```python
#1
import re

text = """Follow us on Twitter: https://twitter.com/ExampleHandle
          and https://twitter.com/Test_Handle123 for updates."""

# Regular expression to extract Twitter handles
handles = re.findall(r"https://twitter\.com/([A-Za-z0-9_]+)", text)

print("Twitter Handles:", handles)

import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

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

# Input text
text = "This is an example of text preprocessing. We are learning Python!"

# Tokenization
tokens = word_tokenize(text)

# Stop Word Removal
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

# Stemming
stemmer = PorterStemmer()
stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]

print("Original Tokens:", tokens)
print("Filtered Tokens:", filtered_tokens)
print("Stemmed Tokens:", stemmed_tokens)

```

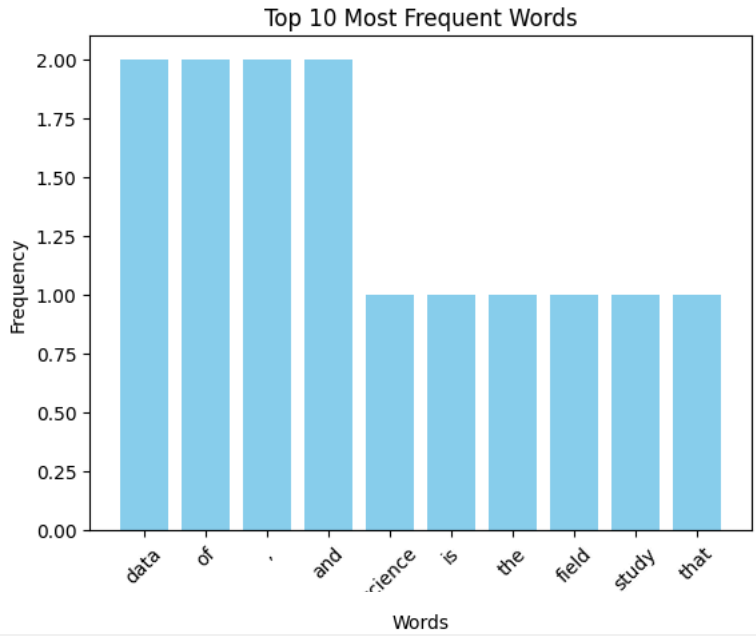
```
Twitter Handles: ['ExampleHandle', 'Test_Handle123']
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
Original Tokens: ['This', 'is', 'an', 'example', 'of', 'text', 'preprocessing', '.', 'We', 'are', 'learning', 'Python', '!']
Filtered Tokens: ['example', 'text', 'preprocessing', '.', 'learning', 'Python', '!']
Stemmed Tokens: ['exampl', 'text', 'preprocess', '.', 'learn', 'python', '!']
```

```
Start coding or generate with AI.
```

```python
#2.1
from collections import Counter
import matplotlib.pyplot as plt
from nltk.tokenize import word_tokenize
import nltk

nltk.download('punkt')

def plot_most_frequent_words(text, top_n=10):
    # Tokenization
    words = word_tokenize(text.lower())

    # Count word frequencies
    word_counts = Counter(words)
    most_common = word_counts.most_common(top_n)

    # Prepare data for plotting
    words, counts = zip(*most_common)

    # Plotting
    plt.bar(words, counts, color='skyblue')
    plt.title(f"Top {top_n} Most Frequent Words")
    plt.xlabel("Words")
    plt.ylabel("Frequency")
    plt.xticks(rotation=45)
    plt.show()
```

```

# Example usage
text = "Data science is the field of study that combines domain expertise, programming skills, and knowledge of mathematics and statistic
plot_most_frequent_words(text)


```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```




```
#2.2
import re
from nltk.tokenize import word_tokenize
import nltk

nltk.download('punkt')

def compare_text_splitting(text):
    # NLTK word_tokenize
    nltk_tokens = word_tokenize(text)

    # Built-in split()
    split_tokens = text.split()

    # Regex splitting
    regex_tokens = re.split(r'\W+', text)  # Splits on non-word characters

    # Print results
    print("NLTK Tokens:", nltk_tokens)
    print("Split() Tokens:", split_tokens)
    print("Regex Tokens:", regex_tokens)

# Example usage
text = "Text preprocessing is crucial! It involves: tokenization, stop word removal, etc."
compare_text_splitting(text)

