MASABATTULA TEJA NIKHIL

MS-Research, Computer Science Department

Indian Institute of Technology, Indore

Report on

Word-to-Word Translation Using Supervised and Unsupervised Approach

Table of contents

Problem Statement

Implement and evaluate a supervised cross-lingual word embedding alignment system for English and Hindi using the Procrustes method. Follow the steps below to complete this task.

Workflow Overview

This section describes about the system design and the work flow of the implementation.

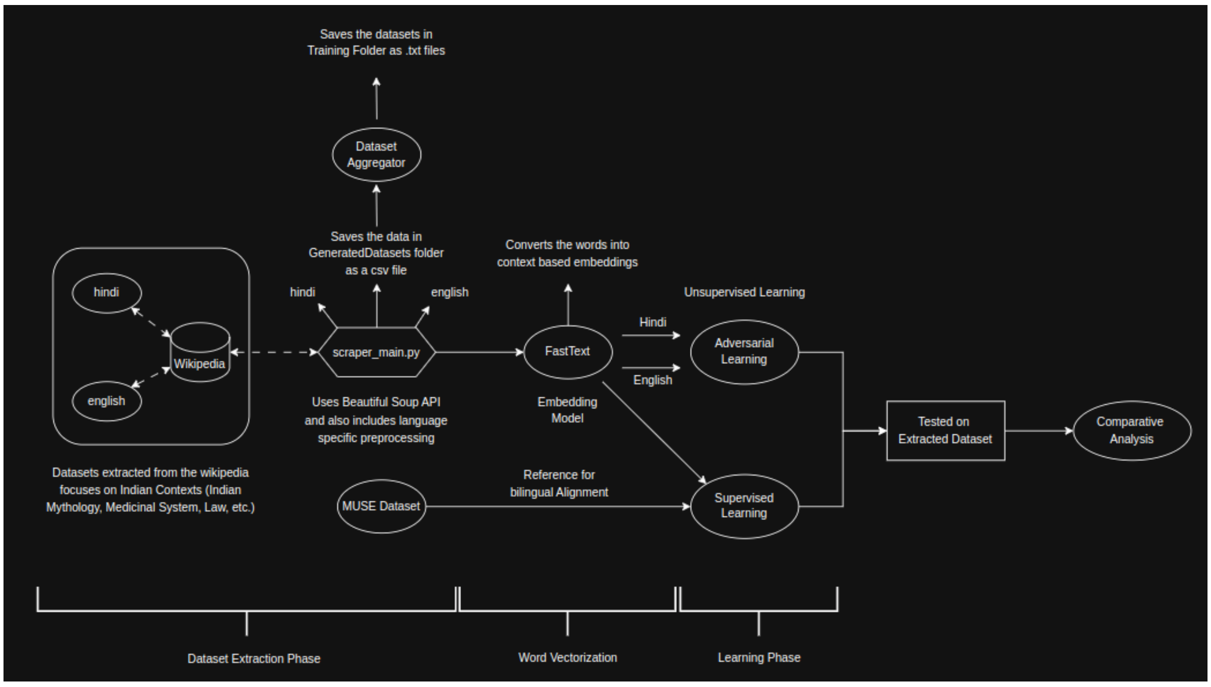
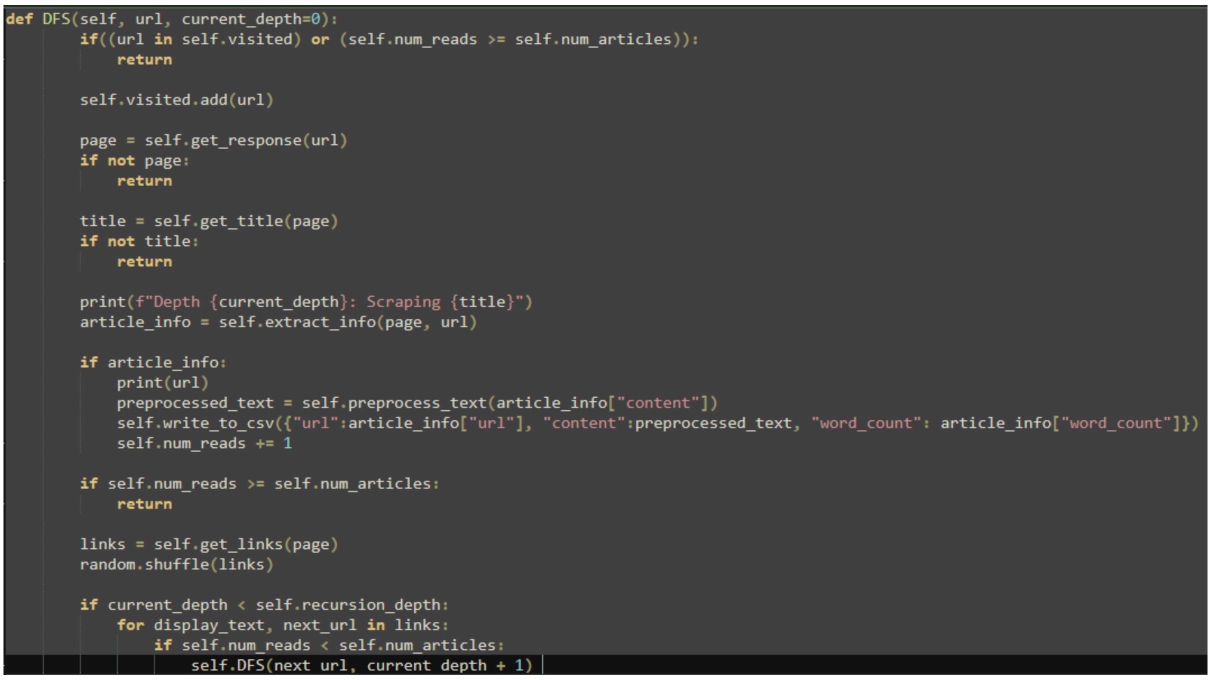

