CHAPTER 5
IMPLEMENTATION AND TESTING

5.1 Implementation

In this project, I’m using Python language to process data. This chapter will explain how the program works. There are 4 main steps, which is : Preprocessing, TF-IDF, K-Means with statistical output, and analyzing output.

5.2 Preprocessing

First thing before working, the data need to preprocessed, so the data became clean and will gave the best result. Before that, the CSV data need to be imported to *Pandas DataFrame.*

```
1. import pandas as pd
2. dataset = pd.read_csv ("NewData/review_dataset/steam_reviews.csv")
3. dataset['review_length']       =       dataset.apply(lambda       row:
   len(str(row['review'])), axis=1)
4. dataset['sum_recommendation']     =     dataset['recommendation']     ==
   'Recommended'
5. dataset['sum_recommendation']                                          =
   dataset['sum_recommendation'].astype(int)
6. clean_data = dataset.dropna()
```

the code above (line 3 – line 5) serves to create a new column in the variable **dataset** which will later be useful for analysis. Code (line 6) **dataset.dropna()** functions to delete all data that has a **NaN** or **Null** value.

```
7. def remove_punctuation(text):
8.     for punctuation in string.punctuation:
9.         text = text.replace(punctuation, '')
10.      return text
11.  stop = [“a”,”about”,”above”,...]
12.  def stopwords(text)
13.    return " ".join([word for word in str(text).split() if word not in
   stop])
14.  import string
15.
16.  from emot.emo_unicode import UNICODE_EMO, EMOTICONS
17.
18.  def remove_emoticons(text):
19.      emoticon_pattern = re.compile(u'(' + u'|'.join(k for k in EMOTICONS)
   + u')')
20.      return emoticon_pattern.sub(r'', text)
21.
22.  def remove_url(text):
```

```
23.     utl_prattern = re.compile(r'https?://\S+|www\.\S+')
24.     return url_pattern.sub(r'', text)
25.
26. # lemmatization
27. from nltk.corpus import wordnet
28. from nltk.stem import WordNetLemmatizer
29.
30. lemmatizer = WordNetLemmatizer()
31. wordnet_map  =  {"N":wordnet.NOUN,  "V":wordnet.VERB,  "J":wordnet.ADJ,
   "R":wordnet.ADV}
32. #noun, verb, adjective, and adverb
33.
34. # function
35. def lemmatize_word(text):
36.     pos_tagged_text = nltk.pos_tag(text.split())
37.     return " ".join([lemmatizer.lemmatize(word, wordnet_map.get(pos[0],
   wordnet.NOUN)) for word, pos in pos_tagged_text])
38.
39. # Tokenization
40. def tokenize(text):
41.     text = re.split('\W+', text)
42.     return text
```

Can be seen in line 11 there are 3 dots after **"above"** this is because there are 1532 words that are inputted as stopwords, I use this code because the stopwords provided by **nlkt.corpus** are incomplete.

```
43. train = clean_data['review'].head(500)
44. length = clean_data['review_length'].head(500)
45.
46. train = train.apply(stopwords)
47. train = train.apply(remove_punctuation)
48. train = train.str.lower()
49. train = train.apply(remove_emoticons)
50. train = train.apply(lemmatize_word)
51. train = train.apply(stopwords)
52. train_tokenize = train.apply(tokenize)returns varchar(100)
```

Code above aims to run all the functions that have been created previously. The code on lines 43 and 44 aims to choose how much data will be processed next, this is happens because there is a limitation on the hardware that is not able to process 400 thousand data.

5.3 TF-IDF

After done with preprocess the data, the next step is Term Frequency – Inverse Document Frequency. This steps will explain how TF-IDF implemented from scratch.

```
53.  train_revamp = [x for x in train if x != '']
54.
55.  length_list = length.tolist()
56.
57.  for i in range(len(train)):
58.      if train[i] == '':
59.          print(i)
60.
61.  for x in range(len(train)):
62.      if train[x] == '':
63.          del length_list[x]
64.
65.  len(length_list)
```

