Analysis Fall 23

Abhi Thanvi, Jonathan Sneh

2023-12-04

Contents

Project Overview

We highly recommend everyone to checkout our Github Repository for all the data cleansing, feature engineering, analysis, and other files! Many of our procedures are not included in the report, and the repo provides a behind-the-scenes access to that information! We understand it is important to understand each procedure but also reproduce results, therefore we have maintained a highly intuitive code-base for it! :)

Goals:

While this project is an assigned project for STAT 432 Fall 2023 (UIUC), our goal is to apply what we have learned in our class and generate not necessarily the "most accurate", but rather the most holistic solution on this unique problem. The topic we are dealing with Linking Writing Processes to Writing Quality where we explore data on typing behavior to predict essay quality between a score of 0-6 (inclusive) using many of the statistical learning techniques. Our work will help explore the relationship between learners' writing behaviors and writing performance, which could provide valuable key insights for writing instruction, the development of automated writing evaluation techniques, and help in educational situations.

Approach:

We divide our work into three main section.

- **Data Processing** → Cleanse, extract, and engineer features for our Supervised and Unsupervised learning portion. Our data processing procedure also has some basic EDA work done to allow us to engineer valuable features.
- **Unsupervised Learning** → Perform clustering algorithms on the cleansed data (and 80-20 split). We chose to do K-Means and Hierarchical Clustering algorithms as we wanted to explore what we learned in class. These unsupervised techniques allow us to find hidden patterns in our data and act as an outlet for advanced EDA. How the clusters were chosen, what insights we drew, the good and bad about these clusters will all be explored later in this section.
- **Regression/Classification Models** → It is in this section where we try to predict scores and aim to achieve our original goal. **JONATHAN ADD YOUR STUFF HERE** :D

Unique Approaches/Techniques:

We generally tried to use as much as we could from our STAT 432 course materials as the source of knowlegde. There were moments where we did consult other topics such as pairing elbow method with Silhouette Plots (to determine how well the cluster fit), **JONATHAN ADD YOUR STUFF HERE**...prolly the STAT 426 stuff? :D

Conclusion: **TBD**

Literature Review

Coming soon :'(

Data Processing

This section dives into the tasks performed for data processing. All the steps ensure the specifications of the projects were met, but some decisions were also made to ensure a more practical data to work with. To be considerate of the pages used for the Data Processing, we performed our Data Engineering steps in a `jupyter notebook` that you can view in our repo!

Feature Engineering

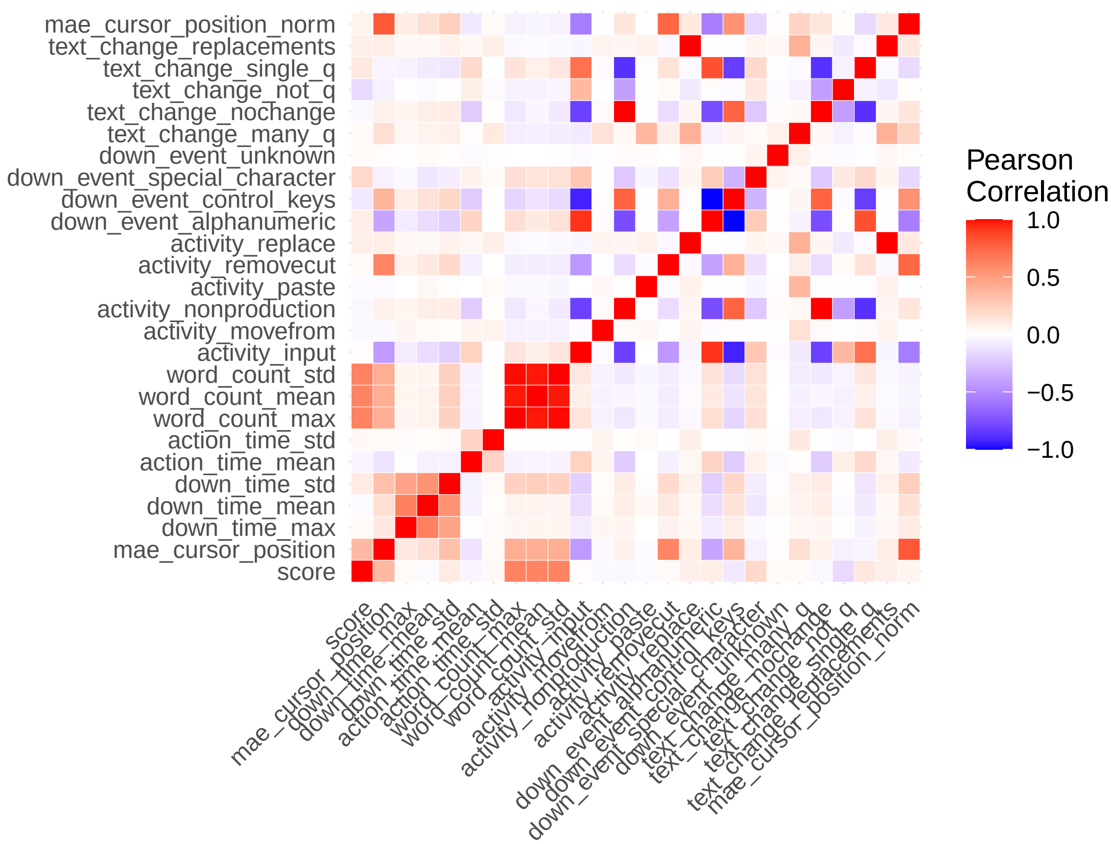
- **User ID** [`id`, `string`] — Unique IDs of each user.
  - We keep this to ensure tracking of user information for processing and analysis work.
- **Event ID** [`event_id`, `string`] — Incremental ID log of all events.
  - We keep this for processing steps, but remove it prior to analysis. The event IDs are useful as an ordinal feature of the log data.
- **Down Time / Up Time** [`down_time` / `up_time`, `integer`] — Time of event on down and up strokes of key or button, in seconds.
  - We summarize these features as an array of summary statistics; min, max, mean, median, and standard deviation. Measures of interest are max (i.e., how long a paper is written) and mean/median (i.e., when the center of most activity is).
- **Action Time** [`action_time`, `integer`] — Difference of time between down time and up time of event, i.e., duration of action in seconds.
  - Similarly, we summarize this feature as min, max, mean, median, and std. This gives insight into "major" consecutive actions, hesitancy, or other special behaviors.
- **Activity** [`activity`, `string`] — Actions to edit or modify the text (input, remove/cut, nonproduction, etc.)
  - We compute the proportions of each of these activities. All of the cursor "Move From" events are mapped to one category called "Move From". We choose proportions over count to avoid undue influence of essays that take longer to write.
- **Down Event**
  - We compute the proportions of each of the activities. The events were pooled into four categories: alphanumeric, special_characters, control_keys, and unknown.
- **Up event**
  - Since these are the same events as down events, we ignore this feature.
- **Text Change**
  - We process and cluster these values into identified patterns of changes: many characters (at least 2 alphanumeric), at least one character (exactly one alphanumeric), non-zero characters (no alphanumeric). We also identified "transition" groups of "X to Y" for each of "many", "single", "none" (e.g., "many" to "many", "many" to "single", "many" to "none", etc.). There was also a "no change" group. We created one additional group to represent the sum of all "transition" events because they coincided exclusively with "replacement" activities.
- **Cursor Position**
  - We computed an artificial array of cursor positions with the assumption that the text was streamed with no edits corresponding to what text changes there are (i.e., non-decreasing and doesn't change if "no change" is observed in text change feature). Then we compute the MAE error metric between this stream version and the actual cursor positions to measure how much error exists between them. Greater errors imply more frequent and/or drastic changes.
- **Word Count**
  - We summarize these features as an array of summary statistics; min, max, mean, median, and standard deviation. We are primarily interested in the maximum measure as it indicates the length of the paper for each user.

Data Summary

Here is a sample of the processed data.

	001519c8	0022f953	0042269b	0059420b	0075873a	0081af50
id	001519c8	0022f953	0042269b	0059420b	0075873a	0081af50
score	3.5	3.5	6.0	2.0	4.0	2.0
mae_cursor_position	527.0469	380.7747	1238.7553	152.3933	640.1616	423.8706
down_time_max	1801877	1788842	1771219	1404394	1662390	1778845
down_time_mean	848180.8	518855.3	828491.8	785483.0	713354.2	544339.2
down_time_std	395112.7	384959.4	489500.8	385205.0	405576.4	484650.6
action_time_mean	116.24677	112.22127	101.83777	121.84833	123.94390	81.40434
action_time_std	91.79737	55.43119	82.38377	113.76823	62.08201	40.65305
word_count_max	256	323	404	206	252	275
word_count_mean	128.1162	182.7148	194.7727	103.6189	125.0830	132.9426
word_count_std	76.49837	97.76309	108.93507	61.88225	77.25505	81.20882
activity_input	0.7860774	0.7897311	0.8498549	0.8380463	0.7672857	0.8113976
activity_movefrom	0.00117325	0.00000000	0.00000000	0.00000000	0.00000000	0.00000000
activity_nonproduction	0.04693000	0.10350448	0.04231141	0.06362468	0.02844725	0.03437359
activity_paste	0.0000000000	0.0004074980	0.0000000000	0.0006426735	0.0000000000	0.0000000000
activity_removecut	0.1630817	0.1059495	0.1061412	0.0970437	0.2042671	0.1528720
activity_replace	0.0027375831	0.0004074980	0.0016924565	0.0006426735	0.0000000000	0.0013568521
down_event_alphanumeric	0.6331639	0.6071720	0.7021277	0.6709512	0.6088503	0.6562641
down_event_control_keys	0.3523661	0.3712306	0.2860251	0.3155527	0.3646780	0.3364993
down_event_special_character	0.014470082	0.021597392	0.011847195	0.013496144	0.026471750	0.007236545
down_event_unknown	0	0	0	0	0	0
text_change_many_q	0.0007821666	0.0000000000	0.0007253385	0.0006426735	0.0000000000	0.0004522840
text_change_nochange	0.04693000	0.10350448	0.04231141	0.06362468	0.02844725	0.03437359
text_change_not_q	0.1908487	0.2041565	0.1677950	0.1985861	0.1955749	0.1804613
text_change_single_q	0.7587016	0.6919315	0.7874758	0.7365039	0.7759779	0.7833559
text_change_replacements	0.0027375831	0.0004074980	0.0016924565	0.0006426735	0.0000000000	0.0013568521

Correlation Summary



Discussion

We observe some pairs of features that show signs of multicolinearity.

- The word count metrics are highly correlated as expected, we could reasonably choose the maximum measure to use.
- Some features form parallel or perpendicular colinearity.
  - activity_input and activity_nonproduction (negative)
  - activity_input and down_event_alphanumeric (positive)
  - activity_input and down_event_control_keys (negative)
  - activity_input and text_change_nochange (negative)
- Importantly, we're interested in what's correlated with the user score feature
  - word count measures have a positive correlation with score, suggesting an association between longer essays and higher scores
  - Error rate of cursor positions against a "streamed" output also shows a positive correlation with score - i.e., essays written with less frequent or extreme edits is somewhat associated with higher scores.
  - Note: a positive correlation is also found between error rate of cursor positions with max word count, suggesting further that longer essays are associated with higher deviation from a "streamed" output. This suggests the possibility that interpretation of "streamed" deviation is influenced by the paper length (i.e., longer papers support possibility of edits being made "further away" from the current "streamed" position, thus increasing the error rate). When we normalized the error rate by the paper size, we see that the correlation between the normalized error rate and the paper score is nearly zero. So, this feature is likely irrelevant for analysis.

Unsupervised Learning Algorithms

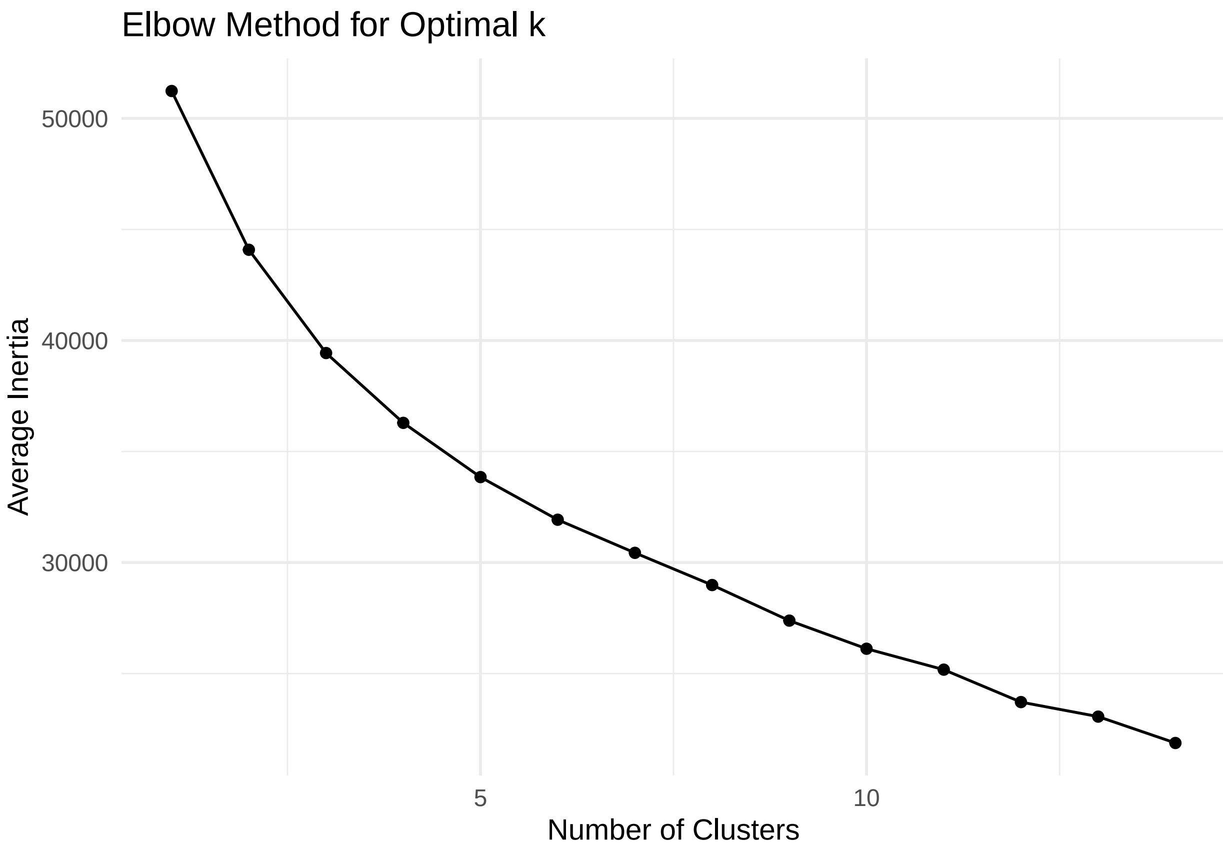
This section dives into the tasks performed for the unsupervised learning algorithms. Currently, focusing on K-Means and Hierarchical Clustering. **THIS SECTION SUPER MESSY RN. Please feel free to edit or improve in any way**

```
# test train split
test_ids = sample(1:nrow(df), as.integer(0.2 * nrow(df)))
data = as.data.frame(scale(df[, -(1:2)]))
data = cbind(score=df$score, data)
train = data[-test_ids, ]
test = data[test_ids, ]
```

