CHAPTER 5
IMPLEMENTATION AND RESULTS

5.1. Implementation

5.1.1 Random Forest

Line 1-5 import package used for Random Forest algorithm. Line 6 setup pandas display so that when reading the dataset it can reach the max column.

```
1. import pandas as pd
2. import numpy as np
3. import random
4. from random import choice
5. import math
6. pd.set_option('display.max_columns', None)
```

Line 7-8 creates a function named PrintTable with data and limit parameters. PrintTable is to record the frame of the array (make the array into a table).

```
7. def PrintTable(data, limit):
8.  return pd.DataFrame(data).head(limit)
```

Lines 9-10 to read the REVIEWS.csv dataset, from the top row to 2000 rows. With the dataset arrangement id, review, rating, author, title.

```
9. review = pd.read_csv("REVIEWS.csv", header=0, nrows=2000)
10. review = review.values
```

Line 12-13 creates a function named PreProcessingData which is taken from the data and a new array named ArrayTemp. Line 13-17 looping for data labels, if the movie rating >= 5 then it is labeled 1 which means positive, otherwise it is labeled 0 which means negative. Line 18-19 appends ArrTemp to the dataset and removes the author column, which means that the arrangement is id, review, rating, title, and label and then returned.

```
11. def PreProcessingData(data):
12.     ArrTemp = []
13. for a in data:
14.         if a[2] >= 5:
15.             label = int(1)
16.         else:
17.             label = int(0)
```

```
18.      ArrTemp.append( [ a[0], a[1], a[2], a[4], label ] )
19. return ArrTemp
20. review = PreProcessingData(review)
```

Line 21 creates a new function to calculate TF-IDF with the name Tfidf taken from the data. Line 24 creates a new array with the name WordArr (counts the number of occurrences of the word in the document) to hold the TempWord. Lines 23-24 for looping a in the data, TempWord contains a review column that is performed lower case and split by space, then displays TempWord. Line 25-34 looping b in TempWord, if the WordArr is not empty then count 0 and looping c for the long range of WordArr, if the word in WordArr is in b then count 1 and then add 1 (eg the word 'do' already exists in WordArr , means the sum is 1 + 1). Line 31-33 if the word in variable b has never existed in WordArr then it is appended to WordArr and calculated 1. Line 34-37 in variable b contains a review that only contains spaces, then it is also appended to WordArr and counted 1.

```
21. def TfIdf(data):
22.   WordArr = []
23.   for a in data:
24.         TempWord = str.lower(a[1]).split(' ')
25.   for b in TempWord:
26.         if WordArr != []:
27.               count = 0
28.                     for c in range(len(WordArr)):
29.                           if WordArr[c][0] == b:
30.                           WordArr[c]=[WordArr[c][0],
   (int(WordArr[c][1]) + int(1))]
33                            count = 1
34                                  break
31.               if count == 0:
32.                           if b != '':
33.                           WordArr.append([b, int(1)])
34.         else:
35.               if b != '':
36.                     WordArr.append([b, int(1)])
```

Line 37-41 creates a new array named TfArr to count the number of occurrences of a word in 1 row of data. Then loop a for the range in the data length. Created a new array named TempTf which will store TfArr. For words that are calculated using TempWord which is already in lower case. Line 43-55 looping b in TempWord, if in b does not contain spaces then looping if TempTf is not empty then count 0 and looping c for a long range of TempTf if the word in TempTf is in b then count 1 then add 1 (the eg word ' i' already exists in WordArr, so the sum is 1 + 1). Line 53-55 if the word in variable b has never existed in TempTf then it is appended to TempTf and calculated 1. Line 57-59 in variable b there is a review that only contains spaces, then it is also appended to TempTf and counted 1. Line 65 TempTf is appended to TfArr.

```
37. TfArr = []
38. for a in range(len(data)):
39.   TempTf = []
40.   CountTf = 0
41.   TempWord = str.lower(data[a][1]).split(' ')
42.
43.     for b in TempWord:
44.       print(b)
45.         if b != '':
46.           if TempTf != []:
47.
48.             count = 0
49.             for c in range(len(TempTf)):
50.
51.               if TempTf[c][0] == b:
52.
53.                 TempTf[c] = [TempTf[c][0], TempTf[c][1],
   (int(TempTf[c][2]) + int(1))]
54.                 count = 1
55.                 break
56.
57.              if count == 0:
58.                if b != '':
59.                  TempTf.append([data[a][0], b, int(1)])
60.
61.           else:
62.              if b != '':
63.                  TempTf.append([data[a][0], b, int(1)])
64.
65.     TfArr.append(TempTf)
```

Lines 66-75 create an array of results by looping an in length TfArr, then in looping a there is looping b in TfArr, in looping b looping c in WordArr to calculate TF-IDF. Line 80-87 TF-IDF calculation, Tf is calculated by the word that appears divided by the length of TfArr[a] (length of 1 sentence or 1 line). IDF uses math previously imported in the package, logs 2000 data divided by the occurrence of letters in the document. Then TF-IDF, the result of Tf is multiplied by the result of IDF. For line 84 CountTemp, in 1 sentence several words will add up the results of each TF-IDF then divided by 2. Line 76-78 append CountTemp on data[a] and return.

```
66. result = []
67.   for a in range(len(TfArr)):
68.     CountTemp = 0
69.       for b in TfArr[a]:
70.         for c in WordArr:
71.
72.           if b[1] == c[0]:
73.             Tf = b[2]/ len(TfArr[a])
74.             Idf = math.log(int(2000) / int(c[1]))
75.             TfIdf = Tf * Idf
76.             CountTemp = (float(CountTemp) + TfIdf) / 2
77.
78.      data[a].append(CountTemp)
79.
80.  return data
81. review = TfIdf(review)
```

