20% testing set. This gives us 123 data points for training and 27 data points for testing, which seems like a reasonable partition.

In [5]:
```
set.seed(777)

iris_split <- initial_split(iris, prop = 0.80, strata = class)
iris_train <- training(iris_split)
iris_test <- testing(iris_split)
```

Summary of the training data

Using only the training data, we summarize the data into 2 tables and count the number of rows with missing values to check that all of our rows contain data that will contribute to building the classifier.

1. We summarize the average value of each column using `summarize()` and `across()`

In [6]:
```
iris_avg_size <- iris_train %>%
        summarize(across(sepal_length_cm:petal_width_cm, mean))

'Figure 0.2'
iris_avg_size
```

'Figure 0.2'

A tibble: 1 × 4

sepal_length_cm	sepal_width_cm	petal_length_cm	petal_width_cm
<dbl>	<dbl>	<dbl>	<dbl>
5.854472	3.06748	3.771545	1.198374

1. We summarize the number of observations in each class (Iris-setosa, Iris-versicolor, or Iris-virginica) and observe that there are an equal number of data points ($n = 41$) in each class.

In [7]:
```
iris_class_count <- iris_train %>%
        count(class)

'Table 0.3'
iris_class_count
```

'Table 0.3'

A tibble: 3 × 2

class	n
<fct>	<int>
Iris-setosa	41
Iris-versicolor	41
Iris-virginica	41

1. We count the number of missing rows using `is.na()` and confirmed that all rows in the training dataset have values and will all make meaningful contribution to the building of the classifier.

In [8]:
```
sum(is.na(iris_train))
```

0

Visualization of the training data

Using only the training data, we conduct an exploratory data analysis by visualizing the data with two scatterplots.

For the first graph, we use `ggplot()` to plot the predictors `sepal_width_cm` on the y-axis against `sepal_length_cm` on the x-axis. We also set `colour = "Class"` to mark each observation according to their class. We chose this pair of predictors as their name suggests that they are both measurements of the flower sepal and could therefore be correlated.

To improve readability, we also use the `options()` and `theme()` functions to modify the size of the plot and labels.

In [9]:
```
# Graph 1
options(repr.plot.width = 8, repr.plot.height = 6)

iris_plot_sepal <- ggplot(data = iris_train,
                          aes(x = sepal_length_cm, y = sepal_width_cm , colour = class )) +
                geom_point() +
                labs(x = "Sepal length (cm)", y = "Sepal width (cm)",
                     colour = "Class", caption = "Figure 0.4") +
                ggtitle("Sepal Width vs Sepal Length") +
                theme(text = element_text(size = 20))

iris_plot_sepal
```

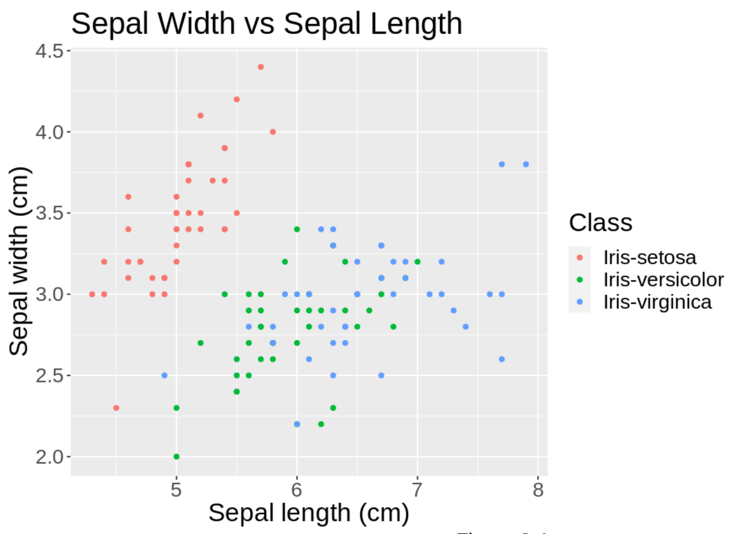

Figure 0.4

From the graph, we can see that the red data points representing the `Iris-setosa` class form a distinct cluster. Compared to data points from the `Iris-versicolor` and `Iris-virginica` class, the graph suggests that Iris Setosas generally have shorter sepal lengths and longer sepal widths. Meanwhile, the data points for `Iris-versicolor` and `Iris-virginica` seem to be blended together, meaning that it might be difficult to distinguish them based on sepal measurements.

