# stat462_ass_3_2024

nki38 96851177

2024-09-27

```
knitr::opts_chunk$set(echo = TRUE)
library(reshape2)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(tree)
library(ISLR)
library(knitr)
library(class)
library(ggplot2)
library(stats)
library(ggplot2)
library(tidyr)
```

```
##
## Attaching package: 'tidyr'
```

```
## The following object is masked from 'package:reshape2':
##
##     smiths
```

```
library(randomForest)
```

```
## randomForest 4.7-1.2
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin

## The following object is masked from 'package:dplyr':
##
##     combine
```

```
library(rpart)
set.seed(123)
```

## Predicting the origin of Volcanic Rocks given present elements

### Preamble & setup

The Rmd doc assumes all data is in the same directory at the same level as the doc

Our goal, is to build some models for predicting the origin of rocks from our dataset that could apply to unseen data with good *performance*.

We will consider classification treese and K-Nearest Neighbours as methods for creating our classifier.

Firstly, we consider the following:

In order to create the *Okaitane* column on the unlabelled set, we will consider the data in our labelled set and implement methods for creating prediction models purely with that set, before finally comparing our predictive model with the ground truth at the end.

Firstly, some R specific setup

```
unlabelled_data <- read.csv('rocks_unlabelled.csv')
labelled_data <- read.csv('rocks.csv')
dim(unlabelled_data)
```

```
## [1] 30 12
```

```
dim(labelled_data)
```

```
## [1] 330  13
```

```
#We also remove the index column....
unlabelled_data <- unlabelled_data[, -which(names(unlabelled_data) == "X")]
labelled_data <- labelled_data[, -which(names(labelled_data) == "X")]
labelled_data$okataina = as.factor(labelled_data$okataina)
```

We split our data into train, test (seed in first chunk)

```
n <- nrow(labelled_data)
train_indices <- sample(1:n, size = 0.8 * n, replace = FALSE)
test_indices <- setdiff(1:n, train_indices)
okaitana <- labelled_data$okataina
labelled_train <- labelled_data[train_indices,]
labelled_test <- labelled_data[test_indices,]
```

We define some functions for use later; also take note of the proportions of our data true/ false.

```r
calculate_metrics <- function(confusion_matrix) {
  TP <- confusion_matrix["TRUE", "TRUE"]
  FP <- confusion_matrix["TRUE", "FALSE"]
  FN <- confusion_matrix["FALSE", "TRUE"]
  TN <- confusion_matrix["FALSE", "FALSE"]

  accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
  precision <- ifelse((TP + FP) == 0, 0, TP / (TP + FP))

  sensitivity <- ifelse((TP + FN) == 0, 0, TP / (TP + FN))
  specificity <- ifelse((TN + FP) == 0, 0, TN / (TN + FP))
  f1_score <- ifelse((precision + sensitivity) == 0, 0,
                     2 * (precision * sensitivity) / (precision + sensitivity))
  return(list(
    Accuracy = accuracy,
    Precision = precision,
    Sensitivity = sensitivity,
    Specificity = specificity,
    F1_Score = f1_score
  ))
}


calculate_gini <- function(predictions, actuals) {
  cm <- table(predictions, actuals)

  TP <- cm[2, 2]  # True Positives
  FP <- cm[1, 2]  # False Positives
  TN <- cm[1, 1]  # True Negatives
  FN <- cm[2, 1]  # False Negatives


  return(2 * (TP / (TP + FP)) - 1)
}

calculate_misclassification_rate <- function(predictions, actuals) {
  cm <- table(predictions, actuals)
  return((cm[1, 2] + cm[2, 1]) / sum(cm))
}

calculate_true_proportion <- function(values) {
  true_count <- sum(values == "TRUE")
  total_count <- length(values)
  proportion_true <- true_count / total_count
  return(proportion_true)
}

proportion_true <- calculate_true_proportion(labelled_test$okataina)
cat("Proportion of entire set which is 'TRUE':", proportion_true, "\n")
```

```
## Proportion of entire set which is 'TRUE': 0.8939394
```

```
proportion_true <- calculate_true_proportion(labelled_train$okataina)
cat("Proportion of training set which is  'TRUE':", proportion_true, "\n")
```

```
## Proportion of training set which is  'TRUE': 0.8787879
```

```
proportion_true <- calculate_true_proportion(labelled_test$okataina)
cat("Proportion of test set which is  'TRUE':", proportion_true, "\n")
```

```
## Proportion of test set which is  'TRUE': 0.8939394
```

Evidently; most of our dataset is *True*

## Decision trees as a classifier

A useful classifier is a *decision tree*. Here we will create a binary tree whose leaf nodes represent the classification of our row given decisions about some members of that row.

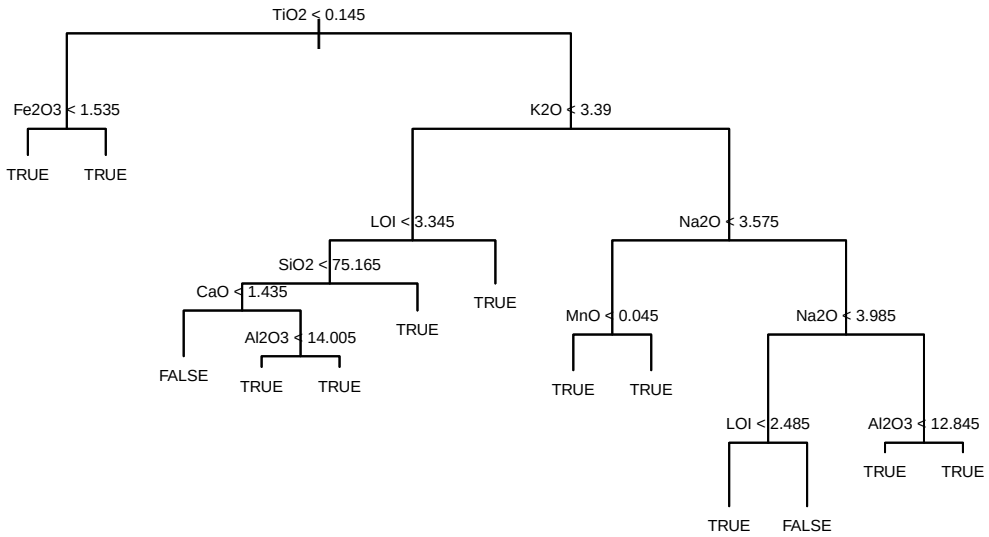
Firstly, will create a basic, first pass decision tree given the labelled data set:

```
class_tree <- tree(formula = okataina ~ . , data = labelled_train)
summary(class_tree)
```

```
##
## Classification tree:
## tree(formula = okataina ~ ., data = labelled_train)
## Variables actually used in tree construction:
## [1] "TiO2"  "Fe2O3" "K2O"   "LOI"   "SiO2"  "CaO"   "Al2O3" "Na2O"  "MnO"
## Number of terminal nodes:  13
## Residual mean deviance:  0.1589 = 39.87 / 251
## Misclassification error rate: 0.03409 = 9 / 264
```

We can visualize the tree as follows; each leaf node indicates whether or not, based on the decision tree, the row which corresponds to this path is for some rock sourced from *Okaitana* or not....

```
plot(class_tree)
#Apologies for the dreadful formatting.
text(class_tree,pretty=0,cex=0.5)
```




It's already clear that there are matching child nodes (or eventual leaf nodes of those children, per the first split); we will mitigate this computational overhead later on.

. . . get a 'feel' for the accuracy of the first tree built on the training set from our test set.

```
tree_predictions <- predict(class_tree, newdata = labelled_test, type = "class")
confusion_matrix <- table(tree_predictions, labelled_test$okataina)
tree_confusion_matrix <- table(tree_predictions, labelled_test$okataina)
tree_metrics <- calculate_metrics(tree_confusion_matrix)
tree_gini <- calculate_gini(tree_predictions, labelled_test$okataina)
tree_misclass <- calculate_misclassification_rate(tree_predictions, labelled_test$okataina)
```

Per the confusion matrix above; our tree is already *pretty good*; that is, false positives more or less do not occur with great frequency. False negatives occur with infrequency similarly.

