CHAPTER 5
IMPLEMENTATION AND TESTING

5.1 Implementation

5.1.1 Term Frequency and Document Frequency

This project uses Python language and in this section explains how the program works. This project also describes the implementation of the program to be able to analyze, the data needed is training data and testing data. Both of these data must have been pre-processed and have TF-IDF values.

To calculate the TF-IDF value, it is necessary to calculate the term frequency value, then calculate the number of document frequencies, calculate the IDF value, and finally calculate the TF-IDF value.

The following program code calculates the Term Frequency and Document Frequency values in the title section.

```
def term_frekuensi_title(all_tot_title,data_all_prepos):
    term_info_frek =[]
    info = []

    for term in all_tot_title:
        i = 0;
        jumlah = 0
        for lists in data_all_prepos['title']:
            count = lists.count(term)
            jumlah += count
            info1 = {"doc": i, "count" : count, "term":term}
            info.append(info1)
            i +=1
        term_info = {"term": term, "info":info, "df" : jumlah}
        term_info_frek.append(term_info)

    df_term_info_title = pd.DataFrame(term_info_frek)
    # df_term_info['info']

    df_count_doc_term_title = pd.DataFrame(info)
    # df_count_doc_term

    group_term_count_title=
df_count_doc_term_title.groupby(['term','doc'])
    # group_term_count.first()
```

```
25.
26.    return
  df_term_info_title,df_count_doc_term_title,group_term_count_tit
  le
```

Line 5 explains the looping of all_tot_title. all_tot_title is a variable that contains the results of the title data that has been split down into words, while the term functions to contain the contents of the all_tot_title list.

Then in the line 8 to output data is only the title column, then it is accommodated to the lists variable. Lines 9 to 13 describe the process of adding the number of doc, count, or term. In line 9, the sum of each document is based on terms, in line 11 describes the contents of the document, the count is based on the terms. If you experience looping, the contents in info1 will increase. Then fill in the doc, count, and terms contained in a list called info.

Whereas the term_info in line 14 is the result of each term having info and df values. Info variable is the result that is accommodated from list info1, then df is the number of frequencies for each document based on term. Next in line 15, the results of term_info will be put into a list called term_info_frek.

Line 23 is the program code that contains term and doc groupings. Lines 17-23 about forming data contents starting with a list are changed to form dataframes. Next line 24, the return function returns results in lines 17, 20 and 23.

If you want to find the term frequency and document frequency results for the content section, simply change the data in line 8 to be converted to lists in data_all_prepos ['content']. The rest is the same process, but the results issued are different. Each of the results returned in line 26 is useful for calculating IDF, and calculating TF-IDF.

5.1.2 Inverse Document Frequency

The following program code calculates Inverse Document Frequency.

```
27.    def idf_title(df_term_info_title,data_all_prepos):
28.
```

```
29.      #Document Frekuensi
30.
31.      import math
32.      idf = []
33.
34.      for tf in df_term_info_title['df']:
35.           a= math.log1p(len(data_all_prepos['title'])/tf)
36.           idf.append(a)
37.
38.      # pd.DataFrame([df_term['term']],columns=['idf'])
39.      df_term = pd.DataFrame(df_term_info_title['term'])
40.      df_idf = pd.DataFrame(idf, columns=['idf'])
41.
42.     df_term_idf=pd.DataFrame({'term':df_term['term'],'idf':df_
   idf['idf']})
43.      group_idf_title =  df_term_idf.groupby(['term'])
44.
45.
46.       return group_idf_title
```

In the above program code starting from line 27-46 is the IDF value calculation code. Line 34 is the repetition done by df_term_info_title [‘df’] then it is stored in the tf variable. The contents of df_term_info_title [‘df’] are the result of term_info_frek, which was previously calculated with term frequency. Next in line 35 is the calculation of the IDF formula, the length of the data results of the title section divided by tf. After getting the value of the division is then performed log operations. Line 36 functions to add a list to the variable a. Lines 38 through 42 change to the form of dataframes, then lines 46 function to return the results to row 43. This function also functions in calculating the Inverse Document Frequency content section.

