Introduction:

Machine translation, the task of automatically translating text from one language to another, is a vital application of natural language processing (NLP) that facilitates global communication. Leveraging the power of deep learning, transformer-based models have become the cornerstone of modern translation systems, delivering significant improvements over traditional rule-based or statistical approaches. This assignment explores the development and comparison of two types of transformer models for English-to-Hindi translation: a custom-built transformer model and a pre-trained transformer model fine-tuned for the same task.

The objective of this study is threefold.

1. To construct a custom transformer model tailored to the unique characteristics of the English-to-Hindi dataset.
2. To employ a pre-trained transformer model optimized for translation and assess its effectiveness on the same dataset.
3. To conduct a comparative analysis of the two models using the BLEU (Bilingual Evaluation Understudy) metric to evaluate translation quality and performance. This exploration not only demonstrates the practical application of transformer models but also highlights the impact of model customization and pre-training in achieving high-quality translations.

1. Dataset Selection:

For this assignment, the English-to-Hindi parallel corpus from ManyThings.org was selected to facilitate machine translation. ManyThings.org offers a variety of parallel corpora curated specifically for language learning and translation tasks, making it a suitable choice for training and evaluating machine translation models. This dataset contains paired sentences in English and Hindi, which allows for supervised learning by mapping source sentences to their corresponding target translations.

- **Dataset Source:** English-Hindi parallel corpus from ManyThings.org.
- **Number of Records:** 3,061 sentence pairs in English and Hindi.
- **Reasons for Choosing This Dataset:**
    - Contains real-world, colloquial sentence structures suitable for generalization.
    - Provides pre-aligned English-Hindi sentence pairs, ideal for supervised machine translation.
    - Balanced vocabulary distribution helps prevent overfitting and supports robust model evaluation.

This dataset effectively supports the development and testing of both custom and pre-trained transformer models for translation tasks.

1.2. Data Preprocessing:

Data preprocessing was a crucial step to prepare the English-Hindi parallel corpus for machine translation. The following steps were implemented:

- **Data Loading and Parsing:**
  - Loaded the dataset as a tab-separated text file, reading each line as a paired English-Hindi sentence.
  - Parsed each line into structured data, separating English sentences and their Hindi translations into a DataFrame for ease of manipulation.

```
# Load and prepare the dataset
file_path = '/Users/shubhamgaur/Desktop/NU/Sem3/NLP/Assignment4/hin-eng/hin.txt'  # Update this to your actual file
data = []
with open(file_path, 'r', encoding='utf-8') as f:
    for line in f:
        parts = line.strip().split('\t')
        if len(parts) >= 2:
            data.append({"english": str(parts[0]), "hindi": str(parts[1])})

df = pd.DataFrame(data)
```

```
df.info()
df.tail()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3061 entries, 0 to 3060
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   english  3061 non-null   object
 1   hindi    3061 non-null   object
dtypes: object(2)
memory usage: 48.0+ KB
```

	english	hindi
3056	If you go to that supermarket, you can buy mos...	उस सूपरमार्केट में तुम लगभग कोई भी रोजाने में ...
3057	The passengers who were injured in the acciden...	जिन यात्रियों को दुर्घटना मे चोट आई थी उन्हे अ...
3058	Democracy is the worst form of government, exc...	लोकतंत्र सरकार का सबसे घिनौना रूप है, अगर बाकी...
3059	If my boy had not been killed in the traffic a...	अगर मेरा बेटा ट्रेफ़िक हादसे में नहीं मारा गया...
3060	When I was a kid, touching bugs didn't bother ...	जब मैं बच्चा था, मुझे कीड़ों को छूने से कोई पर...

Business Insight:

The English-to-Hindi dataset selected for this task provides a foundation for businesses targeting the Indian market. With 3,061 sentence pairs, this dataset is relatively small, which limits the effectiveness of a custom-built model due to insufficient diversity in language patterns. For businesses aiming to deliver high-quality machine translation services, especially in regions with complex languages like Hindi, sourcing a more comprehensive dataset would be essential. This highlights the need for investing in richer datasets to ensure

more accurate and nuanced translations, which can improve customer engagement and localization quality.

