1.Demonstrate Noise Removal for any textual data and remove regular expression pattern such as hash tag from textual data.

AIM:- Demonstrate Noise Removal for any textual data and remove regular expression pattern such as hash tag from textual data.

DESCRIPTION:-

Noise Removal

Text cleaning is a technique that developers use in a variety of domains. Depending on the goal of your project and where you get your data from, you may want to remove unwanted information, such as:

- Punctuation and accents
- Special characters
- Numeric digits
- Leading, ending, and vertical whitespace
- HTML formatting

Here is an example of how lowercasing solves the sparsity issue, where the same words with different cases map to the same lowercase form:

Raw	Lowercased
Canada CanadA CANADA	canada
TOMCAT Tomcat toMcat	tomcat

Another example where lowercasing is very useful is for search. Imagine, you are looking for documents containing “usa”. However, no results were showing up because “usa” was indexed as **“USA”.** Now, who should we blame? The U.I. designer who set-up the interface or the engineer who set-up the search index?

While lowercasing should be standard practice, I’ve also had situations where preserving the capitalization was important. For example, in predicting the programming language of a source code file. The word System in Java is quite different from system in python. Lowercasing the two makes them identical, causing the classifier to lose important predictive features. While lowercasing *is generally* helpful, it may not be applicable for all tasks.

DATE:-_____ PGNO:-__

PROGRAM:-

```
import re

def remove_noise(text):

    # Remove hash tags

    text = re.sub(r'#\w+', '', text)

        # Remove URLs

    text = re.sub(r'http\S+', '', text)

      # Remove mentions

    text = re.sub(r'@\w+', '', text)

    return text

text = "This is a sample text with #hashtags, URLs (http://example.com) and mentions (@mention) to remove."

noisy_text = remove_noise(text)

print(noisy_text)
```

RESULT:-

The program is executed successfully without any errors.

2. Perform lemmatization and stemming using python library nltk.

AIM:- Perform lemmatization and stemming using python library nltk.

DESCRIPTION:-

First, let's understand what lemmatization and stemming are:

- Lemmatization: It is the process of reducing a word to its base or dictionary form (known as the lemma), by considering the context and morphological analysis of the word. For example, the lemma of the word "running" is "run", and the lemma of "am" is "be". Lemmatization produces a meaningful word that can be used in natural language processing applications.
- Stemming: It is the process of reducing a word to its root or base form, by removing the suffixes or prefixes. For example, the stem of the word "running" is "run", and the stem of "am" is "am". Stemming produces an approximation of the base form of a word that can be used in search applications.

Now, let's see how to perform lemmatization and stemming using the NLTK library in Python:

1. Import the NLTK library:

   import nltk

2. Download the necessary NLTK resources for lemmatization and stemming:

```
nltk.download('wordnet')

nltk.download('punkt')
```

nltk.download('averaged_perceptron_tagger')

3. Import the WordNetLemmatizer and PorterStemmer classes from the NLTK library:

   from nltk.stem import WordNetLemmatizer, PorterStemmer

DATE:-_____ PGNO:-__

PROGRAM:-

```
import nltk

from nltk.tokenize import word_tokenize

from nltk.stem import WordNetLemmatizer, PorterStemmer

# Sample text

text = "The boys were running and jumping in the park"

# Tokenize the text

tokens = word_tokenize(text)

# Perform lemmatization using WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

lemmatized_tokens = [lemmatizer.lemmatize(word) for word in tokens]

print("Lemmatized tokens:", lemmatized_tokens)

# Perform stemming using PorterStemmer

stemmer = PorterStemmer()

stemmed_tokens = [stemmer.stem(word) for word in tokens]

print("Stemmed tokens:", stemmed_tokens)
```

RESULT:-

The program is executed successfully without any errors.

DATE:-_____ PGNO:-__

3.Demonstrate object standardization such as replace social media slangs from a text.

AIM:- Demonstrate object standardization such as replace social media slangs from a text.

DESCRPITION:-

Social media slangs are informal language used on social media platforms that may not be recognized by traditional spell-checking tools or natural language processing algorithms. Standardizing these slangs to their proper word forms can help improve the accuracy of natural language processing models.