5.1.3 Term Frequency - Inverse Document Frequency

The following program code calculates Term Frequency - Inverse Document Frequency.

```
47.   def
   tf_idf_title(group_idf_title,group_term_count_title,data_all_pr
   epos):
48.
49.     tf_idf=group_idf_title['idf'].first()*group_term_count_ti
   tle.first()['count']
50.       tf_idf_frame = pd.DataFrame(tf_idf,columns=['tf_idf'])
51.
```

```
52.       #SUM each DOC
53.       tf_idf_doc = tf_idf_frame.groupby(['doc'])
  ['tf_idf'].sum()
54.       df_tf_idf_doc = pd.DataFrame(tf_idf_doc)
55.     df_tf_idf_doc=df_tf_idf_doc.reset_index().drop(columns=['
  doc'])
56.
57.
58.       hasil_tfidf_title = data_all_prepos.join(df_tf_idf_doc)
59.
60.       return hasil_tfidf_title
```

The above program code is the code to calculate the TF-IDF value for each document. Line 49 contains the calculation results of IDF multiplied by the term frequency calculation results. The idf calculation result is obtained at line 43, while group_term_count_title is the calculation result from line 23.

Furthermore, line 53 functions to classify the results of tf_idf based on doc (document), then each doc has a term and the value of tf-idf - each is given a sum() function to get the total of the tf-idf value of all terms in each doc. Lines 54 -58 change the form to dataframe, reset_index() aims to reset the index. While the drop function on line 55 is to delete the doc column in the dataframe.

Then the join function on line 58 functions to combine the results of the df_idf_doc data with the data results from data_all_prepros. Last line 60 the return function is returning the value of the result_tfidf_title.

5.1.4 Category Term Frequency - Inverse Document Frequency

The following program code for calculating categories based on TF-IDF values.

```
61.   def category_tfidf_title(hasil_tfidf_title):
62.
63.     result_category = []
64.       for x in hasil_tfidf_title['tf_idf']:
65.            if x <= 0 :
66.                result_category.append(5)
67.            elif 0 < x <= 50 :
68.                result_category.append(10)
69.            elif 50 < x <= 100 :
70.                result_category.append(15)
71.            elif 100 < x <= 150 :
72.                result_category.append(20)
```

```
73.             elif 150 < x <= 200 :
74.                 result_category.append(25)
75.             elif 200 < x <= 250 :
76.                 result_category.append(30)
77.             elif 250 < x <= 300 :
78.                 result_category.append(35)
79.             elif 300 < x <= 350 :
80.                 result_category.append(40)
81.             elif 350 < x <= 400 :
82.                 result_category.append(45)
83.             elif 400 < x <= 450 :
84.                 result_category.append(50)
85.             elif 450 < x <= 500 :
86.                 result_category.append(55)
87.             elif 500 < x <= 550 :
88.                 result_category.append(60)
89.             else:
90.                 result_category.append(65)
91.
92.      category=pd.DataFrame(result_category,columns=['kategori_tf
    idf_title'])
93.       hasil_tfidf_title= hasil_tfidf_title.join(category)
94.
95.       return hasil_tfidf_title
```

Giving a category to each document for adding data in the analysis also uses the Support Vector Machine algorithm. Because the algorithm requires 2 sample data as x for the first data horizontal section and y for the second data vertical section. Above is the program code for calculating categories based on TF-IDF values.

Row 64 has a function for repeating the results of the result_tfidf_title based on the column tf_idf, the result of the result_tfidf_title is the result of row 48. The TF-IDF results have been calculated TF-IDF in section 5.1.3. In categorizing or grouping something needs to require rules so that the results issued in accordance with the rules made. As lines 65-90 are conditions or requirements in order to produce output in accordance with these conditions.

Following are the rules in classifying each TF-IDF value.

