Global Benchmark for Local Gems: A City Development Recommender System

Anamika Mishra
MT2023147
IIIT Bangalore
Bengaluru, India
Anamika.Mishra@iiitb.ac.in

H Anarghya
MT2023083
IIIT Bangalore
Bengaluru, India
H.Anarghya@iiitb.ac.in

Nishtha Paul
MT2023161
IIIT Bangalore
Bengaluru, India
Nishtha.Paul@iiitb.ac.in

***Abstract*—India is a developing country with vast number of opportunities. Our aim is to recommend European cities which can be similar to the Indian cities, to take inspiration and identify innovative approaches to urban development. We believe that the exchange of ideas and best practices between diverse cultures will play a major factor to enable sustainable development.**

***Index Terms*—indian City, european City, recommendation, scrapping**

I. Introduction

India is a vast country in Southern Asia, and as of June 2023, it is the most populous country. It is a country with diverse cultures and traditions, vibrant tapestry of customs, rituals, and festivals celebrated throughout the year. India has become a fast-growing major economy and a hub for information technology services, with an expanding middle class. It consists of various kinds of industries flourishing in different parts of the country.

Despite its significant progress in various sectors, India still faces several challenges and obstacles on its path to development. There are several issues to be tackled, including infrastructure, gaps in educational and skill development, poverty, corruption. There is a need to come up with initiatives focusing on inclusive growth, sustainable development, and social justice for overcoming the hurdles and accelerating India's progress towards becoming a developed nation.

We believe that in this huge interconnected world, the exchange of ideas and best practices between diverse cultures will play a major factor to enable sustainable development. Many cities in the European countries are renowned for their unique blend of history, culture, and innovation, making them coveted destinations for travelers and admired models for urban development. If you observe the geographic landscape of the cities of Udaipur in India and that of Venice in Italy, they both are pretty much similar, consisting majorly of water bodies. But, compared to Udaipur, tourism in Venice flourishes to a large extent.

Hence, our aim is to is to inspire improvements in Indian cities by learning from the strengths of their European counterparts, in order to bring out the aspects which have been overlooked in these cities. This will help to identify adaptable solutions by considering India's unique social, cultural, and economic realities which can aid to enhance the quality of life, infrastructure, and overall urban experience in the different Indian cities.

II. Dataset

The first step in our project was to collect the data for Indian and European cities.

A. Format of the Data

Our requirement was the data that could represent the culture and the traditions of every city. This type of data is usually available in the form of text that describes each city rather than numbers. Hence it was decided to collect the textual data for Indian and European cities from Wikipedia, which is a free, online encyclopedia, consisting of vast amount of information that is publicly accessible to obtain the latest information.

B. Data Scrapping

Once we decide to collect text data, we collected and curated a list of Wikipedia URLs of all the cities. We used the **Scrapy** library from Python for web scrapping. Scrapy is powerful library that can efficiently crawl through multiple web pages and offers features like handling different website content formats.

Our dataset consisted of a total of 430 cities, with 267 Indian cities and 163 European cities. A Python script was created to perform the web scraping. For every web page, we extracted all the paragraph elements and then obtained the text content of each paragraph element.

The script was executed to obtain all the data in a JSON file.

C. Data Preprocessing

Next step was to pre-process the data to remove the unnecessary information. This was done using the **nltk** library of Python. The sequence of processing is as follows:

- **Lower case** - Converting the flow of text all into lower case.
- **Removing Stop words** - Removing words like I, me, etc. which don't add any extra meaning to the sentence.
- **Punctuation removal** - Removing comma, full stop, apostrophe, etc.

```
paragraphs = []
for paragraph in response.css("p"):
  # Get text content
  paragraph_text = paragraph.css("::text").getall()
  # Join paragraph text parts and clean
  paragraph_text = self.clean_paragraph_text("".join(paragraph_text))
  paragraphs.append(paragraph_text)
  # paragraphs.append("".join(paragraph_text))

# Create a dictionary for the city data (if paragraphs and image exist)
if paragraphs:
  city_data = {
    "city": city_name,
    "paragraphs": " ".join(paragraphs).strip()
  }
  yield city_data
```