K-Means Algorithm

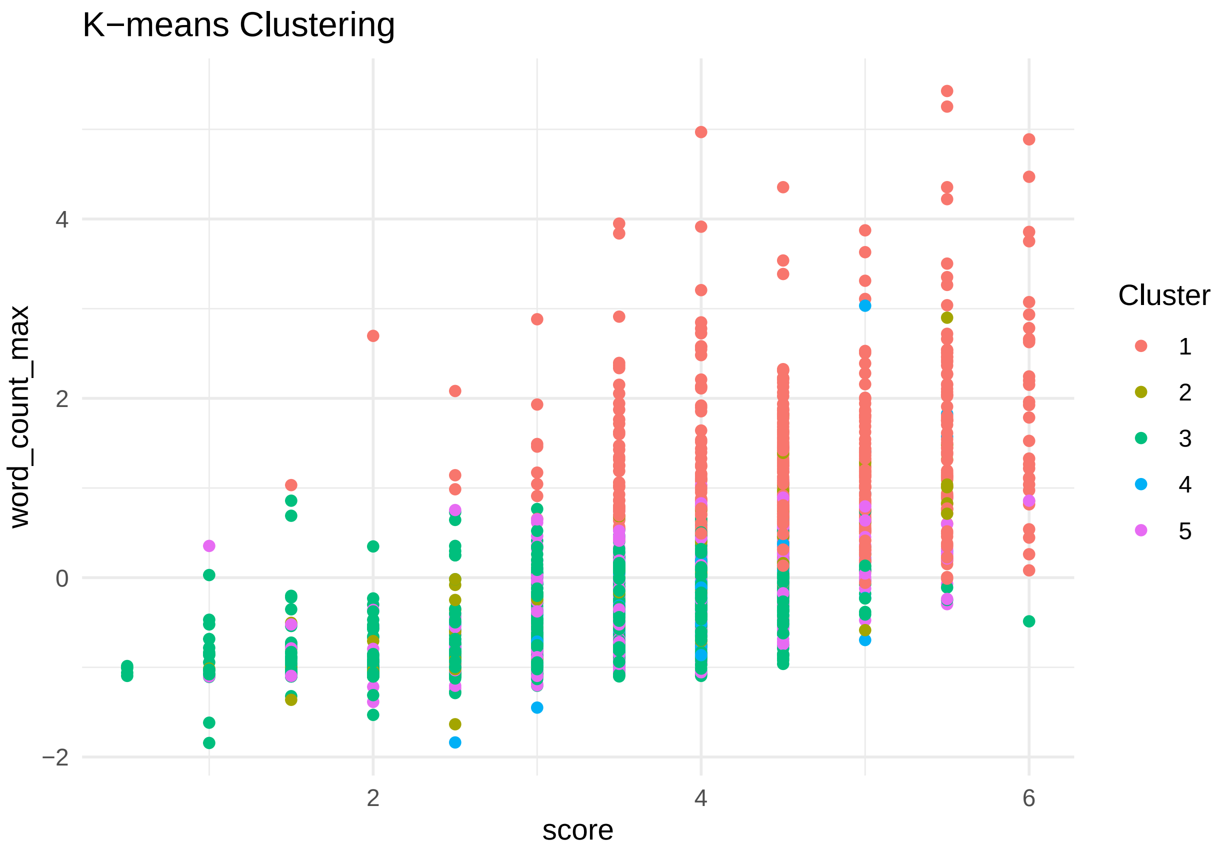
using averaged inertia of a few clustering samples across a range of number of clusters (i.e., k=1, . . . , 14)

```
# Range of k values to try
cluster_num_list <- 1:14

# Initialize a vector to store average inertias
avg_inertia_list <- numeric(length(cluster_num_list))

# Iterate over different values of k
for (k in cluster_num_list) {
  sub_inertia_list <- numeric(3) # For storing inertia of each trial

  for (i in 1:3) {
    set.seed(i) # Setting seed for reproducibility
    kmeans_result <- kmeans(train, centers=k, nstart=25, iter.max = 50)
    sub_inertia_list[i] <- kmeans_result$tot.withinss
  }

  avg_inertia_list[k] <- mean(sub_inertia_list)
}

# Plotting the elbow plot
ggplot(data.frame(Clusters=cluster_num_list, Inertia=avg_inertia_list),
       aes(x=Clusters, y=Inertia)) +
  geom_line() +
  geom_point() +
  theme_minimal() +
  ggtitle("Elbow Method for Optimal k") +
  xlab("Number of Clusters") +
  ylab("Average Inertia")
```



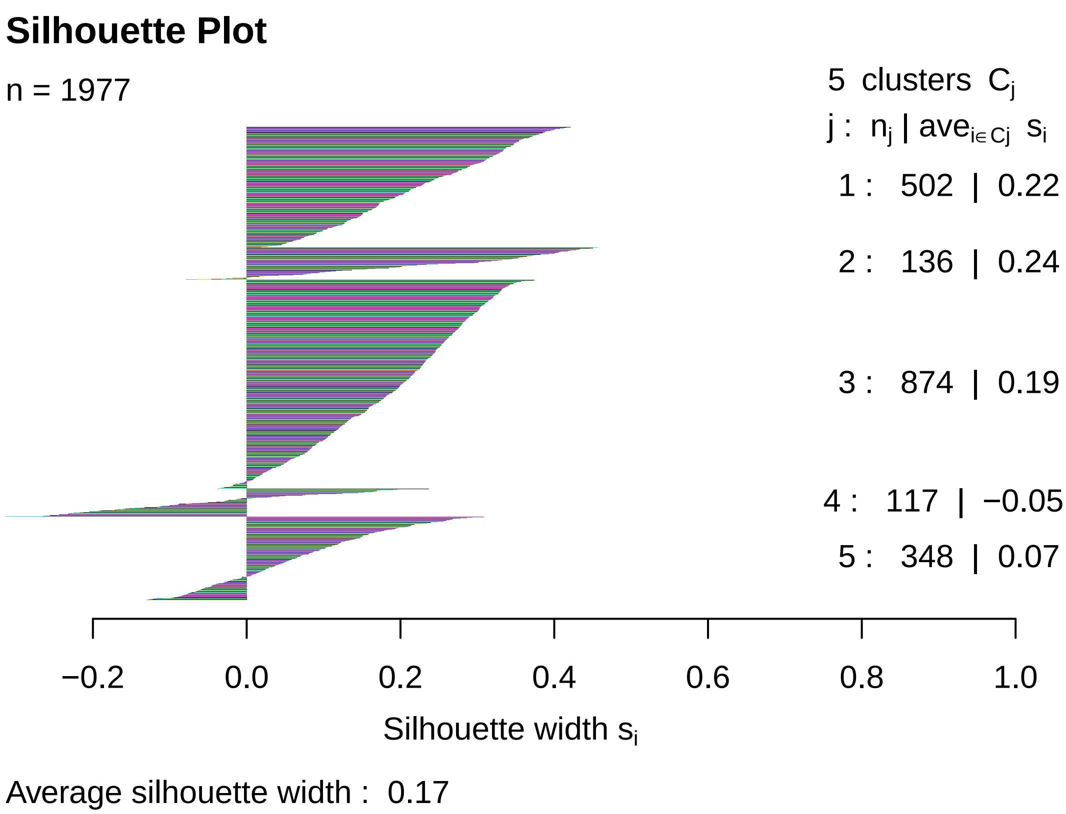
selected `k=5` clusters as closest to "elbow" of the inertia plot

```
# Perform K-means clustering
# Here, we are specifying 3 clusters, but you can change this number
result <- kmeans(train, centers=5)

ggplot(data.frame(train), aes(x=score, y=word_count_max)) +
  geom_point(aes(color=factor(result$cluster))) +
  scale_color_discrete(name="Cluster") +
  theme_minimal() +
  ggtitle("K-means Clustering")
```



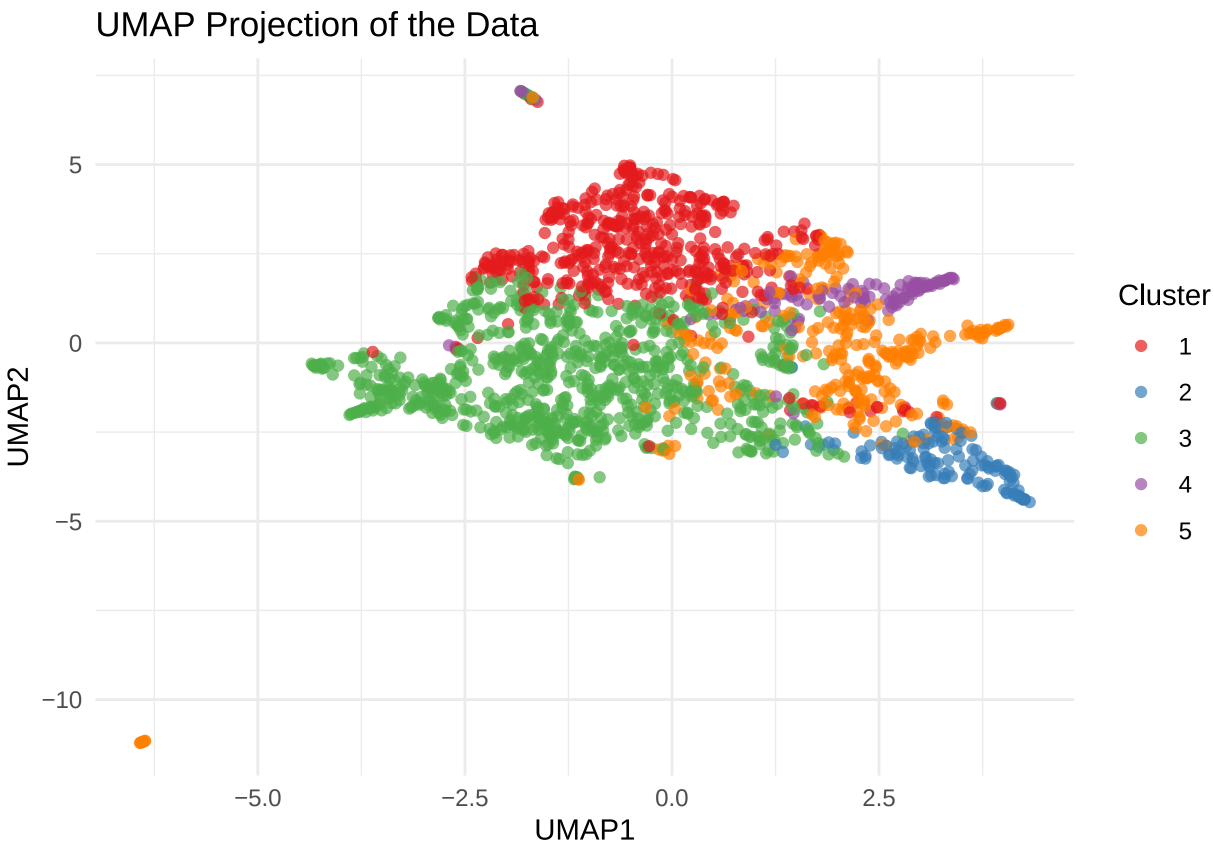
Using silhouette scores to evaluate how well the clustering structure fits in terms of similarity, i.e., more higher scores in a cluster imply greater similarity of points to its own cluster and poorer similarity to other clusters. The average silhouette score of 0.17 suggests that the clusters are somewhat favorably well separated and that the points within clusters aren't too dispersed. However, cluster 4 shows issues with cohesion and separation.

```
# Compute silhouette information
silhouette_info <- silhouette(result$cluster, dist(train))

# Plotting the silhouette plot
plot(silhouette_info, col=1:k, border=NA, main="Silhouette Plot")
```



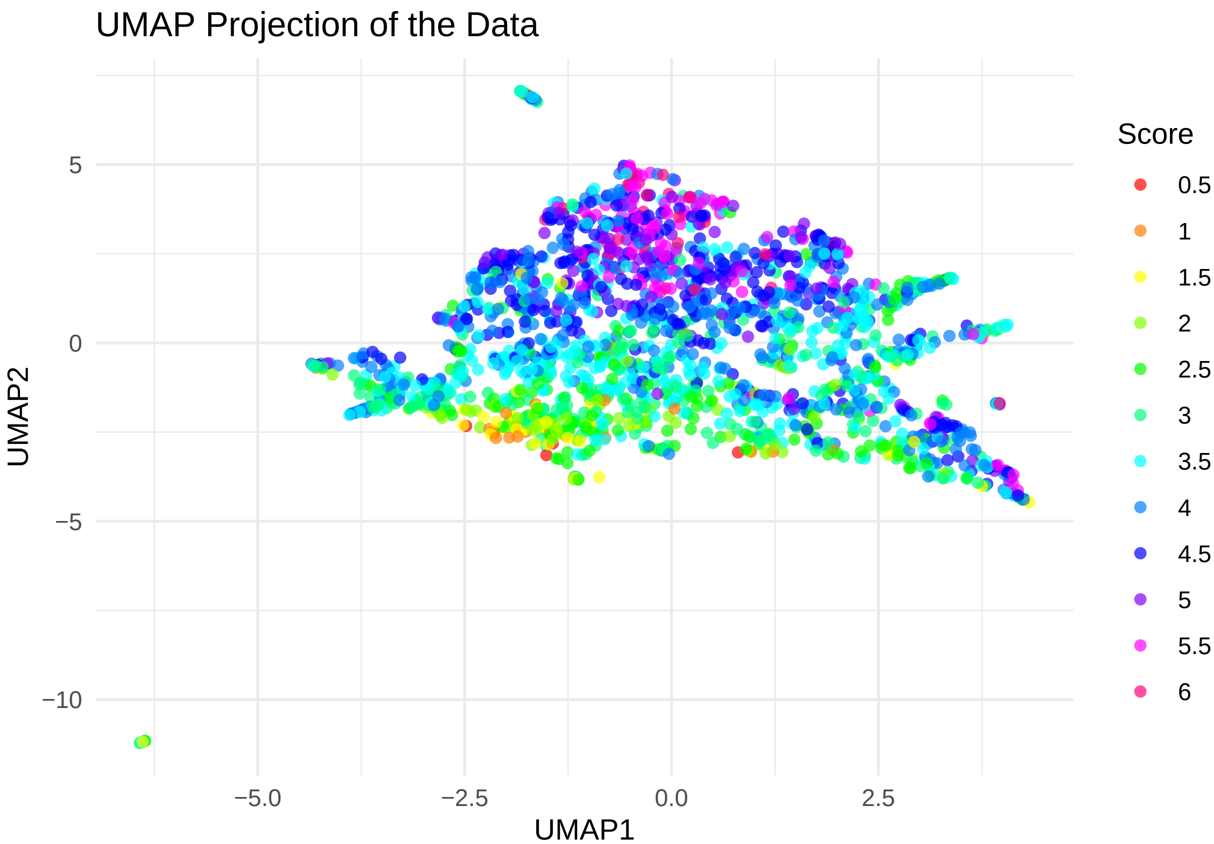
Additionally, with projecting the data using UMAP, we see that the clusters might not be very well separated and aren't globular, so it is reasonable to conclude that KMeans algorithm may struggle with this data.

```
umap_result <- umap(train, n_components = 2)

umap_data <- as.data.frame(umap_result$layout)
colnames(umap_data) <- c("UMAP1", "UMAP2")

clusters <- as.factor(result$cluster)
umap_data$Cluster <- clusters

# Choose a palette
palette <- brewer.pal(n = 5, name = "Set1")  # Adjust 'name' as needed

ggplot(umap_data, aes(x = UMAP1, y = UMAP2, color = Cluster)) +
  geom_point(alpha = 0.7) +
  scale_color_manual(values = palette) +
  theme_minimal() +
  ggtitle("UMAP Projection of the Data")
```