1. Line 65 - 66 is the value of tf-idf less than or equal to 0 give value 5.

2. Lines 67 - 68 are tf-idf values more than 0 and less or equal to 50, give values 10.

3. Lines 69 - 70 are tf-idf values greater than 50 and less than or equal to 100 give a value of 15.

4. Line 71 - 72 is the value of tf-idf more than 100 and less or equal to 150, give a value of 20.

5. Line 73 - 74 is the value of tf-idf more than 150 and less or equal to 200, give the value 25.

6. Line 75 - 76 is the value of tf-idf more than 200 and less than or equal to 250, give value 30.

7. Line 77 - 78 is the value of tf-idf more than 250 and is less than or equal to 300, give a value of 35.

8. Lines 79 - 80 are tf-idf values greater than 300 and less or equal to 350 and give values 40.

9. Lines 81 - 82 are tf-idf values greater than 350 and less than or equal to 400 give values 45.

10. Lines 83 - 84 are tf-idf values more than 400 and less or equal to 450 give values 50.

11. Lines 85-86 are tf-idf values of more than 450 and are less than or equal to 500 and give values 55.

12. Lines 87 - 88 are tf-idf values of more than 500 and less or equal to 550 give a value of 60.

13. Lines 89 - 90 are tf-idf values of more than 550 and give values 60.

Line 92 changes the form of a list to a dataframe based on the results made at lines 65-90. Next, line 93 combines the results of the results_tfidf_title with the results of the category. Row 95 to return the value from row 93.

5.1.5 Implementation using the Random Forest algorithm

5.1.5.1 Code Helper

Before entering the calculation stage of the Random Forest algorithm, a calculation is required for the decision tree algorithm. Because the Random Forest algorithm is an application of the decision tree algorithm. Code helper is the code that will be used in the decision tree method.

Here is the code to check whether the data is pure, there are 1 labels or 2 labels.

```
96.   #Check Data Pure
97.   def check_data_pure(data):
98.       label = data[:,0]
99.
100.  unique_label = np.unique(label)
101.
102.  if len(unique_label) == 1:
103.     return True
104.  else:
105.     return False
```

The code above is a data checking function. Row 98 displays all rows based on column 0, column 0 contains labels. Next 100 lines to display unique words, so the output has different values.

Then lines 102 - 105 describe the condition where the length of the results from line 100 is 1, then returns the value True and if false, returns the False value.

Furthermore, the following program code functions to classify Hoax or Real.

```
106. def klasifikasi(data):
107.    label = data[:,0]
108.
109.  unique_label,jml_label=np.unique(label,return_counts=True)
110.
111.  index = np.argmax(jml_label)
112.
113.  klasifikasi_label = unique_label[index]
114.
115.  return klasifikasi_label
```

At line 107 the same as row 98 is showing all rows based on column 0, column 0 contains labels. Column 109 uses unique from numpy to display different values. While return_counts is used to calculate the total array of data, then the two results are contained in the unique_label and jml_label variables. Furthermore, row 111 looks for the largest integer value from the results of jml_label obtained from row 109. Row 113 is a way to convert the index value obtained from row 111 into the unique_label variable. Then line 115 is the return value from the results of classification_label obtained in line 113.

The following program code has a potential split function.

```
116. def get_potensial_split(data):
117.    potential_splits = {}
118.  x, n_columns = data.shape
119.  for column_index in range(1,n_columns):
120.      potential_splits[column_index] = []
121.      values = data[:, column_index]
122.      unique_values = np.unique(values)
123.
124.      for index in range(len(unique_values)):
125.          if index != 0:
126.              current_value = unique_values[index]
127.              previous_value = unique_values[index - 1]
128.                potential_split     =     (current_value     +
   previous_value) / 2
129.
130.              potential_splits[column_index].append(potential_s
   plit)
131.    return potential_splits
```

Split potential function is a function to separate between the first value and the value afterwards. Row 118 contains data.shape, meaning it provides the total value of rows and columns. So that the total row will go into the x variable while the total column will go to the n_columns variable.

Line 119 is a loop with the n_columns limit of 3, so it prints index column 1 and 2. Index column 0 does not print because 1, n_columns means to pass the initial index in that range. After that, line 124 is a loop with a length limit of unique_values, which is obtained from row 122 containing unique data from all rows in the column named column_index. In that for there is an if statement at line 125 where the index cannot or is equal to 0. If True then enter line 126-130. Line 126 initializes the value in each index, and line 127 initializes the value in the previous index. To find the potential split value by adding up the value in the index and the values in the previous index, then the results are divided in two. After getting the results it is then placed in the dictionary with the potential_splits variable at line 129. On line 131 is returning the value of the potential_splits result, the contents of which are obtained from line 130.

The following code split program data.

```
132. def split_data(data,split_column,split_value):
133.   split_data = data[:,split_column]
134.
135.   data_min =  data[split_data<=split_value]
136.   data_max =  data[split_data>split_value]
137.
138.   return data_min,data_max
```

In the program code above to find the minimum data and maximum data based on split value. Like line 135 to find the minimum data, the split data must be less than the split value. The results of the split data obtained from line 133, the line prints all the rows in the column named split_column. Meanwhile, to find the maximum data in row 136 the data must be greater than the split value. split_column and split_value are obtained when running the best split function first. Then in line 138 returns the minimum data value and maximum data.