Fig. 1. Script for extracting data from the web page

```
# Preprocess text descriptions
def preprocess_text(text):
    text = text.lower()  # Lowercase
    pattern = r'\(((?!(mw-parser-output|English|Bengali pronunciation|Kannada pronunciation|IPA|Gujarati|Hindi))[^)]+\)'
    text = re.sub(pattern, '', text)
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    # Remove stop words
    stop_words = set(stopwords.words('english'))
    lemmatizer = WordNetLemmatizer()

    words = word_tokenize(text)
    # Lemmatize and remove stop words
    filtered_words = [lemmatizer.lemmatize(word) for word in words if word not in stop_words]
    text = ' '.join(filtered_words)

    return text

# Define a function to apply preprocess_text to each row in parallel
def parallel_preprocess_text(text):
    return preprocess_text(text)
```

Fig. 2. Function to pre-process text

- **Tokenisation** - Converting each word in a sentence into an entry in a list.
- **Lemmatisation** - Grouping together multiple forms of a word as a single word.

III. Generating Text Embeddings and Computing Similarities

Once the data pre-processing step is complete, we had to convert text to numbers. The popular model to retrieve embeddings of text in the recent times are **Large Language Models(LLMs)**. In this project, we first used the pre-trained **BERT** model available on the HuggingFace platform to obtain the embeddings. **BERT (Bidirectional Encoder Representations from Transformers)** is a powerful machine learning framework designed for Natural Language Processing (NLP) tasks. It is pre-trained on a massive dataset of text and code, allowing it to learn a deep understanding of language in general. An embedding of 768 dimension was obtained for every text.

After obtaining the embeddings, we found the top 10 European cities similar to every Indian city using the cosine similarity metric, as shown in figure (3)

IV. Improving the Recommendations using Contrastive Learning

Once we had a basic model to obtain the similarities and recommend European cities, the next step was to improve the city's representation, i.e., the text embedding. We decided that images can be used to represent each of our cities, and so we

Fig. 3. Similarity score for some of the Indian cities

searched and collected 5 images for every city. But we had to make sure that the image embeddings and text embeddings for the same city are as close as possible and that for different cities are as far as possible. The text embeddings must be enhanced to represent the city better. This is where we made use of the concept of Contrastive Learning, inspired from the **Contrastive Language-Image Pretraining(CLIP)** model of OpenAI.

A. Using CLIP

CLIP (Contrastive Language-Image Pre-training) is a powerful model developed by OpenAI that learns to understand images and text in a shared space. The groundbreaking novel aspect of CLIP is that unlike its predecessors, it does not have to rely on massive, expensive datasets with manual labels, but instead learns from the wealth of text descriptions that already exist alongside images online. By leveraging contrastive learning, CLIP learns to associate similar semantics across diverse image-text pairs while discerning dissimilarities. This innovative approach enables CLIP to grasp nuanced relationships between visual and textual information, showcasing remarkable generalization, robustness, and adaptability across various downstream tasks.

B. Retrieving Text Encoding

Text encodings are retrieved in the same way as before, using the pretrained BERT model. The pre-trained BERT model is set to evaluation mode as it disables dropout layers and speeds up computations.

C. Retrieving Image Encoding

The encoding of every image is retrieved using the pretrained **ResNet50** Convolutional Neural Network(CNN) model. The model is modified by replacing the last classification layer of the with an identity layer. This effectively removes the classification head and keeps the feature extraction capabilities of the ResNet model.

D. Projection using FNN

We make use of a projection network to scale down the outputs from the BERT and ResNet models to the same dimension. A feed forward neural network with 3 layers is being used, with the first and third layers being Linear layers and the second layer is a ReLU activation layer.

