5.Given the following short movie reviews, each labeled with a genre, either comedy or action:

- **fun, couple, love, love comedy**
- **fast, furious, shoot action**
- **couple, fly, fast, fun, fun comedy**
- **furious, shoot, shoot, fun action**
- **fly, fast, shoot, love action and**

A new document D: fast, couple, shoot, fly
Compute the most likely class for D. Assume a Naive Bayes classifier and use add-1 smoothing for the likelihoods.

Program:

```
from collections import defaultdict
import math

def train_naive_bayes(data):
    class_counts = defaultdict(int)
    word_counts = defaultdict(lambda: defaultdict(int))
    vocab = set()

    # Count occurrences
    for words, label in data:
        class_counts[label] += 1
        for word in words:
            word_counts[label][word] += 1
            vocab.add(word)

    return class_counts, word_counts, vocab

def calculate_probabilities(class_counts, word_counts, vocab, text, alpha=1):
    total_reviews = sum(class_counts.values())
    probabilities = {}

    for label in class_counts:
        # Prior probability: P(Class)
        prob = math.log(class_counts[label] / total_reviews)
        total_words = sum(word_counts[label].values())
        vocab_size = len(vocab)

        # Compute likelihood with add-1 smoothing: P(w|Class)
        for word in text:
            word_freq = word_counts[label][word] + alpha
            prob += math.log(word_freq / (total_words + vocab_size * alpha))

        probabilities[label] = prob

    return probabilities

def classify(class_counts, word_counts, vocab, text):
    probabilities = calculate_probabilities(class_counts, word_counts, vocab, text)
```

```
    return max(probabilities, key=probabilities.get)

# Training Data
reviews = [
    (['fun', 'couple', 'love', 'love'], 'Comedy'),
    (['fast', 'furious', 'shoot'], 'Action'),
    (['couple', 'fly', 'fast', 'fun', 'fun'], 'Comedy'),
    (['furious', 'shoot', 'shoot', 'fun'], 'Action'),
    (['fly', 'fast', 'shoot', 'love'], 'Action')
]

# Train Naive Bayes Classifier
class_counts, word_counts, vocab = train_naive_bayes(reviews)

# New document
D = ['fast', 'couple', 'shoot', 'fly']

# Classify new document
predicted_class = classify(class_counts, word_counts, vocab, D)
print(predicted_class)
```

Output:

Action

3. Investigate the Minimum Edit Distance (MED) algorithm and its application in string comparison and the goal is to understand how the algorithm efficiently computes the minimum number of edit operations required to transform one string into another. ● Test the algorithm on strings with different type of variations (e.g., typos, substitutions, insertions, deletions)
● Evaluate its adaptability to different types of input variations

Program:

```
def min_edit_distance(str1, str2):
    m = len(str1)
    n = len(str2)

    # Create a DP table to store results of subproblems
    dp = [[0] * (n + 1) for _ in range(m + 1)]

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

    # Fill the DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]  # No operation needed
            else:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],  # Deletion
                    dp[i][j - 1],  # Insertion
                    dp[i - 1][j - 1]  # Substitution
                )

    # The final result is in dp[m][n]
    return dp[m][n]

# Test cases
test_cases = [
    ("kitten", "sitting"),  # Substitutions and insertions
    ("intention", "execution"),  # Substitutions and deletions
    ("flaw", "lawn"),  # Substitutions
    ("apple", "aple"),  # Deletion
    ("book", "books"),  # Insertion
    ("abc", "def"),  # All substitutions
    ("", "abc"),  # All insertions
    ("abc", "")  # All deletions
]

# Evaluate MED for each test case
for str1, str2 in test_cases:
```

```
distance = min_edit_distance(str1, str2)
print(f"MED between '{str1}' and '{str2}': {distance}")
```

Output:
MED between 'kitten' and 'sitting': 3
MED between 'intention' and 'execution': 5
MED between 'flaw' and 'lawn': 2
MED between 'apple' and 'aple': 1
MED between 'book' and 'books': 1
MED between 'abc' and 'def': 3
MED between '' and 'abc': 3
MED between 'abc' and '': 3

Explanation:
The **Minimum Edit Distance (MED)** algorithm is a dynamic programming approach used to measure the similarity between two strings. It calculates the minimum number of operations required to transform one string into another.