Here's distribution of scores in UMAP

```
scores <- as.factor(as.data.frame(train)$score)
umap_data$Score <- scores

ggplot(umap_data, aes(x = UMAP1, y = UMAP2, color = Score)) +
  geom_point(alpha = 0.7) +
  scale_color_manual(values = rainbow(12)) +
  theme_minimal() +
  ggtitle("UMAP Projection of the Data")
```



investigation of the score distribution suggests that the underlying clustering structure of the data does not closely align with the structure of scores.

spread of data between kmeans clusters and scores. characterizations are as follows:

- cluster 1 tends to capture those who score 3 to 6.
- cluster 2 tends to capture those who score between 3.5 to 4.5
- cluster 3 tends to capture those who score 1.5 to 4.5
- cluster 4 tends to capture those who score 3.5 to 4
- cluster 5 tends to capture those who score 2.5 to 4.5

e.g., a user who scores around 5 is likely to be in cluster 1

while there are some relationships, it would appear that the dispersions of scores between groups tend to overlap heavily and are not well separate to segment the groups in a very meaningful way. However clusters 1 and 2 vs. clusters 3, 4, and 5 seem to show some disparity.

```
t(table(result$cluster, df[-test_ids, ]$score)) / nrow(df[-test_ids, ])
```

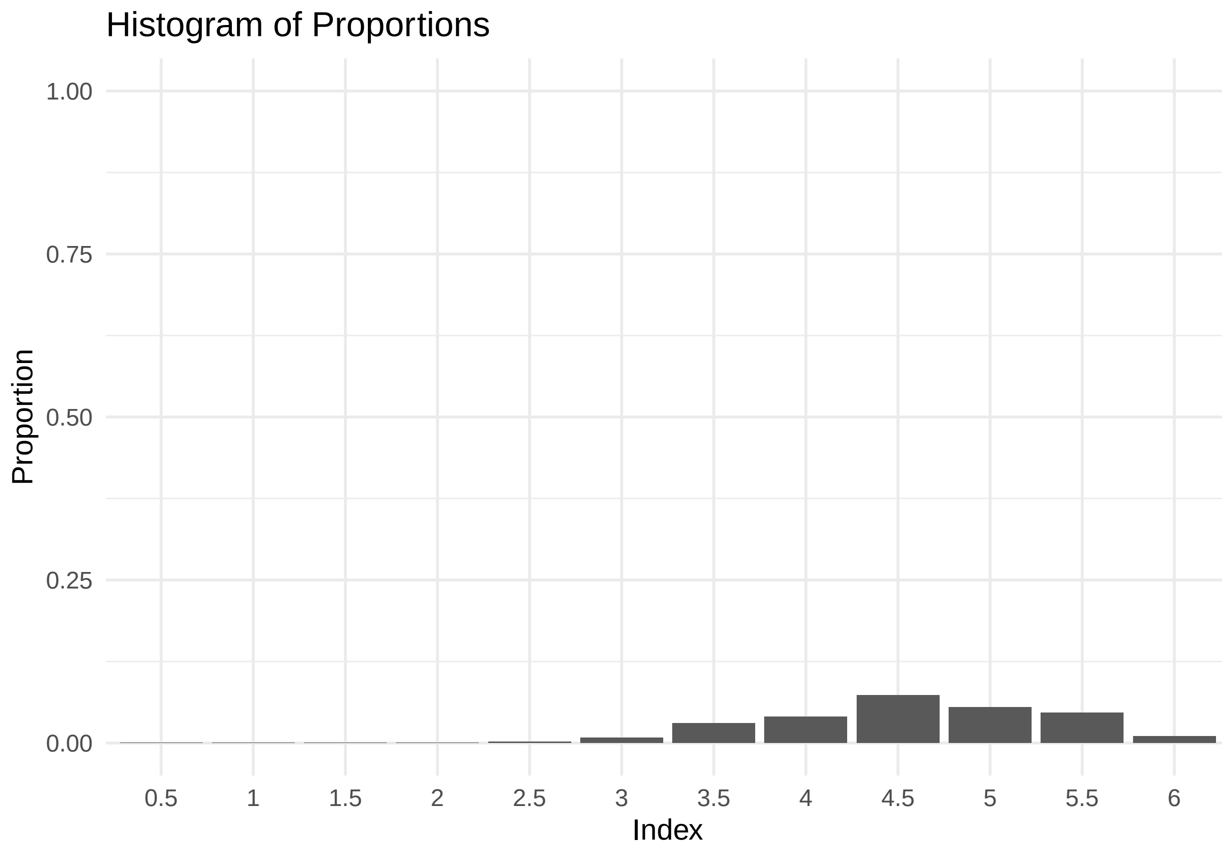
```
##
##                 1            2            3            4            5
##   0.5 0.0000000000 0.0000000000 0.0020232676 0.0000000000 0.0000000000
##   1   0.0000000000 0.0005058169 0.0096105210 0.0000000000 0.0015174507
##   1.5 0.0005058169 0.0020232676 0.0227617602 0.0000000000 0.0030349014
##   2   0.0005058169 0.0030349014 0.0293373799 0.0000000000 0.0055639858
##   2.5 0.0015174507 0.0091047041 0.0490642387 0.0060698027 0.0161861406
##   3   0.0040465352 0.0080930703 0.0875063227 0.0065756196 0.0293373799
##   3.5 0.0212443096 0.0111279717 0.1087506323 0.0141628730 0.0419828022
##   4   0.0450177036 0.0171977744 0.0859888720 0.0192210420 0.0359129995
##   4.5 0.0778958017 0.0101163379 0.0394537178 0.0070814365 0.0293373799
```

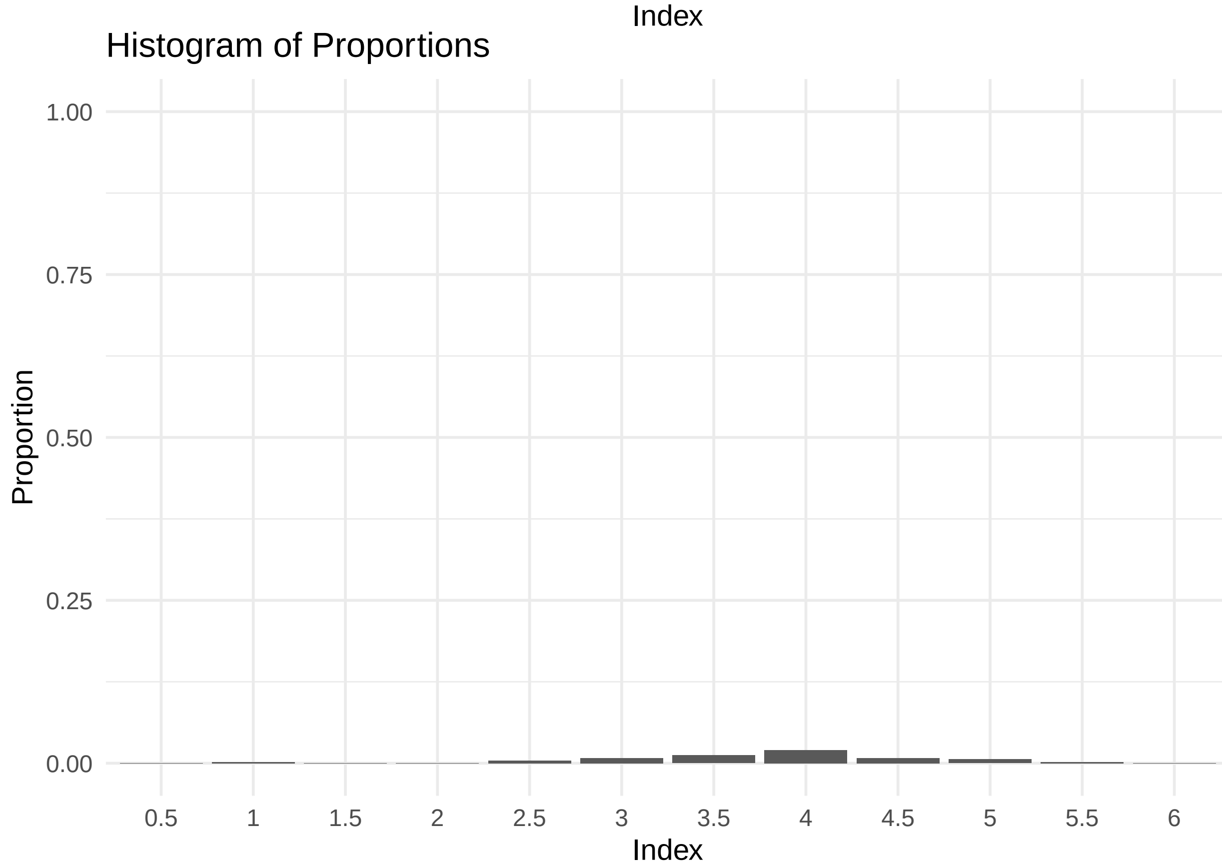
```
##   5   0.0470409712 0.0040465352 0.0060698027 0.0045523520 0.0080930703
##   5.5 0.0409711684 0.0035407183 0.0010116338 0.0015174507 0.0045523520
##   6   0.0151745068 0.0000000000 0.0005058169 0.0000000000 0.0005058169
```

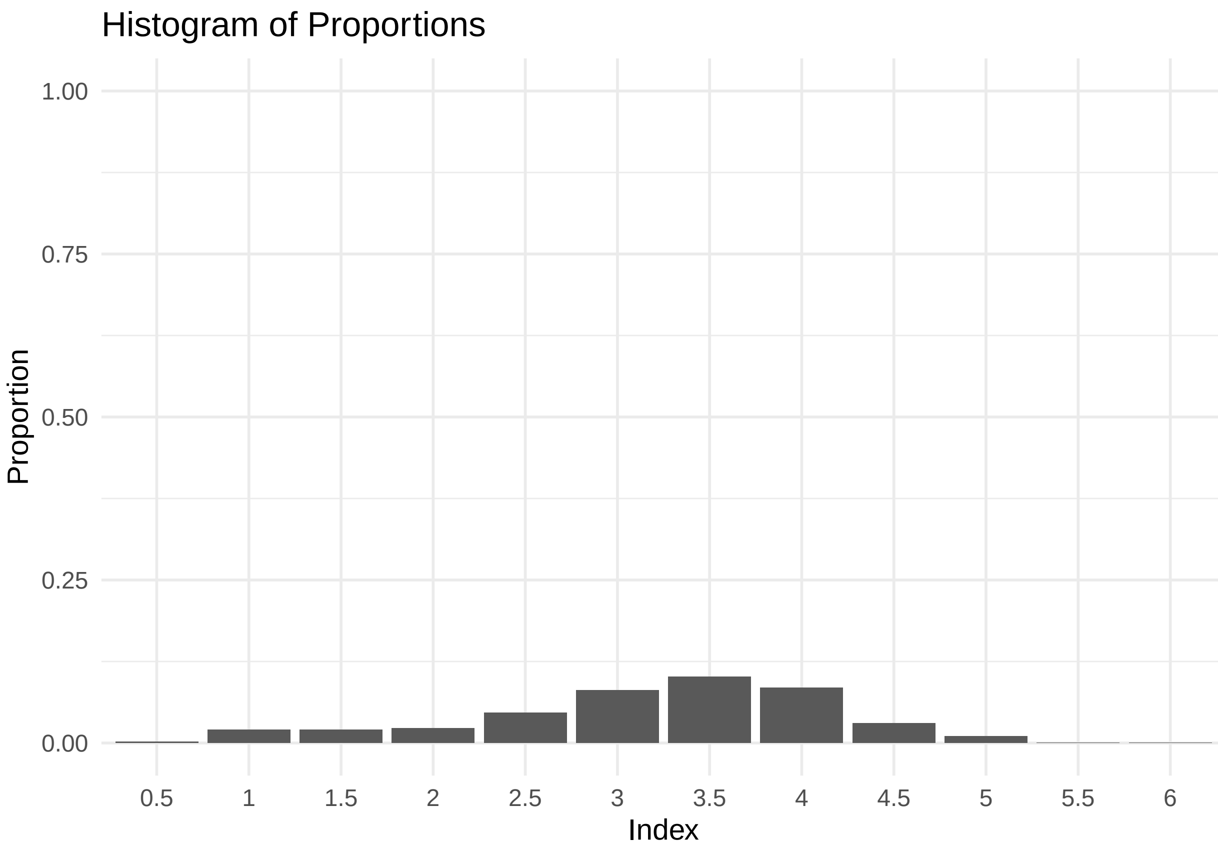
to predict on new data in test…

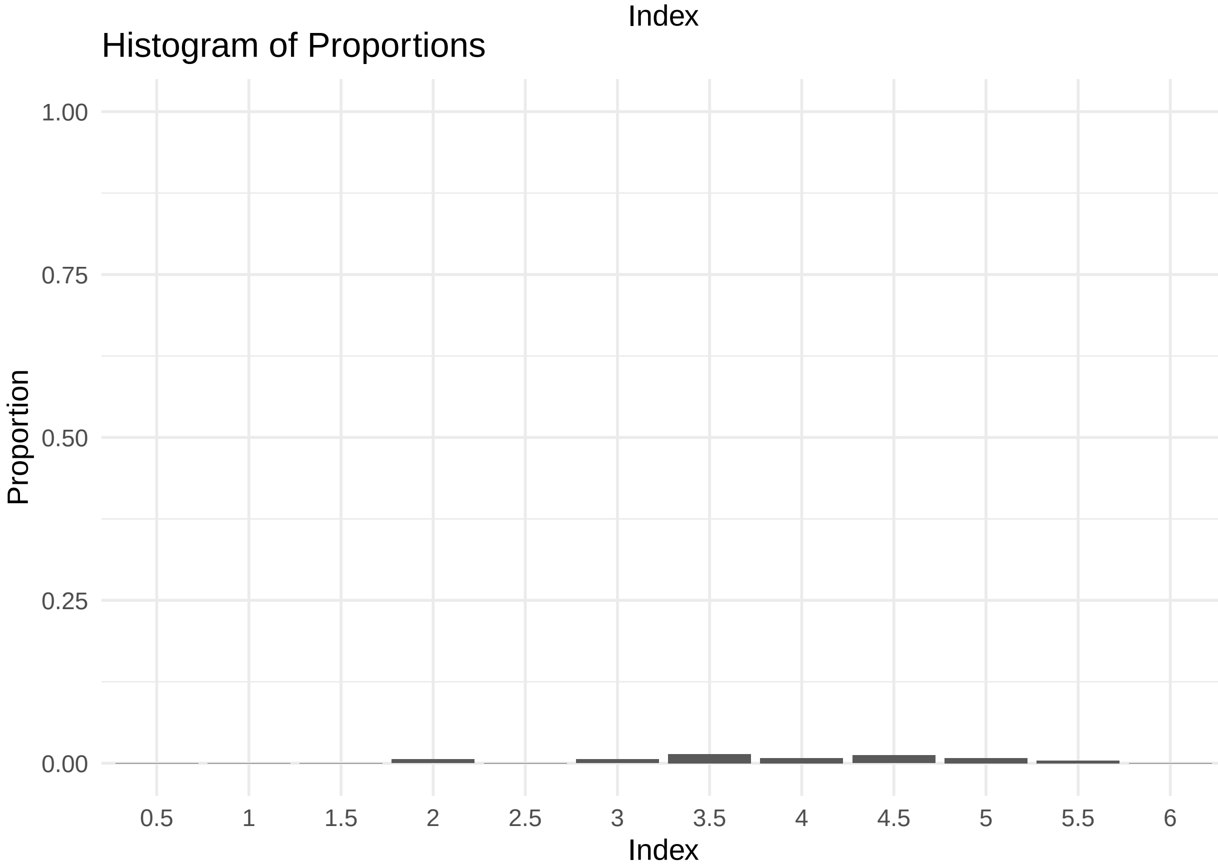
```r
assign_cluster <- function(new_data, centers) {
  # Calculate Euclidean distances from each new data point to each cluster center
  distances <- as.matrix(dist(rbind(centers, new_data), method = "euclidean"))
  distances <- distances[(nrow(centers)+1):nrow(distances), 1:nrow(centers)]

  # Assign each new data point to the nearest cluster
  max.col(-distances)
}

new_clusters <- assign_cluster(test, result$centers)
```