The following program code function hitung_entropy to calculate entropy.

```
139. def hitung_entropy(data):
140.   label = data[:,0]
141.     unique_label,count_label=np.unique(label,return_counts=Tr
  ue)
142.
143.   probabilitas =  count_label / float(sum(count_label))
144.
145.
146.   entropy = sum(probabilitas * -np.log2(probabilitas))
147.
148.   return entropy
```

Furthermore, the entropy calculation function, this function is to find the value of information that states the value of uncertainty. With the formula lined up 139, before that calculate the probability first on line 143. Line 143 total Hoax and

Real labels in the form of an array divided by the total addition of the total Hoax labels and the total Real labels. Then the probability result is used as an entropy calculation in line 146. To calculate the entropy the sum of probabilities times the results of the probability log.

The following program code function hitung_overall_entropy to calculate overall entropy.

```
149.  # Overall Entropy
150.
151. def hitung_overall_entropy(data_min,data_max):
152.       jml_data = len(data_min)+len(data_max)
153.
154.     jml_data_min = len(data_min)/float(jml_data)
155.     jml_data_max = len(data_max)/float(jml_data)
156.
157.       overall_entropy=jml_data_min*  hitung_entropy(data_min)
   + jml_data_max * hitung_entropy(data_max)
158.
159.
160.     return overall_entropy
```

The overall entropy program code function, overall entropy, calculates entropy by using minimal data multiplied by the minimum amount of data. After getting the results, the results are summed by the product of the maximum amount of data and entropy results using the maximum data as in line 157. jml_data_min is obtained from the length of the minimum data divided by the total minimum and maximum data length. While the maximum number of data_ml is obtained from the maximum data length divided by the minimum and maximum total data length. Then end the return line 160 to return the value of overall_entropy.

The following program code function best_split.

```
161.  # BEST SPLIT
162. def best_split(data, potential_splits):
163.     high_entropy = 999 # High Entropy
164.     for col_index in potential_splits:
165.        for value in potential_splits[col_index]:
166.             data_min,data_max=split_data(data,split_column=co
   l_index, split_value=value)
167.             overall_entropy=hitung_overall_entropy(data_min,
   data_max)
168.             if overall_entropy < high_entropy :
169.               overall_entropy = overall_entropy
170.               split_column = col_index
```

```
171.                split_value = value
172.
173.
174.     return split_column, split_value
```

The function above is to find the cut of the clearest line between minimal and maximum data. The first in lines 164-171 is a statement for repetition of potential_splits () data that has been obtained previously in the get_potential_split function. Each potential data is entered into the statement for return in line 165 to display the value in each index. Then in line 166 look for min data and max data using the split data function where the function requires data, col_index and value.

Next on line 167 call the function to get the overall entropy results from minimal data and maximum data. Line 168 is checked if the overall entropy yield is less than high_entropy which is 999, so it will run lines 169-171. Then line 174 to return the results of split_column and split_value.

5.1.5.2 Implement the Decision Tree method

The following is the Decision Tree Program code.

```
175. def decisionTree_fit(df, i=0, min_samples=2, max_depth=2):
176.
177.   # prepare data
178.  if i == 0:
179.          global column_headers
180.          column_headers = df.columns
181.          data = df.values
182.
183.  else:
184.          data = df
185.
186.      # check data pure, or panjang data less then sample or i
   same max_depth
187.  if (check_data_pure(data) or len(data)<min_samples) or (i ==
   max_depth):
188.          classification = klasifikasi(data)
189.          return classification
190.
191.
192.  # i is not 1
193.  else:
194.
195.          i += 1
196.          # function before algorithm (helper)
197.          potential_splits = get_potensial_split(data)
```

```
        split_column, split_value = best_split(data, potential_splits)
        data_min, data_max = split_data(data, split_column, split_value)


        # inisiasi sub-tree
        feature_name = column_headers[split_column]
        question = "{} <= {}".format(feature_name, split_value)
#         print(question)

        sub_tree = {question: []}
#         print(sub_tree)

        # recursif to find question
        small = decisionTree_fit(data_min, i, min_samples, max_depth)
        big = decisionTree_fit(data_max, i, min_samples, max_depth)

        if small == big:
            sub_tree = small
        else:
            sub_tree[question].append(small)
            sub_tree[question].append(big)

        return sub_tree

def predict_X(X, tree):
    question = tree.keys()[0].encode('utf-8')
#     print(question)
    feature_name, operator, value = question.split(" ")


    # ask question
    if operator == "<=":
        if X[feature_name] <= float(value):
            answer = tree[question][0] #answer
        else:
            answer = tree[question][1]

    # recursive
    if isinstance(answer, dict):
        sisa_tree = answer
        return predict_X(X, sisa_tree)

    else:
        return answer


def decisionTree_predict(X_test, tree):
    DT_prediksi = X_test.apply(predict_X, args=(tree,), axis=1)
```

```
246.
247.        return DT_prediksi
```