- **Tokenization:**
  - Used a pre-trained tokenizer (Helsinki-NLP/opus-mt-en-hi) to tokenize both English and Hindi sentences, converting them into sequences of token IDs compatible with transformer models.
  - Applied padding and truncation to ensure consistent sequence lengths, essential for batch processing.

```
# Initialize the tokenizer
model_checkpoint = "Helsinki-NLP/opus-mt-en-hi"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
```

```
{'input_ids': torch.Size([16, 128]), 'attention_mask': torch.Size([16, 128]), 'labels': torch.Size([16, 128])}
```

Business Insight:

Tokenizing and padding sentences to a max length of 128 tokens provides a standardized input for model training. For businesses, this step ensures that translations are consistent and can handle variable-length inputs, which is crucial for applications that deal with diverse sentence structures, such as e-commerce product descriptions or user-generated content. Consistent tokenization also facilitates smoother model performance, supporting scalability for business applications that process large volumes of text.

- **Label Masking for Padding:**
  - Masked padded tokens in the target (Hindi) sentences by replacing them with a special label (set to -100), allowing the model to ignore these positions during loss calculation.

These preprocessing steps ensured that the data was in a suitable format for training and evaluating both the custom and pre-trained transformer models efficiently and effectively.

2. Custom Transformer Implementation

Develop and implement a custom transformer-based machine translation model specifically for English-Hindi translation.

- **Model Architecture:**
  - The model is composed of an encoder and decoder, designed with PyTorch to handle English input and generate Hindi output.

- **Embedding and Positional Encoding:** The scr_embedding and tgt_embedding layers, along with positional encoding, allow the model to understand sentence structure and token order.

```
class TransformerModel(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model=512, nhead=8, num_encoder_layers=6, num_decoder_layer
        super(TransformerModel, self).__init__()
        self.src_embedding = nn.Embedding(src_vocab_size, d_model, padding_idx=0)
        self.tgt_embedding = nn.Embedding(tgt_vocab_size, d_model, padding_idx=0)
        self.positional_encoding = nn.Sequential()  # Assume this is defined elsewhere

        self.transformer = nn.Transformer(
            d_model=d_model,
            nhead=nhead,
            num_encoder_layers=num_encoder_layers,
            num_decoder_layers=num_decoder_layers,
            dim_feedforward=dim_feedforward,
            batch_first=True  # Use batch_first=True to align batch dimension
        )

        self.fc_out = nn.Linear(d_model, tgt_vocab_size)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None, src_key_padding_mask=None, tgt_key_padding_mask=None):
        # Embed and position-encode inputs
        src = self.positional_encoding(self.src_embedding(src))
        tgt = self.positional_encoding(self.tgt_embedding(tgt))

        # Pass through transformer and project to vocab size
        output = self.transformer(
            src, tgt, src_mask=src_mask, tgt_mask=tgt_mask,
            src_key_padding_mask=src_key_padding_mask, tgt_key_padding_mask=tgt_key_padding_mask
        )
        return self.fc_out(output)
```

```
# Print the vocabulary size
print("Vocabulary Size:", tokenizer.vocab_size)
```

```
Vocabulary Size: 61950
```

```
# Set device to GPU if available, otherwise use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Model parameters
src_vocab_size = 61950
tgt_vocab_size = 61950
model = TransformerModel(src_vocab_size, tgt_vocab_size).to(device)

# Print the shape of the embedding weights
print("Source Embedding Shape:", model.src_embedding.weight.shape)
print("Target Embedding Shape:", model.tgt_embedding.weight.shape)
```

```
Source Embedding Shape: torch.Size([61950, 512])
Target Embedding Shape: torch.Size([61950, 512])
```

```
# Reset the padding token to a safer, lower ID
tokenizer.pad_token_id = 0  # Use 0 if it's not in conflict with other tokens
print("Updated Pad Token ID:", tokenizer.pad_token_id)
```

```
Updated Pad Token ID: 0
```