Line 90-100 function for data split which divides data into train and test, with parameters train, test, data. Create a new array to accommodate the results of training data, training labels, testing data, and testing labels. Then calculate CountTraining and CountTesting. Line 101-111 loops for training data, takes as much data as CountTraining at random, and checks whether a word is present or not in the training data. The training data contains the TF-IDF rating and score. For training labels, append from the label. Line 102-110 loops for data testing, takes as much data as CountTesting at random, and checks whether a word exists or not in the testing data. The testing data contains the TF-IDF rating and score. For testing labels, append from labels. Line 123 returns TrainingData, TrainingLabel, TestingData, and TestingLabel. Line 114 training and testing is split 70 and 30.

```
82. def SplitData(train, test, data):
83.     TrainingData = []
84.     TrainingLabel = []
85.     TestingData = []
86.     TestingLabel = []
87.     CountTraining = int(len(data) * (train/100))
88.     CountTesting = int(len(data) * (test/100))
89.
90.     for a in range(CountTraining):
91.         temp = choice(data)
92.         exists = temp in TrainingData
93.         while exists == True:
94.             temp = choice(data)
95.             exists = temp in TrainingData
96.
97.         #temp[4]=label, temp[2]=rating, temp[5]=Tf-Idf
98.         TrainingLabel.append(temp[4])
99.         temp = [temp[2], temp[5]]
100.        TrainingData.append(temp)
101.
102.    for a in range(CountTesting):
103.        temp = choice(data)
104.        exists = temp in TestingData
105.        while exists == True:
106.            temp = choice(data)
107.            exists = temp in TestingData
108.        TestingLabel.append(temp[4])
109.        temp = [temp[2],temp[5]]
110.        TestingData.append(temp)
111.
112.    return TrainingData, TrainingLabel, TestingData, TestingLabel
113.
114.TrainingData, TrainingLabel, TestingData, TestingLabel =  SplitData
   (70, 30, review)
```

Line 115-117 implements the Random Forest algorithm and creates a tree list, OOB list. Line 136-142 creates a new array to hold bootstrap and OOB. Line 127-128 calculates bootstrap indices that take values randomly from the training data along with the training data. Line 130-132 computes OOB indices, retrieving values that are not in the bootstrap indices. Line 134-136 creates bootstrap data which is an append of the training data. The length of the bootstrap data is based on the bootstrap indices. The bootstrap data is labeled in a bootstrap label, based on the training label. Value 1 for positive and value 0 for negative. Line 138-140 creates OOB data which is an append of the training data. The length of the OOB data is based on the OOB indices. The OOB data is labeled in the OOB label. Line 142-143 function to build a random tree. Line 145-149 function to append tree and calculate OOB for each node then return tree_ls.

```
115.def    RandomForest(TrainingData,    TrainingLabel,    nIteration,
   maxFeature, max_depth, min_samples_split):
116.    tree_ls = list()
117.    oob_ls = list()
118.
119.    for i in range(nIteration):
120.        bootstrapData = []
121.        bootstrapLabel = []
122.        oobData = []
123.        oobLabel = []
124.        bootstrapIndices = []
125.        oobIndices = []
126.
127.        for count in range(len(TrainingData)):
128.            bootstrapIndices.append(random.randint(0,
   len(TrainingData)-1))
129.
130.        for count in range(len(TrainingData)):
131.            if count not in bootstrapIndices:
132.                oobIndices.append(count)
133.
134.        for a in range(len(bootstrapIndices)):
135.          bootstrapData.append(TrainingData[bootstrapIndices[a]])
136.         bootstrapLabel.append(TrainingLabel[bootstrapIndices[a]])
137.
138.        for a in range(len(oobIndices)):
139.            oobData.append(TrainingData[oobIndices[a]])
140.            oobLabel.append(TrainingLabel[oobIndices[a]])
141.
```

```
142.        rootNode = countSplitPoint(bootstrapData, bootstrapLabel,
   maxFeature)
143.        splitNode(rootNode, maxFeature, min_samples_split,
   max_depth, 1)
144.
145.        tree_ls.append(rootNode)
146.        oob_error = OobScore(rootNode, oobData, oobLabel)
147.        oob_ls.append(oob_error)
148.
149.    return tree_ls
```

Line 150-161 function for countSplitPoint, there are 2 features used, namely rating and TF-IDF. Feature_ls contains 1 or 0. Feature IDX is randomized if feature_ls is less than the max feature (ie 2). Lines 161-172 define a left child and a right child containing the bootstrap data and bootstrap label. Best_info_gain is made -999 taking the lowest value to get the best information gain value. This is done by choosing a value from the bootstrap randomly, each selected value is iterated and the Information Gain calculation is carried out. The value with the highest Information Gain represents a node in the tree containing IDX features, value, left child node, and right child node. The split point count uses bootstrap data, bootstrap label, and max features parameters. The split point value is obtained from the IDX+1 feature (this IDX feature is obtained randomly from the ls feature). While the value is obtained from the IDX + 1 feature. To determine the left child and right child nodes, if the value <= the value of the split point, it will form a left child node. Otherwise, it forms a right child node. Line 176-186 split information gain obtained from countInformationGain. A high value of split_info_gain will replace the value of best_info_gain. The node later forms the root node. Each node will calculate the Information Gain value.

```
150.def countSplitPoint(bootstrapData, bootstrapLabel, maxFeature):
151.    featureLs = list()
152.    numFeatures = len(bootstrapData[0])
153.
154.    while len(featureLs) < maxFeature:
155.        feature_idx = random.sample(range(numFeatures), 1)
156.        featureLs.extend(feature_idx)
157.
158.    best_info_gain = -999
159.    node = None
160.
161.    for featureIdx in featureLs:
```

```
162.         for splitPoint in bootstrapData[:featureIdx+1]:
163.             leftChild = {'bootstrapData': [], 'bootstrapLabel': []}
164.             rightChild = {'bootstrapData': [], 'bootstrapLabel':
   []}
165.
166.             for i, value in
   enumerate(bootstrapData[featureIdx+1:]):
167.                 if value <= splitPoint:
168.
   leftChild['bootstrapData'].append(bootstrapData[i])
169.
   leftChild['bootstrapLabel'].append(bootstrapLabel[i])
170.                 else:
171.
   rightChild['bootstrapData'].append(bootstrapData[i])
172.
   rightChild['bootstrapLabel'].append(bootstrapLabel[i])
173.
174.             splitInfoGain =
   countInformationGain(leftChild['bootstrapLabel'],
   rightChild['bootstrapLabel'])
175.
176.             if splitInfoGain > best_info_gain:
177.                 best_info_gain = splitInfoGain
178.                 leftChild['bootstrapData'] =
   leftChild['bootstrapData']
179.                 rightChild['bootstrapData'] =
   rightChild['bootstrapData']
180.                 node = {'informationGain': splitInfoGain,
181.                         'leftChild': leftChild,
182.                         'rightChild': rightChild,
183.                         'splitPoint': splitPoint,
184.                         'featureIdx': featureIdx}
185.
186.     return node
```