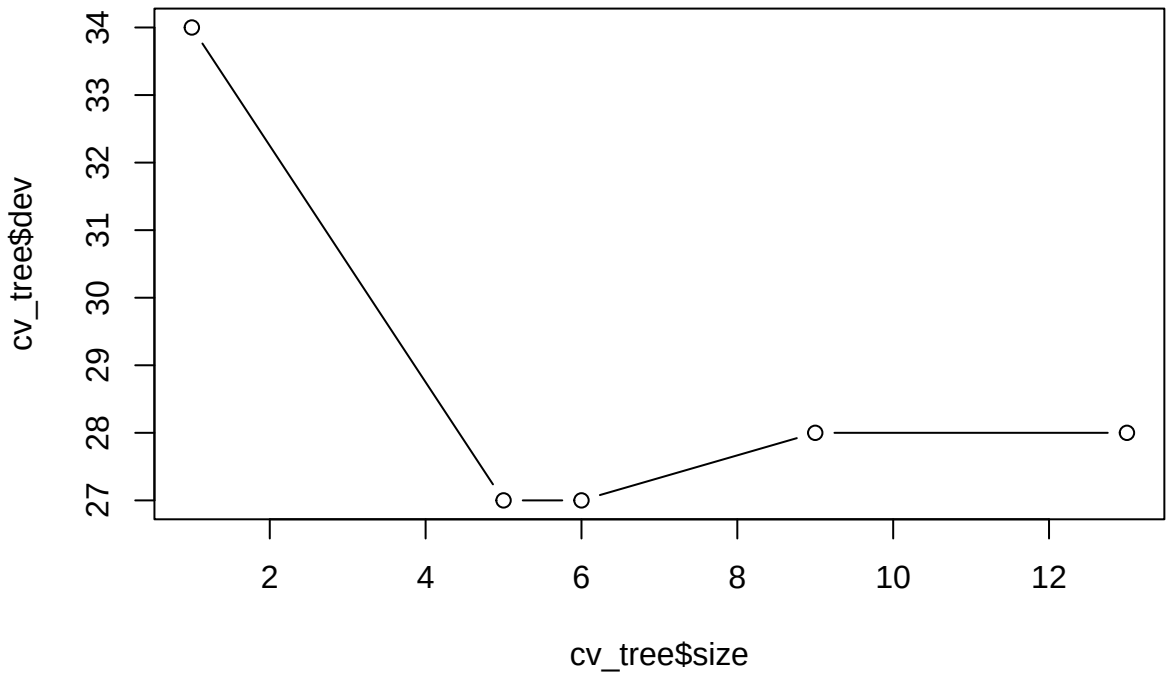
We will use cross validation to assess our tree.

```
set.seed(123)
#had to include this here again?
cv_tree <- cv.tree(class_tree, FUN = prune.misclass)
cv_tree
```

```
## $size
## [1] 13  9  6  5  1
##
## $dev
```

```
## [1] 28 28 27 27 34
##
## $k
## [1] -Inf  0.0  1.0  2.0  4.5
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

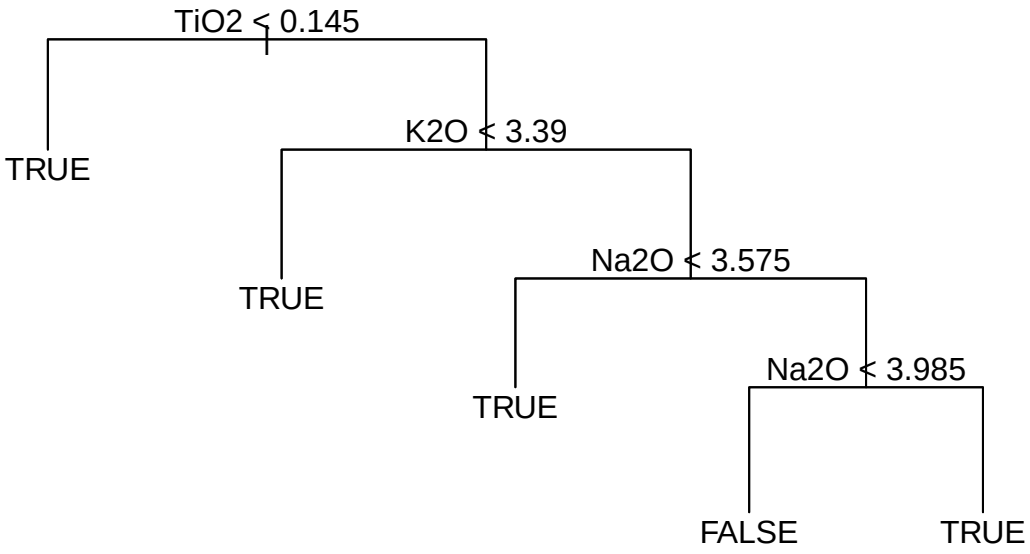
```
plot(cv_tree$size,cv_tree$dev,type="b")
```




We choose the best tree predicated on the lowest SD, which here occurs @ tree_size. = 5 and 6 (our model is particularly small)

5 is the best choice where there are perfomant differnetly sized trees...

```
pruned_class_tree=prune.misclass(class_tree,best=5)
plot(pruned_class_tree)
text(pruned_class_tree,pretty=0)
```




Now we assess the performance of this tree

```
pruned_tree_predictions <- predict(pruned_class_tree, labelled_test, type = "class")
pruned_tree_confusion_matrix <- table(pruned_tree_predictions, labelled_test$okataina)
pruned_tree_metrics <- calculate_metrics(pruned_tree_confusion_matrix)
pruned_tree_gini <- calculate_gini(pruned_tree_predictions, labelled_test$okataina)
pruned_tree_misclass <- calculate_misclassification_rate(pruned_tree_predictions, labelled_test$okataina
```

We can improve our model using ensemble techniques. Very simply, the following defines a 500 tree random Forrest approach:

(note this is separate to our cv model / pruned odel as above. )

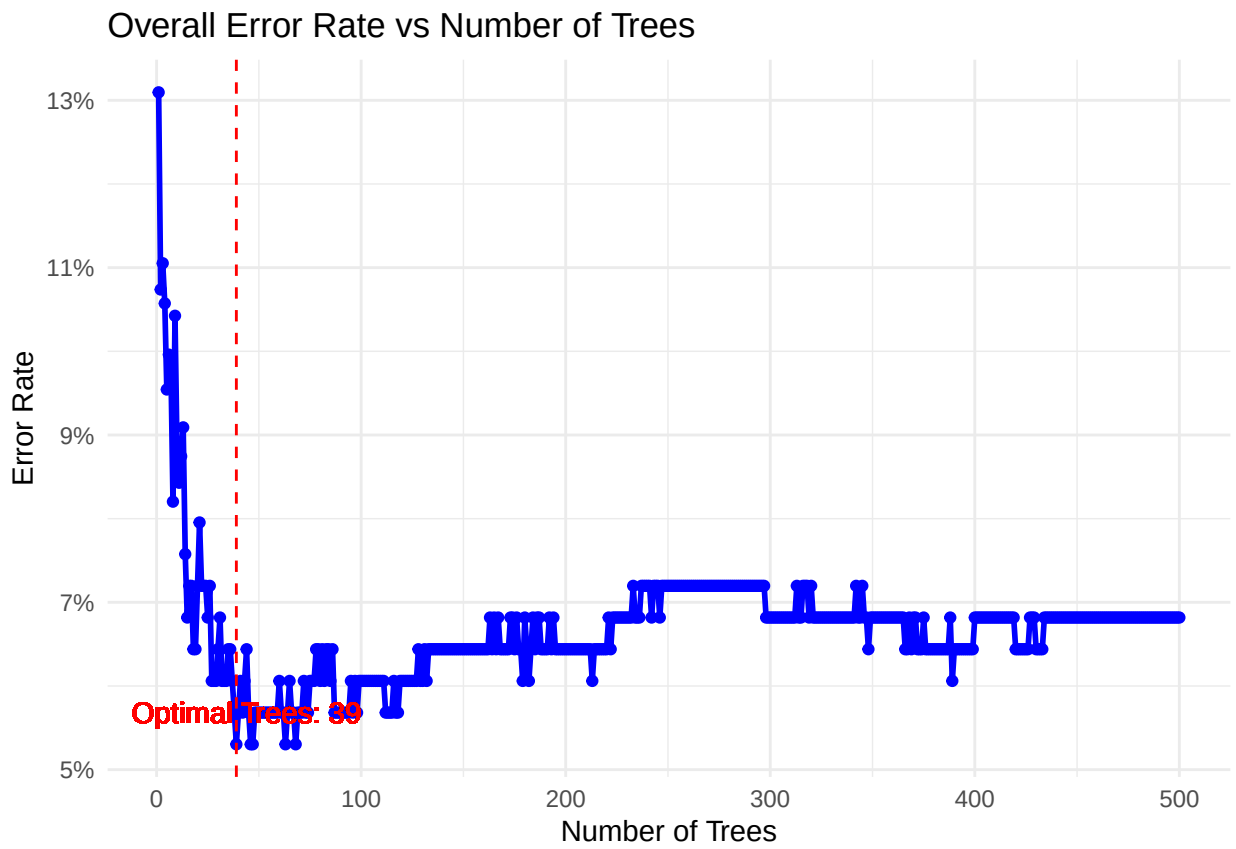
**Aside: Random Forrest:**

```
rf_model <- randomForest(okataina ~ ., data = labelled_train, ntree = 500)
rf_predictions <- predict(rf_model, newdata = labelled_test)
rf_confusion_matrix <- table(rf_predictions, labelled_test$okataina)
rf_metrics <- calculate_metrics(rf_confusion_matrix)
rf_gini <- calculate_gini(rf_predictions, labelled_test$okataina)
rf_misclass <- calculate_misclassification_rate(rf_predictions, labelled_test$okataina)



#visualising this
```

```r
oob_error <- rf_model$err.rate[, "OOB"]
num_trees <- seq(1, length(oob_error))
error_data <- data.frame(
  Trees = num_trees,
  OOB_Error = oob_error
)

optimal_trees <- num_trees[which.min(oob_error)]
optimal_error <- min(oob_error)

ggplot(data = error_data, aes(x = Trees, y = OOB_Error)) +
  geom_line(color = "blue", size = 1) +
  geom_point(color = "blue") +
  geom_vline(xintercept = optimal_trees, linetype = "dashed", color = "red") +
  geom_text(aes(x = optimal_trees + 5, y = optimal_error, label = paste("Optimal Trees:", optimal_trees
            color = "red", size = 4, vjust = -1) +
  labs(title = "Overall Error Rate vs Number of Trees",
       x = "Number of Trees",
       y = "Error Rate") +
  theme_minimal() +
  scale_y_continuous(labels = scales::percent) # Format as percentage
```

```
## Warning: Using ‘size‘ aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use ‘linewidth‘ instead.
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```

```
## Warning in geom_text(aes(x = optimal_trees + 5, y = optimal_error, label = paste("Optimal Trees:", :
## i Please consider using ‘annotate()‘ or provide this layer with data containing
##   a single row.
```




As we chose 500 trees; the visualization is slightly challenging; but what we see is that as the # of trees in our ensemble decreases; achieving an optimal @ 56 trees.

We consider the performance on our dataset at the end of the next section.