```

```
NLTK Tokens: ['Text', 'preprocessing', 'is', 'crucial', '!', 'It', 'involves', ':', 'tokenization', ',', 'stop', 'word', 'removal', ',',
Split() Tokens: ['Text', 'preprocessing', 'is', 'crucial!', 'It', 'involves:', 'tokenization,', 'stop', 'word', 'removal,', 'etc.']
Regex Tokens: ['Text', 'preprocessing', 'is', 'crucial', 'It', 'involves', 'tokenization', 'stop', 'word', 'removal', 'etc', '']
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
#3.1
import nltk
from nltk import pos_tag, word_tokenize, RegexpParser
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger_eng')

```

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

def pos_tagging_and_parsing(text):
    from nltk import pos_tag, word_tokenize, RegexpParser

    # Tokenize the text
    tokens = word_tokenize(text)

    # Part-of-Speech tagging
    pos_tags = pos_tag(tokens)

    # Define a simple grammar for parsing
    grammar = "NP: {<DT>?<JJ>*<NN>}"  # Noun Phrase
    parser = RegexpParser(grammar)

    # Parse the tagged words
    parsed_tree = parser.parse(pos_tags)

    # Print the tree structure
    print(parsed_tree)

# Example usage
text = "Elon Musk founded SpaceX and Tesla."
pos_tagging_and_parsing(text)
```

```
(S Elon/NNP Musk/NNP founded/VBD SpaceX/NNP and/CC Tesla/NNP ./.)
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab 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 averaged_perceptron_tagger_eng to
[nltk_data]     /root/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger_eng is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

```
#3.2
import re

def extract_personal_info(text):
    info = {}

    # Extract name
    name_match = re.search(r"Born ([\w\s]+)", text)
    info['Name'] = name_match.group(1) if name_match else None

    # Extract age
    age_match = re.search(r"age (\d+)", text)
    info['Age'] = age_match.group(1) if age_match else None

    # Extract date of birth
    dob_match = re.search(r"Born [\w\s]+\n([\w\s,]+)\n", text)
    info['Date of Birth'] = dob_match.group(1) if dob_match else None

    # Extract education
    education_match = re.search(r"Education ([\w\s,()]+)", text)
    info['Education'] = education_match.group(1) if education_match else None

    # Extract place of birth
    place_match = re.search(r"Born [\w\s]+\n[\w\s,]+\n([\w\s,]+)\n", text)
    info['Place of Birth'] = place_match.group(1) if place_match else None

    return info

# Example usage
text = ''' Born Elon Reeve Musk
June 28, 1971 (age 50)
Pretoria, Transvaal, South Africa Citizenship
South Africa
```

```
Education University of Pennsylvania (BS, BA)
Title Founder, CEO and Chief Engineer of SpaceX
CEO and product architect of Tesla, Inc.
Founder of The Boring Company and X.com (now part of PayPal)
Co-founder of Neuralink, OpenAI, and Zip2
Spouse(s) Justine Wilson (m. 2000; div. 2008) '''

info = extract_personal_info(text)
print(info)
```

```
{'Name': 'Elon Reeve Musk\nJune 28', 'Age': '50', 'Date of Birth': None, 'Education': 'University of Pennsylvania (BS, BA)\nTitle Founde
```

