Documentation Practical Exercise 3

Student:

- Manuel Buser

Degree:

- MSc Computer Science

Module:

- Multimedia Retrieval

GitHub Repo for this Project:

https://github.com/manuu1999/Ex3_NLTK-Transformers-TextTasks_MultiMediaRetrieval

A. Exercise Description

In this task, we will utilize NLTK and transformers to perform text analytics. You have the option to create custom Python classes and methods, or just create command blocks in a Jupyter notebook. The course material contains many code snippets that you can use as a starting point:

a)[easy] Use NLTK to determine the language of an input text. Download texts in Italian, German, and English (or any other language) with the same encoding for simplicity. Employ NLTK's stop word lists to identify the text's language based on stop word occurrences (pick the language with most stop words).

Solution Code:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# Download stopwords and punkt tokenizer
nltk.download('stopwords')
nltk.download('punkt')


3 usages
def detect_language(text):
    languages = ['english', 'german', 'italian']
    stopwords_count = {}

    # Tokenize the input text
    tokens = word_tokenize(text.lower())

    for language in languages:
        lang_stopwords = set(stopwords.words(language))
        stopwords_count[language] = sum(1 for word in tokens if word in lang_stopwords)

    detected_language = max(stopwords_count, key=stopwords_count.get)
    return detected_language


# Example usage
text_italian = "Ciao, come stai? Questo è un esempio di testo in italiano."
print("Detected Language (Italian):", detect_language(text_italian))

text_english = "Hello, how are you? This is an example of text in English."
print("Detected Language (English):", detect_language(text_english))

text_german = "Hallo, wie geht es dir? Dies ist ein Beispieltext in Deutsch."
print("Detected Language (German):", detect_language(text_german))
```

Explanation of the code:

detect_language(text):

- **Purpose**: Detects the language of the given text by counting overlaps between the text and NLTK's predefined stopwords for English, German, and Italian.
- **Steps**:
    1. Tokenizes the text into lowercase words using nltk.word_tokenize.
    2. Iterates through the stopword lists for the three languages.
    3. Counts how many tokens (words) from the input appear in each language's stopword list.
    4. Returns the language with the highest count.

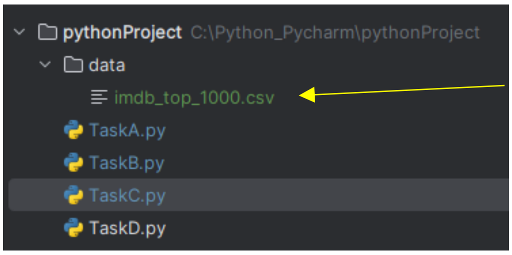
Output to test it:

```
C:\Intelij_MultimediaRetrieval\Ex3_MultimediaRetrieval\Scripts\python.exe C:\Python_Pycharm\pythonProject\TaskA.py
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\buser\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\buser\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
Detected Language (Italian): italian
Detected Language (English): english
Detected Language (German): german

Process finished with exit code 0
```

B. Exercise Description

b)[intermediate] We once again work with the movie dataset from Kaggle.com. This time, we will exclusively use the movie titles and apply sub-word tokenization, which involves fixed-length sequences of 2, 3, or 4 characters within words. You may use special codes to indicate the start or end of words. To simplify the task, utilize unidecode to convert words into non-accented versions. Create a set-of-word representation (using Python's set) for the sub-tokens extracted from the titles and ignore all other fields. For searching, employ the same tokenization and set-of-word approach as for the titles, and establish an appropriate similarity function to match queries with movie titles. https://www.kaggle.com/datasets/harshitshankhdhar/imdb-dataset-of-top-1000-movies-and-tv-shows.




```python
import csv
from unidecode import unidecode
from collections import Counter

# Tokenize text into n-grams
def tokenize_to_ngrams(text, n=3):
    text = unidecode(text.lower())  # Normalize text
    text = f"<{text}>"  # Add start and end markers
    return [text[i:i + n] for i in range(len(text) - n + 1)]

# Create an index of movie titles
def create_title_index(csv_file_path, n=3):
    index = {}
    with open(csv_file_path, mode="r", encoding="utf-8") as file:
        reader = csv.DictReader(file)
        for row in reader:
            title = row["Series_Title"]  # Use the correct column name
            ngrams = set(tokenize_to_ngrams(title, n))
            index[title] = ngrams
    return index

# Search for titles using n-grams
def search_titles(query, index, n=3):
    query_ngrams = set(tokenize_to_ngrams(query, n))
    results = {}
    for title, ngrams in index.items():
        # Calculate Jaccard similarity
        similarity = len(query_ngrams & ngrams) / len(query_ngrams | ngrams)
        results[title] = similarity
    # Sort by similarity
    return sorted(results.items(), key=lambda x: x[1], reverse=True)