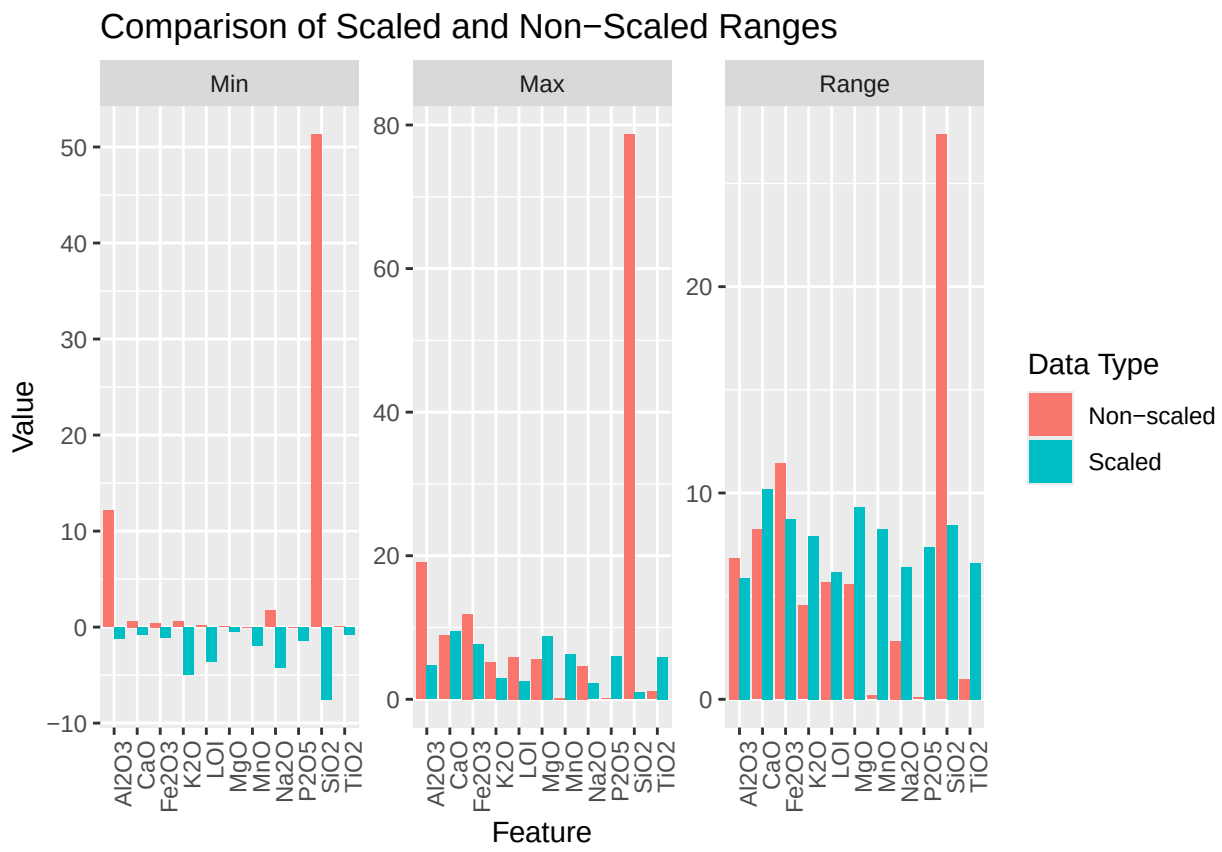
### K-NN As a predictor

Thus far we have considered a decision tree for our classifier. An alternate approach is *K-Nearest Neigbour* analysis. At a very abstract level; we wish to, using the training set, build a n-dimensional *matrix* (a term used loosely) of data and, when we query a point, find the existing number of points who are closest by some distance metric. (i.e similarity metric). In the event of ties for our True / False values; we simply average (i.e median) if a tie continues, we increase the number of candidate neighbors.

Note: It is challenging to visualize our KNN algorithm because we operate in the space of the number of predictors in our dataset, here, 14. It is useful to normalize our data to have mean of 0 and SD of 1, if some features have different scales, the distance between the scales can affect our model, as we're only interested in the relative distances, not the scale distances, we normalize.

```
labelled_data_scaled <- labelled_data
labelled_data_scaled[, -which(names(labelled_data_scaled) == "okataina")] <- scale(labelled_data_scaled

# For visualisation:
ranges_non_scaled <- data.frame(
  Feature = names(labelled_data)[-which(names(labelled_data) == "okataina")],
  Min = apply(labelled_data[, -which(names(labelled_data) == "okataina")], 2, min),
  Max = apply(labelled_data[, -which(names(labelled_data) == "okataina")], 2, max)
)
```

```r
ranges_non_scaled$Range <- ranges_non_scaled$Max - ranges_non_scaled$Min
ranges_non_scaled$Type <- 'Non-scaled'

ranges_scaled <- data.frame(
  Feature = names(labelled_data_scaled)[-which(names(labelled_data_scaled) == "okataina")],
  Min = apply(labelled_data_scaled[, -which(names(labelled_data_scaled) == "okataina")], 2, min),
  Max = apply(labelled_data_scaled[, -which(names(labelled_data_scaled) == "okataina")], 2, max)
)
ranges_scaled$Range <- ranges_scaled$Max - ranges_scaled$Min


ranges_scaled$Type <- 'Scaled'
ranges_combined <- rbind(ranges_non_scaled, ranges_scaled)

ranges_melted <- melt(ranges_combined, id.vars = c("Feature", "Type"))

ggplot(ranges_melted, aes(x = Feature, y = value, fill = Type)) +
  geom_bar(stat = "identity", position = "dodge") +
  facet_wrap(~variable, scales = "free_y") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  labs(title = "Comparison of Scaled and Non-Scaled Ranges",
       x = "Feature",
       y = "Value",
       fill = "Data Type")
```




And we can see our new scale above, all predictors are now on the same (blue) scale.

**What K to choose?** We'll start with k = 5 and explore a few other K's later.

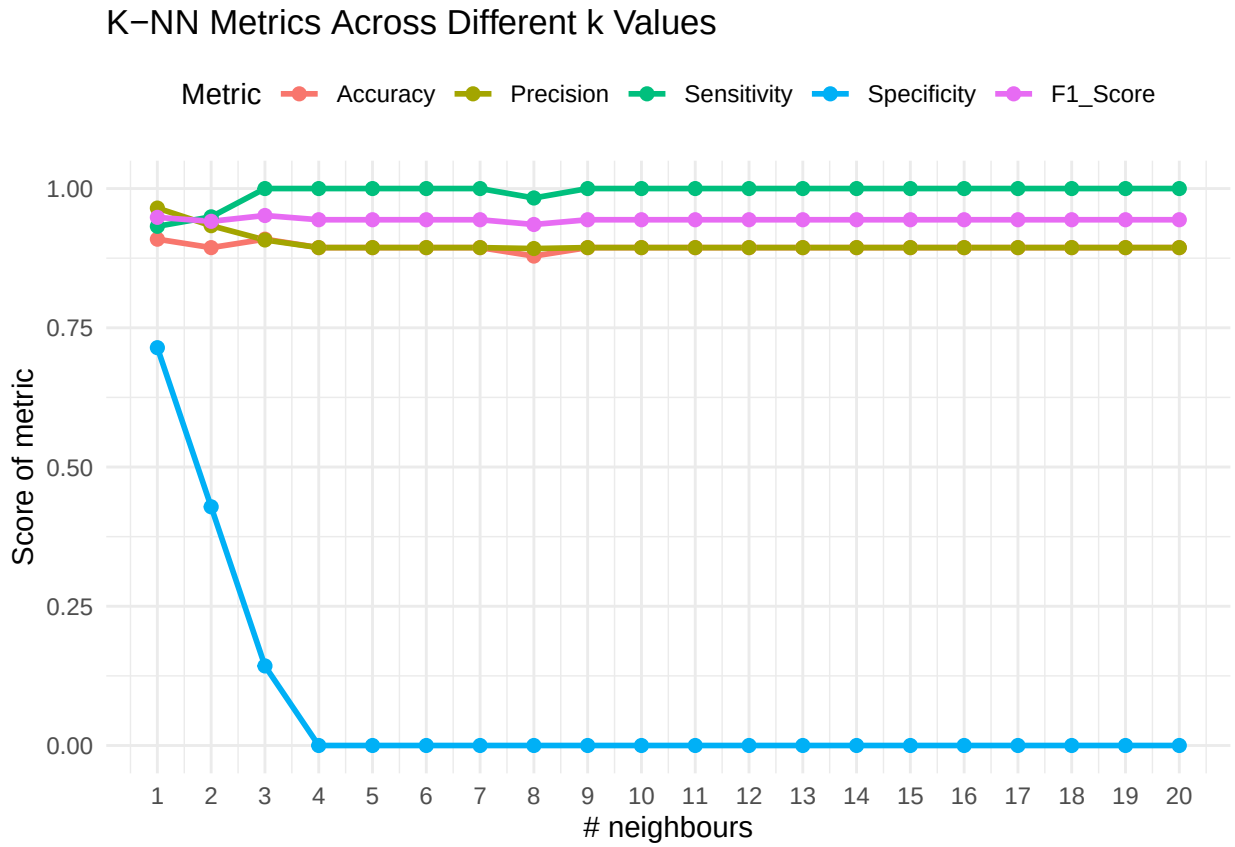
Firstly, scaled training / test sets as follows:

```
# Define the training and test sets
scaled_labelled_train_data <- labelled_data_scaled[train_indices, -which(names(labelled_data_scaled) ==
scaled_labelled_test_data <- labelled_data_scaled[test_indices, -which(names(labelled_data_scaled) == "

train_labels <- labelled_data_scaled$okataina[train_indices]
test_labels <- labelled_data_scaled$okataina[test_indices]
```

```
accuracies <- numeric(22)
metrics_results <- matrix(NA, nrow = 20, ncol = 5)
colnames(metrics_results) <- c("Accuracy", "Precision", "Sensitivity", "Specificity", "F1_Score")

# Loop through k values
for (k in 1:20) {
  knn_predictions <- knn(train = scaled_labelled_train_data, test = scaled_labelled_test_data, cl = trai

  confusion_matrix <- table(knn_predictions, test_labels)
  metrics <- calculate_metrics(confusion_matrix)

  #Unfortunately can't find an R friendly way to do this
  metrics_results[k, "Accuracy"] <- metrics$Accuracy
  metrics_results[k, "Precision"] <- metrics$Precision
  metrics_results[k, "Sensitivity"] <- metrics$Sensitivity
  metrics_results[k, "Specificity"] <- metrics$Specificity
  metrics_results[k, "F1_Score"] <- metrics$F1_Score
}
print(metrics_results)
```

```
##        Accuracy Precision Sensitivity Specificity  F1_Score
##  [1,] 0.9090909 0.9649123   0.9322034   0.7142857 0.9482759
##  [2,] 0.8939394 0.9333333   0.9491525   0.4285714 0.9411765
##  [3,] 0.9090909 0.9076923   1.0000000   0.1428571 0.9516129
##  [4,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
##  [5,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
##  [6,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
##  [7,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
##  [8,] 0.8787879 0.8923077   0.9830508   0.0000000 0.9354839
##  [9,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [10,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [11,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [12,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [13,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [14,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [15,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [16,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [17,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [18,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [19,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
## [20,] 0.8939394 0.8939394   1.0000000   0.0000000 0.9440000
```

```r
#for visualization
metrics_df <- as.data.frame(metrics_results)
metrics_df$k <- 1:20
metrics_long <- melt(metrics_df, id.vars = "k",
                     variable.name = "Metric",
                     value.name = "Value")
ggplot(metrics_long, aes(x = k, y = Value, color = Metric)) +
  geom_line(size = 1) +
  geom_point(size = 2) +
  labs(title = "K-NN Metrics Across Different k Values",
       x = "# neighbours",
       y = "Score of metric",
       color = "Metric") +
  theme_minimal() +
  scale_x_continuous(breaks = 1:20) +
  theme(legend.position = "top")
```




Selecting K from the above metrics is in ways, specific to the domain. We select the K here which maximizes accuracy overall and also precision. We don't weight any measure of false positive, true positive rate, false negative, true negative weight in any particular way, although, depending on the application of the model (i.e if a particular rock's origin has some major cultural significance) we may chose a K which maximimuizes specificity or sensitivity (or strikes a balance therein).