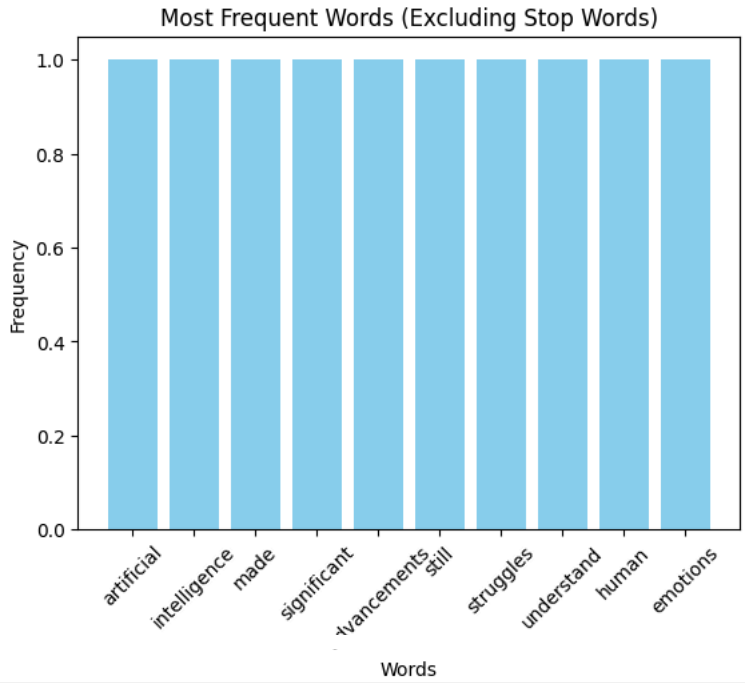
```
#4.1
import nltk
from nltk.util import ngrams
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

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

def ngram_model(text, n=2):
    # Tokenize the text
    tokens = word_tokenize(text.lower())

    # Remove stop words and punctuation
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word.isalnum() and word not in stop_words]

    # Generate N-grams
    n_grams = list(ngrams(filtered_tokens, n))
    return n_grams

# Example usage
text = "Artificial intelligence has made significant advancements, but it still struggles to understand human emotions and context, limit
n_grams = ngram_model(text, n=3)  # Generate trigrams
print(n_grams)
```

```
[('artificial', 'intelligence', 'made'), ('intelligence', 'made', 'significant'), ('made', 'significant', 'advancements'), ('significant
[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!
```

```
#4.2
import nltk
import matplotlib.pyplot as plt
from collections import Counter
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

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

def plot_most_frequent_words(text):
    # Tokenize the text
    tokens = word_tokenize(text.lower())

    # Remove stop words and punctuation
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word.isalnum() and word not in stop_words]

    # Count word frequencies
    word_counts = Counter(filtered_tokens)

    # Get the most common words
    most_common = word_counts.most_common(10)

    # Prepare data for plotting
    words, counts = zip(*most_common)

    # Plotting
    plt.bar(words, counts, color='skyblue')
```

```
    plt.title("Most Frequent Words (Excluding Stop Words)")
    plt.xlabel("Words")
    plt.ylabel("Frequency")
    plt.xticks(rotation=45)
    plt.show()

# Example usage
text = "Artificial intelligence has made significant advancements, but it still struggles to understand human emotions and context, limit
plot_most_frequent_words(text)

```

```
[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!
```




```
#5.1
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
from nltk import pos_tag

nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

def lemmatize_based_on_pos(text):
    # Tokenize the text
    tokens = word_tokenize(text)

    # POS tagging
    pos_tags = pos_tag(tokens)

    # Create lemmatizer
    lemmatizer = WordNetLemmatizer()

    # Lemmatize based on POS
    lemmatized_words = []
    for word, tag in pos_tags:
        if tag.startswith('NN'):  # Noun
            lemmatized_word = lemmatizer.lemmatize(word, wordnet.NOUN)
        elif tag.startswith('VB'):  # Verb
            lemmatized_word = lemmatizer.lemmatize(word, wordnet.VERB)
        elif tag.startswith('JJ'):  # Adjective
            lemmatized_word = lemmatizer.lemmatize(word, wordnet.ADJ)
        else:
            lemmatized_word = word  # No lemmatization for other POS
        lemmatized_words.append(lemmatized_word)
```

```
    return lemmatized_words

# Example usage
text = "The cats are running faster than the dogs."
lemmatized = lemmatize_based_on_pos(text)
print(lemmatized)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
['The', 'cat', 'be', 'run', 'faster', 'than', 'the', 'dog', '.']
```

```
#5.2
import nltk

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

def pos_tagging_with_perceptron(text):
    # Tokenize the text
    tokens = nltk.word_tokenize(text)

    # POS tagging using Averaged Perceptron Tagger
    pos_tags = nltk.pos_tag(tokens)

    return pos_tags

# Example usage
text = "The quick brown fox jumps over the lazy dog."
pos_tags = pos_tagging_with_perceptron(text)
print(pos_tags)
```

```
[('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'), ('lazy', 'JJ'), ('dog'
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

