

Lab 3 – Text Representation

Due Date: Feb 02, 05:00PM

Objectives

- Learn how to transform raw text into numerical formats suitable for machine learning models using various vectorization methods.

Lab Instructions

Step 0:

1. Create a jupyter notebook file called lab04.ipynb
2. For each step below, create a code cell in your notebook to write your code inside.
3. Create the first cell and install the followings

```
pip install -U spacy  
pip install -U gensim
```

Step 1: One-Hot Encoding

In this exercise, you will manually build a vocabulary and create a function to generate one-hot encoded vectors.

1. Preprocess the Data: Take the following corpus and normalize it (lowercase and remove punctuation).

```
documents = ["Dog bites man.", "Man bites dog.", "Dog eats meat.", "Man  
eats food."]
```

```
processed_docs = [doc.lower().replace(".", "") for doc in documents]  
print(processed_docs)
```

2. Build the Vocabulary: Create a dictionary where each unique word is mapped to a unique integer index.

```
vocab = {}  
# Your code here: Loop through processed_docs to populate vocab  
count = 0  
for doc in processed_docs:  
    for word in doc.split():  
        if word not in vocab:  
            count = count + 1  
            vocab[word] = count
```

```
print(vocab)
```

3. One-Hot Vector Function: Complete the function below to return a list of one-hot vectors for a given string.

```
def get_onehot_vector(somestring):
    onehot_encoded = []
    for word in somestring.split():
        temp = [0] * len(vocab)
        if word in vocab:
            temp[vocab[word] - 1] = 1
        onehot_encoded.append(temp)
    return onehot_encoded

# Test the function
print(get_onehot_vector(processed_docs[1]))
```

Step 2: Bag-of-Words (BoW) with Scikit-Learn

Scikit-learn's CountVectorizer automates the process of creating a document-term matrix.

1. Basic BoW: Use CountVectorizer to transform processed_docs into a frequency-based matrix.

```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()

# Build a BOW representation for the corpus
bow_rep = count_vect.fit_transform(processed_docs)

# Look at the vocabulary mapping
print("Our vocabulary: ", count_vect.vocabulary_)

# See the BOW rep for first 2 documents
print("BoW representation for 'dog bites man': ", bow_rep[0].toarray())
print("BoW representation for 'man bites dog': ", bow_rep[1].toarray())

# Get representation for a new text
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':",
temp.toarray())
```

2. Binary and N-Gram Representations: Initialize a CountVectorizer with binary=True to represent only the presence or absence of a word, and another using ngram_range=(1,3) to include n-grams.

```
# Binary representation
count_vect_bin = CountVectorizer(binary=True)
bow_rep_bin = count_vect_bin.fit_transform(processed_docs)
temp_bin = count_vect_bin.transform(["dog and dog are friends"])
print("Binary Bow for 'dog and dog are friends':", temp_bin.toarray())

# N-gram vectorization (unigrams, bigrams, trigrams)
count_vect_ngram = CountVectorizer(ngram_range=(1,3))
bow_rep_ngram = count_vect_ngram.fit_transform(processed_docs)
print("N-gram vocabulary: ", count_vect_ngram.vocabulary_)
```

Step 3: TF-IDF (Term Frequency-Inverse Document Frequency)

1. Calculate TF-IDF: Use `TfidfVectorizer` to transform the corpus and inspect the IDF values for each word.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer()
bow_rep_tfidf = tfidf.fit_transform(processed_docs)
print("IDF values:", tfidf.idf_)
print("Feature names:", tfidf.get_feature_names_out())

# Transform sample texts
temp_tfidf = tfidf.transform(["dog and man are friends"])
print("Tfidf for 'dog and man are friends':\n", temp_tfidf.toarray())
```

Step 4: Word Embeddings (Word2Vec)

1. Download the model from the following link:
<https://drive.google.com/file/d/0B7XkCwpl5KDYNINUTtlSS21pQmM/edit?resourcekey=0-wjGZdNAUop6WykTtMip30g>
2. Extract the model (.bin) to ./model
3. Using Pre-trained Models Load a pre-trained Word2Vec model and find words most similar to "beautiful".

```
from gensim.models import Word2Vec, KeyedVectors

# Note: Ensure the model path is correct for your environment
pretrainedpath = "./model/GoogleNews-vectors-negative300.bin"
w2v_model = KeyedVectors.load_word2vec_format(pretrainedpath,
binary=True)

print("Vocabulary size:", len(w2v_model.key_to_index))
```

```
print("Similar to 'beautiful':", w2v_model.most_similar("beautiful"))
print("Vector for 'beautiful':", w2v_model['beautiful'])
```

4. Train Your Own Model: Use `gensim.test.utils.common_texts` to train a small 10-dimensional Word2Vec model.

```
from gensim.models import Word2Vec
from gensim.test.utils import common_texts

# Build the model
our_model = Word2Vec(common_texts, vector_size=10, window=5,
min_count=1, workers=4)

# Inspect the model
print("Similar to 'computer':", our_model.wv.most_similar('computer',
topn=5))
print("10-dimensional vector for 'computer':",
our_model.wv['computer'])
```

Step 5: Reflection

1. Why is it important to check if a word exists in the model's vocabulary before attempting to retrieve its vector?

Deliverables

Students must submit one single `lab03.ipynb`.