Here's how you can perform object standardization using Python:

1.Import the re library for regular expressions:

```
import re
```

2. Define a dictionary of social media slangs and their corresponding standard forms:

```
slang_dict = { 'lol': 'laughing out loud', 'brb': 'be right back', 'btw': 'by the way', 'omg': 'oh my god', 'ikr': 'I know, right?', 'afk': 'away from keyboard' }
```

DATE:-_____ PGNO:-__

PROGRAM:-

```
import re
# Sample text with social media slangs
text = "OMG! that was so lit yesterday □□"
# Define a dictionary to store the slangs and their standard form
slangs = {
    "OMG": "Oh My God",
    "lit": "cool",
}
# Use regular expressions to replace the slangs with their standard form
for slang, standard in slangs.items():
    text = re.sub(r"\b" + slang + r"\b", standard, text)
print("Standardized text:", text)
```

RESULT:-

The program is executed successfully without any errors.

4.Perform part of speech tagging on any textual data.

AIM:- Perform part of speech tagging on any textual data.

Perform part of speech tagging on any textual data.

DESCRPITION:-

1. Import the necessary NLTK modules:

```
import nltk from nltk.tokenize
 import word_tokenize from nltk
 import pos_tag
```

2. Define the textual data that you want to perform POS tagging on:

```
text = "John is eating a delicious pizza with his friend at the restaurant."
```

3. Tokenize the text into words:

```
words = word_tokenize(text)
```

4. Perform POS tagging on the words:

```
pos_tags = pos_tag(words)
```

As you can see, the POS tagging has identified the parts of speech of each word in the text. The tags are represented using the Penn Treebank POS tagset, which includes tags such as "NN" for noun, "VBG" for verb (gerund), "JJ" for adjective, "PRP$" for possessive pronoun, "IN" for preposition, etc.

You can also use the **pos_tag_sents** function to perform POS tagging on multiple sentences at once. Simply pass a list of tokenized sentences as input to the function, and it will return a list of POS tagged sentences.

DATE:-_____ PGNO:-__

PROGRAM:-

```
import nltk

nltk.download('punkt')

nltk.download('averaged_perceptron_tagger')

def pos_tagging(text):

    # Tokenize the text into words

    words = nltk.word_tokenize(text)

    # Perform POS tagging on the tokenized words

    tagged_words = nltk.pos_tag(words)

    return tagged_words

text = "This is a sample text for part-of-speech tagging."

tagged_text = pos_tagging(text)

print(tagged_text)
```

RESULT:-

The program is executed successfully without any errors.

5.Implement topic modeling using Latent Dirichlet Allocation (LDA) in python.

AIM:- Implement topic modeling using Latent Dirichlet Allocation (LDA) in python.

DESCRIPTION:-

Latent Dirichlet Allocation (LDA) is a widely used topic modeling technique that models topics as a probability distribution over words, and documents as a probability distribution over topics. The LDA algorithm aims to find the best topic-word and document-topic distributions that can generate the observed corpus of documents.

In this implementation, the **preprocess** function takes a document as input and returns its preprocessed version, which is then used to generate a dictionary and a bag of words corpus. Finally, the **gensim.models.LdaMulticore** function is used to fit an LDA model to the corpus, and the topics are printed using the **print_topics** method.

Note that this is just one implementation of LDA and you may want to fine-tune the preprocessing step and the parameters of the LDA model to get the best results for your particular use case.

DATE:-_____ PGNO:-__

PROGRAM:-

```
# Import libraries

from sklearn.datasets import fetch_20newsgroups

from sklearn.feature_extraction.text import CountVectorizer

from sklearn.decomposition import LatentDirichletAllocation

# Load data

newsgroups_train = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'))

data_samples = newsgroups_train.data[:1000]

# Create vectorizer

vectorizer = CountVectorizer(max_df=0.95, min_df=2, max_features=1000, stop_words='english')

data_vectorized = vectorizer.fit_transform(data_samples)

# Build LDA model

lda_model = LatentDirichletAllocation(n_components=10, max_iter=10, learning_method='online', random_state=42)

lda_Z = lda_model.fit_transform(data_vectorized)

# Print top 10 words for each topic

feature_names = vectorizer.get_feature_names()

for topic_idx, topic in enumerate(lda_model.components_):
```

```
print("Topic #%d:" % topic_idx)

print(" ".join([feature_names[i] for i in topic.argsort()[:-10 - 1:-1]]))

print()
```

RESULT:-

The program is executed successfully without any errors.