```
#6
import nltk
from nltk.stem import PorterStemmer, LancasterStemmer, SnowballStemmer
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

nltk.download('punkt')
nltk.download('wordnet')

def analyze_stemmers_and_lemmatizer(text):
    # Tokenize the text
    tokens = word_tokenize(text)

    # Stemmers
    porter = PorterStemmer()
    lancaster = LancasterStemmer()
    snowball = SnowballStemmer("english")

    # Lemmatizer
    lemmatizer = WordNetLemmatizer()

    # Apply stemmers
    porter_stemmed = [porter.stem(word) for word in tokens]
    lancaster_stemmed = [lancaster.stem(word) for word in tokens]
    snowball_stemmed = [snowball.stem(word) for word in tokens]

    # Apply lemmatizer
    lemmatized = [lemmatizer.lemmatize(word) for word in tokens]

```

```
    # Display results
    print("Original Text:", tokens)
    print("Porter Stemmer:", porter_stemmed)
    print("Lancaster Stemmer:", lancaster_stemmed)
    print("Snowball Stemmer:", snowball_stemmed)
    print("Lemmatizer:", lemmatized)

# Example usage
text = "The quick brown fox jumps over the lazy dogs."
analyze_stemmers_and_lemmatizer(text)


```

```
Original Text: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dogs', '.']
Porter Stemmer: ['the', 'quick', 'brown', 'fox', 'jump', 'over', 'the', 'lazi', 'dog', '.']
Lancaster Stemmer: ['the', 'quick', 'brown', 'fox', 'jump', 'ov', 'the', 'lazy', 'dog', '.']
Snowball Stemmer: ['the', 'quick', 'brown', 'fox', 'jump', 'over', 'the', 'lazi', 'dog', '.']
Lemmatizer: ['The', 'quick', 'brown', 'fox', 'jump', 'over', 'the', 'lazy', 'dog', '.']
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

```
#6.2


```

```
import spacy

# Load the SpaCy model
nlp = spacy.load("en_core_web_sm")

def extract_syntactic_dependencies(text):
    # Process the text with SpaCy
    doc = nlp(text)

    # Extract and display syntactic dependencies
    for token in doc:
        print(f"Word: {token.text}, Lemma: {token.lemma_}, POS: {token.pos_}, Dependency: {token.dep_}, Head: {token.head.text}")

# Example usage
text = "The quick brown fox jumps over the lazy dog."
extract_syntactic_dependencies(text)


```

```
Word: The, Lemma: the, POS: DET, Dependency: det, Head: fox
Word: quick, Lemma: quick, POS: ADJ, Dependency: amod, Head: fox
Word: brown, Lemma: brown, POS: ADJ, Dependency: amod, Head: fox
Word: fox, Lemma: fox, POS: NOUN, Dependency: nsubj, Head: jumps
Word: jumps, Lemma: jump, POS: VERB, Dependency: ROOT, Head: jumps
Word: over, Lemma: over, POS: ADP, Dependency: prep, Head: jumps
Word: the, Lemma: the, POS: DET, Dependency: det, Head: dog
Word: lazy, Lemma: lazy, POS: ADJ, Dependency: amod, Head: dog
Word: dog, Lemma: dog, POS: NOUN, Dependency: pobj, Head: over
Word: ., Lemma: ., POS: PUNCT, Dependency: punct, Head: jumps
```

```
#7
from collections import Counter
from itertools import islice

# Step 1: Create a collection of 3 documents
documents = [
    "The quick brown fox jumps over the lazy dog.",
    "The lazy dog sleeps under the tree.",
    "The tree provides shade to the sleeping dog."
]

# Step 2: Tokenize and find bi-grams
def find_bigrams(text):
    words = text.lower().split()
    return zip(words, islice(words, 1, None))

bigrams = []
for doc in documents:
    bigrams.extend(find_bigrams(doc))

# Step 3: Count bi-grams
bigram_counts = Counter(bigrams)