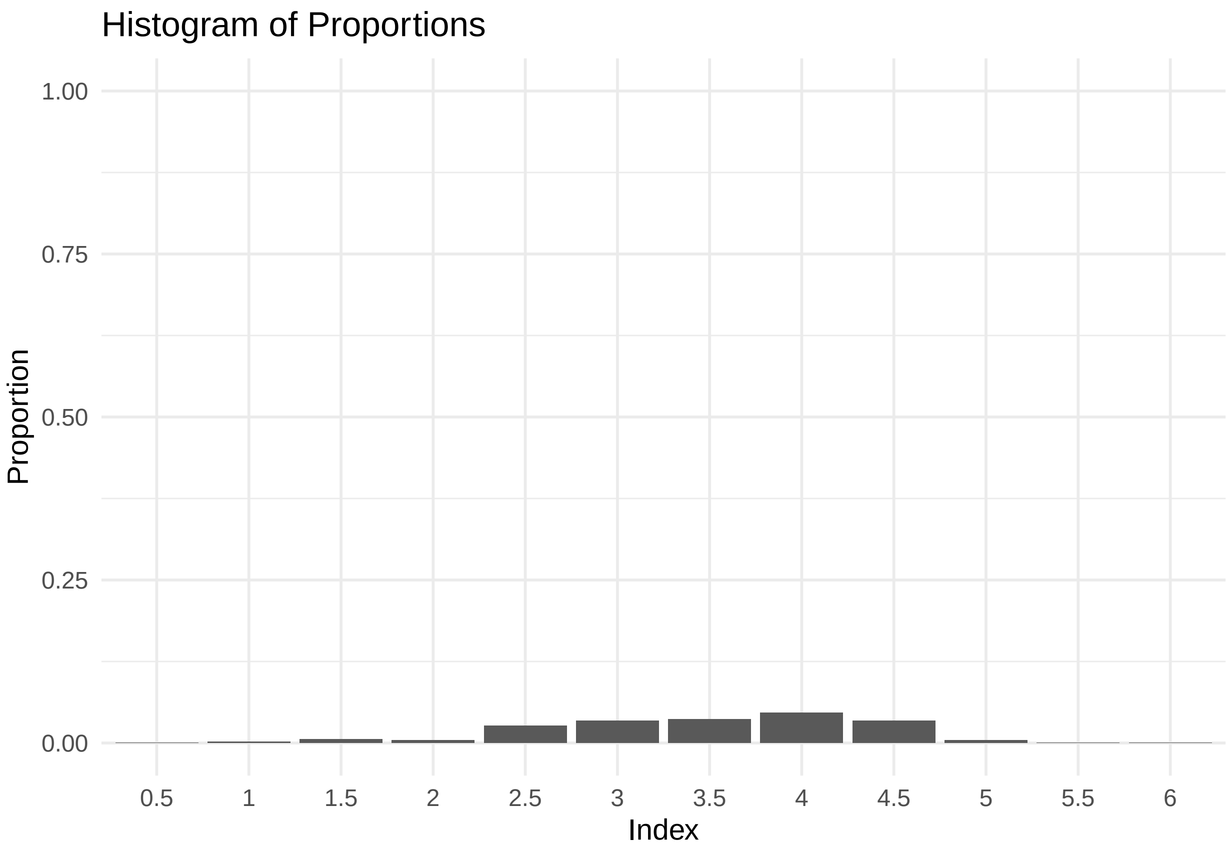
```r
acc_data = as.matrix(
  t(table(new_clusters, df[test_ids, ]$score)) / nrow(df[test_ids, ])
)

for (i in 1:5) {
  prop_plot = ggplot(
  data.frame(
      Index = rownames(acc_data), Proportion = acc_data[, i]
    ),
  aes(x = Index, y = Proportion)) +
  geom_bar(stat = "identity") +
  theme_minimal() +
  xlab("Index") +
  ylab("Proportion") +
  ggtitle("Histogram of Proportions") +
  ylim(c(0, 1))
  print(prop_plot)
}
```


Histogram of Proportions
Proportion
Index
1.00
0.75
0.50
0.25
0.00
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6


Histogram of Proportions
Proportion
Index
1.00
0.75
0.50
0.25
0.00
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6