6. Demonstrate Term Frequency – Inverse Document Frequency (TF – IDF) using python.

AIM:- Demonstrate Term Frequency – Inverse Document Frequency (TF – IDF) using python.

DESCRIPTION:-

TF-IDF stands for Term Frequency Inverse Document Frequency of records. It can be defined as the calculation of how relevant a word in a series or corpus is to a text. The meaning increases proportionally to the number of times in the text a word appears but is compensated by the word frequency in the corpus (data-set).

Terminologies:

Term Frequency: In document d, the frequency represents the number of instances of a given word t. Therefore, we can see that it becomes more relevant when a word appears in the text, which is rational. Since the ordering of terms is not significant, we can use a vector to describe the text in the bag of term models. For each specific term in the paper, there is an entry with the value being the term frequency.

The weight of a term that occurs in a document is simply proportional to the term frequency.

tf(t,d) = count of t in d / number of words in d

Document Frequency: This tests the meaning of the text, which is very similar to TF, in the whole corpus collection. The only difference is that in document d, TF is the frequency counter for a term t, while df is the number of occurrences in the document set N of the term t. In other words, the number of papers in which the word is present is DF.

$$df(t) = \text{occurrence of t in documents}$$

Inverse Document Frequency: Mainly, it tests how relevant the word is. The key aim of the search is to locate the appropriate records that fit the demand. Since tf considers all terms equally significant, it is therefore not only possible to use the term frequencies to measure the weight of the term in the paper. First, find the document frequency of a term t by counting the number of documents containing the term:

$$df(t) = N(t)$$

where

df(t) = Document frequency of a term t

N(t) = Number of documents containing the term t

Term frequency is the number of instances of a term in a single document only; although the frequency of the document is the number of separate documents in which the term appears, it depends on the entire corpus. Now let's look at the definition of the frequency of the inverse paper. The IDF of the word is the number of documents in the corpus separated by the frequency of the text.

$$idf(t) = N/\ df(t) = N/N(t)$$

The more common word is supposed to be considered less significant, but the element (most definite integers) seems too harsh. We then take the logarithm (with base 2) of the inverse frequency of the paper. So the if of the term t becomes:

$$idf(t) = \log(N/\ df(t))$$

DATE:-_____ PGNO:-__

PROGRAM:-

```
from sklearn.feature_extraction.text

import TfidfVectorizer

import pandas as pd

documents = [    "The sky is blue.",    "The sun is bright.",    "The sun in the sky is bright."]

tfidf_vectorizer = TfidfVectorizer()

tfidf_matrix = tfidf_vectorizer.fit_transform(documents)

df = pd.DataFrame(tfidf_matrix.toarray(),
columns=tfidf_vectorizer.get_feature_names())

print(df)
```

RESULT:-

The program is executed successfully without any errors.

7. Demonstrate word embeddings using word2vec.

AIM:- Demonstrate word embeddings using word2vec**.**

DESCRIPTION:-

Word2vec is a popular method for learning dense, continuous-valued representations for words called "word embeddings". The **gensim** library in Python provides an implementation of word2vec. Here is an example of how you can use it.

In this example, we define a list of sentences, where each sentence is represented as a list of words. We then train a Word2Vec model on this list of sentences using the **Word2Vec** class from the **gensim** library. The **size** parameter specifies the dimensionality of the word embeddings, the **window** parameter specifies the maximum distance between the target word and its neighbors, and the **min_count** parameter specifies the minimum frequency of words to be included in the model.

Once the model is trained, you can access the word embeddings using the **wv** property of the model. In this example, we print the word embedding for the word "cat" and the similarity between the words "cat" and "dog". The similarity is computed as the cosine similarity between the word embeddings.

This is just a simple example, and in practice you may want to preprocess the text, filter out stop words, and use a larger corpus to train the model for better results.