For the second graph, we repeat the same things we did for Graph 1 except for changing the two predictors to `petal_length_cm` and `petal_width_cm` .

In [10]:
```r
# Graph 2
iris_plot_petal <- ggplot(data = iris_train,
                          aes(x = petal_length_cm, y = petal_width_cm , colour = class )) +
                    geom_point() +
                    labs(x = "Petal length (cm)", y = "Petal width (cm)",
                         colour = "Class", caption = "Figure 0.5") +
                    ggtitle("Petal Width vs Petal Length") +
                    theme(text = element_text(size = 20))

iris_plot_petal
```

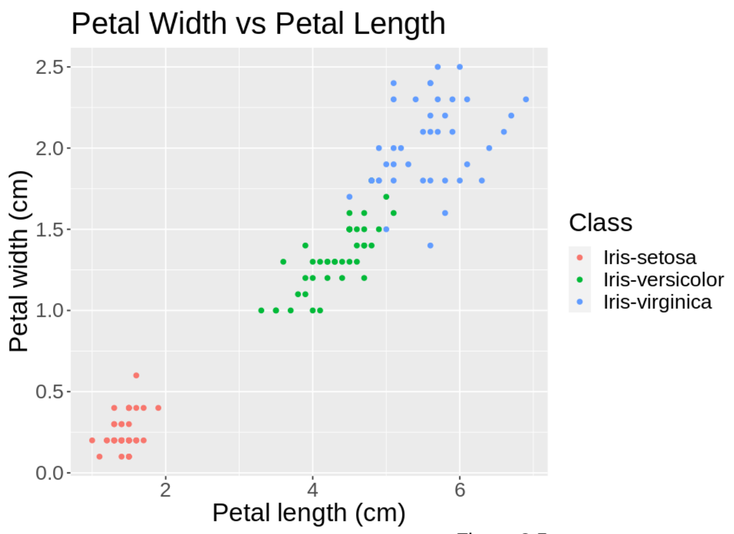

Figure 0.5

Based on the colors, we observe 3 regions and their distinctiveness suggest that we can classify Irises based on their petal length and petal width.

Petal width (in ascending order): `Iris-setosa < Iris-versicolor < Iris-virginica`

Petal length (in ascending order): `Iris-setosa < Iris-versicolor < Iris-virginica`

From the ranking above, we can see that `Iris-setosa` has the shortest petal width and length, `Iris-virginica` has the longest petal width and length, and `Iris-versicolor` is somewhere in between.

Model Training Phase (Building The K-Nearest Neighbour Classifier)

1. Recipe

We choose to use all non-categorical variables as predictors because all of them depict crucial measurements of the Iris flower.

Since the K-nearest neighbors algorithm is sensitive to the scale of the predictors, we use `step_center()` and `step_scale()` with the `all_predictors()` argument passed in to standardize our data.

In [11]:
```
iris_recipe <- recipe(class ~ ., data = iris_train) %>%
            step_center(all_predictors()) %>%
            step_scale(all_predictors())

iris_recipe
```

```
Data Recipe

Inputs:

      role #variables
   outcome          1
 predictor          4

Operations:

Centering for all_predictors()
Scaling for all_predictors()
```

To visualize the scaled `iris_train` data, we use `prep()` and `bake()` to create a new data frame.