Line 187-193 function for entropy which will be used to calculate Information Gain. P is the probability. 1 for positive and 0 for negative.

```
187.def entropy(p):
188.    if p == 0:
189.        return 0
190.    elif p == 1:
191.        return 0
192.    else:
```

```
193.         return - (p * np.log2(p) + (1 - p) * np.log2(1-p))
```

Line 194-198 function to calculate Information Gain. Information Gain uses the left child and right child parameters that have been obtained in countSplitPoint. pParent is obtained by calculating the number of 1 in the parent divided by the length of the parent. Line 200-202 calculates the Information Gain of the parent, left child, and right child using entropy. Line 204-206 implementation of Information Gain and return formulas. This Information Gain calculation is carried out for each loop of the IDX feature that was previously obtained.

```
194.def countInformationGain(leftChild, rightChild):
195.    parent = leftChild + rightChild
196.    pParent = parent.count(1) / len(parent) if len(parent) > 0 else
   0
197.    pLeft = leftChild.count(1) / len(leftChild) if len(leftChild) >
   0 else 0
198.    pRight = rightChild.count(1) / len(rightChild) if
   len(rightChild) > 0 else 0
199.
200.    igParent = entropy(pParent)
201.    igLeft = entropy(pLeft)
202.    igRight = entropy(pRight)
203.
204.    informationGain = igParent - len(leftChild) / len(parent) *
   igLeft - len(rightChild) / len(parent) * igRight
205.
206.    return informationGain
```

Line 207-211 creates a branch to the left child or right child. Split Node is obtained using node parameters, max features, min sample split, max depth, and depth. The max features in this project are 2, namely the TF-IDF rating and score. Max depth = 10, Min sample split = 2. While depth is the depth of the tree that will be created. The contents of the node are deleted first because they will be filled with new values. Line 213-217 conditions if there is one empty child, it will create an empty child. The empty child is obtained from left child [bootstrap label] + right child [bootstrap label]. Line 219-222 condition if the tree depth >= max depth, then the left node splits into a terminal node from the left child. Likewise for the right split. Line 224-228 condition if the length of the left child of the bootstrap data <= min sample split, then the left split node is filled with the terminal node of the left child. If the length of the left child of bootstrap data >= min sample split, it will calculate the split point count of the new left child and recalculate countSplitPoint and countInformationGain. Line 230-234 condition if the length of the right child of bootstrap data <= min sample split, then the right split node is filled with the terminal node of the right child. If the length of the left child of bootstrap data >= min sample split, it will calculate the split point count of the new right child and recalculate countSplitPoint and countInformationGain.

```
207.def splitNode(node, maxFeature, minSampleSplit, maxDepth, depth):
208.    left_child = node['leftChild']
209.    right_child = node['rightChild']
210.    del(node['leftChild'])
211.    del(node['rightChild'])
212.
213.    if len(left_child['bootstrapLabel']) == 0 or
   len(right_child['bootstrapLabel']) == 0:
214.        empty_child = {'bootstrapLabel':
   left_child['bootstrapLabel'] + right_child['bootstrapLabel']}
215.        node['left_split'] = TerminalNode(empty_child)
216.        node['right_split'] = TerminalNode(empty_child)
217.        return
218.
219.    if depth >= maxDepth:
220.        node['left_split'] = TerminalNode(left_child)
221.        node['right_split'] = TerminalNode(right_child)
222.        return node
223.
224.    if len(left_child['bootstrapData']) <= minSampleSplit:
225.        node['left_split'] = node['right_split'] =
   TerminalNode(left_child)
```

```
226.     else:
227.         node['left_split'] =
   countSplitPoint(left_child['bootstrapData'],
   left_child['bootstrapLabel'], maxFeature)
228.         splitNode(node['left_split'], maxDepth, minSampleSplit,
   maxDepth, depth + 1)
229.
230.     if len(right_child['bootstrapData']) <= minSampleSplit:
231.         node['right_split'] = node['left_split'] =
   TerminalNode(right_child)
232.     else:
233.         node['right_split'] =
   countSplitPoint(right_child['bootstrapData'],
   right_child['bootstrapLabel'], maxFeature)
234.         splitNode(node['right_split'], maxFeature, minSampleSplit,
   maxDepth, depth + 1)
```

Line 235-238 function terminal node for each node. Pred will calculate how many numbers 1 and 0, the maximum number will be the final result of the terminal node. The terminal node contains the bootstrap label of the node.

```
235.def TerminalNode(node):
236.     bootstrapLabel = node['bootstrapLabel']
237.     pred = max(bootstrapLabel, key = bootstrapLabel.count)
238.     return pred
```

Line 239-245 function for OobScore. OobScore counts miss labels from the split point root node and OOB data (if labels are not the same, then miss label counts 1), then the number of miss labels is divided by the length of the testing data. This OobScore is performed for each node.

```
239.def OobScore(tree, testingData, testingLabel):
240.     mis_label = 0
241.     for i in range(len(testingData)):
242.         pred = PredictTree(tree, testingData[i])
243.         if pred != testingLabel[i]:
244.             mis_label += 1
245.     return mis_label / len(testingData) if len(testingData) > 0
   else 0
```

Line 246-253 function for PredictTree by comparing testing data and tree split points based on feature_idx that has been obtained by each node previously. If testing data based on feature_idx <= tree split point feature_idx, then return value to left split. If not then return value to right split.

```
246.def PredictTree(tree, testingData):
247.    feature_idx = tree['featureIdx']
248.    if testingData[feature_idx] <= tree['splitPoint'][feature_idx]:
249.        else:
250.            value = tree['left_split']
251.            return value
252.        else:
253.            return tree['right_split']
```