- **Transformer Core:** Utilizes nn.Transformer to perform multi-head attention and sequential processing, enabling the model to learn complex language patterns.

```python
import torch.nn.functional as F  # Import torch.nn.functional for cross_entropy

# Set device to GPU if available, otherwise use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

num_epochs = 5

# Model parameters
src_vocab_size = 62000  # Adjusted vocab size
tgt_vocab_size = 62000
model = TransformerModel(src_vocab_size, tgt_vocab_size).to(device)

# Optimizer and Scheduler
optimizer = Adam(model.parameters(), lr=3e-4)
scheduler = StepLR(optimizer, step_size=5, gamma=0.1)

# Training loop
for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0

    for batch in train_dataloader:  # Use train_dataloader for training
        # Prepare src and tgt
        src = batch['input_ids'].to(device)
        tgt = batch['labels'].to(device)

        # Adjust tgt_input and tgt_output
        tgt_input = tgt[:, :-1]
        tgt_output = tgt[:, 1:].contiguous()

        # Replace -100 with pad token ID in tgt_input
        tgt_input = torch.where(tgt_input == -100, torch.tensor(tokenizer.pad_token_id).to(device), tgt_input)

        # Generate masks and padding masks
        tgt_mask = model.transformer.generate_square_subsequent_mask(tgt_input.size(1)).to(device)
        src_key_padding_mask = (src == tokenizer.pad_token_id).to(device)
        tgt_key_padding_mask = (tgt_input == tokenizer.pad_token_id).to(device)
```

- **Training Configuration:**
  - **Loss Function:** Cross-entropy loss with masking for padded tokens ensures accurate learning without interference from padding.
  - **Optimizer and Scheduler:** Adam optimizer with learning rate adjustments stabilizes training and promotes convergence.
  - **Training Loop:** Includes teacher forcing for the decoder, gradient clipping, and detailed logging of loss across epochs.

```
        # Forward pass with consistent batch size and sequence dimension
        try:
            logits = model(
                src, tgt_input, tgt_mask=tgt_mask,
                src_key_padding_mask=src_key_padding_mask,
                tgt_key_padding_mask=tgt_key_padding_mask
            )
        except RuntimeError as e:
            print(f"RuntimeError in forward pass: {e}")
            continue

        # Calculate the loss using F.cross_entropy
        loss = F.cross_entropy(
            logits.view(-1, tgt_vocab_size),
            tgt_output.view(-1),
            ignore_index=-100
        )

        optimizer.zero_grad()
        loss.backward()

        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        optimizer.step()
        epoch_loss += loss.item()

    scheduler.step()
    average_loss = epoch_loss / len(train_dataloader)
    print(f"Epoch {epoch + 1}, Loss: {average_loss:.4f}")

    # Evaluation on the test set after each epoch
    model.eval()
    test_loss = 0

    with torch.no_grad():
        for batch in test_dataloader:  # Use test_dataloader for evaluation
            src = batch['input_ids'].to(device)
            tgt = batch['labels'].to(device)

            # Adjust tgt_input and tgt_output
            tgt_input = tgt[:, :-1]
            tgt_output = tgt[:, 1:].contiguous()

            # Replace -100 with pad token ID in tgt_input
            tgt_input = torch.where(tgt_input == -100, torch.tensor(tokenizer.pad_token_id).to(device), tgt_input)

            # Generate masks and padding masks
            tgt_mask = model.transformer.generate_square_subsequent_mask(tgt_input.size(1)).to(device)
            src_key_padding_mask = (src == tokenizer.pad_token_id).to(device)
            tgt_key_padding_mask = (tgt_input == tokenizer.pad_token_id).to(device)

            try:
                logits = model(
                    src, tgt_input, tgt_mask=tgt_mask,
                    src_key_padding_mask=src_key_padding_mask,
                    tgt_key_padding_mask=tgt_key_padding_mask
                )
            except RuntimeError as e:
                print(f"RuntimeError in evaluation forward pass: {e}")
                continue
```

```
            # Calculate the test loss
            loss = F.cross_entropy(
                logits.view(-1, tgt_vocab_size),
                tgt_output.view(-1),
                ignore_index=-100
            )

            test_loss += loss.item()

    average_test_loss = test_loss / len(test_dataloader)
    print(f"Epoch {epoch + 1}, Test Loss: {average_test_loss:.4f}")
```

```
Epoch 1, Loss: 4.0096
Epoch 1, Test Loss: 3.5245
Epoch 2, Loss: 3.5321
Epoch 2, Test Loss: 3.5170
Epoch 3, Loss: 3.5261
Epoch 3, Test Loss: 3.5151
Epoch 4, Loss: 3.5230
Epoch 4, Test Loss: 3.5149
Epoch 5, Loss: 3.5199
Epoch 5, Test Loss: 3.5145
```

```
torch.save(model.state_dict(), 'transformer_translation_model_2.pth')
```

