Natural Language Processing Assignment

**Correct the Search Query**

**Explanation:** Here is a basic implementation using Python, focusing on spell correction using edit distance and a predefined corpus of words. This code uses zlib for compression and pickle for serialization, suitable for building an offline model.

**Source Code:**

import re import pickle import zlib

from collections import Counter

# Build corpus from a sample dictionary (you can enhance it with more words)

words = """going to china who was the first president of india winner of the match food in america"""

def words\_list(text):

return re.findall(r'\w+', text.lower())

WORDS = Counter(words\_list( words)) # Compression for large wordlist with open('compressed\_dict. pkl', 'wb') as f:

compressed = zlib.compress(pickle.dumps(WORDS)) f.write(compressed)

# Load dictionary in memory def load\_dictionary():

with open('compressed\_dict. pkl', 'rb') as f:

return pickle.loads(zlib.decompress(f.read())) def edit\_distance\_one(word):

letters = 'abcdefghijklmnopqrstuvwxyz'

splits = [(word[:i], word[i:]) for i in range(len(word) + 1)] deletes = [L + R[1:] for L, R in splits if R] transposes = [L + R[1]

+ R[0] + R[2:] for L, R in splits if len(R) > 1] replaces = [L + c + R[1:] for L, R in splits if R for c in letters] inserts = [L + c + R for L, R in splits for c in letters] return set(deletes + transposes + replaces + inserts) def known(words, dictionary): return set(w for w in words if w in dictionary) def candidates(word, dictionary):

return (known([word], dictionary) or known(edit\_distance\_one(word), dictionary) or

[word])

def correct\_word(word, dictionary):

return max(candidates(word, dictionary), key=dictionary.get) def correct\_query(query, dictionary):

return ' '.join(correct\_word(word, dictionary) for word in query.split()) # Main correction function

if name== " main ":

dictionary = load\_dictionary() n = int(input()) queries = [input().strip() for

\_ in range(n)] for query in queries:

print(correct\_query(query, dictionary))

# Deterministic Url and HashTag Segmentation

**Explanation:** This approach aims to find the most likely and meaningful segmentation of the input strings based on the provided dictionary of words and the constraint of selecting the longest valid tokens from the left.

**Source Code:**

import re

# Load words from words.txt into a set with open("words.txt", "r") as file:

dictionary = set(word.strip().lower() for word in file.readlines())

def is\_number(s):

"""Check if the string is a number.""" try:

float(s) return True

except ValueError:

return False

def tokenize(input\_string, dictionary):

"""

Tokenize the input string using the longest match first approach.

Args:

input\_string: The string to be tokenized. dictionary: A set of valid words.

Returns:

A list of tokens from the input string. """

length = len(input\_string) if length == 0:

return []

# dp[i] stores the tokens for the substring starting from index i dp = [None] \* (length + 1)

dp[0] = [] # Base case: empty string has no tokens

for i in range(1, length + 1):

# Consider all possible ending positions for the current substring for j in range(i):

left\_part = input\_string[j:i]

# Check if left part is a valid word or number if (left\_part in dictionary or is\_number(left\_part)) and ( dp[j] is not None

):

# If left part is valid and remaining part has a valid tokenization right\_part\_tokens = dp[j] right\_part\_tokens.append(left\_part)

# Choose the longest valid tokenization

if len(right\_part\_tokens) > len(dp[i]) or dp[i] is None: dp[i] = right\_part\_tokens

# Return the tokenization for the entire string if it exists return dp[length] if dp[length] is not None else [input\_string]

def main():

"""Read input strings, tokenize them, and print the results.""" num\_test\_cases

= int(input())

for \_ in range(num\_test\_cases):

input\_string = input().strip().lower()

# Remove www and extensions for domain names, # for hashtags if input\_string.startswith("[www.")](http://www/): input\_string = input\_string[4:].rsplit(".", 1)[0] elif input\_string.startswith("#"):

input\_string = input\_string[1:]

tokens = tokenize(input\_string, dictionary) print(f"Segmentation for Input: {' '.join(tokens)}")

if name== " main":

main()

# Disambiguation: Mouse vs Mouse

**Explanation:** This code provides a basic framework for classifying the usage of the word "mouse" in a sentence. You can further improve the accuracy by:

* 1. **Expanding the Training Data:** Use a larger and more diverse dataset of sentences.
  2. **Experimenting with Different Classifiers:** Try other machine learning models like Support Vector Machines (SVM) or Random Forests.
  3. **Using Word Embeddings:** Consider using word embeddings like Word2Vec or GloVe to capture semantic relationships between words.

