DEVELOPMENT OF A MULTILINGUAL CUSTOMER SERVICE CHATBOT

DEVELOPMENT OF A MULTILINGUAL CUSTOMER SERVICE CHATBOT

A MS Course Project Report submitted in partial
fulfillment of the requirements for the degree of
Master of Science in Computer Science

By

Suryakumar Selvakumar
SRM Valliammai Engineering College, Chennai, India, 2023
Bachelor of Science in Computer Science

May 2025
University of Colorado Denver

ABSTRACT

Today's globalized business environments demand the provision of consistent and efficient customer service across multiple languages, and companies around the world are struggling to provide it reliably. The aim of this project is to develop an efficient & seamless **Multilingual Single-turn Customer Service Chatbot** capable of accurately responding to customer inquiries in multiple languages, namely, **English**, **French**, and **Spanish**.

The **Multilingual Customer Support dataset** required to train the chatbot was created by using **MarianMT models –** ***opus-mt-en-fr & opus-mt-en-es*** and translating **Bitext Customer Support LLM Chatbot Training Dataset** containing English interactions, into French and Spanish. This dataset was then used to fine-tune the **mBART-large-50** model to respond to customer service queries in the aforementioned three languages. A web interface was built with **Gradio**, to allow interaction with the chatbot, which is fast and capable of handling customer queries and providing high-quality formatted responses in multiple languages within the same chat interaction.

This Project Report is approved for recommendation to the Graduate Committee.

Project Advisor:

____

Dr. Farnoush Banaei Kashani

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This section aims to introduce the project by providing the problem statement, objective(s), approach, & the organization of this report.

1.1. Problem Statement

Many companies in the United States and around the world, more specifically small to medium-sized businesses (SMBs) offer customer service to their customers primarily in the English language. These companies often implement chatbots into their websites or apps to save potential costs that they might incur from hiring customer service agents and representatives. The chatbots however, are usually designed and optimized to only operate in the English language, catering primarily to English speakers.

In first-world countries such as the United States and Canada, the languages that are most spoken are:

1) English
2) 2nd Most Spoken Language Country-wise
    a. Spanish with 42 million speakers (US)
    b. French with 8 million speakers (Canada)

Many of the companies in these countries, particularly SMBs tend to use English-only Chatbots. These companies are making the mistake or in some cases intentional decision of ignoring the communication and customer service needs of a large subset of their respective country's population i.e., 12% in the United States and 22% in Canada. This is a major problem for these people whose first language is not English.

This problem impacts many parties in these two countries in a variety of ways. The French and Spanish speakers may feel neglected or unheard with them often having trouble communicating their needs effectively in a language that isn't their first. This in a way forces them to learn a language that they otherwise would not have the need to learn. This in-turn leads to lower customer satisfaction which could ultimately lead to the customers switching to a company that does cater to their multilingual needs, thus leading to losses for the companies with English-only chatbots, which could be drastic, particularly for SMBs. These companies could also incur additional costs as they could find themselves in need of multilingual customer service agents when dealing with French and Spanish speakers, thus increasing their operating costs. When these problems are considered holistically, it's evident that they call for a solution that can benefit all parties involved.

1.2. Project Report Statement

The aim of this project is to develop an efficient & seamless Multilingual Single-turn Customer Service Chatbot capable of accurately responding to customer inquiries in multiple languages, namely, English, French, and Spanish.

1.3. Approach

The first task involved preprocessing the Bitext Sample Customer Support Training Dataset and removing inconsistencies and certain entities to prepare it for translation.

The processed dataset, particularly the instructions and responses were then translated into French and Spanish languages using MarianMT models - *opus-mt-en-fr* & *opus-mt-en-es* to create the Multilingual Customer Support Dataset.

The created Multilingual Customer Support Dataset was analyzed and interpreted comprehensively in a number of ways using a series of visualizations.

This dataset is used to fine-tune multilingual Bidirectional and Auto-Regressive Transformer Model (mBART-large-50), enabling it to generate responses to customer queries in English, French, & Spanish.

A user-friendly web interface is built using the Python library Gradio, allowing users to interact with the Multilingual Chatbot in their preferred language (en, fr, or es).

A comprehensive evaluation of the model's responses using conventional industry-relevant metrics such as BLEU, ROUGE, METEOR, TER, ChrF, COMET, & Latency Tests were carried out, both for the overall model responses and for the responses in each of the aforementioned languages.

1.4. Organization of this Project Report

Chapter 2 covers the Background of the project which contains Key Concepts that a reader must understand to understand this project well and Literature Review which includes research papers that are relevant to this project.

Chapter 3 covers the Architecture of the project which provides a High-Level Design of the project and goes into detail on the implementation steps for each of the project sections.

Chapter 4 covers the Methodology used to test the Project, its Results, and the Analysis of those results.

Chapter 5 concludes the Project Report by providing a Summary, Contributions or Potential Impact of the Project, and any Future Work that may be carried out.

2. BACKGROUND

2.1. Key Concepts

The reader must understand these areas of research - Multilingual Natural Language Processing (NLP), Transformer Models and mBART Architecture, and Evaluation Metrics for Language Generation to understand this Project Report.

2.1.1. Multilingual Natural Language Processing (NLP) Concepts

2.1.1.1. Multilingual NLP

Multilingual NLP is a branch of NLP which itself is a branch of Artificial Intelligence that involves the processing of raw text data into usable assets that can be employed in many use cases. Multilingual NLP in particular, would be the process of creating machine learning models that has the capability of understanding text in different languages and generating appropriate responses in those languages. This key concept plays a major role in this project as it's the main component behind enabling the chatbot to respond in multiple languages.

Literature Review:

Relatively recent research shows the capabilities of Multilingual NLP in terms of handling multiple languages effectively. A great proponent of this can be found in the paper by Liu et al. (2020) [1] which goes into detail on how transformer models like mBART can allow for quality multilingual text generation if utilizing cross-lingual pre-training strategies. This can allow for the model to make use of effective techniques like zero-shot learning where the model can predict a class of a new sample without being trained on that class and few-shot learning which allows the model to learn from a small subset of labelled samples.

Capabilities such as the ones above allow models like mBART to perform equally well across a plethora of languages. The paper from Conneau et al. (2020) [2] demonstrates this by means of a comprehensive survey in which the importance of cross-lingual transfer learning using shared representations is highlighted to show that great performance can be achieved even in low-resource languages when such multilingual pre-training strategies are employed.

2.1.1.2. Transfer Learning

Transfer Learning, in simple terms can be defined as the process of using a pre-trained model and fine-tuning it on a task-specific dataset to achieve the language task that is desired. This is an effective approach because it can allow for the model to make full use of all the original understanding it gained from being trained on a larger volume of data and use it to adapt to the current task and perform very well without having to employ significant training. If the model is trained on task-specific dataset right from the get-go, then significantly more data, time and resources would be needed to achieve the same NLP task.

Literature Review:

This ground-breaking idea was introduced in the paper [3] by Howard and Ruder (2018) in the form of Universal Language Model Fine-Tuning (ULMFiT), in which they elaborate that the process of pre-training a general-domain language model on large datasets and then fine-tuning it on a target-task significantly improves the model performance even when the task-specific data is limited. This paper led to transfer learning becoming a staple in the creation of NLP models.

More important information pertaining to this project can be found in paper [4] by Ruder (2019) where cross-lingual transfer learning was shown to benefit all languages equally even when some are them are low-resource languages i.e., sparse train data. This technique when applied to a model allows the model to share knowledge across languages by using a critical idea known as Multilingual Embeddings, which establishes a common understanding of the same data across different languages by representing similar words or phrases closer in the shared vector space. This in-turn optimizes the language-specific performance for all languages by reducing the potential data needed for training.

2.1.2. Transformer Models, MarianMT, and mBART Architecture

2.1.2.1. Transformer models

Deep Learning models are algorithms that are based off of Artificial Neural Networks that simulate the workings of an actual human brain. There are various types of Deep Learning models like CNNs, RNNs, GNNs and Transformer models are one of them. These models however are able to offer a unique advantage over other types because their working differs from traditional neural networks like Recurrent Neural Networks (RNN) due to the presence of a unique mechanism known as an attention mechanism. RNNs are only designed to process text in a sequential manner but transformer models are able to use attention mechanisms to capture relationships between all words simultaneously. This has the ability to allow for something known as parallelized training, therefore processing large amounts of data at once, and ultimately resulting in better speeds and performance.

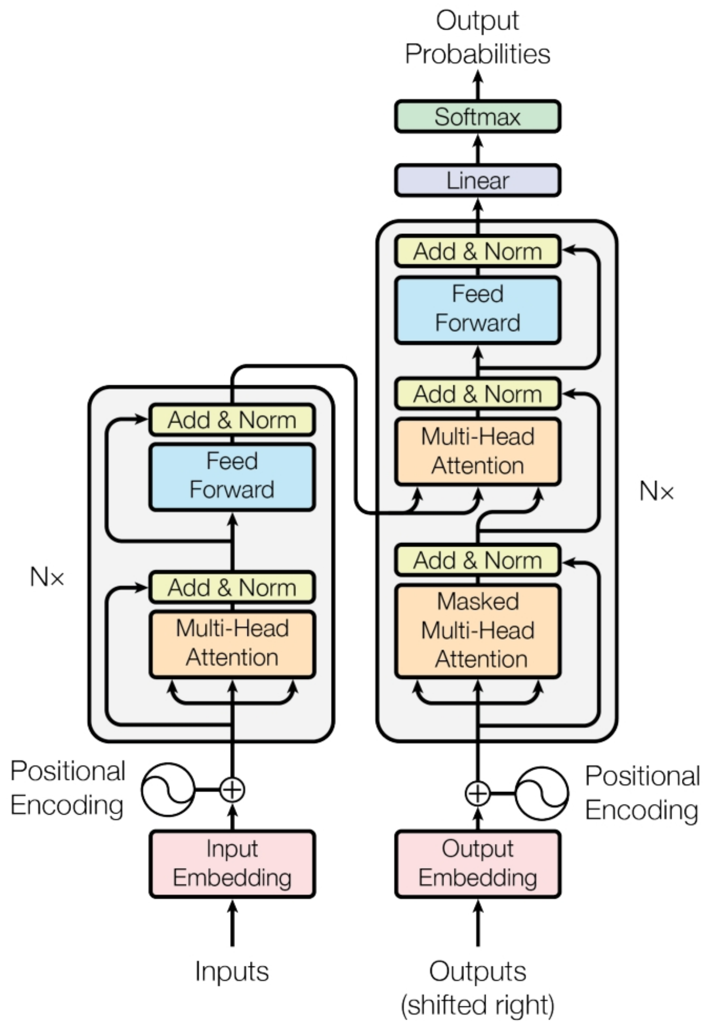

Figure 1. Transformer Model Architecture. Reprinted from [5]

The architecture of a transformer model consists of two blocks known as encoder and decoder, shown in the left and right halves of Figure.1 respectively. Each of the blocks contains a number of layers which themselves contain multi-head self-attention networks and feed-forward neural networks. These networks allow the model to capture the contextual meanings of the words in a long sequence accurately by weighing the importance of the words dynamically. So, this powerful mechanism can allow for the model to focus on multiple areas of the sentence at the same time, which enables it to understand complex relationships between the words.

Literature Review:

Transformer models were first introduced in the paper by Vaswani et al. (2017) [5] which showed that, thanks to their ability to capture complex dependencies, they do not incur the vanishing gradient problem, which is a problem that occurs when the gradients used to update the network become very small as they are backpropagated through the network, causing a network to learn extremely slowly or not learn at all. Thus, the transformer models can able to outperform conventional RNN-based architectures on various NLP tasks.

2.1.2.3. MarianMT Architecture

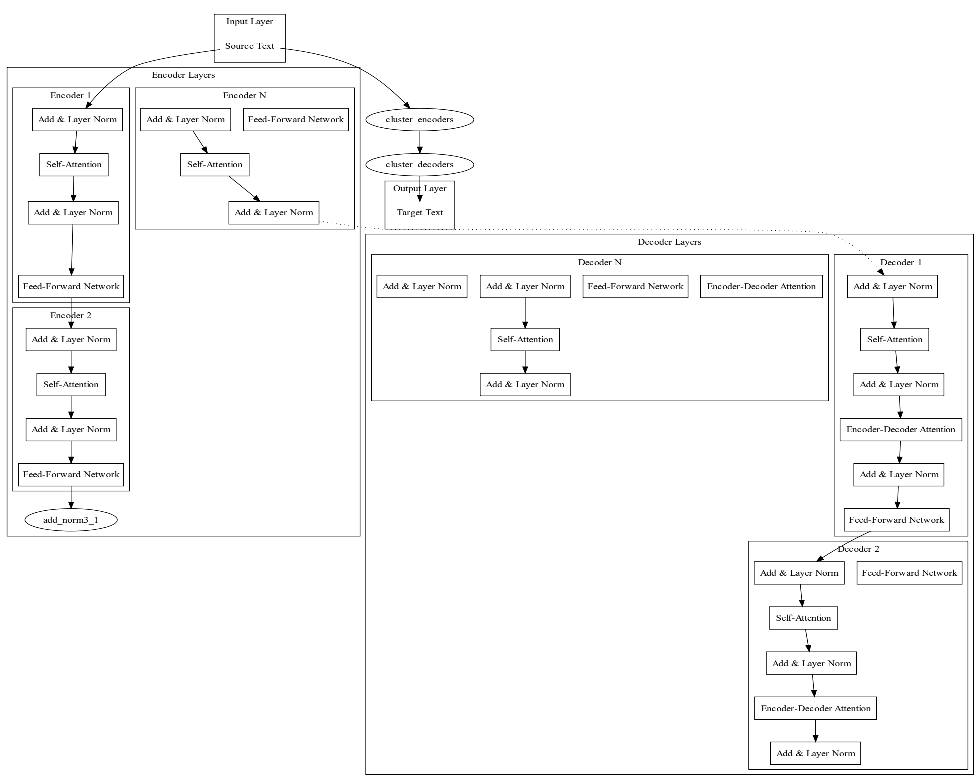

Figure 2. MarianMT Architecture. Reprinted from [6]

MarianMT model is a machine translation model that uses the transformer architecture. The input is processed in the encoder, where the contextual data of the text is interpreted in the input's language. The decoder is responsible for generating the same text data in the target language and it uses the same Self-Attention mechanism and feed-forward-networks as in the encoder but also makes use of an additional mechanism known as Encoder-Decoder-Attention. These mechanisms work in unison to create what's known as a Cross-Attention mechanism. The Encoder and Decoder blocks also capture the positions of words in sentences by using something called Positional Encoding. The concurrent employment of all these techniques enable the model to generate translations by considering entire sentences holistically, resulting in translated output sentences that capture the semantic meaning of the input sentences accurately.

Literature Review:

The Marian Toolkit was introduced in paper by Marcin et al. (2018) [7] as an efficient toolkit for neural machine translation, written purely in C++. C++ was chosen due to its minimal dependencies and maximal optimization capabilities. At the time, Marian was able to outperform the state-of-the-art due to its integrated automatic differentiation engine that produces dynamic computation graphs, allowing it to be trained and used for translation tasks effectively. Marian was also touted for its versatility in that it can be co-adopted with plenty of other neural network models like RNN or Transformers. Another unique feature of Marian is its capability to support multi-device training, scoring, and beam search, providing greater odds for scalability and flexibility for users. These features when considered holistically, make Marian a well-rounded toolkit for Neural Machine Translation and so it was chosen to be used for this project.

2.1.2.3. mBART Architecture

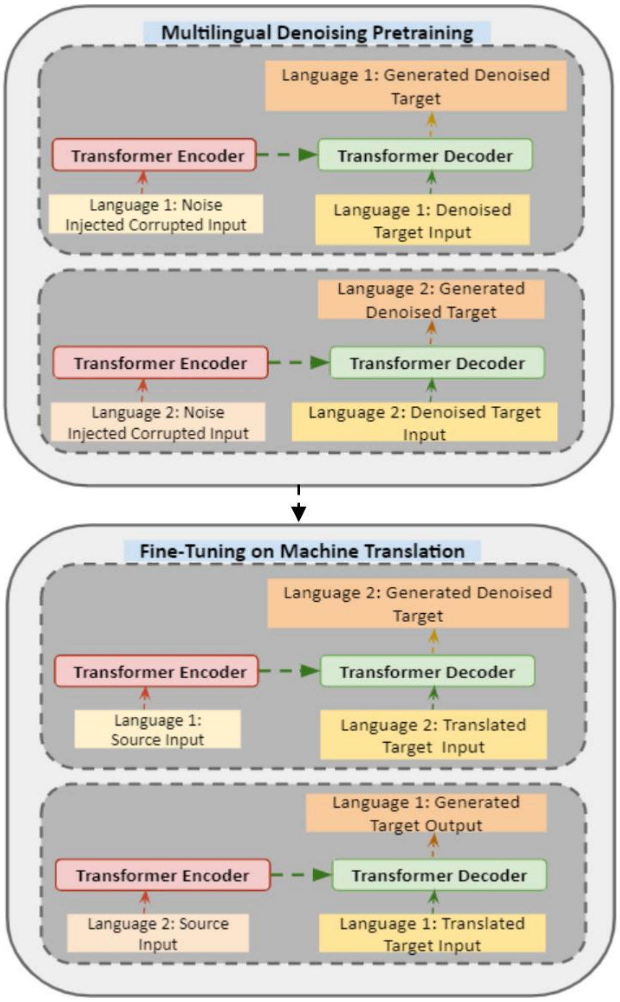

Figure 3. mBART Architecture. Reprinted from [8]

mBART is a transformer model that stands for Multilingual Bidirectional Auto-Regressive Transformers. This model is used for machine translation and it can be considered a child of the BART model. The model is unique because a technique known as denoised training was applied to it during its pre-training process. In this process, the model learns from noisy data to retrieve the correct form of the same data. Noisy data for the training process can be created by introducing some minor changes to the original data, for example, changing some letters, adding spaces, and so on. This unique way of training allows the model to learn not just fixed translations of a sentence but also the semantic meaning, context, and relationships between parts of a sentence irrespective of the language leading to great performance in NLP tasks. It should be noted that this type of training is vastly different from traditional translation models that are usually designed to produce direct translations from a source language to a target language.

Literature Review:

mBART was introduced in paper [9] by Liu et al. (2020), in which they demonstrated the capabilities of mBART in detail by way of a comprehensive comparison with state-of-the-art models, deeming its performance to be superior or on-par. Their research concludes that both supervised and unsupervised machine translation can be improved by mBART at both the sentence and document level due to its unique training method, i.e., multilingual denoising. It is to be noted that mBART's transfer learning ability when dealing with multiple languages was also further emphasized, which is a crucial component for this project as this particular capability is what facilitates the creation of a multilingual chatbot.

2.1.3. mBART Training and Response Generation Arguments

2.1.3.1. mBART Training Arguments

When it comes to Transformer models, its final performance in terms of speed, efficiency, accuracy and translation quality depends on the training arguments chosen for the fine-tuning task by a very large margin. The training arguments that were finalized for this project were discovered to be the optimal one after rigorous research and many iterations of model training and failed attempts in producing a good one. Thus, it is important to understand the logic behind each of the training arguments to better understand the reason behind their choosing. This section goes over what the arguments do and Section 3.4 will cover the specific argument values chosen in detail.

Find explanations for the training arguments below:

i. `evaluation_strategy` – Specifies at what times during the training should the model be evaluated.

ii. `save_strategy` - Specifies at what times during the training should the model be saved.

iii. `learning_rate` - Specifies how well or fast the model should learn from the data initially. This affects the Adam Optimizer which is the conventional algorithm used for training transformers due to its dynamic learning rate, effective convergence capabilities, and low memory usage.

iv. `per_device_train_batch_size` - Specifies how much of the train data should be used for training the model in each iteration. The model weights will be updated after each batch i.e., the no. of samples provided in the batch size.

v. `per_device_eval_batch_size` – Specifies how much of the test data should be used for evaluating the model in each iteration.

vi. `num_train_epochs` – Specifies the number of epochs the model should be trained for. An epoch can be defined as the model making one complete pass through the entire training dataset. Higher number of epochs could result in good performance but the risks for overfitting are higher as well. Lower epochs may lead to underfitting.

vii. `save_total_limit` – Specifies the number of times the model weights should be saved during training. This allows to resume the training in case of any technical issues that causes the training to halt.

viii. `fp16` – Specifies if mixed-precision training should be used. This strategy uses 16-bit floating-point numbers for most computations and 32-bit floating-point numbers where appropriate. This can lead to faster training process and lower memory usage while also retaining the same performance from using 32-bit floating-point numbers.

ix. `weight_decay` – Specifies the decay value for the model weights in each epoch. This value will be used to regularize the total loss in each epoch which helps avoid overfitting.

x. `logging_steps` – Specifies how often training stats should be outputted to the console. This helps to keep track of the training process and ensure the model is coming along as expected.

xi. `load_best_model_at_end` – Specifies whether the best performing version of the model should be loaded at the end. This uses the validation loss of each epoch and chooses the model that has the lowest loss.

xii. `gradient_accumulation_steps` – Specifies how many batches or gradients after which the model should update its weights. Larger batches lead to better performance and convergence as indicated in this study by Popel et. al. (2018) [10]. This argument allows to simulate a larger batch size without directly using one. This allows to save memory and get the same benefits of a large batch size.

xiii. `warmup_steps` – Specifies how many steps the model should go through before reaching the initial value set in the `learning_rate` argument. Throughout this range, the learning rate increases linearly, eventually reaching the argument value. This helps the model generalize better to the train data without the exploding gradient problem, avoiding erratic updates to the weights at the start of the training.

xiv. `report_to` – Specifies whether training stats should be reported to external tools or services.

xv. `gradient_checkpointing` – Specifies whether every intermediate layer's activation during forward pass should be saved in memory or only certain checkpoints should be saved. This can help to save memory if set to true but at the cost of more computation time as the same activations would have to be recomputed during the backward pass.

xvi. `lr_scheduler_type` – Specifies how the learning rate should evolve over the course of model training. This can control how fast/slow/well the model learns from the data.

xvii. `eval_accumulation_steps` – Specifies how many batches of test data to accumulate before calculating the loss metrics. This helps to save memory and ensure stable loss values.

xviii. `data_collator` – It is a function that specifies how the data is prepared before its fed to the model. It deals with adding padding and special tokens to the data while ensuring the inputs are correctly structured.

xix. `callbacks` – Specifies if any additional behavior or actions should be added during model training. This can range from logging training stats, computing metrics to any custom functions and the like.

2.1.3.2. mBART Response Generation Arguments

The same process of trail and error was carried out to finalize the model response generation parameters. These parameters can directly influence many aspects of the model's responses such as length, quality, and so on. As before, it is crucial to understand these parameters and what they do in order to get a full picture of the inner-workings of the model when generating responses to instructions. This section goes over what the parameters do and Section 3.5 will cover the specific values chosen in detail.

Find explanations for the training arguments below:

i. `input_ids` – This is an input parameter which contains the numerical values of the input text obtained using the model's tokenizer. Transformer models store their own vocabulary of words and associated numbers which is what they use to do their tasks. Thus, the input text has to be tokenized such that its words match the correct numbers.

ii. `attention_mask` – This is another input parameter which contains an list of binary values 0 and 1. The value 1 signifies an actual input token and the value 0 signifies a padding token. This helps the model focus on using just the input data and avoid padding that was simply added for making the input sequences uniform.

iii. `forced_bos_token_id` – Ensures the model uses the Beginning of Sequence id for the generated responses. Including this id ensures the response is in the same language as the input text.

iv. `eos_token_id` – Ensures the model use the End of Sequence id for the generated responses. Including this id ensures the model stops generating the response after reaching the eos token id, preventing unnecessarily long responses.

v. `min_length` – Specifies the minimum length the model's response should be.

vi. `max_length` – Specifies the maximum length the model's response should be.

vii. `num_beams` – Specifies the value for the Beam Search algorithm. This algorithm explores multiple paths for the input instruction, chooses the response text bits with the highest cumulative likelihood score and returns the accumulated response.

viii. `no_repeat_ngram_size` – Specifies how repetitive the words in the generated response should be. A n-gram is a contiguous sequence of n words in the response. The number provided will ensure to avoid repetition of exactly that category of word-number sequences (eg: bi-grams or 2-word sequences) in the generated response.

ix. `early_stopping` – Specifies whether the model should stop Beam Search if a response with a high likelihood score is found, avoiding further generation of tokens and saving time.

2.2. Related Work

Multilingual NLP Systems is a field of Artificial Intelligence that has had significant breakthroughs in the past few years. The aim of this project is to use information from the studies, identify areas of improvements, and provide successful implementations that build on these studies. Thus, for this project three such papers with their own primary areas of related work were used as focus points for this project to expand upon. They are:

i. Multilingual Chatbot Systems
ii. Translation Accuracy and Consistency
iii. Cross-Cultural Natural Language Understanding

2.2.1. Multilingual Chatbot Systems

One of the major aims of this project is to use Transformer models such as MarianMT and mBART to enhance the performance of multilingual chatbot systems. The study [11] published by Aggarwal et al. (2023) goes into detail on multilingual NLP models, showing conclusively that they can handle multiple languages effectively when the conversation is a single-turn one. One major gap that was identified in the paper is that these models are performing well in some languages but not at the same level in other languages, and therefore not robust enough to handle complex customer service queries in multiple languages.

This project aims to build on their work by creating a high-performant multilingual customer service chatbot by means of a mBART model finetuned specifically for providing customer service. This largely enhances its capabilities when it comes to delivering contextually appropriate responses that are robust and semantically accurate in multiple languages. Aggarwal et al. 's paper examined a specific technique known as Cross-lingual training where data from one language is used to improve performance of model in another language. This is important to this project as it can allow for the mBART model to adapt to customer instructions data in multiple languages like English, French and Spanish. Thus, allowing the model responses to be culturally sensitive and appropriate in all the languages enhancing the customer service provided by the chatbot.

2.2.2. Translation Accuracy and Consistency

One more relevant area of related work is maintaining fluency and meaning across various languages and this is what the study [12] by Kumar et al. (2022) is focused on. It provides the current challenges that are faced by NLP systems related to this area and they also further talk about the difficulties of maintaining semantics between languages while also ensuring the quality of the responses with regards to multilingual models' responses. One crucial limitation that is pointed out in their study is lack of most translation models' capability to generate responses consistently, in domains such as customer service that involve domain-specific jargon and dynamic terms.

The aim of this project is to directly address this limitation by crafting a chatbot that performs very well with regards to its response quality in customer service scenarios. This can be achieved by providing instructions and responses in multiple languages that are consistent and semantically accurate, ensuring the model learns to generate responses that maintain them. These instructions and responses can be obtained by using MarianMT models to generate translations of the English instructions and responses into French and Spanish. Marian neural machine translation toolkit is highly efficient in translating domain-specific jargon while also maintaining semantic accuracy. Thus, all of these components of the project when combined overcomes the difficulties pointed out by Kumar et al.

2.2.3. Cross-Cultural Natural Language Understanding

Another key area of related work is the paper [13] from Hershcovich et al. (2022). It goes into detail on how building NLP systems that can respect cultural linguistics and respond in a way that is also contextually appropriate in multiple languages is a hard problem, highlighting the challenges of cross-cultural NLP. They show that while multilingual NLP models respond appropriately in one language, they fail to do so in other languages, often being culturally insensitive. They argue that this lack of cultural coherence in multilingual NLP models is an important issue that could alienate people who speak different languages. This lack of tact in providing culturally and contextually appropriate responses by these NLP models affects not only low-resource languages but also high-resource languages. This issue especially plagues the customer service domain, where providing a good service is of great importance and so understanding cultural nuances and taking them into account for responses is crucial.

This project aims to build upon their work by building a chatbot that is not only linguistically accurate but also culturally sensitive in responding in English, Spanish and French languages. By fine-tuning the mBART model on data that is culturally appropriate with precise language-specific grammar and taking advantage of key concepts such as transfer learning, cross-lingual embeddings, and transformer-based architectures, this chatbot can be achieved.

3. ARCHITECTURE

3.1. High Level Design

The first task involved preprocessing the Bitext Sample Customer Support Training Dataset and removing inconsistencies and certain entities to prepare it for translation.

The processed dataset, particularly the instructions and responses were then translated into French and Spanish languages using MarianMT models - *opus-mt-en-fr & opus-mt-en-es* to create the Multilingual Customer Support Dataset.

The created Multilingual Customer Support Dataset was analyzed and interpreted comprehensively in a number of ways using a series of visualizations.

This dataset is used to fine-tune multilingual Bidirectional and Auto-Regressive Transformer Model (mBART-large-50), enabling it to generate responses to customer queries in English, French, & Spanish.

A user-friendly web interface is built using the Python library Gradio, allowing users to interact with the Multilingual Chatbot in their preferred language (en, fr, or es).

Find the flow-chart below that visualizes the architecture of this project:

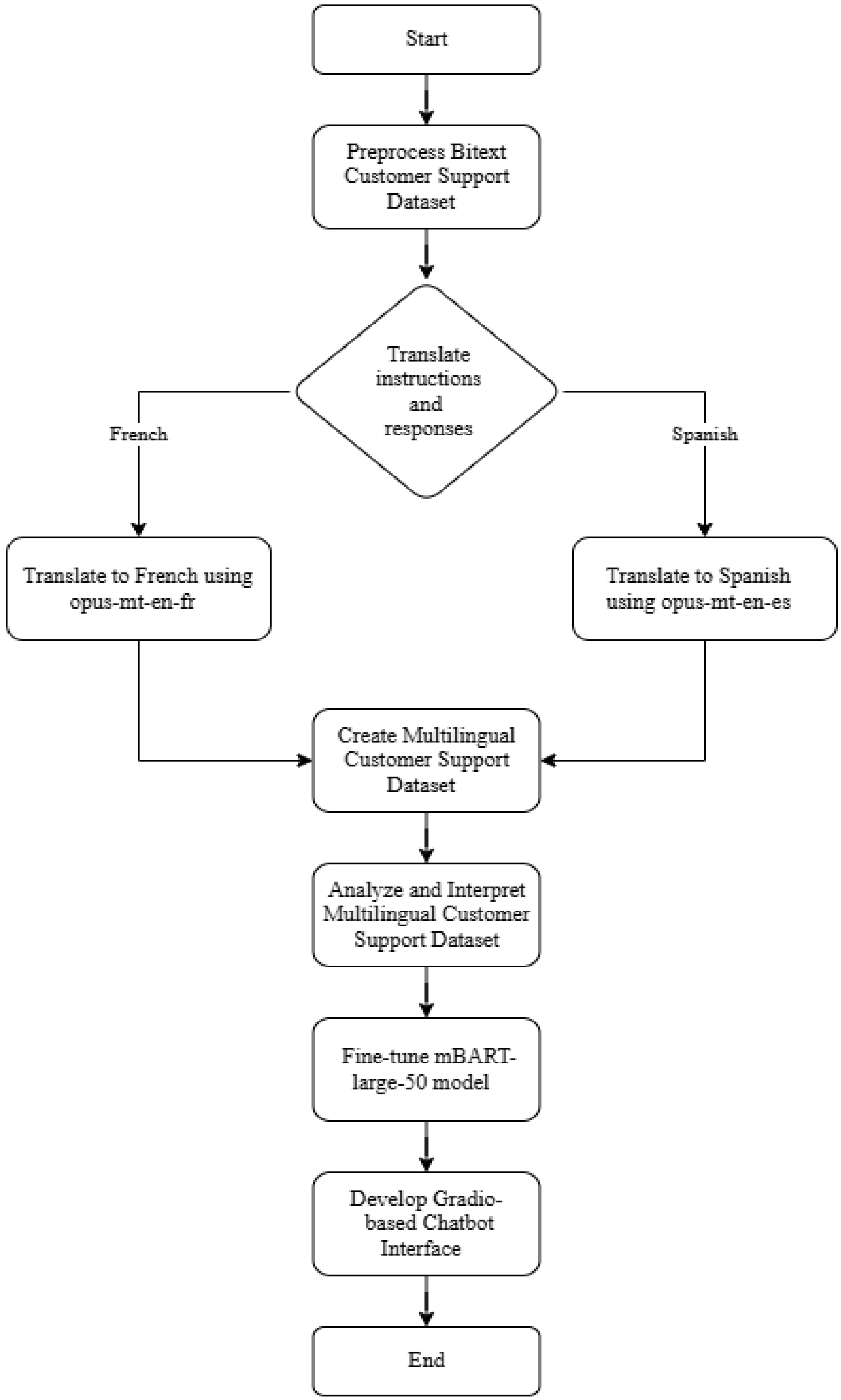

Figure 4. Flow-Chart of Project Architecture. Created using [13]

3.2. Multilingual Dataset Creation

This section is the starting point for the project. It involves preprocessing the Bitext Sample Customer Support Training Dataset and removing unnecessary entities, translating it to French and Spanish, and encoding it with special tokens to create the multilingual dataset needed to train the mBART model. Find the flow-chart that visualizes this section below:

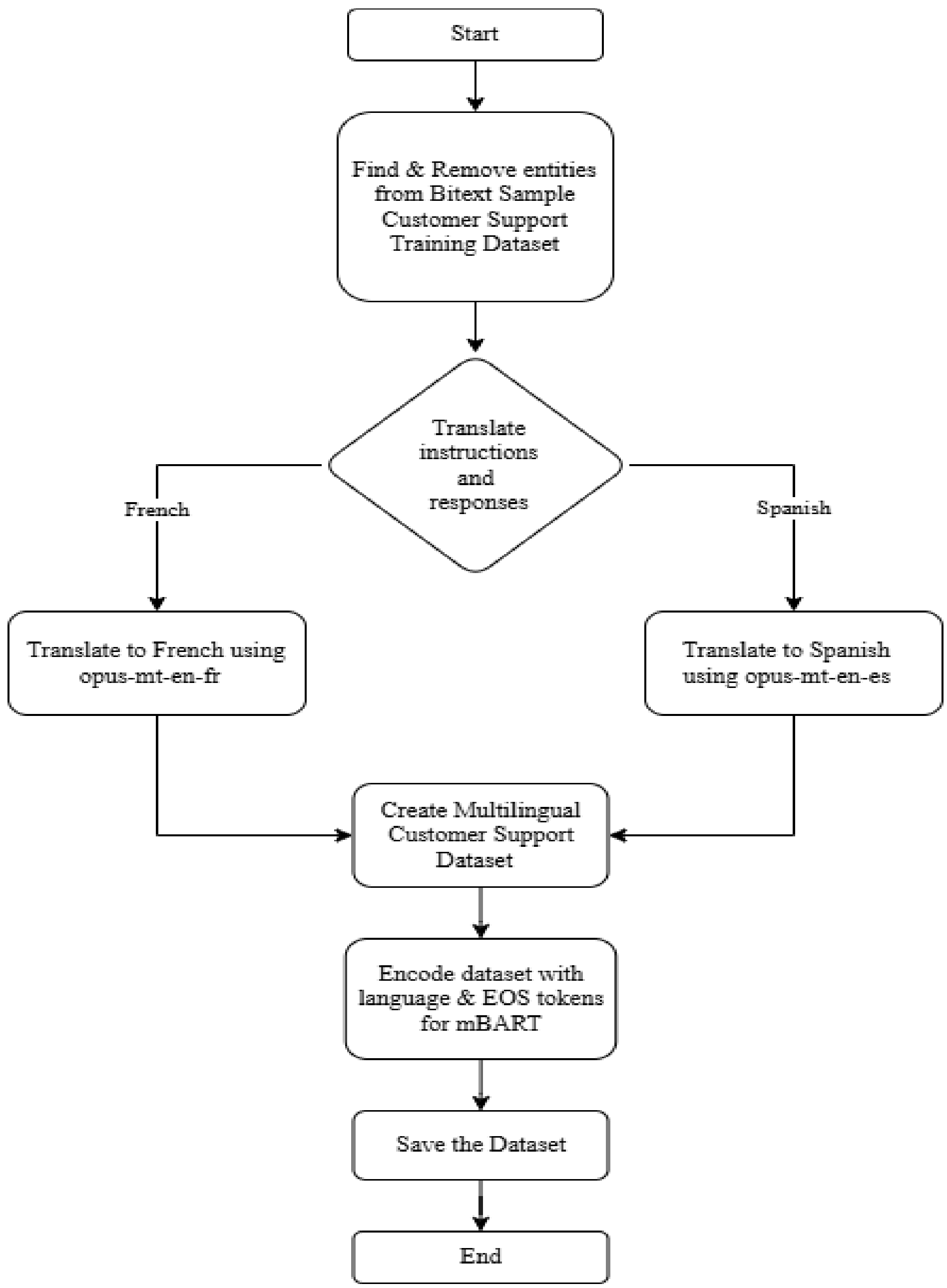

Figure 5. Flow-Chart of Multilingual Dataset Creation. Created using [13]

3.2.1. Implementation

Implementation of this section can be divided into two parts with their own sub-parts:

1. Dataset Preprocessing
   i. Interpret basic details of the Bitext Sample Customer Support Training Dataset
   ii. Detect unnecessary Entities in the Dataset
   iii. Remove those Entities from the Dataset
2. Multilingual Dataset Creation
   i. Create split_text_with_formatting Function to Split Data into Chunks for Translation
   ii. Create Translate function to translate English data to French and Spanish
   iii. Use MarianMT models opus-mt-es-fr and opus-mt-en-es to Create the translated data
   iv. Combine the translated data into one Multilingual Dataframe
   v. Format the data using lang_codes and special-tokens for M-Bart Training

3.2.1.1. Dataset Preprocessing

i. Basic details of the dataset are interpreted using `print(df.info())` which gives us the stats such as length, size, type, etc. of the data and `print(df.head())` which gives us an idea of how the data in each of the columns look. This will help to give an idea on how to prepare the dataset for translation with MarianMT models.

ii. The dataset contains large numbers of entities and nested entities which need to be replaced with dummy values so that they can be translated by the MarianMT models. The entities as they are do not contribute to the linguistic meaning of the sentences they're in so we do this to ensure our dataset contains meaningful sentences that the MarianMT models can translate into coherent sentences. The first step is to find all the entities in the dataset programmatically:

```
nested_entities_replacements = {
     r"{{Account Type}}": "Premium",
     r"{{Account Category}}": "Gold",
}
entity_replacements = {
     r"{{Order Number}}": "ORD12345",
     r"{{Invoice Number}}": "INV56789",
     r"{{Online Order Interaction}}": "order history",
     .
     .
     .
}
```

Two dictionaries to hold the entities and nested entities are initialized

```
entity_pattern = re.compile(r"\{\{.*?\}\}")
```

Regular expression to find the entities

```
def find_entities(text):
    return entity_pattern.findall(text)
```

Function that applies the entity pattern to text from a cell in the dataset

```
instructions_entities =
df['instruction'].apply(find_entities)
responses_entities = df['response'].apply(find_entities)
unique_entities =
set([entity.strip() for sublist in instructions_entities
for entity in sublist] + [entity.strip() for sublist in
responses_entities for entity in sublist])
```

All the entities from the instruction and response columns are found and only the unique ones are preserved using a set.

```
dictionary_keys = set([key.strip('r"') for key in
entity_replacements.keys()])
missing_in_dict = unique_entities - dictionary_keys
extra_in_dict = dictionary_keys - unique_entities
```

Remove regular expression formatting from the entities dictionary and compare with entities obtained from the dataset. Group them into entities that are missing and extra in the dictionary.

```
print(f"Entities missing in the dictionary:
{missing_in_dict}")
```

```python
print(f"Extra entities in the dictionary: {extra_in_dict}")
# Checking if both sets match
if not missing_in_dict and not extra_in_dict:
    print("All entities match!")
else:
    print("There are unmatched entities. Please review the
    results above.")
```

Print the missing and extra entities for comparison. Add missing entities to the dictionary and remove extra entities from the dictionary. Repeat the process until all entities are added to the dictionary. This allows to add dummy values for all the entities in the dataset.

iii. Now that the entities have been found and dummy values were added as values to their keys, we can replace the entities with dummy values in the dataset using a two-pass approach:

```python
def first_pass_replace(text):
    for entity, replacement in
    nested_entities_replacements.items():
        text = re.sub(entity, replacement, text)
    return text
```

Function that replaces all nested entities with dummy values in text from a cell

```python
def second_pass_replace(text):
    for entity, replacement in entity_replacements.items():
        text = re.sub(entity, replacement, text)
```

```
    return text
```

Function that replaces all entities with dummy values in text from a cell

```
df['instruction'] = df['instruction'].apply(lambda x:
second_pass_replace(first_pass_replace(x)))
df['response'] = df['response'].apply(lambda x:
second_pass_replace(first_pass_replace(x)))
```

Applying both functions to the instruction and response columns using nested calls in an anonymous function. This iterates through all cells in the instruction and response columns and replaces any encountered entities with their respective dummy values from the dictionary.

3.2.1.2. Multilingual Dataset Creation

i. The length of the responses from the dataset most of the time exceed 500 and most transformer models have a limit of 500 tokens that they can work with. This is especially true for MarianMT models which can only translate up to 500 tokens at once. Thus, the responses needed to be split into pieces, fed into the model for translation, and reattached after they were translated. The following programmatic approach was employed to achieve this task:

```
def split_text_with_formatting(text, max_chunk_size=500):
```

Function that splits text for the MarianMT model while also preserving the formatting

```
    paragraphs = text.split('\n')
    output_chunks = []
```

```
    temp_chunk = ""
```

All the text from a response cell is split by line-breaks and some variables to store text chunks are initialized.

```
    for paragraph in paragraphs:
        sentences = re.split(r'(?<=[.!?])\s+', paragraph)
        for stc in sentences:
            if len(temp_chunk) + len(stc) <= max_length:
                temp_chunk += stc + " "
            else:
                output_chunks.append(temp_chunk.strip())
                temp_chunk = stc + " "

        if temp_chunk.strip():
            output_chunks.append(temp_chunk.strip())
            temp_chunk = ""
```

Here each of the paragraphs are further divided into sentences by using a regular expression that checks for common sentence terminators. Then, each sentence is added to `temp_chunk` if adding the sentence to it doesn't exceed the max length set. If adding a sentence to `temp_chunk` exceeds the max length, the `temp_chunk` gets added to the `output_chunks` list and gets re-initialized with that sentence. This happens iteratively until all the paragraphs are split into chunks and added to the `output_chunks` list.

```
    if temp_chunk.strip():
        output_chunks.append(temp_chunk.strip())
```

```
        return output_chunks
```

An additional check is done for any sentences remaining in `temp_chunk` and they get added to `output_chunks` as well. This function refactors the input data into a format that MarianMT models can translate without any issues.

ii. The next step is to define the translation function and carry out the translation using MarianMT models opus-mt-en-fr & opus-mt-en-es and their respective tokenizers:

```
models = {
        'en-fr': MarianMTModel.from_pretrained("Helsinki-
        NLP/opus-mt-en-fr").to('cuda'),
        'en-es': MarianMTModel.from_pretrained("Helsinki-
        NLP/opus-mt-en-es").to('cuda'),
    }
tokenizers = {
        'en-fr': MarianTokenizer.from_pretrained("Helsinki-
        NLP/opus-mt-en-fr"),
        'en-es': MarianTokenizer.from_pretrained("Helsinki-
        NLP/opus-mt-en-es"),
    }
```

Loading the models on the GPU for faster processing speed and loading the models' respective tokenizers.

```
def translate(text, src_lang, tgt_lang, max_length=512):
```

Function that translates an input text from a cell from source language to target language.

```
model = models[f'{src_lang}-{tgt_lang}']
tokenizer = tokenizers[f'{src_lang}-{tgt_lang}']
```

Appropriate model and tokenizer are loaded using `src_lang` and `tgt_lang` params.

```
chunks = split_text_with_formatting(text)
  translated_chunks = []
```

chunks to be translated are obtained using `split_text_with_formatting` function on the input text.

```
for chunk in chunks:
  inputs = tokenizer(chunk, return_tensors="pt",
  padding=True, truncation=True,
  max_length=max_length).to('cuda')
  translated = model.generate(inputs['input_ids'],
  attention_mask=inputs['attention_mask'],
  max_length=max_length)
  translated_text = tokenizer.decode(translated[0],
  skip_special_tokens=True,
  clean_up_tokenization_spaces=True)
  translated_chunks.append(translated_text)
return '\n'.join(translated_chunks)
```

Here, each input chunk gets tokenized using the tokenizer, gets loaded on the GPU, translated by the model, decoded by the tokenizer and added to the `translated_chunks` list which is then returned as a single string sentence with line-breaks added between the chunks.

iii. Next, we use the MarianMT models to translate the instructions and responses.

```
tqdm.pandas(desc="Translating Instructions to French")
df['instruction_fr'] =
df['instruction'].progress_apply(lambda x: translate(x,
'en', 'fr'))
tqdm.pandas(desc="Translating Instructions to Spanish")
df['instruction_es'] =
df['instruction'].progress_apply(lambda x: translate(x,
'en', 'es'))
tqdm.pandas(desc="Translating Responses to French")
df['response_fr'] = df['response'].progress_apply(lambda x:
translate(x, 'en', 'fr'))
tqdm.pandas(desc="Translating Responses to Spanish")
df['response_es'] = df['response'].progress_apply(lambda x:
translate(x, 'en', 'es'))
```

TQDM is used for visualization of the translation progress and so it gets initialized with a description for each column that is being translated. The `translate` function is applied to the instruction and response columns which are translated into French and Spanish. The translated data is stored into new columns in the same dataframe with their respective names indicative of the language of the data.

iv. Once we have the translated data, it is reformatted into a dataset that contains the instructions and responses in all the languages within the same two columns and a new language column is added which contains language codes marking the language of each of the rows. These lang codes will help in creation of the dataset, mBART training and also during its evaluation. Programmatic implementation below:

```
multilingual_df = pd.DataFrame()
temp_df_en = df[['flags', 'category', 'intent',
'instruction', 'response']].copy()
temp_df_en['language'] = 'en'

temp_df_fr = df[['flags', 'category', 'intent']].copy()
temp_df_fr['instruction'] = df['instruction_fr']
temp_df_fr['response'] = df['response_fr']
temp_df_fr['language'] = 'fr'
temp_df_es = df[['flags', 'category', 'intent']].copy()
temp_df_es['instruction'] = df['instruction_es']
temp_df_es['response'] = df['response_es']
temp_df_es['language'] = 'es'
multilingual_df = pd.concat([temp_df_en, temp_df_fr,
temp_df_es], ignore_index=True)
```

A new pandas dataframe for the multilingual dataset is created. All the original English data get added to a temporary dataframe `temp_df_en` with a new column `language` containing "en" lang codes. Translated French data and other column data from the original dataset are copied to a temporary dataframe, including a new column `language` with "fr" lang codes. Same approach is carried out for translated Spanish data. All three temporary dataframes are concatenated into one dataframe under the same column names as the original dataset with a new `language` column containing each language's lang code.

```
lang_codes = {
    'en': 'en_XX',
    'fr': 'fr_XX',
    'es': 'es_XX'
}
multilingual_df['instruction'] =
multilingual_df.apply(lambda row:
f"{lang_codes[row['language']]} {row['instruction']} </s>",
axis=1)
multilingual_df['response'] = multilingual_df.apply(lambda
row: f"{lang_codes[row['language']]} {row['response']}
</s>", axis=1)
multilingual_df.to_csv(output_path, index=False)
```

v. The multilingual data in instruction and response columns are encoded for mBART training by adding appropriate language tokens (`en_XX, fr_XX, es_XX`) at the start of each data cell and End-of-Sequence token (`</s>`) at the end of each cell. This is done to allow mBART to identify the language of the input text during training. Finally, we save the final dataframe into a csv file and thus, we have a Multilingual Customer Support Dataset ready for our purposes.

3.3. Multilingual Dataset Analysis

This section involves analyzing and interpreting various features of the newly created Multilingual Customer Support Dataset such as the Distribution of different languages, length of instructions and responses across the languages, and so on. The dataset is also analyzed with respect to mBART using its tokenizer to tokenize the dataset, analyzing the tokenized data for insights. These interpretations will help in determining if the dataset is sufficient and bias-free for our finetuning task.

Find the flow-chart that visualizes this section below:

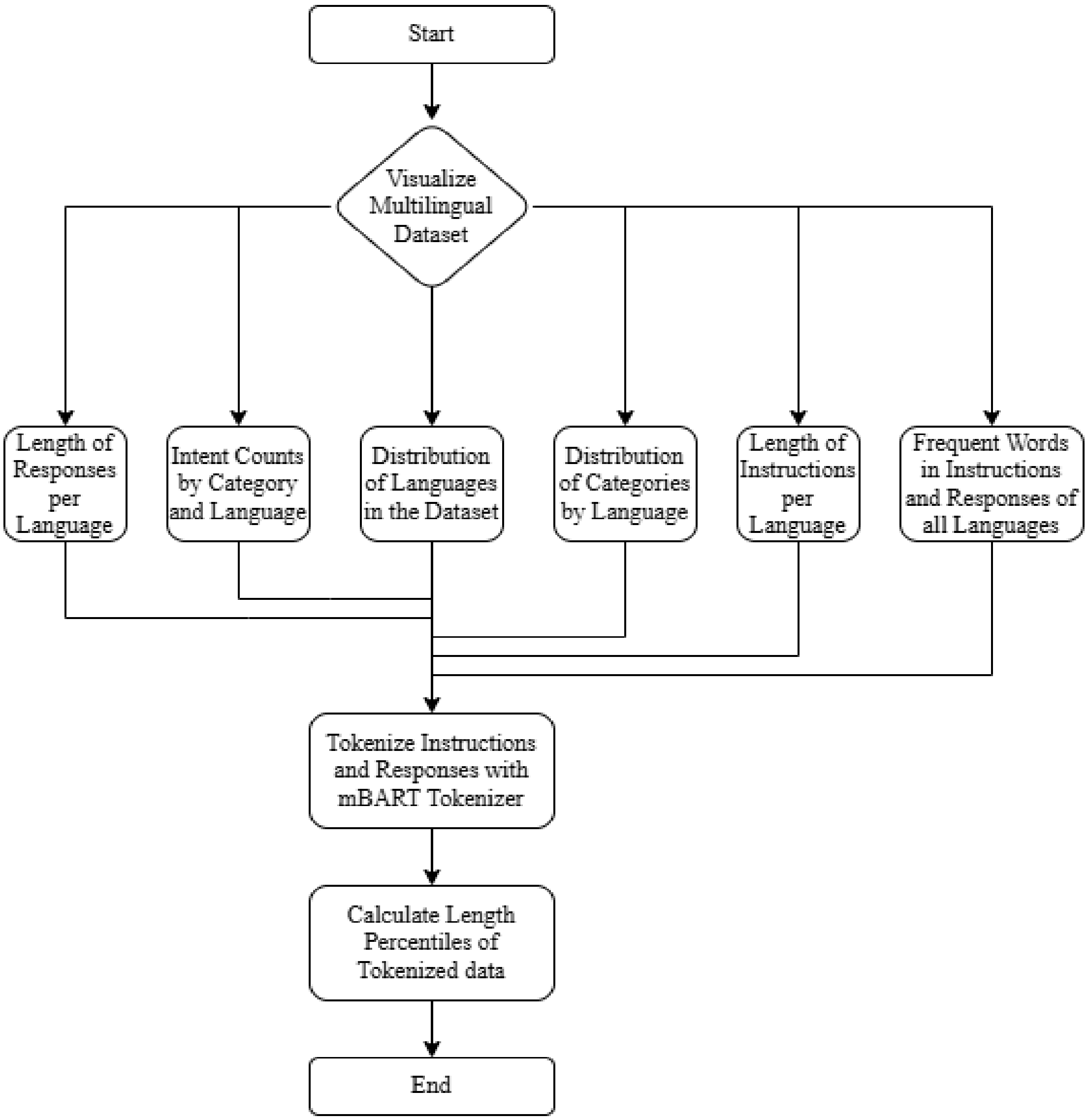

Figure 6. Flow-Chart of Multilingual Dataset Analysis. Created using [13]

3.3.1. Implementation

Implementation of this section can be divided into two parts with their own sub-parts:

1. Visualize Multilingual Dataset
   i. Visualize the Distribution of Languages in the Dataset
   ii. Visualize the Distribution of Categories by Language
   iii. Visualize the Intent Counts by Category and Language

iv. Visualize the Length of Instructions per Language
v. Visualize the Length of Responses per Language
vi. Visualize Frequent Words in Instructions and Responses of all Languages

2. Analyze Dataset with mBART
    i. Tokenize the Instructions and Responses from the Multilingual Dataset using mBART Tokenizer
    ii. Calculate 95-100th Length Percentiles for the Tokenized Instructions and Responses

3.3.1.1. Visualize Multilingual Dataset

i. The first visualization deals with analyzing the distribution of languages in the dataset. We must ensure all languages have an equal number of data samples so that the training of mBART is bias-free. Programmatic implementation:

```
plt.figure(figsize=(8, 5))
sns.countplot(data=multilingual_df, x='language',
palette='Set2')
plt.title('Distribution of Languages in the Dataset')
plt.xlabel('Language')
plt.ylabel('Number of Entries')
plt.show()
```

Matplotlib's pyplot is used to set up the canvas for the plot including its size, title and labels while Seaborn is used to generate a count plot for the language column from the dataset.

Plot:

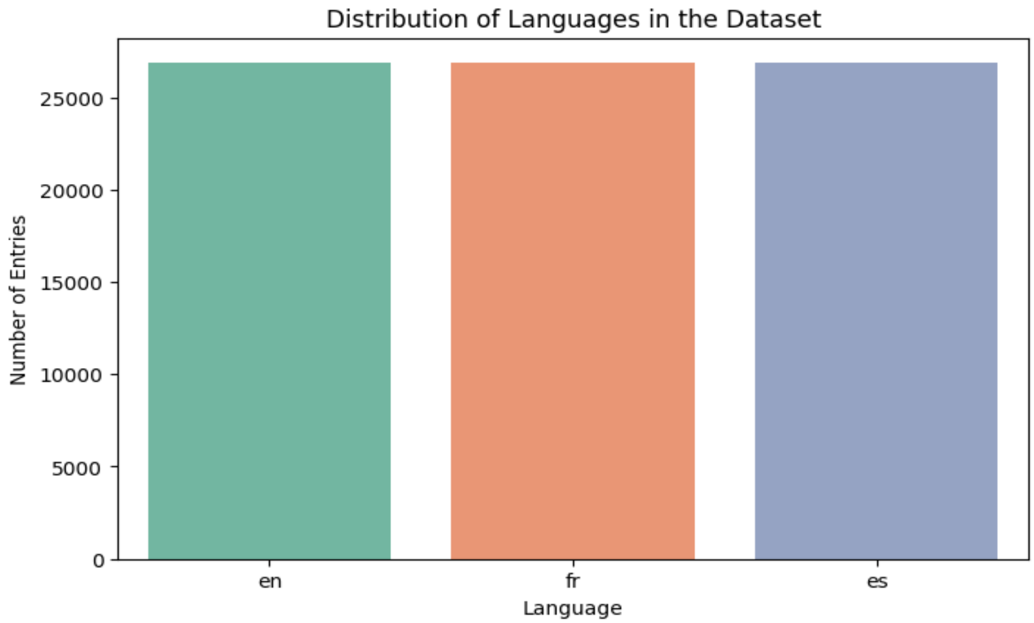

Figure 7. Distribution of Languages in the dataset. Created using Python Libraries

Interpretation: We can clearly see that the number of samples for each of the language's data is exactly the same, around 27000 data samples. This is not surprising as we used the MarianMT models to translate each and every single data cell from the original English dataset. Thus, we can rest assured that mBART will not be biased towards one language with regards to performance.

ii. The second visualization deals with the distribution of categories of the data samples across all the languages. Category column contains what category each instruction and response pair belong to, for example, Order, Shipping, Cancel, etc. It could be used to predict the trained model's performance for each category of responses based on the sample sizes available. Programmatic implementation:

```
plt.figure(figsize=(15, 8))
sns.countplot(data=multilingual_df, x='category',
hue='language', palette='Set1')
```

```
plt.title('Distribution of Categories by Language')
plt.xlabel('Category')
plt.ylabel('Number of Queries')
plt.xticks(rotation=45)
plt.legend(title='Language')
plt.show()
```

Matplotlib's pyplot is used to set up the canvas for the plot including its size, title, labels, legend, and x-axis label rotations while Seaborn is used to generate a count plot for the category column from the dataset with hue set to language which displays the categories distribution for all languages side-by-side allowing for a direct comparison.

Plot:

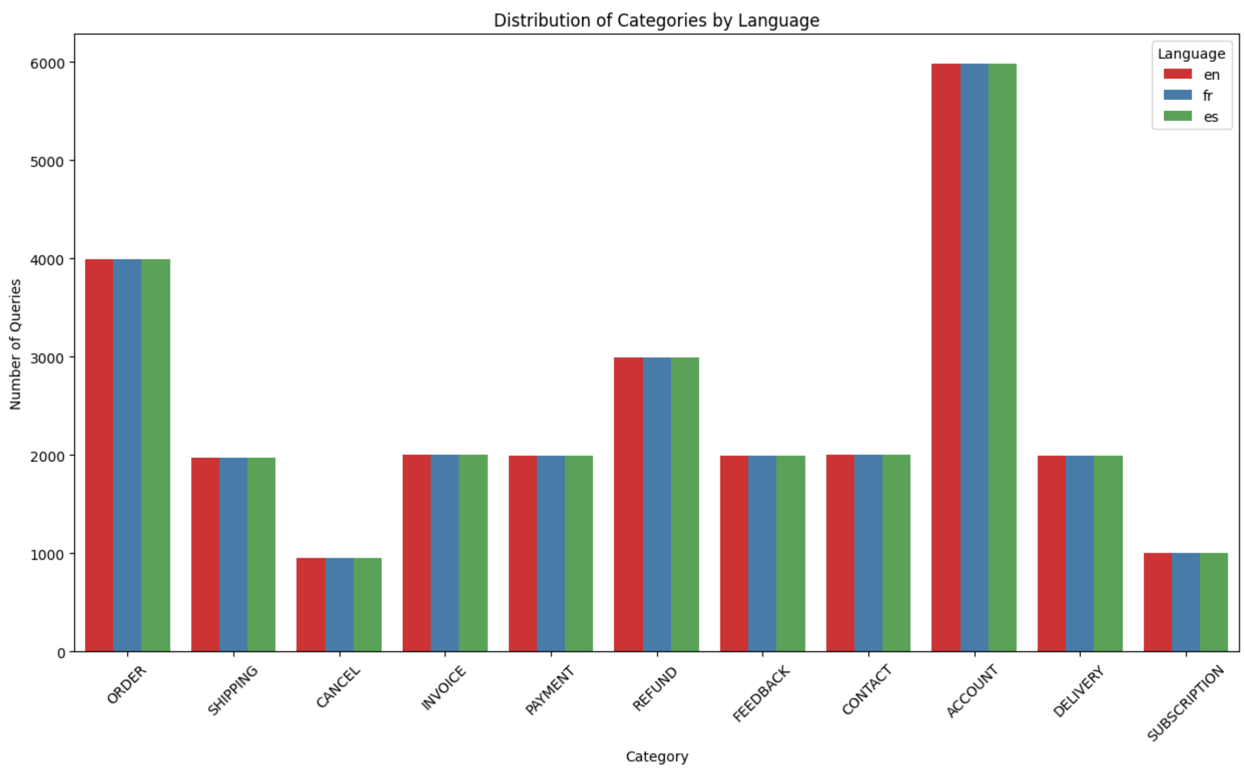

Figure 8. Distribution of Categories by Language. Created using Python Libraries.

Interpretation: We can get the general idea of how many categories of instructions and responses are present in the dataset and how many samples each category has available. We can see three categories Account, Order, and Refund that have a lot more samples compared to other categories, which could potentially lead to better response generation performance for those instructions due to availability of large sample size for training. We also see that each of the categories is equal in number across the languages and thus our dataset is bias-free across categories and languages.

iii. This visualization deals with the distribution of intent counts across categories and languages. Intent column contains what intent each instruction and response pair belong to, for example, cancel_order, delivery_period, complaint, etc. The visualization could be used to predict the trained model's performance for generated responses that belong to each of the intent types. Programmatic implementation:

```
intent_counts = multilingual_df.groupby(['category',
'language'])['intent'].value_counts().unstack(fill_value=0)
plt.figure(figsize=(20, 10))
sns.heatmap(intent_counts, annot=True, fmt='d',
cmap='YlGnBu', cbar_kws={'label': 'Number of Queries'})
plt.title('Heatmap of Intent Counts by Category and
Language')
plt.xlabel('Intent')
plt.ylabel('Category and Language')
plt.show()
```

The category and language columns from the multilingual dataframe are fetched and grouped together and occurrences of the intents are counted. These data are structured in a pivot-table format with intent-counts as the column and category-language pairs as the rows. As before, Matplotlib is used for setting up the canvas and other details of the plot. Seaborn's heatmap is used to visualize the intent counts across each of the category-language pairs.

Plot:

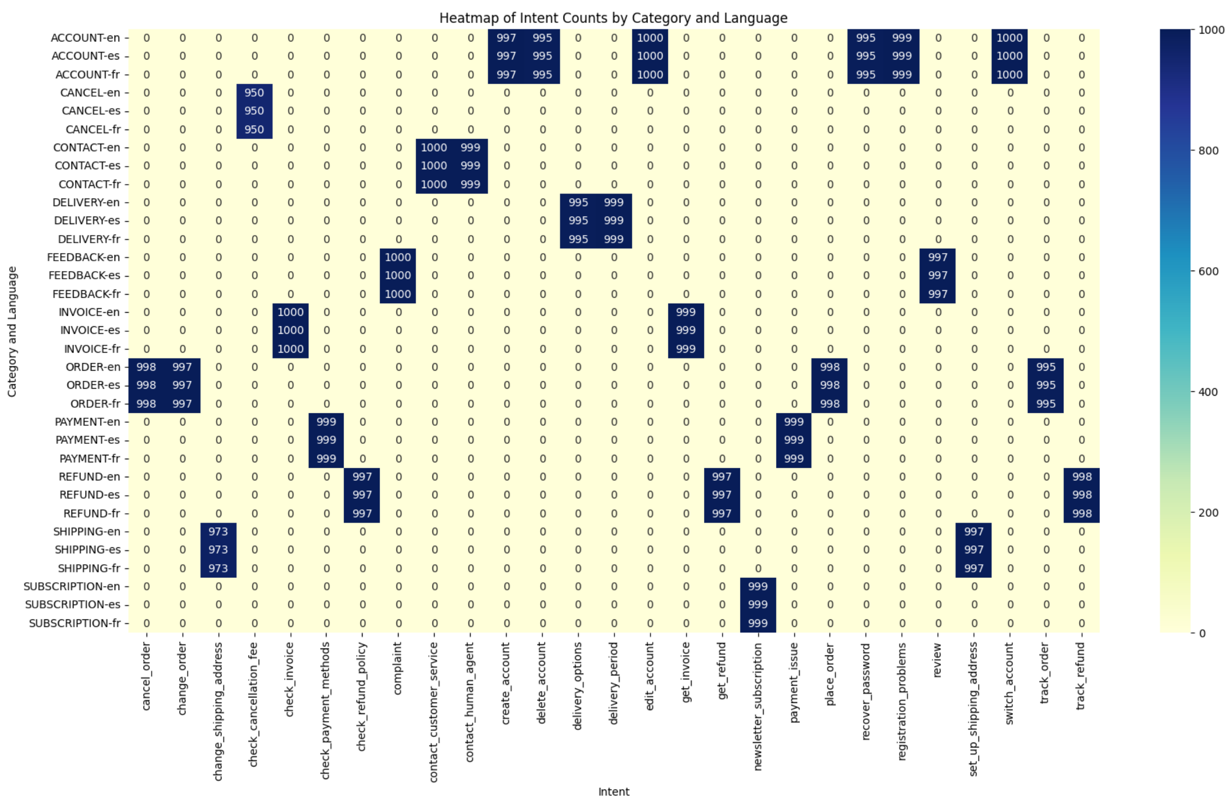

Figure 9. Intent Counts in the Dataset by Category and Language. Created using Python Libraries.

Interpretation: We can get a complete glance at the relationship between categories, languages and intents from this heatmap. Intents seem to be equally distributed across each of the languages meaning our dataset is not biased toward any particular set of intents. Another observation is that each intent only occurs alongside it's parent category, for example, cancel_order intents only occur alongside Order category, renew_subscription intents only occur alongside Subscription category, and so on. Thus, our instructions and responses can be confirmed to be consistently grouped with appropriate intents and categories.

iv. The following visualization deals with computing character-wise length percentiles of instructions across all languages. This could help us see if certain languages have longer instructions than others and decide the appropriate max_length for tokenizing the instructions during mBART fine-tuning. Programmatic Implementation:

```
multilingual_df['instruction_length'] =
multilingual_df['instruction'].apply(len)
plt.figure(figsize=(10, 6))
sns.boxplot(data=multilingual_df, x='language',
y='instruction_length', palette='coolwarm')
plt.title('Length of Instructions per Language')
plt.xlabel('Language')
plt.ylabel('Instruction Length (characters)')
plt.show()
```

The character-wise length of each instruction is calculated. Matplotlib is used for configuring the canvas properties and relevant details of the plot. A boxplot from Seaborn is used to visualize the length percentiles. The box in the plot visualizes the interquartile range of the length with the top edge corresponding to the 75th percentile, median corresponding to the 50th percentile and the bottom edge corresponding to the 25th percentile. The lines around the box itself are called whiskers and they mark the data with length less than 1.5 times the interquartile range. Any points outside the whisker are outliers with lengths greater than 1.5 times the interquartile range.

Plot:

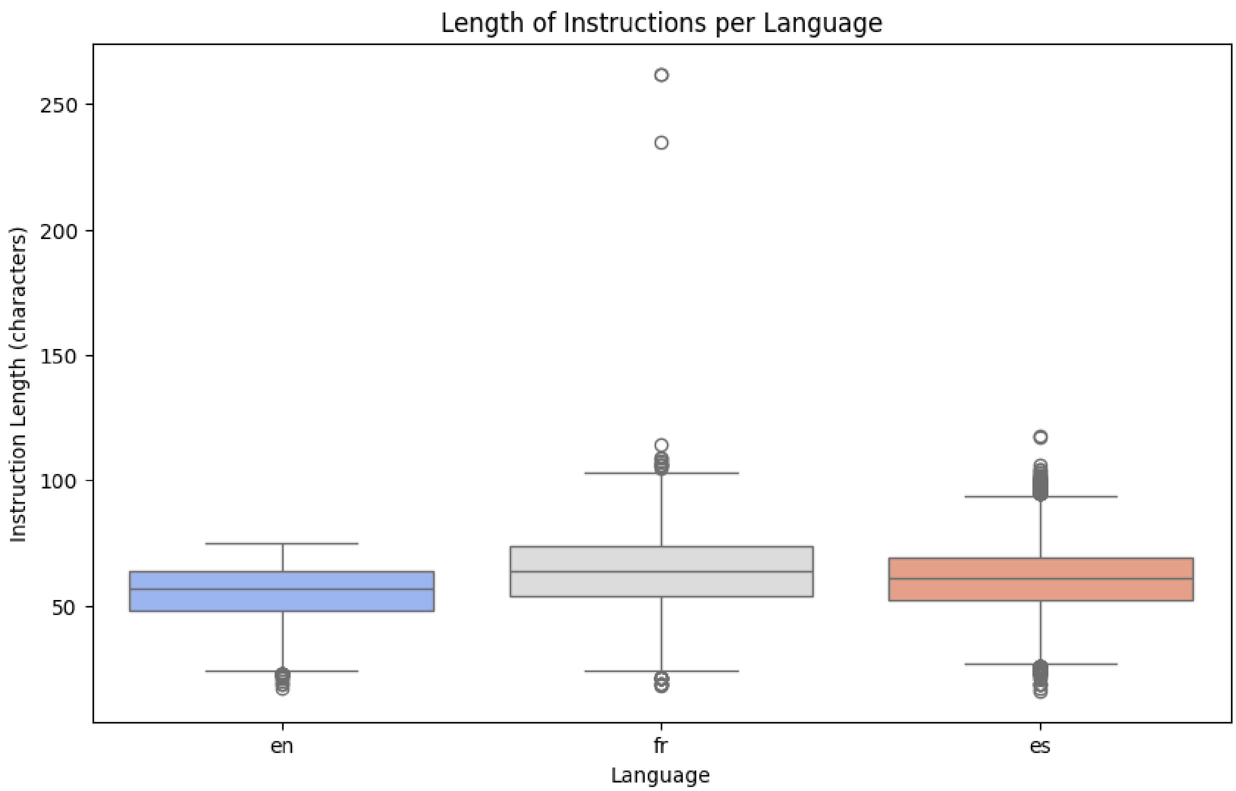

Figure 10. Length of Instructions per Language. Created using Python Libraries.

