WEEK1


```
!pip i
import
from n
from n
from n

nltk.d
nltk.d
nltk.d

def wo

    wo

    wo

    st                                                    )

    wo                                                lnum() and word not in stop_words]

    fr

    pr
    pr
    pr
    pr

if __n
  text = "Natural Language Procesing (NLP) is a field of artificial intelligence that focuses on
  word_analysis(text)
```

```
Requirement already satisfied: nltk in /usr/local/lib/python3.11/dist-packages (3.9.1)
Requirement already satisfied: click in /usr/local/lib/python3.11/dist-packages (from nltk)
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-packages (from nltk)
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from nltk)
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
Total words: 22
Unique words: 19
Most coomon words:
[('natural', 2), ('nlp', 2), ('computers', 2), ('language', 1), ('procesing', 1), ('field',
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

WEEK2

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.probability import FreqDist
import random

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('punkt_tab')

def generate_words(text, num_words=10):

    words = word_tokenize(text.lower())
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word.isalnum() and word not in stop_words]
    freq_dist = FreqDist(words)

    generated_words = []
    for _ in range(num_words):
      generated_words.append(random.choice(list(freq_dist.keys())))
    return generated_words

if __name__ == "__main__":
    text = "Natural Language Processing (NLP) is a field of artificial intelligence that focuses
    generated_words = generate_words(text, num_words=5)
    print("Generated Words:", generated_words)
```

```
Generated Words: ['enables', 'understand', 'humans', 'language', 'language']
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
```

Start coding or generate with AI.

WEEK3

```
import nltk
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

# Download the required resource for the PerceptronTagger
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger_eng') # This line downloads the resource
nltk.download('wordnet')
nltk.download('punkt_tab')
nltk.download('stopwords')

def morphological_analysis(text):
  words = word_tokenize(text)
```

```
    stop_words = set(stopwords.words('english'))

    words = [word for word in words if word.isalnum() and word not in stop_words]
    pos_tags = pos_tag(words)

    lemmatizer = WordNetLemmatizer()

    # corrected function call, using get_wordnet_pos to fetch wordnet compatible pos tags
    lemmatized_words = [lemmatizer.lemmatize(word, get_wordnet_pos(pos)) for word, pos in pos_tags
    print("Original words:",words)
    print("Lemmatized words:",lemmatized_words)

# Corrected function name to get_wordnet_pos, and treebank_tag usage
def get_wordnet_pos(treebank_tag):
   if treebank_tag.startswith('J'):
     return 'j'
   elif treebank_tag.startswith('V'):
     return 'v'
   elif treebank_tag.startswith('N'):
     return 'n'
   elif treebank_tag.startswith('R'):
     return 'r'
   else:
     return 'n' # Default to noun if not found

if __name__=="__main__":
  text = input('enter the text')
  morphological_analysis(text)
  for word, pos in pos_tag(word_tokenize(text)):
    print(f"Word: {word}, POS Tag: {pos}, WordNet POS Tag: {get_wordnet_pos(pos)}")
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package averaged_perceptron_tagger_eng to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger_eng.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
enter the textNatural Language Processing (NLP) is a field of artificial intelligence that
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-1-c22f3cc20c1a> in <cell line: 0>()
     43 if __name__=="__main__":
     44   text = input('enter the text')
---> 45   morphological_analysis(text)
     46   for word, pos in pos_tag(word_tokenize(text)):
     47     print(f"Word: {word}, POS Tag: {pos}, WordNet POS Tag: {get_wordnet_pos(pos)}")

                                 4 frames
/usr/local/lib/python3.11/dist-packages/nltk/corpus/reader/wordnet.py in _morphy(self,
form, pos, check_exceptions)
   2077         #    (edited by ekaf) If there are no matches return an empty list.
   2078
-> 2079         exceptions = self._exception_map[pos]
   2080         substitutions = self.MORPHOLOGICAL_SUBSTITUTIONS[pos]
   2081

KeyError: 'j'
```