DATE:-_____ PGNO:-__

Program:-

```
import gensim

from gensim.models import Word2Vec

sentences = [["cat", "say", "meow"], ["dog", "say", "woof"]]

model = Word2Vec(sentences, size=100, window=5, min_count=1, workers=4)

print(model.wv["cat"])

print(model.wv.similarity("cat", "dog"))
```

RESULT:-

The program is executed successfully without any errors.

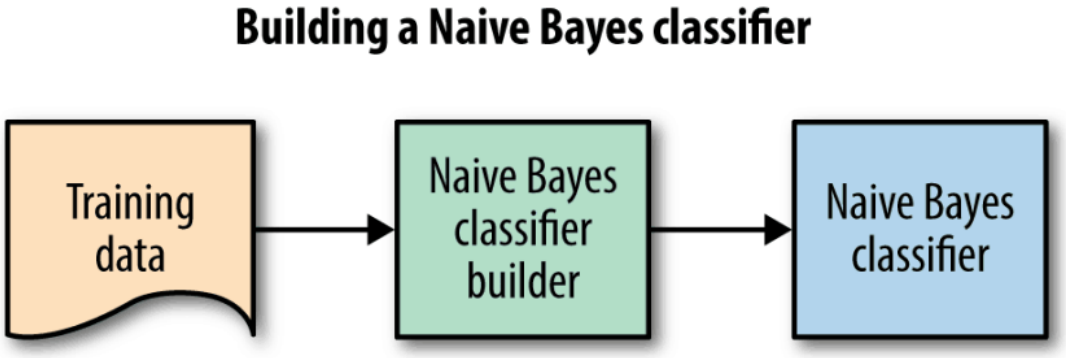
8. Implement Text classification using naïve bayes classifier and text blob library.

AIM:- Implement Text classification using naïve bayes classifier and text blob library,

DESCRIPTION:-

Naive bayesian text classifier using textblob and python

Text classifier are systems that classify your texts and divide them in different classes. In this article we are going to made one such text classifier using textblob and python. You want to read more about **naive bayesian theorem, read it here**.



Naive bayesian text classifier using textblob and python

For this we will be using **textblob**, a library for simple text processing. It provides a simple API for diving into common natural language processing (NLP) tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis, classification, translation, and more.

We will do this in separate python environment for this we need **virtualenv**. How to install **virtualenv**.

sudo pip install virtualenv

Now we have installed virtualenv next step is to create virtaul environment for our little project. Run the below command to create virtualenv.

virtualenv sent

sent is the name of the environment. Now we have created an environment. The above command will create an environment and install setup tools in it. Now we need to launch the environment. For this, run the below command

. sent/bin/activate

Now for installing textblob use below commands

pip install textblob

python -m textblob.download_corpora

The second command will download the data files that textblob uses for its functionality and for **nltk**.

DATE:-_____ PGNO:-__

PROGRAM:-

```
from textblob.classifiers import NaiveBayesClassifier

from textblob import TextBlob

training_data = [ ('This is an positive example', 'pos'), ('This is an negative example', 'neg'), ('This is an average example', 'neutral')]

clf = NaiveBayesClassifier(training_data)

test_data = "This is an positive example"

blob = TextBlob(test_data, classifier=clf)

print(blob.classify())
```

RESULT:-

The program is executed successfully without any errors.

DATE:-_____ PGNO:-__

9. Apply support vector machine for text classification.

AIM:- Apply support vector machine for text classification.

DESCRIPTION:-

There are many different machine learning algorithms we can choose from when doing text classification with machine learning. One of those is Support Vector Machines (or SVM).

In this article, we will explore the advantages of using support vector machines in text classification and will help you get started with SVM-based models with MonkeyLearn.

From Texts to Vectors

Support vector machines is an algorithm that determines the best decision boundary between vectors that belong to a given group (or category) and vectors that do not belong to it.

It can be applied to any kind of vectors which encode any kind of data. This means that in order to leverage the power of svm text classification, texts have to be transformed into vectors.