Saves the datasets in Training Folder as .txt files
Dataset Aggregator
Saves the data in GeneratedDatasets folder as a csv file
hindi
english
hindi
Wikipedia
english
scraper_main.py
Uses Beautiful Soup API and also includes language specific preprocessing
Datasets extracted from the wikipedia focuses on Indian Contexts (Indian Mythology, Medicinal System, Law, etc.)
Converts the words into context based embeddings
FastText
Embedding Model
Unsupervised Learning
Hindi
English
Adversarial Learning
MUSE Dataset
Reference for bilingual Alignment
Supervised Learning
Tested on Extracted Dataset
Comparative Analysis
Dataset Extraction Phase
Word Vectorization
Learning Phase

Figure 1 : Flow chart illustrating different phases of the making a word-to-word translator

As shown in Fig.1. there are three phases of building a word-to-word translator :

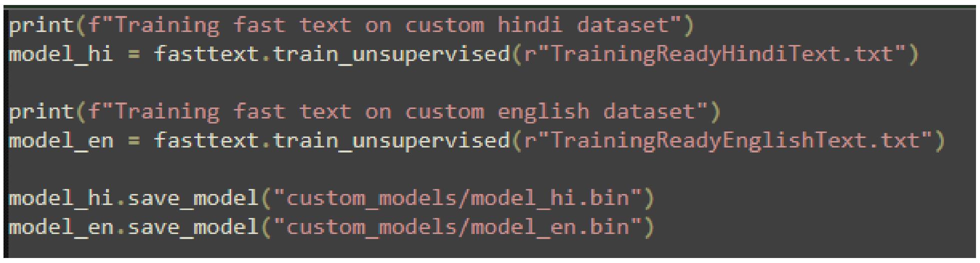
1. ***Dataset Creation Phase:*** In this phase, the English and hindi dataset has been extracted from Wikipedia using beautifulsoup-api. The goal was to build a model focused on Indian literature and history, so the extraction prioritized articles related to Indian mythology, recipes, law, and traditional medicine.
   - ***Preprocessing :*** I have induced the preprocessing code during the dataset extraction phase itself. The kinds of language specific preprocessing involved are as follows :
     1. Removing numerical values
     2. Removing the non-hindi text (while extracting hindi text)
     3. Removing non-alpha numeric characters (while extracting English text)
     4. Took a context length of 500 on each page.
     5. Removing extra blank spaces.
   - ***Challenge :*** Recursive crawling of cross-referenced Wikipedia articles sometimes led to collecting unrelated (non-Indian) content.
   - ***Solution :*** A DFS-based navigation strategy was implemented to limit traversal and maintain focus on Indian-related content.



```python
def DFS(self, url, current_depth=0):
        if((url in self.visited) or (self.num_reads >= self.num_articles)):
            return

        self.visited.add(url)

        page = self.get_response(url)
        if not page:
            return

        title = self.get_title(page)
        if not title:
            return

        print(f"Depth {current_depth}: Scraping {title}")
        article_info = self.extract_info(page, url)

        if article_info:
            print(url)
            preprocessed_text = self.preprocess_text(article_info["content"])
            self.write_to_csv({"url":article_info["url"], "content":preprocessed_text, "word_count": article_info["word_count"]})
            self.num_reads += 1

        if self.num_reads >= self.num_articles:
            return

        links = self.get_links(page)
        random.shuffle(links)

        if current_depth < self.recursion_depth:
            for display_text, next_url in links:
                if self.num_reads < self.num_articles:
                    self.DFS(next_url, current_depth + 1)
```