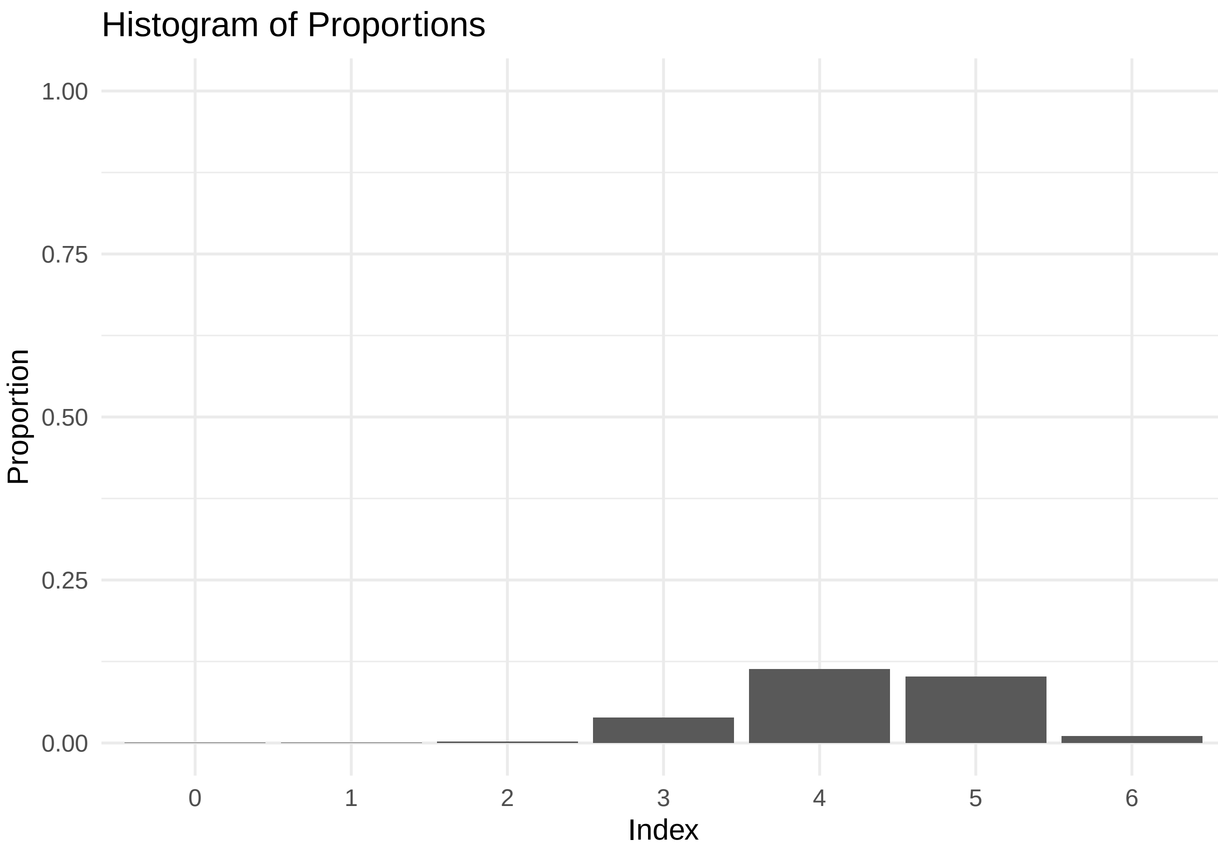

Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
Index

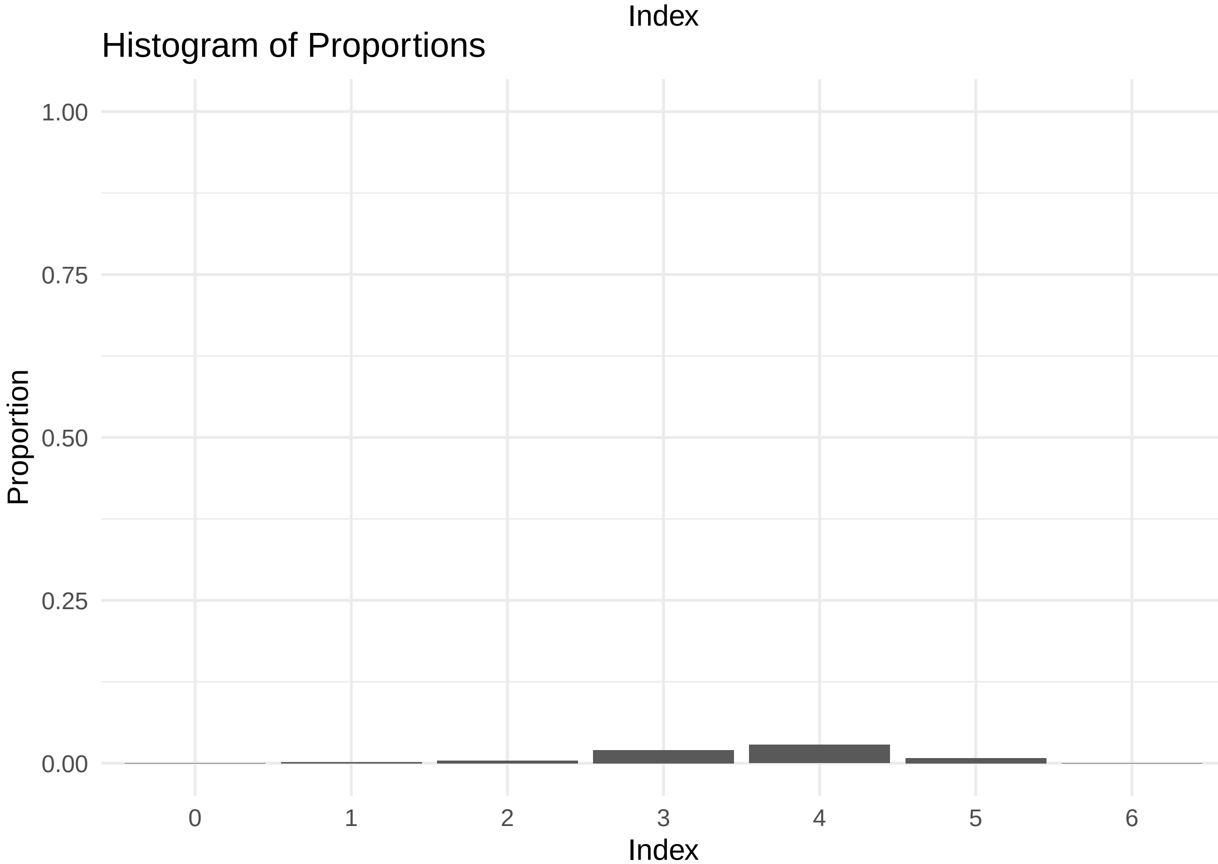

Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
Index

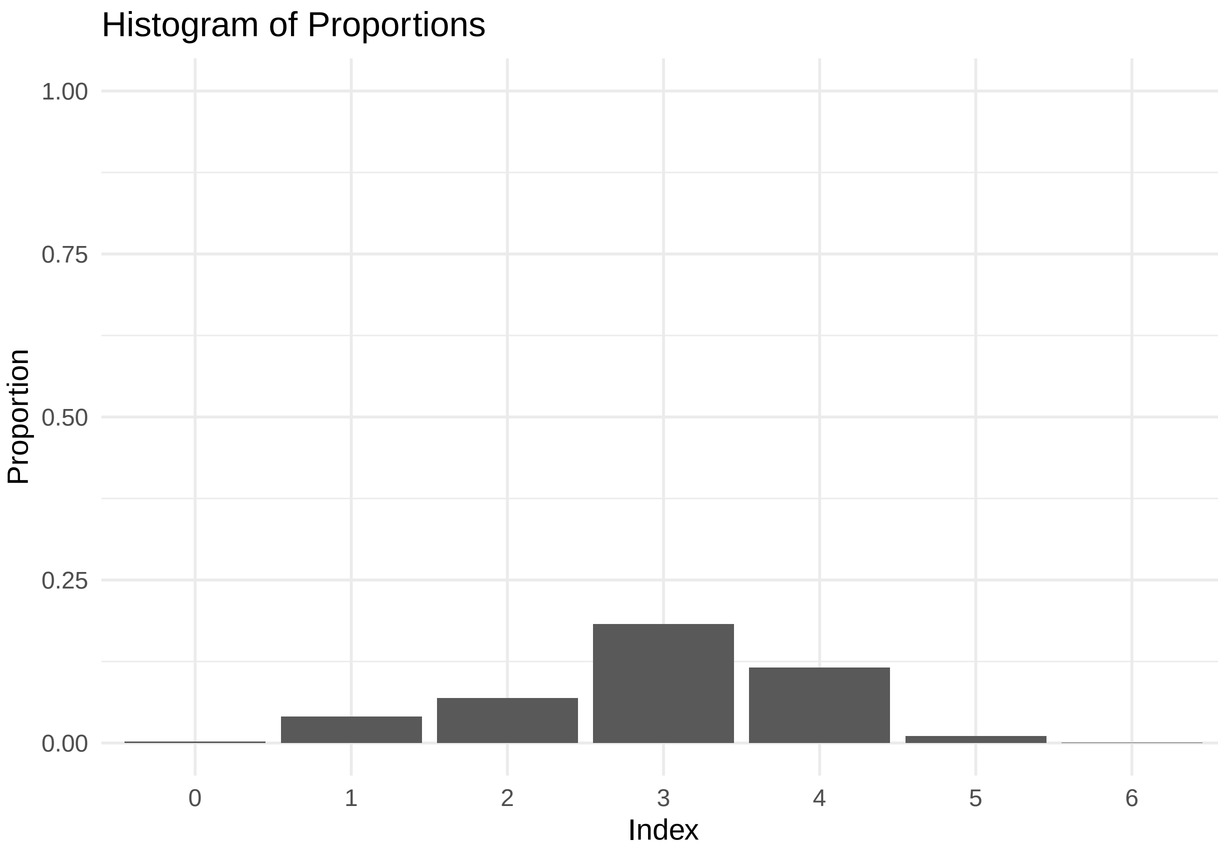


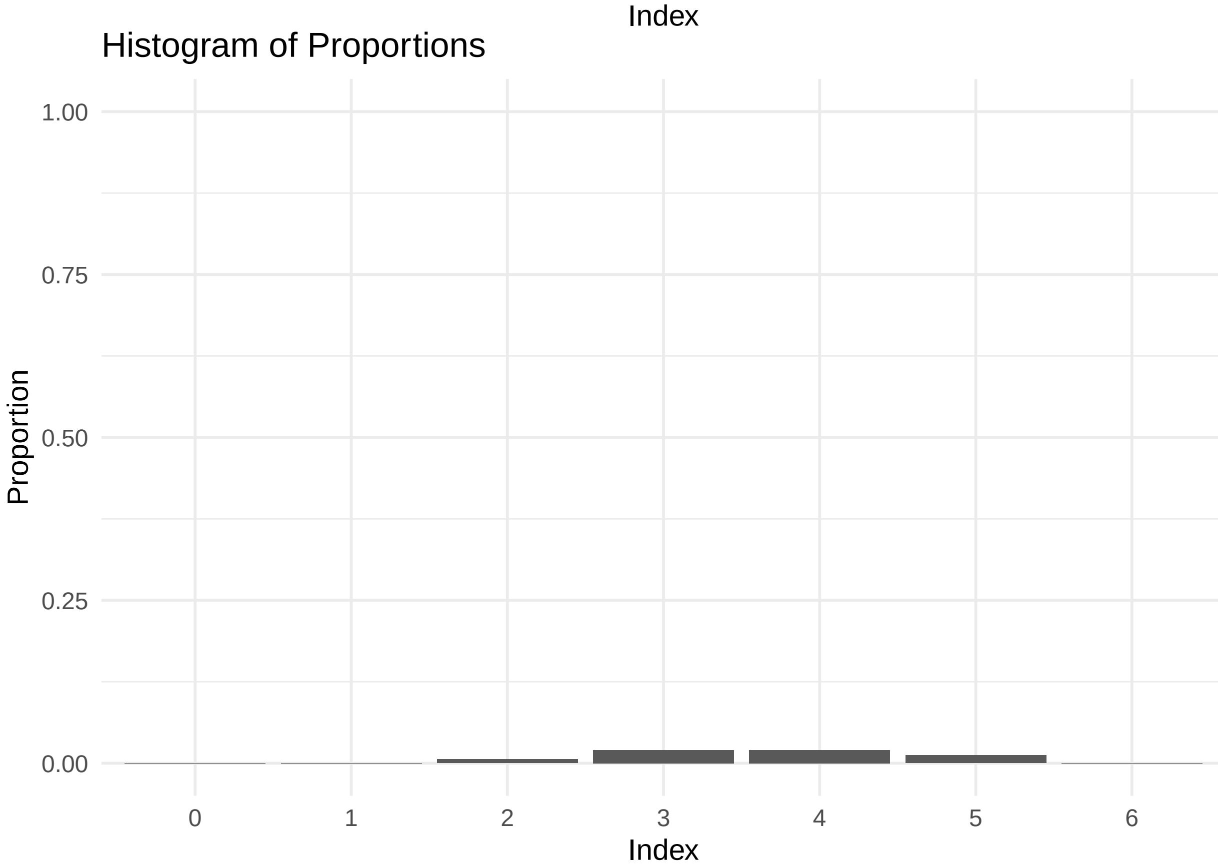

```
acc_data = as.matrix(
  t(table(new_clusters, as.integer(df[test_ids, ]$score))) / nrow(df[test_ids, ])
)
```

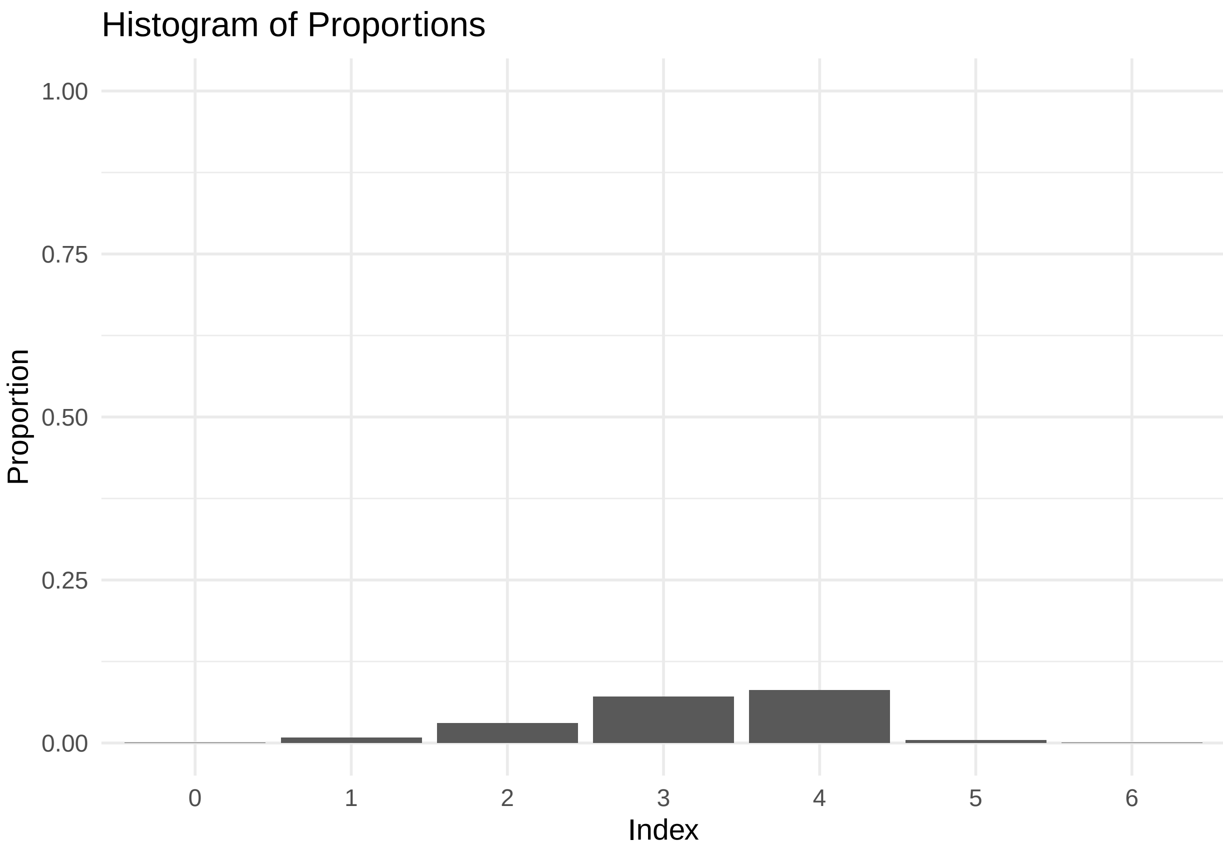
```
for (i in 1:5) {
  prop_plot = ggplot(
  data.frame(
      Index = rownames(acc_data), Proportion = acc_data[, i]
    ),
  aes(x = Index, y = Proportion)) +
  geom_bar(stat = "identity") +
  theme_minimal() +
  xlab("Index") +
  ylab("Proportion") +
  ggtitle("Histogram of Proportions") +
  ylim(c(0, 1))
  print(prop_plot)
}
```


Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0
1
2
3
4
5
6
Index


Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0
1
2
3
4
5
6
Index


Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0
1
2
3
4
5
6
Index

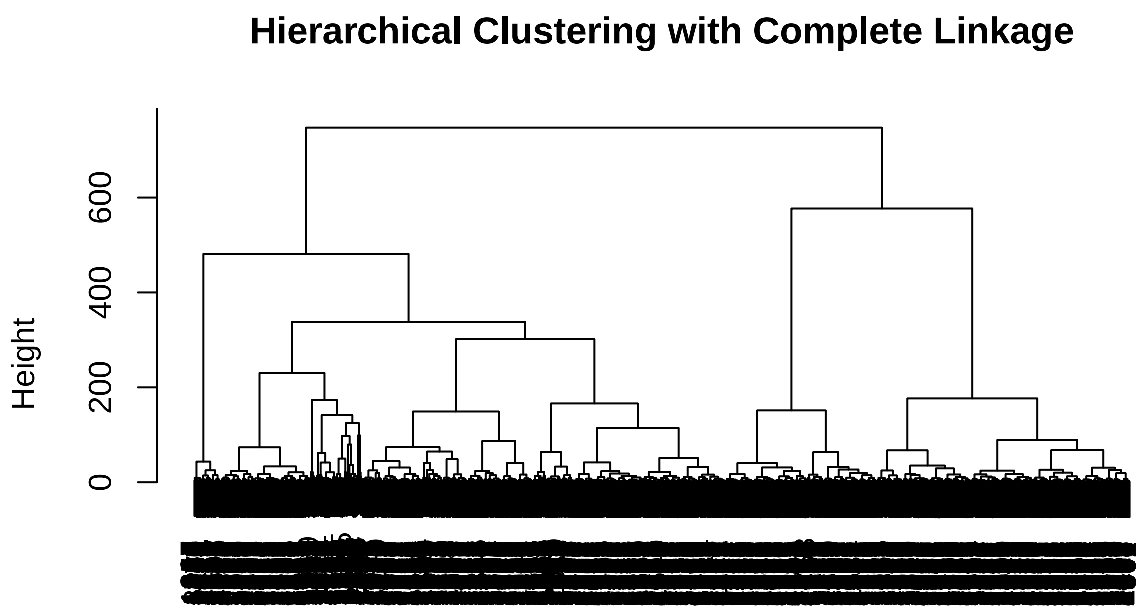

Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0
1
2
3
4
5
6
Index



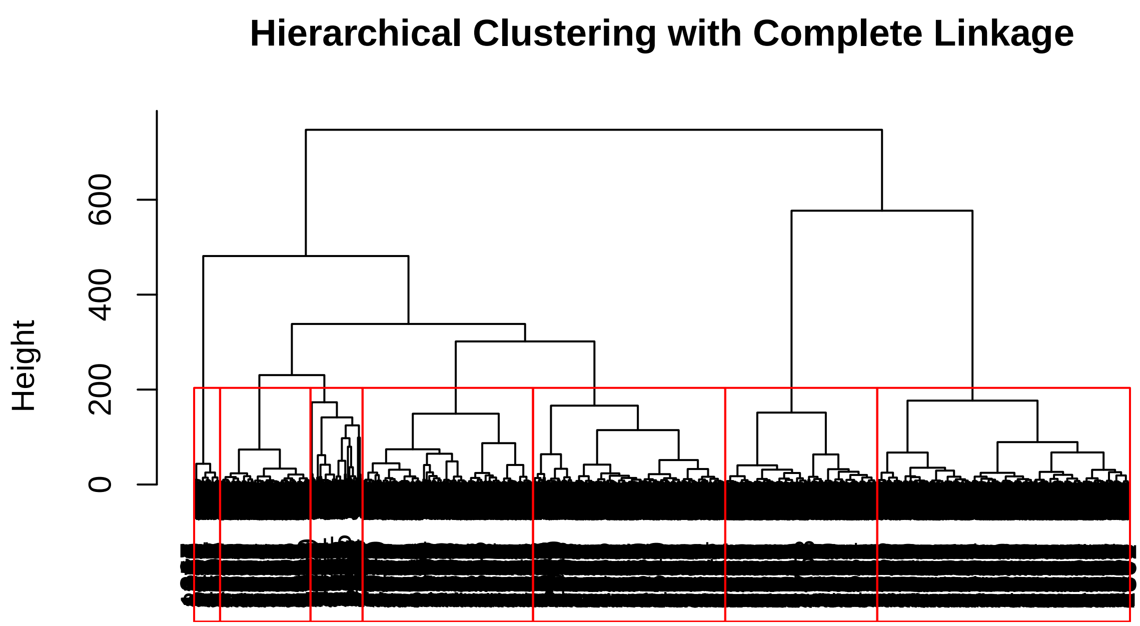
selected Ward's linkage due to cohesion and separability issues in the data. we select 7 clusters because there are 7 integer scores (after rounding half scores). The clustering structure in the data seems to support this number of clusters.