```

```
# Step 4: Display results
print("Total unique bi-grams:", len(bigram_counts))
print("Top 5 most common bi-grams:")
for bigram, count in bigram_counts.most_common(5):
    print(f"{' '.join(bigram)}: {count}")

```

```
Total unique bi-grams: 20
Top 5 most common bi-grams:
the lazy: 2
the quick: 1
quick brown: 1
brown fox: 1
fox jumps: 1
```

```
#8
from collections import Counter, defaultdict
from itertools import islice

# Step 1: Create a collection of documents
documents = [
    "The quick brown fox jumps over the lazy dog.",
    "The lazy dog sleeps under the tree.",
    "The tree provides shade to the sleeping dog."
]

# Step 2: Tokenize documents and find unigrams and bigrams
def tokenize(text):
    return text.lower().split()

def find_bigrams(words):
    return list(zip(words, islice(words, 1, None)))

unigram_counts = Counter()
bigram_counts = Counter()

for doc in documents:
    words = tokenize(doc)
    unigram_counts.update(words)
    bigram_counts.update(find_bigrams(words))

# Step 3: Calculate bigram probabilities with Laplace smoothing
def bigram_probability(bigram, vocab_size):
    word1, word2 = bigram
    return (bigram_counts[bigram] + 1) / (unigram_counts[word1] + vocab_size)

# Step 4: Estimate probabilities for a test bigram
test_bigrams = [("the", "lazy"), ("lazy", "cat")]
vocab_size = len(unigram_counts)

print("Bigram probabilities with Laplace smoothing:")
for bigram in test_bigrams:
    prob = bigram_probability(bigram, vocab_size)
    print(f"P({bigram[1]}|{bigram[0]}) = {prob:.4f}")

```

```
Bigram probabilities with Laplace smoothing:
P(lazy|the) = 0.1304
P(cat|lazy) = 0.0526
```

```
#9
from collections import defaultdict

# Step 1: Define the grammar in Chomsky Normal Form (CNF)
grammar = {
    "S": ["AB", "BC"],
    "A": ["BA", "a"],
    "B": ["CC", "b"],
    "C": ["AB", "a"]
}

# Step 2: Cocke-Younger-Kasami (CYK) Algorithm
def cyk_algorithm(string, grammar):
    n = len(string)
    table = [[set() for _ in range(n)] for _ in range(n)]

    # Base case: Fill diagonal with terminals
```

```
    for i, char in enumerate(string):
        for lhs, rhs_list in grammar.items():
            if char in rhs_list:
                table[i][i].add(lhs)

    # Recursive case: Fill table for substrings of length 2 to n
    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l - 1
            for k in range(i, j):
                for lhs, rhs_list in grammar.items():
                    for rhs in rhs_list:
                        if len(rhs) == 2 and rhs[0] in table[i][k] and rhs[1] in table[k + 1][j]:
                            table[i][j].add(lhs)

    # Check if start symbol 'S' is in the top-right cell
    return "S" in table[0][n - 1]

# Step 3: Test the CYK Algorithm
string = "baab"
result = cyk_algorithm(string, grammar)

print(f"The string '{string}' belongs to the grammar language: {result}")

```

```
The string 'baab' belongs to the grammar language: False
```

```
#10
def min_edit_distance(source, target):
    m, n = len(source), len(target)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize base cases
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    # Fill the DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if source[i - 1] == target[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]  # No cost if characters match
            else:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],      # Deletion
                    dp[i][j - 1],      # Insertion
                    dp[i - 1][j - 1]   # Substitution
                )

    return dp[m][n]

# Example usage
source1, target1 = "intention", "execution"
source2, target2 = "Piece", "Peace"

print(f"Minimum edit distance between '{source1}' and '{target1}': {min_edit_distance(source1, target1)}")
print(f"Minimum edit distance between '{source2}' and '{target2}': {min_edit_distance(source2, target2)}")