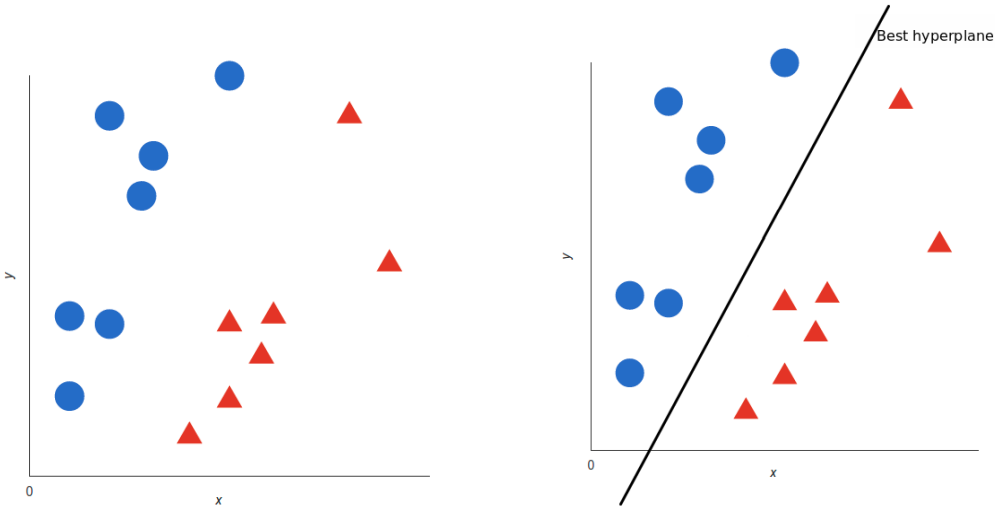
Now, what are vectors?

Vectors are (sometimes huge) lists of numbers which represent a set of coordinates in some space.

So, when SVM determines the decision boundary we mentioned above, SVM decides where to draw the best “line” (or the best hyperplane) that divides the space into two subspaces: one for the vectors which belong to the given category and one for the vectors which do not belong to it.

So, provided we can find vector representations which encode as much information from our texts as possible, we will be able to apply the SVM algorithm to text classification problems and obtain very good results.

Say, for example, the blue circles in the graph below are representations of training texts which talk about the *Pricing* of a SaaS Product and the red triangles are representations of training texts which do not talk about that. What would the decision boundary for the *Pricing* category look like?



The best decision boundary would look like this:

Now that the algorithm has determined the decision boundary for the category you want to analyze, you only have to obtain the representations of all of the texts you would like to classify and check what side of the boundary those representations fall into.

DATE:-_____ PGNO:-__

PROGRAM:-

```
import numpy as np
from sklearn.svm import SVC
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
training_data = [
    ('This is an positive example', 'pos'),
    ('This is an negative example', 'neg'),
    ('This is an average example', 'neutral')
]
X_train = [x[0] for x in training_data]
y_train = [x[1] for x in training_data]
text_clf = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('clf', SVC(kernel='linear'))
])
text_clf.fit(X_train, y_train)
test_data = "This is an positive example"
predicted = text_clf.predict([test_data])[0]
```

```
print(predicted)
```

RESULT:-

The program is executed successfully without any errors.

10. Convert text to vectors (using term frequency) and apply cosine similarity to provide closeness among two text.

AIM:- Convert text to vectors (using term frequency) and apply cosine similarity to provide closeness among two text.

DESCRIPTION:-

In this example, we use the CountVectorizer class from the scikit-learn library to convert the text into numerical vectors using term frequency. The fit_transform method is used to fit the vectorizer to the text data and then transform the text into numerical vectors.

Next, we use the cosine_similarity function from the scikit-learn library to measure the cosine similarity between the two text vectors. The cosine similarity ranges from -1 to 1, where 1 means the texts are exactly similar and -1 means they are completely dissimilar.

Finally, we print out the similarity score. In this case, a score closer to 1 means that the two texts are more similar, and a score closer to -1 means that they are more dissimilar.

DATE:-_____ PGNO:-__

PROGRAM:-

```
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.metrics.pairwise import cosine_similarity

text1 = "This is an example of text"

text2 = "This is an example of a different text"

corpus = [text1, text2]

vectorizer = CountVectorizer().fit_transform(corpus)

text1_vector = vectorizer[0].toarray().reshape(1, -1)

text2_vector = vectorizer[1].toarray().reshape(1, -1)

similarity = cosine_similarity(text1_vector, text2_vector)[0][0]

print("Cosine similarity:", similarity)
```

RESULT:-

The program is executed successfully without any errors.