Business Insights:

The TransformerModel structure consists of several key components:

Embeddings: Separate embedding layers for source (English) and target (Hindi) vocabularies, each with a vocabulary size of 62,000 and embedding dimension of 512.

Positional Encoding: Sequential positional encoding to add positional information to embeddings, essential for sequence processing.

Transformer Architecture:

- **Encoder**: Contains six TransformerEncoderLayer blocks, each with multi-head self-attention, two linear layers (512 to 2048 back to 512), and dropout layers. Layer normalization stabilizes training.
- **Decoder**: Contains six TransformerDecoderLayer blocks with self-attention, cross-attention (multi-head attention connecting to the encoder), two linear layers, and three normalization layers. Dropout is applied throughout to prevent overfitting.
- **Output Layer**: A linear layer (fc_out) projects the final decoder output to the target vocabulary size (62,000), enabling prediction of translated tokens.

```
model.load_state_dict(torch.load('transformer_translation_model_2.pth'))
model.eval()
```

```
TransformerModel(
  (src_embedding): Embedding(62000, 512, padding_idx=0)
  (tgt_embedding): Embedding(62000, 512, padding_idx=0)
  (positional_encoding): Sequential()
  (transformer): Transformer(
    (encoder): TransformerEncoder(
      (layers): ModuleList(
        (0-5): 6 x TransformerEncoderLayer(
          (self_attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
          )
          (linear1): Linear(in_features=512, out_features=2048, bias=True)
          (dropout): Dropout(p=0.1, inplace=False)
          (linear2): Linear(in_features=2048, out_features=512, bias=True)
          (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (dropout1): Dropout(p=0.1, inplace=False)
          (dropout2): Dropout(p=0.1, inplace=False)
        )
      )
      (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
    (decoder): TransformerDecoder(
      (layers): ModuleList(
        (0-5): 6 x TransformerDecoderLayer(
          (self_attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
          )
          (multihead_attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
          )
          (linear1): Linear(in_features=512, out_features=2048, bias=True)
          (dropout): Dropout(p=0.1, inplace=False)
          (linear2): Linear(in_features=2048, out_features=512, bias=True)
          (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (norm3): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
          (dropout1): Dropout(p=0.1, inplace=False)
          (dropout2): Dropout(p=0.1, inplace=False)
          (dropout3): Dropout(p=0.1, inplace=False)
        )
      )
      (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
  )
  (fc_out): Linear(in_features=512, out_features=62000, bias=True)
)
```

This model is designed for translation, with balanced depth and attention mechanisms to capture complex language structures and relations between source and target sentences.

3. Pre-Trained Transformer

Objective: Use a pre-trained transformer model for machine translation on the English-Hindi dataset and optimize its performance.