We select K = 4 for our K-NN model:

```
K = 4
```

```
knn_predictions <- knn(train = scaled_labelled_train_data, test = scaled_labelled_test_data, cl = train
knn_confusion_matrix <- table(knn_predictions, test_labels)
knn_metrics <- calculate_metrics(knn_confusion_matrix)
knn_gini <- calculate_gini(knn_predictions, test_labels)
knn_misclass <- calculate_misclassification_rate(knn_predictions, test_labels)
```

```
model_results <- data.frame(
  Model = c("Decision Tree", "Pruned Tree", "KNN", "Random Forest"),
  Accuracy = c(tree_metrics$Accuracy, pruned_tree_metrics$Accuracy, knn_metrics$Accuracy, rf_metrics$Ac
  Precision = c(tree_metrics$Precision, pruned_tree_metrics$Precision, knn_metrics$Precision, rf_metric
  Sensitivity = c(tree_metrics$Sensitivity, pruned_tree_metrics$Sensitivity, knn_metrics$Sensitivity, r
  Specificity = c(tree_metrics$Specificity, pruned_tree_metrics$Specificity, knn_metrics$Specificity, r
  F1_Score = c(tree_metrics$F1_Score, pruned_tree_metrics$F1_Score, knn_metrics$F1_Score, rf_metrics$F1
  Gini = c(tree_gini, pruned_tree_gini, knn_gini, rf_gini),
  Misclassification_Rate = c(tree_misclass, pruned_tree_misclass, knn_misclass, rf_misclass)
)
# Generate the table using kable
kable(model_results, caption = "Comparison of Model Performance Metrics")
```

Table 1: Comparison of Model Performance Metrics

| Model | Accuracy | Precision | Sensitivity | Specificity | F1_Score | Gini | Misclassification_Rate |
|---|---|---|---|---|---|---|---|
| Decision Tree | 0.9393939 | 0.9365079 | 1.0000000 | 0.4285714 | 0.9672131 | 1.0000000 | 0.0606061 |
| Pruned Tree | 0.9242424 | 0.9218750 | 1.0000000 | 0.2857143 | 0.9593496 | 1.0000000 | 0.0757576 |
| KNN | 0.9090909 | 0.9076923 | 1.0000000 | 0.1428571 | 0.9516129 | 1.0000000 | 0.0909091 |
| Random Forest | 0.8787879 | 0.8923077 | 0.9830508 | 0.0000000 | 0.9354839 | 0.9661017 | 0.1212121 |

The table (above) presents the performance metrics for four different models: the base Decision Tree, Pruned Tree, K-Nearest Neighbors (KNN), and Random Forest. (it made sense to keep these, considering we explored them earlier before ultimately arriving at the rnadom forrest)

The Decision Tree achieves the highest accuracy at approximately 93.94%, along with a precision of 93.65% and perfect sensitivity (100%), demonstrating its strong capabislity in identifying positive cases. However, it also exhibits low specificity, suggesting that many negative cases are incorrectly classified as positive.

While the Pruned Tree shows slightly reduced metrics across the board, it still performs well, maintaining perfect sensitivity and a *good* accuracy of 92.42%. The KNN model, although the least accurate at 89.39%, also demonstrates perfect sensitivity but suffers from an absence of specificity, indicating potential issues in its ability to correctly classify negative instances.

Interestingly, the Random Forest model, despite what we might expect presents the lowest accuracy at 87.88% and a increase in the misclassification rate (12.12%). It shows a significant drop in specificity, i.e. there are challenges in accurately identifying negative classes.

Overall, while the Decision Tree performs the best, the results are still relative to the seed we used. Cross validation across many differing seeds (or the same seed with logical splits on each fold) would help our results greatly.

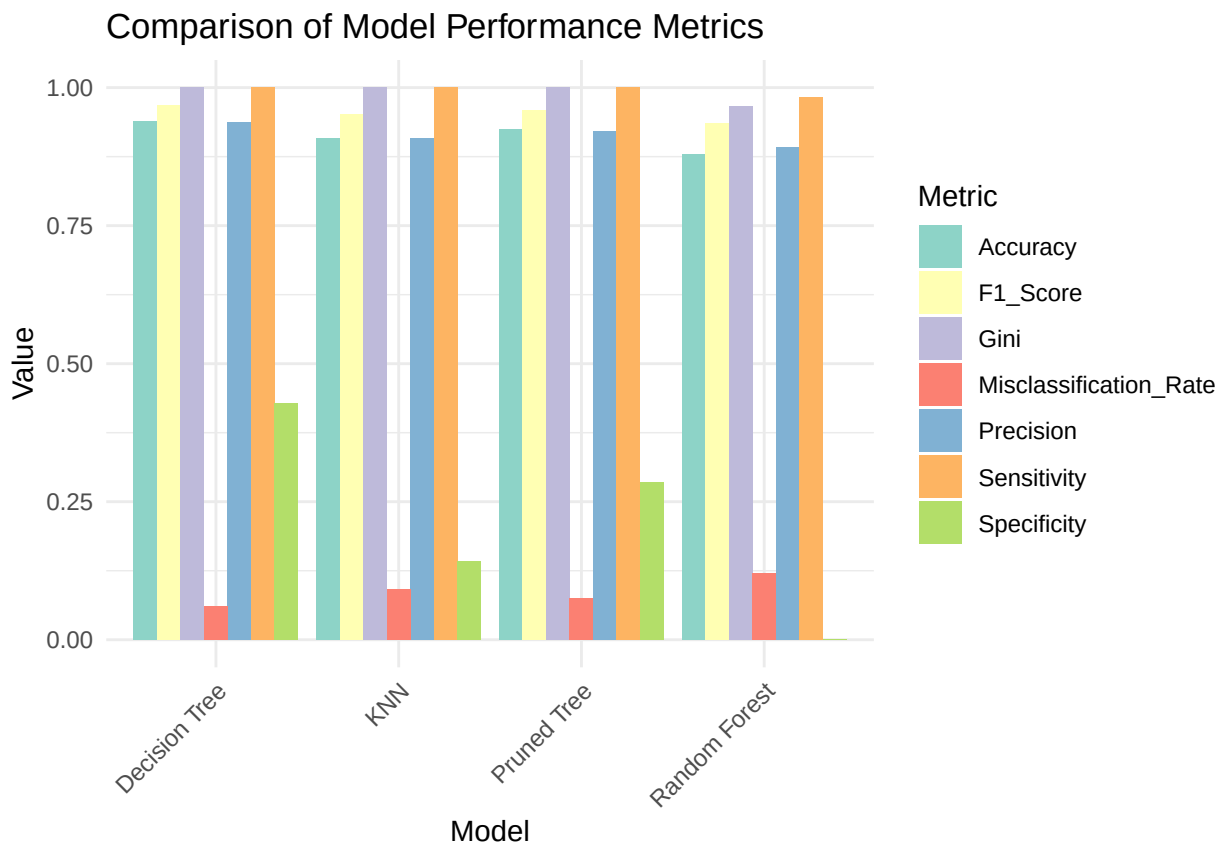
Simply, the seed we used, may just *happen* to perform very well for our base model.

Note, we include gini in our output as an overall measure of performance:

Gini is particularly beneficial as it quantifies the model's discriminatory power, essentially measuring how well the model distinguishes between positive and negative classes.

A higher Gini coefficient indicates a better-performing model, as it reflects a greater separation between the predicted probabilities for the positive and negative classes. In our results, while the Decision Tree has the highest Gini of 1.00, indicating perfect discrimination, the Random Forest's Gini of 0.966 suggests that its ability to distinguish between classes is not as effective (although these are very close and plausibly the difference could even be the result of downstream aggregate floating point errors at a stretch), which aligns with its lower accuracy and higher misclassification rate. Thus, Gini serves as a valuable insight for our accuracy and other metrics.

```
model_results_long <- pivot_longer(model_results,
                                   cols = -Model,
                                   names_to = "Metric",
                                   values_to = "Value")

ggplot(model_results_long, aes(x = Model, y = Value, fill = Metric)) +
  geom_bar(stat = "identity", position = "dodge") +
  labs(title = "Comparison of Model Performance Metrics",
       x = "Model",
       y = "Value") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_fill_brewer(palette = "Set3") # Change color palette as desired
```



In terms of predicting the origin of stones given our dataset input and also our particular random seed (we mentioned this earlier, of course it doesn't make sense to make this a consideration, but consider it a 'future work' section), we should just use a basic tree, no pruning, no cv and no ensemble learning.