# Example usage
csv_file_path = "data/imdb_top_1000.csv"  # Path to the CSV file
index = create_title_index(csv_file_path, n=3)  # Build the index
query = "Shwshank"  # Query with typo
results = search_titles(query, index)

# Print top 10 results
print("Search Results:")
for title, similarity in results[:10]:  # Display top 10 results
    print(f"{title}: {similarity:.2f}")
```

1. **Functions:**
2. **tokenize_to_ngrams(text, n=3)**:
   - **Purpose**: Converts a string into sub-word sequences (n-grams).
   - **Steps**:
     1. Converts the text to lowercase and removes accents using unidecode.
     2. Adds start (<) and end (>) markers to each word.
     3. Extracts all overlapping sequences of length n.
3. **create_title_index(csv_file_path, n=3)**:
   - **Purpose**: Reads the CSV file of movie titles and tokenizes each title into n-grams, creating an index for later search.
   - **Steps**:
     1. Reads the CSV file.
     2. Processes each title using tokenize_to_ngrams and stores the n-grams as a set associated with the title.
4. **search_titles(query, index, n=3)**:
   - **Purpose**: Searches for movie titles similar to the query by comparing n-grams.
   - **Steps**:
     1. Tokenizes the query into n-grams.
     2. Calculates the Jaccard similarity between the query and each title in the index.
     3. Sorts and returns titles ranked by similarity.

Testing:

```
C:\Intelij_MultimediaRetrieval\Ex3_MultimediaRetrieval\Scripts\python.exe C:\Python_Pycharm\pythonProject\TaskB.py
Search Results:
The Shawshank Redemption: 0.15
Forushande: 0.12
Pink: 0.09
Shaun of the Dead: 0.09
Shadow of a Doubt: 0.09
Shrek: 0.08
Shine: 0.08
Sholay: 0.08
Hana-bi: 0.07
Kagemusha: 0.06

Process finished with exit code 0
```

This output from Task B shows the search results for a query based on n-gram similarity using trigrams (n=3). The numbers represent the similarity scores between the query and the movie titles, with higher numbers indicating a better match. For example, "The Shawshank Redemption" has the highest similarity score of 0.15, meaning 15% of the trigrams overlap with the query. The results are sorted by similarity, showing the top 10 closest matches.

C. Exercise Description

c)[intermediate] We perform the same task as in b), but this time we will utilize sentence transformers. Select a suitable model for your hardware and explore various sentence transformer models. Encode the movie titles and proceed with a semantic search for your query:

from sentence_transformers import SentenceTransformer, util
model = SentenceTransformer('all-MiniLM-L6-v2')
a, b = model.encode(title_a), model.encode(title_b)
float(util.dot_score(a, b))

```
from sentence_transformers import SentenceTransformer, util
import csv

# Load the Sentence Transformer model
# Choose a lightweight model if you have limited hardware resources
model = SentenceTransformer('all-MiniLM-L6-v2')

# Create a function to encode movie titles
def encode_movie_titles(csv_file_path):
    titles = []
    embeddings = []

    # Read movie titles from CSV
    with open(csv_file_path, mode="r", encoding="utf-8") as file:
        reader = csv.DictReader(file)
        for row in reader:
            title = row["Series_Title"]  # Ensure this matches the CSV column
            titles.append(title)
            embeddings.append(model.encode(title))  # Encode each title

    return titles, embeddings

# Perform semantic search
def semantic_search(query, titles, embeddings, top_k=10):
    query_embedding = model.encode(query)  # Encode the query
    scores = util.dot_score(query_embedding, embeddings).cpu().numpy()[0]  # Compute similarity scores

    # Pair titles with scores and sort them by score in descending order
    results = sorted(zip(titles, scores), key=lambda x: x[1], reverse=True)

    return results[:top_k]  # Return top-k results

# Example usage
csv_file_path = "data/imdb_top_1000.csv"  # Path to the CSV file
titles, embeddings = encode_movie_titles(csv_file_path)  # Encode all movie titles

query = "dream sharing technology"  # Example query
results = semantic_search(query, titles, embeddings)

# Print top results
print("Search Results:")
for title, score in results:
    print(f"{title}: {score:.2f}")
```

1. **create_title_embeddings(titles)**:
   - **Purpose**: Encodes movie titles into vector representations using a pre-trained SentenceTransformer model.
   - **Steps**:
     1. Loads the all-MiniLM-L6-v2 model.
     2. Encodes all titles from the dataset into vectors.
2. **semantic_search(query, title_embeddings, model, titles)**:
   - **Purpose**: Finds semantically similar titles to a query.
   - **Steps**:
     1. Encodes the query into a vector.
     2. Calculates the cosine similarity between the query and each title's vector.
     3. Sorts titles by similarity scores.
3. **Testing:**