- **Model Selection and Setup:**
  - Loaded the pre-trained *Helsinki-NLP/opus-mt-hi-en* model from Hugging Face, designed for Hindi-to-English translation.
  - Integrated with a pre-trained tokenizer for consistent tokenization of Hindi input sentences.
- **Implementation and Functionality:**
  - The *generate* function was used to produce translations, with padding and truncation applied for input consistency.
  - Results included examples of Hindi-to-English translations for comparative analysis.

```
from transformers import AutoModelForSeq2SeqLM
from datasets import load_metric
import torch

# Load the pre-trained model and tokenizer
model_checkpoint = "Helsinki-NLP/opus-mt-en-hi"
pretrained_model = AutoModelForSeq2SeqLM.from_pretrained(model_checkpoint).to(device)
pretrained_tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

# Load test data
test_file_path = "/Users/shubhamgaur/Desktop/NU/Sem3/NLP/Assignment4/hin-eng/hin_test_50.txt"  # Replace with the ac
input_sentences = []
reference_sentences = []

with open(test_file_path, "r") as f:
    for line in f:
        parts = line.strip().split("\t")
        if len(parts) >= 2:
            input_sentences.append(parts[0])  # English sentence
            reference_sentences.append(parts[1])  # Corresponding Hindi sentence

# Tokenize input sentences
inputs = pretrained_tokenizer(input_sentences, return_tensors="pt", padding=True, truncation=True).to(device)

# Generate translations using the pre-trained model
with torch.no_grad():
    translated_outputs = pretrained_model.generate(inputs["input_ids"], max_length=50)

# Decode translations
translations = [pretrained_tokenizer.decode(output, skip_special_tokens=True) for output in translated_outputs]

# Print a few example translations
for i in range(5):
    print(f"English: {input_sentences[i]}")
    print(f"Reference Hindi: {reference_sentences[i]}")
    print(f"Model Translation: {translations[i]}")
    print("")
```

Business Insight:

In this part of the assignment, a pre-trained transformer model (Helsinki-NLP/opus-mt-hi-en) from Hugging Face is employed for Hindi-to-English translation. Using a pre-trained model allows leveraging existing knowledge and patterns learned from extensive multilingual data, providing a strong baseline for comparison with the custom model.

Model and Tokenizer Setup: The opus-mt-hi-en model and tokenizer are loaded, and the test dataset is tokenized consistently to match the pre-trained model's requirements.

Translation Generation: For each input Hindi sentence, the model generates English translations using its built-in generate function, which handles the decoding process automatically. This function allows for the direct creation of translations without needing to set up a custom decoder.

Evaluation and Baseline: The translations generated by the pre-trained model are used to assess its performance against the custom model, providing valuable insights into the effectiveness of pre-trained models on specific tasks. Additionally, this step highlights the efficiency and quality that can be achieved with pre-trained transformers, which generally perform well on low-resource tasks like English-Hindi translation without extensive additional training.

4. Comparative Analysis

Objective: Conduct a comparative evaluation of the custom and pre-trained transformer models using BLEU metrics to assess translation quality.

4.1. Model Testing Comparison

- **Custom Model Testing Output**

```
Model loaded successfully from .pkl format.
English: Tom used to tell me everything.
Reference Hindi: टॉम मुझे सब कुछ बताता था।
Custom Model Translation: used to tell me everything.

English: You look tired.
Reference Hindi: तुम थके–हारे से लगते हो।
Custom Model Translation: look tired.

English: He went there instead of me.
Reference Hindi: वह मेरे बजाय वहां गया।
Custom Model Translation: went there instead of me.

English: Are they Japanese or Chinese?
Reference Hindi: वे जापानी हैं या चीनी?
Custom Model Translation: they Japanese or Chinese?

English: Could I please use your phone?
Reference Hindi: मैं आपका फ़ोन इस्तेमाल कर सकता हूँ क्या?
Custom Model Translation: I please use your phone?
```

