TUtorial 5

## **Exercise 1**

Question 1 can be found in **decision\_tree** folder submitted along with this external documentation.

Question 2 can be found in **neural\_network** folder submitted along with this external documentation.

## **Exercise 2**

1. **Character-Level Embedding**

**Character-Level Embedding:**

Character-level embedding involves representing text data at the level of individual characters rather than words. This means each character in a word is encoded as a vector, and words are represented as sequences of these character vectors.

**Advantages over Word Embedding:**

- **Handling Out-of-Vocabulary Words:** Character-level embedding can better handle words not seen during training (out-of-vocabulary words) by constructing them from known characters.

- **Morphological Awareness:** They can capture morphological variations of words, such as prefixes, suffixes, and roots, which is particularly useful for languages with rich morphology.

- **Reduced Vocabulary Size:** The vocabulary size is drastically reduced since it consists of characters (typically 26 letters in the English alphabet plus punctuation) instead of a large number of words.

2. **Pre-trained vs. Fine-tuned Models**

**Pre-trained:**

A pre-trained model is a neural network model that has been previously trained on a large dataset and can be use as a starting point for a new, related task. These models have learned general features from the pre-training data that can be beneficial for the new task.

**Example:**

- **BERT (Bidirectional Encoder Representations from Transformer):** Pre-trained on a large corpus of text data and can be used for various NLP tasks.

**Fine-tuned:**

Fine-tuning involves taking a pre-trained model and training it further on a specific task with smaller, tasks-specific dataset. This adjusts the model weights to better suit the new task while leveraging the knowledge gained during pre-training.

**Example:**

- **Fine-tuning BERT for Sentiment Analysis:** A pre-trained BERT model can be fine-tuned on a sentiment analysis dataset to adapt it specifically for classifying the sentiment of text.

3. **Attention Mechanism in Seq2Seq Models**

**Attention Mechanism:**

This attention mechanism allows a model to focus on different parts of the input sequence when generating each part of the output sequence. This is particularly useful in sequence-to-sequence (seq2seq) models, where the length of the input and output sequence may vary, such as in machine translation.

**How it Helps in Seq2Seq Models:**

**- Improved Translation Quality:** By focusing on relevant parts of the input sequence, attention can improve translation quality by ensuring that important context is considered for each word in the output.

- **Handling Long Sequences:** Attention helps in dealing with long sequences where traditional seq2seq models may struggle due to the vanishing gradient problem or memory limitations.

- **Dynamic Alignment:** It provides a dynamic alignment between the input and output sequences, allowing the model to decide which parts of the input are most relevant for generating each part of the output.

4. **N-gram Modelling**

a) **Unigram Modelling (Maximum Likelihood Estimation)**

To calculate the probability of a sequence “eat oats and eat ivy” using unigram modelling:

- Calculate the total number of words in the corpus.

- Calculate the frequency of each word.

- Use the formula

P(eat) = count(eat) / total number of words

P(oats) = count (oats) / total number of words

P(and) = count (and) / total number of words

P(ivy) = count (ivy) / total number of words

Total number of words = 22 (sum of all word in the corpus)

P(eat) = 4 / 22

P(oats) = 2 /22

P(and) = 3 / 22

P(ivy) = 2 / 22

P(eat oats and eat ivy) = P(eat) \* P(oats) \* P(and) \* P(eat) \* P(ivy)

= (4/22) \* (2/22) \* (3/22) \* (4/22) \* (2/22)

b) **Bigram Modelling (Maximum Likelihood Estimation)**

To calculate the probability of a sequence “eat oats and eat ivy” using bigram modelling:

- Calculate the bigram probabilities

P(eat | <start>) = count(<start>, eat) / count(<start>)

P(oats | eat) = count(eat, oats) / count(eat)

P(and | oats) = count(oats, and) / count(oats)

P(eat | and) = count(and, eat) / count(and)

P(ivy | eat) = count(eat, ivy) / count(eat)

Bigram counts:

Count(eat, oats) = 2

Count(oats, and) = 2

Count(and, eat) = 2

Count(eat, ivy) = 2

P(<start>) = 1/22

P(eat | <start>) = count (eat) / total number of words = 4/22

P(oats | eat) = 2/4

P(and |oats) = 2/2

P(eat | and) = 2/3

P(ivy | eat) = 2/4

P(eat oats and eat ivy) = P(eat | <start>) \* P(oats | eat) \* P(and | oats) \* P(eat | and) \* P(ivy | eat) = (4/22) \* (2/4) \* (2/2) \* (2/3) \* (2/4)

c) **Bigram Modelling with Add-k Smoothing (k=1)**

To calculate the probability of “P(oats |eat)” using add-k smoothing with k=1:

- Use the formula

Vocabulary size (V) = 18 # Number of unique words in the corpus count(eat, oats) = 2 count(eat) = 4 k = 1 P(oats | eat) = (count(eat, oats) + k) / (count(eat) + k \* V) = (2 + 1) / (4 + 1 \* 18) = 3 / 22