DSL 504
Natural Language Processing

Assignment 1

Question 1: Word Segmentation and Part-of-Speech Tagging for English and German

Given a contiguous string of characters (without spaces), first segment it into its most probable sequence of words. Then, for this segmented sequence, determine the most probable sequence of Part-of-Speech (POS) tags. This task will be performed for both English and German, allowing for a comparison of language model performance across morphologically distinct languages.

Example (English):

Input String: thequickbrownfoxjumpsoverthelazydog
Expected Output (Segmentation & POS Tagging):
[(the, DT), (quick, JJ), (brown, JJ), (fox, NN), (jumps, VBZ), (over, IN), (the, DT), (lazy, JJ), (dog, NN)]

Example (German):

Input String: hausundgarten
Expected Output (Segmentation & POS Tagging):
[(Haus, NN), (und, CC), (Garten, NN)]
(Note: German nouns are capitalized, but the input string is lowercase to maintain the challenge of segmentation from a raw stream of characters.)

Corpora:

English: Use the Brown Corpus from nltk for training both the word segmentation model and the POS tagging model.
German: Use the UD_German-GSD corpus (Universal Dependencies project) for training both the word segmentation model and the POS tagging model.

You can download the data by cloning the official GitHub repository: `git clone` https://github.com/UniversalDependencies/UD_German-GSD.git

from conllu import parse

```python
from pathlib import Path

# Load helper
def load_ud_conllu(path):
    with open(path, "r", encoding="utf-8") as f:
        sents = parse(f.read())
    tokens = [[tok["form"] for tok in sent] for sent in sents]
    upos   = [[tok["upos"] for tok in sent] for sent in sents]
    return tokens, upos

# Path to your cloned repo
root = Path("UD_German-GSD")

# Load train/dev/test
de_train_tokens, de_train_upos = load_ud_conllu(root / "de_gsd-ud-train.conllu")
de_dev_tokens,   de_dev_upos   = load_ud_conllu(root / "de_gsd-ud-dev.conllu")
de_test_tokens,  de_test_upos  = load_ud_conllu(root / "de_gsd-ud-test.conllu")
```

This corpus provides word segmentation and POS tagging..

Use the train split for training, dev for tuning, and test for evaluation.

Task Breakdown:

1. **Word Segmentation Model (for both English and German):**
   For each language, train a language model (e.g., a trigram model) on its respective corpus to determine the probability of word sequences.
   Implement a dynamic programming algorithm (like Viterbi or similar) to find the most probable word segmentation for a given input string.

2. **Part-of-Speech (POS) Tagging Model (for both English and German):**
   For each language, train a separate language model (e.g., a trigram) on the POS-tagged sentences of its respective corpus. This model should estimate:
   1. The probability of a word given its tag (emission probability, e.g., P(word | tag))
   2.The probability of a tag sequence (transition probability, e.g., P(tag_i | tag_i-1, tag_i-2))

For the segmented word sequence, use this model to find the most probable sequence of POS tags.

Comparative Analysis & Report:

After implementing and testing for both languages:
Observed Differences: How did the performance (e.g., accuracy, types of errors) of the segmentation and POS tagging models differ between English and German?

Test Set and Evaluation:

Test Set Generation: Divide each corpus (Brown for English, Tiger for German) into an 80% training set and a 20% test set. Ensure proper separation (sentences in the test set should not be used for training).
For each sentence in your test set, remove all spaces and create a contiguous character string. You will then attempt to segment and tag these strings.

Evaluation:
For the segmentation task, report the word accuracy for both English and German: (number of correctly segmented words) / (total number of words in the original sentence).
For the POS tagging task, report the POS tagging accuracy for both English and German: (number of correctly tagged words) / (total number of correctly segmented words).
Provide example input strings and their corresponding outputs from your system for both languages.

Example Test Strings (German):

meineelternliebendaswandern (My parents love hiking)
Expected: [(meine, PPER), (Eltern, NN), (lieben, VVFIN), (das, ART), (Wandern, NN)]

autobahnmeistereiverwaltungsgebaeude (Motorway maintenance administration building)
Expected: [(Autobahnmeisterei, NN), (Verwaltungsgebäude, NN)] (Or potentially Autobahn, Meisterei, Verwaltungsgebäude depending on lexicon and probabilities)

diedonnausindwunderschön (The Danube is beautiful)
Expected: [(Die, ART), (Donau, NN), (sind, VAFIN), (wunderschön, ADJD)]

Submission Requirements:

Model Codes: Your Python code for implementing both the word segmentation model and the POS tagging model for both English and German.
Test Set Generation Code: The code used to prepare your test data for both languages.
Evaluation Code: The code used to calculate and report the accuracies for both languages.
Output Examples: Show the system's output for the provided example test strings and a few of your own for both languages.
Comparative Analysis Report.