Line 254-260 function for PredictRf by using parameter tree_ls and testing data. In PredictRf there are 2 predictions, namely Ensemble Prediction and Final Prediction. Ensemble Prediction combines the results from the previous Predict Tree. While Final Prediction counts the number of labels 1 or 0 in Ensemble Prediction. The final prediction results are based on the majority voting of the Final Prediction.

```
254.def PredictRf(tree_ls, testingData):
255.    pred_ls = list()
256.    for i in range(len(testingData)):
257.        ensemble_preds = [PredictTree(tree, testingData[i]) for
   tree in tree_ls]
258.       final_pred = max(ensemble_preds, key = ensemble_preds.count)
259.       pred_ls.append(final_pred)
260.    return np.array(pred_ls)
```

Line 261-270 parameters used in the Random Forest algorithm. Line 325-326 displays the results of the Random Forest model. Then line 328-331 calculates accuracy by matching the prediction results from PredictRf (ensemble_preds) with the testing label (looking for how many results are the same) then summed and divided by the length of the testing label.

```
261.nIteration = 50
262.maxFeature = 2
263.maxDepth = 10
264.minSampleSplit = 2
265.
266.model = RandomForest(TrainingData, TrainingLabel, nIteration, maxFe
   ature, maxDepth, minSampleSplit)
267.
268.preds = PredictRf(model, TestingData)
269.acc = sum(preds == TestingLabel) / len(TestingLabel)
270.print("Testing accuracy: {}".format(np.round(acc,3)))
```

Line 271-281 visualizes the number of positives and negatives to determine whether the sentiment is more inclined to be positive or negative with a pie chart. First, a positive is made to 0 first, then the value of the preds result if there is a label 1 (meaning positive) will be added directly to the post. For the negative number (neg) subtract the total data by the positive number. Then it is displayed in the form of a pie chart, for the number of positive red labels and the number of pink negative labels.

```
271. import matplotlib.pyplot as plt
272. pos = int(0)
273. for value in preds:
274.     if value == 1:
275.         pos += 1
276. neg = len(preds) - pos
277. SenLabels = ['Positif', 'Negatif']
278. preds = np.array([pos, neg])
279.
280. plt.pie(preds, labels= SenLabels, colors= ['red', 'pink'])
281. plt.show()
```

5.1.2 Logistic Regression

Line 1-6 import package used for Logistic Regression algorithm. Line 6 setup pandas display so that when reading the dataset it can reach the max column.

```
1. import pandas as pd
2. import numpy as np
3. import random
4. from random import choice
5. import math
6. pd.set_option('display.max_columns', None)
```

Line 7-8 creates a function named PrintTable with data and limit parameters. PrintTable is to record the frame of the array (make the array into a table).

```
7. def PrintTable(data, limit):
8.     return pd.DataFrame(data).head(limit)
```

Line 9-10 function to split sentences using data and seperator parameters.

```
9. def SplitSentence(data, seperator):
10.        return data.split(seperator)
```

Lines 11-12 to read the REVIEWS.csv dataset, from the top row to 2000 rows. With the dataset arrangement id, review, rating, author, title. Line 14 to display the dataset.

```
11.    review = pd.read_csv("REVIEWS.csv", header=0, nrows=2000)
12.    review = review.values
```

Line 13-31 creates a new function to calculate TF-IDF with the name Tfidf taken from the data. Creates a new array with the name WordArr (counts the number of occurrences of the word in the document) to hold the TempWord. Then for looping an in the data, TempWord contains a review column that is performed lower case and split by space, then displays TempWord. Line After that looping b in TempWord, if the WordArr is not empty then count 0 and looping c for the long-range of WordArr, if the word in WordArr is in b then count 1 and then add 1 (eg the word 'do' already exists in WordArr, means the sum is 1 + 1). Checked if the word in variable b has never existed in WordArr then it is appended to WordArr and counted 1. 29-31 in variable b has a review that only contains spaces, then it is also appended to WordArr and counted 1.

Lines 32-36 create a new array named TfArr to count the number of occurrences of words in 1 row of data. Then loop a for the range in the data length. Created a new array named TempTf which will store TfArr. For words that are calculated using TempWord which is already in lower case. Line 38-47 looping b in TempWord, if in b does not contain spaces then looping if TempTf is not empty then count 0 and looping c for a long-range of TempTf if word in TempTf is in b then count 1 and then add 1 (the eg word ' i' already exists in WordArr, so the sum is 1 + 1). Line 49-51 if the word in variable b has never existed in TempTf then it is appended to TempTf and calculated 1. Line 52-54 in variable b there is a review that only contains spaces, then it is also appended to TempTf and counted 1. Line 56 TempTf is appended to TfArr.

Line 57-65 creates a result array by looping an in length TfArr, then in looping a there is looping b in TfArr, in looping b looping c in WordArr to calculate TF-IDF. Line 91-93 TF-IDF calculation, Tf is calculated by the word that appears divided by the length of TfArr[a] (the length of 1 sentence or 1 line). IDF uses math previously imported in the package, logs 2000 data divided by the occurrence of letters in the document. Then TF-IDF, the result of Tf is multiplied by the result of IDF. For line 66 CountTemp, in 1 sentence several words will add up the results of each TF-IDF then divided by 2. Line 68-70 append CountTemp into a dataset with the format id, rating, review, author, title, CountTemp, and return result.

```
13.  def TfIdf(data):
14.      WordArr = []
15.      for a in data:
16.          TempWord = str.lower(a[1]).split(' ')
17.          for b in TempWord:
18.              if WordArr != []:
19.                  count = 0
20.                  for c in range(len(WordArr)):
21.                      if WordArr[c][0] == b:
22.                          WordArr[c]        =        [WordArr[c][0],
   (int(WordArr[c][1]) + int(1))]
23.                          count = 1
24.                          break
25.
26.                  if count == 0:
27.                      if b != '':
28.                          WordArr.append([b, int(1)])
29.              else:
30.                  if b != '':
31.                      WordArr.append([b, int(1)])
```

```
32.   TfArr = []
33.       for a in range(len(data)):
34.           TempTf = []
35.           CountTf = 0
36.           TempWord = str.lower(data[a][1]).split(' ')
37.
38.           for b in TempWord:
39.               print(b)
40.               if b != '':
41.                   if TempTf != []:
42.                       count = 0
43.                       for c in range(len(TempTf)):
44.                           if TempTf[c][0] == b:
45.                               TempTf[c]       =       [TempTf[c][0],
   TempTf[c][1], (int(TempTf[c][2]) + int(1))]
46.                               count = 1
47.                               break
48.
49.                       if count == 0:
50.                           if b != '':
51.                               TempTf.append([data[a][0],          b,
   int(1)])
52.                   else:
53.                       if b != '':
54.                           TempTf.append([data[a][0], b, int(1)])
55.
56.           TfArr.append(TempTf)
57.       result = []
58.       for a in range(len(TfArr)):
59.           CountTemp = 0
60.           for b in TfArr[a]:
61.               for c in WordArr:
62.                   if b[1] == c[0]:
63.                       Tf = b[2]/ len(TfArr[a])
64.                       Idf = math.log(int(10000) / int(c[1]))
65.                       TfIdf = Tf * Idf
66.                       CountTemp = (float(CountTemp) + TfIdf) / 2
67.
68.           result.append([data[a][0],     data[a][1],     data[a][2],
   data[a][3], data[a][4], CountTemp])
69.       return result
70.   review = TfIdf(review)
```