Interpretation: We can see that English instructions are the smallest of the three with an average instruction length around 55 characters. Spanish is slightly longer with an average instruction length around 60 characters and French is the longest of the three with an average instruction length around 65 characters. While Spanish instructions seem to have a larger number of outliers than French, French outliers are still much longer in length with some reaching around and upwards of 250 characters. Thus, we'd have to pay special attention to French instructions while we optimize our max_length for instructions during mBART training.

v. The fifth visualization is similar to the previous one but instead of instructions, we're computing character-wise length percentiles of responses across all languages. Similarly, this could help us see if certain languages have longer responses than others and decide the appropriate max_length for tokenizing the responses during mBART fine-tuning. Programmatic Implementation:

```
multilingual_df['response_length'] =
multilingual_df['response'].apply(len)
plt.figure(figsize=(10, 6))
sns.boxplot(data=multilingual_df, x='language',
y='response_length', palette='coolwarm')
plt.title('Length of Responses per Language')
plt.xlabel('Language')
plt.ylabel('Response Length (characters)')
plt.show()
```

The character-wise length of each response is calculated. Matplotlib is used for configuring the canvas properties and relevant details of the plot. A boxplot from Seaborn is used to visualize the length percentiles.

Plot:

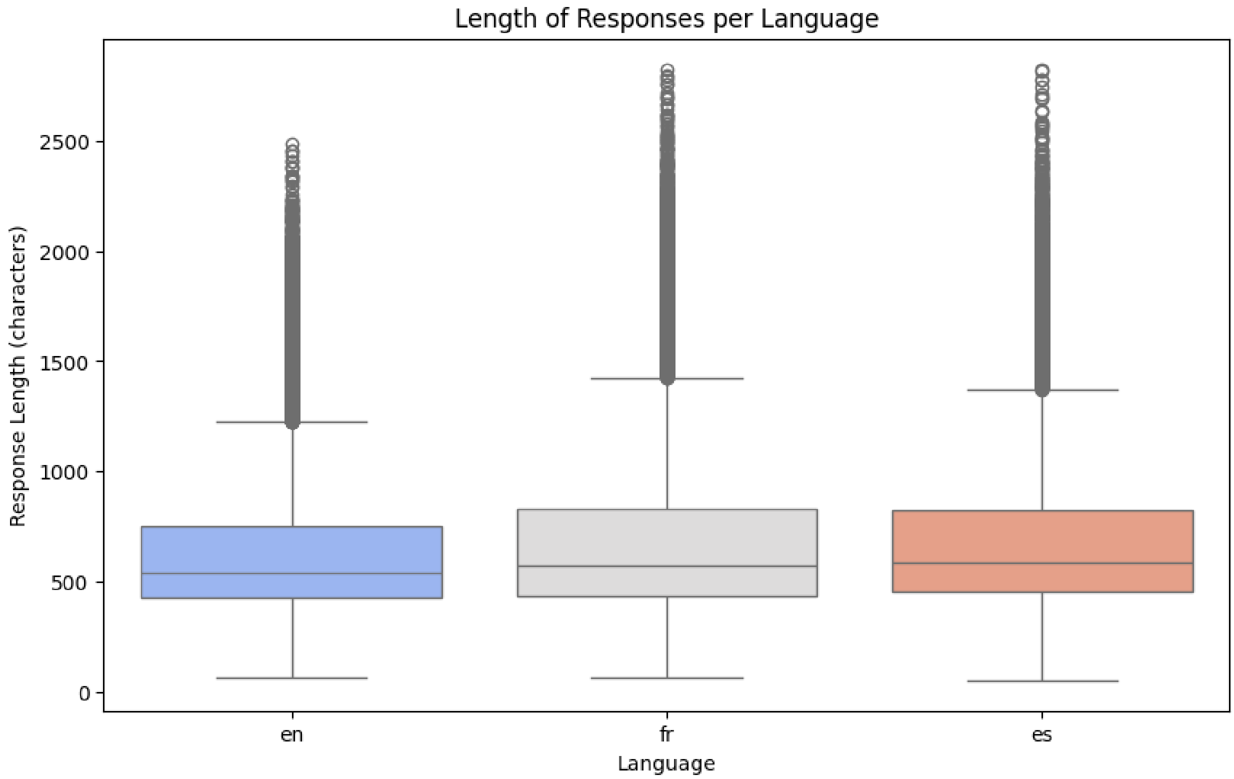

Figure 11. Length of Responses per Language. Created using Python Libraries.

Interpretation: We can see that English responses are the smallest of the three with an average responses length around 525 characters. Spanish and French seem to have the same average response length around 550 characters. A large number of responses for all three languages are outside the whisker range (1.5 times the interquartile range) suggesting that the dataset contains a large subset of outliers. French responses like before seem to have the largest response lengths around 2850 characters long with Spanish very close with around 2825 characters. Thus, both French and Spanish responses should be taken into consideration more while we optimize our max_length for responses during mBART training.

vi. The last set of visualizations are word clouds for data in instruction and response columns across all languages from the dataset. Word clouds are a fusion of the words that occur the most in any given data with the size of the word marking the no. of occurrences. This could be useful in allowing us to see what vocabulary is commonly present in the instructions and responses. We can also see whether the dominant vocabulary in the data matches between all three languages so that any biases between the languages may be found. Programmatic implementation:

```
nltk.download('stopwords')
from nltk.corpus import stopwords
languages = ['en', 'fr', 'es']
stopwords_dict = {
    'en': set(stopwords.words('english')),
    'fr': set(stopwords.words('french')),
    'es': set(stopwords.words('spanish'))
}
custom_stopwords = set(['en_xx', 'fr_xx', 'es_xx', '</s>'])
for lang in stopwords_dict:
    stopwords_dict[lang].update(custom_stopwords)
```

NLTK's stopwords data is used to fetch stopwords for all three languages. Stopwords are words that the word cloud should ignore. This is done so that common words such as "the", "is", etc. are not included in the word cloud as they do not provide any meaningful insights. Each language's respective stopwords are assigned to its code using the `stopwords_dict` dictionary. Custom stopwords containing special tokens that were added for mBART's training are also added to the `stopwords_dict` for the same reason as other stopwords.

```
def generate_word_cloud(text, stopwords, title):
    wordcloud = WordCloud(width=800, height=400,
                          background_color='white',
                          stopwords=stopwords,
                          colormap='viridis',
                          max_words=200,
                          random_state=42).generate(text)
    plt.figure(figsize=(10, 6))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')
    plt.title(title, fontsize=16, pad=20)
    plt.show()
    print("\n")
```

Function that generates the wordcloud using the column data, stopwords and a title. Plot canvas is set up using Matplotlib. The `WordCloud` object from the wordcloud library is initialized with parameters that affect the word cloud visually and they will be used alongside the `generate` function to generate the word cloud.

```
for lang in languages:
    # Filtering dataset for the specific language
```

```
    df_lang = multilingual_df[multilingual_df['language'] ==
lang]
    instructions_text = '
'.join(df_lang['instruction'].astype(str).tolist()).lower()
    responses_text = '
'.join(df_lang['response'].astype(str).tolist()).lower()
    generate_word_cloud(instructions_text,
stopwords_dict[lang], f'Frequent Words in Instructions for
Language: {lang.upper()}')
    generate_word_cloud(responses_text, stopwords_dict[lang],
f'Frequent Words in Responses for Language: {lang.upper()}')
```

The last step is word cloud generation. Word clouds require text in one long sequence and so all the instructions and responses are concatenated into their own single continuous string and all words are also made lowercase to ensure consistency. `generate_word_cloud` function is called with the obtained text, stopwords of the respective language and a custom title. This process is continued iteratively until word clouds for instructions and responses are generated for all three languages.

Plots:

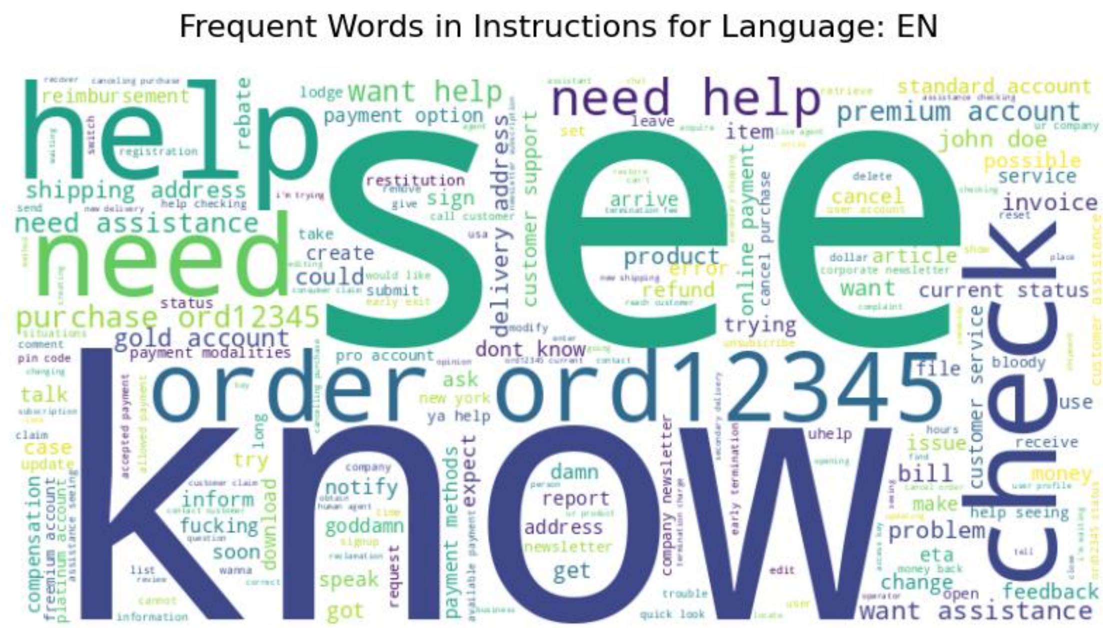

Figure 12. Frequent Words in Instructions for English. Created using Python Libraries.

Interpretation: Words such as “see”, “know”, “help”, “check”, “need” in left to right order seem to be the most commonly occurring words in English instructions.

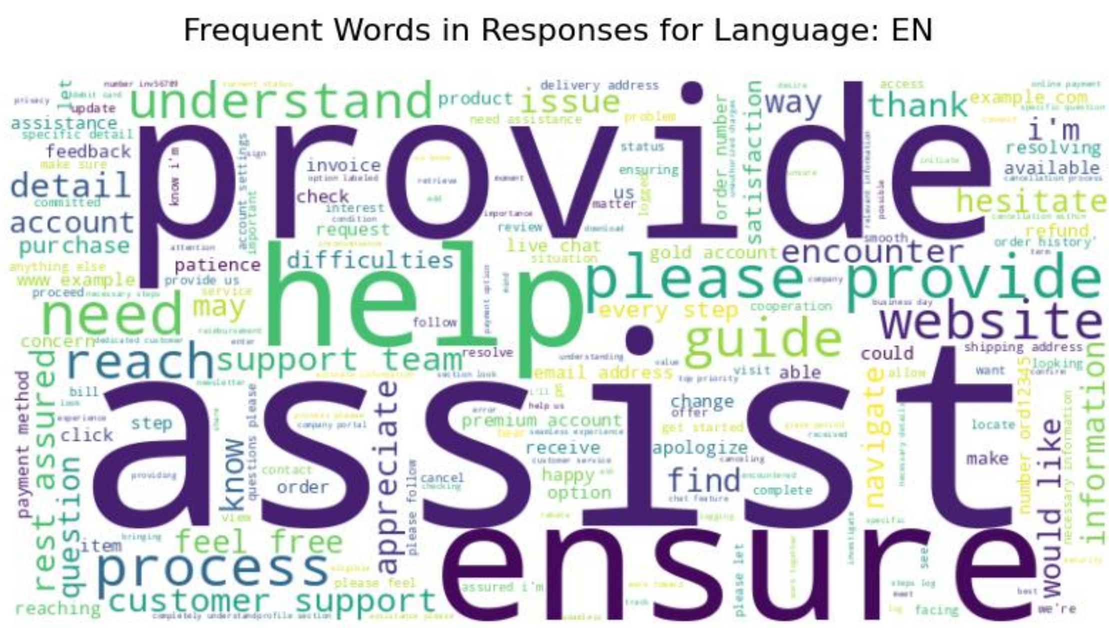

Figure 13. Frequent Words in Responses for English. Created using Python Libraries.

Interpretation: Words such as “assist”, “provide”, “help”, “ensure”, and “process” in left to right order seem to be the most commonly occurring words in English responses.

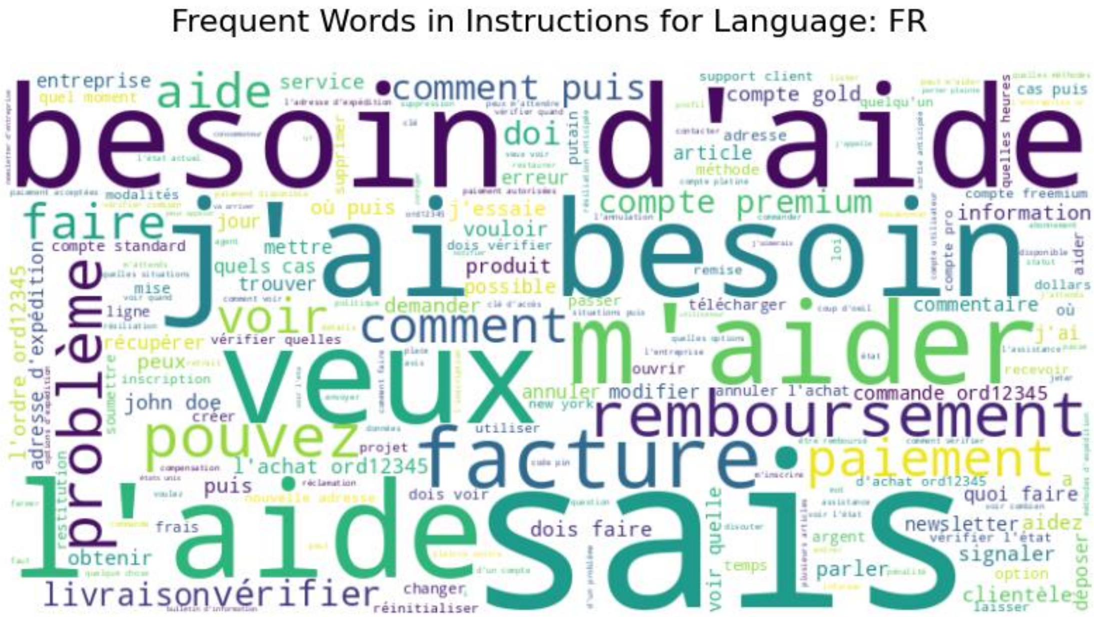

Figure 14. Frequent Words in Instructions for French. Created using Python Libraries.

Interpretation: Words such as “sais”, “besoin d’aide”, “j’ai besoin”, “veux”, “l’aide”, “m’aider” in left to right order seem to be the most common words in French instructions.

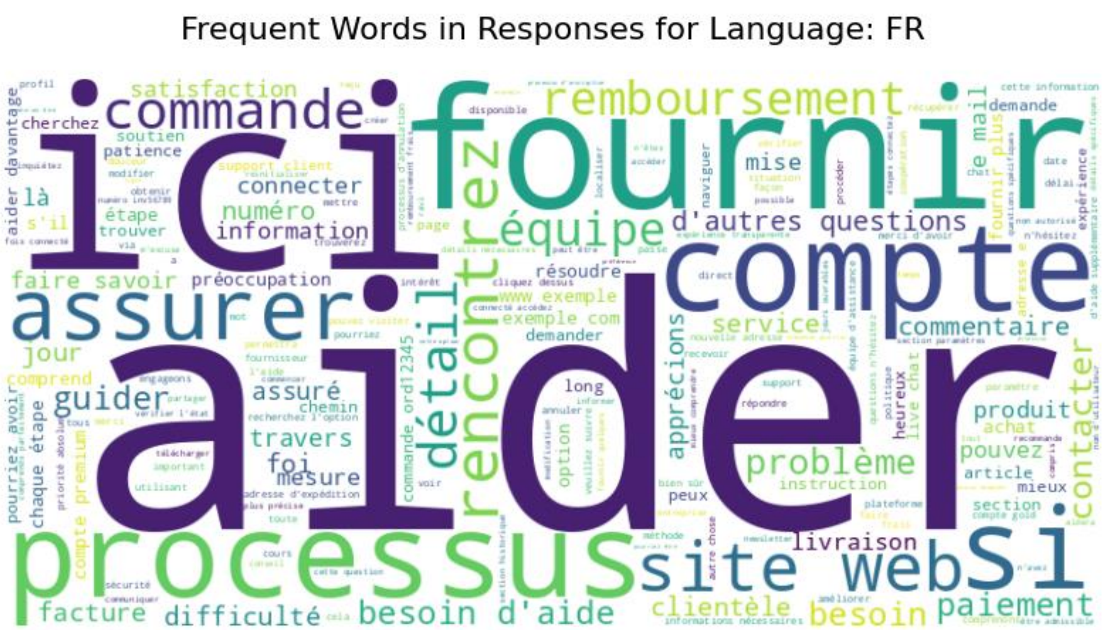

Figure 15. Frequent Words in Responses for French. Created using Python Libraries.

Interpretation: Words such as “aider”, “ici”, “fournir”, “compte”, “processus”, “assurer” in left to right order seem to be the most common words in French responses.

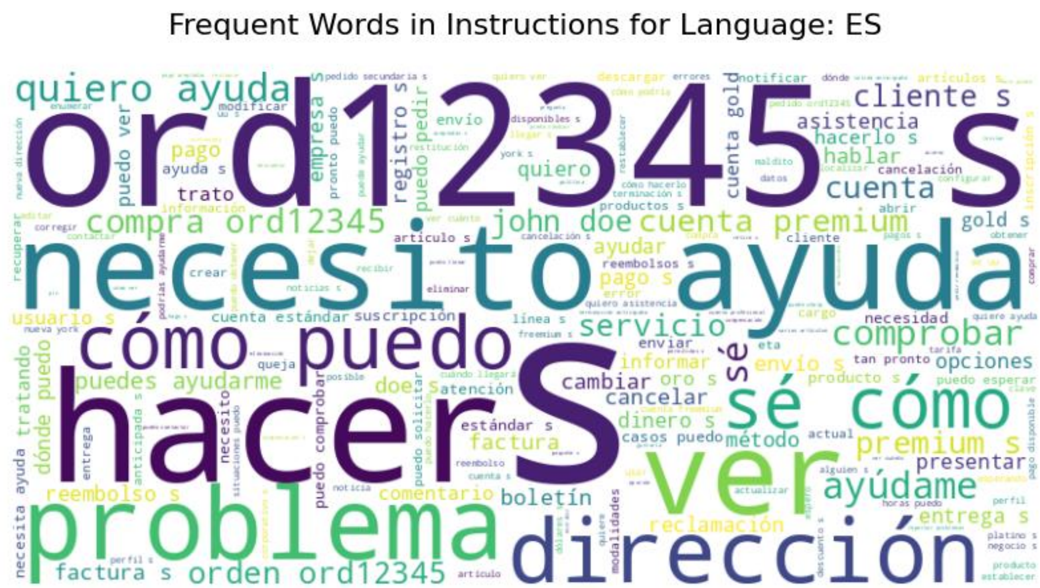

Figure 16. Frequent Words in Instructions for Spanish. Created using Python Libraries.

Interpretation: Words such as “hacer”, “necesito ayuda”, “problema”, “direccion”, “como puedo” in left to right order seem to be the most common words in Spanish instructions.

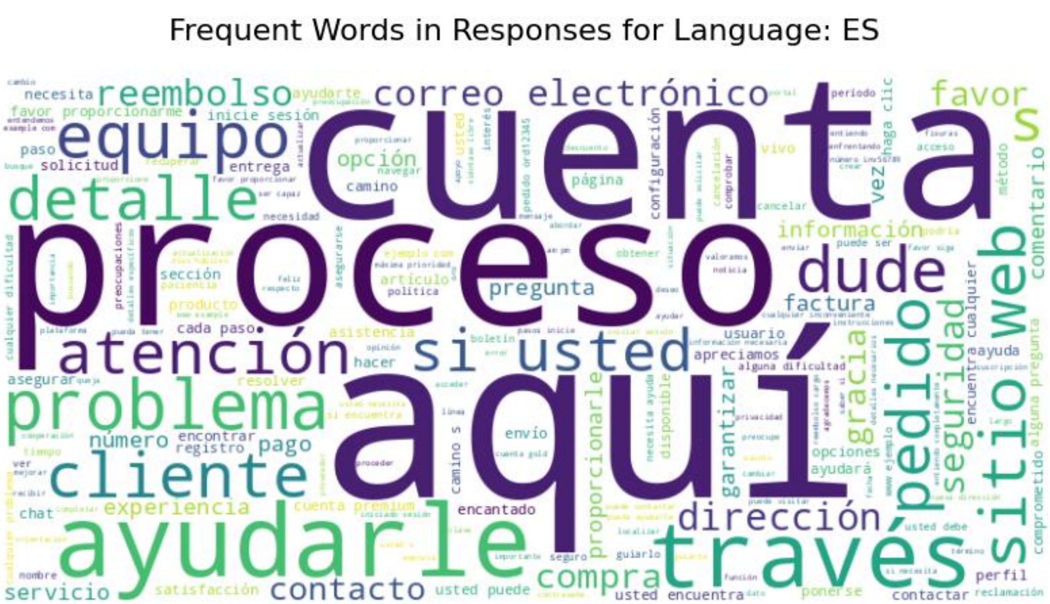

Figure 17. Frequent Words in Responses for Spanish. Created using Python Libraries.

Interpretation: Words such as “cuenta”, “aqui”, “proceso”, “ayudarle” and “traves” in left to right order seem to be the most commonly occurring words in Spanish responses.

Comparison of all the wordclouds:

i. Instruction Word Clouds:

a. The most commonly occurring words in French Instructions can be translated as follows:

- “sais” – know
- “besoin d’aide” – need help
- “j’ai besoin” – I need
- “veux” – want
- “l’aide” – help
- “m’aider” – help me

b. The most commonly occurring words in Spanish Instructions can be translated as follows:

- “hacer” – do
- “necesito ayuda” – I need help
- “problema” – problem
- “direccion” – address
- “como puedo” – how can I

c. The most commonly occurring words in English Instructions are “see”, “know”, “help”, “check”, and “need”.

Comparing the most common words among all three languages, we can see a lot of overlap between them such as “know”, “help”, and “need”. Thus we can conclude that our instructions data across the languages is fairly bias-free.

ii. Responses Word Clouds:

a. The most commonly occurring words in French Responses can be translated as follows:

- “aider” – help
- “ici” – here
- “fournir” – provide
- “compte” – account
- “processus” – process
- “assurer” – ensure

b. The most commonly occurring words in Spanish Responses can be translated as follows:

- “cuenta” – account
- “aqui” – here
- “proceso” – process
- “ayudarle” – help him
- “traves” – crosses

c. The most commonly occurring words in English Responses are “assist”, “provide”, “help”, “ensure”, and “process”.

Comparing the most common words among all three languages, we can see great overlap between them such as “provide”, “help”, “ensure”, and “process”. Thus we can conclude that our responses data across the languages is also bias-free to a good degree.

3.3.1.2. Analyze Dataset with mBART

i. The Multilingual Customer Support Dataset as it is has been visualized thoroughly using a variety of visualizations. The next step is to identify how it would change with respect to the mBART model so that we can finalize our training arguments. We can do this by first using the mBART Tokenizer to tokenize the Instructions and Responses from the Multilingual Dataset, finding each of their length values, and storing them in new columns.

Programmatic Implementation:

```
from transformers import MBart50TokenizerFast
model_path = "models/mbart-large-50"
tokenizer = MBart50TokenizerFast.from_pretrained(model_path,
local_files_only=True, padding_side='right')
multilingual_df['instruction_length'] =
multilingual_df['instruction'].apply(lambda x:
len(tokenizer.tokenize(x)))
multilingual_df['response_length'] =
multilingual_df['response'].apply(lambda x:
len(tokenizer.tokenize(x)))
```

We load the appropriate tokenizer from a local directory for mBART, in this case it is the `MBart50TokenizerFast` which was chosen for its efficiency in tokenizing multilingual text quickly. Then, the tokenizer is used to tokenize the instructions and responses whose tokenized lengths are computed using `len()` and these lengths are stored in new columns `'instruction_length'` and `'response_length'` respectively.

ii. Now that we have the tokenized lengths of the instructions and responses, we can calculate Length Percentiles for the tokenized Instructions and Responses which may help us in determining max lengths for the model inputs during training. Programmatic Implementation:

```
percentiles = [95, 96, 97, 98, 99, 100]
response_percentile_values =
multilingual_df['response_length'].quantile([p/100 for p in
percentiles])
instruction_percentile_values =
multilingual_df['instruction_length'].quantile([p/100 for p
in percentiles])
print("Response Length Percentiles:")
print(response_percentile_values)
print("\nInstruction Length Percentiles:")
print(instruction_percentile_values)
```

We are calculating 95th to 100th Percentiles for both Instructions and Responses as such a list with those values is initialized. These values are converted into percentages and passed iteratively to the `quantile()` function from `pandas` library. This is done for both `'instruction_length'` and `'response_length'` columns and the percentile values are printed for both.

Output:

```
Response Length Percentiles:
0.95    303.0
```

```
0.96    327.4
0.97    372.0
0.98    405.0
0.99    441.0
1.00    627.0
```

Interpretation: We see that the tokenized response lengths increase rapidly even at the highest percentile levels. As previously mentioned, Transformer models have a limit of 512 tokens and our response must fit within that limit. So, we choose 99th percentile – 441 as the max_length for our responses to ensure almost all responses are used in training while also avoiding mBART's limitation of 512 tokens.

```
Instruction Length Percentiles:
0.95     19.0
0.96     20.0
0.97     20.0
0.98     21.0
0.99     22.0
1.00    114.0
```

Interpretation: We see that the tokenized instruction lengths remain relatively stable with marginal increases from 95th percentile all the way to 99th percentile and increase very sharply at the 100th percentile. We choose 97th percentile – 20 as the max_length for our instructions because the difference between 97th and 99th percentiles is quite minor and will not contribute significantly to the model's performance and the 100th percentile – 114 is simply too large a value to be considered for just 1% of the instructions.

3.4. mBART model Finetuning

This section involves finetuning the mBART model on the Multilingual Customer Service Dataset. The training is done on Google Colab utilizing the powerful NVIDIA A100-SXM4-40GB GPU to expediate the time taken and avoid Out of Memory issues. The dataset is tokenized and split into train and test sets. The training arguments are configured with optimal values and this section will explain the reasoning behind their choosing. Explanations for the arguments can be found in section 2.3.1.1. Once training is done, model is saved. Find the flow-chart that visualizes this section below:

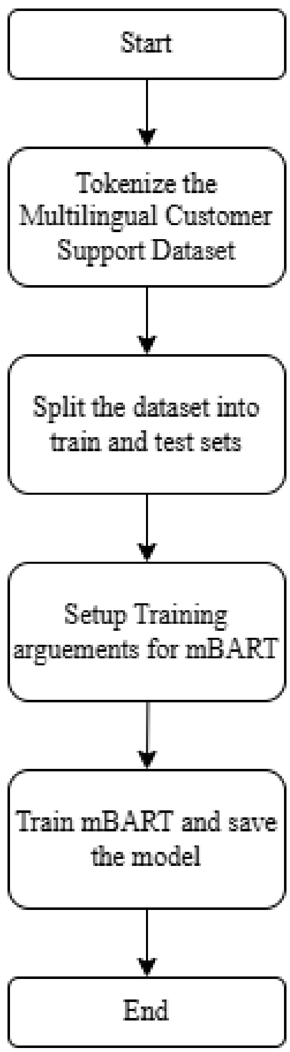

Figure 18. Flow-Chart of mBART model Finetuning. Created using [13]

3.4.1. Implementation

i. The first step involves loading the mBART model and tokenizer. Then the dataset is tokenized with the tokenizer by means of a function:

```
from transformers import MBartForConditionalGeneration,
MBart50TokenizerFast
from datasets import load_dataset
model_path = "drive/MyDrive/Multi-lingual Customer Service
Chatbot - Colab/models/mbart-large-50"
tokenizer = MBart50TokenizerFast.from_pretrained(model_path,
padding_side='right')
model =
MBartForConditionalGeneration.from_pretrained(model_path)
def tokenize_function(examples):
    inputs = tokenizer(examples['instruction'],
truncation=True, padding=False, max_length=20)
    targets = tokenizer(examples['response'],
truncation=True, padding=False, max_length=440)
    inputs["labels"] = targets["input_ids"]
    return inputs
dataset = load_dataset('csv',
data_files='drive/MyDrive/Multi-lingual Customer Service
Chatbot -
Colab/data/Multilingual_Customer_Support_Training_Dataset.csv
')
```

```
tokenized_datasets = dataset.map(tokenize_function,
batched=True)
```

The libraries `MBartForConditionalGeneration` & `MBart50TokenizerFast` are used for loading the model and tokenizer respectively. The `tokenize_function` takes a data sample, tokenizes the instruction and response values, adds the tokenized responses as labels to the inputs dictionary and returns the tokenized data sample. The values from labels key are what mBART will use to know what output text it is supposed to generate given an input instruction. We also use the `max_length` values (20 & 440) that were previously determined to be optimal in the last section. Then the `datasets` library is used to load the dataset and apply the `tokenize_function` to each data sample.

ii. Now that we have the tokenized dataset, the next step is to prepare it for fine-tuning mBART on it. First, we split the dataset by each language, shuffle it, and add them together again. This is to ensure that the dataset is balanced with variability in inputs to avoid overfitting. Then, the dataset is split into train and test sets. 80% of the dataset is used for training and 20% of the dataset is used for validation.

```
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)
```

Here, we set the seeds of Python's `random, numpy & pytorch` to the same number 42. This is done to ensure that the dataset used for training the model is the same one across every run so that we can reproduce the same model and compare them to evaluate what training arguments are the most optimal.

```
sample_size_per_language = 26872
en_samples = tokenized_datasets["train"].filter(lambda
example: example['language'] ==
'en').shuffle(seed=42).select(range(sample_size_per_language)
)
fr_samples = tokenized_datasets["train"].filter(lambda
example: example['language'] ==
'fr').shuffle(seed=42).select(range(sample_size_per_language)
)
es_samples = tokenized_datasets["train"].filter(lambda
example: example['language'] ==
'es').shuffle(seed=42).select(range(sample_size_per_language)
)
full_dataset = concatenate_datasets([en_samples, fr_samples,
es_samples])
```

The same sample size is set for all languages to ensure the dataset is bias-free. The tokenized dataset is filtered by language, shuffled with a seed value of 42 and saved separately for each language. Then all three languages' data is combined using `concatenate_datasets` to form a full dataset.

```
train size = int(0.80 * len(full_dataset))
eval_size = len(full_dataset) - train_size
train_dataset = full_dataset.select(range(train_size))
```

```
eval_dataset = full_dataset.select(range(train_size,
len(full_dataset)))
```

The train set and eval set's size are calculated and applied to the full dataset to create the train and eval sets.

iii. Since the dataset needed to train the model is ready, we move on to the training arguments and configuring those values for each parameter. These arguments values were found to be the most optimal and ultimately chosen after many failed iterations in trying to create a model that performs well.

```
training_args = TrainingArguments(
output_dir="drive/MyDrive/Multi-lingual Customer Service
Chatbot - Colab/models",
evaluation_strategy="epoch",
save_strategy="epoch",
learning_rate=3e-5,
per_device_train_batch_size=12,
per_device_eval_batch_size=12,
num_train_epochs=7,
save_total_limit=3,
fp16=True,
weight_decay=0.01,
logging_steps=200,
load_best_model_at_end=True,
gradient_accumulation_steps=5,
```

```
warmup_steps=500,
report_to="none",
gradient_checkpointing=True,
lr_scheduler_type="cosine",
accumulation_steps=10,
)
```

These values for the training arguments were chosen due to the following reasonings:

- `evaluation_strategy="epoch"` – Evaluating the model at the end of each epoch is standard evaluation procedure for Neural Networks. It's a good point in time to check if the model is coming along well.
- `save_strategy="epoch"` – The model is saved at the end of an epoch so that we can resume training in case the training stops due to any technical issues.
- `learning_rate=3e-5` – A learning rate of three times ten to the power of negative five is a conventional learning rate for mBART and also found to work the best for this task after trying out other learning rates such as 1e-5, 2e-5 and 5e-5.
- `per_device_train_batch_size=12` – Multiple batch sizes were tried including 8, 16, 32 but 12 was found to work the best as it allowed for 7 epochs of training while still avoiding Out of Memory issues on the GPU.
- `per_device_eval_batch_size=12` – Same value as train size for consistency.
- `num_train_epochs=7` – The model's performance after repeat attempts was found to improve more with higher number of epochs. This was the highest number of epochs that could be achieved with the memory availability of the GPU.

- `save_total_limit=3` – Save limit for the model. Even though `save_strategy` is set to "epoch", saving the model's weights at every checkpoint consumes a large amount of memory and since I'm using Google Drive for storage, I was limited to 15GB of storage, and thus this number.
- `fp16=True` – Mixed Precision turned on for lower memory consumption and faster training.
- `weight_decay=0.01` – Standard value to avoid overfitting
- `logging_steps=200` – Reasonable value that allows to keep track of model's training stats while also making sure not to overwhelm the system with logs.
- `load_best_model_at_end=True` – This loads the best performing model at the end and that model will be saved. This is done so that even if the loss values get worse, we still end up with the best model that was obtained from training.
- `gradient_accumulation_steps=5` – This accumulates the current size of our batch – 12 five times to simulate a batch size of 60. This allows the model to learn on more data samples at once, leading to better performance
- `warmup_steps=500` – A value of 500 samples gives enough time for the model to slowly warmup to the learning rate and avoid unstable learning at the beginning.
- `report_to="none"` – Set to none to avoid unnecessary overhead during training as reporting to organizations such as WandB, TensorBoard, etc. could cause the training to become longer.
- `gradient_checkpointing=True` – Another argument that reduces the memory consumed during training.