This code serves to prepare the data before it is processed for next steps. Line 53 serves to delete data on **train** which has a value of **whitespace** (‘ ‘). Line 57 – 59 aims to find out at what index the train variable has a **whitespace**, this aims to be a comparison of the result on line 65. On lines 55 and lines 61 – 63 aims to change the **length** variable into a list that is stored in **length_list** variable and delete all data that has **whitespace** values.

```
66.  removed_comp = [x for x in removed if x != ['']]
67.  removed_comp = [x for x in removed_comp if x !=[]]
68.
69.  word_frequency = {}
70.  for i in range(len(removed_comp)):
71.      tokens = removed_comp[i]
72.      for word in range(len(tokens)):
73.          snap = tokens[word]
74.          try:
75.              word_frequency[snap].add(i)
76.          except:
77.              word_frequency[snap] = {i}
78.
79.  DF = {}
80.  for i in word_frequency:
81.      DF[i] = len(word_frequency[i])
82.
83.  total_vocab = [x for x in DF]
84.  print(total_vocab)
```

Lines 66 – 67 have the same purpose as line 53, the difference is that line 53 processes data that has not been tokenized and lines 66 – 67 on data that has been tokenized. The output for the process on line 69 – 77 is a unique word with the order in which document it appears. On lines 79 – 81 the code processes **word_frequency** so that is easier to read, with this the **Document Frequency** results have been obtained. Line 83 processes words in **DF** and creates a new dictionary.

```
85.  # TermFrequency
86.  idf = {}
87.  N = len(train) #total data inserted
88.  # no = 0
89.  for token in total_vocab:
90.      data_freq = DF[token]
91.      idf[token] = np.log10(len(train)/(data_freq))
92.  #counting idf = total review/DF[i]
93.  
94.  tf = {}
95.  for token in total_vocab:
96.      vector_tf = []
97.      appears = word_frequency[token]
98.      for document in range(len(removed_comp)):
99.          doc_freq = 0
100.         for word in nltk.word_tokenize (train_revamp[document]):
101.             if token == word:
102.                 doc_freq += 1
103.         word_tf = doc_freq/len(removed_comp)
104.         vector_tf.append(word_tf)
105.     tf[token] = vector_tf
106.
107. tfidf = []
108. for token in tf.keys():
109.     sentence_tfidf = []
110.     for tf_sentence in tf[token]:
111.         tfidf_score = tf_sentence * idf[token]
112.         sentence_tfidf.append(tfidf_score)
113.     tfidf.append(sentence_tfidf)
114.
115. model_tfidf = np.asarray(tfidf)
116.
117. weigth = []
118. for i in range(0, len(tfidf[0])):
119.     tmp = 0
120.     for j in range(0, len(tfidf)):
121.         tmp = tmp + tfidf[j][i]
122.     weight.append(tmp)
123.
124. weight_df = pd.DataFrame(data = weight, columns=['Weight'])
```

The final result of this process is **weight** array (line 117), then it converted into **dataframe** in line 124, this process is the preparation for the next step.

5.4 K-Means Implementation

The last step in this chapter is K-Means.

```
125. df_2 =  pd.DataFrame(data = length_list, columns = ['Text Length'])
126.
127. class_row = weight_df.join(df_2)
128.
129. x = np.round(df_row.iloc[:,[0,1]].values,2)
130.
131. m = x.shape[0]
132. n = x.shape[1]
133.
134. import matplotlib.pyplot as plt
135. plt.scatter(x[:,0],x[:,1],c='black',label='unclustered data')
136. plt.xlabel('Weight')
137. plt.ylabel('Text Length')
138. plt.legend()
139. plt.title('Plot of data points')
140. plt.show()
```

in lines 129 -132 determine the value of each variable, **x** contains a 2-dimensional matrix of **weight** and **length_list**, then **m** contains the output of how many columns are in the matrix, and **n** contains how many rows are in the matrix. After input and preparation, the next step is to analyze the shape of the data to determine how many **K** values are needed. The output for line 134 -140 is as follows.