```
Search Results:
The Social Network: 0.39
Network: 0.34
The Nightmare Before Christmas: 0.34
Requiem for a Dream: 0.32
Awakenings: 0.30
Waking Life: 0.28
Night on Earth: 0.28
The Big Sleep: 0.26
Home Alone: 0.25
Sleepers: 0.25

Process finished with exit code 0
```

This output represents the results of a semantic search on movie titles, where each title is given a similarity score. These scores are calculated using sentence-transformer embeddings, which convert the titles and query into numerical vectors. The similarity is determined by comparing these vectors using a dot-product score, where higher values (e.g., **0.39** for *The Social Network*) mean the query and title are more semantically similar. The results are sorted from the highest to the lowest similarity score.

D. Exercise Description

d)[difficult] Let's revisit task a) with language detection. The approach in a) relies on having sufficiently long texts with an adequate number of stop words for accurate language prediction. In the script, we discussed a method based on Naive Bayes, which operates with sub-word sequences. You can reuse the method from b) to generate these sub-sequences, first for learning the Naive Bayes likelihoods, and then for language prediction. To simplify, you can choose an equal prior for all languages and select 3-5 languages that use the same alphabet (eliminating alphabet-related rules). Extract and count all sub-sequences, but only keep the top-n sub-sequences per language (choose n as relatively small, such as 100 or 1000). During prediction, disregard sub-sequences for which we lack likelihoods (to avoid 0-posteriors).

TaskA.py TaskB.py imdb_top_1000.csv TaskC.py TaskD.py ×

```
from collections import Counter
from unidecode import unidecode


# Function to generate n-grams
2 usages  manuu1999 *
def generate_ngrams(text, n=3):
    text = unidecode(text.lower())
    text = f"<{text}>"  # Add start and end markers
    return [text[i:i + n] for i in range(len(text) - n + 1)]


# Function to train Naive Bayes likelihoods for multiple languages
1 usage  manuu1999 *
def train_naive_bayes(language_texts, n=3, top_n=1000):
    likelihoods = {}
    for lang, texts in language_texts.items():
        ngram_counts = Counter()
        for text in texts:
            ngram_counts.update(generate_ngrams(text, n))
        # Select top-n n-grams for the language
        top_ngrams = dict(ngram_counts.most_common(top_n))
        likelihoods[lang] = top_ngrams
    return likelihoods


# Function to predict language based on learned likelihoods
3 usages  manuu1999 *
def predict_language(text, likelihoods, n=3):
    ngrams = generate_ngrams(text, n)
    scores = {lang: 0 for lang in likelihoods}

    for lang, ngram_likelihoods in likelihoods.items():
        for ngram in ngrams:
            if ngram in ngram_likelihoods:
                scores[lang] += ngram_likelihoods[ngram]

    return max(scores, key=scores.get)

```

1) generate_ngrams(text, n=3):

- Purpose: Converts text into trigrams.
- Steps:
- Similar to Task B, it uses unidecode to normalize text and extracts n-grams.

2) train_naive_bayes(language_texts, n=3, top_n=1000):

- Purpose: Trains a Naive Bayes likelihood model using text samples for each language.
- Steps:
- Tokenizes the text samples for each language into trigrams.
- Counts the frequency of each trigram.
- Stores the top n most common trigrams for each language as the likelihoods.

3) predict_language(text, likelihoods, n=3):

- Purpose: Predicts the language of a given text using the trained Naive Bayes model.
- Steps:
- Tokenizes the text into trigrams.
- Checks the likelihoods for each trigram and accumulates scores for each language.
- Returns the language with the highest score.

```

# Example usage
if __name__ == "__main__":
    # Example texts for training
    language_texts = {
        'english': ["Hello world", "This is an example of English text."],
        'german': ["Hallo Welt", "Das ist ein Beispieltext in Deutsch."],
        'italian': ["Ciao mondo", "Questo è un esempio di testo in italiano."]
    }

    # Train Naive Bayes likelihoods
    likelihoods = train_naive_bayes(language_texts, n=3, top_n=100)

    # Input text for prediction
    text = "buongiorno."
    text2 = "hello"
    text3 = "Hallo"

    predicted_language = predict_language(text, likelihoods)
    predicted_language2 = predict_language(text2, likelihoods)
    predicted_language3 = predict_language(text3, likelihoods)

    print("Predicted Language:", predicted_language)
    print("Predicted Language:", predicted_language2)
    print("Predicted Language:", predicted_language3)


```

Now we can predict languages with a short input such as hello or hallo. I tried to input the same in the code for task A and the predictions there were not accurate since it needed more words to predict the language. But since we are using now a stronger predictor model we are able to predict shorter sequences.