Line 71-73 function for text processing, and create an array to group positive data and negative data respectively. Line 75-81 labels reviews and then inserts positive reviews into positive data and negative reviews into negative data. Then, calculate the training count, the test count and create a new array to hold the training and testing. Line 94-144 divides data into positive-negative training and testing, 70% for training data from positive data and negative data, and 30% for testing data from positive data and negative data. Training and testing data are selected randomly based on positive data and negative data, the author column is omitted. Then return training, testing, positive data, and negative data.

```
71.   def TextProcessing(train, test, data):
72.       posData = []
73.       negData = []
74.
75.       for value in data:
76.           if value[2] >= 5:
77.               label = 1
78.               posData.append([value[0],     value[1],      value[2],
   value[4], label, value[5]])
79.           else:
80.               label = 0
81.               negData.append([value[0],     value[1],      value[2],
   value[4], label, value[5]])
82.
83.       countTrainPos = int(train * (len(posData)/100))
84.       countTrainNeg = int(train * (len(negData)/100))
85.       countTestPos = int(test * (len(posData)/100))
86.       countTestNeg = int(test * (len(negData)/100))
87.       sizeArrayTest = countTestPos+countTestNeg
88.
89.       TrainingData = []
90.       TrainingLabel = []
91.       TestingData = []
92.       TestingLabel = [].
93.
94.       for iteration in range(countTrainPos):
95.           temp = choice(posData)
96.           checkTemp = [temp[0], temp[1], temp[2], temp[3], temp[5]]
97.           exists = checkTemp in TrainingData
98.           while exists == True:
99.               temp = choice(posData)
100.              checkTemp  =  [temp[0],  temp[1],  temp[2],  temp[3],
   temp[5]]
```

```
101.                exists = checkTemp in TrainingData
102.            TrainingLabel.append(temp[4])
103.            temp = [temp[0], temp[1], temp[2], temp[3], temp[5]]
104.            TrainingData.append(temp)
105.
106.        for iteration in range(countTrainNeg):
107.            temp = choice(negData)
108.            checkTemp = [temp[0], temp[1], temp[2], temp[3], temp[5]]
109.            exists = checkTemp in TrainingData
110.            while exists == True:
111.                temp = choice(negData)
112.                checkTemp  =  [temp[0],  temp[1],  temp[2],  temp[3],
   temp[5]]
113.                exists = checkTemp in TrainingData
114.            TrainingLabel.append(temp[4])
115.            temp = [temp[0], temp[1], temp[2], temp[3], temp[5]]
116.            TrainingData.append(temp)
117.
118.        for iteration in range(countTestPos):
119.            temp = choice(posData)
120.            checkTemp = [temp[0], temp[1], temp[2], temp[3], temp[5]]
121.            exists = checkTemp in TestingData
122.            while exists == True:
123.                temp = choice(posData)
124.                checkTemp  =  [temp[0],  temp[1],  temp[2],  temp[3],
   temp[5]]
125.                exists = checkTemp in TestingData
126.            TestingLabel.append(temp[4])
127.            temp = [temp[0], temp[1], temp[2], temp[3], temp[5]]
128.            TestingData.append(temp)
129.
130.        for iteration in range(countTestNeg):
131.            temp = choice(negData)
132.            checkTemp = [temp[0], temp[1], temp[2], temp[3], temp[5]]
133.            exists = checkTemp in TestingData
134.            while exists == True:
135.                temp = choice(negData)
136.                checkTemp  =  [temp[0],  temp[1],  temp[2],  temp[3],
   temp[5]]
137.                exists = checkTemp in TestingData
138.            TestingLabel.append(temp[4])
139.            temp = [temp[0], temp[1], temp[2], temp[3], temp[5]]
140.            TestingData.append(temp)
141.
```

```
142.     return     TrainingData,     TrainingLabel,     TestingData,
   TestingLabel, posData, negData
143.
144.  TrainingData, TrainingLabel, TestingData, TestingLabel, posData,
   negData = TextProcessing(70, 30, review)
```

Line 145-155 WordDict calculations to group positive words and negative words. For posWord, it is split based on spaces, then checks a word from the review column whether it is in the positive data or not. If there is, then the number is added by 1, otherwise, the number is counted by 1. Line 157-165 checks a word from the review column whether it is in the negative data or not. If there is, then the number is added by 1, otherwise, the number is counted by 1. Line 167-169 updates posWord and negWord in wordDict and returns wordDict.

```
145.  def WordDict(posData, negData):
146.      posWord={}
147.      for values in posData:
148.
149.          values = SplitSentence(str.lower(values[1]), ' ')
150.          for value in values:
151.              if value != '':
152.                  if (value,1) not in posWord:
153.                      posWord[(value,1)]=1
154.                  else:
155.                      posWord[(value,1)]=posWord[(value,1)]+1
156.
157.      negWord={}
158.      for values in negData:
159.          values = SplitSentence(str.lower(values[1]), ' ')
160.          for value in values:
161.              if value != '':
162.                  if (value,1) not in negWord:
163.                      negWord[(value,0)]=1
164.                  else:
165.                      negWord[(value,0)]=negWord[(value,1)]+1
166.
167.      wordDict = dict(posWord)
168.      wordDict.update(negWord)
169.      return wordDict
```