- `lr_scheduler_type="cosine"` – Cosine curve starts high and decreases slowly and so learning rate scheduler is set to cosine to allow the model to converge smoothly at the end of each epoch.
- `eval_accumulation_steps=10` – Another parameter that helps save memory by accumulating batches before evaluating loss. In this case, for our batch size of 12, the accumulation steps is 10, so 12 x 10 = 120 samples will be used in evaluating before updating the loss.

```
data_collator = DataCollatorForSeq2Seq(tokenizer,
model=model, padding=True)
```

The `DataCollatorForSeq2Seq` object from Transformers allows to enable dynamic padding for the inputs where the inputs only get padded to the longest input in their batch and this helps save memory.

```
trainer = Trainer(
       model=model,
       args=training_args,
       train_dataset=train_dataset,
       eval_dataset=eval_dataset,
       data_collator=data_collator,
      callbacks =
      [EarlyStoppingCallback(early_stopping_patience=2)]
)
```

The `Trainer` object is used to initialize model training:

- `model=model` – The loaded mBART model is set as the model to be trained
- `train_dataset=train_dataset` – Our train set is used for training.
- `eval_dataset=eval_dataset` – Our test set is used for testing.
- `data_collator=data_collator` – The initialized `DataCollatorForSeq2Seq` object is set as the data collator.
- `callbacks=[EarlyStoppingCallback(early_stopping_patience=2)]` – The training is configured to stop if the validation loss doesn't improve for two epochs in a row.

iv. We have set up everything needed for training to commence. Now, we can start training mBART and save the model once it's done.

```
!PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True
trainer.train()
trainer.save_model(model_dir)
tokenizer.save_pretrained(model_dir)
```

One more setting that helps the training process is the `PYTORCH_CUDA_ALLOC_CONF = expandable_segments:True` shell command. The environment variable `PYTORCH_CUDA_ALLOC_CONF` is what PyTorch uses to configure CUDA memory behavior and setting it to `expandable_segments:True` allows it to dynamically allocate memory as and when it needs during training, reducing the chances of Out of Memory errors. `train()` starts the training and once it's done `save_model` saves the trained model including its weights and structure to the directory specified, and `save_pretrained` saves the tokenizer configs to ensure we can produce the same model again.

Output:

[7525/7525 2:48:04, Epoch 7/7]

Epoch	**Training Loss**	**Validation Loss**
1	1.280400	1.306809
2	0.738200	0.787249
3	0.654700	0.748505
4	0.590900	0.693028
5	0.563000	0.676866
6	0.539300	0.670430
7	0.529600	0.670399

Interpretation: We notice that the Training loss starts at 1.280400 in the 1st epoch and ends at 0.529600 at the 7th epoch, meaning that our model has learned effectively over the course of training, minimizing its loss every step of the way. The same pattern can be observed with the Validation loss as well but its loss decreases slower than training loss and plateaus at epoch 5 with 0.676866 loss value. There are improvements in loss after this point but they are marginal so validation loss reaches an optimal point. At the end, Convergence between training and validation loss can be observed as the gap between the two consistently lowered at each epoch, so we can conclude that the model performs well when it encounters unseen data.

3.5. Chatbot Deployment

The last section of the Architecture is deployment of the model we trained on a chatbot interface to allow user interaction. For our chatbot interface, the Python library Gradio is used which allows creation of a custom web interface and the code required is minimal. The first step is to define the back-end that runs our web interface. Then, once we have it working, we design the interface to have user-friendly structure and styles. Find the flow-chart that visualizes this section below:

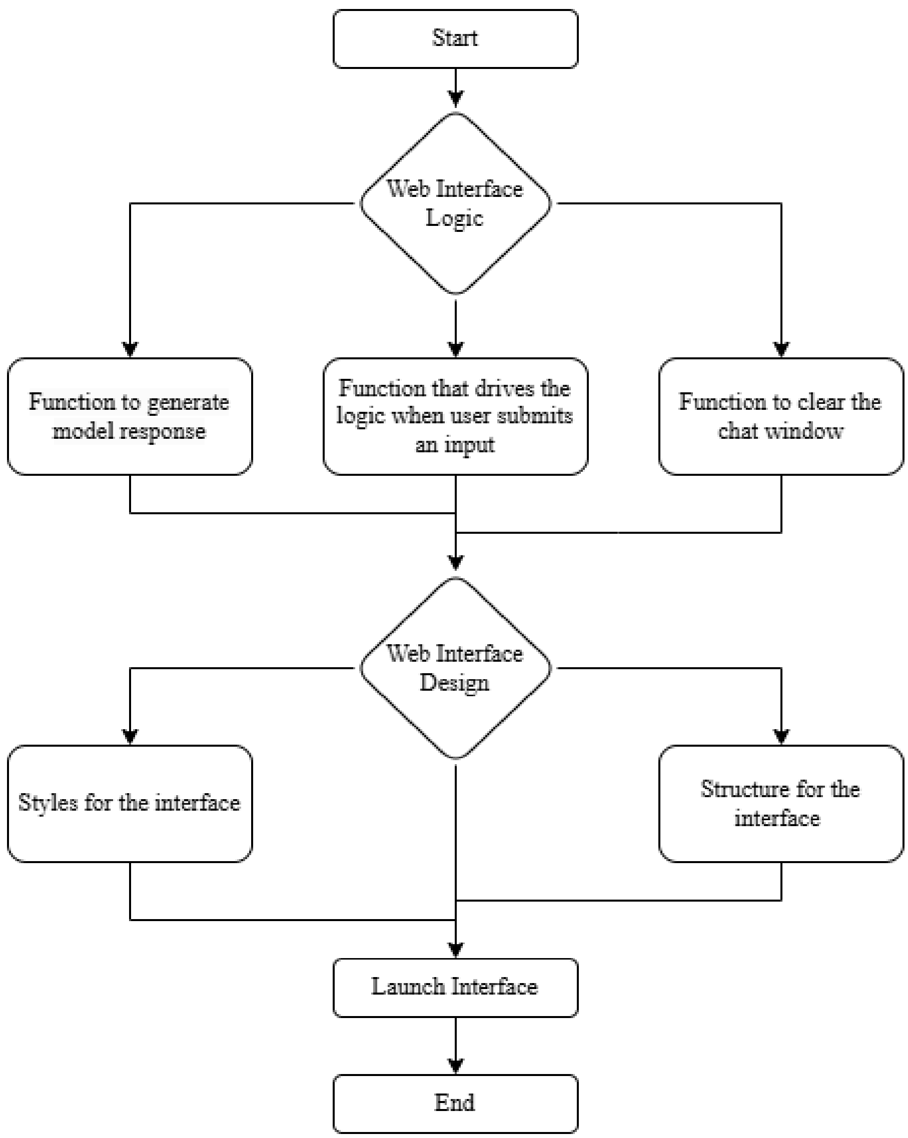

Figure 19. Flow-Chart of Chatbot Deployment. Created using [13]

3.5.1. Implementation

Implementation of this section can be divided into two parts with their own sub-parts:

1. Web Interface Logic
   i. Define function to generate model response from user input
   ii. Define function that drives the web interface logic on user input submission
   iii. Define function to clear the chat
2. Web Interface Design
   i. Define Styles for the interface
   ii. Define Structure for the interface

3.5.1.1. Web Interface Logic

i. The first step involves defining a function that generates a model response from a user's input. A number of steps are involved in generating the final response for the web interface from the user's input. Programmatic Implementation:

```
model =
MBartForConditionalGeneration.from_pretrained(model_dir).to('
cuda')
tokenizer = MBart50TokenizerFast.from_pretrained(model_dir)
```

Load the mBART model and tokenizer.

```
def generate_response(input_text, max_length=256,
num_beams=5, min_length=125):
```

Function that generates response with user input.

```
detected_lang, _ = langid.classify(input_text)
language_code = {
    "en": "en_XX",
    "fr": "fr_XX",
    "es": "es_XX"
}.get(detected_lang, "en_XX")
tokenizer.src_lang = language_code
```

The Python library `langid` is used to detect the language of the user input. Appropriate language token is assigned to the mBART tokenizer using the detected language.

```
model_inputs = tokenizer(input_text, return_tensors="pt",
padding=True)
model_inputs = {key: value.to('cuda') for key, value in
model_inputs.items()}
```

The tokenizer tokenizes the input text and it is passed to the GPU for response generation.

The generate function is used to generate the response text. It takes a number of parameters that affect the quality of the response. Similarly to training, these arguments values were found to be the most optimal and ultimately chosen after many failed attempts in trying to generate coherent and accurate responses. Refer to Section 2.1.3.2 for explanations for these arguments, this section focuses on the values used.

```
generated_tokens = model.generate(
  input_ids=model_inputs['input_ids'],
```

```
    attention_mask=model_inputs['attention_mask'],
    forced_bos_token_id =
    tokenizer.lang_code_to_id[language_code],
     eos_token_id=tokenizer.eos_token_id,
     min_length=min_length,
     max_length=max_length,
     num_beams=num_beams,
     no_repeat_ngram_size=2,
     early_stopping=True
)
```

These values for the generation arguments were chosen due to the following reasonings:

- `input_ids=model_inputs['input_ids']` – The tokenized input is passed to the models as `input_ids`
- `attention_mask=model_inputs['attention_mask']` – This includes the padding that was added to the input by the tokenizer. This is simply added to ensure the model ignores the padding tokens.
- `forced_bos_token_id=tokenizer.lang_code_to_id[language_code]` – The detected language is added as the Beginning Of Sequence token for the response so that the model knows to generate the response in the detected language.

- `eos_token_id=tokenizer.eos_token_id` – The End Of Sequence token is added to the response so that the model knows when to stop response generation. This is to prevent unnecessarily long responses.
- `min_length=min_length` – This argument ensures the model generates a response that is at least as long as the value specified. In this case, the default value set is 124, which ensures the model generates responses that are at least 124 tokens long.
- `max_length=max_length` – Set the maximum length that the generated response can have. Here, its set to 440 to ensure 99% of the responses available to the model are considered during generation.
- `num_beams=num_beams` – Set the number of beams to be searched during beam search.
- `no_repeat_ngram_size=2` – This will ensure that the same word will not be repeated twice in a row, leading to more coherent responses.
- `early_stopping=True` – Ensure the model stops beam search once it finds a satisfactory response with a good cumulative likelihood score.

```
response = tokenizer.decode(generated_tokens[0],
skip_special_tokens=True)
response = re.sub(r'(?<!\d)(\b\d{1,2}\.\s)', r'\n\t\1',
response)
response = re.sub(r'\*', '', response)
return response
```

The tokenizer decodes the response, it is formatted to have line-breaks between points and to not have asterisks in text. The final response ready for the interface is returned.

ii. The next function that is needed is a driver function that defines the logic behind what happens when the user submits the input text.

```
conversation history = []
```

A global list is defined to store the chatbot interactions.

```
def chatbot_interface(user_input):
```

Function that drives the interface logic

```
    global conversation_history
```

Using the global list `conversation_history` in function scope.

```
    response = generate_response(user_input)
```

Using the previously defined `generate_response` function to generate response for the user input.

```
    conversation_history.append((user_input, response))
```

Adding the input text and the generated response to the global list as a tuple. This is done so that the data can be displayed in the conversation window.

```
    conversation_display = ""
    for user, bot in conversation_history:
```

```
        conversation_display += f"""
        <div class="user-name">You</div>
        <div class="user-message">{user}</div>
        <div class="bot-name">Chatbot</div>
        <div class="bot-message">{bot}</div>
        """
    return conversation_display
```

Define a string for the HTML Markup and pass in the input-text and response pairs dynamically. Return this string for use in the interface.

iii. The final function needed for our interface is the clear chat function to allow users the restart the conversation if they so choose.

```
def clear_chat():
    global conversation_history
    conversation_history = []
    return ""
```

A simple function that runs when the clear chat button is pressed. The global list `conversation_history` is used in the local scope here as well and it is simply reinitialized to an empty list, and this clears the entire conversation.

3.5.1.2. Web Interface Design

i. Since we have the logic behind the web interface ready, now we proceed to creating it. To implement the interface, we make use of the Blocks API from Gradio which allows for flexible and modular UIs. Then we setup CSS styling for the HTML elements of our Blocks interface. Programmatic Implementation:

```
with gr.Blocks() as demo:
```

We initialize the Blocks interface in a variable `demo`.

```
    demo.css = """
    .user-name {
        font-weight: 550;
        padding-right: 10px;
        text-align: right;
        font-style: italic;
    }
    .bot-name {
        padding-left: 10px;
        font-style: italic;
        font-weight: 550;
        text-align: left;
    }
    .user-message {
        background-color: #fde68a;
        text-align: right;
        padding: 10px;
        margin: 5px;
        border-radius: 10px;
```

```
    max-width: max-content;
    margin-left: auto;
}
.bot-message {
    background-color: #e0f7fa;
    text-align: left;
    padding: 10px;
    margin: 5px;
    border-radius: 10px;
    max-width: 70%;
    margin-right: auto;
    white-space: pre-wrap;
}
#chat-container {
    border: 1px solid #cccccc;
    min-height: 65vh;
    max-height: 65vh;
    overflow-y: scroll;
    padding: 15px;
    background-color: #ffffff;
    margin-bottom: 15px;
    border-radius: 10px;
}
#user-input {
```

```
        width: 100%;
        padding: 10px;
    }
    .center-text {
        text-align: center;
    }
    #main-container {
        width: 50%;
        margin: 0 auto;
    }
    #user-container {
        display: grid;
        grid-template-columns: 1fr 10rem
    }
    #btns {
        max-width: 10rem;
    }
    #submit-btn, #clear-btn {
        max-width: 10rem;
    }
    """
```

We add CSS for our entire page to the `demo` variable.

ii. We move to the final step which is to define the layout and structure of our interface. We can make use of Gradio’s Row and Column components to define the layout and components like Markdown, HTML, Textbox & Button for adding other DOM elements.

```
with gr.Row(elem_id="main-container"):
```

We set the container that will contain the entire interface and give it an unique id for styling.

```
    with gr.Column(scale=3):
        gr.Markdown("<div class='center-
        text'><h1>Multilingual Customer Service Chatbot -
        Suryakumar Selvakumar</h1></div>")
        gr.Markdown("<div class='center-text'>Interact with
        the multilingual chatbot in your preferred language
        (English, French, or Spanish).</div>")
```

We set the main column that will contain our interface elements and give it a scale of 3 which ensures it is large enough to accommodate our elements. An `<h1>` tag with title of the interface and a `<div>` tag with an instruction is added using the `Markdown()` component.

```
        with gr.Column(elem_id="chat-container"):
            chatbot_output = gr.HTML(label="Chatbot
            Conversation")
```

We set the chat container using a `Column()` component and assign a unique id to it for styling. We also create `chatbot_output` HTML element for displaying our user-chatbot interactions dynamically.

```
    with gr.Row(elem_id="user-container"):
        user_input = gr.Textbox(lines=1,
        placeholder="Enter your question here...",
        label="", elem_id="user-input")
```

A new row using the `Row()` component is created and a unique id is assigned to it for styling. A Textbox input to created to enter user input and is assigned to a variable.

```
        with gr.Column(elem_id="btns"):
            submit_button = gr.Button("Submit",
            elem_id="submit-btn")
            clear_button = gr.Button("Clear Chat",
            elem_id="clear-btn")
```

In the same row, A new column is created to house the buttons that execute the interface logic. We create two buttons, Submit and Clear chat respectively and assign variables to them for use.

```
submit_button.click(fn=chatbot_interface, inputs=user_input,
outputs=chatbot_output).then(lambda: "", inputs=None,
outputs=user_input)
```

```
clear_button.click(fn=clear_chat, inputs=None,
outputs=chatbot_output)
```

This code is what connects our back-end logic with the front-end interactions. When Submit button is pressed, it fetches the input from `user_input` i.e., the Textbox, runs the `chatbot_interface` function with the input, sends the output returned to `chatbot_output` i.e., our HTML element.

```
demo.launch()
```

At last, the interface is launched using the `launch()` function which is called on our Blocks variable `demo`.

Web Interface:

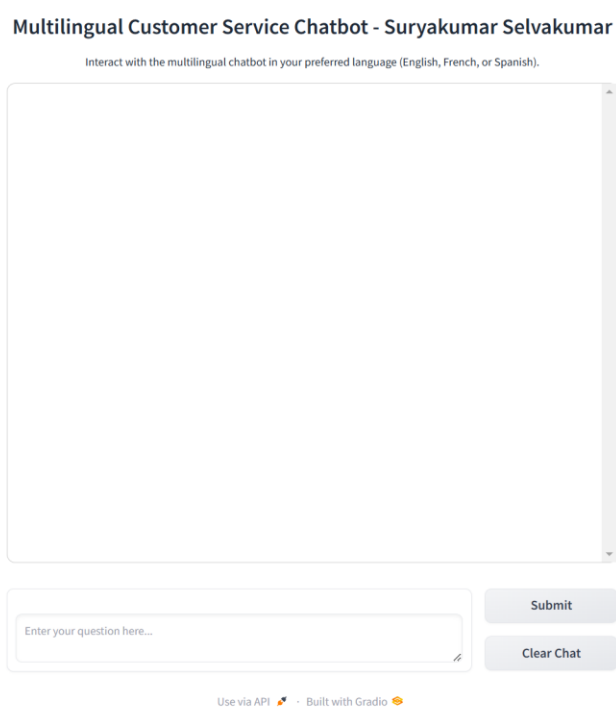

Figure 20. Chatbot Web Interface. Created with Python & Gradio.

4. METHODOLOGY, RESULTS AND ANALYSIS

This section includes the Methodology used to test the Chatbot model - mBART, Results of the testing, and Analysis of the results. The methodology will focus on only the tests, the results will focus on only the results obtained, and analysis will focus on analyzing the results. Same number order will be followed across the three sections to ensure consistency between tests, results, & analyses.

4.1. Methodology

The trained mBART model was tested comprehensively in an array of manners. Immediately after training, Sanity tests were performed to determine if the model is working and whether further testing can be carried out. Each of the model iterations were put to this test and only after passing all of them were they considered for further testing. If the model passes the Sanity tests, then the next set of tests to be carried out involves calculation of a number of industry relevant metrics on model's responses to instructions from a subset of the Multilingual Dataset, evaluating overall model performance. Then the next set of tests evaluates the language-specific performance of the model by calculating the same metrics for model responses to a subset of instructions in each language. Final testing involves testing the chatbot interface, ensuring that it performs as intended in the architecture.

Methodology can be divided into three parts:

1. Sanity Testing
2. Overall Model Performance Evaluation
3. Language-Specific Model Performance Evaluation

4. Interface Testing

Find the flow-chart that visualizes this section below:

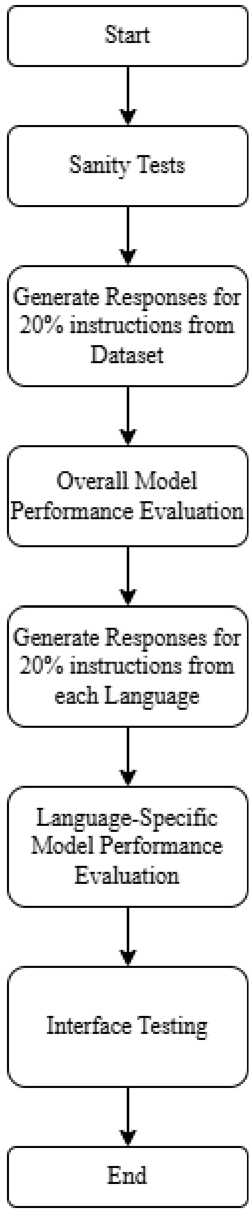

Figure 21. Flow-Chart of Testing Methodology. Created using [13].

4.1.1. Sanity Testing

Many iterations of the mBART model were trained with varying training arguments. Evaluating each of these models comprehensively with all the testing methods is a time-consuming and high-computation task. Thus, a way to check a model and determine if it was worthy of comprehensive testing was required. The solution was to perform some simple tests that test the Sanity of the newly trained model. Find the flow-chart that visualizes this section below:

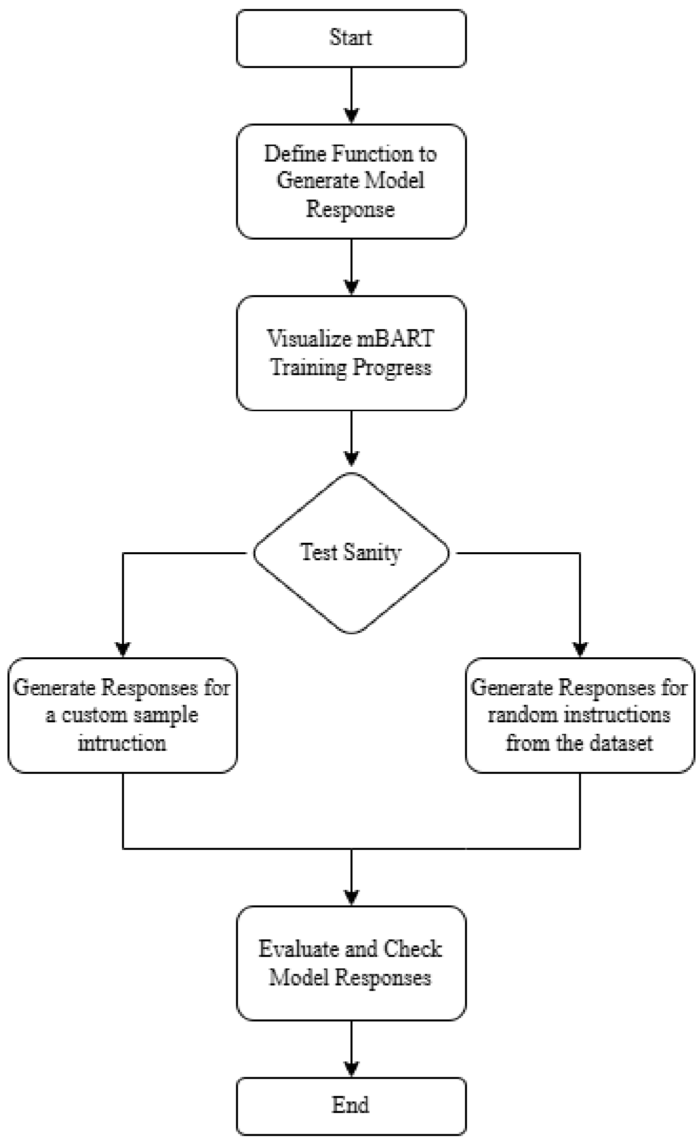

Figure 22. Flow-Chart of Sanity Testing. Created using [13].

4.1.1.1. Implementation

4.1.1.1.1. Response Generation Function

To start generating model responses, we need a function to take input text, related parameters, return the model's response. Luckily, we already have a function that does this for us, the `generate_response` function from 3.5. Chatbot Deployment. It can be used as is with some minor refactoring. Programmatic Implementation:

```
model_dir = "models/final_mbart_model"
model =
MBartForConditionalGeneration.from_pretrained(model_dir).to('
cuda')
tokenizer = MBart50TokenizerFast.from_pretrained(model_dir)
```

Load the mBART model and tokenizer

```
def generate_response(input_text, source_lang, target_lang,
max_length=256, num_beams=5, min_length=125):
    tokenizer.src_lang = source_lang

     model_inputs = tokenizer(input_text,
     return_tensors="pt", padding=True)
     model_inputs = {key: value.to('cuda') for key, value in
     model_inputs.items()}  # Move all inputs to GPU

     generated_tokens = model.generate(
```

```
        input_ids=model_inputs['input_ids'],
        attention_mask=model_inputs['attention_mask'],
        forced_bos_token_id=tokenizer.lang_code_to_id[target_
        lang],
        eos_token_id=tokenizer.eos_token_id,
        min_length=min_length,
        max_length=max_length,
        num_beams=num_beams,
        no_repeat_ngram_size=2,
        early_stopping=True
      )

    return tokenizer.decode(generated_tokens[0],
    skip_special_tokens=True)
```

The exact same function logic as in Chatbot Deployment is used with one key difference. The source and target language are parameters that can be passed in to set the language for the model response. This is crucial as we will be using this function to generate responses for language-specific testing.

4.1.1.1.2. mBART Training Progress Visualization

Visualization of the latest model's training and validation loss is also helpful in determining if the current iteration is a better one than the previous model. The loss values from section 3.4. mBART model Finetuning is used to manually create list data, using which a plot is conceived. The plot can be used to compare with the previous plot with loss values of the last model and determine if the current one is a better one. Programmatic Implementation:

```
data = {
    'epoch': [1, 2, 3, 4, 5, 6, 7],
    'training_loss': [1.280400, 0.738200, 0.654700, 0.590900,
0.563000, 0.539300, 0.529600],
    'validation_loss': [1.306809, 0.787249, 0.748505,
0.693028, 0.676866, 0.670430, 0.670399]
}
```

Create dictionary containing lists of loss values for training and validation of each epoch.

```
loss_df = pd.DataFrame(data)
```

Create a pandas DataFrame with the loss data for plotting

```
plt.figure(figsize=(10, 6))
plt.plot(loss_df['epoch'], loss_df['training_loss'],
label='Training Loss', color='b', marker='o')
plt.plot(loss_df['epoch'], loss_df['validation_loss'],
label='Validation Loss', color='r', marker='o')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss over Epochs')
plt.legend()
```

```
plt.grid(False)
plt.show()
```

Use Matplotlib library to set details such as the figure size, x and y – axis labels, title, legend, and so on. Plot the training and validation losses with their own curves, coded with their own colors – blue and red respectively. Now, the plot is ready for comparison.

4.1.1.1.3. Response Generation for Custom Instruction

Once satisfied with the new model's performance, test its sanity by generating responses in all three languages for a custom instruction. Programmatic Implementation:

```
inputs = {
    "en": "How can I cancel my order?",
    "fr": "Comment puis-je annuler ma commande?",
    "es": "¿Cómo puedo cancelar mi pedido?"
}
```

Define the same custom instruction in all three languages.

```
for lang, input_text in inputs.items():
    response = generate_response(input_text,
    source_lang=f'{lang}_XX', target_lang=f'{lang}_XX',
    max_length=627, min_length=125)
    print(f"Input: {input_text} | Response: {response}")
    print("\n")
```

Iterate through the instructions dictionary, pass in each of the instructions and corresponding language as source and target languages to the previously defined `generate_response` function. This generates responses for our custom instruction in all the three languages formatted as Input: `input_text`, and Response: `response`.

4.1.1.1.4. Response Generation for Random Instructions from Dataset

Once satisfied with the responses for the custom instruction, the next step is generate responses for instructions from the dataset. Before we use the dataset's instructions however, we must ensure they are of the right format, free from any special tokens that were used during mBART training. Once we have a clean dataset, we generate responses for five instructions from the dataset at random to check for sanity. Programmatic Implementation:

```
def clean_special_tokens(text):
```

Function to rid the dataset of special tokens

```
    cleaned_text = text.replace('</s>', '').replace('en_XX',
    '').replace('fr_XX', '').replace('es_XX', '')
    return cleaned_text.strip()
```

The text from a data cell is cleaned in a very simple yet elegant manner. We stack Python's String datatype's `replace()` method, replacing special tokens with a null character in each call.

```
multilingual_df =
pd.read_csv('data/Multilingual_Customer_Support_Training_
Dataset.csv')
```

```
multilingual_df['instruction'] =
multilingual_df['instruction'].apply(clean_special_tokens
)
multilingual_df['response'] =
multilingual_df['response'].apply(clean_special_tokens)

cleaned_dataset_path =
'data/Multilingual_Customer_Support_Training_Dataset_Clea
ned.csv'
multilingual_df.to_csv(cleaned_dataset_path, index=False)
print(f"Cleaned dataset saved to {cleaned_dataset_path}")
```

This function is then applied to the instructions and responses columns in the multilingual customer support dataset using Python library Pandas's `apply()` method. The cleaned dataset is saved as a new `.csv` file.

Since we have a clean dataset, now we can generate responses for random instructions from it.

```
sample_df = multilingual_df.sample(n=5, random_state=42)
```

Choose five samples from the dataset at random with the `random_state` set to 42 to ensure reproducibility of responses. This allows to compare the newer model's responses to the previous model's responses as the responses will be generated for the set of same instructions.

```
for idx, row in sample_df.iterrows():
```

```
lang_code = row['language'][:2]  # Get language prefix
like 'en', 'fr', 'es'
instruction = row['instruction']
response = generate_response(input_text=instruction,
source_lang = f'{lang_code}_XX', target_lang =
f'{lang_code}_XX', max_length=627)
print(f"Input: {instruction} | Response: {response}\n")
```

Iterate through the subset samples, passing in the instruction from the `instruction` column and language code from the `language` column, generate the response using the `generate_response` function, and print the Instruction and Response. Evaluate the responses for sanity.

4.1.2. Overall Model Performance Evaluation

We now have a model that passed the sanity testing. It has proven good enough to be scrutinized further by calculating various metrics from the responses it generates. To ensure the most accurate values for NLP metrics, we need normalized data without line breaks, white spaces, etc. So we normalize our dataset and use it to generate responses for random instructions from 20% of the dataset. We then save and use the model responses, instructions, reference responses from the dataset, and response generation times to calculate a number of conventional metrics. Find the flow-chart that visualizes this section below:

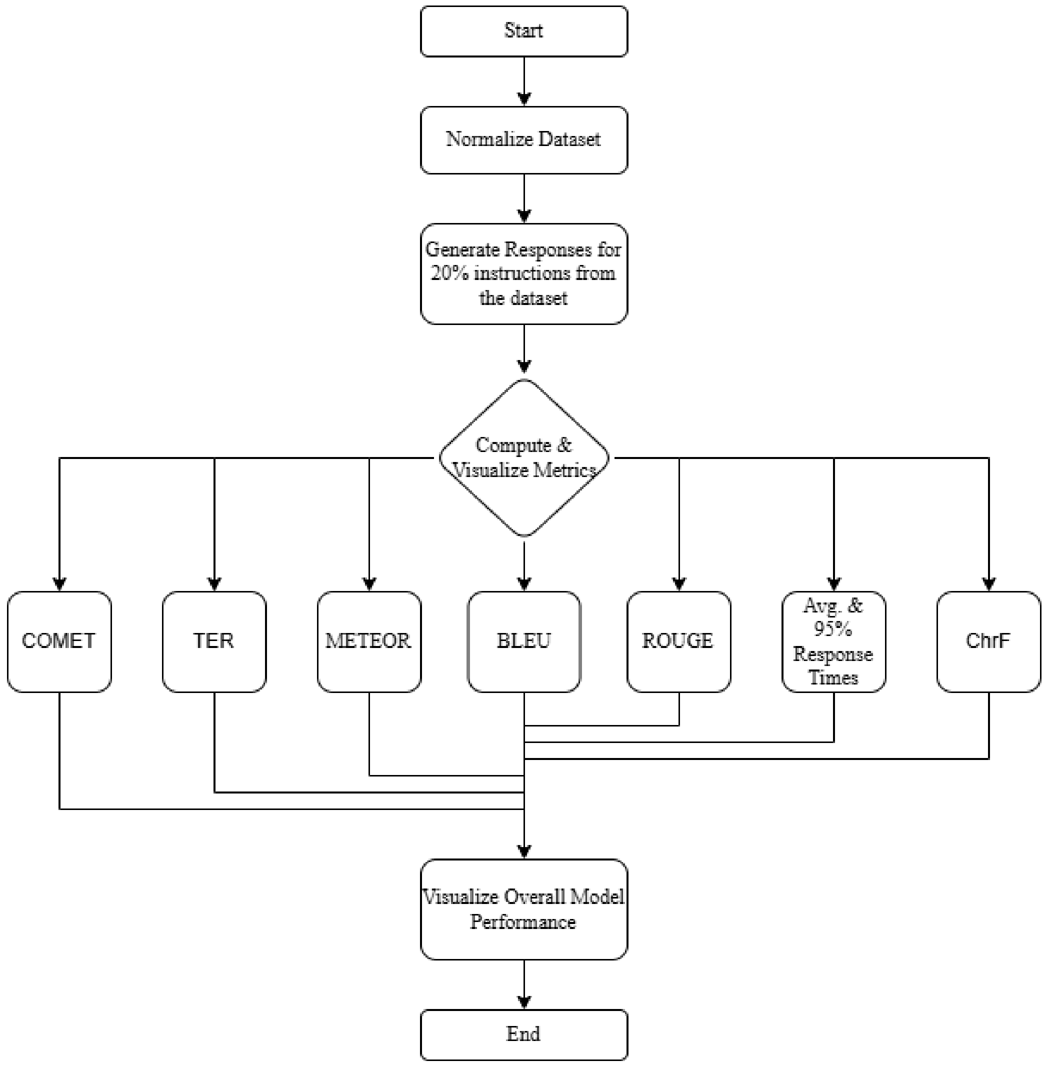

Figure 23. Flow-Chart of Overall Model Performance Evaluation. Created using [13].

4.1.2.1. Implementation

4.1.2.1.1. Dataset Normalization

To begin with the evaluation, we start with normalizing the data:

```
def normalize_text(text):
    return ' '.join(text.replace('\n', ' ').split()).strip()
multilingual_df['normalized_response'] =
multilingual_df['response'].apply(normalize_text)
```

We use String type's `replace()` method to replace any line breaks with spaces and we remove trailing white spaces using `strip()`. Apply this to all responses in the dataset to create a new column with normalized responses which will be used in our metrics calculations.

4.1.2.1.2. Generate Responses for Evaluation Set

For our evaluation, we need a large enough sample size of test set to adequately perform a thorough analysis. For this task, 20% of the multilingual dataset has been chosen for evaluation. We generate responses using the instructions and language codes from each of the rows. The generation times are calculated and stored along with generated responses, instructions, and reference responses from the dataset, all of which will aid in computation of several metrics. Programmatic implementation:

```
subset_size = int(0.2 * len(multilingual_df))
subset_df = multilingual_df.sample(n=subset_size,
random_state=42)
```

A subset dataframe with a size of 20% of the multilingual dataset is created with a random state of 42 using the `sample()` method from pandas.

```
data = []
counter = 1
```

The list to store our evaluation data is initialized. A counter variable is also initialized which will allow to track the progress of the response generation task.

```
for idx, row in subset_df.iterrows():
        lang_code = row['language'][:2]
        instruction = row['instruction']
```

We iterate through each of the samples, also taking its index. The language code are obtained from the `language` column and the instructions are obtained from the `instruction` column both of which are stored in `lang_code` and `instruction` variables respectively.

```
        generation_start = time.time()
        response = generate_response(input_text=instruction,
         source_lang = f'{lang_code}_XX', target_lang =
         f'{lang_code}_XX', max_length=627)
         generation_time = time.time() - generation_start
```

A timer is started, `generate_response()` function is used with the row's instruction and language code as input to generate a response which is stored in the `response` variable. Time taken to generate the response is calculated by subtracting the timer's time from the current time, which is saved in `generation_time` variable.

```
        normalized_generated_response =
        normalize_text(response)
        normalized_reference =
        normalize_text(row['normalized_response'])
        data.append({
            "instruction": instruction,
```

```
            "generated_response":
             normalized_generated_response,
            "reference": normalized_reference,
            "response_time": generation_time
        })
        print(f"Sample: {counter}/{subset_size}\tTime taken
        for Sample {idx}: {generation_time:.2f} seconds")
        counter += 1
```

The response and the reference response are normalized, and all the data used are added to the `data` list. Information to track the progress of response generation is printed with the counter number out of the subset size, including the sample's index from the dataset and the time it took to generate the response.

```
combined_df = pd.DataFrame(data)
combined_df.to_csv(combined_file, index=False)
generated_responses =
combined_df['generated_response'].tolist()
references = combined_df['reference'].tolist()
response_times = combined_df['response_time'].tolist()
instructions = combined_df['instruction'].tolist()
```

We create a dataframe from the list `data` and save it in a csv for future use. We fetch the data from each of the columns in the dataframe and store them in their own respective list variables. These variables would be used in metrics calculations.