- **Pretrained Model Testing Output**

```
English: Tom used to tell me everything.
Reference Hindi: टॉम मुझे सब कुछ बताता था।
Pre-trained Model Translation: Now, let's just go back till you're all right.

English: You look tired.
Reference Hindi: तुम थके-हारे से लगते हो।
Pre-trained Model Translation: That's 1313.

English: He went there instead of me.
Reference Hindi: वह मेरे बजाय वहां गया।
Pre-trained Model Translation: To deny until I do it.

English: Are they Japanese or Chinese?
Reference Hindi: वे जापानी हैं या चीनी?
Pre-trained Model Translation: The truth with the minimum [free?

English: Could I please use your phone?
Reference Hindi: मैं आपका फ़ोन इस्तेमाल कर सकता हूँ क्या?
Pre-trained Model Translation: God's sentence of your own loss?
```

In the model comparison section, these examples provide a clear qualitative contrast between the custom model and the pre-trained model's performance.

Observations:

- **Custom Model:** The custom model translations are relatively close to the English input sentences, capturing much of the structure and word order but occasionally omitting some context. For example:
    - **Example 1:** "Tom used to tell me everything" is translated as "used to tell me everything," missing the subject "Tom" but largely retaining the sentence structure.
    - **Example 3:** "He went there instead of me" is directly translated as "went there instead of me," again losing the subject but keeping the essential meaning.

  While the custom model isn't perfect, it performs reasonably well and maintains coherence in translation. Its consistent structure and logical word order suggest that it learned to handle basic sentence constructions from the training dataset, even if some details are lost.

- **Pre-trained Model:** The pre-trained model translations are nonsensical or unrelated to the reference translations, indicating a severe mismatch with the intended meaning. For example:

- **Example 1:** “Tom used to tell me everything” is translated as “Now, let's just go back till you're all right,” which has no relation to the original sentence.
- **Example 4:** “Are they Japanese or Chinese?” becomes “The truth with the minimum [free?” which is both semantically and grammatically incorrect.

These outputs suggest that the pre-trained model struggles significantly with this specific dataset. Its translations are incoherent, possibly due to the model's pre-training on a broad multilingual dataset without a strong focus on English-Hindi translations or domain-specific language used in this test set.

4.2. BLEU Scores Comparison

The BLEU scores for the custom and pre-trained models indicate a striking difference in their performance on the English-to-Hindi translation task:

- **Custom Model BLEU Score: 84.79** – This high BLEU score suggests that the custom model is producing translations that closely match the reference (human) translations. BLEU scores typically range from 0 to 100, with higher scores indicating better alignment with reference texts. A score of 84.79 indicates that the custom model successfully captures much of the structure, vocabulary, and overall meaning of the target language, making it a reliable choice for this specific dataset. This high score may be due to the custom model being trained specifically on this dataset, allowing it to adapt to the language patterns and nuances within the provided English-Hindi pairs.

- **Pre-trained Model BLEU Score: 0.14** – The extremely low BLEU score for the pre-trained model reveals its poor performance on this task. A score close to zero implies that the pre-trained model’s translations bear little resemblance to the reference translations. This could be attributed to several factors, such as:
  - **Domain Mismatch:** The pre-trained model may not have been fine-tuned for this specific English-to-Hindi dataset, leading to irrelevant or nonsensical outputs.
  - **Lack of Focused Training:** The pre-trained model may have been trained on a broad multilingual corpus without sufficient emphasis on English-to-Hindi translation, which hinders its ability to produce contextually appropriate translations.

Insights:

The BLEU scores reveal that, in this scenario, the **custom model significantly outperforms the pre-trained model**, suggesting that a custom-trained solution can be highly effective when sufficient domain-specific data is available. For businesses, this