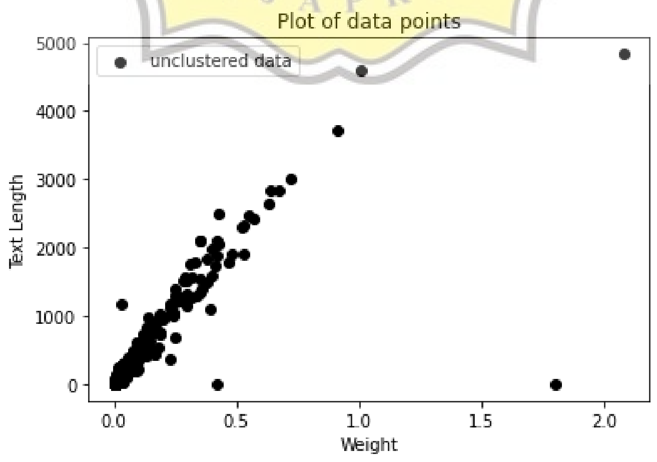


Figure 5. 1 Output from line 134 - 140

```
K = 3

Centroids = np.array([]).reshape(n,0)

import random as rd
for i in range(K):
    rand = rd.randint(0,m-1)
    print(rand)
    Centroids = np.c_[Centroids, x[rand]]

Output = {}
count = 0

while True:
    EuclidianDistance = np.array([]).reshape(m,0)
    for k in range(K):
        tempDist = np.sum((x-Centroids[:,k])**2,axis=1)
        EuclidianDistance = np.c_[EuclidianDistance, tempDist]
    C = np.argmin(EuclidianDistance, axis=1)+1
    Y = {}
    for k in range(K):
        Y[k+1] = np.array([]).reshape(2,0)

    for i in range(m):
        Y[C[i]] = np.c_[Y[C[i]], x[i]]

    for k in range(K):
        Y[k+1] = Y[k+1].T

    for k in range(K):
        Centroids[:, k] = np.mean(Y[k+1],axis=0)

    if Output == {}:
        Output = Y
        count += 1
        print('First Iteration')
    else:
        if K == 2:
            if np.array_equal(Output[1], Y[1]) == True:
                if np.array_equal(Output[2], Y[2]) == True:
                    if (Output[1] == Y[1]).all() == True:
                        if (Output[2] == Y[2]).all() == True:
                            print("Operation Done with : ", count, " Iteration")
                            break
                        else:
                            Output = Y
                            count += 1
                    else:
                        Output = Y
                        count += 1
                else:
                    Output[2] = Y[2]
                    count += 1
            else:
                Output[1] = Y[1]
```

```
196.                count += 1
```

On line 143 – 150 is a process to get the initial Centroids value, the process is done randomly and stored in an array. Variable **K** (line 141) is the number of clusters that will be used. Line 151 to completion is the initiation of the **K-Means** process. At the beginning of program of initiation, an **Output** variable as a *Dictionary* data type is created, this aims to store data of distance of each point to **Centroids**, so when using **K = n** then the **Output** dictionary will contain **n** arrays of lists. On line 154 – finished is the main code of **K-Means.** *While* (line 154) is used so that the program can count the number of iterations needed. **tempDist** (line 157) is a variable to store the value of the distance between **Centroids** and each data point. On line 173 a value checker is made for the first iteration, the reason why this is done is because an error will occur if program directly check the **Output** and **Y** variables, because these two variables have an output array, so comparing on different shapes will result in an error.

After finishing with the initial check, the next step is to see whether each array has a different value or not, on line 179 – 181 is a code to check whether each designated array has the same value, but will not return an error if it has a different shapes, return of this code is **True** and **False**. Because previously it has been confirmed to have the same shape, the on lines 181 – 190 will be checking for the equality of each value, the **.all()** argument is used because when comparing 2 arrays or more it will also return an array, therefore the **.all()** argument will function to returns only one value, if all return outputs are **True** then the argument will return **True**, but if there is one value with return **False**, then the argument will return **False** too. Furthermore, if both **if** returns **True**, then the next step is to stop the loop operation.