Figure 2: Code block illustrating the DFS traversal for traversing the Wikipedia pages recursively

2. ***Word Vectorization Phase :*** In this phase, words are converted into vector representations called *embeddings*. While methods like Word2Vec exist, this implementation uses the **FastText** model due to its superior CPU and learning efficiency. Separate FastText models were trained on the extracted English and Hindi datasets, and these embeddings are used in the next phase.



```python
print(f"Training fast text on custom hindi dataset")
model_hi = fasttext.train_unsupervised(r"TrainingReadyHindiText.txt")

print(f"Training fast text on custom english dataset")
model_en = fasttext.train_unsupervised(r"TrainingReadyEnglishText.txt")

model_hi.save_model("custom_models/model_hi.bin")
model_en.save_model("custom_models/model_en.bin")
```

Figure 3: Code for training fasttext model on custom datasets

3. ***Learning Phase:*** This is the phase where the alignment of embedding spaces happens as follows

$$y = f(x)$$

Where y is target language embedding (`hi`)

x is source language embedding (`en`)

f(x) is the function that maps source embedding to target embedding

Consider a simplest mapping, where f(x) is a transformation matrix when multiplied with `x` gives `y`

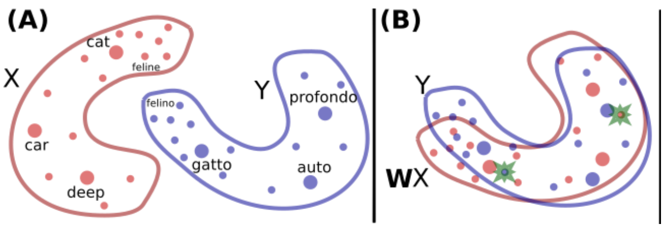
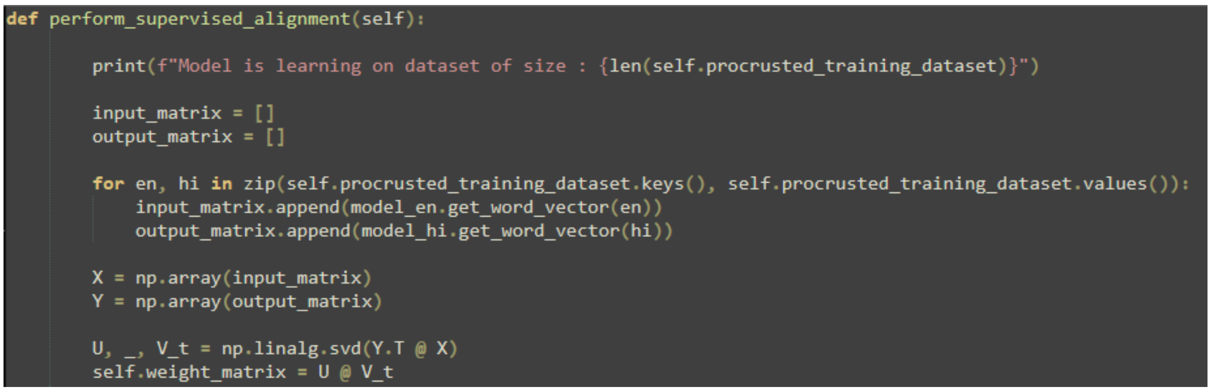