4.1.2.1.3. BLEU

The first metric to be computed is the Bilingual Evaluation Understudy score for the generated responses using the SacreBLEU Python library. This metric is one of the most commonly used in the industry to evaluate NLP models and it captures word overlap between a model's generated responses and the reference responses. This metric can be further divided into four types – BLEU – 1, 2, 3, 4, each corresponding to how many n-grams they compute the precision for, where 1-gram corresponds to individual words, 2-grams correspond to pairs of consecutive words, 3-grams correspond to triplets of consecutive words, and 4-grams correspond to sets of four consecutive words. While BLEU-4 is the industry standard, BLEU – 1, 2, 3 are also included in the calculation for a more holistic evaluation. Programmatic implementation:

```
bleu = sacrebleu.corpus_bleu(generated_responses, [[ref] for
ref in references])
precisions = bleu.precisions
```

The generated responses and references are passed to the `corpus_bleu()` method from SacreBLEU library. Since multiple references can be used to test against a single response, `corpus_bleu()` expects a list of references to be passed. In our case, we only have one reference per response, but it is still passed as a list for accordance with SacreBLEU's expectations. The `precisions` argument contains a list of all the n-gram precision scores after BLEU score computation. We fetch and store this list in a variable of the same name.

```
print("\nBLEU Scores:")
```

```
print(f"BLEU-1 Score: {precisions[0]:.2f}")
print(f"BLEU-2 Score: {precisions[1]:.2f}")
print(f"BLEU-3 Score: {precisions[2]:.2f}")
print(f"BLEU-4 Score: {precisions[3]:.2f}")
```

Each of the BLEU scores are printed from the `precisions` list, formatted to 2 decimal places.

After we have the scores, we visualize them in a bar plot for a better interpretation.

```
bleu_scores = [precisions[0], precisions[1], precisions[2],
precisions[3]]
bleu_labels = ['BLEU-1', 'BLEU-2', 'BLEU-3', 'BLEU-4']
plt.figure(figsize=(10, 6))
sns.barplot(x=bleu_labels, y=bleu_scores, palette='viridis')
plt.ylim(0, 100)
plt.ylabel('BLEU Score (%)')
plt.title('BLEU Score Breakdown')
plt.show()
```

The BLEU scores are added to a list `bleu_scores`. Labels are assigned to each score in the `bleu_labels` list. Both are used in a bar plot defined by Seaborn whose properties such as axis-labels, title, size, etc. are set by Matplotlib's Pyplot.

The BLEU score can be used in a variety of ways to interpret the performance of our model. One such way is the compute the score across different sentence lengths and analyze how the model performs for each range of sentences. Programmatic implementation:

```
combined_df['length'] =
combined_df['generated_response'].apply(len)
max_length = combined_df['length'].max()
bins = [0, 250, 500, max(501, max_length + 1)]
labels = ['Short', 'Medium', 'Long']
combined_df['length_category'] =
pd.cut(combined_df['length'], bins=bins, labels=labels,
include_lowest=True)
```

First, the length of each of the responses is calculated and stored in a separate column. We determine the maximum length of the responses and store it in a `max_length` variable. Bins are created at steps of 250 to create three ranges of response lengths – (0, 250), (250, 500), (500, `max_length`) each of which are labeled Short, Medium and Long respectively. The lengths in the `length` column are categorized using the `bins` and appropriate labels are assigned for each row in a new column `length_category`.

```
bleu_scores_by_length = []
categories = []
for category, group in
combined_df.groupby('length_category'):
    if len(group) > 0:
        bleu_score =
        sacrebleu.corpus_bleu(
        group['generated_response'].tolist(), [[ref] for ref
        in group['reference']])
```

```
        bleu_scores_by_length.append(
        bleu_score.precisions[3])
        categories.append(category)
```

Lists to hold the BLEU scores and the categories it was computed for are initialized. We iterate through each of the groups of length categories, computing BLEU scores if the group is not empty. The BLEU-4 scores are added to the `bleu_scores_by_length` list and categories are added to the `categories` list.

```
plt.figure(figsize=(10, 6))
sns.barplot(x=categories, y=bleu_scores_by_length,
palette='autumn')
plt.ylim(0, 100)
plt.ylabel('BLEU-4 Score (%)')
plt.title('BLEU-4 Score by Sentence Length Category')
plt.show()
```

Categories and BLEU scores by length are used to generate a bar plot, made possible by Seaborn. Title, y-label, figure size, and y-limit are setup by Matplotlib's pyplot.

4.1.2.1.4. ROUGE

It stands for Recall-Oriented Understudy for Gisting Evaluation and it computes a score between 0 and 1 based on n-gram overlap between the generated responses and references. It differs from BLEU in that it is recall-oriented and focuses mainly on coverage of words from reference text (ground truth) in generated text while BLEU is precision-oriented and focuses on accuracy of generated text checking how many of the n-grams from generated text exist in the reference. While there are many types of ROUGE scores, for our evaluation, three types were selected – ROUGE – 1, 2, L. ROUGE – 1 checks for 1-gram recall meaning overlap of individual words, ROUGE – 2 checks for bi-gram recall meaning overlap of pairs of consecutive words, and ROUGE – L measures the Longest Common Subsequence (LCS) which is the longest sequence of words that occur in both texts in the same order. While ROUGE-L is the convention, ROUGE – 1, 2 are also included in the calculation for a more holistic evaluation. Programmatic implementation:

```
rouge_scorer_instance = rouge_scorer.RougeScorer(['rouge1',
'rouge2', 'rougeL'], use_stemmer=True)
rouge1_scores, rouge2_scores, rougeL_scores = [], [], []
```

We use the `RougeScorer` method from the `rouge_score` library and initialize an instance for it including the names of metrics that will be computed. `use_stemmer=True` argument allows the words to be reduced to their root forms, for example, walking becomes walk and so on, making the score more accurate. Since ROUGE scores are computed individually for each (response, reference) pair, we initialize seperate lists to store those scores for each type of scores.

```
for gen_resp, ref_resp in zip(generated_responses,
references):
    scores = rouge_scorer_instance.score(ref_resp, gen_resp)
    rouge1_scores.append(scores['rouge1'].fmeasure)
    rouge2_scores.append(scores['rouge2'].fmeasure)
    rougeL_scores.append(scores['rougeL'].fmeasure)
```

We iterate through the list of generated responses and references, compute ROUGE – 1, 2 , L scores for each instance using the `rouge_scorer_instance` and storing them in their own lists.

```
avg_rouge1 = sum(rouge1_scores) / len(rouge1_scores)
avg_rouge2 = sum(rouge2_scores) / len(rouge2_scores)
avg_rougeL = sum(rougeL_scores) / len(rougeL_scores)
```

Average scores for each score is computed by dividing the sum of all the scores by the length of the entire list of scores.

```
print("\nROUGE Scores:")
print(f"ROUGE-1 Score: {avg_rouge1:.2f}")
print(f"ROUGE-2 Score: {avg_rouge2:.2f}")
print(f"ROUGE-L Score: {avg_rougeL:.2f}")
```

Average ROUGE – 1, 2, L scores are printed, formatted to 2 decimal places.

After we have the scores, we visualize them in a bar plot for a better interpretation.

```
rouge_scores = [avg_rouge1, avg_rouge2, avg_rougeL]
rouge_labels = ['ROUGE-1', 'ROUGE-2', 'ROUGE-L']
plt.figure(figsize=(10, 6))
sns.barplot(x=rouge_labels, y=rouge_scores, palette='magma')
plt.ylim(0, 1)
plt.ylabel('ROUGE Score (F1)')
plt.title('ROUGE Score Breakdown')
```

```
plt.show()
```

The ROUGE scores are added to a list `rouge_scores`. Labels are assigned to each score in the `rouge_labels` list. Both are used in a bar plot defined by Seaborn whose properties such as axis-labels, title, figsize, etc. are set by Matplotlib's Pyplot.

4.1.2.1.5. METEOR

METEOR stands for Metric for Evaluation of Translation with Explicit Ordering. This metric is one of the most comprehensive ways to test your NLP model's performance. It checks for not only the harmonic mean of Precision and Recall, but also for Synonymy, Stemming, & Alignment. Synonymy check ensures even if the words in the reference and response are different, if they mean the same, they will contribute to a positive score. Stemming check ensures words are compared at their root forms removing influence of past and present tenses of the words. Alignment check ensures words in the references and responses are also compared in terms of their uni-gram mappings. METEOR was invented as a way to overcome the shortcomings of BLEU and thus, makes for a better way to evaluate an NLP model's performance. Programmatic implementation:

```
meteor_scores = []
for gen_resp, ref_resp in zip(generated_responses,
references):
    tokenized_generated_response = word_tokenize(gen_resp)
    tokenized_reference = word_tokenize(ref_resp)
    meteor_scores.append(meteor_score.meteor_score(
    [tokenized_reference], tokenized_generated_response))
```

A list to store the scores is initialized since METEOR score is calculated for each (reference, response) pair. Generated responses and references are iterated through, and each response and reference get tokenized. `meteor_score()` method from the `meteor_score` library is used to generate the meteor score using the tokenizer response and reference, which after computation get added to the `meteor_scores` list.

```
avg_meteor = sum(meteor_scores) / len(meteor_scores)
print(f"\nMETEOR Score: {avg_meteor:.2f}")
```

The average METEOR score is computed by dividing the sum of all scores by the length of the list and printed.

```
plt.figure(figsize=(8, 6))
plt.bar(['METEOR'], [avg_meteor], color='coral')
plt.ylim(0, 1)
plt.ylabel('METEOR Score')
plt.title('METEOR Score for Generated Responses')
plt.show()
```

We plot the score as a bar plot for better visualization using the `bar()` method from Matplotlib's Pyplot.

4.1.2.1.6. Response Time Evaluation

The speed at which the model generates responses is also a crucial element of its performance. While this is largely dependent on the system that hosts the model, evaluating its response times in a decent system still gives us a good idea of how it can perform on server systems optimized to host chatbot models. Computing this is also relatively simple as we have with us the response times for all the generated responses by the model. Thus, using that data, the average response time and 95th percentile response time of the model are computed. Programmatic implementation:

```
percentile_95_response_time=np.percentile(response_times, 95)
total_time = sum(response_times)
average_time_per_response=total_time/len(generated_responses)
print("\nLatency Metrics:")
print(f"Total time for generating {len(generated_responses)}
responses: {total_time:.2f} seconds")
print(f"Average time per response:
{average_time_per_response:.2f} seconds")
print(f"95th Percentile Response Time:
{percentile_95_response_time:.2f} seconds")
```

Numpy's `percentile()` method is used to compute the 95th percentile from the response times. Average response time is calculated by adding all the response times and dividing the total by the length of the list of responses. Then, both are printed.

The response times can be visualized in an insightful manner using a histogram.

```
plt.figure(figsize=(10, 6))
sns.histplot(response_times, bins=30, kde=True, color='blue')
plt.xlabel('Response Time (seconds)')
plt.ylabel('Frequency')
```

```
plt.title('Distribution of Response Times for Generated
Responses')
plt.show()
```

We use the `histplot()` method from Seaborn to generate the histogram with our list of response times. 30 bins are used to divide our histogram which allows to get a detailed visualization of the response times. Kernel Density Estimate (KDE) is also set to true which produces a curve over the histogram that shows us a smoother representation of how the values are distributed. Remaining properties such as labels, figure sizes, and title are configured using Matplotlib's pyplot.

Another way to visualize our response times which can give us better insights is by means of box plot.

```
plt.figure(figsize=(10, 4))
sns.boxplot(response_times, color='skyblue')
plt.xlabel('Response Time (seconds)')
plt.title('Boxplot of Response Times')
plt.show()
```

We use Seaborn's `boxplot()` to generate our box plot and Matplotlib's pyplot to configure other settings. The box plot will give us the median value, 25th percentile & 75th percentile values, whiskers and outliers of our response times, thus providing a comprehensive outlook at the model's response time performance.

4.1.2.1.7. TER

TER stands for Translation Edit Rate and it can be interpreted as a number that indicates the number of edits needed to transform the generated response into the reference text. TER will perform a number of edit operations to achieve this task, they can be anything from deleting words, insertions, shifting words or substitutions, thus giving us an estimate of the effort required to make the responses and references match, which in-turn shows us the model's capacity to capture the ground truth. Programmatic implementation:

```
ter = sacrebleu.corpus_ter(generated_responses, [[ref] for
ref in references])
avg_ter = ter.score / 100
print(f"\nTER Score: {avg_ter:.2f}")
```

We use the `corpus_ter()` method from SacreBLEU library to compute the TER scores. Similar to BLEU and ROUGE, this metric also expects a list of references so we pass each of our references as a list. The overall TER score returned by the method is stored in the `ter` variable and this is a percentage value and so we divide by 100 to convert to a decimal value so that we have consistency with the range of the other metrics.

```
plt.bar(['TER'],[avg_ter], color='limegreen')
plt.ylim(0, 1)
plt.ylabel('TER Score')
plt.title('Translation Edit Rate for Generated Responses')
plt.show()
```

We plot the score as a bar plot for better visualization using the `bar()` method from Matplotlib's Pyplot.

4.1.2.1.8. ChrF

ChrF can be abbreviated as CHaRacter-level F-score and it is a metric that calculates overlap between a machine generated response and a reference text. It is different from BLEU or ROUGE in that it calculates the similarity based on characters as opposed to words. There are three types of this metric – ChrF, ChrF++, ChrF3, where ChrF calculates the F1 score with equal importance to Precision and Recall of character n-grams overlap, ChrF++ is a type that considers both character-level and word-level F1 scores, giving equal importance to both, and ChrF3 gives more importance to Recall in the F1 score calculation. We compute all three of these metrics for our model's responses for comprehensive evaluation of its accuracy. Programmatic implementation:

```
chrf3 = sacrebleu.corpus_chrf(generated_responses, [[ref] for ref in references], beta=3)
print(f"\nChrF3 Score: {chrf3.score:.2f}")
chrf = sacrebleu.corpus_chrf(generated_responses, [[ref] for ref in references])
print(f"\nChrF Score: {chrf.score:.2f}")
chrf_plus = sacrebleu.corpus_chrf(generated_responses, [[ref] for ref in references], word_order=2)
print(f"\nChrF++ Score: {chrf_plus.score:.2f}")
```

The `corpus_chrf()` method from the SacreBLEU library is used to compute all three types of ChrF scores. For ChrF3, setting `beta=3` emphasizes Recall giving it 3x more importance than Precision. For ChrF, no additional parameters are set, calculating the regular F1 score. For Chrf++, setting `word_order=2` adds word-level n-grams in addition to the default character-level n-grams to the metric calculation. Generated

Responses and the References are passed into each of the method calls, then the metrics are calculated and printed.

```
chrf_scores = [chrf.score, chrf3.score, chrf_plus.score]
chrf_labels = ['ChrF', 'ChrF3', 'ChrF++']
plt.figure(figsize=(10, 6))
sns.barplot(x=chrf_labels, y=chrf_scores, palette='cool')
plt.ylim(0, 100)
plt.ylabel('ChrF Score (%)')
plt.title('Character-Level F-Score Breakdown (ChrF, ChrF3,
ChrF++)')
plt.show()
```

We plot each of the scores in a bar plot for better visualization of the comparison using the `barplot()` method from Seaborn. Limits, labels, title, size, etc. are configured by Matplotlib's pyplot.

4.1.2.1.9. COMET

It stands for Crosslingual Optimized Metric for Evaluation of Translation, and it is a ML-based metric used to evaluate text responses generated by a model. This metric uses a neural network model that was trained to evaluate text responses the same way a human would, mimicking a human evaluation of the responses. The score it returns is meant to closely resemble the score that would be given from a human evaluation. This rather than checking for word or character overlap like the previous metrics, checks for sematic consistency and fluency between responses and references, checking whether they convey the same information, thus providing a deeper understanding of the model's performance. There are two types of COMET scores – Average score and System-level score where the former is found by taking the mean of all the individual scores of (instruction, response, reference) sets, and the latter is the global summary score computed holistically by the model after considering all of the generated responses and their quality. Programmatic implementation:

```
model_path = "models/comet/models--Unbabel--wmt20-comet-qe-
da/snapshots/2e7ffc84fb67d99cf92506611766463bb9230cfb/checkpoint
s/model.ckpt"
comet_model = load_from_checkpoint(model_path)
data = [
     {"src": src, "mt": mt, "ref": ref}
     for src, mt, ref in zip(instructions, generated_responses,
     references)
]
```

Load the previously downloaded COMET model from our saved checkpoint. COMET requires the three inputs, src – source i.e., instructions, mt – machine translation i.e., our model's responses, ref – references i.e., responses from the dataset. The `zip()` function iterates through the instructions, responses, and references combining each of the sets into individual dictionaries which get stored in the data list.

```
comet_output = comet_model.predict(data, batch_size=8, gpus=1)
comet_scores = comet_output.scores
system_comet_score = comet_output.system_score
avg_comet = sum(comet_scores) / len(comet_scores)
```

```
print(f"\nAverage COMET Score (from individual scores):
{avg_comet:.2f}")
print(f"System-level COMET Score: {system_comet_score:.2f}")
```

The `data` is passed as input to the model, a conventional batch size of 8 is set, GPU utilization is enabled, following which the model starts computing the COMET score. Average score is computing by adding all the individual scores and dividing them by the length of the list of the scores. System-level score is fetched from the `comet_output` dictionary returned by the model. Both scores are printed, formatted to two decimal points.

```
comet_labels = ['Average COMET Score', 'System-Level COMET
Score']
comet_scores = [avg_comet, system_comet_score]
plt.figure(figsize=(10, 6))
sns.barplot(x=comet_labels, y=comet_scores, palette='spring')
plt.ylim(0, 1)
plt.ylabel('COMET Score')
plt.title('COMET Scores for Generated Responses')
plt.show()
```

We add the scores and the appropriate labels to their newly created lists. We plot the scores along with their labels using the `barplot()` method given by Seaborn and set other properties using Matplotlib's pyplot.

4.1.2.1.10. Overall Performance Metrics Summary

A number of industry relevant metrics have been computed on our model's responses with respect to instructions, references, and response times, thus concluding our comprehensive evaluation of the fine-tuned mBART model's performance. A visualization that provides a holistic view of our most important metrics to summarize the evaluation results can be informative. Thus, we use a Radar chart to summarize our results. Programmatic implementation:

```
metrics = ['BLEU-4', 'ROUGE-L', 'METEOR', 'ChrF3', 'TER']
max_values = [100, 1, 1, 100, 1]
values = [
    precisions[3] / max_values[0],
    avg_rougeL / max_values[1],
    avg_meteor / max_values[2],
    chrf3.score / max_values[3],
    avg_ter / max_values[4]
]
values += values[:1]
```

Labels are assigned for each of the metrics to be included in the Radar chart. Before we generate the chart, it should be ensured that they all fall within the same range. So, we define max values for each metric and normalize them using their respective max values. The first value is added again to the end to close the radar chart loop during generation.

```
fig, ax = plt.subplots(figsize=(8, 8),
subplot_kw=dict(polar=True))
```

```
angles = [n / float(len(metrics)) * 2 * pi for n in
range(len(metrics))]
angles += angles[:1]
```

We use the `subplots()` method from Matplotlib's pyplot to generate the radar chart. An axes object is created with `subplot_kw=dict(polar=True)` which makes it a polar circular plot. We calculate the angles for each axis using each of the metrics and then add the first angle at the end again to close the loop.

```
ax.plot(angles, values, linewidth=2, linestyle='solid')
ax.fill(angles, values, alpha=0.4)
ax.set_xticks(angles[:-1])
ax.set_xticklabels(metrics)
plt.title('Overall Performance Metrics Summary', pad=30)
plt.show()
```

The chart is plotted with the angles and values that were defined, including various properties set such as linewidth, linestyle, color intensity, labels, position of labels, and so on.

4.1.3. Language-Specific Model Performance Evaluation

The model's overall performance has been evaluated thoroughly. However, it is also important to evaluate the model's performance for each language it was trained to generate responses in. This additional evaluation should be carried out to ensure that our model is not biased towards or against any of the languages. So we generate responses for random instructions comprising 20% from each language in the dataset. We then save and use each language's responses, instructions, reference responses, and response generation times to calculate the same metrics as before and compare the results across the languages. Find the flow-chart that visualizes this section below:

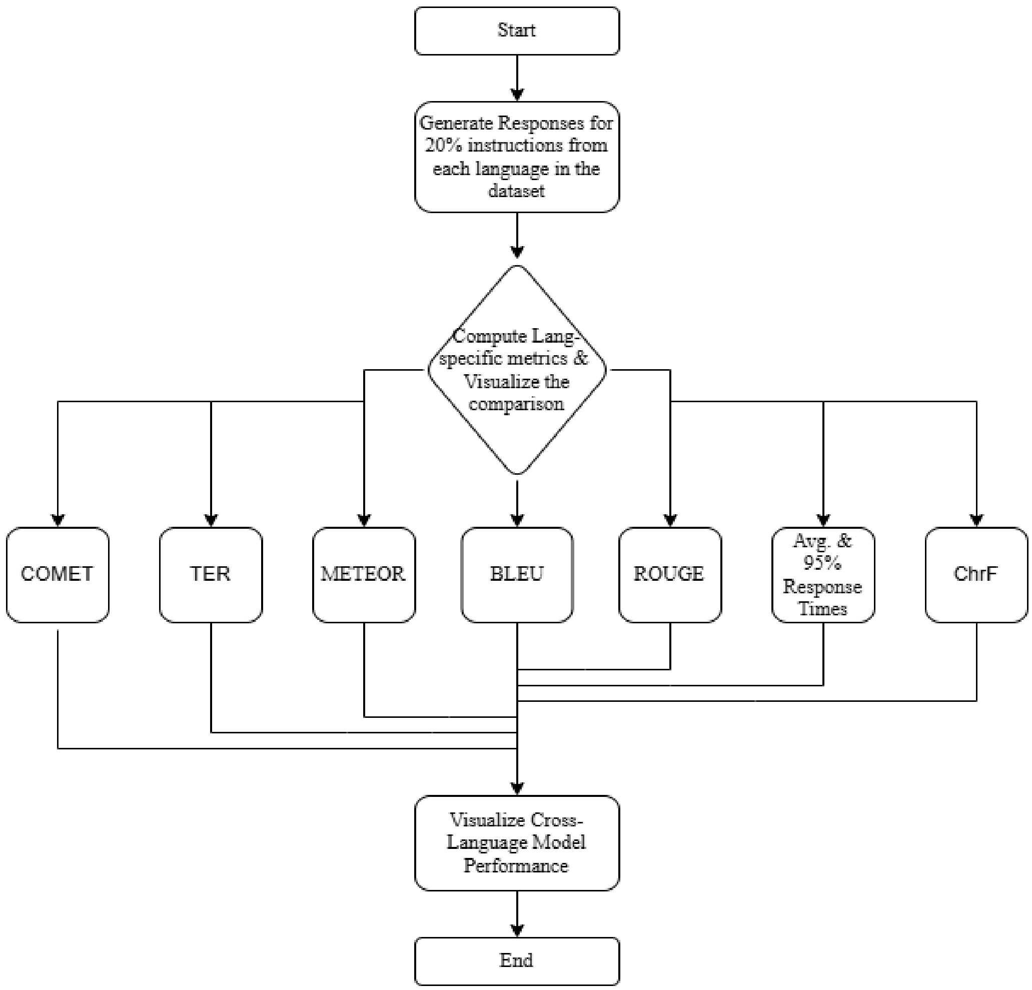

Figure 24. Flow-Chart of Language-Specific Model Performance Evaluation. Created using [13].

4.1.3.1. Implementation

4.1.3.1.1. Generate Responses for Each Language's Evaluation Set

For our evaluation, we need a large enough sample size of test set to adequately perform a thorough analysis. For this task, 20% of the multilingual dataset has been chosen for evaluation. We generate responses using the instructions and language codes from each of the rows. The generation times are calculated and stored along with generated responses, instructions, and reference responses from the dataset, all of which will aid in computation of several metrics. Programmatic implementation:

```
languages = ['en', 'fr', 'es']
language_codes = {
    'en': 'en_XX',
    'fr': 'fr_XX',
    'es': 'es_XX'
}
data_save_path = "data/language_specific/"
```

We define the language codes for each language to help retrieve the test data and in generation of responses. We also specify the common path where all languages test data should be saved.

```
for lang in languages:
     csv_file_path=f"{data_save_path}generated_responses_{lang}.
     csv"
     print(f"\n--- Generating Responses and Latencies for
     Language: {lang} ---\n")
     language_sample_df=multilingual_df[multilingual_df['languag
     e'] == lang].sample(n=5374, random_state=42)
```

We iterate through the list of languages, to generate responses for each language's instructions. We print the current language at the start, and then an eval set with 20% of that language's instructions (5374) is created with a seed value of 42 to ensure reproducibility.

```
latencies = []
generated_responses = []
references = []
counter = 1
subset_size = len(language_sample_df)
for idx, row in language_sample_df.iterrows():
    instruction = row['instruction']
    start_time = time.time()
    response = generate_response(input_text=instruction,
    source_lang=language_codes[lang],
    target_lang=language_codes[lang], max_length=627)
    latency = time.time() - start_time
    latencies.append(latency)
    generated_responses.append(normalize_text(response))
    references.append(normalize_text(row['response']))
    print(f"Sample: {counter}/{subset_size}\tTime taken for
    Sample {idx}: {latency:.2f} seconds")
    counter += 1
    data = {
       "instruction": language_sample_df['instruction']
       .tolist(),
       "generated_response": generated_responses,
       "reference": references,
```

```
            "latency": latencies
        }
        language_data_df = pd.DataFrame(data)
        language_data_df.to_csv(csv_file_path, index=False)
        print(f"Data for language {lang} saved to
        {csv_file_path}")
```

The remaining logic is exactly the same as in section 4.1.2.1.2 but instead of using random instructions from the entire dataset, we use random instructions from one language at a time, repeating for all three languages.

4.1.3.1.2. Load Language-Specific Data

All the data needed to begin Language-Specific evaluation using our list of metrics is ready. We load each language's evaluation data in their own set of variables all of which are named to reflect the language and the data they contain. Programmatic implementation:

```
en_data_df =
pd.read_csv("data/language_specific/generated_responses_en.csv")
generated_responses_en =
en_data_df['generated_response'].tolist()
references_en = en_data_df['reference'].tolist()
latencies_en = en_data_df['latency'].tolist()
instructions_en = en_data_df['instruction'].tolist()
```

We load instructions, responses, references, and latencies for English from its saved csv file for evaluation.

```
fr_data_df =
pd.read_csv("data/language_specific/generated_responses_fr.csv")
generated_responses_fr =
fr_data_df['generated_response'].tolist()
references_fr = fr_data_df['reference'].tolist()
latencies_fr = fr_data_df['latency'].tolist()
instructions_fr = fr_data_df['instruction'].tolist()
```

We load instructions, responses, references, and latencies for French from its saved csv file for evaluation.

```
es_data_df =
pd.read_csv("data/language_specific/generated_responses_es.csv")
generated_responses_es =
es_data_df['generated_response'].tolist()
references_es = es_data_df['reference'].tolist()
latencies_es = es_data_df['latency'].tolist()
instructions_es = es_data_df['instruction'].tolist()
```

We load instructions, responses, references, and latencies for Spanish from its saved csv file for evaluation.

Since the same metrics will be calculated again, the metric's background/definition, the code and their explanations will not be repeated for all the metrics as the logic involved is exactly the same with just the variable names changed to use each language's data instead of the overall data. Please refer to each metric's analogous section in the previous sub-heading which has the same section number, for the aforementioned details. For example, for the details of section 4.1.3.1.3. BLEU, refer to section 4.1.2.1.3 and so on. The following sections will focus on the plots visualizing the comparison of the metrics across the languages and their logic.

4.1.3.1.3. BLEU

BLEU – 1, 2, 3, 4 scores are computed for model responses in all three languages and the comparison is visualized using a line chart. For the code and explanation of the metric, please refer to Section 4.1.2.1.3 as the code used is exactly the same but with language-specific data used instead. Programmatic Implementation of the plot:

```
bleu_scores = {
    'BLEU-1': [precisions_en[0], precisions_fr[0],
precisions_es[0]],
    'BLEU-2': [precisions_en[1], precisions_fr[1],
precisions_es[1]],
    'BLEU-3': [precisions_en[2], precisions_fr[2],
precisions_es[2]],
    'BLEU-4': [precisions_en[3], precisions_fr[3],
precisions_es[3]]
}
```

Computed BLEU – 1, 2, 3, 4 scores of each language are added to their respective lists, which are stored in a dictionary with their respective keys.

```
plt.figure(figsize=(14, 8))
```

```
for bleu_level, scores in bleu_scores.items():
    plt.plot(languages, scores, marker='o', linestyle='-',
label=bleu_level)
plt.ylim(0, 120)
plt.ylabel('BLEU Score (%)')
plt.title('BLEU Scores Across Languages')
plt.legend(title='BLEU Level')
plt.grid(False)
plt.show()
```

We iterate through the dictionary and for each score type, get the items (scores) of each of the languages, use language codes from the previously defined languages list and plot the scores as separate lines. Four lines with markers for each language in the x-axis indicating the score in y-axis are plotted for each BLEU level, creating a line-chart with all the score values. Title, labels, legend, and other properties are added to the plot using Matplotlib's Pyplot.

4.1.3.1.4. ROUGE

ROUGE – 1, 2, L scores are computed for model responses in all three languages and the comparison is visualized using a line chart. For the code and explanation of the metric, please refer to Section 4.1.2.1.4 as the code used is exactly the same but with language-specific data used instead. Programmatic Implementation of the plot:

```
rouge_scores = {
    'ROUGE-1': [avg_rouge1_en, avg_rouge1_fr, avg_rouge1_es],
    'ROUGE-2': [avg_rouge2_en, avg_rouge2_fr, avg_rouge2_es],
```

```
    'ROUGE-L': [avg_rougeL_en, avg_rougeL_fr, avg_rougeL_es]
}
```

Computed ROUGE – 1, 2, L scores of each language are added to their respective lists, which are stored in a dictionary with their respective keys.

```
plt.figure(figsize=(14, 8))
for rouge_level, scores in rouge_scores.items():
    plt.plot(languages, scores, marker='o', linestyle='-',
label=rouge_level)
plt.ylim(0, 1)
plt.ylabel('ROUGE Score (F1)')
plt.title('ROUGE Scores Across Languages')
plt.legend(title='ROUGE Level')
plt.grid(False)
plt.show()
```

We iterate through the dictionary and for each score type, get the items (scores) of each of the languages, use language codes from the previously defined languages list and plot the scores as separate lines. Three lines with markers for each language in the x-axis indicating the score in y-axis are plotted for each ROUGE level, creating a line-chart with all the score values. Title, labels, legend, and other properties are added to the plot using Matplotlib's Pyplot.

4.1.3.1.5. METEOR

METEOR scores are computed for model responses in all three languages and the comparison is visualized using a Bar Plot. For the code and explanation of the metric, please refer to Section 4.1.2.1.5 as the code used is exactly the same but with language-specific data used instead. Programmatic Implementation of the plot:

```
meteor_scores = [avg_meteor en, avg_meteor_fr, avg_meteor_es]
plt.figure(figsize=(10, 6))
sns.barplot(x=languages, y=meteor_scores, palette='Oranges')
plt.ylim(0, 1)
plt.ylabel('METEOR Score')
plt.title('METEOR Scores Across Languages')
plt.show()
```

Each of the scores are added to a list `meteor_scores`. The scores list and language codes from the previously defined `languages` list are used in the `barplot()` method from Seaborn to generate the plot. Remaining properties of the plot are configured with Matplotlib's Pyplot.

4.1.3.1.6. Response Time Evaluation

Average Response times and 95th Percentile Response times are computed for model responses in all three languages and the comparisons are visualized using Bar Plots. For the code and explanation of the metric, please refer to Section 4.1.2.1.6 as the code used is exactly the same but with language-specific data used instead. Programmatic Implementation of the plots:

```
languages = ['English', 'French', 'Spanish']
average_latencies = [average_latency_en, average_latency_fr,
average_latency_es]
```

```
percentile_95_latencies = [percentile_95_latency_en,
percentile_95_latency_fr, percentile_95_latency_es]
```

Labels for the languages are added to the `languages` list, average latencies and 95th percentile latencies of each of the languages are added to their own lists `average_latencies` and `percentile_95_latencies`.

```
plt.figure(figsize=(12, 6))
sns.barplot(x=languages, y=average_latencies,
palette='Blues')
plt.ylabel('Average Response Time (seconds)')
plt.title('Average Response Time Across Languages')
plt.show()
```

The Average response times of each of the languages are plotted with the `languages` in the x-axis and the y-axis being the `average_latencies`. Seaborn's `barplot()` is used to generate the plot with those inputs and Matplotlib's Pyplot is used to set title, labels, size and show the plot.

```
plt.figure(figsize=(12, 6))
sns.barplot(x=languages, y=percentile_95_latencies,
palette='Greens')
plt.ylabel('95th Percentile Response Time (seconds)')
plt.title('95th Percentile Response Time Across Languages')
plt.show()
```

The 95th Percentile response times of each of the languages are plotted with the `languages` in the x-axis and the y-axis being the `percentile_95_latencies`. Seaborn's `barplot()` is used to generate the plot with those inputs and Matplotlib's Pyplot is used to set title, labels, size and show the plot.

4.1.3.1.7. TER

TER scores are computed for model responses in all three languages and the comparison is visualized using a Bar Plot. For the code and explanation of the metric, please refer to Section 4.1.2.1.7 as the code used is exactly the same but with language-specific data used instead. Programmatic Implementation of the plot:

```
ter_scores = [avg_ter_en, avg_ter_fr, avg_ter_es]
plt.figure(figsize=(10, 6))
sns.barplot(x=languages, y=ter_scores, palette='Purples')
plt.ylim(0, 1)
plt.ylabel('TER Score')
plt.title('TER Scores Across Languages')
plt.show()
```

Each of the scores are added to a list `ter_scores`. The scores list and language codes from the previously defined `languages` list are used in the `barplot()` method from Seaborn to generate the plot. Remaining properties of the plot are configured with Matplotlib's Pyplot.