WEEK4

```
!pip install nltk
import nltk
from nltk import ngrams
from collections import Counter
import re

def clean_text(text):
  cleaned_text = re.sub(r'[^a-zA-Z0-9\s]',' ',text).lower()
  return cleaned_text

def ngram_analysis(text,n):
  cleaned_text = clean_text(text)
  words = cleaned_text.split()
  ngrams_list = list(ngrams(words,n))
  ngrams_count = Counter(ngrams_list)
  return ngrams_count

if __name__ == "__main__":
  text = input("Enter the text")

  n=2
  result=ngram_analysis(text,n)

  print(f"{n}-Gram Analysis:")
```

```
    for ngram, count in result.items():
      print(f"{ngram}:{count} times")
```

WEEK5

```
import nltk
from nltk import ngrams
from collections import Counter
import re

def preprocess_text(text):
  text = re.sub(r'[^\w\s]','',text)
  text = text.lower()
  return text

def generate_bigrams(tokens):
  return list(zip(tokens,tokens[1:]))

def calculate_bigram_probabilities(corpus):
  bigrams = generate_bigrams(corpus)
  bigram_counts = Counter(bigrams)
  vocabulary_size = len(set(corpus))
  bigram_probabilities = {}

  for bigram in bigram_counts:
    bigram_probabilities[bigram] = (bigram_counts[bigram] + 1)/(corpus.count(bigram[0]) + vocabu

  return bigram_probabilities

def bigram_smoothing(text):
  preprocessed_text = preprocess_text(text)
  tokens = preprocessed_text.split()
  bigram_probabilities = calculate_bigram_probabilities(tokens)

  print("Bigram Probabilities:")
  for bigram, probability in bigram_probabilities.items():
    print(f"{bigram}: {probability:.4f}")
if __name__ == "__main__":
  text = input("Enter the text")
  bigram_smoothing(text)
  preprocess_text(text)
```

WEEK6

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dense, Reshape
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import skipgrams

corpus = ["The cat sat on the mat"]
tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
total_words = len(tokenizer.word_index) + 1

vocabulary_size = total_words

skip_grams = [skipgrams(sequence, vocabulary_size, window_size=5) for sequence in tokenizer.text
```

```
pairs, labels = skip_grams[0][0], skip_grams[0][1]

embedding_dim = 100

model = Sequential()
model.add(Embedding(input_dim=total_words, output_dim=embedding_dim, input_length=1))

model.add(Reshape((embedding_dim,)))
model.add(Dense(units=total_words, activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

from tensorflow.keras.utils import to_categorical
labels = to_categorical(labels, num_classes=total_words)
model.fit(np.array(pairs)[:,0], labels, epochs=10, batch_size=32)

word_embeddings = model.get_layer(index=0).get_weights()[0]

for word, token in tokenizer.word_index.items():
  print(f"{word}:{word_embeddings[token]}")
```

Start coding or generate with AI.

Start coding or generate with AI.

WEEK7

```
import nltk
from nltk.tag import hmm
from nltk.corpus import treebank

nltk.download('treebank')
nltk.download('universal_tagset')

data  = treebank.tagged_sents(tagset='universal')
train_data = data[:3500]
test_data = data[3500:]

trainer = hmm.HiddenMarkovModelTrainer()

hmm_tagger = trainer.train_supervised(train_data)

sentence = "The cat sat on the mat".split()

tags = hmm_tagger.tag(sentence)
print("Tagged Sentence ",tags)
```

WEEK8

```
import numpy as np

def viterbi(words, tags, transition_prob, emission_prob, initial_prob):

    num_words = len(words)
    num_tags = len(tags)
```

```
        # Initialize the Viterbi table and backpointer table
        viterbi_table = np.zeros((num_tags, num_words))
    backpointer = np.zeros((num_tags, num_words), dtype=int)

        # Step 1: Initialization
        for i, tag in enumerate(tags):
            viterbi_table[i, 0] = initial_prob.get(tag, 0) * emission_prob.get((words[0], tag), 0)
            backpointer[i, 0] = -1 # No previous tag for the first word