Figure 4: Projecting source embedding space onto target embedding phase using a transformation matrix (W) [1]

The objective of this phase is to learn the transformation matrix W such that the mapped source embeddings closely align with their corresponding target embeddings.

There are two ways to learn this transformation matrix :

1. ***Supervised :*** the model is provided with a bilingual dictionary containing source words and their corresponding target words. Using this supervision, the transformation matrix W is learned to align the embedding spaces.



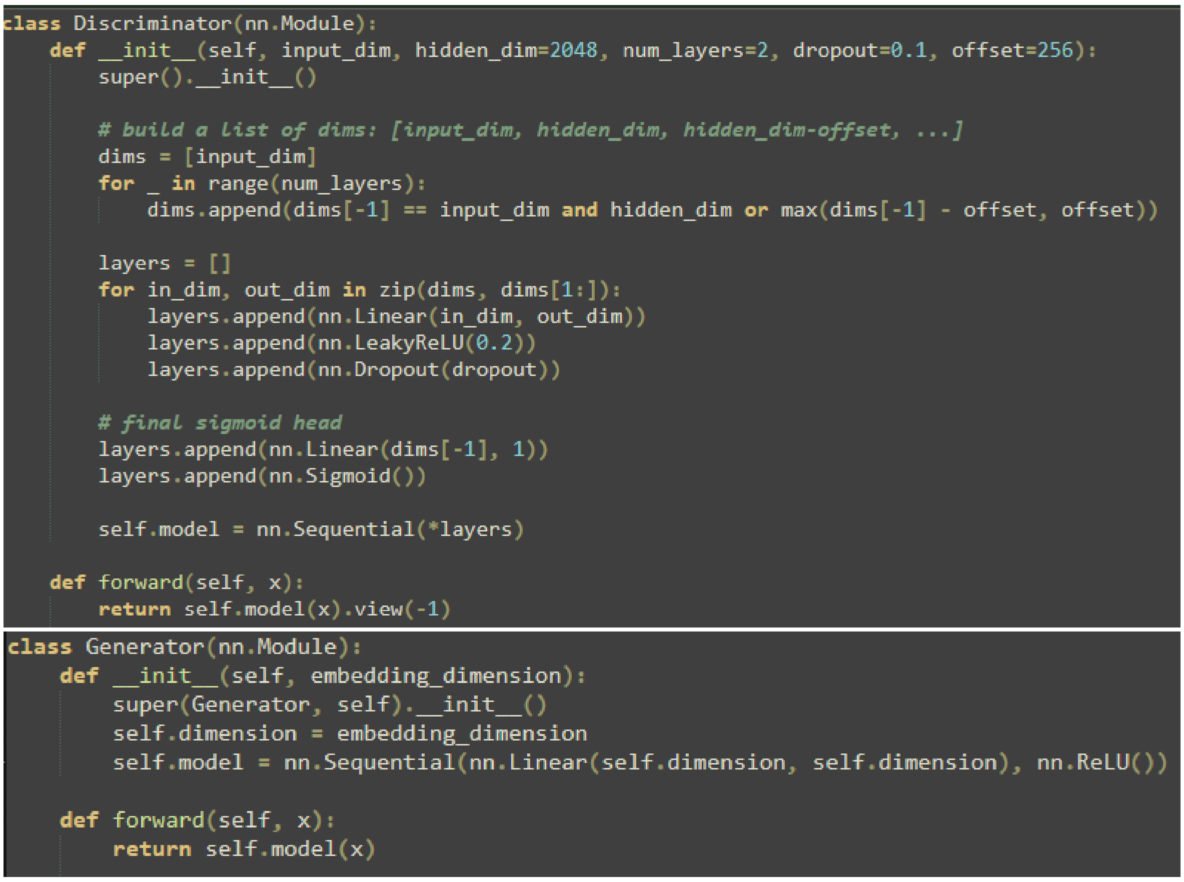
```
def perform_supervised_alignment(self):

        print(f"Model is learning on dataset of size : {len(self.procrusted_training_dataset)}")

        input_matrix = []
        output_matrix = []

        for en, hi in zip(self.procrusted_training_dataset.keys(), self.procrusted_training_dataset.values()):
            input_matrix.append(model_en.get_word_vector(en))
            output_matrix.append(model_hi.get_word_vector(hi))

        X = np.array(input_matrix)
        Y = np.array(output_matrix)

        U, _, V_t = np.linalg.svd(Y.T @ X)
        self.weight_matrix = U @ V_t
```