4.1.3.1.8. ChrF

ChrF, ChrF++, and ChrF3 scores are computed for model responses in all three languages and the comparison is visualized using a Heatmap. For the code and explanation of the metric, please refer to Section 4.1.2.1.8 as the code used is exactly the same but with language-specific data used instead. Programmatic Implementation of the plot:

```
languages = ['English', 'French', 'Spanish']
metrics = ['ChrF', 'ChrF3', 'ChrF++']
chrf_scores = [
    [chrf_en.score, chrf3_en.score, chrf_plus_en.score],  # 
English
    [chrf_fr.score, chrf3_fr.score, chrf_plus_fr.score],  # 
French
    [chrf_es.score, chrf3_es.score, chrf_plus_es.score]   # 
Spanish
]
chrf_scores = np.array(chrf_scores)
```

Add labels for the Heatmap in the `languages` and `metrics` lists. Compile the ChrF scores in three lists differentiated by language and add those to the `chrf_scores` list. Convert the list to a numpy array for faster generation of the plot.

```
plt.figure(figsize=(10, 6))
ax = sns.heatmap(chrf_scores, annot=True, cmap="YlGnBu", 
xticklabels=metrics, yticklabels=languages, fmt=".2f")
plt.title('ChrF Scores for Different Languages and Metrics')
plt.xlabel('Metric')
```

```
plt.ylabel('Language')
plt.show()
```

`heatmap()` function from Seaborn generates the heatmap using the numpy array of scores. `annot=True` adds the respective scores to each cell in the plot, `cmap="YlGnBu"` sets the color-map to Yellow-Green-Blue color scheme where cells with lower values are colored in yellows (lighter color) and cells with higher values are colored in Blues (darker color), `xticklabels=metrics` makes the scores become the x-axis labels, `yticklabels=languages` makes the languages become the y-axis labels, `fmt=".2f"` formats the scores to two decimal places. Remaining properties of the plot are configured using Matplotlib's Pyplot.

4.1.3.1.9. COMET

COMET – Average & System-level scores are computed for model responses in all three languages. For the code and explanation of the metric, please refer to Section 4.1.2.1.9 as the code used is exactly the same but with language-specific data used instead. Since the Average and System-level scores produced were the same for all three of the languages, a plot was unnecessary as it would have yielded any insights.

4.1.3.1.10. Cross-Language Performance Metrics Summary

All the same metrics have been computed on our model's responses with respect to instructions, references, and response times in all three languages - English, French, & Spanish, thus concluding the comprehensive evaluation of the fine-tuned mBART model's language-specific performance. A visualization that provides a holistic view of the important metrics across all three languages can show us how the model performs for each. Thus, we use a Radar chart to summarize our results. Please refer to section 4.1.2.1.10 for the code and explanation as the plot uses the same code and follows the same structure except here it uses language-specific data and each language's chart is color-coded.

4.1.4. Interface Testing

The chatbot model mBART's performance has been tested rigorously. The final test is to ensure that it operates as intended in the Chatbot Web-Interface that has been designed. This section will test the interface and all its functionalities, including the model's interactions with the user. Find the Flow-Chart that visualizes this section below:

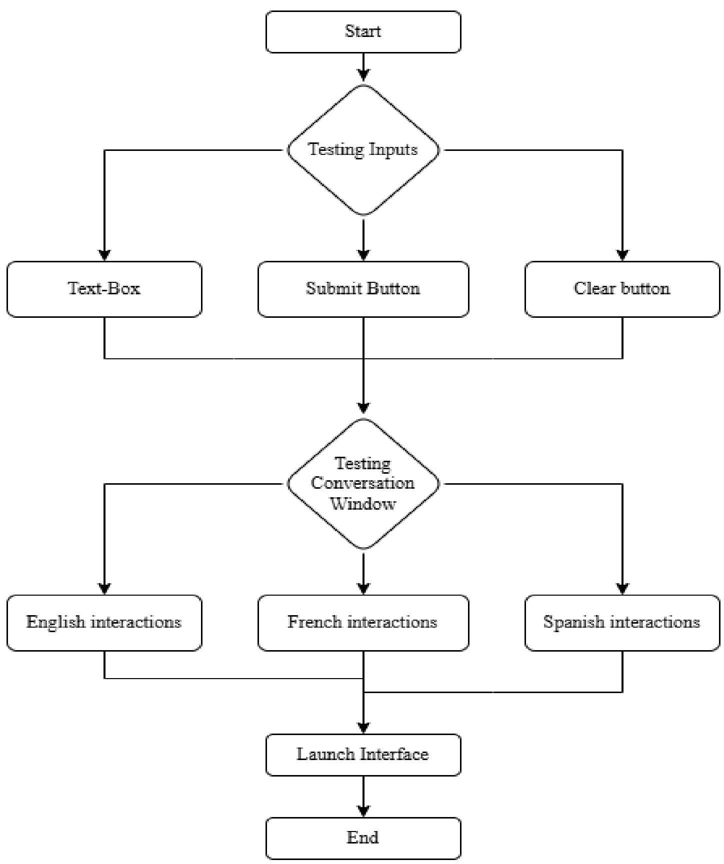

Figure 25. Flow-Chart of Interface Testing. Created using [13].

4.1.4.1. Testing Inputs

The Web interface contains three inputs:

i. Text-Box – Takes user input
ii. Submit Button – Submits user input to the model for a response
iii. Clear Chat Button – Resets the Conversation

All three of these inputs will be tested to ensure they function correctly.

4.1.4.2. Testing Conversation Window

The Conversation Window in the Web Interface has to display the interactions between the user and the model, i.e., the user input and the model's response formatted appropriately, without visual bugs in all three languages:

i. English
ii. French
iii. Spanish

Thus, the conversation window will be tested visually for all three languages.

4.2. Results

This section contains the raw results of each of the tests that were described in 4.1. Methodology. For the interpretation of these results, refer to 4.3. Analysis which contains the explanations under the same section numbers and headings.

Results can be divided into four parts:

1. Sanity Testing Results
2. Overall Model Performance Evaluation Results
3. Language-Specific Model Performance Evaluation Results
4. Interface Testing Results

Find the flow-chart that visualizes this section below:

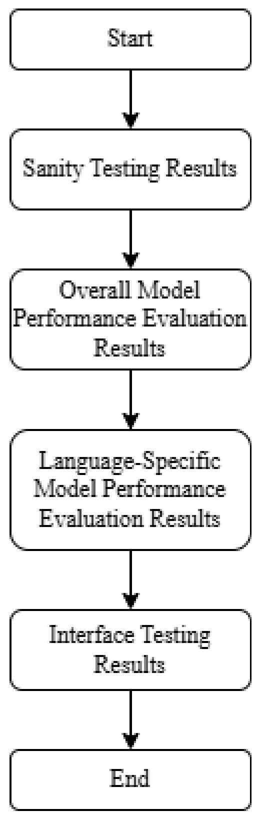

Figure 26. Flow-Chart of Testing Results. Created using [13].

4.2.1. Sanity Testing Results

This section contains all the results from 4.1.1. Sanity Testing.

4.2.1.1. Implementation Results

4.2.1.1.1. Response Generation Function

This section simply deals with Function Definition so it does not yield any results

4.2.1.1.2. mBART Training Progress Visualization

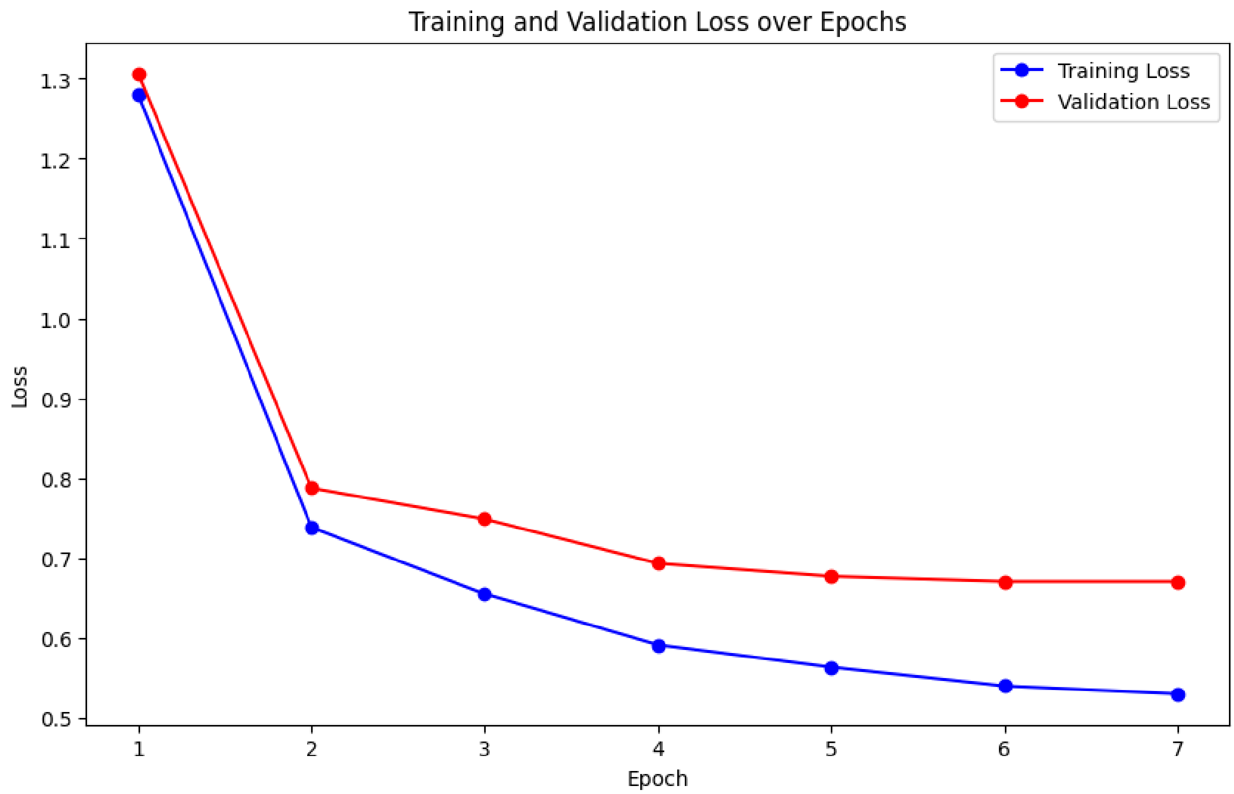

Figure 27. Plot of Training Progress. Created using Python Libraries.

4.2.1.1.3. Response Generation for Custom Instruction

These are the responses generated by the model for the custom instruction:

```
Input: How can I cancel my order? | Response: I've picked
up that you're looking for guidance on canceling your
order. We apologize for any inconvenience this may have
caused. To assist you with the cancellation process, please
follow these steps: 1. Access Your Account: Log in to your
company portal using your credentials. 2. Locate Your
```

Orders: Once logged in, navigate to the 'order history' section. 3. Find the Relevant Order: Look for the specific order you wish to cancel and click on it. 4. Initiate Cancellation: Within the order details, you should find an option labeled 'cancel your purchase here' or something similar. Please select this option. 5. Confirm and Complete: The system may prompt you for confirmation or ask for feedback. Kindly provide the necessary information to proceed. If you encounter any difficulties or have further questions, our dedicated support team is available during 9 AM - 5 PM. You can reach us at 123-456-7890 or through the Live Chat feature on our website at www.example.com. Your satisfaction is our top priority, and we appreciate your understanding throughout this process.

Input: Comment puis-je annuler ma commande? | Response: Je comprends que vous cherchez de l'aide pour annuler votre commande, et je suis ici pour vous guider à travers le processus. 1. Connectez-vous à votre compte: Commencez par vous connecter au portail de votre entreprise en utilisant vos identifiants. 2. Localisez vos commandes : Une fois connecté, accédez à la section « Historique de la commande ». 3. Identifier votre achat: Recherchez le numéro de commande ORD12345 et cliquez dessus pour accéder aux détails. 4. Lancez la demande d'annulation: Vous devriez trouver une option intitulée 'historique des ordres' ou similaire. Veuillez sélectionner cette option. 5. Suivez toutes les étapes supplémentaires: Le système peut vous inciter à confirmer ou à fournir des commentaires. Si vous rencontrez des difficultés ou avez des questions en cours de route, notre équipe dédiée est disponible de 9h à 17h. Vous pouvez nous joindre au 123-456-7890 ou via la fonctionnalité Live Chat sur notre site Web à www.exemple.com. Nous nous engageons à assurer votre satisfaction et à vous aider à chaque étape du chemin.

Input: ¿Cómo puedo cancelar mi pedido? | Response: Entiendo que usted está buscando orientación sobre cómo cancelar su pedido, y estoy aquí para ayudarle. Cancelar un pedido es un proceso sencillo que puede seguir siguiendo estos pasos: 1. Inicie sesión en su cuenta: Visite nuestro portal de la empresa e inicie sesión utilizando sus credenciales. 2. Navegue a sus pedidos: Una vez que haya iniciado sesión, localice la sección 'Historial del pedido'. 3. Encuentre la orden pertinente: Busque el orden asociado con el número de pedido ORD12345. Haga clic en él para ver los detalles. 4. Iniciar la cancelación: Usted debe ver una opción etiquetada como "historial de orden" o algo similar. Por favor, seleccione esta opción. 5. Siga cualquier paso adicional: El sistema puede pedirle que proporcione alguna información adicional o responda a algunas preguntas para completar el proceso. Sírvase seguir las instrucciones proporcionadas. Si usted encuentra cualquier dificultad o tiene más preguntas, nuestro equipo de soporte dedicado está disponible durante 9 AM - 5 PM. Puede comunicarse con nosotros en 123-456-7890 o utilizar la función de Chat en vivo en nuestro www.ejemplo.com. Estamos comprometidos a garantizar su satisfacción y resolver cualquier preocupación que pueda tener.

4.2.1.1.4. Response Generation for Random Instructions from Dataset

These are the responses generated by the model for five random instructions from the dataset fetched with a random state of 42:

Input: No sé cómo puedo comprar un producto | Response: Gracias por contactar! Entiendo que no está seguro de cómo comprar un producto. No se preocupe, estoy aquí para ayudarle en cada paso del camino. Para empezar, ¿podría por favor dejarme saber qué producto específico que está interesado en comprar? Una vez que tenga esa información, puedo guiarlo a través del proceso de compra y asegurarse de que tiene una experiencia suave y satisfactoria. Vamos a hacer este viaje de compras lo más fácil posible para usted! ¿Cómo si no puedo ayudarte hoy en día en la compra de los productos que te interesan? ¡Trabajaremos juntos para encontrar el producto perfecto que se adapte a tus necesidades!

Input: Aidez-moi à vérifier dans quels cas puis-je demander à être remboursé | Response: Je comprends parfaitement votre besoin d'avoir une compréhension claire des situations dans lesquelles vous pouvez demander un remboursement. Laissez-moi vous fournir une ventilation complète: 1. **Défaut de produit/service :** Si le produit ou le service que vous avez acheté a un défaut ou ne répond pas à la description fournie, vous êtes généralement admissible à un remplacement. 2. **Annulation dans la période de grâce:** Beaucoup de vendeurs offrent une fenêtre de gratification dans le temps qui vous permet de annuler votre achat. Assurez-vous de vérifier les conditions spécifiques pour l'annulation. 3. **Coûts non autorisés ou transactions frauduleuses**: Si vous remarquez des frais inconnus sur votre compte, il est important de les déclarer immédiatement. Ils seront en mesure de vous aider à enquêter et à résoudre le problème. 4. **Non-livraison ou livraison tardive** : Si votre commande n'a pas été livrée dans les délais prévus, ou s'il y a eu des retards importants au cours du processus de livraison, soyez assuré qu'un remboursement peut être accordé. 5. ** Erreurs de facturation ou de surfacturation/chargement/parfait/fausse publicité** Dans les cas où des erreurs se produisent, telles que des numéros de facture ou des charges incorrectes, le remboursement pourrait être émis. Veuillez noter que ce sont des lignes directrices générales et que les politiques de remboursement peuvent varier selon le vendeur et le fournisseur de services. C'est toujours une bonne idée de revoir la politique de remboursement spécifique du produit, du service ou du fournisseur avec lequel vous envisagez de communiquer pour obtenir des renseignements plus détaillés. J'espère que cette information clarifie les situations où vous pourriez être eligible pour un rabais. Cependant, je vous prie de bien vouloir me fournir votre numéro de commande ou toute autre information pertinente.

Input: Je dois voir quand mon achat va arriver, aide-moi. | Response: Nous comprenons votre empressement à connaître l'heure d'arrivée estimée de votre achat. Pour vous fournir les informations les plus précises, pourriez-vous nous fournir le TRK123456789 ou ORD12345? Avec ces informations, nous serons en mesure de vous donner une mise à jour précise sur la date de livraison prévue. Votre patience est grandement appréciée car nous travaillons ensemble pour vous assurer une expérience fluide et satisfaisante. Comment votre expérience globale a-t-elle été avec notre service jusqu'à présent? Faites-nous savoir!

Input: L'addition de chck de John Doe | Response: Je comprends votre besoin de vérifier la facture de Dear Customer Doe. Travaillons ensemble pour trouver l'information que vous recherchez. Pourriez-vous me fournir des détails supplémentaires tels que la plage de dates ou tout numéro de facture spécifique? Cela me permettra de vous aider plus efficacement. Soyez assuré, je suis ici pour vous guider à chaque étape du chemin et vous assurer d'avoir une expérience satisfaisante avec notre système de

```
recouvrement des comptes. Faites-moi savoir s'il y a autre
chose que je peux faire pour améliorer votre expérience.

Input: problem setting a new delivery address up |
Response: I'm sorry to hear that you're experiencing
difficulties setting up a new delivery address. Let's work
together to resolve this issue. Here are a few steps you
can try: 1. Log in to your account on our website. 2.
Navigate to the "My Account" or "Profile" section. 3. Look
for the option to manage your shipping addresses. 4. Click
on "Add New Address" to proceed. 5. Fill in the required
fields with your new address details, such as street name,
city, state, and zip code. 6. Double-check the accuracy of
the information you entered. 7. Save the changes. If you
encounter any specific issues or error messages during this
process, please provide more details so that I can assist
you further. Your satisfaction is our top priority.
```

4.2.2. Overall Model Performance Evaluation Results

This section contains all the results from 4.1.2. Overall Model Performance Evaluation.

4.2.2.1. Implementation Results

4.2.2.1.1. Dataset Normalization

This section deals with applying a Function to the Multilingual Dataset, so it does not yield any results.

4.2.2.1.2. Generate Responses for Evaluation Set

The output shows a summary of the progress of the response generation by the model for 20% of instructions from the dataset, the index of all the samples used, including the time taken in seconds to generate a response for each sample.

```
Sample: 1/16123    Time taken for Sample 73049: 1.03 seconds
Sample: 2/16123    Time taken for Sample 33252: 3.65 seconds
Sample: 3/16123    Time taken for Sample 39784: 0.96 seconds
```

.

.

.

```
Sample: 16122/16123Time taken for Sample 15469: 0.94 seconds
Sample: 16123/16123Time taken for Sample 62852: 0.92 seconds
Generated responses, references, and response times saved to
data/generated_responses_combined.csv
```

4.2.2.1.3. BLEU

Below are BLEU – 1, 2, 3, 4 scores computed from the Model's Responses and References:

```
BLEU Scores:
BLEU-1 Score: 100.00
BLEU-2 Score: 96.58
BLEU-3 Score: 86.21
BLEU-4 Score: 72.17
```

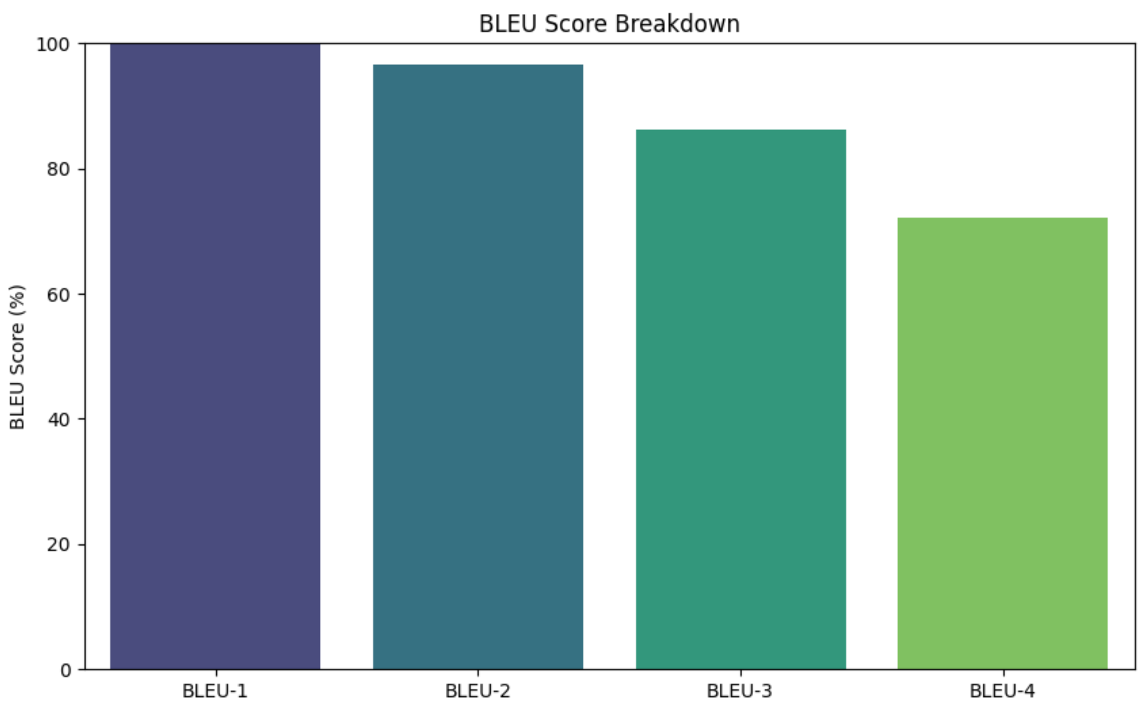

Figure 28. BLEU Scores Comparison. Created using Python Libraries.

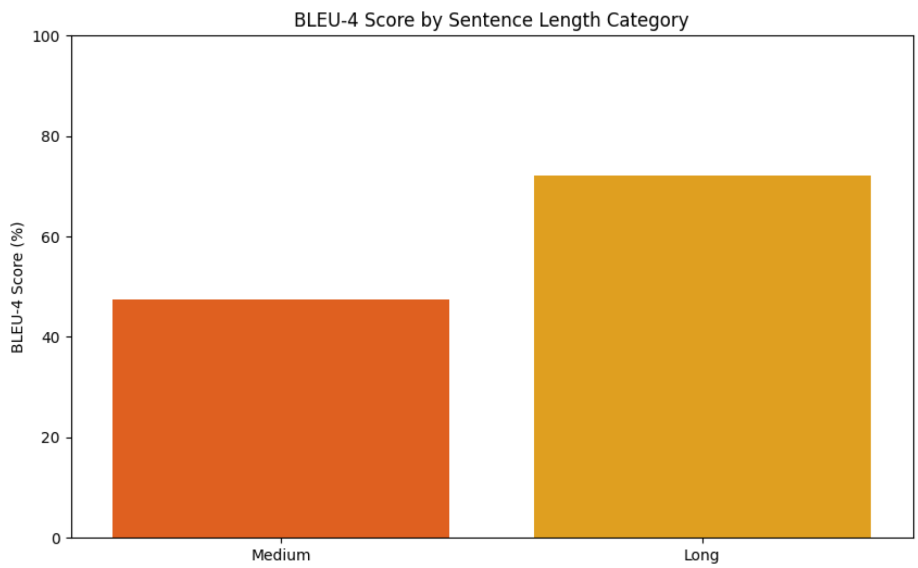

Figure 29. BLEU-4 Score by Sentence Length Category. Created using Python Libraries.

4.2.2.1.4. ROUGE

Below are ROUGE – 1, 2, 3, 4 scores computed from the Model's Responses and References:

```
ROUGE Scores:
ROUGE-1 Score: 0.55
ROUGE-2 Score: 0.30
ROUGE-L Score: 0.38
```

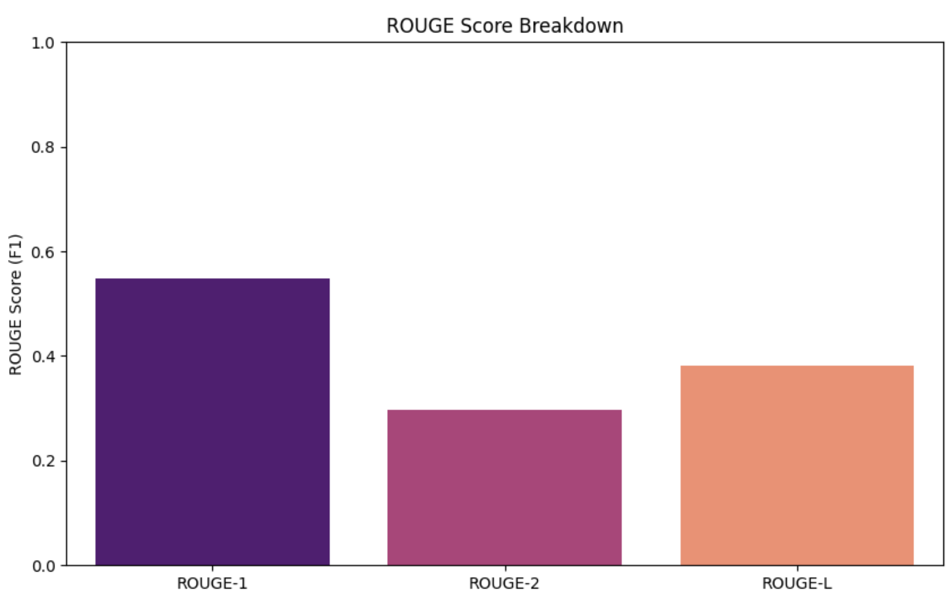

Figure 30. ROUGE Scores Comparison. Created using Python Libraries.

4.2.2.1.5. METEOR

Below is the METEOR score computed for the Model's responses:

```
METEOR Score: 0.45
```

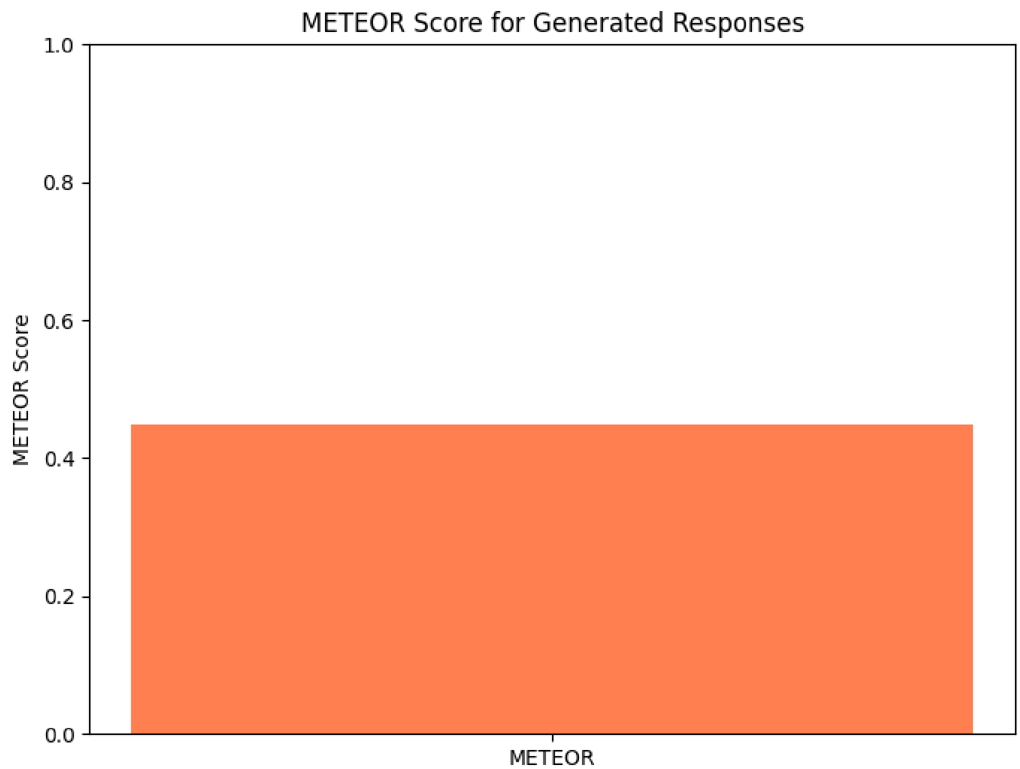

Figure 31. METEOR Score Visualization. Created using Python Libraries.

4.2.2.1.6. Response Time Evaluation

Below are the Total, Average, and 95th Percentile Response times of the Model's Responses:

```
Latency Metrics:

Total time for generating 16123 responses: 19702.30 seconds

Average time per response: 1.22 seconds
```

```
95th Percentile Response Time: 2.28 seconds
```

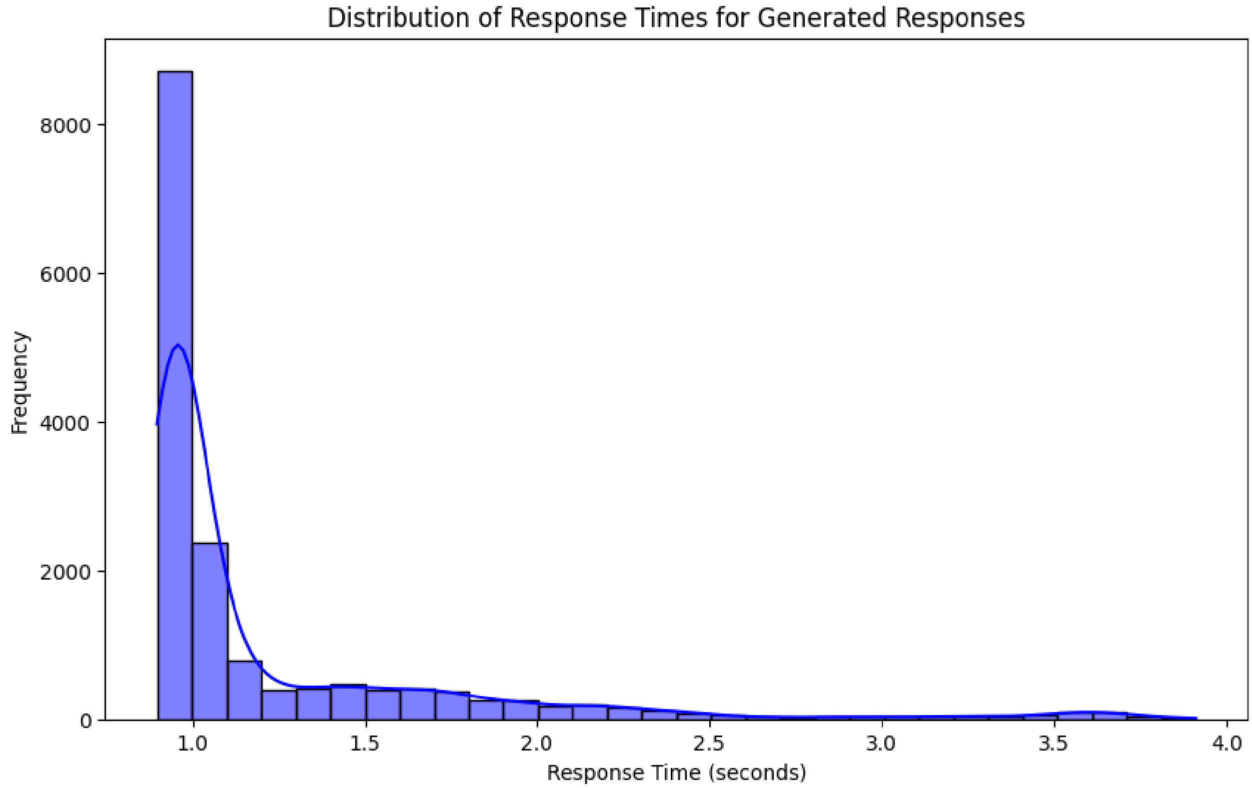

Figure 32. Distribution of Response Times. Created using Python Libraries.

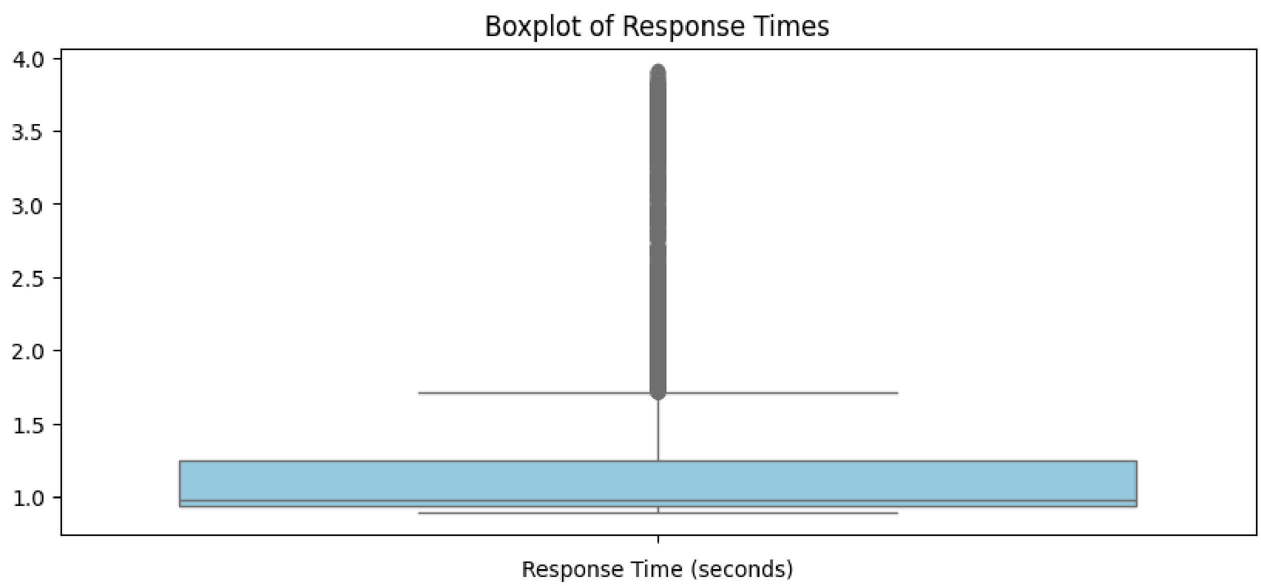

Figure 33. Boxplot of Response Times. Created using Python Libraries.