1. **Substitutions and Insertions**:
   - "kitten" → "sitting": Replace 'k' with 's', replace 'e' with 'i', and insert 'g'.
   - MED = 3.
2. **Substitutions and Deletions**:
   - "intention" → "execution": Replace 'i' with 'e', replace 'n' with 'x', delete 'n'.
   - MED = 5.
3. **Substitutions**:
   - "flaw" → "lawn": Replace 'f' with 'l', replace 'w' with 'n'.
   - MED = 2.
4. **Deletion**:
   - "apple" → "aple": Delete 'p'.
   - MED = 1.
5. **Insertion**:
   - "book" → "books": Insert 's'.
   - MED = 1.
6. **All Substitutions**:
   - "abc" → "def": Replace all characters.
   - MED = 3.
7. **All Insertions**:
   - "" → "abc": Insert all characters.
   - MED = 3.
8. **All Deletions**:
   - "abc" → "": Delete all characters.
   - MED = 3.

1. **Write a program to implement top-down and bottom-up parser using appropriate context free grammar.**

Program:

```
import nltk
```

```
from nltk import CFG

# Define a simple Context-Free Grammar (CFG)
grammar = CFG.fromstring("""
    S -> NP VP
    NP -> Det N | N
    VP -> V NP | V
    Det -> 'the' | 'a'
    N -> 'cat' | 'dog'
    V -> 'chased' | 'barked'
""")

# Create Top-Down (Recursive Descent) and Bottom-Up (Chart) parsers
top_down_parser = nltk.RecursiveDescentParser(grammar)
bottom_up_parser = nltk.ChartParser(grammar)

# Input sentence
sentence = "the cat chased a dog".split()

# Top-Down Parsing
print("Top-Down Parsing Results:")
for tree in top_down_parser.parse(sentence):
    print(tree)

# Bottom-Up Parsing
print("\nBottom-Up Parsing Results:")
for tree in bottom_up_parser.parse(sentence):
    print(tree)
```

Output:

```
Top-Down Parsing Results:
(S (NP (Det the) (N cat)) (VP (V chased) (NP (Det a) (N dog))))

Bottom-Up Parsing Results:
(S (NP (Det the) (N cat)) (VP (V chased) (NP (Det a) (N dog))))
```

1. **Write a Python program for the following preprocessing of text in NLP:**
   - **Tokenization**
   - **Filtration**
   - **Script Validation**
   - **Stop Word Removal**
   - **Stemming**

Program:

```
import nltk
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('stopwords')

def preprocess_text(text):
     text = text.replace('\u00A0', ' ')
    # Step 1: Tokenization
    tokens = word_tokenize(text)
    print("Tokens:", tokens)

    # Step 2: Filtration (remove special characters, numbers, etc.)
    filtered_tokens = [word for word in tokens if re.match(r'^[a-zA-Z]+$', word)]
    print("Filtered Tokens:", filtered_tokens)

    # Step 3: Script Validation (ensure all tokens are in English script)
    # Assuming the text is already in English, no further action is needed.
    # If not, you can use a language detection library like `langdetect`.

    # Step 4: Stop Word Removal
    stop_words = set(stopwords.words('english'))
    tokens_without_stopwords = [word for word in filtered_tokens if word.lower() not
in stop_words]
    print("Tokens without Stopwords:", tokens_without_stopwords)

    # Step 5: Stemming
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(word) for word in tokens_without_stopwords]
    print("Stemmed Tokens:", stemmed_tokens)

    return stemmed_tokens

# Example Usage
text = "This is an example text! It includes different words, numbers like 123, and
punctuation."
processed_text = preprocess_text(text)
print("Processed Tokens:", processed_text)
```

Output:
Tokens: ['This', 'is', 'an', 'example', 'text', '!', 'It', 'includes', 'different', 'words', ',', 'numbers', 'like', '123', ',', 'and', 'punctuation', '.']
Filtered Tokens: ['This', 'is', 'an', 'example', 'text', 'It', 'includes', 'different', 'words', 'numbers', 'like', 'and', 'punctuation']
Tokens without Stopwords: ['example', 'text', 'includes', 'different', 'words', 'numbers', 'like', 'punctuation']
Stemmed Tokens: ['exampl', 'text', 'includ', 'differ', 'word', 'number', 'like', 'punctuat']
Processed Tokens: ['exampl', 'text', 'includ', 'differ', 'word', 'number', 'like', 'punctuat']

2.Demonstrate the N-gram modeling to analyze and establish the probability distribution across sentences and explore the utilization of unigrams, bigrams, and trigrams in diverse English sentences to illustrate the impact of varying n-gram orders on the calculated probabilities.