Figure 5 : Code block illustrating the supervised learning of transformation matrix using SVD.

2. ***UnSupervised :*** This approach is more challenging and does not require explicit (source_word, target_word) pairs. Instead, it works with the source and target vocabularies independently, without any prior cross-lingual supervision.

   The method is inspired from Generative Adversarial Networks (GANs) and has two main phases:

   - ***Adversarial Phase*** : This is where a **Generator(Transformation Matrix)** learns to transform source embeddings to appear like target embeddings and a **Disciminator** learns to distinguish between real target embeddings and generated ones.
   - ***Pseudo Target word generation phase :*** In this phase, we use the trained Generator to produce a target embedding and find the nearest neighbors to get the top `k` embeddings nearer to the predicted target embedding in that target embedding space using two different approaches such as KNN and CSLS (Cross-Domain Similarity Locality Sampling)



```python
class Discriminator(nn.Module):
    def __init__(self, input_dim, hidden_dim=2048, num_layers=2, dropout=0.1, offset=256):
        super().__init__()

        # build a list of dims: [input_dim, hidden_dim, hidden_dim-offset, ...]
        dims = [input_dim]
        for _ in range(num_layers):
            dims.append(dims[-1] == input_dim and hidden_dim or max(dims[-1] - offset, offset))

        layers = []
        for in_dim, out_dim in zip(dims, dims[1:]):
            layers.append(nn.Linear(in_dim, out_dim))
            layers.append(nn.LeakyReLU(0.2))
            layers.append(nn.Dropout(dropout))

        # final sigmoid head
        layers.append(nn.Linear(dims[-1], 1))
        layers.append(nn.Sigmoid())

        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x).view(-1)
```

```python
class Generator(nn.Module):
    def __init__(self, embedding_dimension):
        super(Generator, self).__init__()
        self.dimension = embedding_dimension
        self.model = nn.Sequential(nn.Linear(self.dimension, self.dimension), nn.ReLU())

    def forward(self, x):
        return self.model(x)
```

Figure 6: Code illustrating the GAN architecture for unsupervised learning

Fig. 6. illustrates the custom created GAN architecture where the discriminator uses a layered architecture whose input dimension is the embedding dimension in source_vocab space and output dimension is the embedding dimension in target_vocab space.

[1] introduced an extension of KNN for finding nearest neighbour in the phase 2 of learning process called as Cross-Domain Similarity Locality Sampling aka CSLS

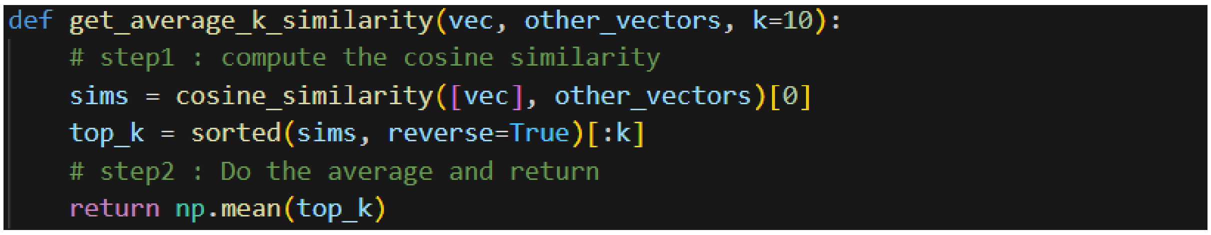
- ***Cross*-Domain *Similarity Local Sampling:*** One of the major problem with finding the nearest neighbours using KNN is its property of asymmetricity. KNN search isn't symmetric: y might be in KNN of x, but x isn't necessarily in KNN of y [1] and CSLS is a method to tackle this problem. And here is how it works :
    - Compute the cosine similarity between the two sets of embeddings X and Y
    - For each vector `x` in the source space, compute the average cosine similarity between x and its k-nearest neighbors in the target space. Similary do for each `y` in vector space Y as follows :

