**PROGRAM:**

class PREDICATE:

def \_\_str\_\_(self):

pass

def \_\_repr\_\_(self):

pass

def \_\_eq\_\_(self, other) :

pass

def \_\_hash\_\_(self):

pass

def get\_action(self, world\_state):

pass

class Operation:

def \_\_str\_\_(self):

pass

def \_\_repr\_\_(self):

pass

def \_\_eq\_\_(self, other) :

pass

def precondition(self):

pass

def delete(self):

pass

def add(self):

pass

class ON(PREDICATE):

def \_\_init\_\_(self, X, Y):

self.X = X

self.Y = Y

def \_\_str\_\_(self):

return "ON({X},{Y})".format(X=self.X,Y=self.Y)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def \_\_hash\_\_(self):

return hash(str(self))

def get\_action(self, world\_state):

return StackOp(self.X,self.Y)

class ONTABLE(PREDICATE):

def \_\_init\_\_(self, X):

self.X = X

def \_\_str\_\_(self):

return "ONTABLE({X})".format(X=self.X)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def \_\_hash\_\_(self):

return hash(str(self))

def get\_action(self, world\_state):

return PutdownOp(self.X)

class CLEAR(PREDICATE):

def \_\_init\_\_(self, X):

self.X = X

def \_\_str\_\_(self):

return "CLEAR({X})".format(X=self.X)

self.X = X

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def \_\_hash\_\_(self):

return hash(str(self))

def get\_action(self, world\_state):

for predicate in world\_state:

if isinstance(predicate,ON) and predicate.Y==self.X:

return UnstackOp(predicate.X, predicate.Y)

return None

class HOLDING(PREDICATE):

def \_\_init\_\_(self, X):

self.X = X

def \_\_str\_\_(self):

return "HOLDING({X})".format(X=self.X)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def \_\_hash\_\_(self):

return hash(str(self))

def get\_action(self, world\_state):

X = self.X

if ONTABLE(X) in world\_state:

return PickupOp(X)

else:

for predicate in world\_state:

if isinstance(predicate,ON) and predicate.X==X:

return UnstackOp(X,predicate.Y)

class ARMEMPTY(PREDICATE):

def \_\_init\_\_(self):

pass

def \_\_str\_\_(self):

return "ARMEMPTY"

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def \_\_hash\_\_(self):

return hash(str(self))

def get\_action(self, world\_state=[]):

for predicate in world\_state:

if isinstance(predicate,HOLDING):

return PutdownOp(predicate.X)

return None

class StackOp(Operation):

def \_\_init\_\_(self, X, Y):

self.X = X

self.Y = Y

def \_\_str\_\_(self):

return "STACK({X},{Y})".format(X=self.X,Y=self.Y)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def precondition(self):

return [ CLEAR(self.Y) , HOLDING(self.X) ]

def delete(self):

return [ CLEAR(self.Y) , HOLDING(self.X) ]

def add(self):

return [ ARMEMPTY() , ON(self.X,self.Y) ]

class UnstackOp(Operation):

def \_\_init\_\_(self, X, Y):

self.X = X

self.Y = Y

def \_\_str\_\_(self):

return "UNSTACK({X},{Y})".format(X=self.X,Y=self.Y)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def precondition(self):

return [ ARMEMPTY() , ON(self.X,self.Y) , CLEAR(self.X) ]

def delete(self):

return [ ARMEMPTY() , ON(self.X,self.Y) ]

def add(self):

return [ CLEAR(self.Y) , HOLDING(self.X) ]

class PickupOp(Operation):

def \_\_init\_\_(self, X):

self.X = X

def \_\_str\_\_(self):

return "PICKUP({X})".format(X=self.X)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def precondition(self):

return [ CLEAR(self.X) , ONTABLE(self.X) , ARMEMPTY() ]

def delete(self):

return [ ARMEMPTY() , ONTABLE(self.X) ]

def add(self):

return [ HOLDING(self.X) ]

class PutdownOp(Operation):

def \_\_init\_\_(self, X):

self.X = X

def \_\_str\_\_(self):

return "PUTDOWN({X})".format(X=self.X)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other) :

return self.\_\_dict\_\_ == other.\_\_dict\_\_ and self.\_\_class\_\_ == other.\_\_class\_\_

def precondition(self):