Marking Scheme (Total 70 Marks):

5 Marks: Proper training-test data distribution and separation (for both languages).

20 Marks: Word Segmentation Model Implementation:
Trigram language model for segmentation (10 marks - 5 for English, 5 for German)
Dynamic programming algorithm for segmentation (10 marks - 5 for English, 5 for German)

25 Marks: POS Tagging Model Implementation:
Training emission probabilities P(word | tag) (10 marks - 5 for English, 5 for German)
Training transition probabilities P(tag | tag_prev, tag_prev2) (10 marks - 5 for English, 5 for German)
Viterbi-like algorithm for POS tagging (5 marks - 2.5 for English, 2.5 for German)

10 Marks: Evaluation Code and Accuracy Report (5 for English, 5 for German).

10 Marks: Comparative Analysis Report.

Question 2 : Implement and Train a Transition-Based Dependency Parser

The goal of this assignment is to build a simple, data-driven dependency parser from scratch. You will implement a transition-based parser that uses a classifier trained on a treebank to make its decisions.

Dataset: You will use the **Universal Dependencies English-EWT** corpus. You should use the `en_ewt-ud-train.conllu` file for training your model and the `en_ewt-ud-dev.conllu` file for evaluating its performance.

You can download the data by cloning the official GitHub repository: `git clone https://github.com/UniversalDependencies/UD_English-EWT.git`

Transition System: You are required to implement a dependency parser based on the **arc-standard** transition system. The system uses a stack, a buffer, and a set of arcs to build a parse tree. The possible transitions are:

1. **SHIFT**: Move the first word from the buffer to the top of the stack.
2. **LEFT-ARC(label)**: The word at the top of the stack becomes the head of the second word on the stack. The second word is then popped from the stack.
3. **RIGHT-ARC(label)**: The second word on the stack becomes the head of the word at the top of the stack. The top word is then popped from the stack.

Implementation Details:

Your implementation should be divided into the following parts:

Part 1: Data Processing and Oracle Simulation

1. **CoNLL-U Parser:** Write a function to read the `.conllu` files. It should parse each sentence and store its words, POS tags, and gold-standard head-dependent relationships.
2. **Oracle Simulator:** Write a function that takes a gold-standard parsed sentence (from the CoNLL-U file) and simulates the parsing process. For each step (configuration), it should determine the correct **oracle transition** (SHIFT, LEFT-ARC, or RIGHT-ARC) that leads to the gold-standard tree. This function will generate your training data: a list of `(configuration, correct_transition)` pairs.

Part 2: Feature Extraction and Model Training

1. **Feature Extractor:** Write a function that takes a parser configuration `(stack, buffer)` and extracts features. For this assignment, you should implement the following simple features:
    - POS tag of the word on top of the stack.

    - POS tag of the second word on the stack (if it exists).
    - POS tag of the first word in the buffer (if it exists).
    - POS tag of the second word in the buffer (if it exists).
2. **Model Training:** Use the training data generated in Part 1 to train a classifier. This classifier will act as your **oracle** to predict the next transition for a given configuration.

Part 3: Parser Implementation and Evaluation

1. **Parser:** Implement the main parser function. It should take a sentence (a list of words and their POS tags) as input. It will initialize a configuration and then loop, at each step doing the following:
    - Extract features from the current configuration.
    - Use your trained classifier to predict the next transition.
    - Apply the predicted transition to update the configuration. The loop continues until the parsing is complete. The function should return the set of predicted dependency arcs.
2. **Evaluation:** Write an evaluation script that runs your trained parser on the `en_ewt-ud-dev.conllu` dataset. You should calculate the **Labeled Attachment Score (LAS)**, which is the percentage of words that were assigned the correct head and the correct dependency label.

Some example sentences to test your system:

- "The cat sat on the mat."
- "She eats a green salad."
- "I saw the man with a telescope."

Marking Scheme: (40)

1. **Part 1: Data Processing and Oracle Simulation (12 marks)**
    - 5 marks - Correctly parsing the CoNLL-U file and storing the data structures.
    - 7 marks - Correct implementation of the oracle simulator to generate training instances.
2. **Part 2: Feature Extraction and Model Training (10 marks)**
    - 5 marks - Implementation of the feature extraction function.
    - 5 marks - Correctly training a scikit-learn classifier with the generated features and labels.