**Source Code:**

import pickle

from sklearn.feature\_extraction.text import CountVectorizer from sklearn.naive\_bayes import MultinomialNB

# Training data (sample corpus) training\_sentences

= [

"The complete mouse reference genome was sequenced in 2002.",

"Tail length varies according to the environmental temperature of the mouse " "during postnatal development.",

"A mouse is an input device.", "Many mice have a pink tail.",

"The mouse pointer on the screen helps in navigation.", "A rodent like a mouse has sharp teeth.",

"The mouse was connected to the computer using a USB port.", "The house was infested with mice.",

"Computer users often prefer a wireless mouse."

]

# Labels corresponding to the training sentences labels = [

"animal",

"animal",

"computer-mouse",

"animal", "computer-mouse", "animal", "computer-mouse", "animal", "computer-mouse"

]

# Vectorize the training sentences vectorizer

= CountVectorizer()

X\_train = vectorizer.fit\_transform(training\_sentences)

# Create and train the Naive Bayes classifier classifier = MultinomialNB() classifier.fit(X\_train, labels)

# Function to predict the type of mouse def predict\_mouse\_type(se ntence):

"""

Predicts whether the 'mouse' in the sentence refers to an animal or a computer mouse.

Args:

sentence: The input sentence.

Returns:

"animal" or "computer-mouse"

|  |  |  |
| --- | --- | --- |
| """ |  | |
| vectorized\_sentence  vectorizer.transform([sentence]) | prediction | =  = |

classifier.predict(vectorized\_sentence)[0] return prediction

# Get number of test cases num\_test\_cases

= int(input())

# Process each test case

for \_ in range(num\_test\_cases):

sentence = input() prediction = predict\_mouse\_type(sentence) print(prediction)

# Optionally, save the trained model for later use with open('mouse\_classifier. pkl', 'wb') as f:

pickle.dump((vectorizer, classifier), f)

# Language Detection

* 1. **Explanation:** This function loads the pre-trained model from a serialized file.
  2. It takes a text snippet as input, normalizes it to ASCII, and converts it into a TF- IDF vector using the loaded vectorizer.
  3. The function then uses the trained classifier to predict the language of the snippet based on the extracted features.

**Source Code:**

import pickle import unicodedata from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.naive\_bayes import MultinomialNB

def normalize\_to\_ascii(text):

"""Remove non-ASCII characters and normalize text."""

return unicodedata.normalize("NFKD", text).encode("ascii", "ignore").decode("ascii")

# Step 1: Training Data training\_texts

= {

"English": [

"The quick brown fox jumps over the lazy dog.",

"Rip Van Winkle is a story set in the years before the American Revolutionary War.",

],

"French": [

"Le renard brun rapide saute par-dessus le chien paresseux.",

"La revolution francaise a marque une periode importante de l'histoire.",

],

"German": [

"Der schnelle braune Fuchs springt uber den faulen Hund.",

"Die deutsche Wiedervereinigung war ein historisches Ereignis.",

],

"Spanish": [

"El rapido zorro marron salta sobre el perro perezoso.",

"La Revolucion Espanola fue un momento clave en la historia." "Si quieres que te asciendan te tienes que poner las pilas.",

],

}

# Normalize training data to ASCII labels

= []

texts = []

for language, samples in training\_texts.items(): labels.extend([language] \* len(samples)) texts.extend([normalize\_to\_ascii(sample) for sample in samples])

# Step 2: Preprocessing and Feature Extraction vectorizer = TfidfVectorizer(ngram\_range=(2, 4), analyzer="char")

X\_train = vectorizer.fit\_transform(texts)

# Step 3: Train the Model classifier =

MultinomialNB() classifier.fit(X\_train, labels)

# Step 4: Serialize the Model

with open("language\_model. pkl", "wb") as model\_file: pickle.dump((vectorizer, classifier), model\_file)

# Step 5: Language Detection Function def detect\_language(snippet):

with open("language\_model. pkl", "rb") as model\_file:

vectorizer, classifier = pickle.load(model\_file) # Normalize snippet to ASCII

snippet = normalize\_to\_ascii(snippet) X\_test =

vectorizer.transform([snippet]) prediction = classifier.predict(X\_te st) return prediction[0]

# Input Processing

if \_name\_ == "\_main\_": # Read multi-line input snippet = "" while True:

try:

line = input() if line.strip(): snippet += line + " " except EOFError:

break

# Predict and Output

detected\_language = detect\_language(snippet.strip()) print(detected\_language)

# The Missing Apostrophes

**Explanation Apostrophe Handling:** The code defines a function restore\_apostrophes that iterates through each word in the input text. It uses a combination of explicit checks for common contractions (e.g., "don't," "can't," "I've") and a regular expression to handle possessive nouns (e.g., "cat's," "dog's") to restore apostrophes where appropriate.

