Write a program to implement sentence segmentation and word tokenization

```
pip install nltk
```

```
import nltk
nltk.download('punkt')
```

```
import nltk

def tokenize_sentences(text):
    # Use the NLTK tokenizer to segment the text into sentences
    sent_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
    sentences = sent_tokenizer.tokenize(text)
    return sentences

def tokenize_words(sentence):
    # Use the NLTK tokenizer to tokenize the sentence into words
    words = nltk.word_tokenize(sentence)
    return words

def main():
    text = input("Enter text: ")

    # Tokenize the sentences
    sentences = tokenize_sentences(text)

    # Tokenize the words in each sentence
    tokenized_sentences = []
    for sentence in sentences:
        words = tokenize_words(sentence)
        tokenized_sentences.append(words)

    # Print the tokenized sentences and words
    print("Tokenized Sentences:")
    for i, words in enumerate(tokenized_sentences):
        print(f"Sentence {i + 1}: {words}")

if __name__ == '__main__':
    main()
```

Write a program to Implement stemming and lemmatization

```
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

```
nltk.download('wordnet')
```

```
import nltk
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.corpus import wordnet

def stem_words(words):
    # Initialize the PorterStemmer
    stemmer = PorterStemmer()

    # Stem each word in the list
    stemmed_words = [stemmer.stem(word) for word in words]
    return stemmed_words

def lemmatize_words(words):
    # Initialize the WordNetLemmatizer
    lemmatizer = WordNetLemmatizer()

    # Lemmatize each word in the list
    lemmatized_words = [lemmatizer.lemmatize(word, get_wordnet_pos(word)) for word in word
    return lemmatized_words

def get_wordnet_pos(word):
    # Map POS tag to first character lemmatize() accepts
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {
        "J": wordnet.ADJ,
        "N": wordnet.NOUN,
        "V": wordnet.VERB,
        "R": wordnet.ADV
    }
    return tag_dict.get(tag, wordnet.NOUN)

def main():
    text = input("Enter text: ")

    # Tokenize the text into words
    words = nltk.word_tokenize(text)

    # Perform stemming
    stemmed_words = stem_words(words)
    print("Stemmed Words:", stemmed_words)

    # Perform lemmatization
    lemmatized_words = lemmatize_words(words)
    print("Lemmatized Words:", lemmatized_words)

if __name__ == '__main__':
    main()
```

```
    main()
```

Write a program to Implement a tri-gram model

```
import nltk
nltk.download('punkt')
```

```
import nltk
from nltk import trigrams

def build_trigram_model(sentences):
    trigram_model = {}

    # Iterate over each sentence
    for sentence in sentences:
        # Generate trigrams from the sentence
        sentence_trigrams = list(trigrams(sentence, pad_left=True, pad_right=True))

        # Update the trigram model
        for trigram in sentence_trigrams:
            prefix = (trigram[0], trigram[1])
            suffix = trigram[2]
            if prefix in trigram_model:
                trigram_model[prefix].append(suffix)
            else:
                trigram_model[prefix] = [suffix]

    return trigram_model

def generate_text(trigram_model, num_words=10):
    generated_text = []

    # Start the text generation with a random trigram
    prefix = list(trigram_model.keys())[0]
    generated_text.extend(prefix)

    # Generate the rest of the text
    for _ in range(num_words - 2):
        if prefix in trigram_model:
            next_word = nltk.probability.FreqDist(trigram_model[prefix]).max()
            generated_text.append(next_word)
            prefix = (prefix[1], next_word)
        else:
            break

    return generated_text
```

```
def main():
    text = input("Enter text: ")

    # Tokenize the text into sentences
    sentences = nltk.sent_tokenize(text)

    # Tokenize the sentences into words
    tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in sentences]

    # Build the trigram model
    trigram_model = build_trigram_model(tokenized_sentences)

    # Generate text using the trigram model
    generated_text = generate_text(trigram_model)

    # Print the generated text
    print("Generated Text:")
    print(" ".join(generated_text))

if __name__ == '__main__':
    main()
```

Write a program to Implement PoS tagging using HMM & Neural Model

```
import nltk
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import SGDClassifier

# Training data
training_data = [
    ("The cat is sitting on the mat", "DET NOUN VERB VERB DET NOUN"),
    ("I love to eat pizza", "PRON VERB TO VERB NOUN")
]