- Unigrams (n=1): Single words (e.g., "quick", "brown", "fox").
- Bigrams (n=2): Pairs of consecutive words (e.g., "quick brown", "brown fox").
- Trigrams (n=3): Triplets of consecutive words (e.g., "quick brown fox").

Steps:

1. Tokenize sentences into unigrams, bigrams, and trigrams.
2. Calculate the probability distribution of these N-grams.
3. Analyze how the order of N-grams affects the probabilities.

Program:

```
import nltk
from nltk.util import ngrams
from collections import Counter
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist
# Download necessary NLTK resources
nltk.download('punkt_tab')

# Sample sentences
sentences = [
    "The quick brown fox jumps over the lazy dog.",
    "A quick brown fox jumps over the lazy dog.",
    "The lazy dog is jumped over by the quick brown fox."
]

# Function to generate N-grams and calculate probabilities
def ngram_probability(sentences, n):
    # Tokenize sentences and generate N-grams
    tokens = []
    for sentence in sentences:
        tokens.extend(word_tokenize(sentence.lower()))

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

    # Calculate frequency distribution
    freq_dist = FreqDist(n_grams)

    # Calculate probabilities
    total_ngrams = len(n_grams)
    probabilities = {gram: count / total_ngrams for gram, count in freq_dist.items()}

    return probabilities
```

```
# Unigrams (n=1)
unigram_probs = ngram_probability(sentences, 1)
print("Unigram Probabilities:")
for gram, prob in unigram_probs.items():
    print(f"{gram}: {prob:.4f}")

# Bigrams (n=2)
bigram_probs = ngram_probability(sentences, 2)
print("\nBigram Probabilities:")
for gram, prob in bigram_probs.items():
    print(f"{gram}: {prob:.4f}")


# Trigrams (n=3)
trigram_probs = ngram_probability(sentences, 3)
print("\nTrigram Probabilities:")
for gram, prob in trigram_probs.items():
    print(f"{gram}: {prob:.4f}")
```

Output:

```
Unigram Probabilities:
('the',): 0.1562
('quick',): 0.0938
('brown',): 0.0938
('fox',): 0.0938
('jumps',): 0.0625
('over',): 0.0938
('lazy',): 0.0938
('dog',): 0.0938
('.',): 0.0938
('a',): 0.0312
('is',): 0.0312
('jumped',): 0.0312
('by',): 0.0312


Bigram Probabilities:
('the', 'quick'): 0.0645
('quick', 'brown'): 0.0968
('brown', 'fox'): 0.0968
('fox', 'jumps'): 0.0645
('jumps', 'over'): 0.0645
('over', 'the'): 0.0645
('the', 'lazy'): 0.0968
('lazy', 'dog'): 0.0968
('dog', '.'): 0.0645
('.', 'a'): 0.0323
('a', 'quick'): 0.0323
```

('.', 'the'): 0.0323
('dog', 'is'): 0.0323
('is', 'jumped'): 0.0323
('jumped', 'over'): 0.0323
('over', 'by'): 0.0323
('by', 'the'): 0.0323
('fox', '.'): 0.0323

Trigram Probabilities:
('the', 'quick', 'brown'): 0.0667
('quick', 'brown', 'fox'): 0.1000
('brown', 'fox', 'jumps'): 0.0667
('fox', 'jumps', 'over'): 0.0667
('jumps', 'over', 'the'): 0.0667
('over', 'the', 'lazy'): 0.0667
('the', 'lazy', 'dog'): 0.1000
('lazy', 'dog', '.'): 0.0667
('dog', '.', 'a'): 0.0333
('.', 'a', 'quick'): 0.0333
('a', 'quick', 'brown'): 0.0333
('dog', '.', 'the'): 0.0333
('.', 'the', 'lazy'): 0.0333
('lazy', 'dog', 'is'): 0.0333
('dog', 'is', 'jumped'): 0.0333
('is', 'jumped', 'over'): 0.0333
('jumped', 'over', 'by'): 0.0333
('over', 'by', 'the'): 0.0333
('by', 'the', 'quick'): 0.0333
('brown', 'fox', '.'): 0.0333

6 .Demonstrate the following using appropriate programming tool which illustrates the use of information retrieval in NLP:

- **Study the various Corpus – Brown, Inaugural, Reuters, udhr with various methods like filelds, raw, words, sents, categories**
- **Create and use your own corpora (plaintext, categorical)**
- **Study Conditional frequency distributions**
- **Study of tagged corpora with methods like tagged_sents, tagged_words**
- **Write a program to find the most frequent noun tags**
- **Map Words to Properties Using Python Dictionaries**
- **Study Rule based tagger, Unigram Tagger**

Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.