The code that will be displayed below is a continuation of the code above. It has same function, only the difference is on **if** operand.

```
197. elif K == 3:
198.    if np.array_equal(Output[1], Y[1]) == True:
199.     if np.array_equal(Output[2], Y[2]) == True:
200.      if np.array_equal(Output[3], Y[3]) == True:
201.       if (Output[1] == Y[1]).all() == True:
202.        if (Output[2] == Y[2]).all() == True:
203.         if (Output[3] == Y[3]).all() == True:
204.          print("Operation Done with : ", count, " Iteration")
205.          break
206.         else:
207.          Output = Y
```

```
         count += 1
       else:
        Output = Y
        count += 1
      else:
       Output = Y
       count += 1
     else:
      Output[3] = Y[3]
      count += 1
    else:
     Output[2] = Y[2]
     count += 1
   else:
    Output[1] = Y[1]
    count += 1

elif K == 4:
   if np.array_equal(Output[1], Y[1]) == True:
    if np.array_equal(Output[2], Y[2]) == True:
     if np.array_equal(Output[3], Y[3]) == True:
      if np.array_equal(Output[4], Y[4]) == True:
       if (Output[1] == Y[1]).all() == True:
        if (Output[2] == Y[2]).all() == True:
         if (Output[3] == Y[3]).all() == True:
          if (Output[4] == Y[4]).all() == True:
           print("Operation Done with : ", count, " Iteration")
           break
          else:
           Output = Y
           count += 1
         else:
          Output = Y
          count += 1
        else:
         Output = Y
         count += 1
       else:
        Output = Y
        count += 1
      else:
       Output[4] = Y[4]
       count += 1
     else:
      Output[3] = Y[3]
      count += 1
    else:
     Output[2] = Y[2]
     count += 1
   else:
    Output[1] = Y[1]
    count += 1

elif K == 5:
   if np.array_equal(Output[1], Y[1]) == True:
    if np.array_equal(Output[2], Y[2]) == True:
     if np.array_equal(Output[3], Y[3]) == True:
      if np.array_equal(Output[4], Y[4]) == True:
       if np.array_equal(Output[5], Y[5]) == True:
        if (Output[1] == Y[1]).all() == True:
         if (Output[2] == Y[2]).all() == True:
          if (Output[3] == Y[3]).all() == True:
           if (Output[4] == Y[4]).all() == True:
            if (Output[5] == Y[5]).all() == True:
             print("Operation Done with : ", count, " Iteration")
             break
            else:
             Output = Y
             count += 1
           else:
            Output = Y
            count += 1
          else:
           Output = Y
           count += 1
         else:
          Output = Y
          count += 1
        else:
         Output[5] = Y[5]
         count += 1
       else:
        Output[4] = Y[4]
        count += 1
      else:
       Output[3] = Y[3]
       count += 1
     else:
      Output[2] = Y[2]
      count += 1
    else:
     Output[1] = Y[1]
     count += 1

elif K > 5:
   print("Process canceled, the maximum number of K is 5!!!")
   Break

The next code is to show the plot graph.
color=['red','blue','green','cyan']
labels=['cluster1','cluster2','cluster3','cluster4']
for k in range(K):
    plt.scatter(Output[k+1][:,0],Output[k+1][:,1],          c=color[k],label=labels[k])
plt.scatter(Centroids[0,:],Centroids[1,:],s=500,c='yellow',label='Centroids')
plt.xlabel('weight')
plt.ylabel('HourPlay')
plt.legend()
plt.show()
```

Output from code above is a plot graph with **Centroids** and different colors for each cluster.

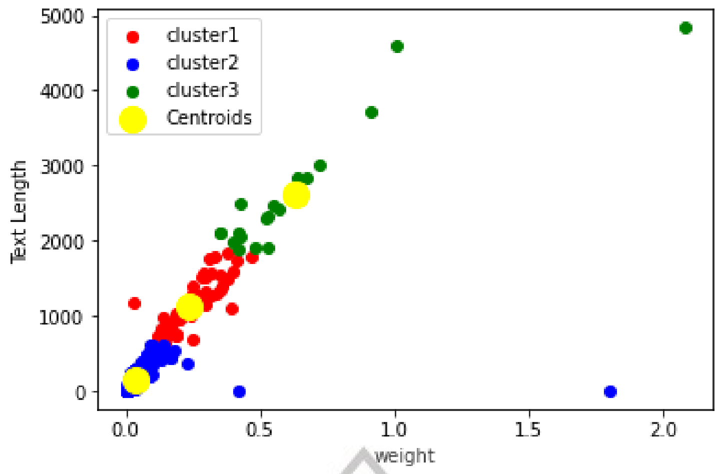


Figure 5. 2 Plot Graph for Clustered Data

To obtain accuracy value, there some code needs to be added and changed, the changed code will be shown below. First thing line 129 need to be change with code below:

```
x = np.round(df_rec.reset_index().values,2)
```

The code above will change the matrix and save the index data which will be used in accuracy test. The next line is a new code, will be added after line 129. The code will show below:

```
l = x.T
l = l[1:].T
```

The code above will create new variable that not contain index data. After this, line 157, and 162 will be replace with the following code sequentially:

```
tempDist = np.sum((l-Centroids[1:,k])**2,axis=1)
Y[k+1] = np.array([]).reshape(3,0)
```