return [ HOLDING(self.X) ]

def delete(self):

return [ HOLDING(self.X) ]

def add(self):

return [ ARMEMPTY() , ONTABLE(self.X) ]

def isPredicate(obj):

predicates = [ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY]

for predicate in predicates:

if isinstance(obj,predicate):

return True

return False

def isOperation(obj):

operations = [StackOp, UnstackOp, PickupOp, PutdownOp]

for operation in operations:

if isinstance(obj,operation):

return True

return False

def arm\_status(world\_state):

for predicate in world\_state:

if isinstance(predicate, HOLDING):

return predicate

return ARMEMPTY()

class GoalStackPlanner:

def \_\_init\_\_(self, initial\_state, goal\_state):

self.initial\_state = initial\_state

self.goal\_state = goal\_state

def get\_steps(self):

steps = []

stack = []

#World State/Knowledge Base

world\_state = self.initial\_state.copy()

#Initially push the goal\_state as compound goal onto the stack

stack.append(self.goal\_state.copy())

#Repeat until the stack is empty

while len(stack)!=0:

#Get the top of the stack

stack\_top = stack[-1]

#If Stack Top is Compound Goal, push its unsatisfied goals onto stack

if type(stack\_top) is list:

compound\_goal = stack.pop()

for goal in compound\_goal:

if goal not in world\_state:

stack.append(goal)

elif isOperation(stack\_top):

operation = stack[-1]

all\_preconditions\_satisfied = True

for predicate in operation.delete():

if predicate not in world\_state:

all\_preconditions\_satisfied = False

stack.append(predicate)

if all\_preconditions\_satisfied:

stack.pop()

steps.append(operation)

for predicate in operation.delete():

world\_state.remove(predicate)

for predicate in operation.add():

world\_state.append(predicate)

elif stack\_top in world\_state:

stack.pop()

else:

unsatisfied\_goal = stack.pop()

action = unsatisfied\_goal.get\_action(world\_state)

stack.append(action)

for predicate in action.precondition():

if predicate not in world\_state:

stack.append(predicate)

return steps

if \_\_name\_\_ == '\_\_main\_\_':

initial\_state = [

ON('B','A'),ON('E', 'B'),

ONTABLE('A'),ONTABLE('C'),ONTABLE('D'),

CLEAR('B'),CLEAR('C'),CLEAR('D'),CLEAR('E'),

ARMEMPTY()

]

goal\_state = [

ON('B','D'),ON('D','C'), ON('C', 'A'),ON('A', 'E'),

ONTABLE('A'),

CLEAR('B'),CLEAR('C'), CLEAR('D'),CLEAR('E'),

ARMEMPTY()

]

goal\_stack = GoalStackPlanner(initial\_state=initial\_state, goal\_state=goal\_state)

steps = goal\_stack.get\_steps()

print("UNSTACK(E,B)")

print("PUTDOWN(E)")

for i in steps:

print(i)

**OUTPUT:**

****

**PROGRAM:**

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

def splitdataset(balance\_data):

X = balance\_data.values[:, 1:5]

Y = balance\_data.values[:, 0]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, Y, test\_size = 0.3, random\_state = 100)

return X, Y, X\_train, X\_test, y\_train, y\_test

def prediction(X\_test, clf\_object):

y\_pred = clf\_object.predict(X\_test)

print("Predicted values:")

print(y\_pred)

return y\_pred

if \_\_name\_\_=="\_\_main\_\_":

data = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/balance-scale/balance-scale.data', sep= ',', header = None)

X = data.values[:, 1:5]

Y = data.values[:, 0]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size = 0.3, random\_state = 100)