4.2.2.1.7. TER

Below is the TER score computed for the Model's responses:

```
TER Score: 0.65
```

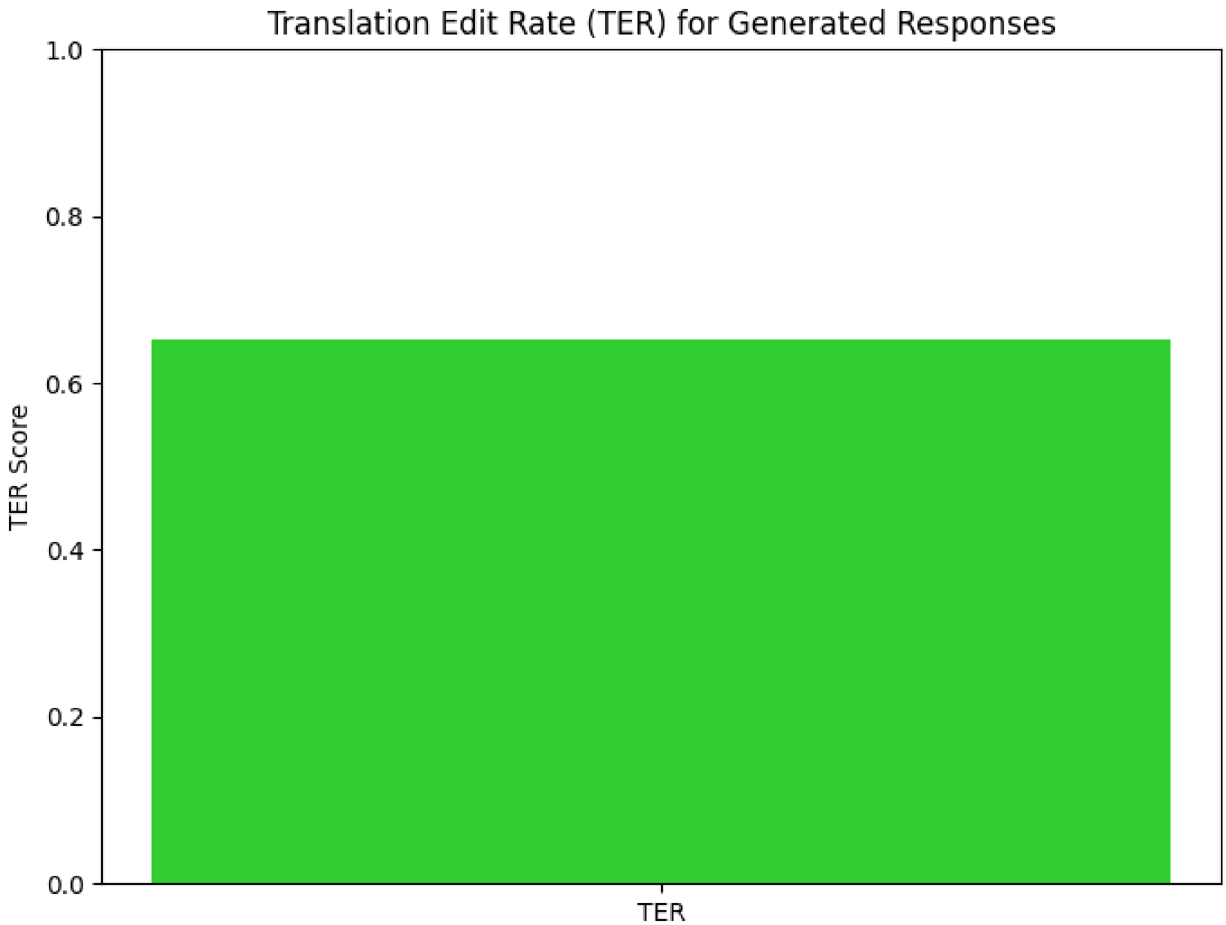

Figure 34. TER Score Visualization. Created using Python Libraries.

4.2.2.1.8. ChrF

Below are the ChrF, ChrF++, ChrF3 scores computed from the Model's Responses and References:

```
ChrF3 Score: 68.86

ChrF Score: 66.44
```

```
ChrF++ Score: 64.09
```

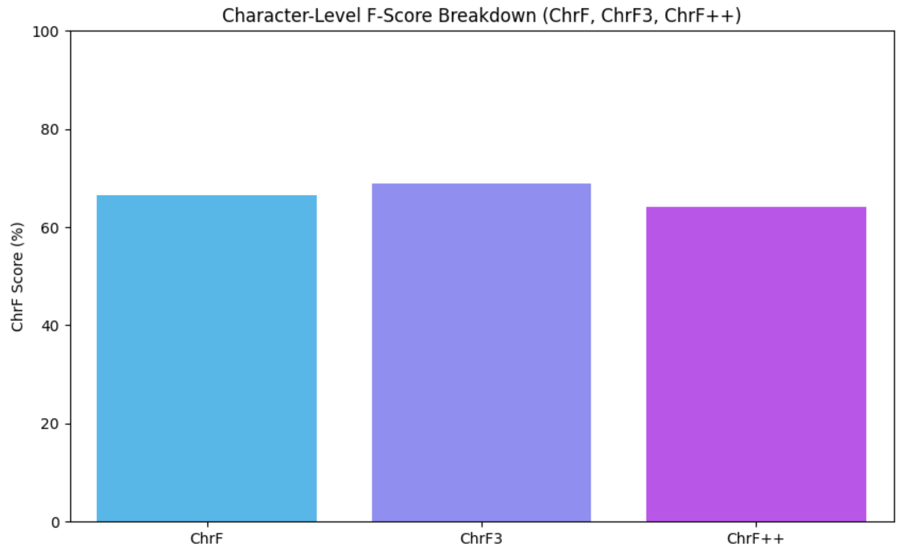

Figure 35. ChrF Scores Comparison. Created using Python Libraries.

4.2.2.1.9. COMET

Below are the COMET – Average & System-level scores computed from the Instructions, model's Responses and References:

```
Average COMET Score (from individual scores): 0.18
System-level COMET Score: 0.18
```

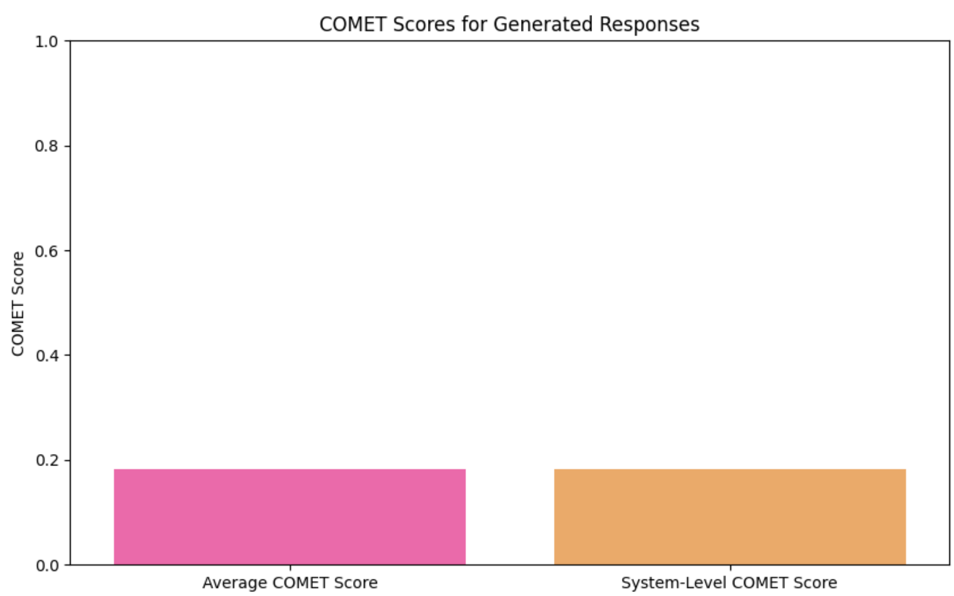

Figure 36. COMET Scores Comparison. Created using Python Libraries.

4.2.2.1.10. Overall Performance Metrics Summary

Radar Chart summarizing all the computed metrics:

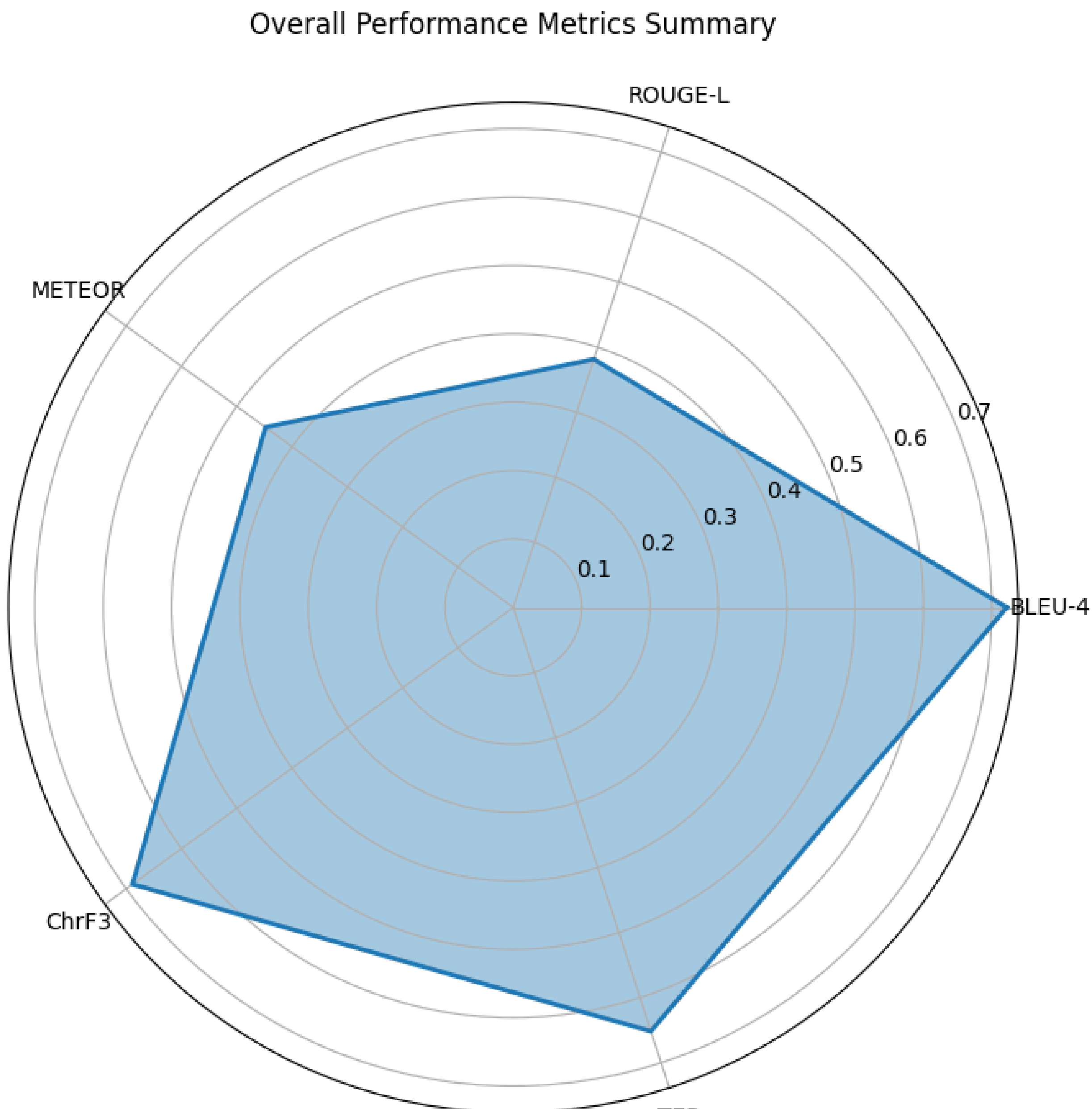

Figure 37. Radar Chart of Overall Model Performance. Created using Python Libraries.

4.2.3. Language-Specific Model Performance Evaluation

This section contains all the results from 4.1.3. Language-Specific Model Performance Evaluation.

4.2.3.1. Implementation Results

4.2.3.1.1. Generate Responses for Each Language's Evaluation Set

The output shows a summary of the progress of the response generation by the model for 20% of instructions from each language in the dataset, the index of all the samples used, including the time taken in seconds to generate a response for each sample.

```
--- Generating Responses and Latencies for Language: en ---

Sample: 1/5374     Time taken for Sample 9329: 0.94 seconds
Sample: 2/5374     Time taken for Sample 4160: 0.94 seconds
Sample: 3/5374     Time taken for Sample 18500: 1.06 seconds
.

.

Sample: 5373/5374  Time taken for Sample 23653: 1.45 seconds
Sample: 5374/5374  Time taken for Sample 836: 1.89 seconds
Data for language en saved to
data/language_specific/generated_responses_en.csv

--- Generating Responses and Latencies for Language: fr ---

Sample: 1/5374     Time taken for Sample 36201: 1.12 seconds
Sample: 2/5374     Time taken for Sample 31032: 0.97 seconds
Sample: 3/5374     Time taken for Sample 45372: 0.95 seconds
.

.

Sample: 5373/5374  Time taken for Sample 50525: 1.83 seconds
Sample: 5374/5374  Time taken for Sample 27708: 2.19 seconds
Data for language fr saved to
data/language_specific/generated_responses_fr.csv
```

```
--- Generating Responses and Latencies for Language: es ---

Sample: 1/5374     Time taken for Sample 63073: 0.95 seconds
Sample: 2/5374     Time taken for Sample 57904: 0.95 seconds
Sample: 3/5374     Time taken for Sample 72244: 0.94 seconds
.

.

Sample: 5373/5374  Time taken for Sample 77397: 1.75 seconds
Sample: 5374/5374  Time taken for Sample 54580: 1.98 seconds
Data for language es saved to
data/language_specific/generated_responses_es.csv
```

4.2.3.1.2. Load Language-Specific Data

This section deals with loading the Evaluation data from each of the saved language-specific datasets, so it does not yield any results.

4.2.3.1.3. BLEU

Below are BLEU – 1, 2, 3, 4 scores computed from the Model's Responses & References in each language and a visualization comparing the scores:

```
--- BLEU Score Evaluation for All Languages ---

BLEU Scores for English:
BLEU-1 Score: 100.00
BLEU-2 Score: 100.00
BLEU-3 Score: 98.02
BLEU-4 Score: 92.00

BLEU Scores for French:
BLEU-1 Score: 100.00
BLEU-2 Score: 100.00
BLEU-3 Score: 98.20
BLEU-4 Score: 95.45

BLEU Scores for Spanish:
BLEU-1 Score: 100.00
BLEU-2 Score: 100.00
BLEU-3 Score: 98.06
```

```
BLEU-4 Score: 93.14
```

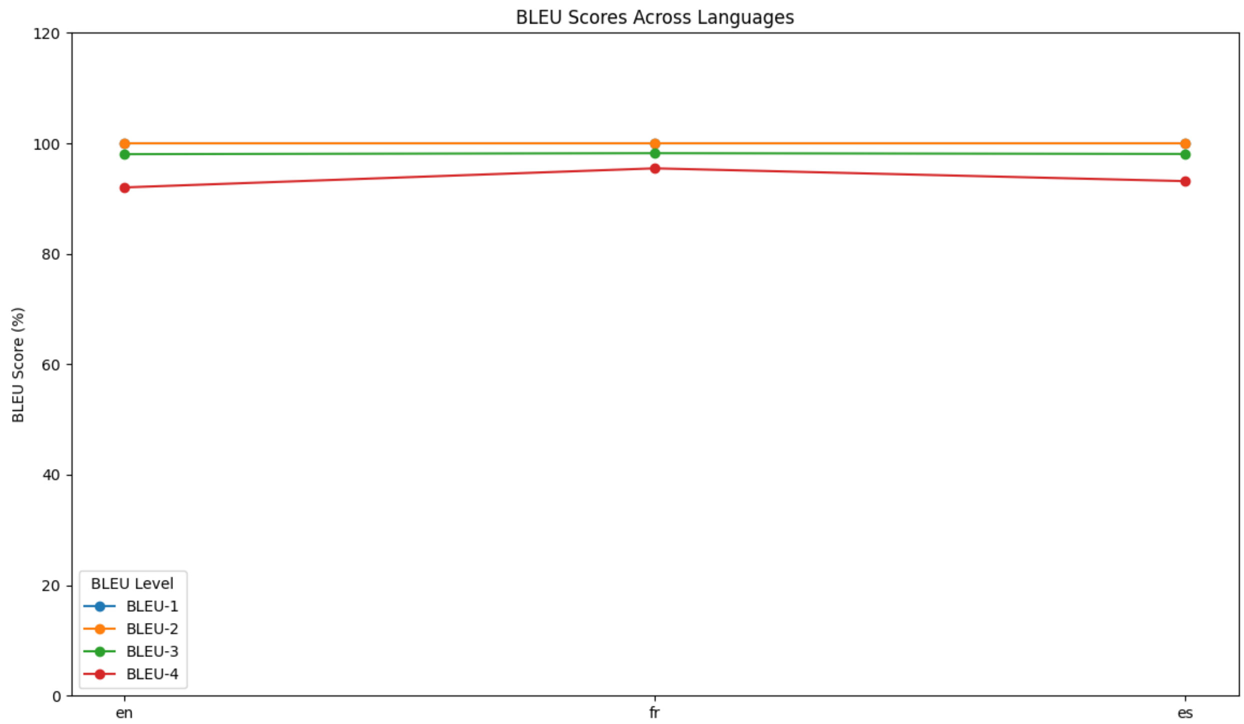

Figure 38. BLEU Scores Comparison across Languages. Created using Python Libraries.

4.2.3.1.4. ROUGE

Below are ROUGE – 1, 2, 3, 4 scores computed from the Model's Responses & References in each language and a visualization comparing the scores:

```
--- ROUGE Score Evaluation for All Languages ---

ROUGE Scores for English:
ROUGE-1 Score: 0.58
ROUGE-2 Score: 0.32
ROUGE-L Score: 0.41

ROUGE Scores for French:
ROUGE-1 Score: 0.53
ROUGE-2 Score: 0.29
ROUGE-L Score: 0.36

ROUGE Scores for Spanish:
ROUGE-1 Score: 0.53
```

```
ROUGE-2 Score: 0.29
ROUGE-L Score: 0.37
```

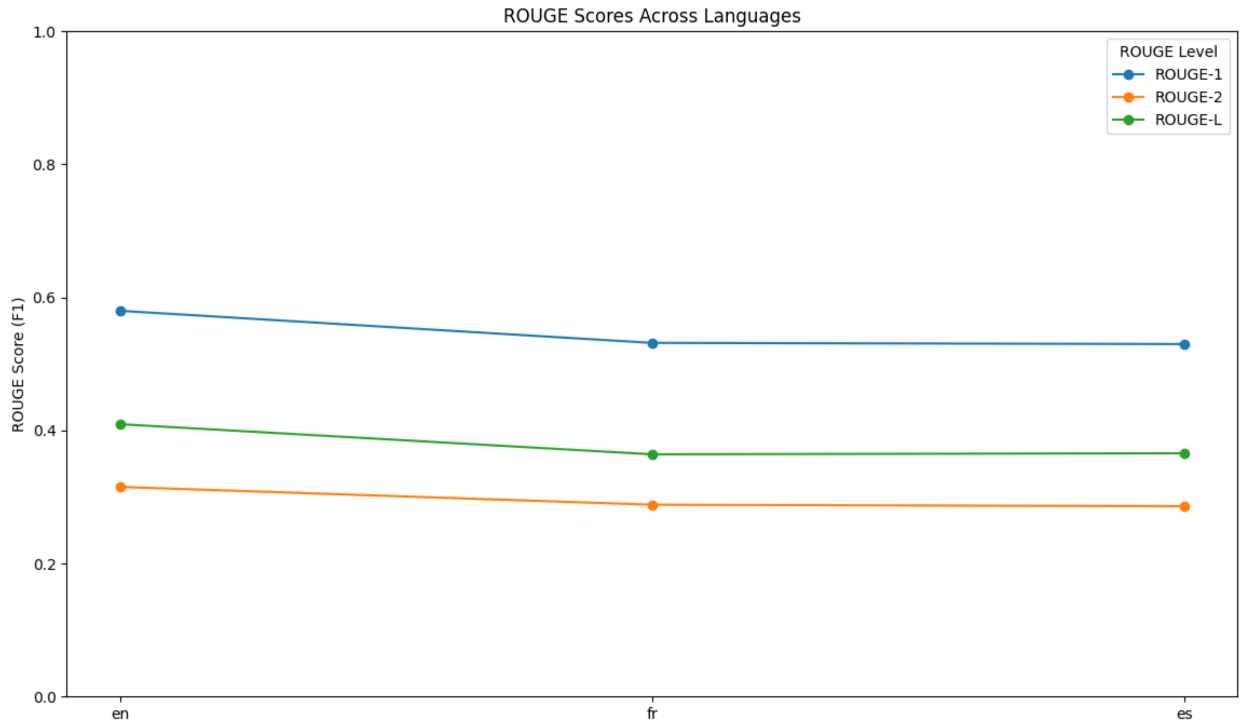

Figure 39. ROUGE Scores Comparison across Languages. Created using Python Libraries.

4.2.3.1.5. METEOR

Below are the METEOR scores computed from the Model's Responses & References in each language and a visualization comparing the scores:

```
--- METEOR Score Evaluation for All Languages ---

METEOR Score for English: 0.48

METEOR Score for French: 0.43

METEOR Score for Spanish: 0.43
```

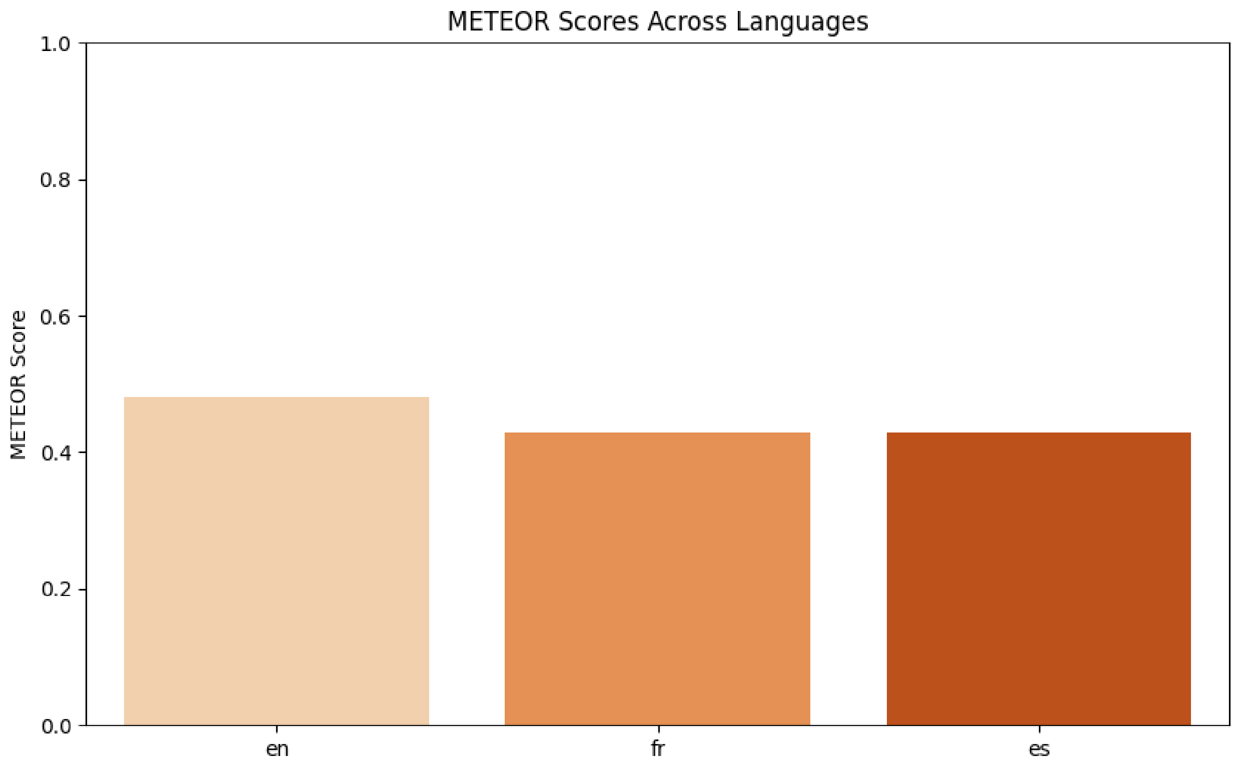

Figure 40. METEOR Scores Comparison across Languages. Created using Python Libraries.

4.2.3.1.6. Response Time Evaluation

Below are the Average, and 95th Percentile Response times of the Model's Responses for all languages and plots visualizing the comparison of these times across them:

```
--- Latency Evaluation for All Languages ---

Average Response Time for English: 1.18 seconds
95th Percentile Response Time for English: 2.04 seconds

Average Response Time for French: 1.28 seconds
95th Percentile Response Time for French: 2.46 seconds

Average Response Time for Spanish: 1.21 seconds
95th Percentile Response Time for Spanish: 2.20 second
```

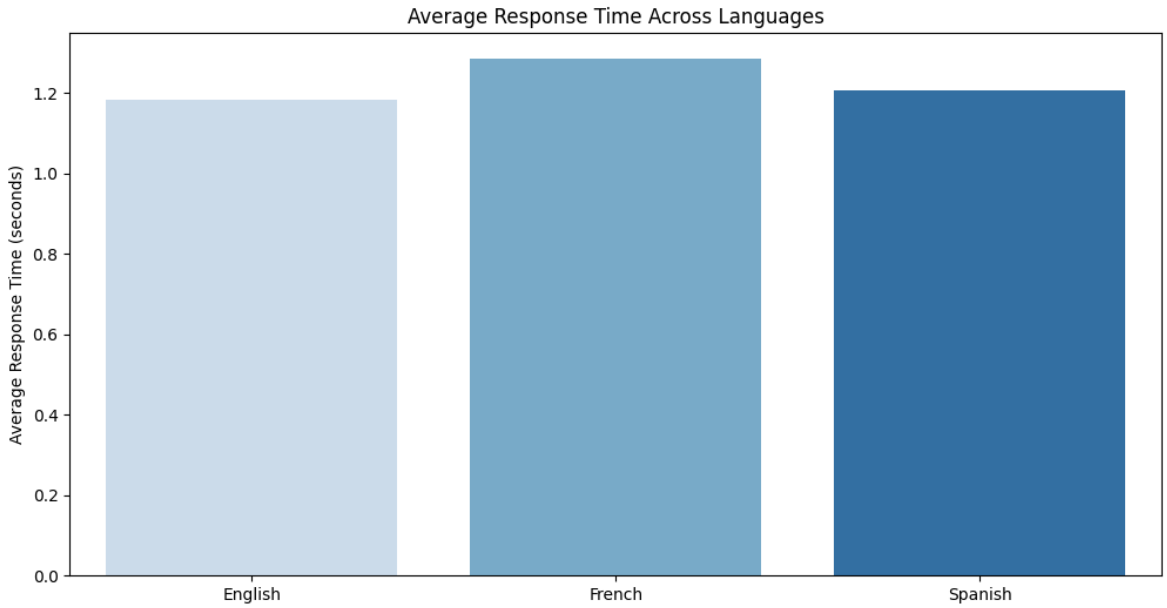

Figure 41. Average Response Times Comparison across Languages. Created using Python Libraries.

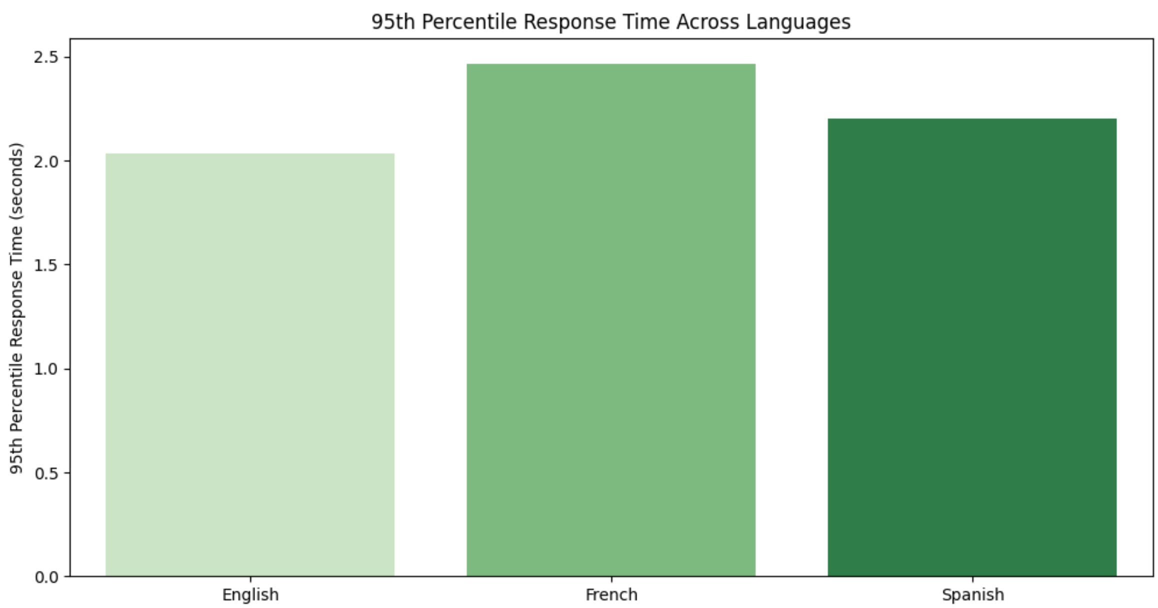

Figure 42. 95th Percentile Response Times Comparison across Languages. Created using Python Libraries.

4.2.3.1.7. TER

Below are the TER scores computed from the Model's Responses & References in each language and a visualization comparing the scores across them:

```
--- TER Score Evaluation for All Languages ---

TER Score for English: 0.54

TER Score for French: 0.55

TER Score for Spanish: 0.58
```

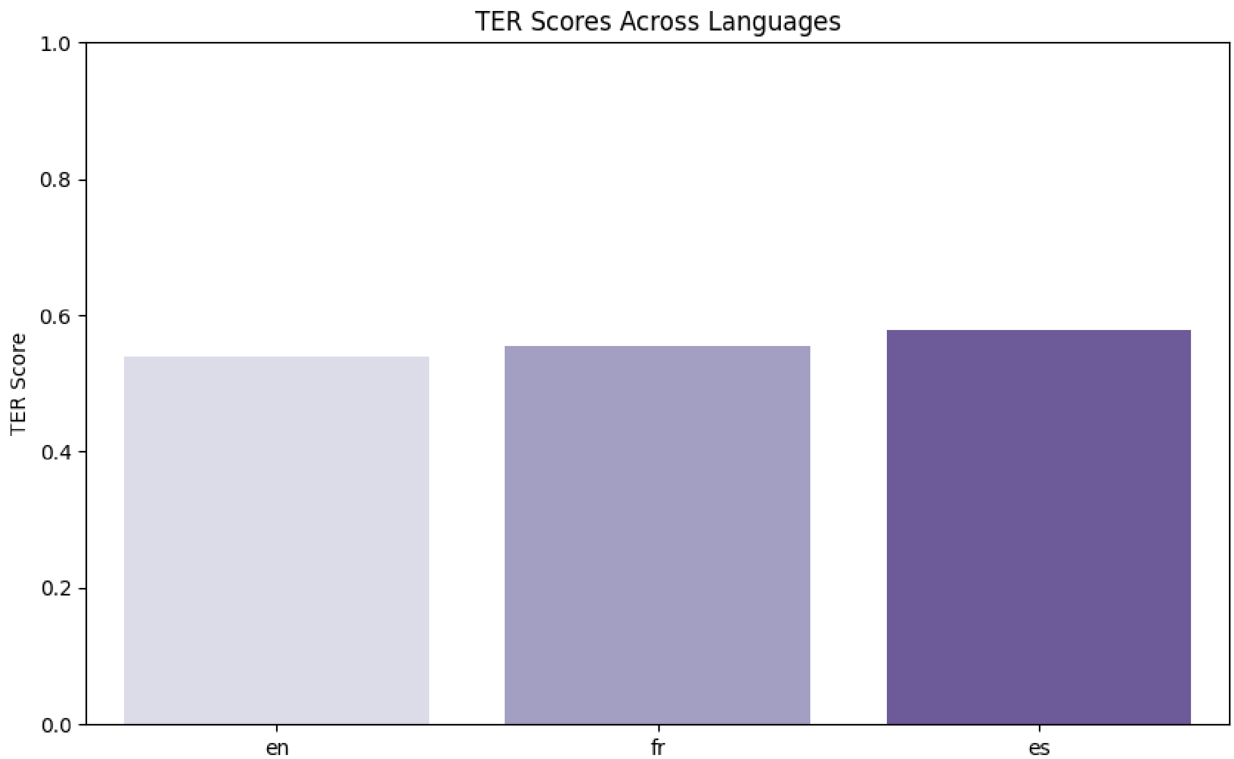

Figure 43. TER Scores Comparison across Languages. Created using Python Libraries.