# HMM POS tagging
def hmm_pos_tagging(sentences, training_data):
    # Preprocess training data
    states = set()
    observations = set()
    transition_counts = {}
    emission_counts = {}

    for sentence, tags in training_data:
        words = nltk.word_tokenize(sentence)
        tags = nltk.word_tokenize(tags)

        for tag in tags:
```

```
            states.add(tag)

        for word, tag in zip(words, tags):
            observations.add(word)

            if tag in emission_counts:
                if word in emission_counts[tag]:
                    emission_counts[tag][word] += 1
                else:
                    emission_counts[tag][word] = 1
            else:
                emission_counts[tag] = {word: 1}

        for i in range(len(tags)-1):
            current_tag = tags[i]
            next_tag = tags[i+1]

            if current_tag in transition_counts:
                if next_tag in transition_counts[current_tag]:
                    transition_counts[current_tag][next_tag] += 1
                else:
                    transition_counts[current_tag][next_tag] = 1
            else:
                transition_counts[current_tag] = {next_tag: 1}

    # Calculate transition probabilities
    transition_probabilities = {}
    for current_tag in transition_counts:
        total_count = sum(transition_counts[current_tag].values())
        transition_probabilities[current_tag] = {}
        for next_tag in transition_counts[current_tag]:
            transition_probabilities[current_tag][next_tag] = transition_counts[current_ta

    # Calculate emission probabilities
    emission_probabilities = {}
    for tag in emission_counts:
        total_count = sum(emission_counts[tag].values())
        emission_probabilities[tag] = {}
        for word in emission_counts[tag]:
            emission_probabilities[tag][word] = emission_counts[tag][word] / total_count

    tagged_sentences = []

    for sentence in sentences:
        words = nltk.word_tokenize(sentence)
        num_words = len(words)

        viterbi = [{}]
        backpointer = {}

        # Initialization step
```

```
        for state in states:
            viterbi[0][state] = transition_probabilities['<START>'].get(state, 0) * \
                                emission_probabilities[state].get(words[0], 0)
            backpointer[state] = '<START>'

        # Recursion step
        for t in range(1, num_words):
            viterbi.append({})
            new_backpointer = {}

            for state in states:
                max_probability = max(viterbi[t-1][prev_state] * \
                                      transition_probabilities[prev_state].get(state, 0) *
                                      emission_probabilities[state].get(words[t], 0) \
                                      for prev_state in states)
                viterbi[t][state] = max_probability
                new_backpointer[state] = max(states, key=lambda prev_state: \
                                             viterbi[t-1][prev_state] * \
                                             transition_probabilities[prev_state].get(stat
                                             emission_probabilities[state].get(words[t], 0

            backpointer = new_backpointer

        # Termination step
        max_probability = max(viterbi[-1][state] * transition_probabilities[state].get('<E
                              for state in states)
        final_state = max(states, key=lambda state: viterbi[-1][state] * \
                                                    transition_probabilities[state].get('<E

        # Backtrace to get the sequence of tags
        tags = [final_state]
        for t in range(num_words-1, 0, -1):
            tags.insert(0, backpointer[tags[0]])

        tagged_sentence = " ".join([f"{word}/{tag}" for word, tag in zip(words, tags)])
        tagged_sentences.append(tagged_sentence)

    return tagged_sentences


# Neural model POS tagging
def neural_pos_tagging(sentences, training_data):
    # Preprocess training data
    X = []
    y = []

    for sentence, tags in training_data:
        words = nltk.word_tokenize(sentence)
        tags = nltk.word_tokenize(tags)

        for word, tag in zip(words, tags):
```

```
            X.append({'word': word})
            y.append(tag)

    # Vectorize features
    vectorizer = DictVectorizer(sparse=False)
    X = vectorizer.fit_transform(X)

    # Split data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

    # Train the classifier
    classifier = SGDClassifier()
    classifier.fit(X_train, y_train)

    # Predict tags for test set
    y_pred = classifier.predict(X_test)

    # Tag new sentences
    tagged_sentences = []

    for sentence in sentences:
        words = nltk.word_tokenize(sentence)
        X_new = vectorizer.transform([{'word': word} for word in words])
        y_pred_new = classifier.predict(X_new)
        tagged_sentence = " ".join([f"{word}/{tag}" for word, tag in zip(words, y_pred_new
        tagged_sentences.append(tagged_sentence)

    return tagged_sentences


def main():
    sentences = [
        "The cat is sleeping on the mat",
        "I like to eat ice cream"
    ]