In [12]:
```
iris_scaled <- iris_recipe %>%
                prep() %>%
                bake(iris_train)

'Figure 1.1'
iris_scaled
```

'Figure 1.1'

A tibble: 123 × 5

sepal_length_cm	sepal_width_cm	petal_length_cm	petal_width_cm	class
<dbl>	<dbl>	<dbl>	<dbl>	<fct>
-1.157990	-0.15110630	-1.329810	-1.308899	Iris-setosa
-1.400635	0.29675092	-1.385883	-1.308899	Iris-setosa
-1.521958	0.07282231	-1.273736	-1.308899	Iris-setosa
⋮	⋮	⋮	⋮	⋮
0.78317214	-0.1511063	0.8009859	1.0509564	Iris-virginica
0.41920423	0.7446081	0.9131331	1.4442657	Iris-virginica
0.05523632	-0.1511063	0.7449123	0.7887503	Iris-virginica

2. Model Specification

Now we create a nearest neighbour model for our training set using `nearest_neighbor()` and set the neighbors argument to `tune()` for finding the optimal K value.

As this is a KNN classification problem, we set the engine to `"kknn"` and the mode to `"classification"`.

In [13]:
```
iris_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = tune()) %>%
            set_engine("kknn") %>%
            set_mode("classification")
```

```
iris_spec
Main Arguments:
  neighbors = tune()
  weight_func = rectangular

Computational engine: kknn
```

3. Vfold

When it comes to choosing the number of folds for performing cross-validation, we need to take into account that having too few folds (such as 1,2) could result in a biased estimation of our model performance. On the other hand, having too many folds (such as 20) could increase the runtime and cause the overall process to be computationally expensive. Taking these facts into consideration, $K = 5$ folds seems to be a reasonable choice for our data set.

In [14]:
```
iris_vfold <- vfold_cv(iris_train, v = 5, strata = class)
```

4. Workflow

We create a workflow and use the `tune_grid()` function to try K values from 1 to 20 by passing in a tibble object that we created called `k_vals`.

In [15]:
```
k_vals <- tibble(neighbors = seq(from=1, to=20, by=1))

iris_tune_workflow <- workflow() %>%
    add_recipe(iris_recipe) %>%
    add_model(iris_spec) %>%
    tune_grid(resamples = iris_vfold, grid = k_vals)
```

5. Collect Metrics

From the `iris_tune_workflow`, we extract the metrics by using the `collect_metrics()` function. We also filter for `accuracy` from the `.metric` column. However, this gives us a table of unordered rows. To see the `neighbors` value that gives us the highest accuracy, we use `arrange()` with `desc()` to arrange the rows by their `mean` column in descending order.

In [16]:
```
iris_metrics_tune <- iris_tune_workflow %>%
    collect_metrics() %>%
    filter(.metric == "accuracy") %>%
    arrange(desc(mean))

'Figure 1.2'
iris_metrics_tune
```

'Figure 1.2'

A tibble: 20 × 7

neighbors	.metric	.estimator	mean	n	std_err	.config
<dbl>	<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>
7	accuracy	multiclass	0.9685185	5	0.0141488	Model07
8	accuracy	multiclass	0.9685185	5	0.0141488	Model08
9	accuracy	multiclass	0.9685185	5	0.0141488	Model09
⋮	⋮	⋮	⋮	⋮	⋮	⋮
2	accuracy	multiclass	0.9435185	5	0.016064214	Model02
19	accuracy	multiclass	0.9435185	5	0.009189552	Model19
20	accuracy	multiclass	0.9435185	5	0.009189552	Model20

We see from the first three rows of `Figure 1.2` above that $K = 7, 8, 9$ all yield the highest accuracy of **96.85%**.

6. Plot Neighbors vs Accuracy Graph

To visualize `Figure 1.2`, we create a scatterplot with `neighbors` on the x-axis and `mean` on the y-axis. The `geom_line()` function connects all the points with lines and `scale_x_continuous()` is used to to adjust the x-axis.

