Introduction to Automatic Speech Recognition: Assignment 2

Given: 8 Dec 2025

Due: 24 Dec 2025

Exercise 1

In the attached file `sequence_alignment.py` you will find the implementation of the following function:

```
align_sequences(first_seq: Iterable,
                second_seq: Iterable,
                weights: EditWeights)
```

This function accepts a pair of sequences, and an implementation of the abstract base class `EditWeights` (also defined in this file), which should implement the weight function – see its documentation.

1. Implement a derived class that will let you compute the Levenshtein distance between a pair of strings. Use the output of the function to compute the Levenshtein distance between the strings "`compassion`" and "`comparison`", and to give details which edit operation exactly take place (an edit operation is either “replace A by B”, “delete A” from the first string, or “insert B” from the seconds string).

2. Use the class you have implemented to compute the alignment between the two word sequences:

   Text#1: “friends romans countrymen lend me your ears i come to bury caesar not to praise him”
   Text#2: “my friends romans country men land me your ears i come to bury caesar note raise”

3. Implement a class that lets you compute the alignment between a pair of strings using the uniform weights: 2 for a character match, (-1) for a substitution, and (-0.75) for a character insertion or a character deletion. Given these weights, what is the alignment score between the strings `"compassion"` and `"comparison"`?

4. Use the class you implemented in the previous section to implement another derived class from `EditWeights` that can operate on a pair of strings:
   - `pair_weight(s_1, s_2)` is the alignment score of the two strings.
   - `insertion_weight(s)` or `deletion_weight(s)` is the alignment score of the string versus an empty string.

   Use this class to compute the alignment between the pair of word sequences from Section 2, and compare the alignment results to the ones you obtained in Section 2. Which ones are better?

Exercise 2

Here are some statistics collected from the text of the book "Alice in Wonderland" by Lewis Carroll: Vocabulary size is 2666 with a total of 28521 words.

Below are the statistics of several unigrams. Recall that the number of prefixes for a word **W** is the number of unique bigrams seen in the text of the form (**W**, X).

Word	Number of occurrences	Number of prefixes
alice	386	133
cat	35	23
cheshire	7	3
hatter	55	32
mad	15	9
queen	68	32
red	15	9
the	1641	449
white	30	5

1. Compute the unigram weights for each of these unigrams, and for the special <unk> unigram (the weight for an out-of-vocabulary word). Use the following three methods and compare the resulting weights:
   - No smoothing (not need to compare the weight for the <unk> unigram in this case, it should be $-\infty$).
   - Using Laplace smoothing with $\alpha = 1$.
   - Allocating 5 additional occurrences for the <unk> unigram.

Below are the statistics for several bigrams:

Word 1	Word 2	Number of occurrences
the	queen	65
white	rabbit	22
cheshire	cat	5
the	hatter	51
alice	</s>	59

2. Compute the bigram weights for all these bigrams using the next three methods:
   - No smoothing.
   - Using Laplace smoothing with $\alpha = 1$.
   - Using Witten-Bell smoothing.

3. Using the latter two methods, compute the bigram weights for the following unseen bigrams (i.e. these bigrams have 0 occurrences in the text):

   white queen

   cheshire and

   mad hatter

Note: Make sure that when you compute the logarithmic weights, use the base-10 logarithm (and not the natural logarithm).

Exercise 3

In the file `spanish_gtp_rules.json` you will find the definition of a set of grapheme-to-phonemes rules for European Spanish. The file contains the following sections:

- `graphemes` – a list of all accepted graphemes, or characters, in the language (blank-delimited). In addition to these graphemes, there is another implicitly defined auxiliary grapheme: The character `$`, which defines either the word start or the word end.
- `subsets` – definitions of subsets of graphemes: Each subset has a name, and the set of graphemes that it contains. This section is optional: In principle, there may be no subsets defined.
- `rules` – Each rule contains a *center* grapheme, or a sequence of several graphemes. The subset of graphemes (either a subset name or a blank-delimited list of graphemes) that are allowed as at the *predecessor* position, and that are allowed at the *successor* position, may be specified as well, but are not required. Finally, the resulting *phonemes* from the rule are specified.

Write a class named `GtpConverter` that is initialized with a file containing the definitions of the GTP rules, as described above. The class has a function `process()` which accepts a string of graphemes (that is, an input word), and outputs a string stating the word's phonetic transcription, or `None` if the input word contains illegal characters.

Use the given rules file and the class you have written, and convert the following words into phonemes:

sueño, pequenita, desarrollar, guitarra, cigüeña, alburquerque, atenúas, zorro, muchacho, hierro, mándamelo, rápidamente, chiringuitos, caballeros, escribí

Just in so you are able to read the output to yourselves: Most phonemes in Spanish are quite straightforward. Only the following symbols require special clarification:

- x sounds like the Hebrew כ.
- j sound like **y** in the English word yacht.
- θ sounds like **th** in the English word *thin*.
- ʧ sounds like **ch** in the English word *chin*.
- ɾ is a soft **r** sound and r is a rolled **r**.
- ʝ sounds something in between the English y (as in *yacht*) and j (as in *jam*).
- ɲ sound like **ny** in English word canyon.