        # Step 2: Recursion
        for t in range(1, num_words): # For each word in the sentence
            for s, tag in enumerate(tags): # For each possible tag
                max_prob = -1
                best_tag = -1
                for s_prev, prev_tag in enumerate(tags): # For each previous tag
                    prob = viterbi_table[s_prev, t - 1] * transition_prob.get((prev_tag, tag), 0) *
                    if prob > max_prob:
                        max_prob = prob
                        best_tag = s_prev
                viterbi_table[s, t] = max_prob
                backpointer[s, t] = best_tag

        # Step 3: Termination
        best_last_tag = np.argmax(viterbi_table[:, -1])
        best_path = [best_last_tag]

        # Step 4: Backtracking
        for t in range(num_words - 1, 0, -1):
            best_last_tag = backpointer[best_last_tag, t]
            best_path.insert(0, best_last_tag)

        # Convert tag indices to tag names
        best_path_tags = [tags[idx] for idx in best_path]
        return best_path_tags


    # Example usage
    if __name__ == "__main__":
        # Define the sentence and possible tags
        words = ["The", "cat", "sat"]
        tags = ["DT", "NN", "VB"]

        # Define probabilities (these would typically come from a trained model)
        transition_prob = {
            ("DT", "NN"): 0.8,
            ("NN", "VB"): 0.6,
            ("VB", "NN"): 0.1,
            ("DT", "VB"): 0.1,
            ("NN", "NN"): 0.2,
            ("VB", "VB"): 0.1,
        }

        emission_prob = {
            ("The", "DT"): 0.9,
            ("cat", "NN"): 0.8,
            ("sat", "VB"): 0.7,
            ("The", "NN"): 0.1,
            ("cat", "VB"): 0.1,
            ("sat", "NN"): 0.1,
        }
```

```
    initial_prob = {
        "DT": 0.6,
        "NN": 0.3,
        "VB": 0.1,
    }

    # Run Viterbi algorithm
    best_tags = viterbi(words, tags, transition_prob, emission_prob, initial_prob)
    print("Most likely POS tags:", best_tags)
```

```
  File "<ipython-input-3-ee89a01dceb5>", line 13
    for i, tag in enumerate(tags):
    ^
IndentationError: unexpected indent
```

WEEK9

```
import spacy
def pos_tagger_spacy(text):
  nlp = spacy.load("en_core_web_sm")
  doc = nlp(text)

  tagged_words = [(token.text,token.pos_) for token in doc]
  return tagged_words

if __name__ == "__main__":
  text = input("Enter the text")
  tagged_result = pos_tagger_spacy(text)

  print("Input Text",text)
  print("\n POS Tagged :")
  for word, pos in tagged_result:
    print(f"{word}: {pos}")
```

```
Enter the textIshan played very well in the yesterday match
Input Text Ishan played very well in the yesterday match

 POS Tagged :
Ishan: PROPN
played: VERB
very: ADV
well: ADV
in: ADP
the: DET
yesterday: NOUN
match: NOUN
```

WEEK10

```
import nltk
from nltk import pos_tag, RegexpParser
from nltk.tokenize import word_tokenize
nltk.download('averaged_perceptron_tagger_eng')
nltk.download('punkt')
nltk.download('punkt_tab')
```

```
def chunker(text):
  words = word_tokenize(text)
  tagged_words = pos_tag(words)
  chunk_grammar =r"""
    NP: {<DT>?<JJ>*<NN>}
    PP: {<IN><NP>}
    VP: {<VB.*><NP|PP>+$}
  """
  chunk_parser = RegexpParser(chunk_grammar)
  chunked_text = chunk_parser.parse(tagged_words)
  return chunked_text

text = "The quick brown fox jumps over the lazy dog"
result = chunker(text)
print(result)
```

WEEK11

WEEK11

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

# Download the necessary resource
nltk.download('averaged_perceptron_tagger_eng')
nltk.download('punkt')
nltk.download('punkt_tab')

text ="The quick brown fox jumps over the lazy dog"
words = word_tokenize(text)
tagged_words = pos_tag(words)

chunk_grammar = r"""
  NP: {<DT>?<JJ>*<NN>}
  PP: {<IN><NP>}
  VP: {<VB.*><NP|PP>+$}
  CLAUSE: {<NP><VP>}
"""

chunk_parser = RegexpParser(chunk_grammar)

chunks = chunk_parser.parse(tagged_words)

print(chunks)
```