In [17]:
```
neighbors_plot <- ggplot(iris_metrics_tune, aes(x=neighbors, y=mean)) +
    geom_point() +
    geom_line() +
    labs(x = "Neighbors", y = "Accuracy Estimate", caption="Figure 1.3") +
    ggtitle("Accuracy vs K") +
    theme(text = element_text(size = 20)) +
    scale_x_continuous(breaks = seq(0, 25, by = 2))

neighbors_plot
```

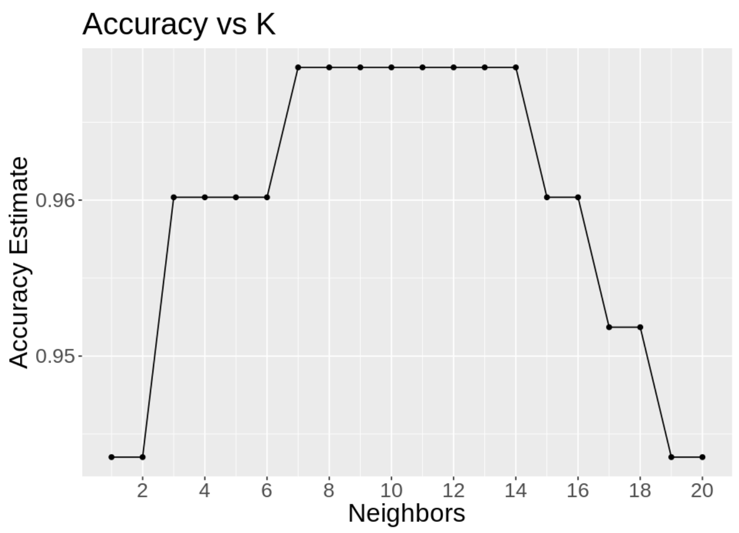

Figure 1.3

We want our model to have the right amount of sensitivity and avoid under or overfitting. From `Figure 1.3` above, we see that the accuracy estimate peaks from $K = 7$ neighbors to $K = 14$ neighbors. **We choose the optimal K to be** 7 as it is the point where the accuracy estimate first peaked.

7. Rebuilding Model Specification and Workflow with K = 7

With the optimal K found, we rebuild the model with the `neighbors` argument equal to 7 and name it `iris_spec_optimal`. We also create a new workflow object called `iris_fit` that uses the new model.

In [18]:
```
iris_spec_optimal <- nearest_neighbor(weight_func = "rectangular", neighbors = 7) %>%
            set_engine("kknn") %>%
            set_mode("classification")

iris_spec_optimal

iris_fit <- workflow() %>%
    add_recipe(iris_recipe) %>%
    add_model(iris_spec_optimal) %>%
    fit(iris_train)

iris_fit
```

```
K-Nearest Neighbor Model Specification (classification)

Main Arguments:
  neighbors = 7
  weight_func = rectangular

Computational engine: kknn
══ Workflow [trained] ══════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: nearest_neighbor()

── Preprocessor ────────────────────────────────────────────────────────────────
2 Recipe Steps

• step_center()
• step_scale()

── Model ───────────────────────────────────────────────────────────────────────

Call:
kknn::train.kknn(formula = ..y ~ ., data = data, ks = ~7, kernel = ~"rectangular")
```

```
Type of response variable: nominal
Minimal misclassification: 0.03252033
Best kernel: rectangular
Best k: 7
```

Model Testing Phase

1. Predict

We can now use the classifier to predict labels for our test dataset by using `predict()` and `bind_cols()` to bind the predicted class onto the test dataset for comparison.

In [19]:
```
iris_test_predictions <- predict(iris_fit, iris_test) %>%
    bind_cols(iris_test)

iris_test_predictions
```

A tibble: 27 × 6

.pred_class	sepal_length_cm	sepal_width_cm	petal_length_cm	petal_width_cm	class
<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<fct>
Iris-setosa	5.1	3.5	1.4	0.2	Iris-setosa
Iris-setosa	4.4	2.9	1.4	0.2	Iris-setosa
Iris-setosa	4.9	3.1	1.5	0.1	Iris-setosa
⋮	⋮	⋮	⋮	⋮	⋮
Iris-virginica	6.4	3.1	5.5	1.8	Iris-virginica
Iris-virginica	6.9	3.1	5.4	2.1	Iris-virginica
Iris-virginica	6.7	3.0	5.2	2.3	Iris-virginica

2. Assess accuracy of predictions

First, we use `metrics()` to check the accuracy of our predictions for the iris labels in the test set. We see that we have achieved 96% accuracy, which is the same as the accuracy estimate for our optimal $K = 7$ model during training phase.