---

## Clustering Seeds

Given the unannotated dataset, we can use clustering methods such as K-NN to determine *How many clusters exist?* in our seed dataset, that is, *How many seeds are represented in our dataset?*

As per usual, we load in the data set and flush the environment (for the sake of being able to reuse some variable names in the IDE without too much trouble). We'll also filter for *na* values in the data if they exist...

```
rm(list = ls())
seeds <- read.csv('seeds.csv')
seeds <- seeds[, !names(seeds) %in% c('X')]
seeds_annotated <- read.csv('seeds_annotated.csv')
seeds_annotated <- seeds_annotated[, !names(seeds_annotated) %in% c('X')]

#sanity check
sum(is.na(seeds))
```

```
## [1] 0
```

```
sum(is.na(seeds_annotated))
```

```
## [1] 0
```

```
dim(seeds)
```

```
## [1] 13511    16
```

In part 1, we used *K-Nearest-Neighbors* to determine our categorization in a binary space; however now we are *categorizing* to an unknown number of seed types. Essentially, with our K-means approach, we will train a number of K-Means models until we see no noticeable improvement in MSE

We firstly define some explicit bounds; Our K-means classifier works by iteratively 'converging' the dataset to clusters until there is no change. However, we do define a ceiling on the *off chance* that we don't converge in a useful time or that our *updates* oscillate around some minimum.

```
max_iterations <- 100
```

We define some preprocessing steps here. The first is reasonably self explanatory; we simply change the datatype. More importantly, however, we do scale the data to have mean = 0 and sd = 1 in our preprocess step. This is the same process that we saw in part 1, as such we don't visualize it.

```r
convert_to_float <- function(column) {
  as.numeric(gsub(",", "", column))
}

preprocess_data <- function(data) {
  #convert to numeric for R then just scale like in K-NN
  data[] <- lapply(data, function(col) {
    if (is.character(col)) {
      convert_to_float(col)
    } else {
      col
    }
  })
  seeds_scaled <- scale(data)
  return(seeds_scaled)
}
```

We load in the unlabelled set

```r
seeds_preprocessed <- preprocess_data(seeds)
```

And we define some useful functions for later.

Firstly; the actual K-means

```r
k_means_iter <- function(data, K, max_iterations = 100) {

  n <- nrow(data)

  C <- sample(1:K, n, replace = TRUE)

  clusters_and_means <- matrix(0, nrow = K, ncol = ncol(data))
  for (i in 1:max_iterations) {
    Prev_C <- C


    for (k in 1:K) {
      cluster_data <- data[C == k, , drop = FALSE]
      if (nrow(cluster_data) > 0) {
        clusters_and_means[k, ] <- colMeans(cluster_data, na.rm = TRUE)  # Calculate mean
      }
    }




    distances <- as.matrix(dist(rbind(data, clusters_and_means)))
    distances <- distances[1:n, (n + 1):(n + K)]
    C <- apply(distances, 1, which.min)

    if (identical(C, Prev_C)) {
      cat("Converged", K,  "after", i, "iterations.\n")
      break
```

```
    }
  }

  return(C)  # Return cluster assignments only
}
k_means <- function(data, K, max_iterations = 100) {
  #Same idea as above but now using lapply
  #not really any more performant than the above code in a way that helps with our results
  n <- nrow(data)
  C <- sample(1:K, n, replace = TRUE)

  for (i in 1:max_iterations) {
    Prev_C <- C

    clusters_and_means <- do.call(rbind, lapply(1:K, function(k) {
      cluster_data <- data[C == k, , drop = FALSE]
      if (nrow(cluster_data) > 0) {
        colMeans(cluster_data, na.rm = TRUE)
      } else {
        rep(NA, ncol(data))
      }
    }))

    distances <- as.matrix(dist(rbind(data, clusters_and_means)))
    distances <- distances[1:n, (n + 1):(n + K)]

    C <- apply(distances, 1, which.min)

    if (identical(C, Prev_C)) {
      cat("Converged on", K, "clusters after", i, "iterations.\n")
      break
    }
  }

  return(C)  # Return cluster assignments only
}


silhouette_score <- function(data, cluster_assignments) {
  n <- nrow(data)
  sil_widths <- numeric(n)
  dist_matrix <- as.matrix(dist(data))
  for (i in 1:n) {
    same_cluster <- cluster_assignments == cluster_assignments[i]
    if (sum(same_cluster) > 1) {
      a_i <- mean(dist_matrix[i, same_cluster & (1:n != i)])
    } else {
      a_i <- 0
    }

    # Calculate ave dists to other clusters
    other_clusters <- unique(cluster_assignments[!same_cluster])
    b_i_values <- sapply(other_clusters, function(k) {
```

```
      mean(dist_matrix[i, cluster_assignments == k])
    })

    b_i <- min(b_i_values)
    sil_widths[i] <- (b_i - a_i) / max(a_i, b_i)
  }

  return(mean(sil_widths, na.rm = TRUE))  # Return average silhouette width
}


wss_score <- function(data, cluster_assignments) {
  total_wss <- 0
  for (cluster in unique(cluster_assignments)) {
    cluster_points <- data[cluster_assignments == cluster, ]
    cluster_centroid <- colMeans(cluster_points)
    squared_distances <- rowSums((cluster_points - cluster_centroid)^2)
    total_wss <- total_wss + sum(squared_distances)
  }
  return(total_wss)
}
```

There are a few items in the above chunk. There are two k-means algorithms (or implementations) the first is the iterative approach and the latter attempts to utilise paralellization by applying lapply to calculate the mean of data points per cluster. Both implementations are included, both are not *quick* to compute; and we have *hard coded* results in a few cells down. The reason for including both is to 1. show the best choice and 2. show what was actually used.

Both kmeans algorithms broadly work in the following way

Firstly, they begin by randomly assigning each data point to one of the K clusters.

In each iteration, the centroids (mean points) of each cluster are recalculated based on the current assignments.

In the case of lapply this is done per cluster in a parallelized way. The function then computes the Euclidean distance between each data point and the centroids, reassigning each point to the nearest centroid. This iterative process continues until the cluster assignments no longer change, signaling convergence. The function returns the final cluster assignments after either convergence or the completion of a maximum number of iterations, which we defined as 100 earlier. This iterative refinement ensures that the clusters progressively improve to minimize the distance between points and their respective cluster centroids.

Because of the high dimensionality of our data, it is difficult to visualize the clustering / outputs. Let's begin with a base case for the sake of exploration. Here K (number of seeds) is assigned to be $K = 3$.

```
K <- 3
C <- k_means(seeds_preprocessed, K)
```

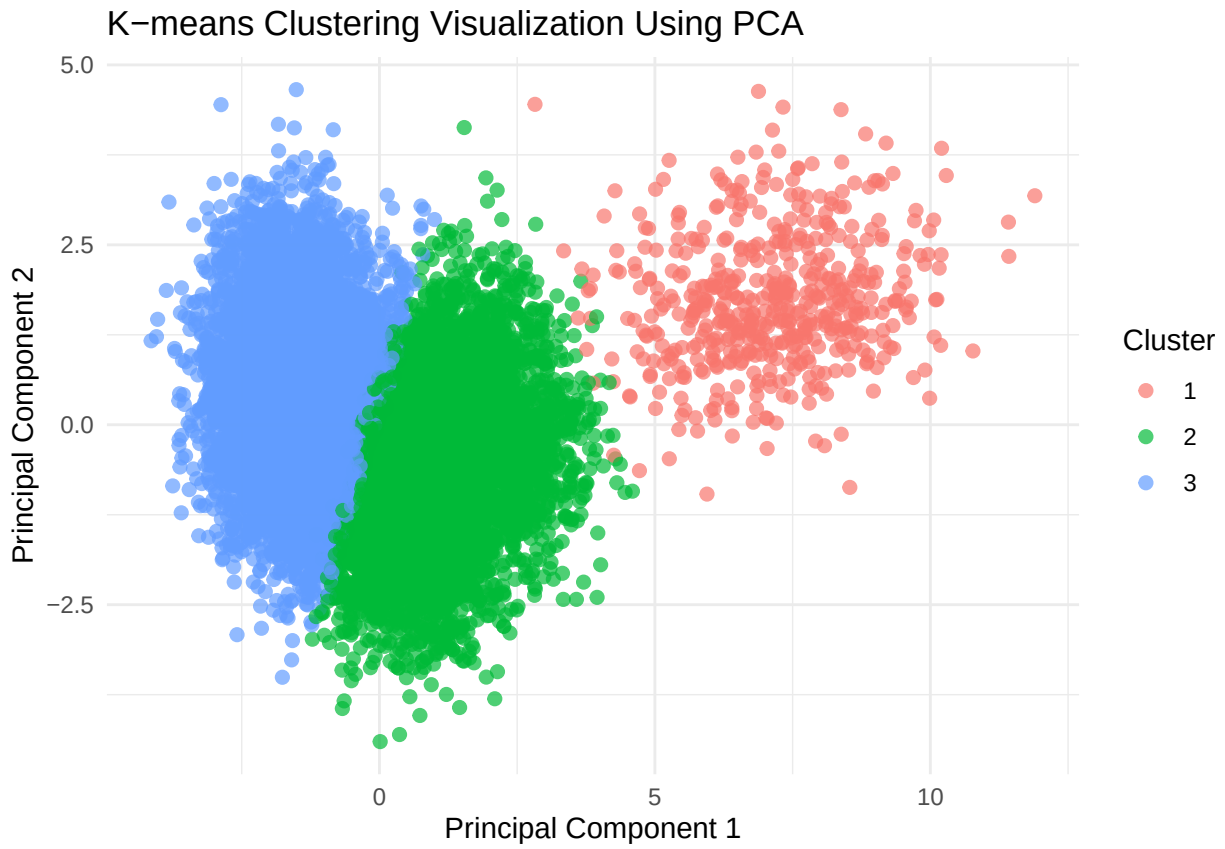
```
## Converged on 3 clusters after 51 iterations.
```

```
sil_score <- silhouette_score(seeds_preprocessed, C)
print(paste("Average Silhouette Score:", sil_score))
```

```
## [1] "Average Silhouette Score: 0.154429404936928"
```

Due to the high dimensionality of our data, it is challenging to display the clusters in a human readable way. We *can* use PCA for dimensionality reduction to create a somewhat useful graphic, but this is cursory at best as much information is lost in the PCA process. PCA primarily preserves *variance* but may overlook important relationships that contribute to cluster separation.

```
pca_result <- prcomp(seeds_preprocessed, scale. = TRUE)
pca_data <- data.frame(pca_result$x[, 1:2])
colnames(pca_data) <- c("PC1", "PC2")

pca_data$Cluster <- as.factor(C)  # Directly use C as it holds the cluster assignments

library(ggplot2)
ggplot(pca_data, aes(x = PC1, y = PC2, color = Cluster)) +
  geom_point(size = 2, alpha = 0.7) +
  theme_minimal() +
  labs(title = "K-means Clustering Visualization Using PCA",
       x = "Principal Component 1",
       y = "Principal Component 2") +
  scale_color_discrete(name = "Cluster")
```




Our next step is to decide which $K$ to use; we're going to arbitrarily chose K = 4 through K = 40; a good upper bound is sqrt(N/2) which for our ~13500 rows is around 80. We won't sample the data. we compute on the entire set. 40 *should* provide a good upper bound

```r
# we'll just worry about NA later...
silhouette_scores <- rep(NA, 50)
wss_per_k <- rep(NA, 50)
# This will take a long time.....
if (FALSE){
  for (K in 3:40) {q
    C <- k_means(seeds_preprocessed, K)
    sil_score <- silhouette_score(seeds_preprocessed, C)
    silhouette_scores[K] <- sil_score
            wss_score_ <- wss_score(seeds_preprocessed, C)
            wss_per_k[K] <- wss_score_
  }
}
```

The above chunk of R code would usually output along the lines of. In the interest of generating this document we *did not* run the above cells. We ran this earlier and the output is available below.... (it is simply too time consuming to run again at compile time for the doc)

Sample output includes:

‘

Converged 8 after 41 iterations.