Hierarchial Clustering

```
distance_matrix <- dist(train, method = "euclidean")

hc <- hclust(distance_matrix, method = "ward.D")

plot(hc, main = "Hierarchical Clustering with Complete Linkage",
     xlab = "", sub = "", cex = 0.9)
```



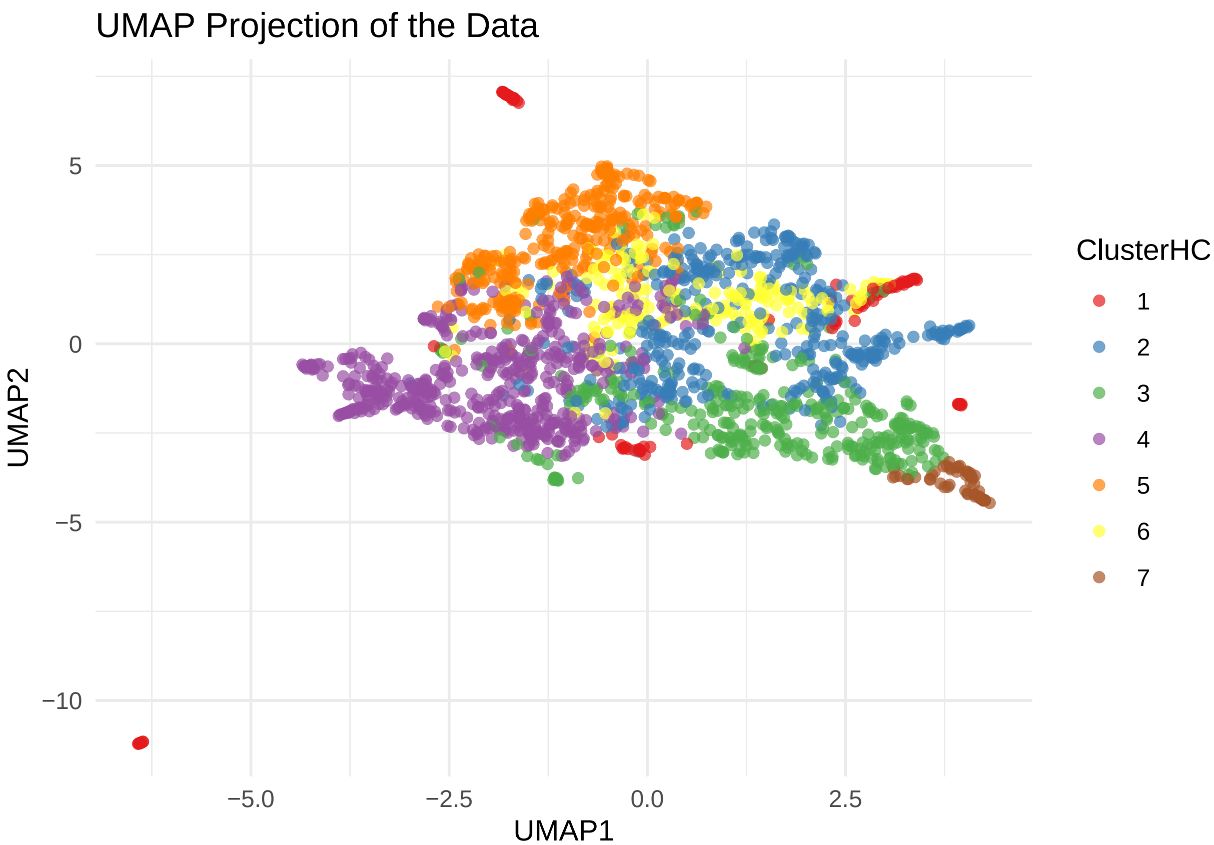

```r
k <- 7  # Number of clusters
plot(hc, main = "Hierarchical Clustering with Complete Linkage",
     xlab = "", sub = "", cex = 0.9)
rect.hclust(hc, k = k, border = "red")  # You can change the border color if you like
```



clustering structure in UMAP

```r
clusters <- as.factor(cutree(hc, k = k))
umap_data$ClusterHC <- clusters

# Choose a palette
palette <- brewer.pal(n = k, name = "Set1")  # Adjust 'name' as needed

ggplot(umap_data, aes(x = UMAP1, y = UMAP2, color = ClusterHC)) +
  geom_point(alpha = 0.7) +
  scale_color_manual(values = palette) +
```

```r
  theme_minimal() +
  ggtitle("UMAP Projection of the Data")
```



investigation of the score distribution suggests that the underlying clustering structure of the data …

spread of data between kmeans clusters and scores. characterizations are as follows:

- cluster 1 tends to capture those who score 2.5 to 4.5
- cluster 2 tends to capture those who score 3.5 to 4.5
- cluster 3 tends to capture those who score 1.5 to 4.5
- cluster 4 tends to capture those who score 2.5 to 4
- cluster 5 tends to capture those who score 3.5 to 6
- cluster 6 tends to capture those who score 2.5 to 4.5
- cluster 7 tends to capture those who score 3 to 4.5

e.g., a user who scores around 5 is likely to be in cluster 5

```r
t(table(umap_data$ClusterHC, df[-test_ids, ]$score)) / nrow(df[-test_ids, ])
```

```
##
##                  1            2            3            4            5
##   0.5 0.0000000000 0.0000000000 0.0010116338 0.0010116338 0.0000000000
##   1   0.0000000000 0.0015174507 0.0030349014 0.0065756196 0.0000000000
##   1.5 0.0020232676 0.0040465352 0.0050581689 0.0151745068 0.0010116338
##   2   0.0025290845 0.0075872534 0.0106221548 0.0161861406 0.0005058169
##   2.5 0.0080930703 0.0136570561 0.0247850278 0.0268082954 0.0020232676
##   3   0.0080930703 0.0283257461 0.0288315630 0.0520991401 0.0045523520
##   3.5 0.0151745068 0.0490642387 0.0349013657 0.0647445625 0.0151745068
##   4   0.0121396055 0.0384420840 0.0359129995 0.0531107739 0.0318664643
```

```
##   4.5 0.0040465352 0.0394537178 0.0242792109 0.0273141123 0.0485584219
##   5   0.0015174507 0.0151745068 0.0080930703 0.0055639858 0.0247850278
##   5.5 0.0010116338 0.0070814365 0.0050581689 0.0015174507 0.0242792109
##   6   0.0010116338 0.0010116338 0.0005058169 0.0000000000 0.0096105210
##
##                  6            7
##   0.5 0.0000000000 0.0000000000
##   1   0.0005058169 0.0000000000
##   1.5 0.0000000000 0.0010116338
##   2   0.0010116338 0.0000000000
##   2.5 0.0055639858 0.0010116338
##   3   0.0096105210 0.0040465352
##   3.5 0.0126454224 0.0055639858
##   4   0.0263024785 0.0055639858
##   4.5 0.0146686899 0.0055639858
##   5   0.0121396055 0.0025290845
##   5.5 0.0101163379 0.0025290845
##   6   0.0040465352 0.0000000000
```

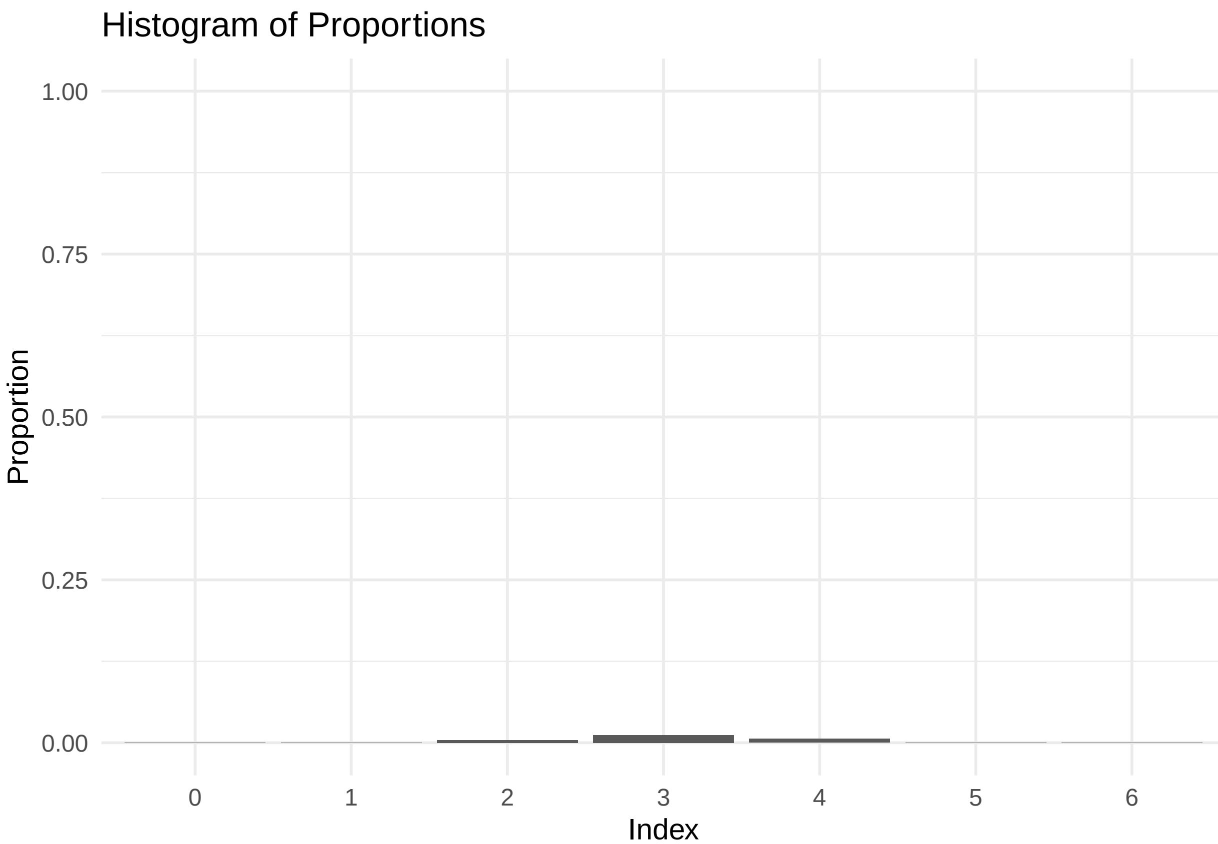
to predict on new data in test…

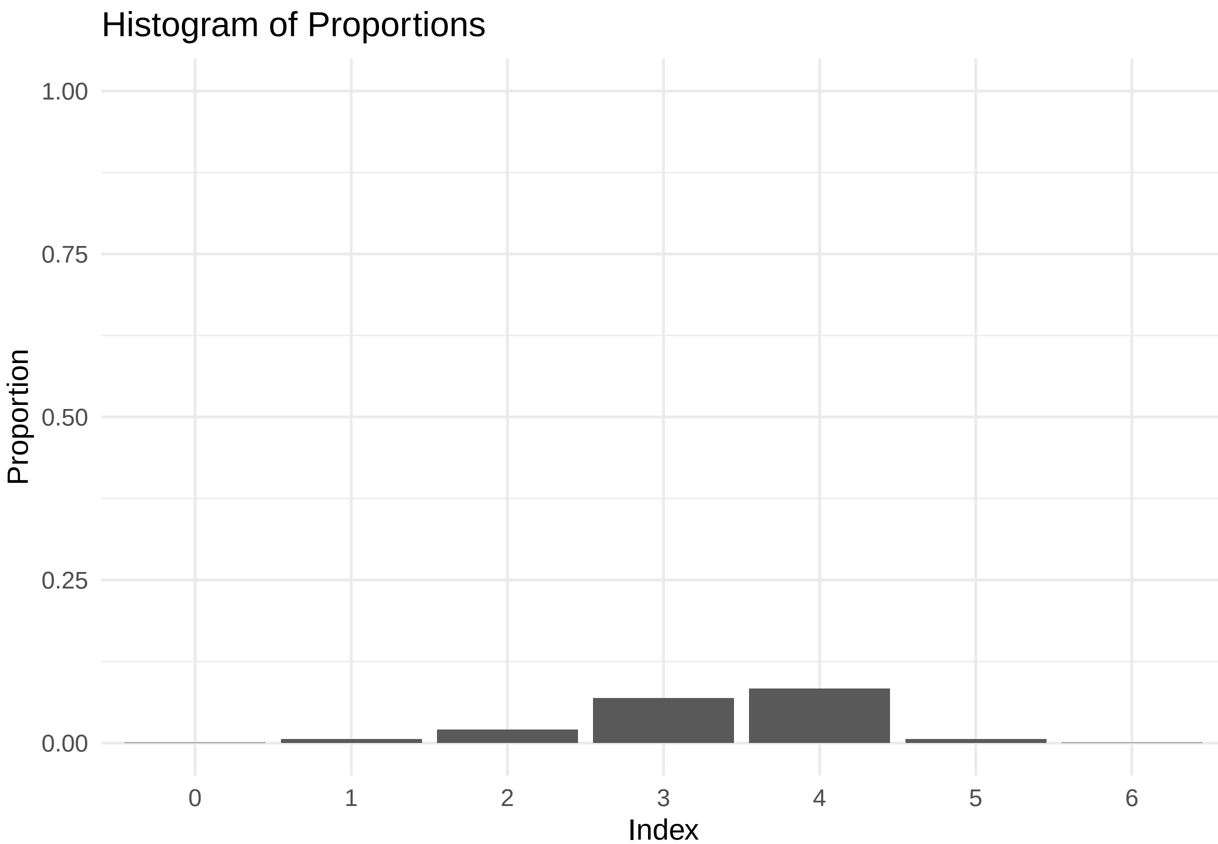
```r
# Function to calculate centroids of clusters
calculate_centroids <- function(data, clusters) {
  aggregate(data, by=list(cluster=clusters), FUN=mean)
}

# Function to predict the cluster of new data
predict_cluster <- function(new_data, train_data, clusters) {
  centroids <- calculate_centroids(train_data, clusters)
  # Remove the cluster column
  centroids <- centroids[, -1]

  # Function to find nearest centroid
  find_nearest_centroid <- function(point, centroids) {
    dists <- apply(centroids, 1, function(centroid) dist(rbind(centroid, point)))
    which.min(dists)
  }

  apply(new_data, 1, find_nearest_centroid, centroids = centroids)
}

new_clusters <- predict_cluster(test, train, clusters)
```

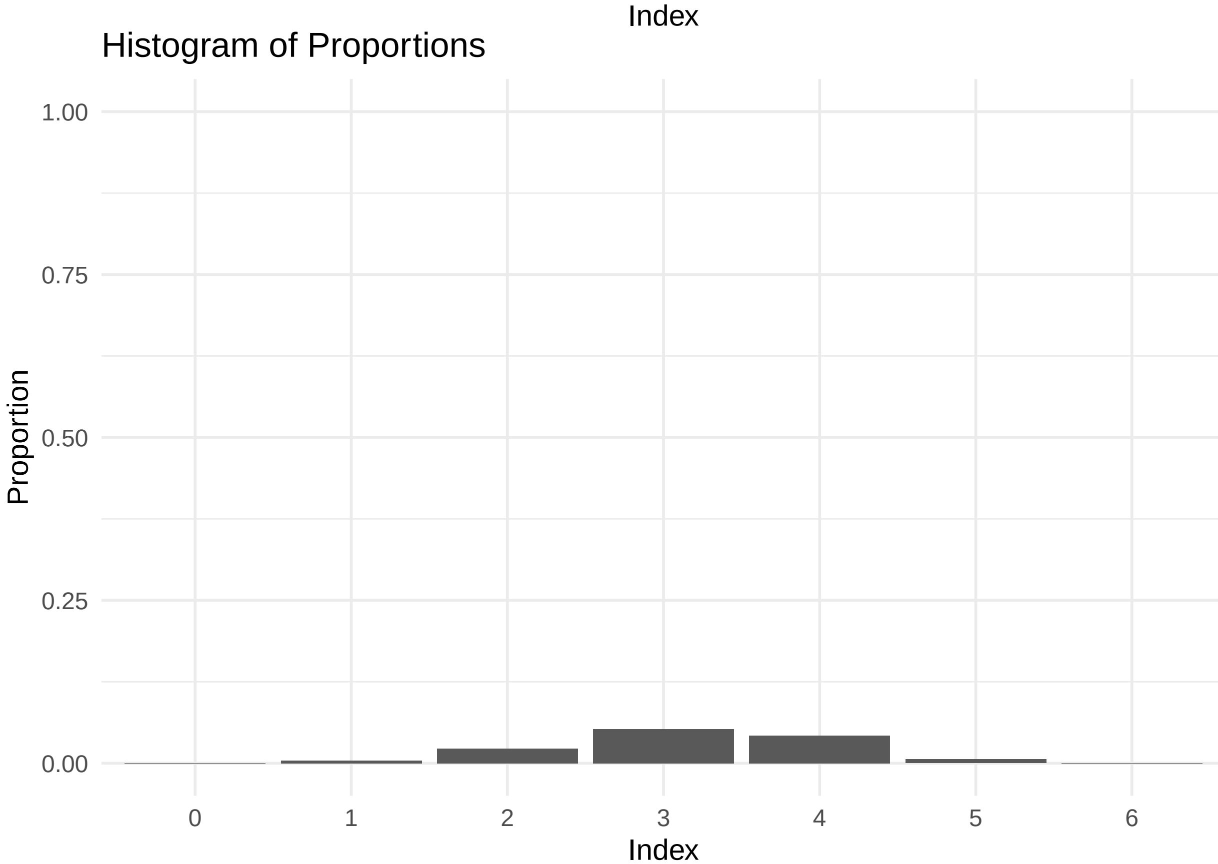
```r
acc_data = as.matrix(
  t(table(new_clusters, as.integer(df[test_ids, ]$score))) / nrow(df[test_ids, ])
)
```