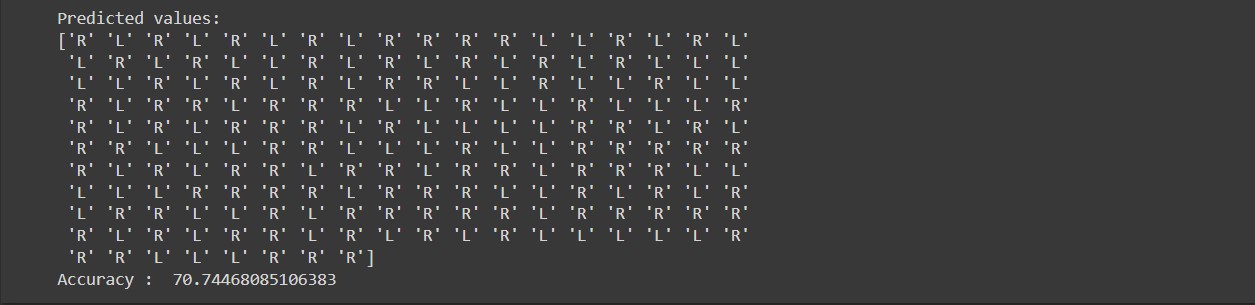
clf\_entropy = DecisionTreeClassifier(criterion = "entropy", random\_state = 100,max\_depth = 3, min\_samples\_leaf = 5)

clf\_entropy.fit(X\_train, y\_train)

y\_pred = prediction(X\_test, clf\_entropy)

print("Accuracy : ", accuracy\_score(y\_test,y\_pred)\*100)

**OUTPUT:**

****

**PROGRAM:**

**# Bagged Decision Trees for Classificationfrom inspect import Traceback**

from pandas import read\_csv

from sklearn.model\_selection import KFold

from sklearn.model\_selection import cross\_val\_score

from sklearn.ensemble import BaggingClassifier

from sklearn.tree import DecisionTreeClassifier

path="/content/pima-indians-diabetes.csv"

headernames = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

data = read\_csv(path, names=headernames)

array = data.values

X = array[:,0:8]

Y = array[:,8]

seed = 7

kfold = KFold(n\_splits = 10,random\_state=seed,shuffle=True)

cart = DecisionTreeClassifier()

num\_trees = 150

model = BaggingClassifier(base\_estimator = cart, n\_estimators = num\_trees, random\_state = seed)

results = cross\_val\_score(model, X, Y, cv=kfold)

print(results.mean())

**# Random Forest Classification**

import pandas

from sklearn import model\_selection

from sklearn.ensemble import RandomForestClassifier

url = "/content/pima-indians-diabetes.csv"

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = pandas.read\_csv(url, names=names)

array = dataframe.values

X = array[:,0:8]

Y = array[:,8]

seed = 7

num\_trees = 100

max\_features = 3

kfold = model\_selection.KFold(n\_splits=10, random\_state=seed,shuffle=True)

model = RandomForestClassifier(n\_estimators=num\_trees, max\_features=max\_features)

results = model\_selection.cross\_val\_score(model, X, Y, cv=kfold)

print(results.mean())

**# Extra Trees Classification**

import pandas

from sklearn import model\_selection

from sklearn.ensemble import ExtraTreesClassifier

url = "/content/pima-indians-diabetes.csv"

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = pandas.read\_csv(url, names=names)

array = dataframe.values

X = array[:,0:8]

Y = array[:,8]

seed = 7

num\_trees = 100

max\_features = 7

kfold = model\_selection.KFold(n\_splits=10, random\_state=seed,shuffle=True)

model = ExtraTreesClassifier(n\_estimators=num\_trees, max\_features=max\_features)

results = model\_selection.cross\_val\_score(model, X, Y, cv=kfold)

print(results.mean())

**# AdaBoost Classification**

import pandas

from sklearn import model\_selection

from sklearn.ensemble import AdaBoostClassifier

url = "/content/pima-indians-diabetes.csv"

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = pandas.read\_csv(url, names=names)

array = dataframe.values

X = array[:,0:8]

Y = array[:,8]

seed = 7

num\_trees = 30

kfold = model\_selection.KFold(n\_splits=10, random\_state=seed,shuffle=True)

model = AdaBoostClassifier(n\_estimators=num\_trees, random\_state=seed)

results = model\_selection.cross\_val\_score(model, X, Y, cv=kfold)

print(results.mean())

**# Stochastic Gradient Boosting Classification**

import pandas

from sklearn import model\_selection

from sklearn.ensemble import GradientBoostingClassifier

url = "/content/pima-indians-diabetes.csv"

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = pandas.read\_csv(url, names=names)

array = dataframe.values

X = array[:,0:8]

Y = array[:,8]

seed = 7

num\_trees = 100

kfold = model\_selection.KFold(n\_splits=10, random\_state=seed,shuffle=True)

model = GradientBoostingClassifier(n\_estimators=num\_trees, random\_state=seed)

results = model\_selection.cross\_val\_score(model, X, Y, cv=kfold)

print(results.mean())