Line 170-174 function for feature extraction that converts data into vectors. The data is converted into a 3-dimensional (X) vector containing, rating, TF-IDF score, posWord dict (number of positive words in positive data for each id), and negWord dict (number of negative words in negative data for each id). Line 176-183 counts the total occurrences of positive and negative words from wordDict (for try). Line 181 if there is no word in wordDict then it will be added to 0. Then it is inserted into the matrix and returns x. The initial 190 X line is set to 0 throughout the training data. Line 192 fills X with the new value that was calculated on lines 176-183. This extraction feature is repeated for all ids based on training data.

```
170.  def FeatureExtraction(data, wordDict):
171.      word_l = SplitSentence(str.lower(data[1]), ' ')
172.      x = np.zeros((1, 4))
173.      x[0,0] = data[2]
174.      x[0,1] = data[4]
175.      for word in word_l:
176.          try:
177.              x[0,2] += wordDict[(word,1)]
178.          except:
179.              x[0,2] += 0
180.          try:
181.              x[0,3] += wordDict[(word,0.0)]
182.          except:
183.              x[0,3] += 0
184.
185.      assert(x.shape == (1, 4))
186.      return x
187.
188.  WordDicts = WordDict(posData, negData)
189.
190.  X = np.zeros((len(TrainingData), 4))
191.
192.  for i in range(len(TrainingData)):
193.      X[i, :]= FeatureExtraction(TrainingData[i], WordDicts)
```

Line 194-197 calculates the intercept, xData, yData, and weight used for the next step. Initializes the intercept to 1 throughout the training data. The xData intercept is merged vertically with feature extraction. yData here converts the data array into a NumPy array of training labels. As for the weight, it is initialized and set the value to 0 accordingly, the length corresponds to the number of contents of the matrix in 1 row and will be updated later.

```
194.  intercept = np.ones((X.shape[0], 1))
195.  xData = np.concatenate((intercept, X), axis=1)
196.  weight = np.zeros(xData.shape[1])
197.  yData = np.asarray(TrainingLabel)
```

Line 198-200 function for the sigmoid which is used for later model training functions. The final result of this sigmoid will be compared with the predicted label.

```
198.  def countSigmoid(x, weight):
199.      z = np.dot(x, weight)
200.      return 1 / (1 + np.exp(-z))
```

Line 201-202 function to calculate the loss. H here is the sigmoid result and y is yData. Line 204-209 the result of the loss is there, then it will be added to 0 and divided by 2, if the loss result is in the form of a value or number, then add the value and divide by 2. The smaller the loss value will produce a good prediction.

```
201.  def countLoss(h, y):
202.      result = (-y * np.log(h) - (1 - y) * np.log(1 - h))
203.      mean = 0
204.      for value in result:
205.          if np.isnan(value):
206.              mean = (mean + 0) / 2
207.          else:
208.              mean = (mean + value) / 2
209.      return mean
```

Line 210-214 gradient descent function to find the optimal value of the parameter. X is xData, h is the result of sigmoid and y is yData. The result of the loss that will be made is 0.

```
210.  def countGradientDescent(X, h, y):
211.      hY = np.nan_to_num(h-y, copy=True, nan=0.0)
212.      X = np.nan_to_num(X, copy=True, nan=0.0)
213.      result = np.dot(X.T, hY)/ y.shape[0]
214.      return result
```

Line 215 function for model training with learning rate, iteration, intercept, xData, weight, and yData parameters. Line 217-218 displays the sigmoid value results. Line 219 displays the loss values of sigmoid and yData. Line 220 displays the gradient descent value. Line 221 updates the weight value with a learning rate = 0.1. The new Weight will predict the label. Line 222-223 enters the new weight. This calculation is carried out in 50 iterations.

```
215.  def fit(lr , iterations, intercept, xData, weight, yData):
216.      for i in range(iterations):
217.          sigma = countSigmoid(xData, weight)
218.          sigma = np.nan_to_num(sigma, copy=True, nan=0.0)
219.          loss = countLoss(sigma,yData)
220.          dW = countGradientDescent(xData , sigma, yData)
221.          weight -= lr * dW
222.      return weight
223.  weight = fit(0.1 , 50, intercept, xData, weight, yData)
```

Line 224 function for prediction. Line 225 predicts the class label by calculating the new sigmoid first using the new weight. Line 226 looks for the result after from result before with a threshold. If result before >= threshold then True, if result before <= threshold then False. Threshold = 0.5. Line 227 label prediction is made, the value is set to 0 first as long as the result before. Line 228-233 compares the result after with the predicted label. If the result value is after 'True', then it is labeled 1, if 'False' then it is labeled 0. Line 235 calculates the accuracy of the model by checking the similarity of yData and predicting the label of the percentage of the same number divided by the length of yData.

```
224.  def predict(xData , treshold, intercept, weight):
225.      result = countSigmoid(xData, weight)
226.      result = result >= treshold
227.      predLabel = np.zeros(result.shape[0])
228.      for i in range(len(predLabel)):
229.          if result[i] == True:
230.              predLabel[i] = 1
231.          else:
232.              continue
233.      return predLabel
234.  predLabel = predict(xData, 0.5, intercept, weight)
235.  print('\nAccuracy  ->  {}\n'.format(sum(predLabel  ==  yData)  /
   yData.shape[0]))
```

Line 236-246 visualizes the number of positives and negatives to determine whether sentiment is more inclined to be positive or negative with a pie chart. First, a positive is made to 0 first, then the value of the preds result if there is a label 1 (meaning positive) will be added directly to the post. For the negative number (neg) subtract the total data by the positive number. Then it is displayed in the form of a pie chart, for the number of positive red labels and the number of pink negative labels.

```
236.  import matplotlib.pyplot as plt
237.  pos = int(0)
238.  for value in predLabel:
239.      if value == 1:
240.          pos += 1
241.  neg = len(predLabel) - pos
242.  SenLabels = ['Positif', 'Negatif']
243.  predLabel = np.array([pos, neg])
244.
245.  plt.pie(predLabel, labels= SenLabels, colors= ['red', 'pink'])
246.  plt.show()
```