Next code will be changed is line 311 and 310, because now the matrix has changed into three-dimensional matrix so the pointer need to be fixed

```
plt.scatter(Output[k+1][:,1],Output[k+1][:,2],                c=color[k]
  ,label=labels[k])
plt.scatter(Centroids[1,:],Centroids[2,:],s=500,c='yellow',label='Cent
  roids')
```

5.5 Analysis

In this step a result will be carried out, there are several results and analysis that will be carried out, including how **weight** data compares to **hour_play** and **helpful**, then how the data distribution for the cluster using **hour of played** and **helpful** data.

5.5.1 Data Spread on hour_play and helpful

In this sub-step, we will discuss how to spread the data if **weight** of TF-IDF is compared with **hour_play** and **helpful** data.

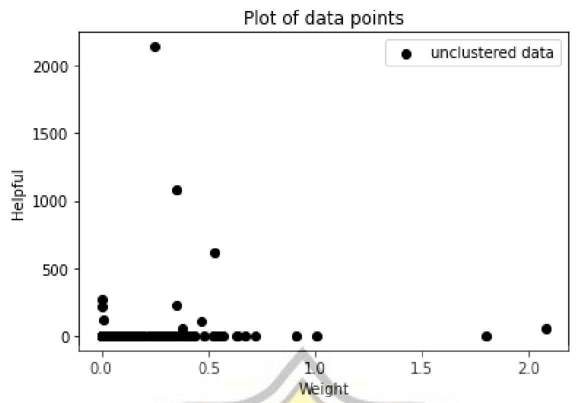


Figure 5. 3 Helpful and Weight Plot

It can be seen from the results above, the distribution of the data is not good, but this indicates that a review with a high **weight** which means it contains a lot of words and sentences does not determine that the review is good and helpful. For example, a review that has a **weight** exceeding 2.0 only get a **helpful** number that is even less than 250, while data with a **weight** of 0.25 is able to get a **helpful** number more than **2000.**

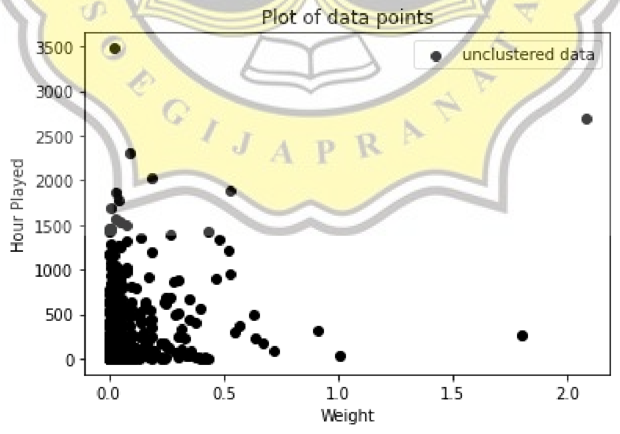


Figure 5. 4 Hour Played and Weight Plot

Meanwhile, if viewed from the graph above, it proves that from 500 reviews collected, the total hours of played from the average reviewers are still in the range 0 – 1000 hours. It can also be seen that reviewers who provide reviews with a **weight** more than **2.0** have a fairly long hour of played, which is between 2500 and 3000 hours.

5.5.2 Correlation between Hour of Played and Helpfulness

This sub-step will discuss how the correlation between hours of played and helpful.