**Source Code:**

import re

# Function to handle apostrophes for contractions and possessives def restore\_apostrophes( text):

restored\_text = [] words

= text.split()

for word in words: lower\_word = word.lower()

# Handle contractions if word.lower() == "dont": restored\_text.append("don't") elif word.lower() == "wont":

restored\_text.append("won't") elif word.lower() == "cant":

restored\_text.append("can't") elif word.lower() == "isnt":

restored\_text.append("isn't") elif word.lower() == "arent":

restored\_text.append("aren't") elif word.lower() == "wasnt": restored\_text.append("wasn't") elif word.lower() == "werent":

restored\_text.append("weren't") elif word.lower() == "hasnt":

restored\_text.append("hasn't") elif word.lower() == "havent":

restored\_text.append("haven't") elif word.lower() == "hadnt": restored\_text.append("hadn't") elif word.lower() == "didnt":

restored\_text.append("didn't") elif word.lower() == "ive":

restored\_text.append("I've") elif word.lower() == "were":

restored\_text.append("we're") elif word.lower() == "i":

restored\_text.append("I") elif word.lower() == "id":

restored\_text.append("I'd") elif word.lower() == "ive":

restored\_text.append("I've") elif word.lower() == "youve":

restored\_text.append("you've") elif word.lower() == "hes":

restored\_text.append("he's")

elif word.lower() == "shes":

restored\_text.append("she's") elif word.lower() == "its":

restored\_text.append("it's") elif word.lower() == "were":

restored\_text.append("we're")

# Handle possessives (only add 's when it makes sense)

elif re.match(r'\w+s$', word) and lower\_word not in ["its", "hers", "ours", "yours", "theirs"]: restored\_text.append(re.sub(r"s$", "'s", word))

# For normal words that don't need apostrophes, keep them as is else: restored\_text.append(word)

return " ".join(restored\_text)

# Input

input\_text = """At a news conference Thursday at the Russian manned-space facility in Baikonur, Kazakhstan, Kornienko said "we will be missing nature, we will be missing landscapes, woods." He admitted that on his previous trip into space in 2010 "I even asked our psychological support folks to send me a calendar with photographs of nature, of rivers, of woods, of lakes."

Kelly was asked if hed miss his twin brother Mark, who also was an astronaut.

"Were used to this kind of thing," he said. "Ive gone longer without seeing him and it was great."

The mission wont be the longest time that a human has spent in space - four Russians spent a year or more aboard the Soviet-built Mir space station in the 1990s.

SCI Astronaut Twins

Scott Kelly (left) was asked Thursday if hed miss his twin brother, Mark, who also was an astronaut. Were used to this kind of thing, he said. Ive gone longer without seeing him and it was great. (NASA/Associated Press)

"The last time we had such a long duration flight was almost 20 years and of course al{-truncated-}"""

# Restore apostrophes output\_text = restore\_apostrophes(input\_text) print(output\_text)

# Segment the Twitter Hashtags

**Explanation: Tokenization with Dynamic Programming:** The segment\_hashtag function uses dynamic programming to break down the hashtag into a sequence of words. It iterates through the hashtag, checking for valid word combinations from a given dictionary and selecting the longest possible valid sequence.