Program:

```
import nltk
from nltk.corpus import brown, inaugural, reuters, udhr
from nltk import FreqDist, ConditionalFreqDist, pos_tag, word_tokenize
from nltk.tag import DefaultTagger, UnigramTagger
from nltk.corpus import PlaintextCorpusReader

# Download required datasets
nltk.download('brown')
nltk.download('inaugural')
nltk.download('reuters')
nltk.download('udhr')
nltk.download('averaged_perceptron_tagger')
nltk.download('punkt')

# Study Various Corpora
def study_corpus():
    print("Brown Corpus Categories:", brown.categories())
    print("First 100 words of Inaugural Corpus:", inaugural.words()[:100])
    print("First 100 words of Reuters Corpus:", reuters.words()[:100])
    print("First 100 words of UDHR Corpus:", udhr.words('English-Latin1')[:100])

# Create and Use Custom Corpora
corpus_root = 'custom_corpus/'  # Ensure this folder exists with text files
custom_corpus = PlaintextCorpusReader(corpus_root, '.*')

# Study Conditional Frequency Distributions
def study_cfd():
    cfd = ConditionalFreqDist(
        (genre, word)
        for genre in brown.categories()
        for word in brown.words(categories=genre)
    )
    print("Most common words in 'news' category:", cfd['news'].most_common(10))

# Study Tagged Corpora
def study_tagged_corpora():
```

```
    print("First 10 Tagged Sentences from Brown:", brown.tagged_sents()[:10])
    print("First 10 Tagged Words from Brown:", brown.tagged_words()[:10])

# Find Most Frequent Noun Tags
def most_frequent_nouns(text):
    tokens = word_tokenize(text)
    tagged_words = pos_tag(tokens)
    fdist = FreqDist(tag for word, tag in tagged_words if tag.startswith('NN'))
    return fdist.most_common(10)

# Map Words to Properties Using Python Dictionaries
word_properties = {
    'run': {'POS': 'verb', 'meaning': 'move swiftly'},
    'book': {'POS': 'noun', 'meaning': 'collection of pages'}
}

# Study Rule-Based Tagger and Unigram Tagger
def study_taggers():
    default_tagger = DefaultTagger('NN')
    unigram_tagger = UnigramTagger(brown.tagged_sents(categories='news')[:500])
    sample_text = word_tokenize("The quick brown fox jumps over the lazy dog")
    print("Default Tagger Output:", default_tagger.tag(sample_text))
    print("Unigram Tagger Output:", unigram_tagger.tag(sample_text))

# Function to find words from a given text without spaces
def split_text_to_words(text, corpus_words):
    found_words = []
    i = 0
    while i < len(text):
        for j in range(i + 1, len(text) + 1):
            if text[i:j] in corpus_words:
                found_words.append(text[i:j])
                i = j - 1
                break
        i += 1
    return found_words, len(found_words)

# Example Usage
study_corpus()
study_cfd()
study_tagged_corpora()
study_taggers()

text = "runningbookfastcar"
corpus_words = set(brown.words())
found_words, score = split_text_to_words(text, corpus_words)
print("Extracted Words:", found_words)
print("Score:", score)
```

Output:

Brown Corpus Categories: ['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance', 'science_fiction']
First 100 words of Inaugural Corpus: ['Fellow', '-', 'Citizens', 'of', 'the', 'Senate', ...]
First 100 words of Reuters Corpus: ['ASIAN', 'EXPORTERS', 'FEAR', 'DAMAGE', 'FROM', 'U', ...]
First 100 words of UDHR Corpus: ['Universal', 'Declaration', 'of', 'Human', 'Rights', 'Preamble', 'Whereas', 'recognition', 'of', 'the', 'inherent', 'dignity', 'and', 'of', 'the', 'equal', 'and', 'inalienable', 'rights', 'of', 'all', 'members', 'of', 'the', 'human', 'family', 'is', 'the', 'foundation', 'of', 'freedom', ',', 'justice', 'and', 'peace', 'in', 'the', 'world', ',', 'Whereas', 'disregard', 'and', 'contempt', 'for', 'human', 'rights', 'have', 'resulted', 'in', 'barbarous', 'acts', 'which', 'have', 'outraged', 'the', 'conscience', 'of', 'mankind', ',', 'and', 'the', 'advent', 'of', 'a', 'world', 'in', 'which', 'human', 'beings', 'shall', 'enjoy', 'freedom', 'of', 'speech', 'and', 'belief', 'and', 'freedom', 'from', 'fear', 'and', 'want', 'has', 'been', 'proclaimed', 'as', 'the', 'highest', 'aspiration', 'of', 'the', 'common', 'people', ',', 'Whereas', 'it', 'is', 'essential', ',', 'if']
Most common words in 'news' category: [('the', 5580), (',', 5188), ('.', 4030), ('of', 2849), ('and', 2146), ('to', 2116), ('a', 1993), ('in', 1893), ('for', 943), ('The', 806)]
First 10 Tagged Sentences from Brown: [[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-TL'), ('Jury', 'NN-TL'), ('said', 'VBD'), ('Friday', 'NR'), ('an', 'AT'), ('investigation', 'NN'), ('of', 'IN'), ("Atlanta's", 'NP$'), ('recent', 'JJ'), ('primary', 'NN'), ('election', 'NN'), ('produced', 'VBD'), ('``', '``'), ('no', 'AT'), ('evidence', 'NN'), ("''", "''"), ('that', 'CS'), ('any', 'DTI'), ('irregularities', 'NNS'), ('took', 'VBD'), ('place', 'NN'), ('.', '.')], [('The', 'AT'), ('jury', 'NN'), ('further', 'RBR'), ('said', 'VBD'), ('in', 'IN'), ('term-end', 'NN'), ('presentments', 'NNS'), ('that', 'CS'), ('the', 'AT'), ('City', 'NN-TL'), ('Executive', 'JJ-TL'), ('Committee', 'NN-TL'), (',', ','), ('which', 'WDT'), ('had', 'HVD'), ('over-all', 'JJ'), ('charge', 'NN'), ('of', 'IN'), ('the', 'AT'), ('election', 'NN'), (',', ','), ('``', '``'), ('deserves', 'VBZ'), ('the', 'AT'), ('praise', 'NN'), ('and', 'CC'), ('thanks', 'NNS'), ('of', 'IN'), ('the', 'AT'), ('City', 'NN-TL'), ('of', 'IN-TL'), ('Atlanta', 'NP-TL'), ("''", "''"), ('for', 'IN'), ('the', 'AT'), ('manner', 'NN'), ('in', 'IN'), ('which', 'WDT'), ('the', 'AT'), ('election', 'NN'), ('was', 'BEDZ'), ('conducted', 'VBN'), ('.', '.')], [('The', 'AT'), ('September-October', 'NP'), ('term', 'NN'), ('jury', 'NN'), ('had', 'HVD'), ('been', 'BEN'), ('charged', 'VBN'), ('by', 'IN'), ('Fulton', 'NP-TL'), ('Superior', 'JJ-TL'), ('Court', 'NN-TL'), ('Judge', 'NN-TL'), ('Durwood', 'NP'), ('Pye', 'NP'), ('to', 'TO'), ('investigate', 'VB'), ('reports', 'NNS'), ('of', 'IN'), ('possible', 'JJ'), ('``', '``'), ('irregularities', 'NNS'), ("''", "''"), ('in', 'IN'), ('the', 'AT'), ('hard-fought', 'JJ'), ('primary', 'NN'), ('which', 'WDT'), ('was', 'BEDZ'), ('won', 'VBN'), ('by', 'IN'), ('Mayor-nominate', 'NN-TL'), ('Ivan', 'NP'), ('Allen', 'NP'), ('Jr.', 'NP'), ('.', '.')], [('``', '``'), ('Only', 'RB'), ('a', 'AT'), ('relative', 'JJ'), ('handful', 'NN'), ('of', 'IN'), ('such', 'JJ'), ('reports', 'NNS'), ('was', 'BEDZ'), ('received', 'VBN'), ("''", "''"), (',', ','), ('the', 'AT'), ('jury', 'NN'), ('said', 'VBD'), (',', ','), ('``', '``'), ('considering', 'IN'), ('the', 'AT'), ('widespread', 'JJ'), ('interest', 'NN'), ('in', 'IN'), ('the', 'AT'), ('election', 'NN'), (',', ','), ('the', 'AT'), ('number', 'NN'), ('of', 'IN'), ('voters', 'NNS'), ('and', 'CC'), ('the', 'AT'), ('size', 'NN'), ('of', 'IN'), ('this', 'DT'), ('city', 'NN'), ("''", "''"), ('.', '.')], [('The', 'AT'), ('jury', 'NN'), ('said', 'VBD'), ('it', 'PPS'), ('did', 'DOD'), ('find', 'VB'), ('that', 'CS'), ('many', 'AP'), ('of', 'IN'), ("Georgia's", 'NP$'), ('registration', 'NN'), ('and', 'CC'), ('election', 'NN'), ('laws', 'NNS'), ('``', '``'), ('are', 'BER'), ('outmoded', 'JJ'), ('or', 'CC'), ('inadequate', 'JJ'), ('and', 'CC'), ('often', 'RB'), ('ambiguous', 'JJ'), ("''", "''"), ('.', '.')], [('It', 'PPS'), ('recommended', 'VBD'), ('that', 'CS'), ('Fulton', 'NP'), ('legislators', 'NNS'), ('act', 'VB'), ('``', '``'), ('to', 'TO'), ('have', 'HV'), ('these', 'DTS'), ('laws', 'NNS'), ('studied', 'VBN'), ('and', 'CC'), ('revised', 'VBN'), ('to', 'IN'), ('the', 'AT'), ('end', 'NN'), ('of', 'IN'), ('modernizing', 'VBG'), ('and', 'CC'), ('improving', 'VBG'), ('them', 'PPO'), ("''", "''"), ('.', '.')], [('The', 'AT'), ('grand', 'JJ'), ('jury', 'NN'), ('commented',