$$r(x) = \frac{1}{k} * \sum_{y' € kNN(x)} \cos\,(x, y')$$

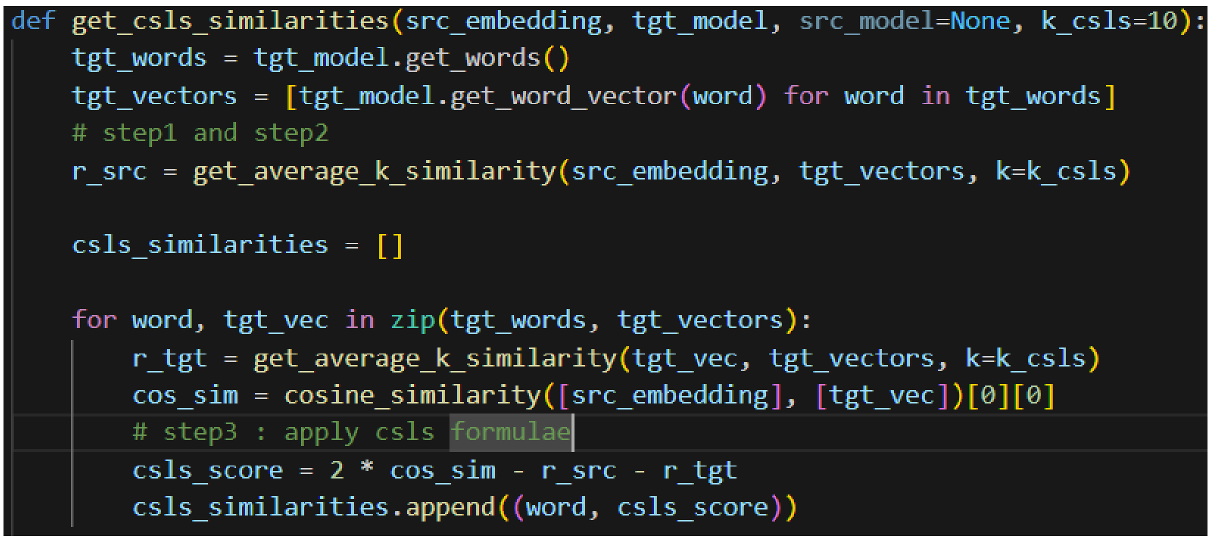
$$r(y) = \frac{1}{k} * \sum_{x' € kNN(y)} \cos\,(y, x')$$

- Apply the CSLS formulae as follows :

$$CSLS(x, y) = 2 * \cos(x, y) - r(x) - r(y)$$



```python
def get_average_k_similarity(vec, other_vectors, k=10):
    # step1 : compute the cosine similarity
    sims = cosine_similarity([vec], other_vectors)[0]
    top_k = sorted(sims, reverse=True)[:k]
    # step2 : Do the average and return
    return np.mean(top_k)
```



```python
def get_csls_similarities(src_embedding, tgt_model, src_model=None, k_csls=10):
    tgt_words = tgt_model.get_words()
    tgt_vectors = [tgt_model.get_word_vector(word) for word in tgt_words]
    # step1 and step2
    r_src = get_average_k_similarity(src_embedding, tgt_vectors, k=k_csls)

    csls_similarities = []

    for word, tgt_vec in zip(tgt_words, tgt_vectors):
        r_tgt = get_average_k_similarity(tgt_vec, tgt_vectors, k=k_csls)
        cos_sim = cosine_similarity([src_embedding], [tgt_vec])[0][0]
        # step3 : apply csls formulae
        csls_score = 2 * cos_sim - r_src - r_tgt
        csls_similarities.append((word, csls_score))
```

Figure 7: Code block for CSLS based cosine similarity

Results and Analysis

Throughout the implementation, it is assumed that the src language is English and target language is Hindi. One of the study that I have performed is based on the varying training dataset sizes for supervised training and the comparision is as show in fig. 7.