Above is the Decision Tree program code starting from line 175 to 248. First on lines 178 - 184 is the beginning of incoming data and will be checked if the data has not undergone a decsion tree process then it will run lines 179-181. Meanwhile, if you have experienced the decision tree process, the data has an i value of more than 0, then run row 184.

Then line 187 checks whether the data is pure or not. If true then it will run lines 188-189 which calls the classification function as classifying data. If the data is not pure, enter row 194-199. Where the line calls the helper code such as function get_potential_split (), best_split (), and split_data (). Also gives the value i increased by value 1.

Furthermore, before creating a sub tree, initiation of a sub tree is like row 203-204. At line 203 there is a feature name that contains columns of data, namely tf_idf and category_tfidf. Line 204 will be filled with questions with the operator "<=" with the format of the column name and split value that has been calculated in the split data function. After getting the question results, the results are entered into a dictionary called sub_tree.

To find the appropriate question, the if statement on lines 214-220 is needed, because the operator is less than then to find the question, it needs to be recursive to the decisionTree_fit function. If the result of small value is equal to big, it will display small result. Small and big on lines 211 and 212 are different. If small uses minimal data, while big uses maximum data. Next, row 217 - 218, if the small and big results are different, then it will run lines 217 - 218. Line 220 returns the sub_tree result value.

After getting the results of the sub tree, the sub tree can be used for prediction using the decision tree method at lines 245 - 246. At line 246, the function is to pass the X_test data to the predict_X function with the position argument, namely tree. The test data will enter the predict_X function with X, the test data and the tree using the results of the sub tree that has been obtained. At line 223 is the initialization of the tree in the keys in list to 0. Then it is divided into 3 parts, namely feature_name, operator, value. After it is broken into 3 parts, then it will be checked with an if statement. If the operator is equal to "<=" then it will run lines 230 - 233. In line 230 there is a re-check of the test data with feature_name less than the value, then enter row 231 and produce a branch value in list 0, if wrong will enter line 233 and produces the second branch value in list 1.

Next, line 236 - 241 checks whether this answer is a dictionary type or not. If true, then it will run lines 237 - 239 containing the answer, the line will be forwarded to the predict_X function with test and answer data. If it is incorrect then it will run line 241 which is the return value from the answer that was previously checked in line 230.

5.1.5.3 Implementation of the Random Forest algorithm

Following is the implementation of the Random Forest algorithm.

```
248.  #Predict RANDOM FOREST
249.
250.  # random data
251. def split_random_data(X, n_data):
252.     index_data = np.random.randint(low=0, high=len(X)-1,
     size=n_data)
253.     split_random_df = X.iloc[index_data]
254.     return split_random_df
255.
256. def randomforest_fit(X, n_trees, n_data, max_depth):
257.     rf = []
258.     for i in range(n_trees):
259.         split_random_df = split_random_data(X, n_data)
260.         tree = decisionTree_fit(split_random_df,
     max_depth=max_depth)
261.         rf.append(tree)
262.
263.     return rf
264.
265. def randomforest_predict(X_test, rf):
266.     data_predict = {}
267.
268.
```

```
269.        for i in range(len(rf)):
270.            column_name = "{}".format(i)
271.
272.            if isinstance(rf[i], dict):
273.                print(rf[i])
274.                column_name = "i{}".format(i)
275.                  predict     =     decisionTree_predict(X_test,
  tree=rf[i])
276.                data_predict[column_name] = predict
277.                data_prediksi = pd.DataFrame(data_predict)
278.            else:
279.                continue
280.
281.          random_forest_predictions = data_prediksi.mode(axis=1)
  [0]
282.        return random_forest_predictions
```

After getting the tree results from the decision tree calculation, enter the Random Forest algorithm stage. Where this stage starts from breaking down branches based on the number of n_trees and dividing the data in each branch by n_data.