Converged 9 after 22 iterations.

Converged 10 after 34 iterations.

Converged 11 after 23 iterations.

... .

And the following is output from chatgpt, given a copy of the output vector from the above chunk (and simply asking for it to be formatted as an R vector)

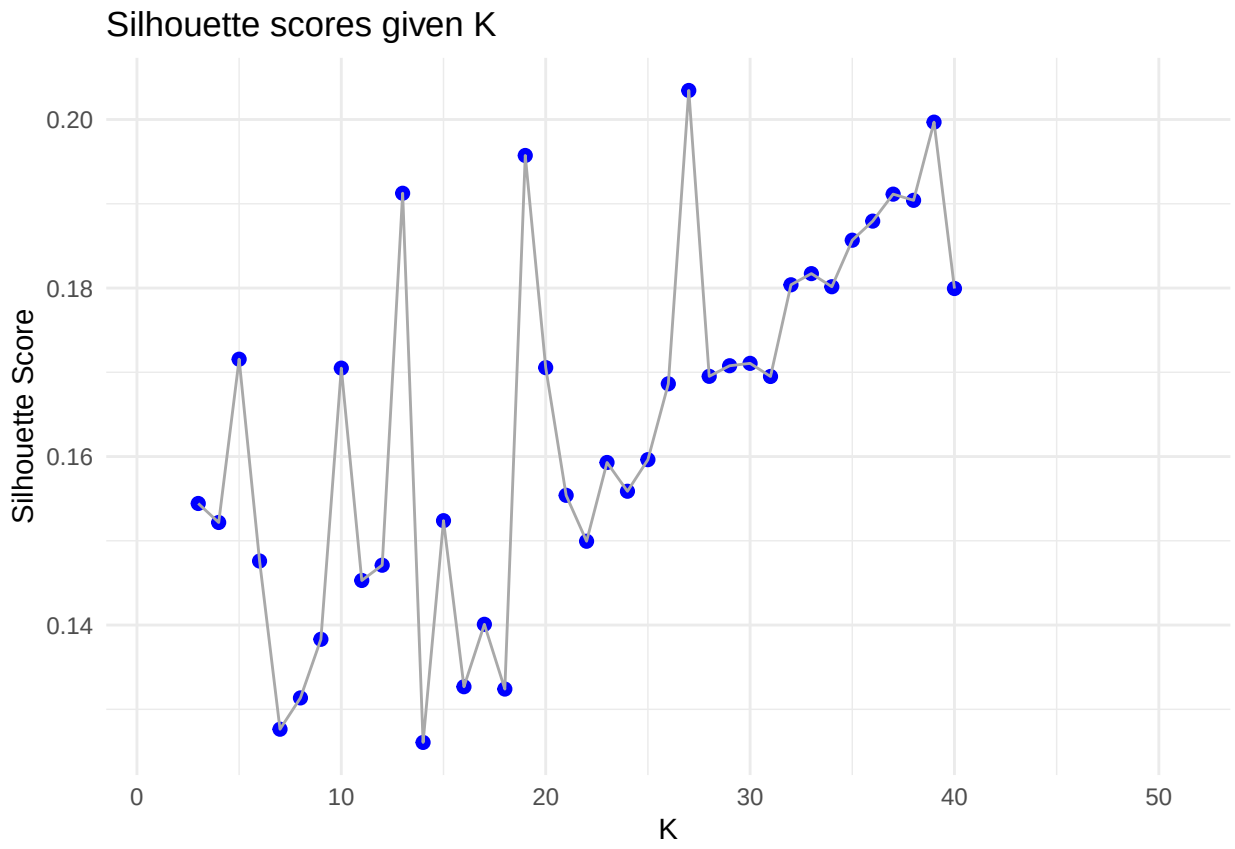
```r
# these are derived from silhouette_scores and wss_per_k per the above chunk

wss_scores <- c(NA, NA, 253301.9, 258458.8, 263044.6, 271071.6, 279436.5, 281432.4, 290238.0,
                294728.3, 296757.2, 298967.6, 301748.4, 301005.3, 307316.0, 305555.6, 308457.8,
                310581.0, 314337.8, 315455.2, 317265.2, 317115.4, 319128.6, 320518.2, 322653.6,
                323348.3, 320019.1, 323854.1, 323603.4, 324505.7, 325580.1, 326319.3, 327273.4,
                325252.5, 329094.5, 329517.1, 329447.6, 332130.9, 331679.1, 330582.9, NA, NA,
                NA, NA, NA, NA, NA, NA, NA, NA)

sil_scores <- c(NA, NA, 0.1544294, 0.1521766, 0.1715518, 0.1476006, 0.1276367, 0.1313523, 0.1383110,
              0.1704923, 0.1452927, 0.1471020, 0.1912484, 0.1260696, 0.1523873, 0.1326807, 0.1400883,
              0.1324051, 0.1957321, 0.1705505, 0.1553975, 0.1499468, 0.1592949, 0.1558798, 0.1596299,
              0.1686347, 0.2034484, 0.1695190, 0.1707770, 0.1710606, 0.1695030, 0.1803850, 0.1817097,
              0.1801543, 0.1856765, 0.1879409, 0.1911431, 0.1904090, 0.1996871, 0.1799416, NA, NA,
              NA, NA, NA, NA, NA, NA, NA, NA, NA)

convergence_df <- c(NA, NA, 23, 27, 29, 17, 42, 41, 22, 34, 23, 25, 27, 27, 41, 29, 46, 28, 30, 29,
                    89, 28, 40, 29, 19, 35, 28, 48, 28, 27, 32, 37, 56, 21, 33, 37, 36, 38, 35, 19,
                    NA, NA, NA, NA, NA, NA, NA, NA, NA, NA)
```

```r
df <- data.frame(Index = seq_along(sil_scores), Value = sil_scores)
```

```
# Plot the data using ggplot
ggplot(df, aes(x = Index, y = Value)) +
  geom_point(color = "blue", size = 2) +      # Plot the points
  geom_line(color = "darkgray") +             # Connect the points with a line
  labs(title = "Silhouette scores given K",
       x = "K",
       y = "Silhouette Score") +
  theme_minimal()
```

```
## Warning: Removed 13 rows containing missing values or values outside the scale range
## (`geom_point()`).
```

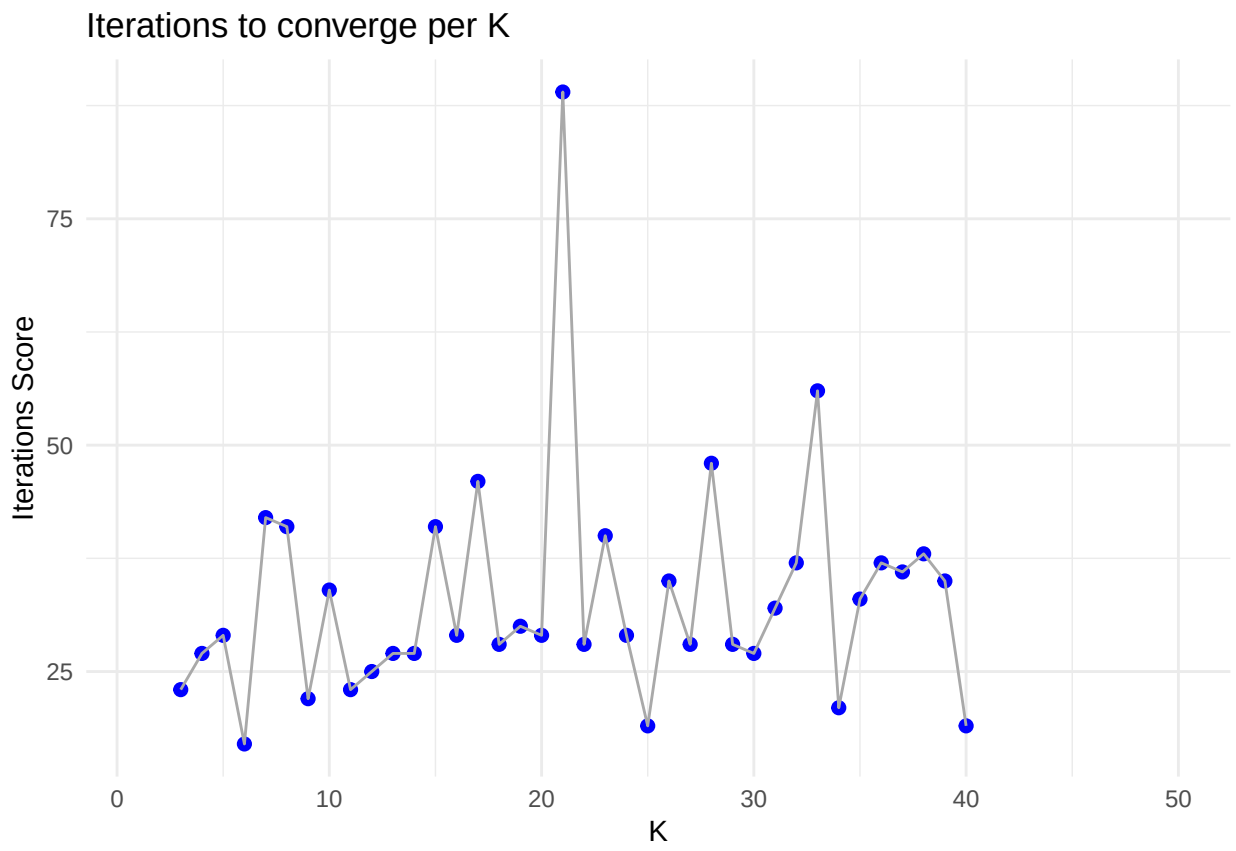
```
## Warning: Removed 13 rows containing missing values or values outside the scale range
## (`geom_line()`).
```




```
df <- data.frame(Index = seq_along(convergence_df), Value = convergence_df)

# Plot the data using ggplot
ggplot(df, aes(x = Index, y = Value)) +
  geom_point(color = "blue", size = 2) +      # Plot the points
  geom_line(color = "darkgray") +             # Connect the points with a line
  labs(title = "Iterations to converge per K",
       x = "K",
       y = "Iterations Score") +
  theme_minimal()
```

```
## Warning: Removed 12 rows containing missing values or values outside the scale range
## (`geom_point()`).
```

```
## Warning: Removed 12 rows containing missing values or values outside the scale range
## (`geom_line()`).
```