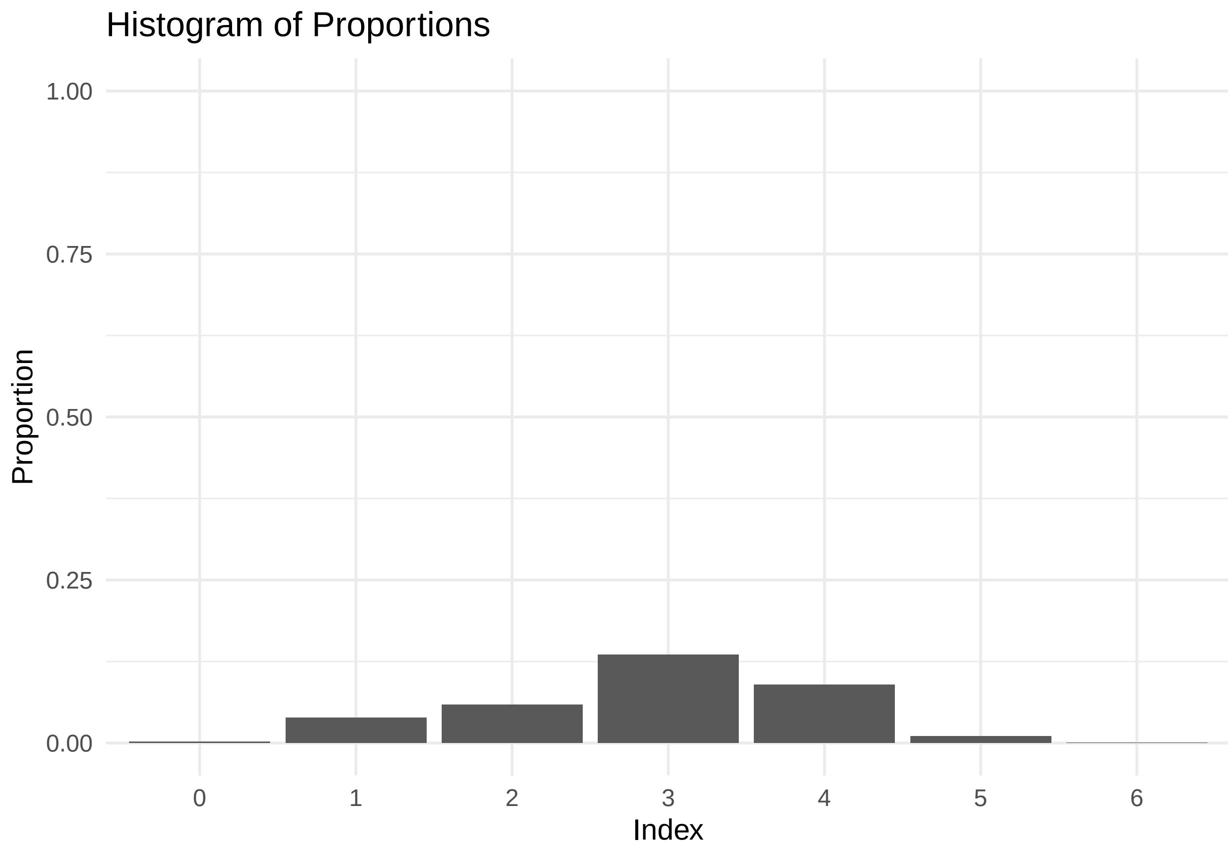
```r
for (i in 1:7) {
  prop_plot = ggplot(
  data.frame(
      Index = rownames(acc_data), Proportion = acc_data[, i]
    ),
  aes(x = Index, y = Proportion)) +
  geom_bar(stat = "identity") +
  theme_minimal() +
```

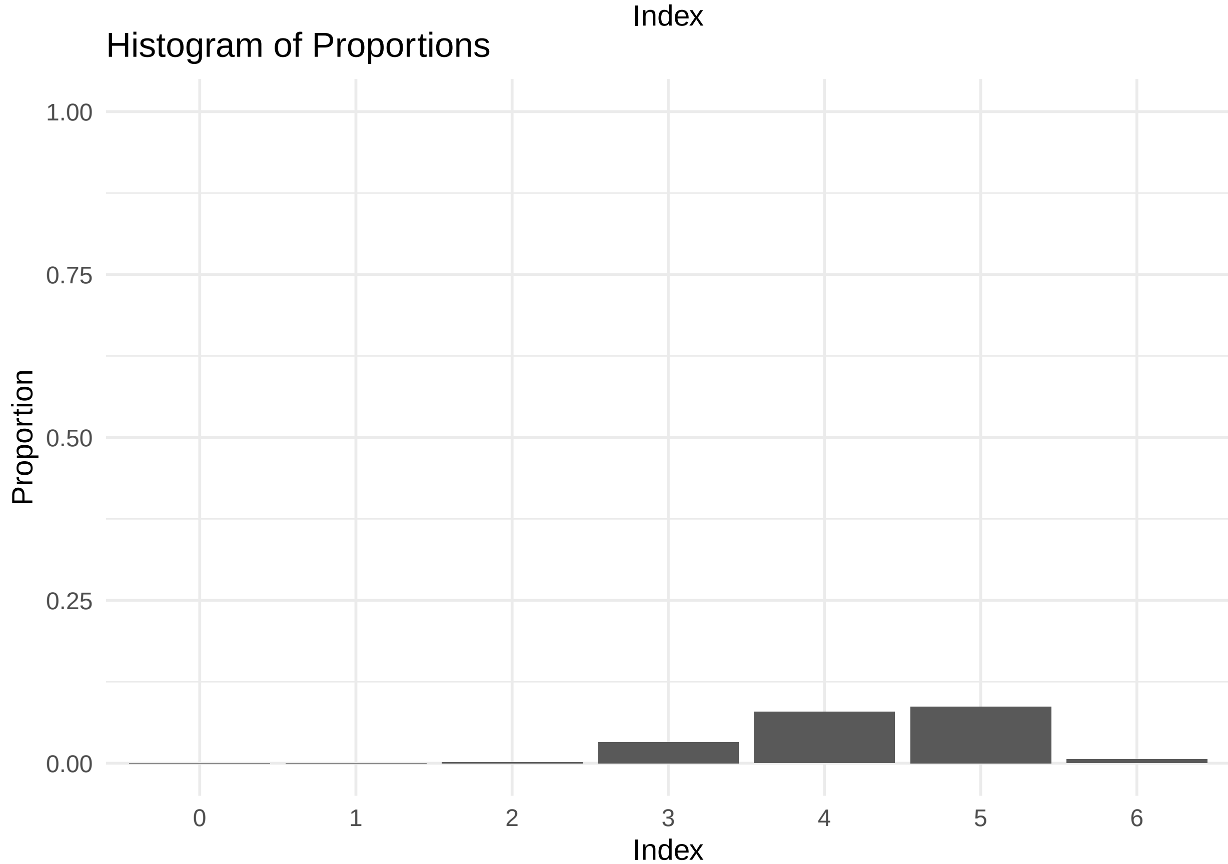
```
  xlab("Index") +
  ylab("Proportion") +
  ggtitle("Histogram of Proportions") +
  ylim(c(0, 1))
  print(prop_plot)
}
```

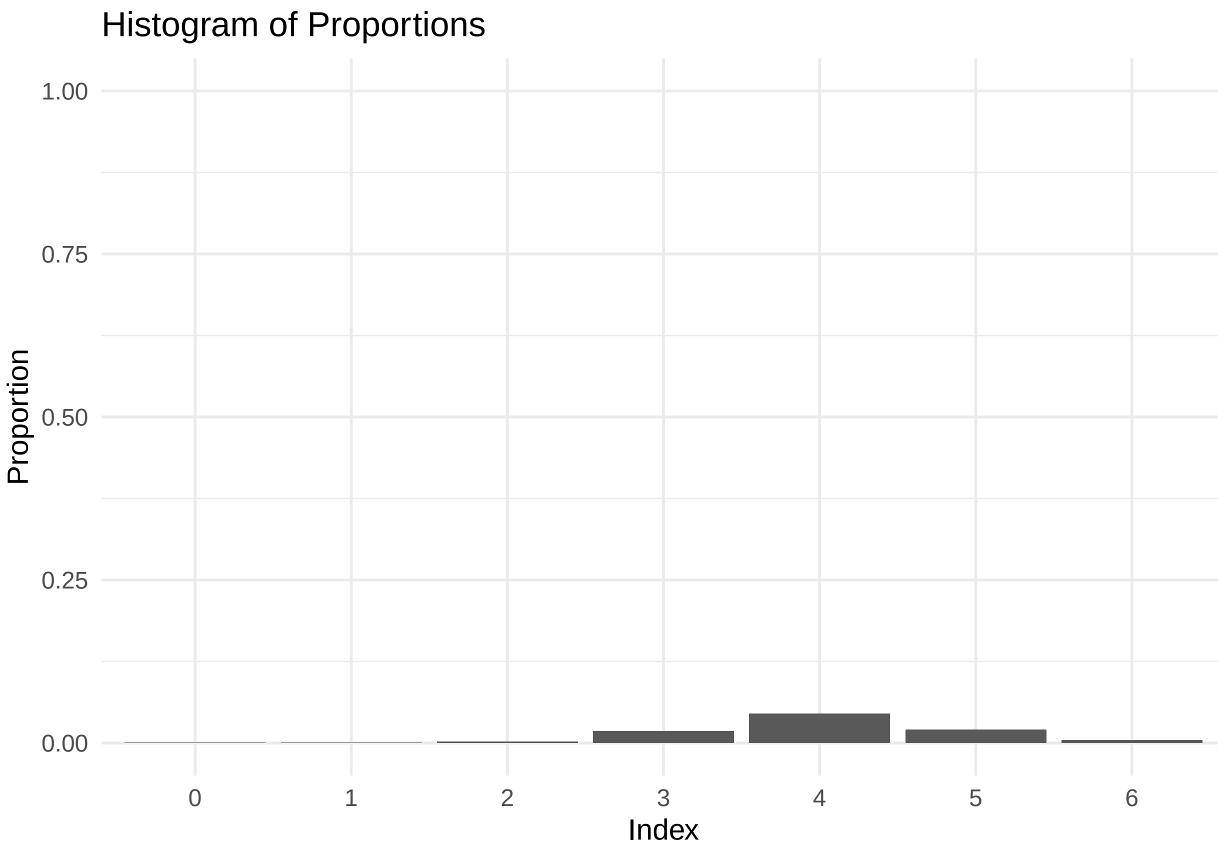


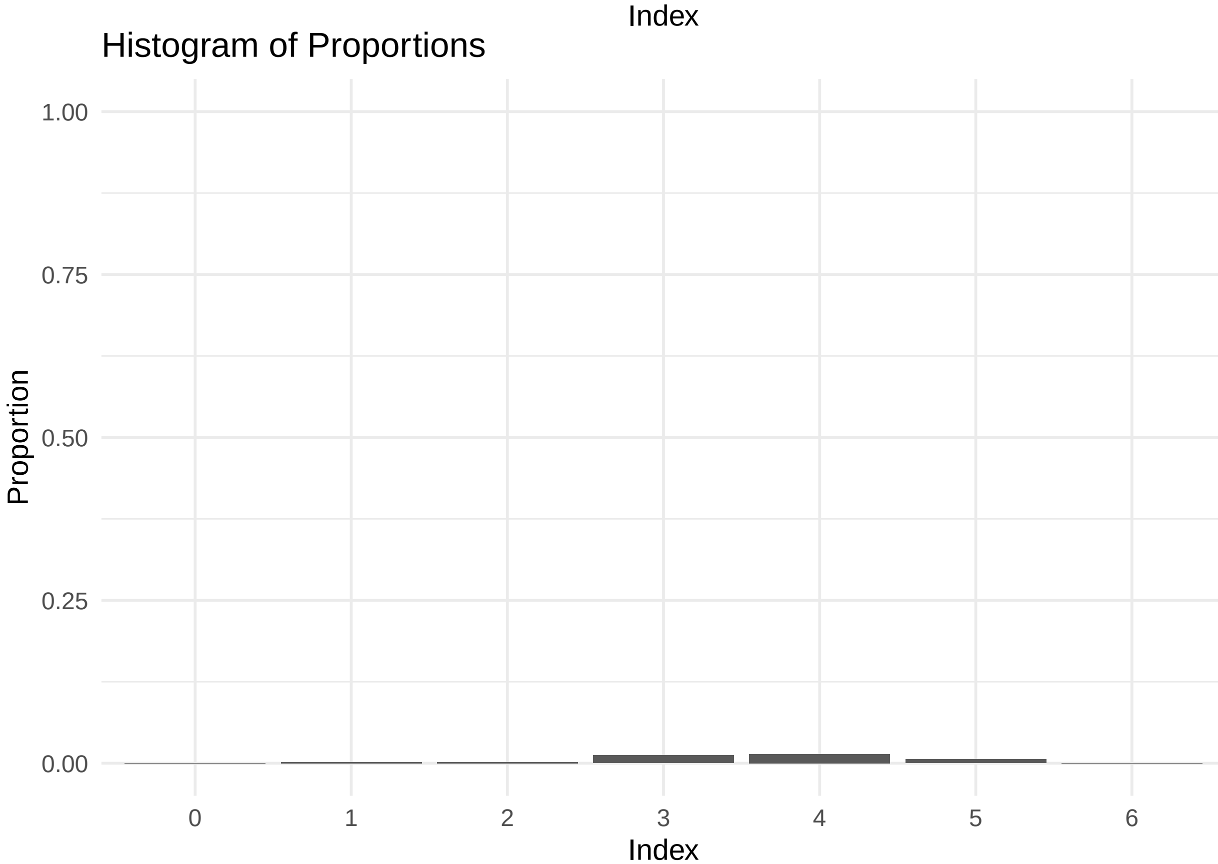

Histogram of Proportions
Proportion
1.00
0.75
0.50
0.25
0.00
0
1
2
3
4
5
6
Index


Histogram of Proportions
Proportion
1.00
0.75
0.50
0.25
0.00
0
1
2
3
4
5
6
Index


Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0
1
2
3
4
5
6
Index


Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0
1
2
3
4
5
6
Index


Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0
1
2
3
4
5
6
Index


Histogram of Proportions
1.00
0.75
0.50
0.25
0.00
Proportion
0
1
2
3
4
5
6
Index

Supervised Learning Algorithms

Proportional Odds Model (GLM)

Since the score is an ordinal variable, we can use a proportional odds model to predict the score. We'll use the `VGAM` package to fit the model and use the `step4` function to do a stepwise selection to find the best model. `VGAM` is a package that allows for fitting of a multinomial/propodds (vector based) linear model. It assumes cumulative probabilities and keeps the same β for all categories, but allows for different intercepts.

