KNN model using TF-IDF approach.

Introduction Text classification is a vital task in machine learning, and one of the most common techniques used is the TF-IDF approach. In combination with the K-nearest neighbors (KNN) algorithm, it provides a powerful approach for text classification. In this report, we discuss the implementation of the TF-IDF KNN classification using Python libraries.

Libraries Used The following libraries were imported to implement the TF-IDF KNN classification in Python:

- Scikit-learn: This library provides various tools for machine learning, including algorithms for classification, regression, and clustering. It also provides tools for data preprocessing, feature extraction, and model evaluation.
- Numpy: This library provides support for large, multi-dimensional arrays and matrices, and mathematical functions to operate on them.

Implementation The following steps were followed to implement the TF-IDF KNN classification using Python libraries:

1. Data Preprocessing: The first step in any machine learning project is to preprocess the data. In this case, we tokenize the text, remove stop words, and perform stemming and lemmatization.
2. Feature Extraction: The next step is to extract features from the text. In this case, we use the TF-IDF approach to convert the text into a vector of features. This is done using the TfidfVectorizer class from the Scikit-learn library. The class takes as input the preprocessed text and outputs the TF-IDF scores for each word.
3. Train/Test Split: After extracting the features, we split the data into training and testing sets. This is done using the train_test_split function from Scikit-learn.
4. KNN Classification: Once the data is split, we use the K-nearest neighbors algorithm to classify the text. This is done using the KNeighborsClassifier class from Scikit-learn. We pass the training data and the number of neighbors as input to the class. We then fit the model to the training data and predict the labels for the testing data.
5. Model Evaluation: Finally, we evaluate the model using various metrics, such as accuracy, precision, recall, and F1 score. These metrics are computed using the classification_report function from Scikit-learn.

Conclusion In conclusion, the TF-IDF KNN classification is a powerful technique for text classification, and its implementation in Python is straightforward using the Scikit-learn library. The approach involves preprocessing the data, extracting features using TF-IDF, training and testing the model using KNN, and evaluating the model using various metrics. With the right parameter tuning, this technique can achieve high accuracy in various text classification tasks.

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.62      0.80      0.70      2106
           1       0.68      0.46      0.55      1931

    accuracy                           0.64      4037
   macro avg       0.65      0.63      0.62      4037
weighted avg       0.64      0.64      0.62      4037
```

KNN model using TF-IDF report.

BERT MODEL

Introduction:

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained natural language processing model that has achieved state-of-the-art performance on a variety of natural language understanding tasks. In this report, we will discuss the preprocessing and encoding stages of the BERT model and the libraries that can be used to implement these stages in a program.

Preprocessing:

Preprocessing is an important step in any machine learning model, and it is especially crucial in NLP. The goal of preprocessing is to transform the raw text into a format that the machine learning model can understand. In the case of BERT, the preprocessing steps include tokenization, sentence segmentation, and special token insertion.

Tokenization:

The tokenization process can be implemented using the Hugging Face Transformers library. This library provides a variety of tokenizers, including the WordPiece tokenizer used by BERT. Here's an example of how to tokenize a sentence using the WordPiece tokenizer:

python

Copy code

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

sentence = "This is a sample sentence."
tokens = tokenizer.tokenize(sentence)

print(tokens)
```

This will output: ['this', 'is', 'a', 'sample', 'sentence', '.']

Sentence Segmentation:

Sentence segmentation can be implemented using the NLTK library. Here's an example of how to segment a paragraph into sentences using the NLTK library:

python

Copy code

```
import nltk

paragraph = "This is a sample paragraph. It contains multiple sentences. The sentences are separated by periods."
```

sentences = nltk.sent_tokenize(paragraph)

print(sentences)

This will output: ['This is a sample paragraph.', 'It contains multiple sentences.', 'The sentences are separated by periods.']

Special Token Insertion:

The Hugging Face Transformers library provides the BertTokenizer class, which automatically inserts special tokens into the input sequence. Here's an example of how to use the BertTokenizer class to preprocess a sentence:

python

Copy code

from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

sentence = "This is a sample sentence."

encoded_input = tokenizer(sentence, padding=True, truncation=True, return_tensors='pt')

print(encoded_input)

This will output: {'input_ids': tensor([[101, 2023, 2003, 1037, 7099, 6251, 1012, 102]]), 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1]])}

Encoding:

Once the text has been preprocessed, it is encoded into numerical representations that can be fed into the machine learning model. BERT uses a transformer-based architecture that is capable of encoding the entire input sequence at once.

Embedding:

The Hugging Face Transformers library provides a BertModel class, which can be used to encode the input sequence into embeddings. Here's an example of how to use the BertModel class to encode an input sequence:

python

Copy code

```
from transformers import BertTokenizer, BertModel

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

sentence = "This is a sample sentence."
encoded_input = tokenizer(sentence, padding=True, truncation=True, return_tensors='pt')

outputs = model(**encoded_input)

embeddings = outputs.last_hidden_state

print(embeddings)
```

This will output a tensor of shape (1, 8, 768), where 1 is the batch size, `

```
==========================================================================================================
Total params: 505,908,226
Trainable params: 1,025
Non-trainable params: 505,907,201
__________________________________________________________________________________________________
None
15137
Epoch 1/2
474/474 [==============================] - 12934s 27s/step - loss: 0.5572 - accuracy: 0.7087
Epoch 2/2
474/474 [==============================] - 82490s 174s/step - loss: 0.4757 - accuracy: 0.7669
158/158 [==============================] - 3944s 25s/step - loss: 0.4524 - accuracy: 0.7866
1/1 [==============================] - 7s 7s/step
[[0.883282  ]
 [0.11233513]
 [0.11178539]
 [0.484107  ]
 [0.1167829 ]]
```

BERT model.