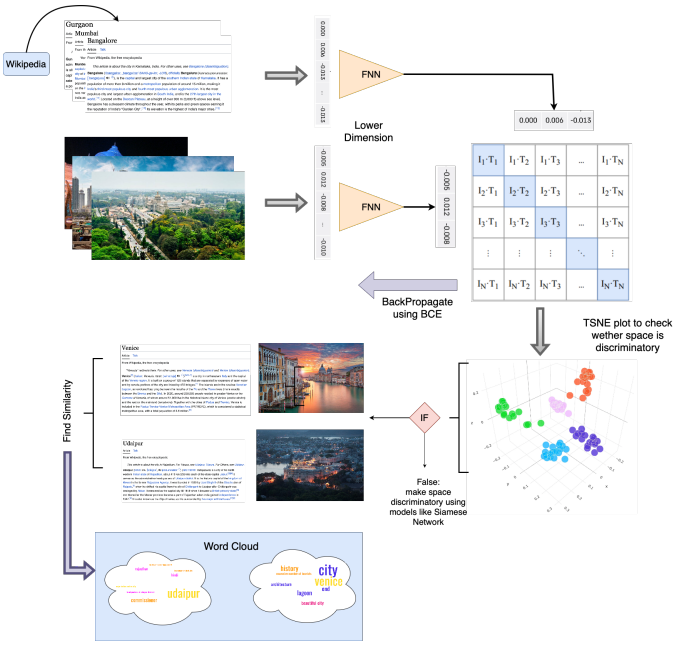

Fig. 4. Modified CLIP model to handle multiple images for each city

```python
class TextEncoder(nn.Module):
    def __init__(self, bert_model_name=bert_model_name):
        super(TextEncoder, self).__init__()
        # Load a pre-trained BERT model
        self.bert = BertModel.from_pretrained(bert_model_name)
        self.bert = self.bert.eval()

        config = BertConfig.from_pretrained(bert_model_name)

        # Get the number of units in the last layer
        self.out_features = config.hidden_size

    def forward(self, x):
        # Encode text using BERT tokenizer
        input_ids = x['input_ids']
        attention_mask = x['attention_mask']
        with torch.no_grad():
            output = self.bert(input_ids=input_ids.squeeze(1), attention_mask=attention_mask.squeeze(1))
        pooled_output = output.pooler_output
        return pooled_output
```

Fig. 5. Text encoder using the pre-trained BERT model

```python
class ImageEncoder(nn.Module):
  def __init__(self, pretrained_model_name="resnet50"):
    super(ImageEncoder, self).__init__()
    # Load a pre-trained image classification model (e.g., ResNet50)
    self.resnet = getattr(torchvision.models, pretrained_model_name)(pretrained=True)
    self.out_features = self.resnet.fc.in_features  # Assuming fc is the last layer
    # Remove the last classification layer
    self.resnet.fc = nn.Identity()

  def forward(self, x):
    # Pass image through the pre-trained model (excluding classification layer)
    x = x.view(-1, x.size(2), x.size(3), x.size(4))
    x = self.resnet(x)
    return x
```

Fig. 6. Image Encoder using the pre-trained ResNet model

```python
class ProjectionNetwork(nn.Module):
  def __init__(self, input_dim, hidden_dim, output_dim):
    super(ProjectionNetwork, self).__init__()
    self.fc1 = nn.Linear(input_dim, hidden_dim)
    self.relu = nn.ReLU(inplace=True)
    self.fc2 = nn.Linear(hidden_dim, output_dim)

  def forward(self, x):
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    return x
```

Fig. 7. Feed Forward Neural network for projection

```python
class CityDataset(Dataset):
    def __init__(self, data, image_dir):
        self.data = data
        self.image_dir = Path(image_dir)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        # Load image
        city_name = self.data.loc[idx, 'city']
        image_folder = self.image_dir / city_name  # Path to the city folder

        # Get all image file paths in the folder
        image_paths = list(image_folder.glob("*.jpg")) + list(image_folder.glob("*.jpeg")) +  list(image_folder.glob("*.png")) + list(image_folder.glob("*.webp")) + l
        #print('All images: ', image_paths)

        # Check if any image found
        if not image_paths:
            raise FileNotFoundError(f"No image found in folder: {image_folder}")

        images = []
        for path in image_paths[:5]:
            image = Image.open(str(path)).convert('RGB')  # Ensure RGB format

            # Apply transformations using your existing transform function
            image = transform(image)
            images.append(image)

        # Load and encode text
        tokenizer = BertTokenizer.from_pretrained(bert_model_name)
        text = self.data.loc[idx, 'text']
        encoded_text = tokenizer(text, padding="max_length", truncation=True, return_tensors="pt")

        # Get city name
        city_name = self.data.loc[idx, 'city']

        return torch.stack(images), encoded_text, city_name
```

Fig. 8. Dataset Class

E. Preparing the data

The **Dataset** class from PyTorch is inherited to create a custom dataset class. For each city in our dataset, we read the 5 images from the respective folder, and apply the transformations such as resizing, center cropping and normalizing the pixel values of the image tensor. We then encode the preprocessed text of the city into a format suitable for the model using the BERT Tokenizer.