3. **Part 3: Parser and Evaluation (15 marks)**
    - 10 marks - Implementation of the core parsing loop that uses the trained model.
    - 5 marks - Correct implementation of the LAS evaluation metric.
4. **Code Quality and Report (3 marks)**
    - 3 marks - Well-structured, commented code and a brief report explaining your design choices and final LAS score.

Question3: Building and Evaluating an Efficient Spelling Corrector

The goal of this assignment is to build a robust spelling corrector that can handle both non-word and real-word errors within one edit distance. A key part of this assignment is to implement and compare two different methods for generating candidate corrections, analyzing their impact on performance and speed.

Dataset: You will use the **Brown Corpus** from the NLTK library to build your vocabulary and language model. This corpus is well-balanced and suitable for gathering word frequency and contextual probabilities.

Implementation Details:

Your task is to create a spelling corrector by implementing the following components.

Part 1: Corpus and Model Preparation

1. **Vocabulary and Frequencies:** Process the **Brown corpus** to create a vocabulary of unique words and a frequency distribution (unigram model).
2. **Language Model:** Create a bigram probability model from the corpus. This will be used to handle real-word errors by considering the context of a word.

Part 2: Candidate Generation Methods You must implement two distinct methods for generating candidate corrections for a given misspelled word.

1. **Method A: Standard Edit Distance 1 Generation**
    - Write a function that takes a word and generates a set of all possible words at an edit distance of 1 (deletions, transpositions, replacements, and insertions).

2. **Method B: Symmetric Delete Spelling Correction**
   - This is an efficient method for finding candidates.
   - **Preprocessing Step:** Create a dictionary that maps every possible one-character deletion of a vocabulary word back to the original word (e.g., `{'ello': ['hello'], 'hllo': ['hello'], ...}`).
   - **Candidate Generation Step:** To find candidates for a misspelled word, first generate all its one-character deletions. Then, look up these deleted variants in your pre-processed dictionary to find matching vocabulary words.

Part 3: Spelling Correction Logic

1. **Non-Word Error Correction:** For a word not found in your vocabulary, use both Method A and Method B to generate candidate sets. The best correction is the candidate with the highest frequency (unigram probability).
2. **Real-Word Error Correction:** For a word that *is* in the vocabulary but may be incorrect in its context (e.g., "I ate an **apple**" vs. "I ate an **apply**"), you must:
   - Generate candidate corrections for the target word using one of your methods.
   - Use your bigram model to calculate the probability of the original phrase (e.g., P("an apple")) versus the probability of phrases with the corrected candidates (e.g., P("an apply")).
   - If a candidate phrase has a significantly higher probability, suggest it as the correction.

Part 4: Evaluation

1. **Test Set Generation:** Create a test set by taking 10% of the sentences from the Brown corpus. For each sentence, randomly select one word and introduce a single-edit spelling mistake to create both a non-word and a real-word error version.
2. **Accuracy:** Report the accuracy of your spelling corrector on both the non-word and real-word test sets.
3. **Performance Comparison:** For the non-word error test set, measure and compare the total time taken to generate candidates using Method A versus Method B. Write a brief conclusion on which method is faster and why.

Some example sentences to test your system:

- Non-word error: "I hav a good feeling about this."
- Non-word error: "This is a test sentnce."
- Real-word error: "I would like to sea the world."

- Real-word error: "Please meat me at the station."

Marking Scheme: (40)

1. **Part 1: Corpus and Model Preparation (8 marks)**
   - 4 marks - Correctly building the vocabulary and unigram frequency model.
   - 4 marks - Correct implementation of the bigram probability model.
2. **Part 2: Candidate Generation Methods (12 marks)**
   - 6 marks - Correct implementation of the standard edit distance 1 generation (Method A).
   - 6 marks - Correct implementation of the symmetric delete method, including the pre-processing step (Method B).
3. **Part 3: Spelling Correction Logic (8 marks)**
   - 4 marks - Implementation for handling non-word errors using unigram probabilities.
   - 4 marks - Implementation for handling real-word errors using the bigram model for context.
4. **Part 4: Evaluation and Analysis (12 marks)**
   - 4 marks - Test set generation code.
   - 4 marks - Accuracy calculation for both error types.
   - 4 marks - Code for performance comparison (timing) and a clear written conclusion analyzing the speed difference between Method A and Method B.