The next step is to enter the decision tree using random data. So get a tree from Random Forest. After getting the results of the Random Forest tree, the next step is to implement using testing data based on the tree that is trained using training data.

In lines 252 - 254 is the first stage function which is to choose data randomly which will be placed in the tree branch in a row of 256. Then enter the second stage which is to place several branches of the tree and implement using the decision tree method. Line 258 is a repetition of each number of n_trees, and entering line 258-263 there are random data from data X as many as n_data. Then each branch enters a decision tree function with a number of max_depth. The result of the decision tree function is a tree, the results will be placed in a list called rf and returned in the rf value 263.

Line 265 is the Random Forest prediction function that requires test data, and the calculated rf results. In line 269, the number of rf lengths is repeated. Next, each value is checked and passed to the decision tree function shown in line 275, using test data and the tree uses the results of rf every i. Then the prediction results are added to the dictionary named data_predict. Meanwhile, if the answer is not dictionary, it will be skipped, because it cannot enter the prediction function using the decision tree. After getting each predictive data, the next step in line 281 is to determine the most repeatable data frame value. The results from row 281 will be returned in row 282.

5.1.6 Implementation uses the Support Vector Machine algorithm

The following is the Vector Machine Support program code.

```
283. import numpy as np
284.
285.  class SVM :
286. def__init__(self,learning_rate=0.0001,lambda_parameter=0.01,
  n_iter=100):
287.          #constructors
288.          self.learning_rate=learning_rate
289.          self.lambda_parameter = lambda_parameter
290.          self.n_iter=n_iter
291.          self.w = None
292.          self.b = None
293.
294. def fit(self,X,y):
295.          n_sample=len(X)
296.          n_feature=2
297.          y = np.where(y<=0,-1,1)
298.          self.w = np.zeros(n_feature)
299.          self.b = 0
300.
301.          for x in range(self.n_iter):
302.              for i,xi in enumerate(X):
303.                   kondisi=   y[i]    *    (np.dot(self.w,xi)-
  self.b)>=1

304.                 if kondisi:
305.                     self.w = self.w - self.learning_rate * (2
  * self.lambda_parameter * self.w)
306.                 else:
307.                     self.w = self.w - self.learning_rate * (2
  * self.lambda_parameter * self.w - np.dot(y[i],xi))
308.                       self.b = self.b - self.learning_rate *
  y[i]
309.  def predict(self,X):
310.          hasil = np.dot(X,self.w)-self.b
311.          return np.where(hasil<=0,-1,1)
```

Above is the program code for implementation using the support vector machine algorithm. Line 283 aims to activate the numpy library, lines 286 - 292 have a def init that is a function constructor in a class in the python programming language. The init function is used to initialize objects from the class. Inside the def init there are learning rate variables, lambda parameters, n_iter, w and b. While self means that the method is owned and registered in the class to distinguish methods outside and functions outside the class. Lines 291 and 292 are given the value None, which means they have no value and are not equal to 0. The formula for the Support Vector Machine algorithm for linear models: in the program code already has the variable w as the hyperplane being sought and b as the hyperplane bias that is being sought.

Next on line 294-308 is a function fit that is used for machines in learning training data. Line 295 is n_samples which contains the length of the line in data X, line 298 has n_feature which has value 2. This n_feature variable has value 2 because in data X there are 2 columns that will be used in the analysis later. Then line 297 is y which contains values 0 and 1. If it is 0 then hoax and if it is 1 then real, in the above code is if it has y which is less than or equal to 0 then it is -1, and if y is more than 0 then given a value of 1. Then line 298 there is np.zeros gives the value of n_feature to 0 and forms an array.

Line 301 is a loop that is limited to n_iter that is 100. Similarly line 302 is an array loop in data X, which contains i (index) and xi (data value). Next there are conditions on lines 304 - 308 if y index times w. xi - b which is worth more than 1 or equal to 1. If true then enter the formula at line 305 while if wrong enter into 307 and 308.