Next, we split the data into training and test data, where 20% of the data is kept for testing. Then we use the DataLoader class to manage the process of loading data in batches of 10 and shuffling the data order for training. Using batches improves training efficiency by utilizing the capabilities of modern GPUs that can handle multiple data points simultaneously.

F. Training the model

During training, the data loader will fetch data points from the training dataset in batches of 10. Then a forward pass is performed through the model with the current batch of images and text data. During the forward pass, the images and text are first encoded using the pre-trained models and then projected to a lower dimension using the projection networks. Then the contrastive loss is calculated as follows:

- First the logits (unnormalized scores) are calculated between each text embedding and all image embeddings using matrix multiplication.

```python
class ContrastiveModel(nn.Module):
  def __init__(self, image_encoder, text_encoder, projection_dim, temperature=0.7):
    super(ContrastiveModel, self).__init__()
    self.image_encoder = image_encoder
    self.text_encoder = text_encoder
    self.image_projection = ProjectionNetwork(image_encoder.out_features, 512, projection_dim)
    self.text_projection = ProjectionNetwork(text_encoder.out_features, 512, projection_dim)
    self.temperature = temperature

  def forward(self, images, text):
    # Encode image and text
    #image_features = self.image_encoder(image)
    image_features = self.image_encoder(images)
    text_features = self.text_encoder(text)

    # Project embeddings to a lower dimension
    image_projections = self.image_projection(image_features)
    text_projections = self.text_projection(text_features)
    text_projections = text_projections.repeat(5, 1)

    image_embeddings = image_projections / torch.norm(image_projections, dim=1, keepdim=True)
    text_embeddings = text_projections / torch.norm(text_projections, dim=1, keepdim=True)

    # Calculating the Loss
    logits = (text_embeddings @ image_embeddings.T) / self.temperature
    # print('Logits: ', logits)
    images_similarity = image_embeddings @ image_embeddings.T
    texts_similarity = text_embeddings @ text_embeddings.T
    targets = (images_similarity + texts_similarity) / 2 * self.temperature
    # print('Target: ', targets)
    texts_loss = cross_entropy(logits, targets, reduction='none')
    images_loss = cross_entropy(logits.T, targets.T, reduction='none')
    loss =  (images_loss + texts_loss) / 2.0 # shape: (batch_size)
    # print('Batch loss: ', loss.mean())
    return loss.mean()
```

Fig. 9. Model Definition

```python
def train_model(model, train_dataloader, test_dataloader, optimizer, num_epochs, accumulation_steps=1):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    train_loss_list = []
    test_loss_list = []

    for epoch in range(num_epochs):
        print(f"Epoch: {epoch+1}/{num_epochs}")
        running_loss = 0.0
        accumulated_steps = 0  # Counter for accumulated steps

        for batch_idx, (images, texts, city_names) in enumerate(train_dataloader):
            images = images.to(device)
            texts = texts.to(device)

            loss = model(images, texts)
            # Backward pass
            loss.backward()

            # Accumulate gradients
            accumulated_steps += 1

            if accumulated_steps == accumulation_steps:
                # Update weights after accumulation_steps batches
                optimizer.step()
                optimizer.zero_grad()
                accumulated_steps = 0  # Reset accumulated_steps

            running_loss += loss.item()

        # Print average training loss per epoch
        epoch_loss = running_loss / len(train_dataloader)
        train_loss_list.append(epoch_loss)
        print(f"Training loss: {epoch_loss:.4f}")

        # Evaluation on test set
        with torch.no_grad():  # Disable gradient calculation for test phase
            test_loss = 0.0
            for images, texts, city_names in test_dataloader:
                images = images.to(device)
                texts = texts.to(device)
                test_loss += model(images, texts).item()

            test_loss /= len(test_dataloader)
            test_loss_list.append(test_loss)
            print(f"Test loss: {test_loss:.4f}")

    # Save the model after training
    #torch.save(model.state_dict(), "model_weights.pth")
    torch.save(model, "model_20ep_500.pth")
    print("Model saved")
    plot_graph(train_loss_list, test_loss_list, epoch+1)
```

Fig. 10. Function to train the model

- Two similarity matrices are calculated, one between all image embeddings and the other between all text embeddings using matrix multiplication.
- A target matrix is constructed that represents the desired similarity scores between each text representation and its corresponding image(s). The diagonal elements of this matrix should ideally be high, indicating high similarity between an image and its corresponding text.
- The corss entropy loss is calculated between the logits and the target similarity matrix from two perspectives, from the image perspective and the text perspective. Their average represents the overall contrastive loss for each data point in the batch.