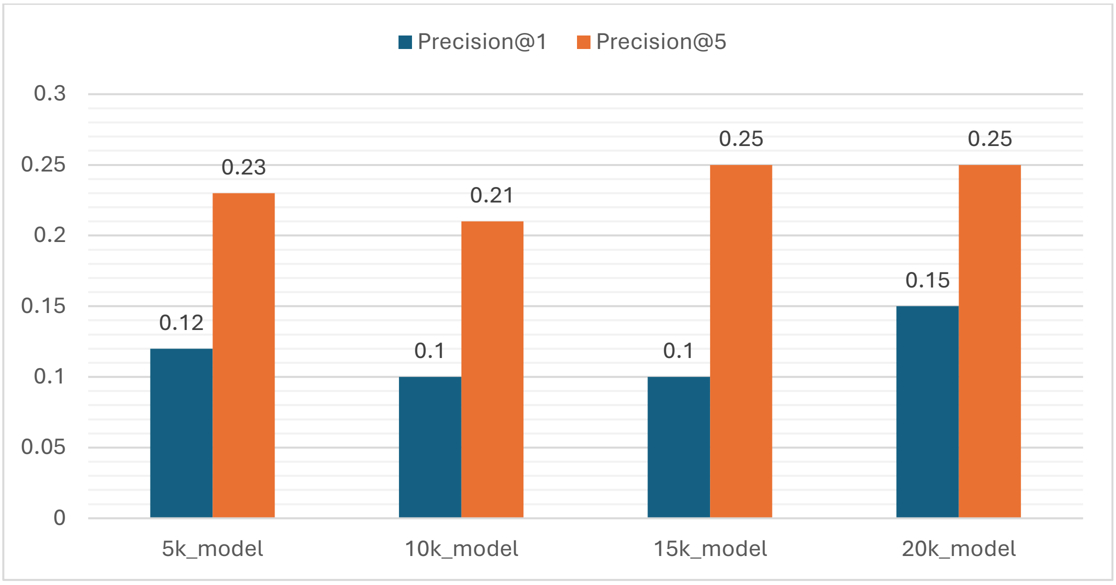


Figure 8: Bar plot illustrating a comparison between the precisions when varying the dataset sizes.

Observation :

As we are increasing the dataset sizes for training, ideally the accuracy is supposed to increase but opposingly here the precision is going up and down and the reason is attributed to Poor Coverage of High-Frequency Words that are present in the testing dataset (custom dataset).

As far as unsupervised approach is concerned, the model has been trained on the custom dataset of vocab size 10000 for 500 epochs and the learning curves are as shown in fig. 8. Unfortunately, the discriminator is learning too fast which is should in Fig. 8 and the generator is failing to fool the discriminator. This abnormal nature of the GAN might be due to the architecture of discriminator which is more deep as that of generator. And to overcome this, I have tried tweaking the hyper-parameters as follows :

- Changing the discriminators architecture
- Tweaking the hyper-parameters parameters of generator and discriminator such as {learning_rates, number_of_steps_per_epoch, batch_size}
- Soft labelling rather than giving hard labels