**Source Code:**

# Define a function that segments a single hashtag into words def segment\_hashtag(hashtag, word\_dict):

n = len(hashtag)

dp = [None] \* (n + 1)

dp[0] = [] # Base case: empty string can be segmented as an empty list

# Iterate over the hashtag string for i in range(1, n + 1):

for j in range(max(0, i - 20), i): # Limit the length of words checked word = hashtag[j:i]

if word in word\_dict and dp[j] is not None:

dp[i] = dp[j] + [word] break

return " ".join(dp[n]) if dp[n] is not None else hashtag

# Main function to process input and output results

def process\_hashtags( num\_hashtags, hashtags, word\_dict): result = []

for hashtag in hashtags:

segmented = segment\_hashtag(hashtag, word\_dict) result.append(segmented) return result

# Sample dictionary of common words (expand this as needed) word\_dict

= {

"we", "are", "the", "people", "mention", "your", "faves",

"now", "playing", "walking", "dead", "follow", "me"

}

# Sample input num\_hashtags

= int(input())

hashtags = [input().strip() for \_ in range(num\_hashtags)]

# Process the hashtags and print the result

segmented\_hashtags = process\_hashtags(num\_hashtags, hashtags, word\_dict) for segmented in segmented\_hashtags:

print(segmented)

1. **Expand the Acronyms**

**Explanation: Acronym Extraction:** The code extracts acronyms and their potential expansions from a given set of text snippets by identifying uppercase words within parentheses and searching for preceding phrases. It also attempts to extract acronyms not explicitly defined in parentheses by analyzing the surrounding context.

**Source Code:**

import re

def extract\_acronyms\_and\_expansions(snippe ts):

"""

Extract acronyms and their expansions from the provided snippets. """

acronym\_dict = {}

for snippet in snippets:

# Find all potential acronyms (uppercase words typically enclosed in parentheses) matches

= re.findall(r'\((\b[A-Z]+\b)\)', snippet)

for match in matches:

# Extract the preceding text (potential expansion) preceding\_text

= snippet.split(f"({match})")[0].strip()

# Look for the last meaningful phrase before the acronym expansion\_candidates = re.split(r'[.,;:-]', preceding\_text) if expansion\_candidates:

expansion = expansion\_candidates[-1].strip() acronym\_dict[match] = expansion

# Additionally, handle acronyms not in parentheses but defined explicitly words = snippet.split() for i, word in enumerate(words):

if word.isupper() and len(word) > 1: # Likely anacronym if word not in acronym\_dict:

# Try to extract its expansion from the surrounding context if i > 0:

preceding\_context = " ".join(words[max(0, i- 5):i]) if preceding\_context:

acronym\_dict[word] = preceding\_context return acronym\_dict

def process\_tests(acronym\_dict, tests):

"""

Process test acronyms and return their expansions. """

results = []

for test in tests:

# Normalize the test acronym (case insensitive) expansion = acronym\_dict.get(test.upper(), "Not Found") results.append(expansion)

return results

def main():

# Read input

n = int(input().strip())

snippets = [input().strip() for \_ in range(n)] tests = [input().strip() for \_ in range(n)]

# Extract acronyms and expansions acronym\_dict = extract\_acronyms\_and\_expansions( snippets)

# Process test queries results = process\_tests(acronym\_dict, tests)

# Output results print("\n".join(results))

if \_name\_ == "\_main\_":main()

# Correct the Search Query

**Explanation:** Here is a basic implementation using Python, focusing on spell correction using edit distance and a predefined corpus of words. This code uses zlib for compression and pickle for serialization, suitable for building an offline model.

**Source Code:**

import re import pickle import zlib

from collections import Counter

# Build corpus from a sample dictionary (you can enhance it with more words)

words = """going to china who was the first president of india winner of the match food in america"""

def words\_list(text):

return re.findall(r'\w+', text.lower())

WORDS = Counter(words\_list( words))

# Compression for large wordlist

with open('compressed\_dict. pkl', 'wb') as f: compressed = zlib.compress(pickle.dumps(WORDS)) f.write(compressed)

# Load dictionary in memory def load\_dictionary():

with open('compressed\_dict. pkl', 'rb') as f: return pickle.loads(zlib.decompress(f.read()))

def edit\_distance\_one(word):

letters = 'abcdefghijklmnopqrstuvwxyz'

splits = [(word[:i], word[i:]) for i in range(len(word) + 1)] deletes = [L + R[1:] for L, R in splits if R]

transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R) > 1] replaces = [L + c + R[1:] for L, R in splits if R for c in letters] inserts = [L + c + R for L, R in splits for c in letters]

return set(deletes + transposes + replaces + inserts)

def known(words, dictionary):

return set(w for w in words if w in dictionary)

def candidates(word, dictionary):

return (known([word], dictionary) or known(edit\_distance\_one(word), dictionary) or

[word])

def correct\_word(word, dictionary):

return max(candidates(word, dictionary), key=dictionary.get)

def correct\_query(query, dictionary):

return ' '.join(correct\_word(word, dictionary) for word in query.split())