**# Voting Ensemble for Classification**

import pandas

from sklearn import model\_selection

from sklearn.linear\_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.svm import SVC

from sklearn.ensemble import VotingClassifier

url = "/content/pima-indians-diabetes.csv"

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = pandas.read\_csv(url, names=names)

array = dataframe.values

X = array[:,0:8]

Y = array[:,8]

seed = 7

kfold = model\_selection.KFold(n\_splits=10, random\_state=seed,shuffle=True)

# create the sub models

estimators = []

model1 = LogisticRegression()

estimators.append(('logistic', model1))

model2 = DecisionTreeClassifier()

estimators.append(('cart', model2))

model3 = SVC()

estimators.append(('svm', model3))

# create the ensemble model

ensemble = VotingClassifier(estimators)

results = model\_selection.cross\_val\_score(ensemble, X, Y, cv=kfold)

print(results.mean())

**OUTPUT:**

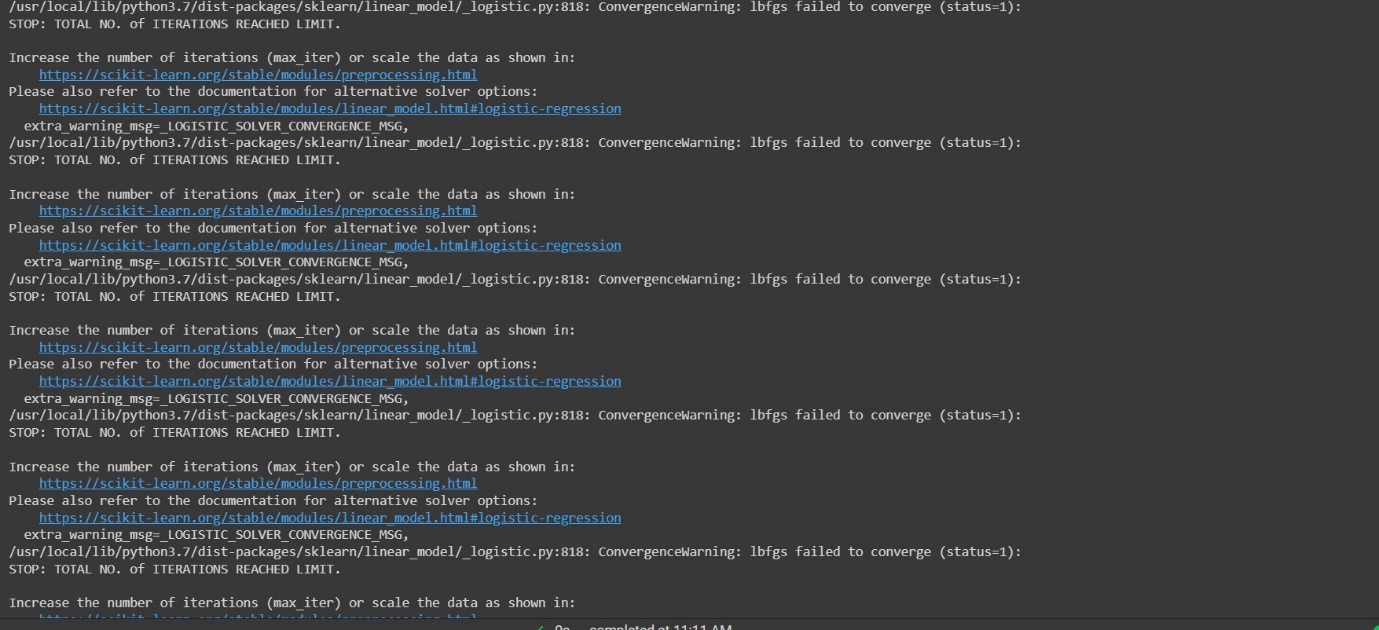
****

****

****

****

****

****

**PROGRAM:**

# Load libraries

from sklearn.ensemble import AdaBoostClassifier

from sklearn import datasets

# Import train\_test\_split function

from sklearn.model\_selection import train\_test\_split

#Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

# Load data

iris = datasets.load\_iris()

X = iris.data

y = iris.target

# Split dataset into training set and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3) # 70% training and 30%

# Create adaboost classifer object

abc = AdaBoostClassifier(n\_estimators=50,

learning\_rate=1)