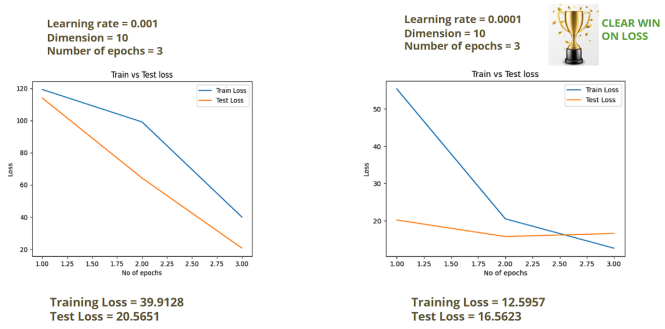

Fig. 11. Tuning the learning rate

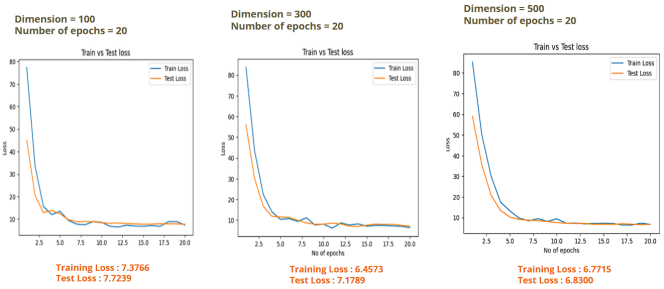

Fig. 12. Tuning the projetion dimension

G. Hyperparameter Tuning

We have various hyperparameters in our model, which influence the performance of our model. It is not easy to fix a single value to them and then directly get on with the training. Here, we have considered 2 parameters - learning rate and dimension of the projection.

1) Learning Rate: A good learning rate balances the speed and stability of learning. Too slow, and the model takes forever to learn. Too fast, and it might miss the optimal solution or become unstable. Here, we considered two leaning rates - 0.001 and 0.0001. We ran the model for the above learning rates while keeping the number of epochs constant. We observed that the loss was lower when the learning rate is 0.0001.

2) Dimension of the Projection: An ideal projection dimension is necessary to not lose the important information. We ran the model with 100, 300 and 500 dimensions, while keeping the number of epochs as 20, and the learning rate as 0.0001. Even though the difference is not very significant, we chose the dimension as 500, as that had the least loss and to not hinder the network's ability to learn important relationships.

The embeddings for each city were the concatenation of the output from the pre-trained text encoder and the output from the text projection network.

H. Analysing the embeddings

Once the training was completed, the embeddings for all the cities were obtained and stored. In order to analyze how good or bad the embeddings were, we used the technique of t-SNE, which is a popular technique used for visualizing

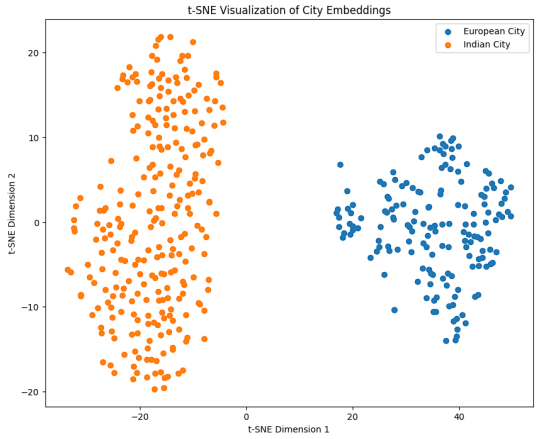

Fig. 13. Visualizing the embeddings using t-SNE

Modified CLIP Model		
Rank	City	Similarity
1	Sheffield	0.8454
2	Slough	0.8356
3	Leeds	0.8334
4	Milton Keynes	0.8262

Cosine Similarity (w/o CLIP)		
Rank	City	Similarity
25	Sheffield	0.7848
32	Slough	0.7798
2	Leeds	0.8310
12	Milton Keynes	0.7940

Fig. 14. Comparison of using CLIP Model with w/o CLIP Model

high-dimensional data in a low-dimensional space. From the Figure 13, we observed that the inter class separability of the Indian and European cities is very high, thus ensuring that the model is working as expected. Had this not been the case, we would have to resort to some of the techniques such as metric learning, to make our embeddings more separable.