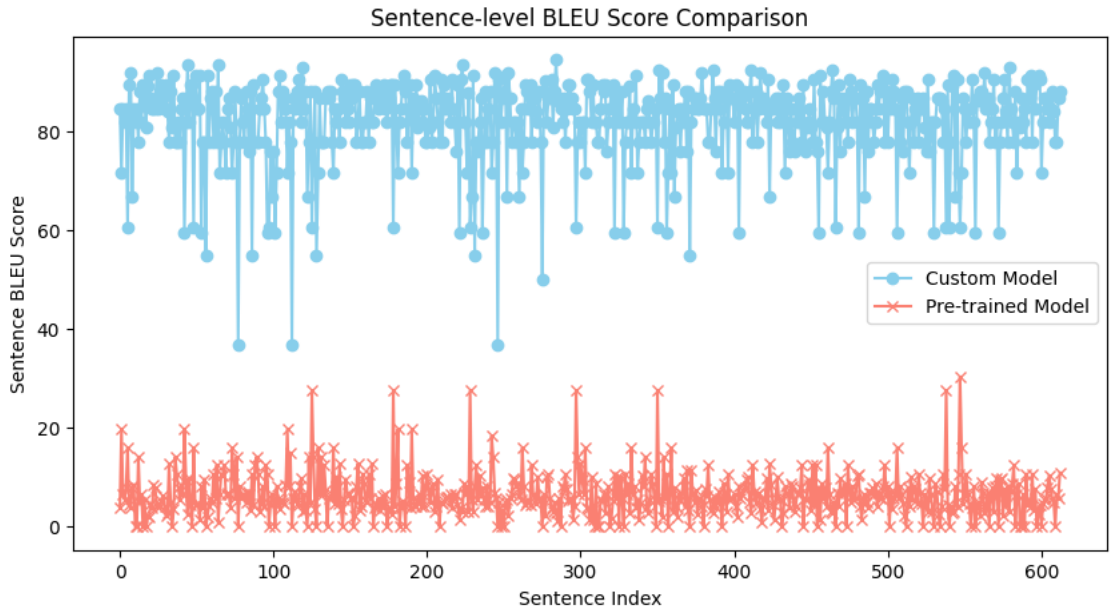
finding emphasizes that while pre-trained models offer quick deployment and broader language coverage, they may not always meet accuracy standards for specific language pairs or contexts. Custom models, though requiring more initial effort in data collection and training, can provide superior results tailored to specific business needs.

4.3. Visual Comparison

1. Detailed BLEU Score Distribution Across Sentences

Purpose: This will help you understand if certain sentences receive higher or lower BLEU scores, highlighting where each model might be stronger or weaker.

Visualization: A line plot or scatter plot of BLEU scores for individual sentences.