# Main correction function if name== " main ":

dictionary = load\_dictionary() n = int(input())

queries = [input().strip() for \_ in range(n)]

for query in queries:

print(correct\_query(query, dictionary))

# A Text-Processing Warmup

**Explanation: Article and Date Counting:** The code defines a function count\_articles\_and\_dates that takes a text fragment as input. It first normalizes the text to lowercase for case-insensitive article counting. Then, it uses regular

expressions to count occurrences of the definite and indefinite articles ("a," "an," "the") and identify valid dates in various formats (e.g., "DD Month YYYY," "Month DD, YYYY," etc.)

**Source Code:**

import re

def count\_articles\_and\_dates(fragment):

"""

Count occurrences of 'a', 'an', 'the', and valid dates in a given text fragment. """

# Normalize text for article counting lower\_fragment = fragment.lower()

# Count articles

a\_count = len(re.findall(r'\b[a]\b', lower\_fragment)) an\_count = len(re.findall(r'\b[an]\b', lower\_fragment)) the\_count = len(re.findall(r'\b[the]\b', lower\_fragment))

# Identify valid dates date\_patterns = [

r'\b\d{1,2}(?:st|nd|rd|th)?(?:\s+of)?\s+(January|February|March|April| May|June|July|August|September|Oc tober|November|December)\s+\d{2,4}\b', # Day Month

Year

r'\b(January|February|March|April|May|June|July|August|September|October|November|December)

\s+\d{

1,2}(?:st|nd|rd)?,?\s+\d{2,4}\b', # Month Day Year r'\b\d{1,2}/\d{1,2}/\d{2,4}\b', # Day/Month/Year

r'\b\d{4}-\d{2}-\d{2}\b' # ISO format: Year-Month-Day

]

# Combine all date patterns date\_regex = '|'.join(date\_patterns) dates = re.findall(date\_regex, fragment, re.IGNORECASE) date\_count = len(dates)

return a\_count, an\_count, the\_count, date\_count

def main():

import sys

input = sys.stdin.read

# Read input data

data = input().strip().split("\n")

t = int(data[0]) # Number of test cases

fragments = data[1:] # Remaining lines contain the fragments

results = []

for i in range(t):

fragment = fragments[i].strip() # Count articles and dates a\_count, an\_count, the\_count, date\_count = count\_articles\_and\_dates(fragment) results.append(f"{a\_count}\n{an\_count}\n{the\_count}\n{ date\_count}")

# Output results print("\n".join(results))

if \_name\_ == "\_main\_":

main()

# 10.Who is it?

**Explanation: Pronoun Identification and Entity Matching:** The code first finds all pronouns (words enclosed in double backslashes) and their positions in the text. It then cleans the text by removing the backslashes. Next, it iterates through each pronounand searches for the closest matching entity (from a provided list) that appears before the pronoun in the text.

**Source Code:**

import re

def resolve\_pronouns(text, entities): # Extract all pronouns and their positions pronoun\_pattern = r'\\(\w+)\\'

pronouns = [(match.group(1), match.start()) for match in re.finditer(pronoun\_pattern, text)]

# Clean the text by removing \*\* markers clean\_text

= re.sub(r'\\(\w+)\\', r'\1', text)

# Initialize a list to store the resolved entities resolved = []

# For each pronoun, find the corresponding entity for pronoun, pos in pronouns:

closest\_entity = None closest\_distance = float('inf')

# Iterate through all entities to find the best match for entity in entities:

entity\_pos = clean\_text.rfind(entity, 0, pos) # Find the last occurrence of the entity before the pronoun

if entity\_pos != -1:

distance = pos - (entity\_pos + len(entity)) if distance < closest\_distance:

closest\_distance = distance closest\_entity

= entity

# Append the resolved entity to the list resolved.append(closest\_entity)

return resolved

def main():

import sys

input = sys.stdin.read

data = input().strip().split("\n")

# Read the number of lines in the text snippet n = int(data[0])

# Combine the next N lines into the full text snippet text\_snippet = " ".join(data[1:n + 1])

# Read the list of entities

entities = [e.strip() for e in data[n + 1].split(';')]

# Resolve pronouns

result = resolve\_pronouns(text\_snippet, entities)

# Output the resolved entities for entity in result: print(entity)

if \_name\_ == "\_main\_": main()