**DATA SCIENCE NATURAL LANGUAGE PROCESSING LAB**

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

**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)**

**OUTPUT**

**This is a sample text with , URLs () and mentions to remove.**

**(**This function uses regular expressions to remove hash tags, URLs and mentions from the text. The **re.sub** function replaces the matched patterns with an empty string.**)**

**2.** Perform lemmatization and stemming using python library nltk

**Description :** Lemmatization and stemming are two important techniques for text normalization and preprocessing in natural language processing. Here's how you can use the Natural Language Toolkit (nltk) library in Python to perform lemmatization and stemming on a sample text:

**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)**

**output:**

**Lemmatized tokens: ['The', 'boy', 'were', 'running', 'and', 'jumping', 'in', 'the', 'park']**

**Stemmed tokens: ['the', 'boy', 'were', 'run', 'and', 'jump', 'in', 'the', 'park']**

**(**As you can see, the lemmatization process returns the base or dictionary form of the words, while stemming returns the root form by removing the suffixes. Both techniques can be useful for reducing words to their most basic form, making it easier to compare and analyze text data.)

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

(Object standardization, such as replacing social media slangs from a text, can be done using string operations and regular expressions in Python. Here's an example:))

**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)**

**output:**

**Standardized text: Oh My God! that was so cool yesterday 🔥👌**

**(**In this example, we define a dictionary **slangs** to store the social media slangs and their standard form. Then, we use the **re.sub** function to replace the slangs in the text with their standard form. The regular expression pattern **\b** + slang + **\b** is used to match the exact word, ensuring that only the slangs are replaced and not the substrings within other words.)

Top of Form

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

**(**Part-of-speech (POS) tagging is the process of marking each word in a text with its corresponding part of speech, such as noun, verb, adjective, adverb, etc.

Here's an example of performing POS tagging on textual data in Python using the **nltk** library:**)**

**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)**

**output:**

**[('This', 'DT'), ('is', 'VBZ'), ('a', 'DT'), ('sample', 'NN'), ('text', 'NN'), ('for', 'IN'), ('part-of-speech', 'JJ'), ('tagging', 'NN'), ('.', '.')]**

In this example, the function **pos\_tagging** takes a string of text as input and uses the **nltk.word\_tokenize** function to tokenize the text into individual words. The **nltk.pos\_tag** function is then used to perform POS tagging on the tokenized words, and the list of tagged words is returned as the output.

**5.** Implement topic modeling using latent dirichlet allocation (LDA) in python

**(**Latent Dirichlet Allocation (LDA) is a popular topic modeling algorithm used to identify latent topics in a collection of documents. Here is an implementation of LDA using Python's **gensim** library:))

**Program:**

**import gensim**

**from gensim.utils import simple\_preprocess**

**from gensim.parsing.preprocessing import STOPWORDS**

**from nltk.stem import WordNetLemmatizer, SnowballStemmer**

**from nltk.stem.porter import \***

**import numpy as np**

**import nltk**

**nltk.download('wordnet')**

**def lemmatize\_stemming(text):**

**stemmer = SnowballStemmer("english")**

**return stemmer.stem(WordNetLemmatizer().lemmatize(text, pos='v'))**

**def preprocess(text):**

**result = [ ]**

**for token in gensim.utils.simple\_preprocess(text):**

**if token not in gensim.parsing.preprocessing.STOPWORDS and len(token) > 3:**

**result.append(lemmatize\_stemming(token))**

**return result**

**processed\_docs = [ ]**

**for doc in documents:**

**processed\_docs.append(preprocess(doc))**

**dictionary = gensim.corpora.Dictionary(processed\_docs)**

**bow\_corpus = [dictionary.doc2bow(doc) for doc in processed\_docs]**

**num\_topics = 5**

**lda\_model = gensim.models.LdaMulticore(bow\_corpus, num\_topics=num\_topics, id2word=dictionary, passes=2, workers=2)**

**for idx, topic in lda\_model.print\_topics(-1):**

**print("Topic: {} \nWords: {}".format(idx, topic ))**

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.

**6.** Demonstrate term frequency - inverse document frequency (TF - IDF) using python

**(**TF-IDF is a statistical measure used to evaluate the importance of a word in a document with respect to an entire corpus of documents. Here is an implementation of TF-IDF in Python using the **scikit-learn** library:**)**

**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)**

In this implementation, the **TfidfVectorizer** class from the **scikit-learn** library is used to fit a TF-IDF model to the list of documents. The resulting TF-IDF matrix is then converted to a pandas DataFrame and printed.

The output shows the TF-IDF scores for each word in each document. For example, in the first document, the word "blue" has a high TF-IDF score, indicating that it is an important word in that document relative to the other documents in the corpus.

**7.** Demonstrate word embeddings using word2vec

**(**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:**)**

**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"))**

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.

8. Implement text classification using naive bayes classifier and text blob library

**(**The **TextBlob** library in Python provides an easy-to-use implementation of a Naive Bayes classifier for text classification. Here is an example of how you can use it:**)**

**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())**

In this example, we define a list of training data, where each item in the list is a tuple of a string of text and a label indicating the sentiment of the text. We then use the **NaiveBayesClassifier** class from the **textblob** library to train a Naive Bayes classifier on the training data.

Once the classifier is trained, we can use it to classify new text by creating a **TextBlob** object and passing it to the classifier. In this example, we create a **TextBlob** object for a test string of text and then call the **classify** method to get the sentiment label.

This is just a simple example, and in practice you may want to preprocess the text and use a larger training dataset for better results.

**9.** Apply support vector machine for text classification

**(**Support Vector Machines (SVM) can also be used for text classification. Here is an example of how you can use the **scikit-learn** library in Python to implement SVM for text classification:**)**

**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)**

In this example, we first define a list of training data, where each item in the list is a tuple of a string of text and a label indicating the sentiment of the text.

We then use a pipeline in **scikit-learn** to fit a text classification model. The pipeline consists of two steps: first, we use the **TfidfVectorizer** class to extract features from the text, and then we fit an SVM model using the **SVC** class.

Once the model is trained, we can use it to predict the sentiment label of new text by passing the text as input to the pipeline. In this example, we pass a test string of text to the pipeline and print the predicted sentiment label.

This is just a simple example, and in practice you may want to preprocess the text and use a larger training dataset for better results. You may also want to try different hyperparameters and kernels for the SVM model to see which ones work best for your particular task.

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

**(**Here's an example of how you can convert text to vectors using term frequency and apply cosine similarity to measure the similarity between two texts in Python:**)**

**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)**

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.

11. **case study**

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

(Here's an example of how you can use the **TextBlob** library in Python to identify the sentiment of tweets about electronic products:)

**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)**

In this example, we have a list of tweets about electronic products. For each tweet, we use the **TextBlob** library to perform sentiment analysis and obtain the polarity of the tweet. The polarity ranges from -1 to 1, where 1 indicates a positive sentiment, -1 indicates a negative sentiment, and 0 indicates a neutral sentiment.

We then use an if-else statement to categorize each tweet as positive, neutral, or negative based on its polarity score. Finally, we print the sentiment category and the text of each tweet.

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 sentiment analysis techniques, such as machine learning algorithms, for more complex sentiment analysis tasks.

12. **case study**

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.

(Here's an example of how you can use a machine learning algorithm, such as a support vector machine (SVM), to detect hate speech in tweets:)

**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)**

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.