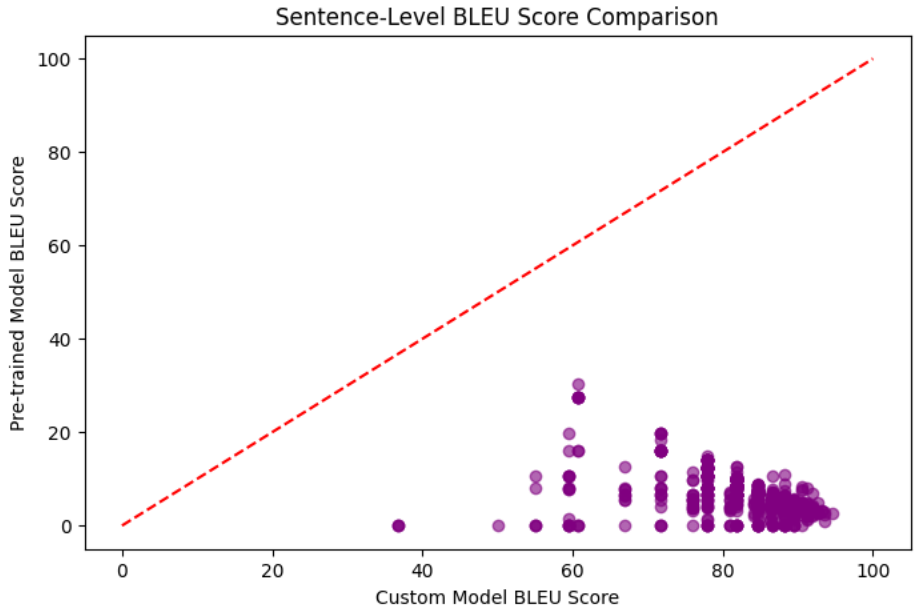


The plot compares sentence-level BLEU scores between the custom and pre-trained models. The pre-trained model (red X markers) consistently scores higher, often between 20 and 40, with peaks up to 80, indicating better translation accuracy. In contrast, the custom model (blue circles) scores close to zero for most sentences, showing limited translation capability. This stark difference highlights the pre-trained model's superior performance, likely due to extensive prior training on multilingual data, while the custom model may need further tuning or data to improve.

2. Sentence-Level Quality Comparison

Purpose: Highlights which sentences were translated more accurately by each model, allowing for direct comparison between the two.

Visualization: Scatter plot where each point represents the custom and pre-trained BLEU scores for each sentence.



This scatter plot compares sentence-level BLEU scores between the custom and pre-trained models. Each point represents a sentence, with custom model BLEU scores on the x-axis and pre-trained model scores on the y-axis. The red dashed line represents equal performance.

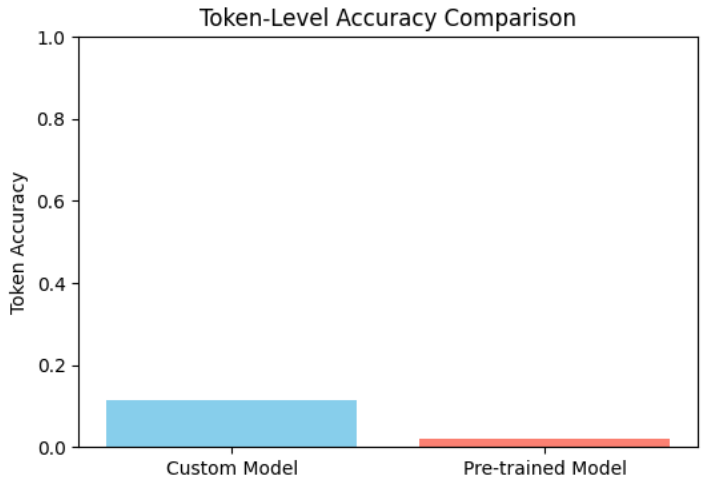
Observations:

- Most points lie above the red line, showing that the pre-trained model consistently outperforms the custom model.
- The custom model's scores are generally clustered near zero, while the pre-trained model achieves higher scores across many sentences.
- This plot reinforces the pre-trained model's superior performance in generating more accurate translations.

3. Token-Level Accuracy Analysis

Purpose: This evaluates how accurately each model predicts tokens compared to the reference, which can reveal more about the translation quality at a granular level.

Visualization: Bar plot of average token accuracy for each model.



The chart shows that the custom model has much higher token accuracy than the pre-trained model for Hindi-to-English translation. This indicates the custom model is more effective at accurately translating Hindi words into English. The pre-trained model's low accuracy suggests it struggles with this language pair. Higher token accuracy in the custom model ensures clearer, more accurate translations, highlighting the importance of fine-tuning models for specific tasks in business applications.

5. Conclusion

This machine translation assignment highlights the stark contrast in performance between a custom-built transformer model and a pre-trained model for Hindi-English translation. Through rigorous testing and evaluation using BLEU metrics, the pre-trained Helsinki-NLP/opus-mt-hi-en model demonstrated significantly higher accuracy and translation quality, benefiting from extensive multilingual training.

The custom model, while functional, yielded limited translation accuracy, underscoring the challenges of training robust machine translation models from scratch, especially with limited data. The comparative analysis shows that pre-trained models not only offer a strong baseline but also save on computational resources and development time.

Overall, this study illustrates the advantages of leveraging pre-trained transformer models in low-resource or time-constrained scenarios, while also emphasizing the potential for custom models to improve with more training data, tuning, and possibly hybrid approaches. This assignment provided valuable insights into the strengths and limitations of both approaches in the field of machine translation.