```

```
Minimum edit distance between 'intention' and 'execution': 5
Minimum edit distance between 'Piece' and 'Peace': 2
```

```
#11
import nltk
from nltk import pos_tag, word_tokenize, RegexpParser
from nltk.sentiment import SentimentIntensityAnalyzer


# Ensure necessary NLTK resources are downloaded
nltk.download('punkt_tab')
nltk.download('averaged_perceptron_tagger_eng')
nltk.download('vader_lexicon')

# Function to extract nouns and adjectives and calculate sentiment score
def analyze_sentence(sentence):
```

```
    # Tokenize and POS tag
    tokens = word_tokenize(sentence)
    pos_tags = pos_tag(tokens)

    # Chunking grammar to extract noun and adjective phrases
    grammar = """NP: {<JJ>*<NN>}"""
    chunk_parser = RegexpParser(grammar)
    chunks = chunk_parser.parse(pos_tags)

    # Extract nouns and adjectives
    nouns = [word for word, pos in pos_tags if pos.startswith('NN')]
    adjectives = [word for word, pos in pos_tags if pos.startswith('JJ')]

    # Calculate sentiment score of adjectives
    sia = SentimentIntensityAnalyzer()
    sentiment_score = sum(sia.polarity_scores(adj)['compound'] for adj in adjectives)

    return nouns, adjectives, sentiment_score

# Example usage
sentence = "The beautiful garden has vibrant flowers and a serene atmosphere."
nouns, adjectives, sentiment_score = analyze_sentence(sentence)

print("Nouns:", nouns)
print("Adjectives:", adjectives)
print("Overall Sentiment Score of Adjectives:", sentiment_score)
```

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
[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 vader_lexicon to /root/nltk_data...
[nltk_data]   Package vader_lexicon is already up-to-date!
Nouns: ['garden', 'flowers', 'serene']
Adjectives: ['beautiful', 'vibrant']
Overall Sentiment Score of Adjectives: 1.1261
```

```
#12
import pandas as pd
from nltk.sentiment import SentimentIntensityAnalyzer
import nltk

# Ensure necessary NLTK resources are downloaded
nltk.download('vader_lexicon')

# Load the dataset
def load_data(file_path):
    return pd.read_csv(file_path)

# Conduct sentiment analysis
def sentiment_analysis(data):
    sia = SentimentIntensityAnalyzer()
    data['sentiment_score'] = data['tweet'].apply(lambda x: sia.polarity_scores(x)['compound'])
    data['sentiment'] = data['sentiment_score'].apply(lambda x: 'positive' if x > 0 else ('negative' if x < 0 else 'neutral'))
    return data

# Main function
if __name__ == "__main__":
    file_path = "vaccination_tweets.csv"  # Replace with the actual path to your dataset
    data = load_data(file_path)
    analyzed_data = sentiment_analysis(data)

    # Display results
    print(analyzed_data[['tweet', 'sentiment', 'sentiment_score']].head())
```

```
#13
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, accuracy_score

# Load the dataset
def load_data(file_path):
```

```
    data = pd.read_csv(file_path, encoding='latin-1')
    data = data.rename(columns={"v1": "label", "v2": "message"})
    data = data[["label", "message"]]
    data['label'] = data['label'].map({"ham": 0, "spam": 1})
    return data

# Train a spam detection model
def train_model(data):
    X_train, X_test, y_train, y_test = train_test_split(data['message'], data['label'], test_size=0.2, random_state=42)
    vectorizer = CountVectorizer()
    X_train_vec = vectorizer.fit_transform(X_train)
    X_test_vec = vectorizer.transform(X_test)

    model = MultinomialNB()
    model.fit(X_train_vec, y_train)
    predictions = model.predict(X_test_vec)

    print("Accuracy:", accuracy_score(y_test, predictions))
    print("Classification Report:\n", classification_report(y_test, predictions))

# Main function
if __name__ == "__main__":
    file_path = "spam.csv"  # Replace with the actual path to your dataset
    data = load_data(file_path)
    train_model(data)

```