11. Case study 1:

Identify the sentiment of tweets In this problem, you are provided with tweet data to predict sentiment on electronic products of netizens.

AIM:- Identify the sentiment of tweets In this problem, you are provided with tweet data to predict sentiment on electronic products of netizens.

DESCRIPTION:-

To identify the sentiment of tweets related to electronic products, you can use Natural Language Processing (NLP) techniques to analyze the text data and classify it into different sentiment categories such as positive, negative or neutral.

1. Preprocessing the text data: This involves cleaning the text data by removing stop words, punctuation marks, special characters, and other irrelevant information from the text data.
2. Tokenizing: After preprocessing, the text data can be split into individual words or tokens.
3. Vectorizing: The tokenized words can then be converted into a numerical vector format for analysis by techniques such as Word Embedding, Bag-of-Words or TF-IDF.
4. Sentiment classification: Once the text data is preprocessed and vectorized, a machine learning model such as Naive Bayes, Support Vector Machines or Recurrent Neural Network can be trained on labeled data to classify the tweets into sentiment categories.
5. Evaluation: The trained model can be evaluated on test data to determine its accuracy and performance.

Program:-

```
from textblob import TextBlob

tweets = [
    "I love my new phone, it's amazing!",
    "This laptop is the worst, I regret buying it.",
    "The camera on this tablet is fantastic!",
    "I hate this new gaming console, it's so slow.",
    "The battery life on this e-reader is amazing, I love it."
]

for tweet in tweets:
    analysis = TextBlob(tweet)
    if analysis.sentiment.polarity > 0:
        print("Positive:", tweet)
    elif analysis.sentiment.polarity == 0:
        print("Neutral:", tweet)
    else:
        print("Negative:", tweet)
```

RESULT:-

The program is executed successfully without any errors.

12. Case study 2:

Detect hate speech in tweets. The objective of this task is to detect hate speech in tweets. For the sake of simplicity, we say a tweet contains hate speech if it has a racist or sexist sentiment associated with it. So, the task is to classify racist or sexist tweets from other tweets.

AIM:- Detect hate speech in tweets. The objective of this task is to detect hate speech in tweets. For the sake of simplicity, we say a tweet contains hate speech if it has a racist or sexist sentiment associated with it. So, the task is to classify racist or sexist tweets from other tweets.

DESCRIPTION:-

In this example, we start by loading the tweet data into a pandas DataFrame. Then, we split the data into features (the text of the tweet) and labels (the target class of either racist or sexist).

Next, we divide the data into training and testing sets using the train_test_split function from scikit-learn.

We then convert the text of the tweets into numerical vectors using TF-IDF (term frequency-inverse document frequency). This helps to represent the text data in a way that can be used by machine learning algorithms.

We then train a support vector machine (SVM) classifier on the training data using the SVC class from scikit-learn. Finally, we use the trained classifier to predict the target class for the test data and evaluate the performance of the classifier using accuracy.

This is just a simple example, and in practice, you may want to preprocess the text data and use a larger and more diverse dataset for more accurate results. Additionally, you may want to consider using other machine learning algorithms or advanced techniques, such as deep learning, for more complex hate speech detection tasks.

PROGRAM:-

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn import svm

from sklearn.metrics import accuracy_score

# Load the tweet data into a pandas DataFrame

df = pd.read_csv("tweets.csv")

# Divide the data into features (the tweet text) and labels (the target class)

X = df["text"]

y = df["label"]

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Convert the tweet text into numerical vectors using TF-IDF

vectorizer = TfidfVectorizer()

X_train = vectorizer.fit_transform(X_train)

X_test = vectorizer.transform(X_test)

# Train a support vector machine classifier on the training data
```

```
clf = svm.SVC(kernel='linear', C=1, random_state=0)

clf.fit(X_train, y_train)

# Predict the target class for the test data

y_pred = clf.predict(X_test)

# Evaluate the performance of the classifier using accuracy

accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
```

RESULT:-

The program is executed successfully without any errors.