We'll use only non-collinear features and do a subset selection. We'll also convert `score` to an ordered factor to ensure that the model knows that it is an ordinal variable.

```
library(VGAM)
```

```
## Loading required package: stats4

## Loading required package: splines
```

```
train.propodds <- train
train.propodds$score <- as.ordered(train.propodds$score)
train.propodds <- subset(train.propodds, select=c(score, word_count_max, down_event_special_character, m

test.propodds <- test
test.propodds$score <- as.ordered(test.propodds$score)
test.propodds <- subset(test.propodds, select=c(score, word_count_max, down_event_special_character, mae

head(train.propodds)
```

```
##   score word_count_max down_event_special_character mae_cursor_position
## 1   3.5    -0.77681809                   -0.3196739          0.01863502
## 3     6     0.08137522                   -0.7978911          1.71740117
## 4     2    -1.06674826                   -0.4972469         -0.87562012
## 5     4    -0.80001250                    1.8685274          0.28862673
## 6     2    -0.66664462                   -1.6385269         -0.22763492
## 7   4.5    -0.85799854                   -0.2721243         -0.81875573
##   down_time_std down_event_control_keys text_change_not_q activity_input
## 1    -0.2375672              0.22353369        0.04048465    -0.30942004
## 3     0.6356203             -0.56578102       -0.50933975     0.34880782
## 4    -0.3292231             -0.21446637        0.22502129     0.22693429
## 5    -0.1407668              0.37001939        0.15320382    -0.50336416
## 6     0.5907508              0.03475356       -0.20725031    -0.04809887
## 7    -1.6535710             -0.98260633       -0.36283777     0.83417878
```

```
prop.wc <- vglm(score ~ word_count_max, data = train.propodds, family = propodds(reverse = F))

prop.upper <- vglm(score ~ ., data = train.propodds, family = propodds(reverse = F))

summary(prop.upper)
```

```
##
## Call:
## vglm(formula = score ~ ., family = propodds(reverse = F), data = train.propodds)
##
## Coefficients:
##                               Estimate Std. Error z value Pr(>|z|)
## (Intercept):1               -7.4087544  0.4930588 -15.026  < 2e-16 ***
## (Intercept):2               -5.4712050  0.2146390 -25.490  < 2e-16 ***
## (Intercept):3               -4.2559272  0.1303735 -32.644  < 2e-16 ***
```

```
## (Intercept):4                -3.4809118  0.0989883 -35.165  < 2e-16 ***
## (Intercept):5                -2.5008470  0.0748756 -33.400  < 2e-16 ***
## (Intercept):6                -1.4222969  0.0603575 -23.565  < 2e-16 ***
## (Intercept):7                -0.1678452  0.0540169  -3.107  0.00189 **
## (Intercept):8                 1.1712724  0.0611149  19.165  < 2e-16 ***
## (Intercept):9                 2.6892437  0.0874343  30.757  < 2e-16 ***
## (Intercept):10                3.8371052  0.1194225  32.130  < 2e-16 ***
## (Intercept):11                5.8637961  0.2140699  27.392  < 2e-16 ***
## word_count_max               -1.6155668  0.0639356 -25.269  < 2e-16 ***
## down_event_special_character -0.1977653  0.0484185  -4.085 4.42e-05 ***
## mae_cursor_position          -0.3351079  0.0614728  -5.451 5.00e-08 ***
## down_time_std                 0.1300749  0.0432599   3.007  0.00264 **
## down_event_control_keys       0.8554073  0.3476065   2.461  0.01386 *
## text_change_not_q            -0.0005675  0.1366863  -0.004  0.99669
## activity_input                0.8716901  0.3799373   2.294  0.02177 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Number of linear predictors:  11
##
## Names of linear predictors: logitlink(P[Y<=1]), logitlink(P[Y<=2]),
## logitlink(P[Y<=3]), logitlink(P[Y<=4]), logitlink(P[Y<=5]), logitlink(P[Y<=6]),
## logitlink(P[Y<=7]), logitlink(P[Y<=8]), logitlink(P[Y<=9]),
## logitlink(P[Y<=10]), logitlink(P[Y<=11])
##
## Residual deviance: 7036.579 on 21729 degrees of freedom
##
## Log-likelihood: -3518.29 on 21729 degrees of freedom
##
## Number of Fisher scoring iterations: 12
##
## Warning: Hauck-Donner effect detected in the following estimate(s):
## '(Intercept):1', '(Intercept):2', '(Intercept):3', '(Intercept):4', '(Intercept):10', '(Intercept):1
##
##
## Exponentiated coefficients:
##               word_count_max down_event_special_character
##                    0.1987780                    0.8205624
##          mae_cursor_position                down_time_std
##                    0.7152609                    1.1389137
##      down_event_control_keys            text_change_not_q
##                    2.3523322                    0.9994327
##               activity_input
##                    2.3909484
```

We can do a stepwise selection, starting from all variables, to find the best model. This will pick the model with the lowest AIC, which aims for better prediction error.

```
prop.step <- step4(prop.upper, scope = list(lower = prop.wc, upper = prop.upper), direction = "both")
```

```
## Start:  AIC=7072.58
## score ~ word_count_max + down_event_special_character + mae_cursor_position +
##     down_time_std + down_event_control_keys + text_change_not_q +
##     activity_input
##
```

```
##                                Df Deviance    AIC
## - text_change_not_q             1   7036.6 7070.6
## <none>                              7036.6 7072.6
## - activity_input                1   7040.9 7074.9
## - down_event_control_keys       1   7041.5 7075.5
## - down_time_std                 1   7046.1 7080.1
## - down_event_special_character  1   7053.0 7087.0
## - mae_cursor_position           1   7062.4 7096.4
##
## Step:  AIC=7070.58
## score ~ word_count_max + down_event_special_character + mae_cursor_position +
##     down_time_std + down_event_control_keys + activity_input
##
##                                Df Deviance    AIC
## <none>                              7036.6 7070.6
## + text_change_not_q             1   7036.6 7072.6
## - down_time_std                 1   7046.1 7078.1
## - down_event_special_character  1   7057.9 7089.9
## - mae_cursor_position           1   7072.3 7104.3
## - down_event_control_keys       1   7087.5 7119.5
## - activity_input                1   7090.1 7122.1
```

```r
summary(prop.step)
```

```
##
## Call:
## vglm(formula = score ~ word_count_max + down_event_special_character +
##     mae_cursor_position + down_time_std + down_event_control_keys +
##     activity_input, family = propodds(reverse = F), data = train.propodds)
##
## Coefficients:
##                              Estimate Std. Error z value Pr(>|z|)
## (Intercept):1                -7.40885    0.49093 -15.091  < 2e-16 ***
## (Intercept):2                -5.47121    0.21461 -25.494  < 2e-16 ***
## (Intercept):3                -4.25592    0.13037 -32.646  < 2e-16 ***
## (Intercept):4                -3.48091    0.09898 -35.169  < 2e-16 ***
## (Intercept):5                -2.50085    0.07487 -33.404  < 2e-16 ***
## (Intercept):6                -1.42230    0.06035 -23.566  < 2e-16 ***
## (Intercept):7                -0.16785    0.05402  -3.107  0.00189 **
## (Intercept):8                 1.17126    0.06111  19.165  < 2e-16 ***
## (Intercept):9                 2.68923    0.08743  30.759  < 2e-16 ***
## (Intercept):10                3.83710    0.11942  32.132  < 2e-16 ***
## (Intercept):11                5.86378    0.21407  27.392  < 2e-16 ***
## word_count_max               -1.61549    0.06189 -26.101  < 2e-16 ***
## down_event_special_character -0.19786    0.04283  -4.620 3.84e-06 ***
## mae_cursor_position          -0.33525    0.05459  -6.141 8.22e-10 ***
## down_time_std                 0.13007    0.04319   3.011  0.00260 **
## down_event_control_keys       0.85386    0.11380   7.503 6.24e-14 ***
## activity_input                0.87000    0.11371   7.651 1.99e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Number of linear predictors:  11
##
## Names of linear predictors: logitlink(P[Y<=1]), logitlink(P[Y<=2]),
```

```
## logitlink(P[Y<=3]), logitlink(P[Y<=4]), logitlink(P[Y<=5]), logitlink(P[Y<=6]),
## logitlink(P[Y<=7]), logitlink(P[Y<=8]), logitlink(P[Y<=9]),
## logitlink(P[Y<=10]), logitlink(P[Y<=11])
##
## Residual deviance: 7036.579 on 21730 degrees of freedom
##
## Log-likelihood: -3518.29 on 21730 degrees of freedom
##
## Number of Fisher scoring iterations: 10
##
## Warning: Hauck-Donner effect detected in the following estimate(s):
## '(Intercept):1', '(Intercept):2', '(Intercept):3', '(Intercept):4', '(Intercept):10', '(Intercept):1
##
##
## Exponentiated coefficients:
##               word_count_max down_event_special_character
##                    0.1987933                    0.8204858
##          mae_cursor_position                down_time_std
##                    0.7151620                    1.1389032
##      down_event_control_keys               activity_input
##                    2.3486901                    2.3869162
```

Using our selected model, we can predict the probabilities of falling into each category and selecting the category with the highest probability as the predicted score. We can then compare the predicted scores to the actual scores to see how well our model did.

```
head(predict(prop.step, newdata = train.propodds, type = "response"))
```

```
##            0.5           1         1.5           2        2.5          3
## 1 0.0020129636 0.011796510 0.031269678 0.047865882 0.12153415 0.23084370
## 3 0.0003173617 0.001881721 0.005176134 0.008496561 0.02533182 0.07098226
## 4 0.0048587446 0.027925705 0.069767940 0.096190939 0.19918450 0.26232947
## 5 0.0012012464 0.007079096 0.019097278 0.030203441 0.08243029 0.18372622
## 6 0.0028241651 0.016458518 0.042881802 0.063617802 0.15135297 0.25278677
## 7 0.0024170826 0.014125500 0.037121667 0.055935443 0.13738186 0.24396235
##         3.5          4        4.5           5         5.5            6
## 1 0.2925349 0.17696374 0.06518104 0.013564148 0.005580823 0.0008524905
## 3 0.1948199 0.32130131 0.25692429 0.075254932 0.034121927 0.0053917620
## 4 0.2117575 0.09094503 0.02868084 0.005691597 0.002315422 0.0003523535
## 5 0.3028962 0.23830362 0.10197308 0.022347083 0.009313586 0.0014288804
## 6 0.2681490 0.13974156 0.04786306 0.009734125 0.003982928 0.0006072798
## 7 0.2808066 0.15631445 0.05523180 0.011341926 0.004651544 0.0007097741
```

```
train.prop.probs <- predict(prop.step, newdata = train.propodds, type = "response")

labels <- sort(unique(train.propodds$score))
get_score <- function(x) { labels[which.max(x)] }
# Column with maximum probability is the predicted score (label)
train.prop.pred_score <- apply(train.prop.probs, 1, get_score)

# Compare predicted scores to actual scores
table(train.prop.pred_score, train.propodds$score)
```

```
##
## train.prop.pred_score 0.5   1 1.5   2 2.5   3 3.5   4 4.5   5 5.5   6
##                   0.5   0   0   1   0   0   2   0   0   0   0   0   0
```

```
##                       1     0   0   0   1   2   1   0   0   0   0   0   0
##                       1.5   0   0   0   2   2   0   2   0   0   0   0   0
##                       2     0   0   0   0   0   0   0   0   0   0   0   0
##                       2.5   1   3   6   2   8   2   1   0   0   0   0   0
##                       3     3  10  27  52  85  91  39   9   1   0   0   0
##                       3.5   0   8  18  16  54 129 221 150  62   6   3   1
##                       4     0   2   2   2   7  32  83 157 114  42  15   4
##                       4.5   0   0   2   0   3   9  34  71 124  72  47  12
##                       5     0   0   0   0   0   0   1   1   2   4   1   0
##                       5.5   0   0   0   1   1   2   7  12  19  11  29  11
##                       6     0   0   0   0   0   0   2   2   2   3   7   4
```

```r
mean(as.numeric(train.prop.pred_score) == as.numeric(train.propodds$score))
```

```
## [1] 0.3227112
```

```r
test.prop.probs <- predict(prop.step, newdata = test.propodds, type = "response")

# Column with maximum probability is the predicted score (label)
test.prop.pred_score <- apply(test.prop.probs, 1, get_score)

# Compare predicted scores to actual scores
table(test.prop.pred_score, test.propodds$score)
```

```
## 
## test.prop.pred_score 0.5  1 1.5  2 2.5  3 3.5  4 4.5  5 5.5  6
##                  0.5   0  0   0  0   0  0   0  0   0  0   0  0
##                  1     0  0   0  0   0  1   0  0   0  0   0  0
##                  1.5   0  0   0  0   0  0   0  0   0  0   0  0
##                  2     0  0   0  0   0  0   0  0   0  0   0  0
##                  2.5   0  2   1  0   2  1   0  0   0  0   0  0
##                  3     1  7   9 11  20 15  11  5   0  0   0  0
##                  3.5   0  3   3  4  11 44  49 35  10  3   0  0
##                  4     0  0   0  1   5  3  23 37  31 10   5  1
##                  4.5   0  0   0  0   0  1  10 17  31 19  10  3
##                  5     0  0   0  0   0  0   1  1   0  1   2  0
##                  5.5   0  0   0  0   1  3   2  4   4  7   7  1
##                  6     0  0   0  0   0  0   0  0   2  1   2  0
```

```r
mean(as.numeric(test.prop.pred_score) == as.numeric(test.propodds$score))
```

```
## [1] 0.2874494
```

Unfortunately, our model does not do very well. It predicts the correct score only around one third of the time on both the training and testing data. From the confusion matrix, we see that the model is not very good at predicting the lower scores. It predicts a lot of the mid level scores (3 - 4.5), but is not good at differentiating between them.

Instead of looking at the predicted classification accuracy, we can look at the MSE. This will give us a better idea of how far off the predictions are.

```r
mean(abs(as.numeric(train.prop.pred_score) - as.numeric(train.propodds$score)))
```

```
## [1] 1.113303
```

```r
mean(abs(as.numeric(test.prop.pred_score) - as.numeric(test.propodds$score)))
```

```
## [1] 1.184211
```

On average, the model is off by around 1.1-1.2 points on both the training and testing data. This is still a pretty large difference considering that the scores range from 1 to 6, but not horrible.