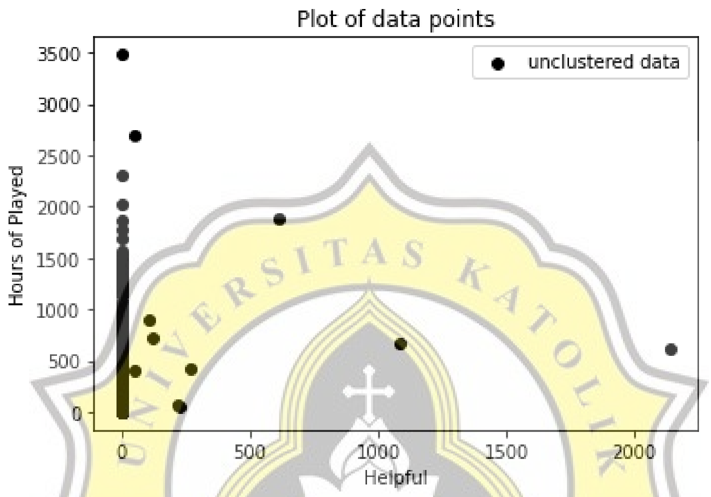
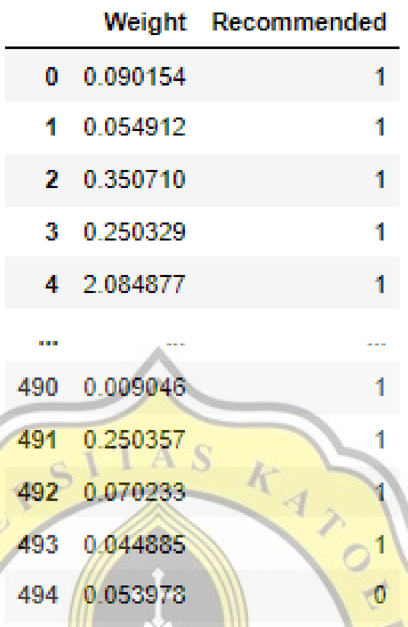


Figure 5. 5 Hours of Played and Helpful Plot

In the plot graph above, it can be seen that most of the reviewers who have high hour of played actually get **0** upvote, this will certainly raise suspicion, it because people believe that if reviewer have a higher play hour, it means they know the game more, about the mechanics and etc. which means they understand about the game better. But there are two possibilities that can make reviewers have the high number of **upvotes**: the first is **Steam Curators,** Steam Curators are individuals or organizations that make recommendations to help others discover interesting games on Steam catalog [14]. Then the second thing is *fake reviews and fake upvotes*, fake upvotes and reviews are usually done by the game developer, there is a way where developers can play the overall review by only adds a few positive reviews or ***downvotes*** only a few different negative reviews [15], of course this can affect the number of ***upvotes*** or **helpful**. However, if you look at *Figure 5.5*, reviewers who have 3500 hours of play actually give a review with a weight that is almost 0.

5.5.3 Correlation between Recommendation and Review

Like the previous step, this step will discuss how the correlation between Recommendation and Review's **weight**



	Weight	Recommended
0	0.090154	1
1	0.054912	1
2	0.350710	1
3	0.250329	1
4	2.084877	1
...	...	...
490	0.009046	1
491	0.250357	1
492	0.070233	1
493	0.044885	1
494	0.053978	0

Figure 5. 6 Weight and Recommendation DataFrame

In the figure above, it can be seen that each **weight** of the review has been paired with the **Recommended** column, in that column number **1** means **recommended**, and **0** means **not recommended**. With the data above, the distribution of data in the plot is obtained as follows