    # Perform PoS tagging using HMM
    hmm_tagged_sentences = hmm_pos_tagging(sentences, training_data)

    print("HMM Tagging:")
    for sentence, tagged_sentence in zip(sentences, hmm_tagged_sentences):
        print(f"Original Sentence: {sentence}")
        print(f"Tagged Sentence: {tagged_sentence}")
        print()

    # Perform PoS tagging using Neural Model
    neural_tagged_sentences = neural_pos_tagging(sentences, training_data)

    print("Neural Model Tagging:")
    for sentence, tagged_sentence in zip(sentences, neural_tagged_sentences):
        print(f"Original Sentence: {sentence}")
```

```
        print(f"Tagged Sentence: {tagged_sentence}")
        print()


if __name__ == '__main__':
    main()
```

Write a program to Implement syntactic parsing of a given text

```
import nltk

# Text for syntactic parsing
text = "The cat is sitting on the mat"

# Perform syntactic parsing
def syntactic_parsing(text):
    # Download the pre-trained parser
    nltk.download('punkt')
    nltk.download('averaged_perceptron_tagger')
    nltk.download('maxent_ne_chunker')
    nltk.download('words')
    nltk.download('treebank')

    # Tokenize the text into sentences
    sentences = nltk.sent_tokenize(text)

    # Tokenize each sentence into words and part-of-speech tags
    tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in sentences]
    pos_tagged_sentences = [nltk.pos_tag(sentence) for sentence in tokenized_sentences]

    # Perform syntactic parsing using the pre-trained parser (e.g., the Stanford Parser)
    parser = nltk.parse.CoreNLPParser()
    parsed_sentences = [list(parser.parse(sentence)) for sentence in pos_tagged_sentences]

    # Print the parsed trees
    for i, parsed_sentence in enumerate(parsed_sentences):
        print(f"Parsed Tree {i+1}:")
        for tree in parsed_sentence:
            print(tree)

# Perform syntactic parsing on the given text
syntactic_parsing(text)
```

Write a program to Implement dependency parsing of a given text

```
pip install spacy
python -m spacy download en_core_web_sm
```

```
import spacy

# Text for dependency parsing
text = "The cat is sitting on the mat"

# Perform dependency parsing
def dependency_parsing(text):
    # Load the English language model in spaCy
    nlp = spacy.load("en_core_web_sm")

    # Process the text
    doc = nlp(text)

    # Print the dependency parse tree
    for token in doc:
        print(f"Token: {token.text}, Dependency: {token.dep_}, Head Token: {token.head.te

# Perform dependency parsing on the given text
dependency_parsing(text)
```

7 Write a program to Implement Named Entity Recognition (NER)

```
pip install spacy
python -m spacy download en_core_web_sm
```

```
import spacy

# Text for Named Entity Recognition
text = "Apple Inc. was founded in 1976 by Steve Jobs, Steve Wozniak, and Ronald Wayne. It

# Perform Named Entity Recognition
def named_entity_recognition(text):
    # Load the English language model in spaCy
    nlp = spacy.load("en_core_web_sm")

    # Process the text
    doc = nlp(text)

    # Extract named entities
    named_entities = []
    for entity in doc.ents:
        named_entities.append((entity.text, entity.label_))

    # Print the named entities
```

```
    for entity in named_entities:
        print(f"Entity: {entity[0]}, Label: {entity[1]}")

# Perform Named Entity Recognition on the given text
named_entity_recognition(text)
```

Write a program to Implement Text Summarization for the given sample text

```
import nltk
from nltk.tokenize import sent_tokenize
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample text for text summarization
text = """
Text summarization refers to the technique of shortening long pieces of text. The intentic
"""

# Perform text summarization
def text_summarization(text):
    # Tokenize the text into sentences
    sentences = sent_tokenize(text)

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    words = nltk.word_tokenize(text)
    words = [word.lower() for word in words if word.isalnum()]
    words = [word for word in words if word not in stop_words]

    # Calculate TF-IDF scores
    tfidf = TfidfVectorizer()
    word_scores = tfidf.fit_transform(sentences)
    sentence_scores = word_scores.sum(axis=1)

    # Sort sentences by scores in descending order
    ranked_sentences = sorted(((score, sentence) for score, sentence in zip(sentence_score

    # Select the top 3 sentences as the summary
    summary_sentences = ranked_sentences[:3]

    # Join the summary sentences to form the final summary
    summary = ' '.join([sentence for _, sentence in summary_sentences])

    return summary

# Perform text summarization on the given sample text
summary = text_summarization(text)
print("Text Summary:")
```

```
print(summary)
```