Then lines 309 - 311 are prediction functions to produce predictive values which will be tested using testing data. Line 310 is a linear formula from, where w and b are obtained from the final result of the function fitting class. The last line of line 311 is to return the result of row 310 where if y is less than 0 or equal to 0 then it is -1 and if more than 0 then it is 1.

5.1.7 Calculation of Accuracy, Precision, Recall, F1-Score

The following program code calculates accuracy, precision, recall, f1-score using predictions from Support Vector Machine.

```
from sklearn.metrics import confusion_matrix
#SVM
def performance(y_test_title, prediksi_svm):

    #matrix

    tp = confusion_matrix(y_test_title, prediksi_svm).item(0) #TP
    fn = confusion_matrix(y_test_title, prediksi_svm).item(1) #FP
    fp = confusion_matrix(y_test_title, prediksi_svm).item(2) #FN
    tn = confusion_matrix(y_test_title, prediksi_svm).item(3) #TN


    # (TP + TN ) / (TP+FP+FN+TN)
    accuracy_svm = (tp+tn)/float((tp+fp+fn+tn))

    # (TP) / (TP+FP)
    precision_svm = tp/float((tp+fp))

    # (TP) / (TP + FN)
    recall_svm = tp/float((tp+fn))

    #  2 * (Recall*Precission) / (Recall + Precission)
    f1score_svm=2*(recall_svm*precision_svm)/float((recall_svm+precision_svm))
    return tp,fn,fp,tn,accuracy_svm,precision_svm,recall_svm,f1score_svm
```

The following program code calculates accuracy, precision, recall, f1-score using predictions from Random Forest.

```
# Random Forest
def performance(y_test_title, prediksi_rf):

    #matrix

    tp = confusion_matrix(y_test_title, prediksi_rf).item(0) #TP
    fn = confusion_matrix(y_test_title, prediksi_rf).item(1) #FP
    fp = confusion_matrix(y_test_title, prediksi_rf).item(2) #FN
    tn = confusion_matrix(y_test_title, prediksi_rf).item(3) #TN


```

```
347.      # (TP + TN ) / (TP+FP+FN+TN)
348.      accuracy_rf = (tp+tn)/float((tp+fp+fn+tn))
349.
350.      # (TP) / (TP+FP)
351.      precision_rf = tp/float((tp+fp))
352.
353.      # (TP) / (TP + FN)
354.      recall_rf = tp/float((tp+fn))
355.
356.      #  2 * (Recall*Precission) / (Recall + Precission)
357.      f1score_rf=2*(recall_svm*precision_svm)/float((recall_sv
   m+precision_svm))
358.
359.
360.                                                     return
   tp,fn,fp,tn,accuracy_rf,precision_rf,recall_rf,f1score_rf
```

In section 5.1.7, the program code calculates accuracy, precision, recall, f1-score. First activate the library from sklearn namely classification_report, confusion_matrix. The two libraries are to search for true positive, false negative, false positive, and false negative, as in lines 341-344. The variable tp is true positive, fn is false negative, fp is false positive, and finally tn is true negative.

Then in line 347 is the formula to calculate accuracy is true positive plus true negative after that the results are divided by the sum of true positive, false positive, false negative, true negative.

Then the formula for calculating precision in line 351 is positive true divided by the sum of true positive and false positive. Next calculate the recall in line 354 is the total positive true value divided by the results of the sum of the total true positive values and the total false negative value. The last formula to calculate f1-score is 2 times the product of recall and precision multiplied by the sum of recall and precision. Line 357 returns the results of the calculation of accuracy, precision, recall, f1-score, true positive, false negative, false positive, and true negative.

Line 337 to 361 is the same as line 315 - 336 except that the line uses the prediction results using the Support Vector Machine algorithm while lines 337 to 361 use the prediction results using the Random Forest algorithm.

5.2 Testing

5.2.1 Training data and Testing data

Table 36: Training Data and Testing Data Tables

No	Pembanding	All Training Data	Training Data (Hoax)	Testing Data (Real)	All Testing Data	All Training Data and Testing Data
1	Training 60%, Testing 40%	644	283 data	361 data	456	1100
2	Training 70%, Testing 30%	760	357 data	403 data	340	1100
3	Training 80%, Testing 20%	886	413 data	473 data	214	1100
4	Training 90%, Testing 10%	1007	457 data	550 data	93	1100

Table 5.1 is a table of training data and testing data which will be tested, and trained in a machine. Training data and testing data have a portion of each - each as a differentiator with other data. The four data will be processed, trained and tested using two algorithms including Random Forest algorithm and Support Vector Machine algorithm.