It is not immediately clear as to which K is best;

Our greatest overall silhouette score, (i.e. the K which indicates the best cluster separation and cohesion) occurs at K = 28

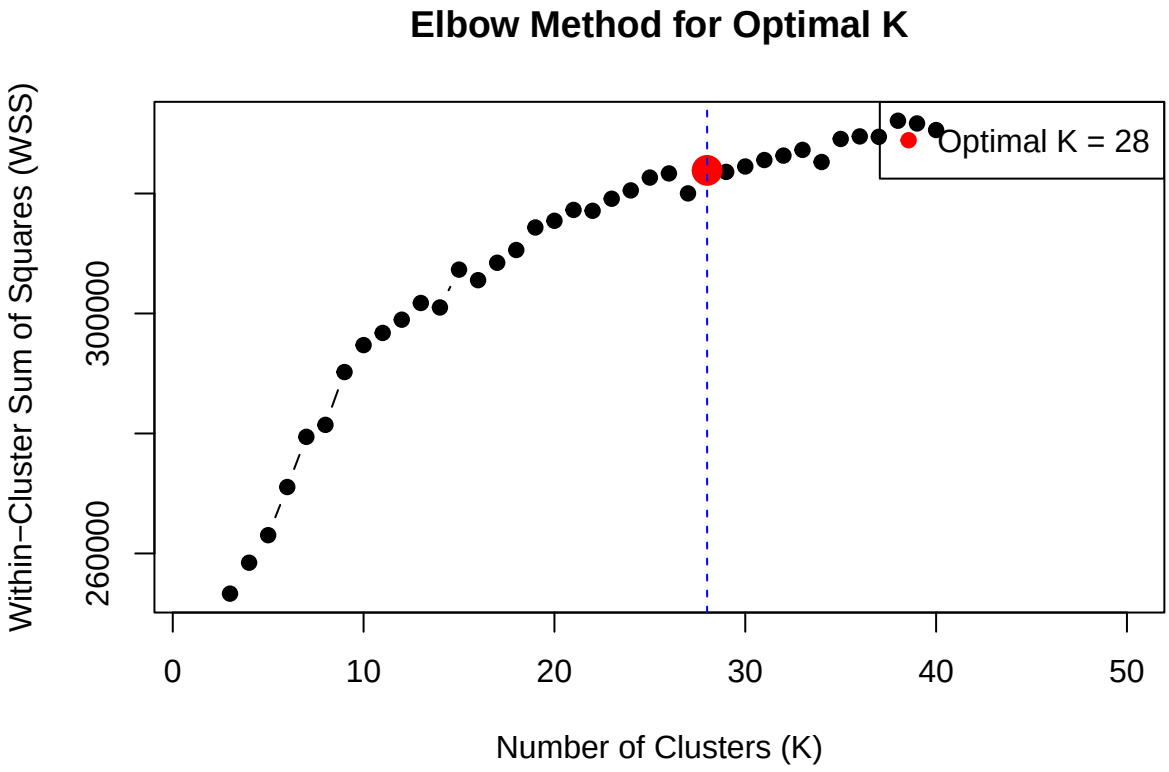
The silhouette score here measures how similar a point is to it's own cluster in comparison to other clusters; a high score (average) would indicate a good clustering (noting clustering *goodness i.e. minimization of in cluster MSS* is congruent with # of seeds). We choose the max of our K's. However we also consider our Within-Cluster-Sum-Of-Squares ads a measure of quantifying the total variance within each cluster. We do not simply choose the max or min; rather the point where the rate of increase begins to slow *or is the 'most changing towards slowing*. This is done via the *elbow method* as seen below:

```
K_values <- 1:length(wss_scores)
plot(K_values, wss_scores, type = "b", pch = 19,
     xlab = "Number of Clusters (K)",
     ylab = "Within-Cluster Sum of Squares (WSS)",
     main = "Elbow Method for Optimal K",
     ylim = range(c(wss_scores), na.rm = TRUE))

wss_diff <- c(NA, diff(wss_scores))
wss_diff2 <- c(NA, diff(wss_diff))
```

```r
elbow_index <- which.max(wss_diff2)
optimal_K <- elbow_index
cat("The optimal number of clusters (K) is:", optimal_K, "\n")
```

```
## The optimal number of clusters (K) is: 28
```

```r
points(optimal_K, wss_scores[optimal_K], col = "red", pch = 19, cex = 2)
abline(v = optimal_K, col = "blue", lty = 2)
legend("topright", legend = paste("Optimal K =", optimal_K), col = "red", pch = 19)
```




We find that our optimal WSS is at K = 28;

It holds therefore that the number of clusters in our dataset; i.e. the number of seeds is very likely 28.

..... That being said.

We also must consider that the intiital randomized seed ... for the seed clusters... strongly impacts the performance of our model; we have tried only a single seed. The large compute time and R natively not scaling well too multi core / multi gpu are a factor (we did not use lapply in our implementation, for example, which does achieve better parallelization)

A better approach would be to create numerous models with differing seeds and average the output on an unseen row. We can also use cv and other methods for determing the most optimal set. We do not have a test set at this point, however.

This is the same PCA code as earlier, providing the same *hesitant* visualizaation but now with our decided K = 28....

We did not keep all of the models in memory; recompute.

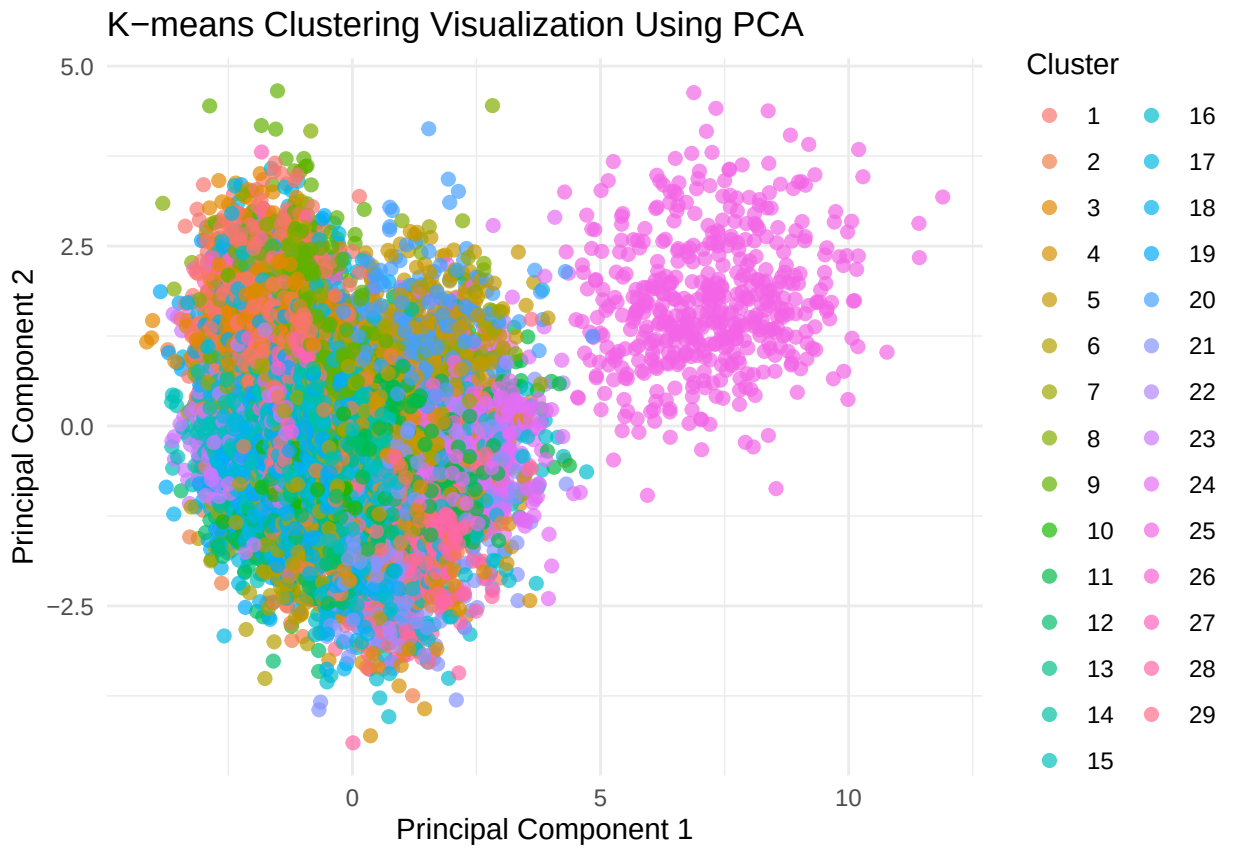
```
K <- 29
C <- k_means(seeds_preprocessed, K)
```

```
## Converged on 29 clusters after 30 iterations.
```

```
sil_score <- silhouette_score(seeds_preprocessed, C)
print(paste("Average Silhouette Score:", sil_score))
```

```
## [1] "Average Silhouette Score: 0.167529003693569"
```

```
pca_result <- prcomp(seeds_preprocessed, scale. = TRUE)
pca_data <- data.frame(pca_result$x[, 1:2])
colnames(pca_data) <- c("PC1", "PC2")

pca_data$Cluster <- as.factor(C)  # Directly use C for cluster assignments!
library(ggplot2)
ggplot(pca_data, aes(x = PC1, y = PC2, color = Cluster)) +
  geom_point(size = 2, alpha = 0.7) +
  theme_minimal() +
  labs(title = "K-means Clustering Visualization Using PCA",
       x = "Principal Component 1",
       y = "Principal Component 2") +
  scale_color_discrete(name = "Cluster")
```




We can't *really* use the above as a method for understanding our clusters; but it is clear that, at the least, there are defined areas per cluster. which holds... in R^2 it is simply not viable to display the 16d data.