5.2. Results

Based on the program code above, the results obtained from the implementation of the Random Forest and Logistic Regression algorithms on film review sentiment analysis. Random Forest and Logistics Regression can analyze movie review sentiment. In training or testing data this time from existing data, the two algorithms are more dominant to positive so that the sentiment results obtained are positive. Random forest program requires 4 hours of running time for 2000 data, 50 iterations, and the number of trees set in this project is 50 trees. Here are the results of the program by implementing the Random Forest algorithm:

Table 5.2.1 Results of Random Forest

Total number of Positive Label Prediction	Total Number of Negative Label Prediction	Total Data	Accuracy
458	142	2.000	79,97%

Table 5.2.1 above shows the results from Random Forest. From 2,000 data obtained the number of iterations and the tree is 50, the number of positive labels, the number of negative labels, and accuracy with a maximum depth of 10. It can be seen from the table that the more data, the accuracy will increase. And for analysis, later the maximum depth in this project is made 2, 5, 10, and 12 to see its effect on accuracy. The accuracy itself is obtained by matching the predicted results of the label with the test label then divided by the test data.

Meanwhile, the Logistic Regression algorithm takes 3 hours and 30 minutes to run the program for 2.000 data, 50 iterations with a learning rate = 0.1, and a threshold = 0.5. The following are the results of the program by implementing the Logistic Regression algorithm:

Table 0.1 Results of Logistic Regression

Total number of Positive Label Prediction	**Total Number of Negative Label Prediction**	**Total Data**	**Accuracy**
1.363	36	1.399	73,21%

In **Table 5.2.2** above, the results from Logistic Regression are obtained, with the number of positive labels being 1.363 obtained from the predicted label results, as well as for the number of negative labels. Then obtained a total of 1.393 data was taken from testing labels. Then the accuracy is obtained by matching the predicted results of the label with the testing label and then divided by the testing data. The prediction label is obtained. Predict Label: [1 1 1 … 1 1 1] with a total data of 1.399, so the accuracy is 73,21%.

After getting the accuracy of the Random Forest and Logistic Regression algorithms, an accuracy comparison was made. The Random Forest accuracy is 79,97% and Logistics accuracy is 73,21%, the accuracy for Random Forest is better than Logistic Regression, so the Random Forest algorithm is the best and is suitable for this case.

5.3. Analysis

In this project, the authors label the reviews based on ratings. It is labeled positive if the movie rating is more than equal to 5, and labeled negative if the movie rating is less than equal to 5. To see that sentiment is more inclined to negative or positive, the author uses the Random Forest and Logistic Regression algorithm using 2 factors, namely TF-IDF score and rating. This TF-IDF score is obtained from the calculation of movie reviews. Then this project compares the accuracy of the two algorithms. Accuracy is obtained from matching the predicted results of the training label with the testing label and then dividing by the testing data.

Random Forest and Logistics Regression can analyze movie review sentiment. In training or testing data this time from existing data, the two algorithms are more dominant to positive so that the sentiment results obtained are positive. Random forest program requires 4 hours of running time for each data with a range of 650-2.000 data, 50 iterations, and the number of trees set in this project is 50 trees. Here are the results of the trial:

Table 5.3.1 Analysis of Random Forest

Total Data	Max Depth	Total number of Positive Label Prediction	Total number of Negative Label Prediction	Sentiment	Accuracy
650	2	3	192	Negative	52,31%
	5	16	179	Negative	55,9%
	10	192	3	Positive	64,41%
	12	104	91	Positive	65,71%
800	2	225	15	Positive	54,77%
	5	45	195	Negative	58,14%
	10	162	78	Positive	66,97%
	12	213	27	Positive	67,55%
950	2	246	39	Positive	56,89%
	5	210	75	Positive	60,33%

Total Data	Max Depth	Total number of Positive Label Prediction	Total number of Negative Label Prediction	Sentiment	Accuracy
	10	193	93	Positive	68,45%
	12	239	47	Positive	69,27%
1100	2	205	125	Positive	58,31%
	5	89	241	Negative	61,09%
	10	234	96	Positive	71,68%
	12	58	272	Negative	72,50%
1250	2	91	375	Negative	59,23%
	5	322	53	Positive	62,67%
	10	311	64	Positive	72,44%
	12	318	57	Positive	73,12%
1400	2	351	69	Positive	61,35%
	5	150	270	Negative	63,05%
	10	312	108	Positive	74,19%
	12	384	36	Positive	75,70%
1550	2	412	53	Positive	62,33%
	5	434	31	Positive	64,81%
	10	439	26	Positive	75,22%
	12	375	90	Positive	76,78%
1700	2	508	2	Positive	63,79%
	5	459	51	Positive	65,12%
	10	417	93	Positive	77,93%

Total Data	Max Depth	Total number of Positive Label Prediction	Total number of Negative Label Prediction	Sentiment	Accuracy
	12	484	26	Positive	78,48%
1850	2	468	87	Positive	65,05%
	5	243	312	Negative	66,70%
	10	140	415	Negative	78,35%
	12	457	98	Positive	79,43%
2000	2	474	126	Positive	66,89%
	5	575	25	Positive	68,11%
	10	458	142	Positive	79,97%
	12	507	93	Positive	80,43%

Table 5.3.1 above shows the results from Random Forest. From the range of 650-2.000 data, the number of iterations and the tree is 50, the number of positive labels, the number of negative labels, and accuracy with different max depths are obtained. It can be seen from the table that the more the amount of data the accuracy will increase. And for the max depth in this project, 2, 5, 10, and 12 were made to see their effect on accuracy. The greater the number of max depths, will affect the accuracy obtained (the accuracy increases). Accuracy itself is obtained by matching the predicted results of the label with the test label then divided by the test data. In this training or data testing, the existing data is more dominant to positive so that the sentiment results obtained are positive.