'VBD'), ('on', 'IN'), ('a', 'AT'), ('number', 'NN'), ('of', 'IN'), ('other', 'AP'), ('topics', 'NNS'), (',', ','), ('among', 'IN'), ('them', 'PPO'), ('the', 'AT'), ('Atlanta', 'NP'), ('and', 'CC'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('purchasing', 'VBG'), ('departments', 'NNS'), ('which', 'WDT'), ('it', 'PPS'), ('said', 'VBD'), ('``', '``'), ('are', 'BER'), ('well', 'QL'), ('operated', 'VBN'), ('and', 'CC'), ('follow', 'VB'), ('generally', 'RB'), ('accepted', 'VBN'), ('practices', 'NNS'), ('which', 'WDT'), ('inure', 'VB'), ('to', 'IN'), ('the', 'AT'), ('best', 'JJT'), ('interest', 'NN'), ('of', 'IN'), ('both', 'ABX'), ('governments', 'NNS'), ("''", "''"), ('.', '.')], [('Merger', 'NN-HL'), ('proposed', 'VBN-HL')], [('However', 'WRB'), (',', ','), ('the', 'AT'), ('jury', 'NN'), ('said', 'VBD'), ('it', 'PPS'), ('believes', 'VBZ'), ('``', '``'), ('these', 'DTS'), ('two', 'CD'), ('offices', 'NNS'), ('should', 'MD'), ('be', 'BE'), ('combined', 'VBN'), ('to', 'TO'), ('achieve', 'VB'), ('greater', 'JJR'), ('efficiency', 'NN'), ('and', 'CC'), ('reduce', 'VB'), ('the', 'AT'), ('cost', 'NN'), ('of', 'IN'), ('administration', 'NN'), ("''", "''"), ('.', '.')], [('The', 'AT'), ('City', 'NN-TL'), ('Purchasing', 'VBG-TL'), ('Department', 'NN-TL'), (',', ','), ('the', 'AT'), ('jury', 'NN'), ('said', 'VBD'), (',', ','), ('``', '``'), ('is', 'BEZ'), ('lacking', 'VBG'), ('in', 'IN'), ('experienced', 'VBN'), ('clerical', 'JJ'), ('personnel', 'NNS'), ('as', 'CS'), ('a', 'AT'), ('result', 'NN'), ('of', 'IN'), ('city', 'NN'), ('personnel', 'NNS'), ('policies', 'NNS'), ("''", "''"), ('.', '.')]]
First 10 Tagged Words from Brown: [('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-TL'), ('Jury', 'NN-TL'), ('said', 'VBD'), ('Friday', 'NR'), ('an', 'AT'), ('investigation', 'NN'), ('of', 'IN')]
Default Tagger Output: [('The', 'NN'), ('quick', 'NN'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'NN'), ('over', 'NN'), ('the', 'NN'), ('lazy', 'NN'), ('dog', 'NN')]
Unigram Tagger Output: [('The', 'AT'), ('quick', None), ('brown', None), ('fox', None), ('jumps', None), ('over', 'IN'), ('the', 'AT'), ('lazy', None), ('dog', None)]
Extracted Words: ['r', 'u', 'n', 'n', 'i', 'n', 'g', 'b', 'o', 'o', 'k', 'f', 'a', 't', 'car']
Score: 15