**With a semi-labelled set?** There are a number of rows available in *seeds_annotated.c* who have human annotated assignments (of seed type). Before exploring this, we note that there are plausibly clusters which non of the seeds we've seen so far belong to and it is also plausible that the human annotating the seeds has made some mistake. We will, however, consider this now as the *ground truth* for our data.

Our first step is to *define* the clusters based on the annotated dataset.

For the sake of some of the helper functions, we consider the Kmeans method from the default R environment. K_means refers to our implementation, kmeans() to the inbuilt (and as such we convert to df... )

We define a kmeans prediction method as follows

```
predict_kmeans <- function(new_data, centroids) {

  num_centroids <- nrow(centroids)

  predicted_clusters <- numeric(nrow(new_data))

  # Calculate the distance from each new data point to each centroid
  for (i in 1:nrow(new_data)) {

    distances <- colSums((t(centroids) - new_data[i, ])^2, na.rm = TRUE)  # Euclidean distance
    predicted_clusters[i] <- which.min(distances)
  }

  return(predicted_clusters)
}
```

which just returns the closest cluster to the unseen row (rows)

We consider the following:

What are the classes of the known good data given our model? We load in the model (using the much more efficient inbuilt method) with the preprocessed df, we then return the cluster whose euclidean distance best indicates the clsuter assignment for the row in the annotated dataset...

```
known_seeds <- seeds_annotated
actual_assignments <- known_seeds$Class

known_seeds_no_prediction <- known_seeds[, !names(known_seeds) %in% "Class"]
preprocessed_known_seeds <- preprocess_data(known_seeds_no_prediction)

model <- kmeans(seeds_preprocessed, 28)
centroids <- model$centers
predicted_clusters <- predict_kmeans(preprocessed_known_seeds, centroids)



seeds_preprocessed_annotated <- cbind(preprocessed_known_seeds, Class = known_seeds$Class)
```

```
results_matrix <- matrix(nrow = nrow(preprocessed_known_seeds), ncol = 2)
colnames(results_matrix) <- c("Predicted_Cluster", "Actual_Class")
results_matrix[, 1] <- predicted_clusters  # Predicted cluster assignments
results_matrix[, 2] <- actual_assignments[!is.na(actual_assignments)]  # Actual class labels

results_df <- as.data.frame(results_matrix)
```

```
grouped_results <- aggregate(Predicted_Cluster ~ Actual_Class, data = results_df,
                             FUN = function(x) {
                               # Sort all predicted clusters without removing duplicates
                               paste(sort(x), collapse = ", ")
                             })

print(grouped_results)
```

```
##   Actual_Class
## 1            A
## 2            B
## 3            C
## 4            D
## 5            E
## 6            F
## 7            G
##                                                                Predicted_Cluster
## 1                                                 11, 11, 13, 14, 14, 21, 23, 9
## 2                                                                 8, 8, 8, 8, 8
## 3                  10, 11, 11, 12, 12, 14, 14, 18, 18, 2, 23, 26, 28, 5, 5, 5
## 4 10, 15, 16, 19, 2, 2, 2, 2, 21, 21, 21, 25, 25, 25, 25, 25, 25, 25, 4, 5, 7
## 5                                          14, 14, 14, 14, 14, 18, 25, 28, 5
## 6  13, 15, 16, 16, 16, 16, 16, 17, 2, 20, 24, 24, 24, 25, 3, 4, 4, 4, 4, 7, 7
## 7      12, 13, 14, 15, 2, 2, 22, 25, 25, 25, 25, 25, 25, 28, 3, 4, 4, 5, 7, 9
```

We see above the clustering assignment for our clustering on the new data. Some clusters are very good, i.e. 17 is well defined i.e. seed type *B*. There is, however overlap across most of the other clusters in our broad model.

This indicates that K = 28 is insufficient for determining the model well.

We could make a *better* model (i.e one whose misclassification is lower for the true clusters) by...

We will build a *Semi Supervised Random Forrest* model. We consider that we have the labelled and unlabelled set.

```
seeds_preprocessed_annotated <- as.data.frame(seeds_preprocessed_annotated)
  seeds_preprocessed_annotated$Class <- as.factor(seeds_preprocessed_annotated$Class)
dim(seeds_preprocessed_annotated)
```

```
## [1] 100  17
```

```
dim(seeds_preprocessed)
```

```
## [1] 13511    16
```

We have the above two dataframes; where the first contains the annotataed set with an extra column defining the class.

```
train_indices <- sample(1:nrow(seeds_preprocessed_annotated), size = 0.8 * nrow(seeds_preprocessed_anno

train_data <- seeds_preprocessed_annotated[train_indices, ]
```

```r
test_data <- seeds_preprocessed_annotated[-train_indices, ]

# Fit Random Forest model
rf_model <- randomForest(Class ~ ., data = train_data, importance = TRUE)


predictions <- predict(rf_model, test_data)
confusion_matrix <- table(test_data$Class, predictions)

print(confusion_matrix)
```

```
##    predictions
##     A B C D E F G
##   A 0 0 0 0 0 0 2
##   B 0 0 0 0 0 0 2
##   C 0 0 0 0 0 0 6
##   D 0 0 0 0 0 0 2
##   E 0 0 0 0 0 0 1
##   F 0 0 0 0 0 0 5
##   G 0 0 0 0 0 0 2
```

```r
# Calculate metrics (copy/ paste from ealier)
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
sensitivity <- confusion_matrix[2, 2] / sum(confusion_matrix[2, ])
specificity <- confusion_matrix[1, 1] / sum(confusion_matrix[1, ])  #
misclassification_rate <- 1 - accuracy

#  F1 Score
precision <- confusion_matrix[2, 2] / sum(confusion_matrix[, 2])  # True Positive / (True Positive + Fa
F1_score <- 2 * (precision * sensitivity) / (precision + sensitivity)

cat("Accuracy:", accuracy, "\n")
```

```
## Accuracy: 0.1
```

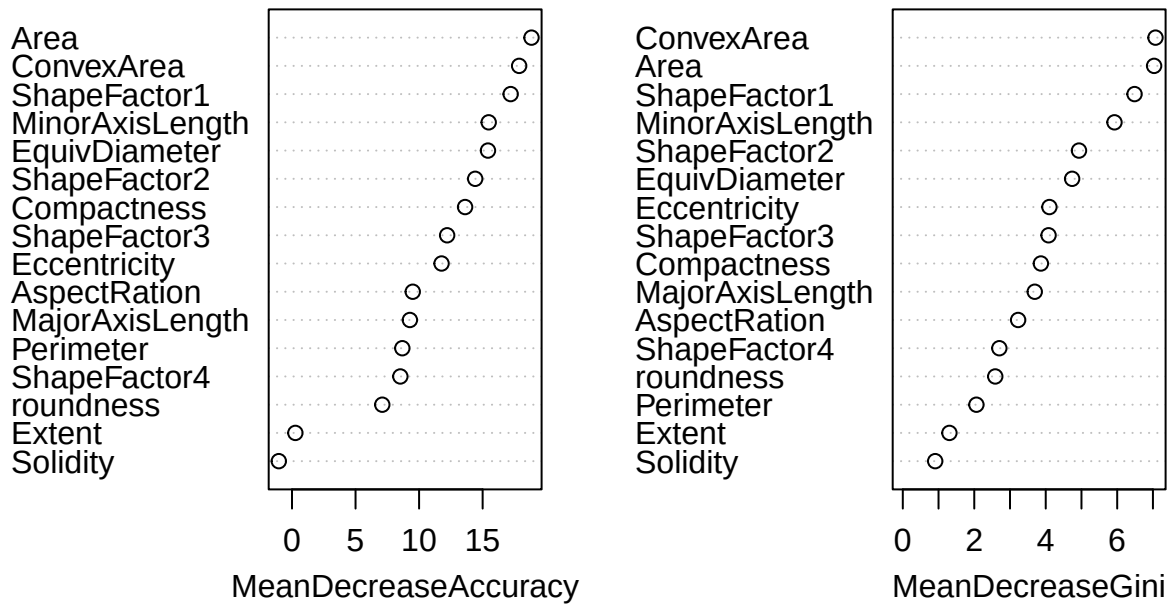
```r
cat("Misclassification Rate:", misclassification_rate, "\n")
```

```
## Misclassification Rate: 0.9
```

```r
# Number of trees
cat("Number of Trees in the Random Forest Model:", rf_model$ntree, "\n")
```

```
## Number of Trees in the Random Forest Model: 500
```

```r
# Variable Importance
varImpPlot(rf_model)
```




We choose to use random forrests to leverage our small (100 : 13500) amount of labelled data. We still split said data into 80:20 (which is not a sufficient test amount). Random forrests are a good choice here because they learn the underlying patterns over multiple trees and combines those for outputs. This makes this approach, with the labelled data a better architectural choice for our predictor than K means, generally.

Comparing the results (above) to our kmeans algorithm is challenging becacuse the scope of misclassification etc is more ill defined (no labels on the clustering set, just metrics of distance as discussed earlier).

However, we do have convienient plots above for the effect which each variable has on our classifier. I.e we can read off which variable most decreases gini (zero gini = perfect classification) and also accuracy decrease. Convex area is the most telling factor, for example.

We also consider: the model output: With accuracy at just 15%.. this indicates that the model is struggling to correctly classify instances, with a high misclassification rate suggesting it may need further tuning or additional data to improve performance.

We have thus far made the assumption that our only plausible test data is the annotated set.

A prudent next step is to treat the clusters as truth and train the same network over many folds; treating the true set as a validation set that is never tested on nor involved in the training of the set. The steps to do this are trivial when derived