Meanwhile, the Logistic Regression algorithm takes 3 hours and 30 minutes to run the program for 2000 data, 50 iterations with a learning rate = 0.1, and a threshold = 0.5. The following are the results of the program by implementing the Logistic Regression algorithm:

Table 5.3.2 Analysis of Logistic Regression

Number of Positive Labels	Number of Negative Labels	Accuracy	Sentiment
445	9	59,04%	Positive
548	12	60,11%	Positive
664	0	61,45%	Positive
749	20	63,53%	Positive
854	20	65,61%	Positive
995	24	67,31%	Positive
84	1080	68,34%	Negative
175	1014	69,44%	Negative
1263	31	72,09%	Positive
1363	36	74,21%	Positive

In **Table 5.3.2** above, the results of Logistic Regression are obtained, with a range of 650-2.000 data and 50 iterations, the number of positive labels, the number of negative labels, and different accuracy are obtained. It can be seen from the table that the more amount of data, the accuracy will increase. So the amount of data influences the accuracy of the results. In training or testing data, this time the existing data is more dominant to positive so that the sentiment results obtained are postive.

It can be seen in the **Table 5.3.1** and **Table 5.3.2** above, that the sentiment results obtained by the two algorithms produce more positive sentiments. This is because the training data and testing data are taken randomly. In training and testing, the randomized data this time is more dominant to positive so that the sentiment results obtained are positive.

Random Forest Graph

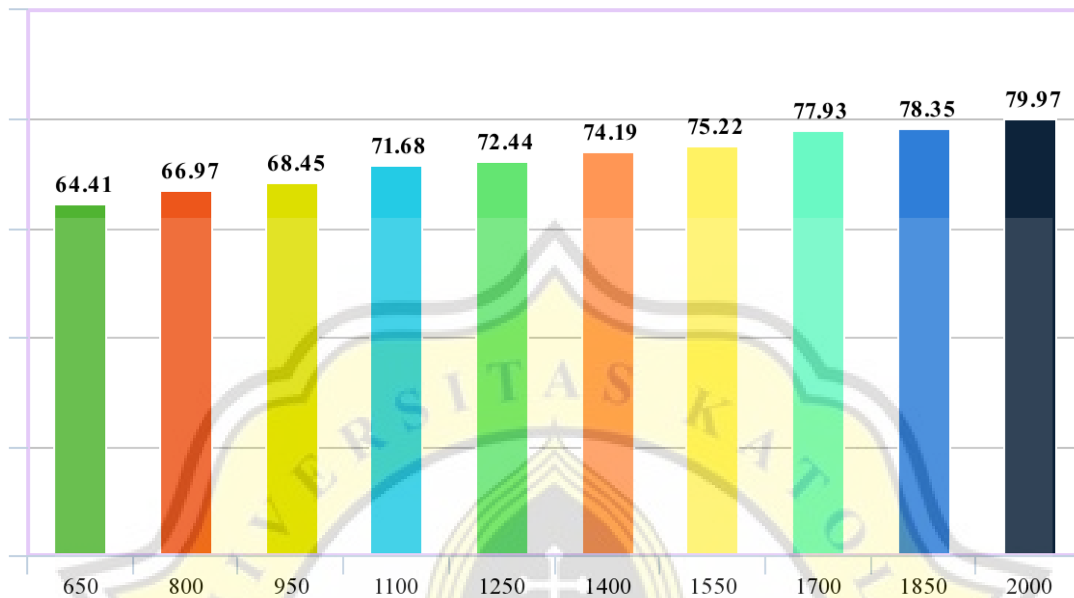

Figure 0.1 Random Forest Graph

It can be seen in **Figure 5.3**, from 10 trials with the number of datasets from the range of 650-2.000, the accuracy of Random Forest data continues to increase. This accuracy can continue to increase if the amount of data used is increasing. To determine the final accuracy result from Random Forest, the accuracy of 10 trials was averaged. The average result is 72,961%.

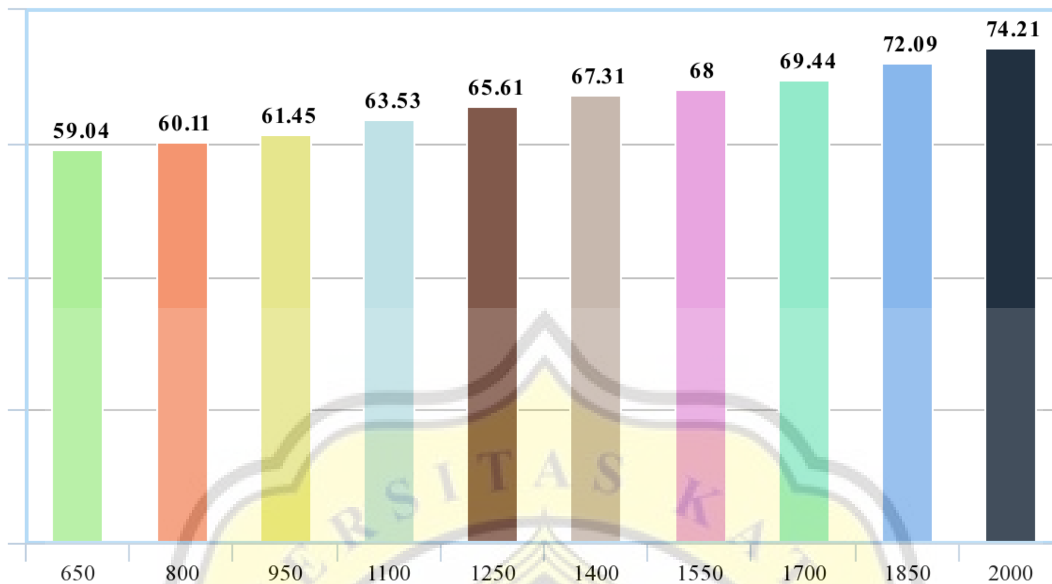

Figure 0.2 Logistic Regression Graph

It can be seen in **Figure 5.3.1** and **Figure 5.3.2**, from 10 trials with the number of datasets from the range 650-2.000, the accuracy of Logistic Regression data continues to increase. This accuracy can continue to increase if the amount of data used is increasing. To determine the final accuracy result of Logistic Regression, the accuracy of 10 trials is carried out on average. The average result is 66,310%.

So from the comparison of accuracy, Random Forest is better and suitable for this research because it has an average accuracy of 72,961% while the average accuracy of Logistic Regression is 66,310%.

The accuracy obtained by both algorithms from 10 experiments with a data range of 650-2000 data, the accuracy obtained is small (60%-70%). Because in this project the dataset and variables that I use are few. The dataset used 2,000 data and variables used 2 variables, namely rating and score TF-IDF.