7 Write a Python program to find synonyms and antonyms of the word "active" using WordNet.

Program:

```
from nltk.corpus import wordnet
import nltk

# Download WordNet dataset
nltk.download('wordnet')
nltk.download('omw-1.4')

def get_synonyms_antonyms(word):
    synonyms = set()
    antonyms = set()

    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            synonyms.add(lemma.name())
            if lemma.antonyms():
                antonyms.add(lemma.antonyms()[0].name())

    return synonyms, antonyms
```

```
# Example Usage
word = "active"
synonyms, antonyms = get_synonyms_antonyms(word)
print("Synonyms:", synonyms)
print("Antonyms:", antonyms)
```

Output:

Synonyms: {'active', 'participating', 'active_voice', 'fighting', 'alive', 'dynamic', 'combat-ready', 'active_agent'}
Antonyms: {'passive_voice', 'stative', 'passive', 'inactive', 'quiet', 'dormant', 'extinct'}

8 Implement the machine translation application of NLP where it needs to train a machine translation model for a language with limited parallel corpora. Investigate and incorporate techniques to improve performance in low-resource scenarios.

Program:

```
import tensorflow as tf
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

# Example small parallel corpus
data = [
    ("hello", "hola"),
    ("how are you", "como estas"),
    ("good morning", "buenos dias"),
    ("thank you", "gracias"),
    ("good night", "buenas noches")
]

# Tokenization
english_texts, spanish_texts = zip(*data)
eng_tokenizer = Tokenizer()
spa_tokenizer = Tokenizer()
eng_tokenizer.fit_on_texts(english_texts)
spa_tokenizer.fit_on_texts(spanish_texts)

# Convert text to sequences
eng_sequences = eng_tokenizer.texts_to_sequences(english_texts)
spa_sequences = spa_tokenizer.texts_to_sequences(spanish_texts)

# Padding
max_length = max(len(seq) for seq in spa_sequences)
eng_sequences = pad_sequences(eng_sequences, maxlen=max_length, padding='post')
spa_sequences = pad_sequences(spa_sequences, maxlen=max_length, padding='post')

# Define model
```

```
embedding_dim = 64
hidden_units = 128

encoder_inputs = tf.keras.Input(shape=(max_length,))
encoder_embedding = Embedding(len(eng_tokenizer.word_index) + 1,
embedding_dim)(encoder_inputs)
encoder_lstm = LSTM(hidden_units, return_state=True)
_, state_h, state_c = encoder_lstm(encoder_embedding)
encoder_states = [state_h, state_c]

decoder_inputs = tf.keras.Input(shape=(max_length,))
decoder_embedding = Embedding(len(spa_tokenizer.word_index) + 1,
embedding_dim)(decoder_inputs)
decoder_lstm = LSTM(hidden_units, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_embedding, initial_state=encoder_states)
decoder_dense = Dense(len(spa_tokenizer.word_index) + 1, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

# Compile model
model = tf.keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Prepare decoder targets
spa_sequences_output = np.array(spa_sequences).reshape((-1, max_length, 1))

# Train model
model.fit([eng_sequences, spa_sequences], spa_sequences_output, epochs=100, verbose=1)

# Translation function
def translate(sentence):
    sequence = eng_tokenizer.texts_to_sequences([sentence])
    sequence = pad_sequences(sequence, maxlen=max_length, padding='post')
    prediction = model.predict([sequence, sequence])
    predicted_words = [spa_tokenizer.index_word.get(np.argmax(word)) for word in
prediction[0]]
    return " ".join([w for w in predicted_words if w])

# Example translation
print("Translation:", translate("hello"))
```

Output:

```
1/1 [==============================] - 1s 802ms/step
Translation: hola
```