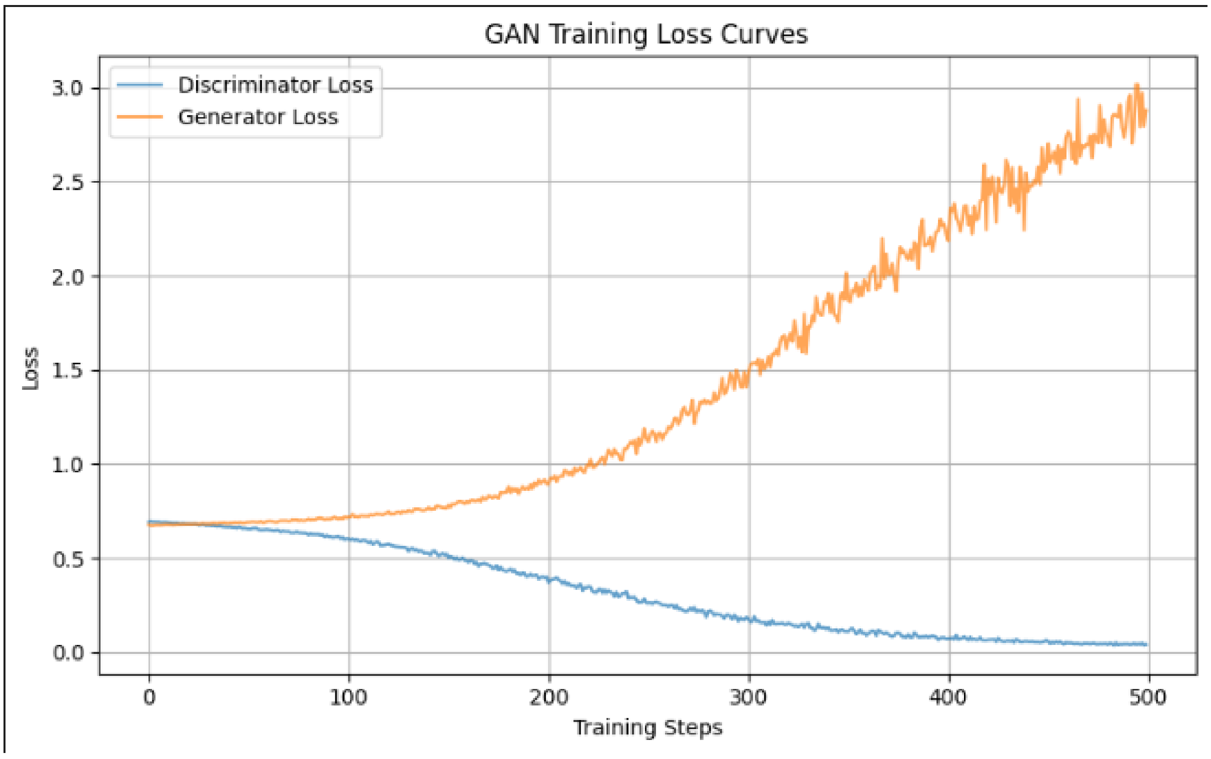


Figure 9: Training loss for generator and discriminator over the epochs

Increasing the number of epochs wouldn't help as the losses are becoming stabilized. As a result of this behaviour, the generator is failing to capture and project the src_word embeddings onto the target_word embedding space.

Conclusion

In this assignment, we implemented a supervised cross-lingual word embedding alignment system for English and Hindi using the Procrustes method. The process involved training FastText models on a custom dataset focused on Indian literature and history, followed by aligning the monolingual embeddings using a bilingual lexicon from the MUSE dataset. The Procrustes method ensured an orthogonal mapping between the source and target embedding spaces, preserving their geometric properties. Evaluation using Precision@1 and Precision@5 metrics on the MUSE test dictionary revealed how the size and quality of the bilingual lexicon affected alignment performance. Interestingly, increasing the training data size did not always lead to better results, primarily due to poor coverage of high-frequency words in the test set, emphasizing the importance of domain-relevant and well-distributed datasets.

For the extra credit component, an unsupervised approach based on adversarial training combined with CSLS was explored. Going forward, the use of efficient libraries like FAISS by Meta can significantly speed up nearest neighbour search, making these systems more practical for large-scale multilingual applications. Overall, the project highlighted the crucial role of both data preparation and model design in achieving meaningful cross-lingual alignment.

References

[1] Conneau, A., Lample, G., Ranzato, M., Denoyer, L., Jegou, H. **"Word translation without parallel data"**, arXiv:1710.04087, 2017

[2] **Multilingual Unsupervised and Supervised Embeddings (MUSE)** - Facebook Research

[3] Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. **"Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks"**, *IEEE ICCV 2017*, DOI: 10.1109/ICCV.2017.244