5.2.2 Analysis

Table 37: Analysis Table of the News Title section

Title								
No	Training Data	Testing Data	Run Time	similarity to the original data	Random Forest Algorithm			
					Accuracy	Precision	Recall	F1-Score

1	644	456	6 second	295 data	64.83516483516483%	36.86635944700461%	77.66990291262135%	50%
2	760	340	6 second	294 data	64.61538461538461%	36.405529953917046%	77.45098039215686%	49.52978056426331%
3	886	214	7 second	148 data	69.1588785046729%	29.885057471264377%	83.87096774193549%	44.067796610169495%
4	1007	93	7 second	67 data	72.04301075268818%	39.534883720930233%	100%	56.666666666666664%
Title								
No	**Training Data**	**Testing Data**	**Run Time**	**similarity to the original data**	**Support Vector Machine Algorithm**			
					Accuracy	**Precision**	**Recall**	**F1-Score**
1	644	456	4 second	226 data	49.67032967032967%	95.39170506912443%	48.59154929577465%	64.38569206842924%
2	760	340	5 second	215 data	63.609467455621306%	24.475524475524477%	70%	36.26943005181347%
3	886	214	5 second	139 data	64.95327102803739%	17.24137931034483%	83.33333333333334%	28.571428571428577%
4	1007	93	5 second	61 data	65.59139784946237%	30.23255813953488%	86.66666666666667%	44.82758620689655%

Table 5.2 is the analysis table of the title section based on the table shows that the time needed to analyze news title using the Support Vector Machine algorithm is faster than the Random Forest 1-2 second difference. Random Forest obtained the highest accuracy value of 72.04301075268818% with training data of 1007 data and the lowest accuracy value of 64.61538461538461% with the amount of training data of 760 data. Then from the test results using Support Vector Machine obtained the highest accuracy value is 65.59139784946237%

with training data of 1007 data and the lowest accuracy value is 49.67032967032967% with the number of training data 644 data. Also the results of the two algorithms in analyzing the news title section have increased accuracy.

Table 38: Analysis Table of the News Content section

Content								
No	**Training Data**	**Testing Data**	**Run Time**	**similarity to the original data**	**Random Forest Algorithm**			
					Accuracy	**Precision**	**Recall**	**F1-Score**
1	644	456	7 second	265 data	58.24175824175825%	12.442396313364055%	100%	22.131147540983605%
2	760	340	7 second	201 data	59.46745562130178%	4.195804195804196%	100%	8.053691275167786%
3	886	214	7 second	134 data	62.616822429906534%	8.045977011494253%	100%	14.893617021276595%
4	1007	93	8 second	59 data	63.44086021505376%	20.930232558139537%	100%	34.61538461538462%

Content								
No	**Training Data**	**Testing Data**	**Run Time**	**similarity to the original data**	**Support Vector Machine Algorithm**			
					Accuracy	**Precision**	**Recall**	**F1-Score**
1	644	456	4 second	212 data	46.59340659340659%	97.6958525345622%	47.111111111111111%	63.56821589205397%
2	760	340	4 second	195 data	57.692307692307686%	82.51748251748252%	50%	62.26912928759894%
3	886	214	4 second	86 data	40.18691588785047%	94.25287356321839%	40%	56.16438356164384%

4	1007	93	4 second	43 data	46.236559139784944%	90.69767441860465%	45.88235294117647%	60.9375 %

Table 5.3 is a content analysis table based on the table shows that the time needed for news content analysis using the Support Vector Machine algorithm is faster than the Random Forest 3-4 seconds difference. Random Forest obtained the highest accuracy value of 63.44086021505376% with training data of 1007 data and the lowest accuracy value of 58.24175824175825% with the number of training data 644 data. Then from the test results using the Support Vector Machine algorithm has the highest accuracy value of 57.692307692307686% with training data of 760 data and the lowest accuracy value of 40.18691588785047% with the amount of training data 886 data. The decrease in accuracy occurs in the analysis using the algorithm support vector machine which initially 57.692307692307686% to 40.18691588785047%. This shows that the TF-IDF weight value can affect the decrease in accuracy.