In [20]:
```
iris_metrics_final <- iris_test_predictions %>%
    metrics(truth = class, estimate = .pred_class)

'Figure 2.1'
iris_metrics_final
```

'Figure 2.1'

A tibble: 2 × 3

.metric	.estimator	.estimate
<chr>	<chr>	<dbl>
accuracy	multiclass	0.9629630
kap	multiclass	0.9444444

We can also summarize the prediction outcomes by using a confusion matrix. By looking at the values in the matrix, we are provided detail on our model's prediction performance as we see that one `Iris-virginica` is mislabeled as an `Iris-versicolor` .

In [21]:
```
iris_conf_mat <- iris_test_predictions %>%
    conf_mat(truth = class, estimate = .pred_class)


'Figure 2.2'
iris_conf_mat
```

'Figure 2.2'

```
                 Truth
Prediction        Iris-setosa Iris-versicolor Iris-virginica
  Iris-setosa               9               0              0
```

```
Iris-versicolor          0               9               1
Iris-virginica           0               0               8
```

3. Visualization of Data Analysis Result

First, we created a data set containing the results from the confusion matrix model. Using semi_join(), group_by(), and summarize(), we created a data set showing the count of each class and the values that are close to our predicted outcomes.

We decide to visualize the performance of our classifier through a mosaic plot. Using `semi_join()` , we join the two tables `iris_test_predictions` and `iris_test` together. Followed by `group_by()` and `summarize()` , we create a data frame that is similar to a confusion matrix that shows the count of predicted outcomes for each class.

In [22]:
```
mos_data<- iris_test_predictions %>%
    semi_join(iris_test) %>%
    group_by(class, .pred_class)%>%
    summarize(n=n())

'Figure 2.3'
mos_data
```

```
Joining, by = c("sepal_length_cm", "sepal_width_cm", "petal_length_cm", "petal_width_cm", "class")

`summarise()` regrouping output by 'class' (override with `.groups` argument)
```

'Figure 2.3'

A grouped_df: 4 × 3

class	.pred_class	n
<fct>	<fct>	<int>
Iris-setosa	Iris-setosa	9
Iris-versicolor	Iris-versicolor	9
Iris-virginica	Iris-versicolor	1
Iris-virginica	Iris-virginica	8

Then, using `ggplot()` and `geom_bar(position="fill", stat="identity")` , we obtain a mosaic plot that shows the class prediction of Iris flowers (in percentage). We can see that all observations in the test data set belonging to the `Iris-setosa` or `Iris-virginica` class are correctly labeled, whereas one `Iris-versicolor` is falsely predicted as `Iris-virginica` .

The scatterplot graphs (`Figure 0.4` and `Figure 0.5`) in the "Preliminary Data Analysis" section both show that `Iris-virginica` and `Iris-versicolor` have similar sepal and petal measurements. Therefore, it is unsurprisingly for there to be cases where a data point has attributes that lie between the ranges of the two classes and gets mispredicted.