# Train Adaboost Classifer

model = abc.fit(X\_train, y\_train)

#Predict the response for test dataset

y\_pred = model.predict(X\_test)

# Model Accuracy, how often is the classifier correct?

print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

**OUTPUT:**



**PROGRAM:**

**Sentiment Analysis**

import pandas as pd

import nltk

from nltk.sentiment.vader import SentimentIntensityAnalyzer

nltk.download('all')

# reading and wragling data

df\_avatar = pd.read\_csv('avatar.csv', encoding = 'unicode\_escape', engine='python')

df\_avatar\_lines = df\_avatar.groupby('character').count()

df\_avatar\_lines = df\_avatar\_lines.sort\_values(by=['character\_words'], ascending=False)[:10]

top\_character\_names = df\_avatar\_lines.index.values

# filtering out non-top characters

df\_character\_sentiment = df\_avatar[df\_avatar['character'].isin(top\_character\_names)]

df\_character\_sentiment = df\_character\_sentiment[['character', 'character\_words']]

# calculating sentiment score

sid = SentimentIntensityAnalyzer()

df\_character\_sentiment.reset\_index(inplace=True, drop=True)

df\_character\_sentiment[['neg', 'neu', 'pos', 'compound']] = df\_character\_sentiment['character\_words'].apply(sid.polarity\_scores).apply(pd.Series)

df\_character\_sentiment

## **Plotting sentiment**

import matplotlib.pyplot as plt

import numpy as np

# preparing data

X = np.arange(len(df\_character\_sentiment['pos']))

#bar plot

fig = plt.figure(figsize = (17, 12))

plt.barh(X, df\_character\_sentiment['pos'], facecolor='#9999ff', edgecolor='white')

plt.barh(X, -df\_character\_sentiment['neg'], facecolor='#ff9999', edgecolor='white')

# plt.rcParams.update({'font.size':13})

plt.xlim([-.16,.22])

plt.yticks(ticks=X, labels=df\_character\_sentiment['character'], rotation='0')

plt.show()

# Named Entity Recognition (NER)

# import spacy

# nlp = spacy.load("en\_core\_web\_sm")

# doc = nlp("Biden invites Ukrainian president to White House this summer")

# print([(X.text, X.label\_) for X in doc.ents])

**Stemming**

from nltk.stem import PorterStemmer

from nltk.stem import LancasterStemmer

# PorterStemmer

porter = PorterStemmer()

# LancasterStemmer

lancaster = LancasterStemmer()

print(porter.stem("friendship"))

print(lancaster.stem("friendship"))