V. Results

After obtaining the final embeddings of all cities, we picked an Indian city and found out its similarity with every European city using Pearson Correlation Coefficient. The 4 european cities whose similarity score is highest are recommended for that particular Indian city. All these steps are performed for all the Indian cities in our dataset.

The pair which gave the highest similarity score was Ludhiana and Sheffield with a score of 0.8454. The other 3 european cities recommended for Ludhiana were Slough, Leeds and Milton Keynes. Figure 14 shows the rise in the similarity score after applying CLIP model as well a rise in the ranking of the cities. For analysis, we created word clouds of the cities to analysis whether an European city has been correctly matched to an Indian city and on which cultural features.

From Figure 15, we can see that some common words between Ludhiana and Sheffield are university, college, school, education, industry, manchester, stadium, airport, hospital and medical. These 2 cities are matched on the Center of Education. Actually Sheffield is a center of education and it has 2 world class univs - The University of Sheffield and Hallam University. Ludhiana has 363 senior secondary, 367 high, 324 middle, 1129 primary schools. So, if we want to improve and develop education in Ludhiana, our model recommends to learn from Sheffield.

Fig. 15. Word Clouds of Ludhiana and Sheffield

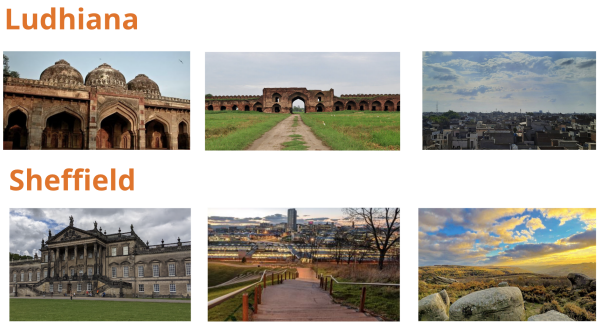

Fig. 16. Images of Ludhiana and Sheffield

We observed that the images of Ludhiana and Sheffield have a pinch of similarity (shown in Figure 16) which has helped us infer how the similarity came so high.

The next in line recommendation for Ludhiana was Slough and the word clouds (Figure 17) depict the following similar words - industrial, manchester, office, factory, business, company, engineering, railway, station, roads, etc. These 2 cities are matched on the Industrial estates. Slough is the largest industrial estate of Europe providing 17,000 jobs cross 400 businesses. Ludhiana is known for its textile industry. So, if we want to improve and develop industries and market in Ludhiana, our model recommends to learn from Slough.

VI. Limitations

The lack of computing resources was the major limitation. The non-availability of more GPU resources restricted us to run the model for only 20 epochs, and used a three-layer architecture for the projection network. We were restricted to train only the projection networks, which itself took around 6 hours with the computing power we had.

Fig. 17. Word cloud of Slough

VII. Future Scope

The following improvements can be a part of the future scope:

- The number of images for each city can be increased, in order to improve the city representations.
- Currently, the only source of text for each city was the Wikipedia page. It is quite natural for some data to be missing, hence we can curate the text for the city from multiple sources of text.
- The text and image encoders could also be fine tuned for this specific task, rather than training just the projection networks.
- The project can be extended to incorporate more foreign cities, as required.

References

[1] Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., … Sutskever, I. (2021), "Learning transferable visual models From natural language supervision," In Proceedings of Machine Learning Research (Vol. 139, pp. 8748–8763). ML Research Press.

[2] Language Agnostic BERT sentence encoder from HuggingFace platform, https://huggingface.co/setu4993/LaBSE

[3] Simple Implementation of OpenAI CLIP model: A Tutorial, https://towardsdatascience.com/simple-implementation-of-openai-clip-model-a-tutorial-ace6ff01d9f2