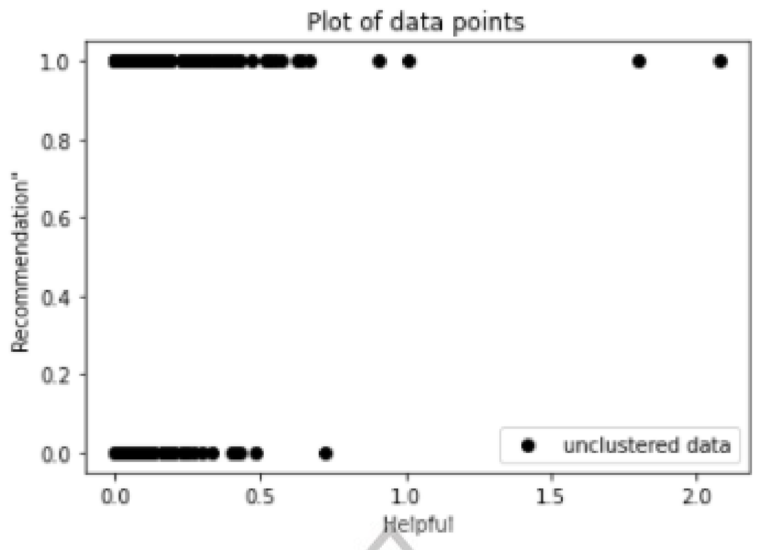


Figure 5. 7 Plot of Unclusterd Weight and Recommendation data

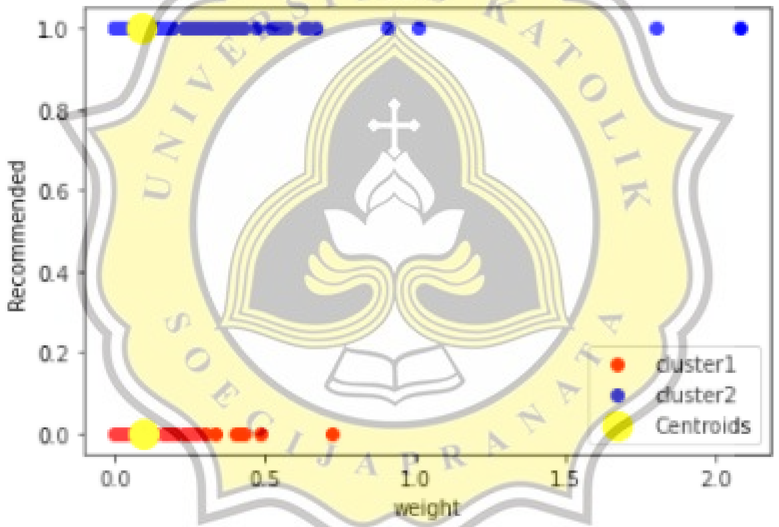


Figure 5. 8 Plot of Clustered Weight and Reccomendation data

Table 5. 1 Table of Figure 5. 8 contents

Cluster	**Recommendation**	**Description**	**Count**
1	0	Not Recommended	116
2	1	Recommended	379
Total			495

From the results above, it can be concluded that **K-Means** is able to perform clustering correctly. From 500 raw data, 495 data have been clustered accurately. Actual data includes **379 Recommended** data and **116 Not Recommended** data. From the reviews that were inputted and clustered, more than 75% of the reviews stated **Recommended**, this means more reviews stating that the game is worth buying and recommending.

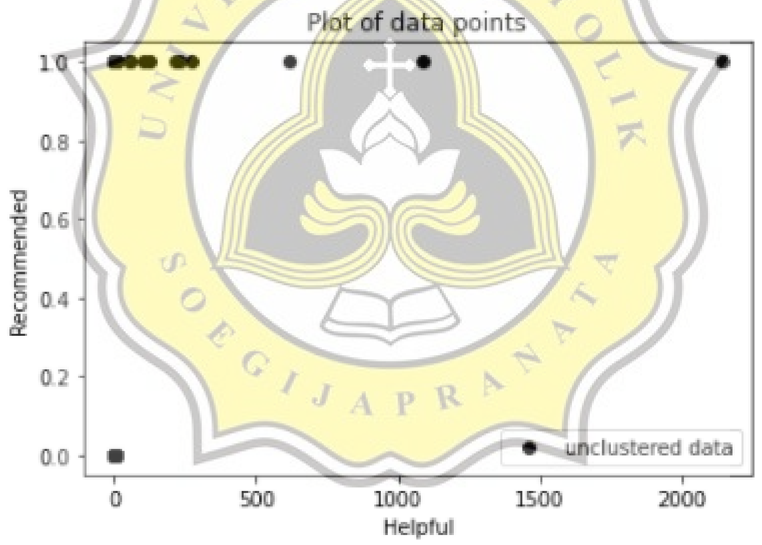


Figure 5. 9 Correlation between Recommendation and Helpful

If *Figure 5.9* compared to *Figure 5.10*, it is seen that less than 5% review state **Not Recommended** and have very low **helpful** numbers.

5.5.4 Accuracy

Calculation of accuracy in this study will use *Confusion Matrix* method. The parameter for finding the accuracy is **TP,** and **TN. TP** or **True Positive** is when actual data is **Recommended** and predicted as **Recommended** as well. **TN** or **True Negative** is when actual data is **Not Recommended** and predicted as **Not Recommended** too. The accuracy for 100 sample will be shown below.

Table 5. 2 Accuracy Table for 100 Sample Data

100 data	Predicted = Recommended	Predicted = Not Recommended
Actual = Recommended	TP = 82	FN = 3
Actual = Not Recommended	FP = 12	TN = 0
(TP + TN)/TOTAL=	(82 + 0)/97*100	84.5360824

The final accuracy for 100 sample data is 84.5%.