# Lemmatization

# from nltk import WordNetLemmatizer

# lemmatizer = WordNetLemmatizer()

# words = ['articles', 'friendship', 'studies', 'phones']

# for word in words:

# print(lemmatizer.lemmatize(word))

# from nltk import WordNetLemmatizer

# lemmatizer = WordNetLemmatizer()

# words = ['be', 'is', 'are', 'were', 'was']

# for word in words:

# print(lemmatizer.lemmatize(word, pos='v'))

# Bag of Words (BoW)

# import pandas as pd

# from sklearn.feature\_extraction.text import CountVectorizer

# text = ["I love writing code in Python. I love Python code",

# "I hate writing code in Java. I hate Java code"]

# df = pd.DataFrame({'review': ['review1', 'review2'], 'text':text})

# cv = CountVectorizer(stop\_words='english')

# cv\_matrix = cv.fit\_transform(df['text'])

# df\_dtm = pd.DataFrame(cv\_matrix.toarray(),

# index=df['review'].values,

# columns=cv.get\_feature\_names())

# df\_dtm

# TF-IDF

# import pandas as pd

# from sklearn.feature\_extraction.text import TfidfVectorizer

# text = ["I love writing code in Python. I love Python code",

# "I hate writing code in Java. I hate Java code"]

# df = pd.DataFrame({'review': ['review1', 'review2'], 'text':text})

# tfidf = TfidfVectorizer(stop\_words='english', norm=None)

# tfidf\_matrix = tfidf.fit\_transform(df['text'])

# df\_dtm = pd.DataFrame(tfidf\_matrix.toarray(),

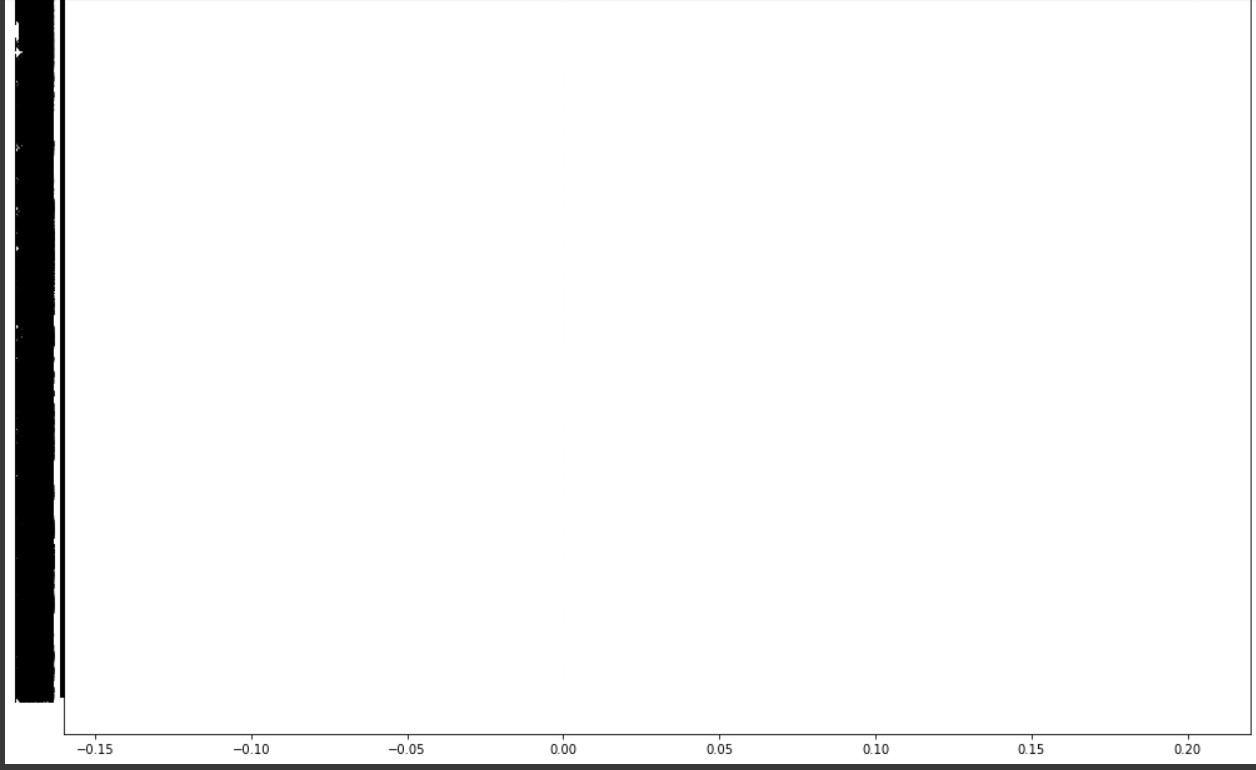
# index=df['review'].values,

# columns=tfidf.get\_feature\_names())

# df\_dtm

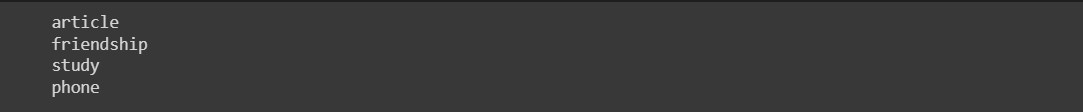
**OUTPUT**

****

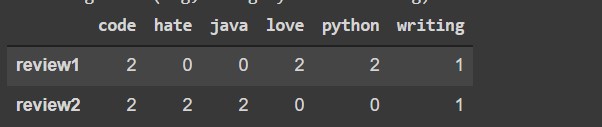
****

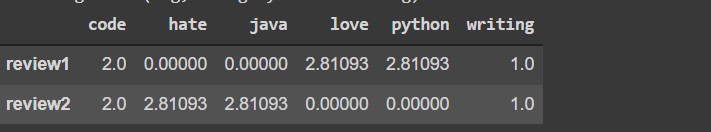
****

****

****

****

****

****