In [23]:
```
mos_plot <- ggplot(mos_data, aes(x=class, y=n, fill=.pred_class))+
        geom_bar(position="fill", stat="identity")+
        labs(x="Classes of Iris Flower", y="Percentage Value",
             fill="Iris Classes", caption="Figure 2.4") +
        ggtitle("Predictions of Iris Flowers") +
        theme(text = element_text(size = 15))

mos_plot
```

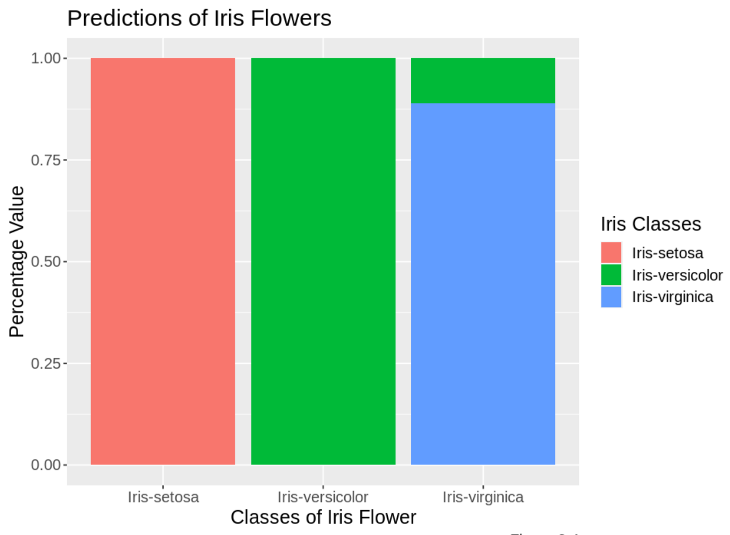

Figure 2.4

Discussion:

Summary of our findings:

Through our analysis, we saw a strong correlation between certain ranges of petal lengths and widths, and certain species of flowers.

The smallest petal widths (0cm to 0.75cm) and lengths (0cm to 2cm) corresponded to Setosa Irises, the medium length petal widths (1cm to 1.75 cm) and lengths (2.5cm to 5cm)corresponded to Versicolor Irises, and the largest petal widths (1.75cm to 2.5 cm) and lengths(5cm to 7cm) corresponded to virginica irises. *(Note: all values are approximated)*

However, there was no such clear division when we compared sepal width and length to the species of flower, since Versicolor Irises and Virginica Irises had generally similar sepal measurements.

We used the K-nearest neighbor Classifier model to predict the species of the flowers using all four variables (i.e. petal length, petal width, sepal length, and sepal width). Through our 5-fold cross-validation, we identified $K = 7$ to be the ideal number of nearest neighbors for the classifier to consider. Upon testing, we found that $K = 7$ accurately predicted the floral species 96.29% of the time.

Did we find what we expected?

Originally, we expected to see how specific variables influenced speciation differently, which we did! Our petal length vs petal width scatterplot had 3 distinct groups, emphasizing the importance of petal properties in species identification. Our sepal length vs sepal width scatterplot had one distinct group, setosa irises, and one less distinct group, suggesting that sepal length and width are more influential in identifying setosa irises than virginica and versicolor irises.

What impact could our findings have?

Identifying which factors are more or less influential to iris flower speciation can help with accurately assessing a specimen, in order to identify if it is an existing species or a newly identified one. By knowing which variables are more or less characteristic of certain species, it becomes easier to know which variables to focus on when trying to see if a new flower is an existing species or potentially a new one.

What future questions could this lead to?

This could lead to a number of different questions -- firstly, could the morphology of the flowers contribute to their lifespan? How can we apply similar classifiers to other flowers and plants? And, what other morphological characteristics of the flower can we consider in order to aid in species identification? Furthermore, the dataset has been instrumental in training many machine learning algorithms over time, such as the quantum neural network discussed in “Training a Quantum Neural Network” (Ricks & Ventura, 2003).

References:

Relevant Literature:

Armbruster, W. S. (2014). Floral specialization and angiosperm diversity: Phenotypic divergence, fitness trade-offs and realized pollination accuracy. AoB PLANTS, 6. https://doi.org/10.1093/aobpla/plu003

University of Hawai'i. (2022). Classification of Life. Classification of Life | manoa.hawaii.edu/ExploringOurFluidEarth. Retrieved April 5, 2022, from https://manoa.hawaii.edu/exploringourfluidearth/biological/what-alive/classification-life

Data Source:

Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.