The accuracy of 500 sample data will be shown below:

Table 5. 3 Accuracy Table for 500 Sample Data

500 data	Predicted = Recommended	Predicted = Not Recommended
Actual = Recommended	TP = 379	FN = 0
Actual = Not Recommended	FP = 0	TN = 116
(TP + TN)/TOTAL=	(379 + 116)/495 * 100	100

The final accuracy for 500 sample data is 100%

The accuracy score above taken with this utilizing the code below:

```
df_clust1_rec = pd.DataFrame(data = Output[1][0:,1:], index = Output[1][0:,0].astype(int), columns = ['Weight','Recommendation'])

df_clust2_rec = pd.DataFrame(data = Output[2][0:,1:], index = Output[2][0:,0].astype(int), columns = ['Weight','Recommendation'])

actual = pd.DataFrame(data = x[0:,1:], index = x[0:,0].astype(int), columns = ['Weight','Recommendation'])

TP = 0
FP = 0
for i in range(len(df_rec)):
    try:
        if actual['Recommendation'][i] == 1:
            if df_clust1_rec['Weight'][i] == actual['Weight'][i]:
                TP += 1
        else:
            if df_clust1_rec['Weight'][i] == actual['Weight'][i]:
                FP += 1
    except KeyError:
        Continue

TN = 0
FN = 0
for i in range(len(df_rec)):
    try:
        if actual['Recommendation'][i] == 0:
            if df_clust2_rec['Weight'][i] == actual['Weight'][i]:
                TN += 1
        else:
            if df_clust2_rec['Weight'][i] == actual['Weight'][i]:
                FN += 1
    except KeyError:
        Continue
```

5.5.5 Evaluation

Based on the results and analysis above, the K-Means algorithm has given quite good result, the actual data and clustered data given the same result. However, for data that has values that tend to be same and less varied, it produces a bad clustered data.

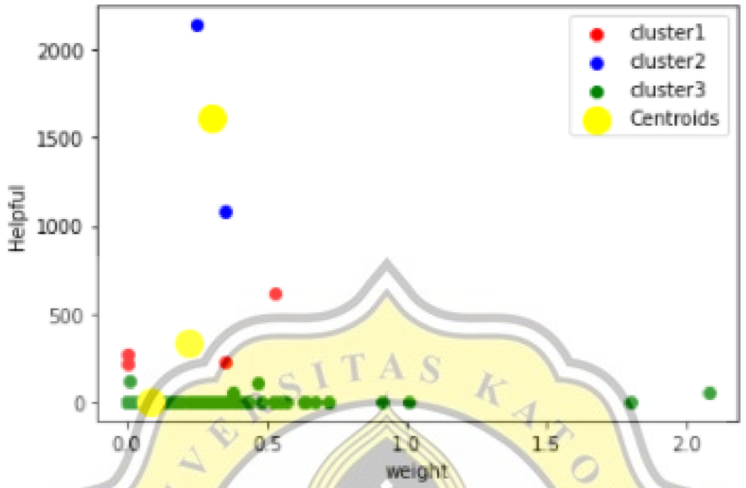


Figure 5. 10 Plot of Clustered Data between Weight and Helpful

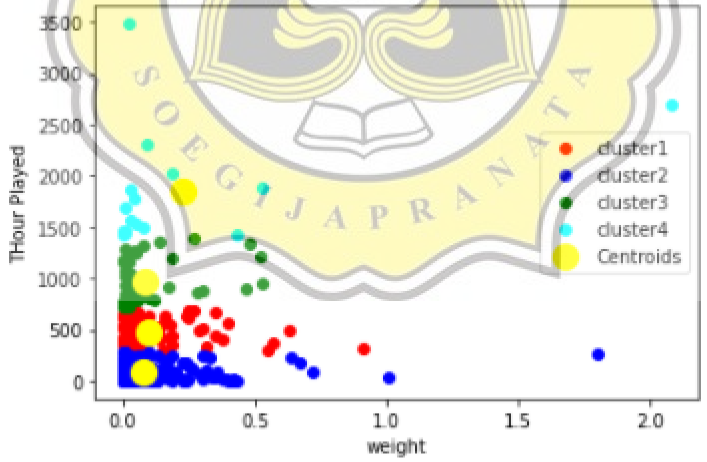


Figure 5. 11 Plot of Clustered Data between Weight and Hour of Played

As in *Figure 5.11* and *5.12*, there are still many data that deviate and too far from the **Centroids** point.