4.2.3.1.8. ChrF

Below are the ChrF, ChrF++, ChrF3 scores computed from the Model's Responses & References in each language and a visualization comparing the scores across the languages:

```
--- ChrF, ChrF3, and ChrF++ Score Evaluation for All Languages -
--

ChrF Score for English: 62.69
ChrF3 Score for English: 64.37
```

```
ChrF++ Score for English: 61.07

ChrF Score for French: 67.06
ChrF3 Score for French: 71.44
ChrF++ Score for French: 66.45

ChrF Score for Spanish: 64.45
ChrF3 Score for Spanish: 70.05
ChrF++ Score for Spanish: 61.41
```

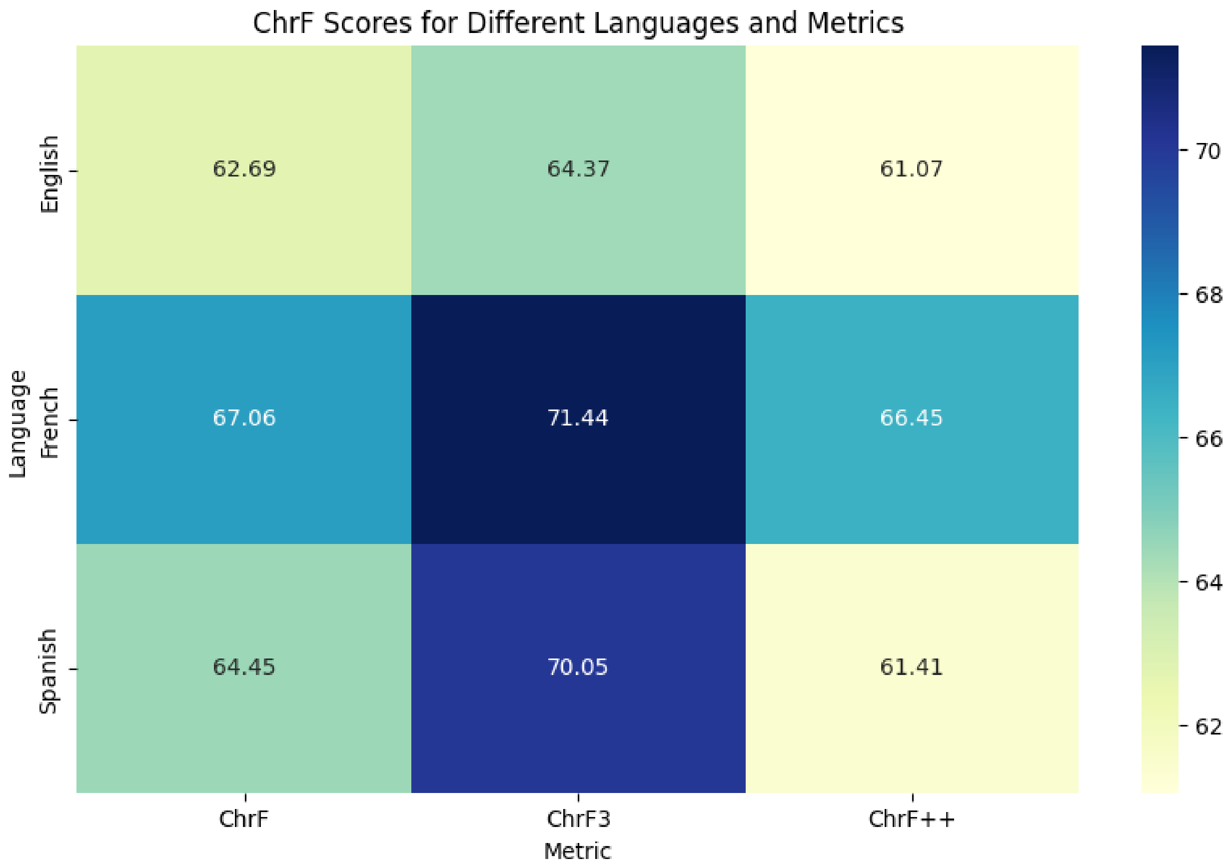

Figure 44. ChrF Scores Comparison across Languages. Created using Python Libraries.

4.2.3.1.9. COMET

Below are the COMET – Average & System-level scores computed from the Instructions, model's Responses and References in each language:

```
--- COMET Score Evaluation for All Languages ---
```

```
Average COMET Score for English (from individual scores): 0.16
System-level COMET Score for English: 0.16

Average COMET Score for French (from individual scores): 0.14
System-level COMET Score for French: 0.14

Average COMET Score for Spanish (from individual scores): 0.25
System-level COMET Score for Spanish: 0.25
```

4.2.3.1.10. Cross-Language Performance Metrics Summary

Radar Chart summarizing all the computed metrics for each of the languages English, French, and Spanish:

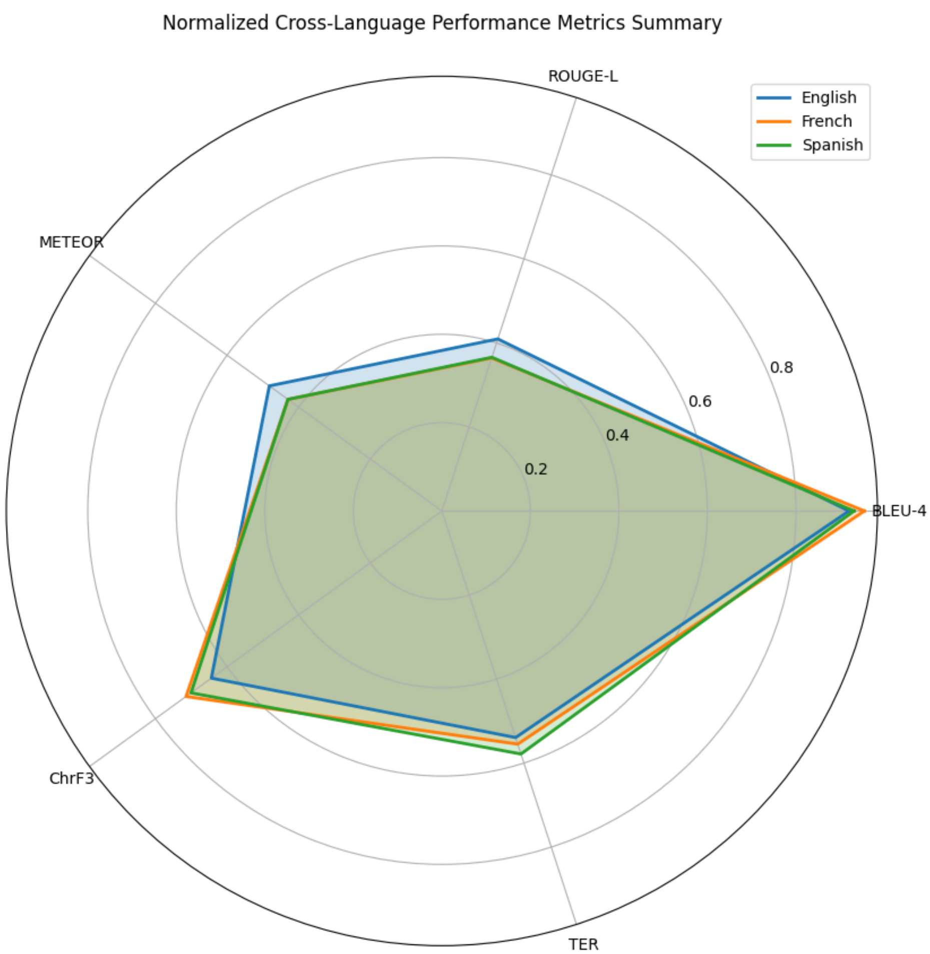

Figure 45. Radar Chart of Language-Specific Model Performance. Created using Python Libraries.

4.2.4. Interface Testing Results

This section contains all the results from 4.1.4. Interface Testing

4.2.4.1. Results from Testing Inputs

4.2.4.1.1. Text-Box

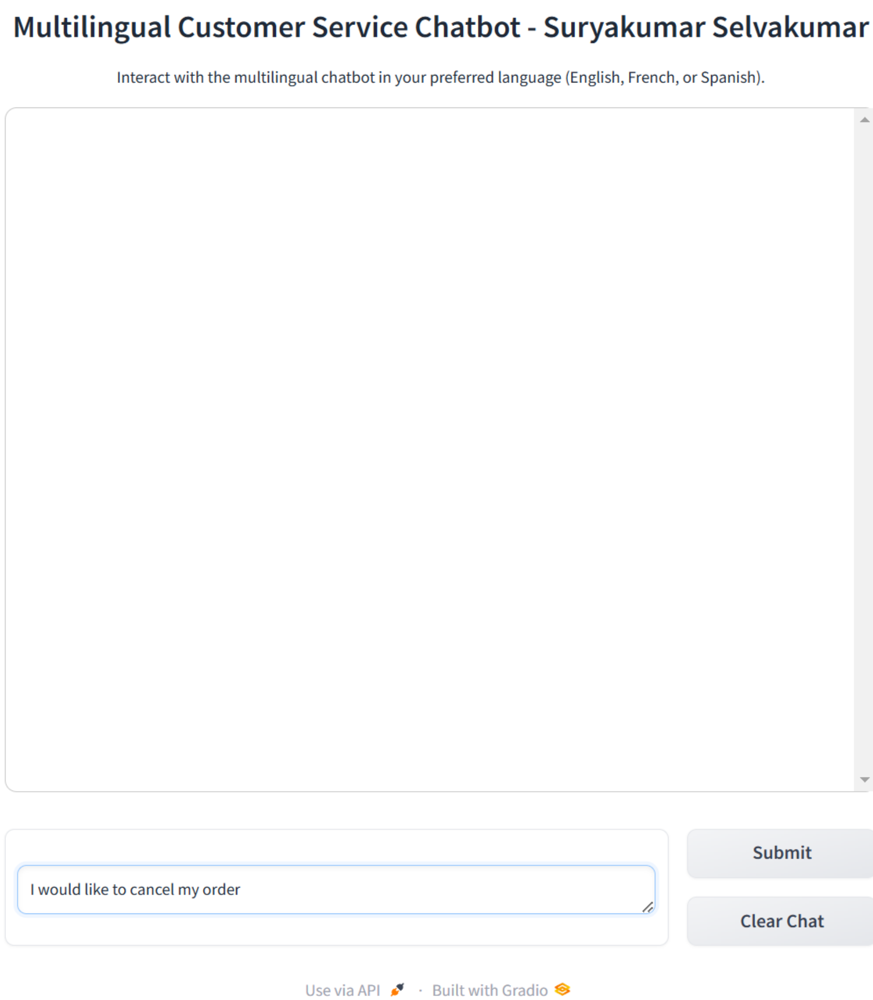

Figure 46. Text-Box Working. Created with Python & Gradio.

4.2.4.1.2. Submit Button

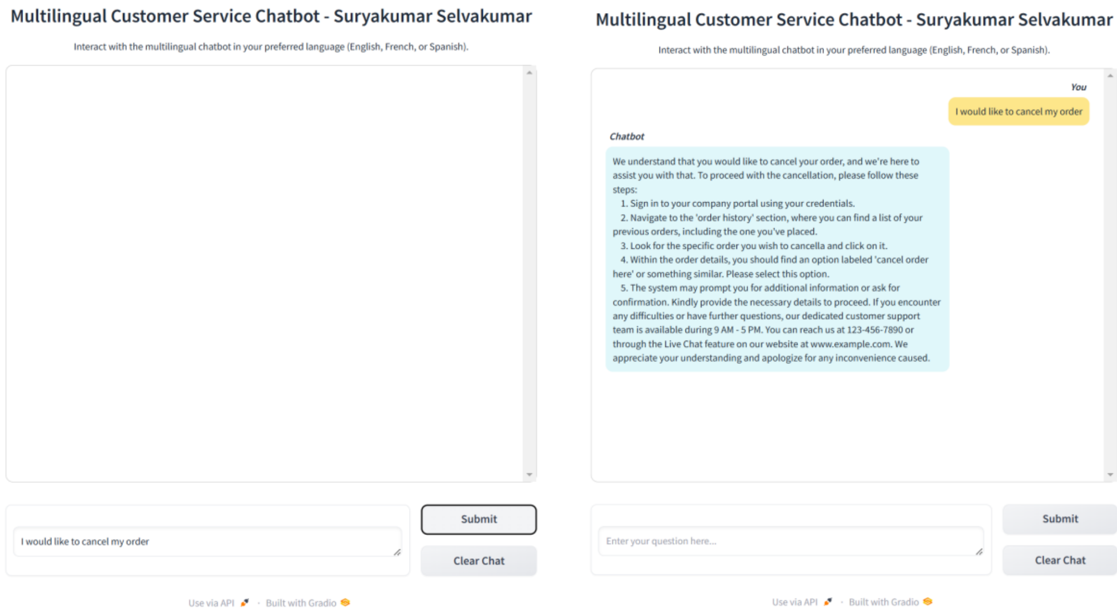

Figure 47. Submit Button Working. Created with Python & Gradio.

4.2.4.1.3. Clear Chat Button

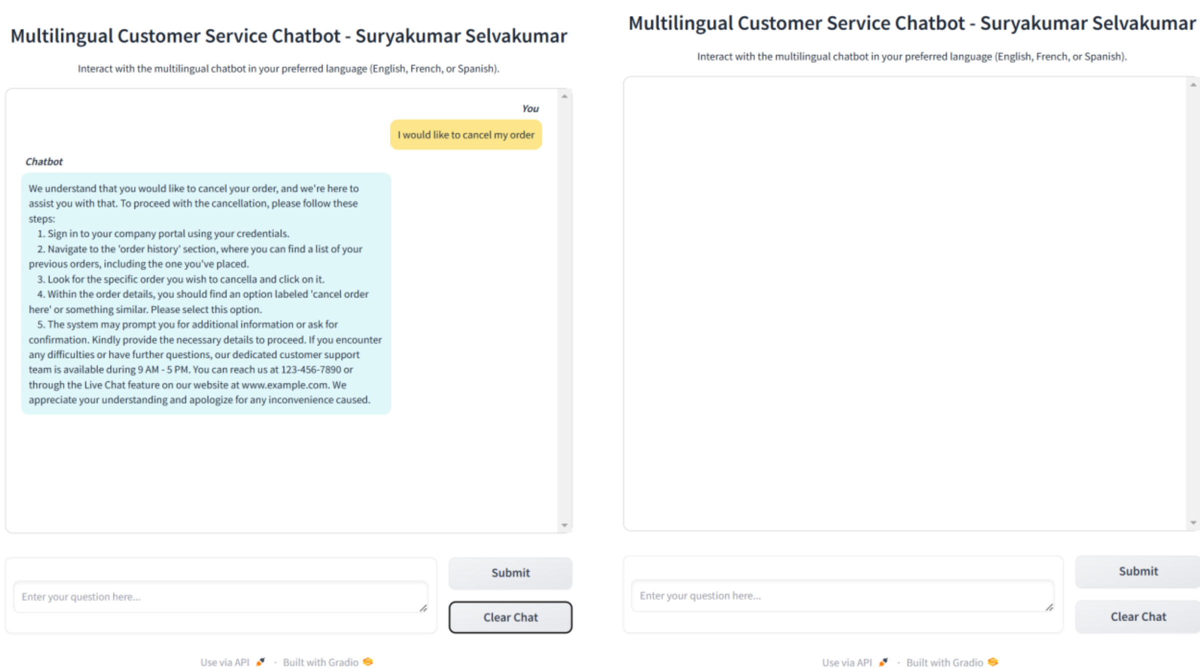

Figure 48. Clear Chat Button Working. Created with Python & Gradio.

4.2.4.2. Results from Testing Conversation Window

4.2.4.2.1. English Interactions

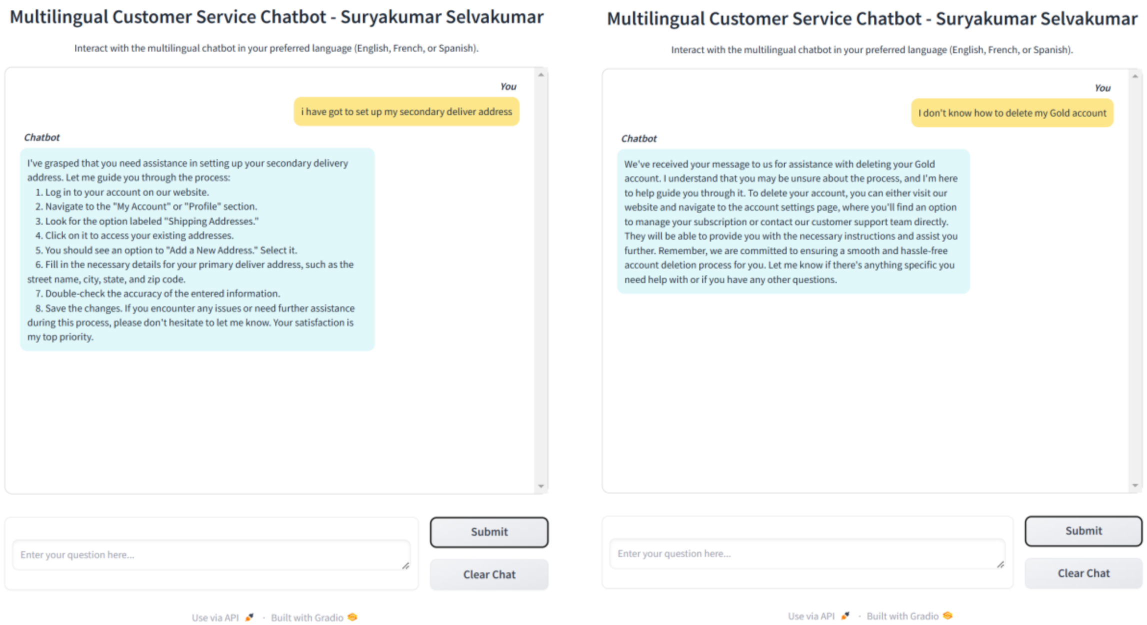

Figure 49. English Interactions. Created with Python & Gradio.

4.2.4.2.2. French Interactions

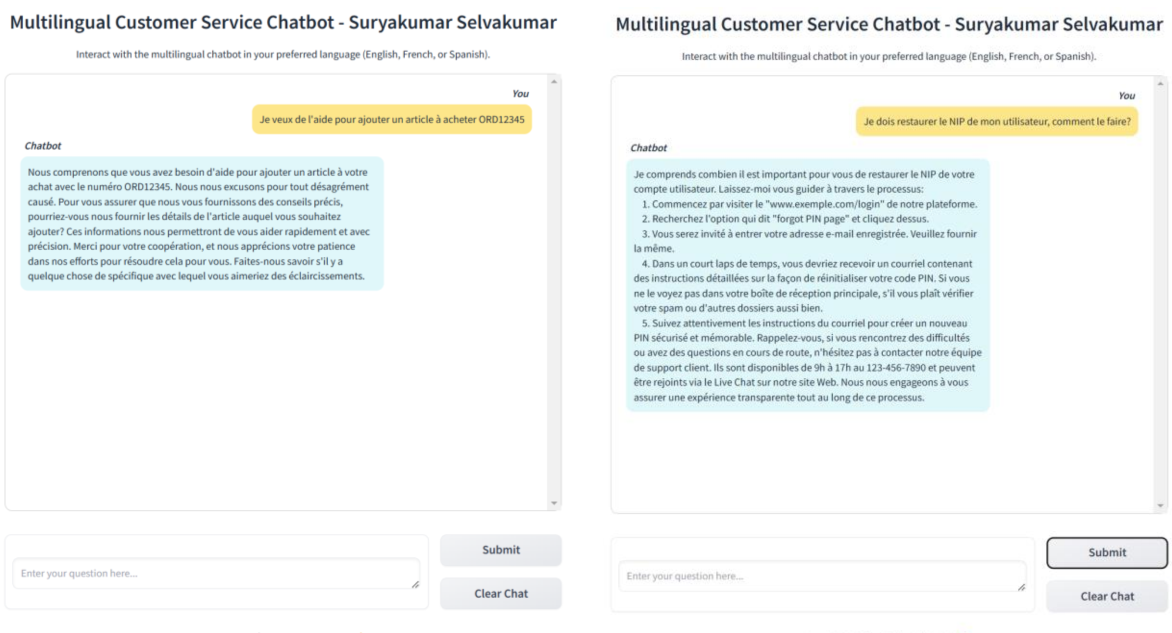

Figure 50. French Interactions. Created with Python & Gradio.

4.2.4.2.3. Spanish Interactions

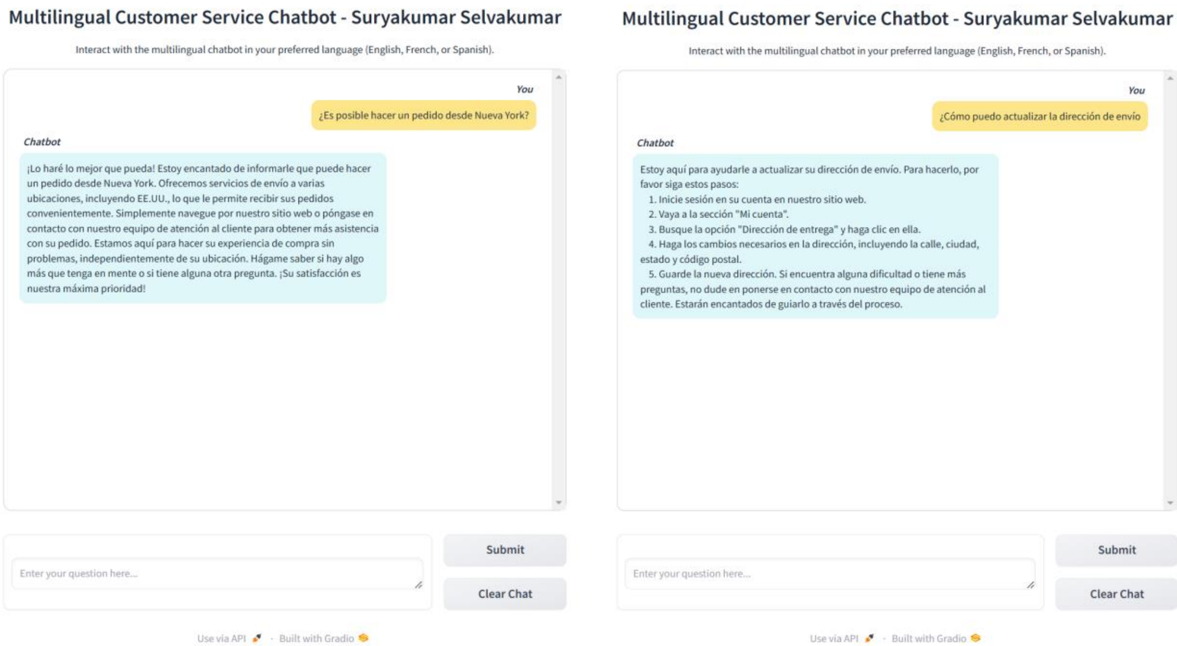

Figure 51. Spanish Interactions. Created with Python & Gradio.

4.3. Analysis

This section contains the explanations/interpretations of each of the results that were described in 4.2. Results. For the actual raw results, please refer to 4.2. Results which contains the results under the same section numbers and headings.

Analysis can be divided into four parts:

1. Analysis of Sanity Testing Results
2. Analysis of Overall Model Performance Evaluation Results
3. Analysis of Language-Specific Model Performance Evaluation Results
4. Analysis of Interface Testing Results

Find the flow-chart that visualizes this section below:

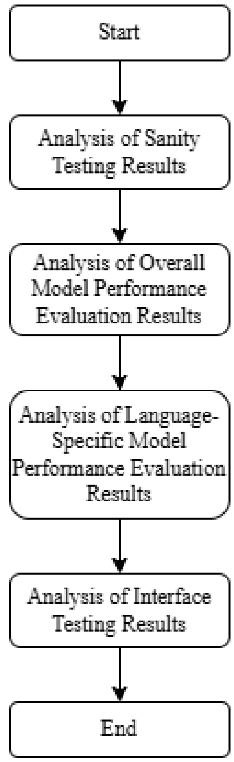

Figure 52. Flow-Chart of Testing Results Analysis. Created using [13].

4.3.1. Analysis of Sanity Testing Results

This section contains the interpretation of all the results from 4.2.1. Sanity Testing Results.

4.3.1.1. Analysis of Implementation Results

4.3.1.1.1. Analysis of Response Generation Function

This section simply deals with Function Definition so it does not yield any results, so no analysis required.

4.3.1.1.2. Analysis of mBART Training Progress Visualization

Interpretation of Figure 27. Plot of Training Progress:

We notice that the Training loss starts at 1.280400 in the 1st epoch and ends at 0.529600 at the 7th epoch, meaning that our model has learned effectively over the course of training, minimizing its loss every step of the way. The same pattern can be observed with the Validation loss as well but its loss decreases slower than training loss and plateaus at epoch 5 with 0.676866 loss value. There are improvements in loss after this point but they are marginal so validation loss reaches an optimal point. At the end, Convergence between training and validation loss can be observed as the gap between the two consistently lowered at each epoch, so we can conclude that the model performs well when it encounters unseen data.

4.3.1.1.3. Response Generation for Custom Instruction

Analysis of the responses generated by the model for the custom instruction:

The model's responses are coherent and structurally well put-together in each of the languages. The English Response has logical start and end points, with a natural progression of the instructions to be followed for cancelling the order, thus forming a grammatically and logically robust response, which was further bolstered by a Grammar score of 89% from quillbot.com's Grammar Checker. The French and Spanish responses were also checked for fluency and grammatical accuracy using each language's Grammar checkers online – bonpatron.com & linguix.com which resulted in a score of 72% for French and 91% for Spanish. Thus, the same pattern can be observed with French and Spanish responses as well for the input custom instruction.

4.3.1.1.4. Response Generation for Random Instructions from Dataset

Analysis of the responses generated by the model for five random instructions from the dataset fetched with a random state of 42:

The first random instruction that was used to generate a model response is in Spanish. The generated response was checked using linguix.com's grammar checker which returned a score of 62%.

The second random instruction that was used is a French instruction. The generated response was checked using bonpatron.com's grammar checker which returned a score of 86%

The third random instruction that was used is a French instruction. The generated response was checked using bonpatron.com's grammar checker which returned a score of 86%.

The fourth instruction is also French. The generated response was checked using bonpatron.com's grammar checker and it returned a score of 82%.

The last instruction and response are in English. The generated response returned a score of 98% when checked with quillbot.com's tool.

4.3.2. Analysis of Overall Model Performance Evaluation Results

This section contains the interpretation of all the results from 4.2.2. Overall Model Performance Evaluation.

4.3.2.1. Interpretation of Implementation Results

4.3.2.1.1. Dataset Normalization

This section deals with applying a Function to the Multilingual Dataset, so it does not yield any results, so no analysis is required.

4.3.2.1.2. Generate Responses for Evaluation Set

Responses for 16123 instructions (20% of the dataset), are generated. The entire progress is logged, including the index of the instruction to which the response was generated for, and the time taken to generate the response.

4.3.2.1.3. BLEU

Interpretation of the BLEU – 1, 2, 3, 4 scores and figure. 28:

- The BLEU – 1 score is a found to be 100% which means each word present in the generated responses also show up in the references.
- The BLEU – 2 score is 96.58% which means almost all pairs of consecutive words that appear in generated responses also appear in the references. This is a great score which contributes to the coherency check of our responses.
- The BLEU – 3 score is 86.21% which means tri-grams or sets of three consecutive words overlap between generated responses and references 86.21% of the time. This score, while lower than the previous one is still a good one.
- 72.17% is the BLEU – 4 score that was computed and this indicates that when phrases with 4 words are checked across responses and references, 72.17% match which is also a very good score.

- A common pattern that can be observed is that when the complexity of the word overlap check increases, the metric computed lowers.

Interpretation of the BLEU – 4 scores by Sentence Length Category and figure. 29:

- The BLEU – 4 score for Sentences that are short in Length is not computed, meaning, none of the responses are in the length range (0, 250).
- The BLEU – 4 score for Sentences that are of Medium Length is around 50%.
- The BLEU – 4 score for Sentences that are of Long Length is around 72%.
- Longer responses seems to have a better BLEU – 4 score. The reason could be that when the model has a bigger sample size, it gets trained with more context, and so it is able to generate more phrases of 4-grams that match with the reference responses.

4.3.2.1.4. ROUGE

Interpretation of the ROUGE – 1, 2, L scores and figure. 30:

- The ROUGE – 1 score is 55% meaning that many percentage of 1-grams or single words from the references are also existing in the model's responses. This score is good and satisfies industry standards for a good ROUGE – 1 score.
- The ROUGE – 2 score is 30% meaning that pairs of consecutive words overlap between references and responses 30% of the time. This is a moderate score as scores between 20% and 40% are considered moderate as per industry standards.

- The ROUGE – L score is 38% which means that the LCS (Longest Common Subsequence) between the references and responses is 38% similar. This is a good score as scores around 40% are considered good scores.
- Overall, we can conclude that our model's performance as determined by ROUGE is good.

4.3.2.1.5. METEOR

Interpretation of the METEOR score and figure. 31:

- The METEOR score that is returned is the computed average of the METEOR scores of each (reference, response) pair and this value is 0.45.
- METEOR score is a comprehensive metric that checks for precision, recall, synonyms, stemming, and alignment between each of the references and responses.
- Thus, we can conclude that our model captures 45% of the key information from the references when all the above factors are considered.
- This score is very good as scores above 30% are considered good in the NLP industry.

4.3.2.1.6. Response Time Evaluation

Interpretation of the Latency Metrics:

- The Average time taken by the model for generating responses is 1.22 seconds, which is very good as industry standards dictate that response times of a chatbot that are below 5 seconds are good, with response times closer to 1 second considered the best and leads to good user experience.

- The 95th percentile response time for the model is 2.28 seconds, which means 95% of the model's responses get generated with 2.28 seconds. This is again a good response time by the current convention.
- While the model's response times are largely dependent on the system generating the response and the internet connection of the user, this gives us a benchmark for the kind of performance that users can expect.
- The GPU used to generate the responses is the Nvidia RTX 4070 Ti and since the response times are good with this one, it will most likely be better if the model is hosted on a server with a state-of-the-art GPU.

Interpretation of the Histogram in figure. 32:

- The x-axis denotes the response times of the model and the y-axis represents the frequency of each range of response times.
- The first bar in the plot is the tallest exactly before the 1 second mark, reaching up to a frequency of 9000. This means that around 9000 responses were generated within a time in the range (0.9, 1.0) seconds.
- The next couple bars are the other two noteworthy bars in the plot, with around 2250 responses generated in time (1.0, 1.1) seconds and around 900 responses generated in time (1.1, 1.2) seconds.

- The remaining responses are relatively marginal in frequency meaning that only a small range of responses took much longer to generate with response times in the range 1.5 seconds to 4 seconds, compared to the entire set.
- This suggests that most of the model's responses get generated in less than 1.5 seconds with rare cases taking longer, thus we can conclude that our model's response time performance is good.

Interpretation of the Box-Plot in figure. 33:

- The Box-Plot is another way to visualize the distribution of the model's response times.
- The 25th percentile response time is around 0.9 seconds, median is around 1 second, and 75th percentile is around 1.25 seconds.
- The whiskers specifying 1.5 times the interquartile range are at 0.8 seconds and 1.75 seconds respectively and there are huge numbers of outliers outside the whisker range.
- This reflects the same pattern we observed from the histogram, with the median response time being 1 second and a very narrow interquartile range, meaning our model is almost always fast and responsive.

4.3.2.1.7. TER

Interpretation of the TER score and figure. 34:

- The TER score that is computed is 0.65.

- This score indicates that 65% of the model's response has to be edited for it to match the reference response for each response on average.
- This score is poor when judged by industry standards which sets the bar at 30% or lower for the TER score to be considered a good score. This tells us the model could use some further fine-tuning to facilitate a more efficient capture of the reference responses.

4.3.2.1.8. ChrF

Interpretation of the ChrF, ChrF++, ChrF3 scores and figure. 35:

- The ChrF score which gives equal weightage to Precision and Recall is 66.44, which means there is 66.44% of perfect overlap with respect to characters between the model's responses and references. This score satisfies industry standards for a good ChrF score as a score above 60 is considered to be good.
- The ChrF3 score which gives more weightage to Recall is 68.86, meaning that when trying to ensure the model's response captures as much of the reference text as possible, the match between the two increases. This is also a good score as the standard to be met is the same at above 60.
- The Chrf++ score which gives equal importance to character-level and word-level convergence is 64.09, meaning that when both character and word overlap are checked between the model's responses and references, together they result in a 64.09% match. This score is also above 60 so it is also a good score.

4.3.2.1.9. COMET

Interpretation of the COMET – Average & System-level scores and figure. 36:

- The average COMET score from averaging all the individual scores computed from each instruction, response, reference triplet is 0.18. This score is quite low, as the industry standard for a good score is one above 0.5.w
- The System-level COMET score computed by the COMET model is also 0.18 meaning that when the model considers all the responses holistically, the resultant score matches the average score.
- This conveys that when model responses are evaluated by the COMET metric, it performance is poor. Since the COMET model checks for both semantic meaning and fluency, this indicates that our training data could use improvement in terms of scale and quality of responses.

4.3.2.1.10. Analysis of Overall Performance Metrics Summary

Interpretation of the Plot in figure. 37:

- The Radar Chart gives us a holistic view of the model's performance by visualizing the most vital metrics.
- The plot shows a balanced performance between BLEU – 4 & ChrF3, and METEOR & ROUGE – L.
- The performance that is noticeably poor is the TER score.
- Overall, the mBART model's performance satisfies the many criteria followed in the industry for a good NLP model, however, since the radar chart point is not as smaller than 0.3 for TER, perhaps the model could benefit from further training

with better parameters or our response generation parameters could potentially use some reconfiguration to allow the model to capture the references better.

4.3.3. Language-Specific Model Performance Evaluation

This section contains the interpretation of all the results from 4.2.3. Language-Specific Model Performance Evaluation.

4.3.3.1. Interpretation of Implementation Results

4.3.3.1.1. Generate Responses for Each Language's Evaluation Set

Responses for 5374 instructions (20%) in each language are generated. The entire progress is logged, including the index of the instruction to which the response was generated for, and the time taken to generate the response.

4.3.3.1.2. Load Language-Specific Data

This section deals with loading the Evaluation data from each of the saved language-specific datasets, so it does not yield any results, so no analysis is required.

4.3.3.1.3. BLEU

Interpretation of the BLEU – 1, 2, 3, 4 scores of English, French, & Spanish and figure. 38:

- BLEU – 1 & 2 scores for all three languages are a perfect 100%, meaning that all the model generated responses' uni-grams and bi-grams are present in the references. This can also be observed in the line chart, where lines for BLEU – 1 & 2 overlap perfectly with BLEU – 2 score on top.

- BLEU – 3 scores across the languages are almost the same at 98% with marginal differences, indicating that overlap of three consecutive words from the generated responses in the references is also quite high, almost perfect.
- Although minor, there is some difference between the BLEU – 4 scores across the languages with English responses having a score of 92%, French responses having a score of 95.45%, and Spanish responses having a score of 93.14%. This suggests that 4-gram phrases in French responses overlap the most with the references, followed by Spanish and then English. This could be due to the nature of French, where the same sentence is usually longer than in Spanish and English, leading to more overlap between 4-gram phrases. Same is true with respect to Spanish and English where Spanish sentences are longer.
- Overall, the model's performance in terms of Precision is extremely good, rivaling the BLEU scores of state-of-the-art models, and the differences across the languages are negligible and so do not form a case for bias.

4.3.3.1.4. ROUGE

Interpretation of the ROUGE – 1, 2, L scores of English, French, & Spanish and figure. 39:

- ROUGE – 1 score is the highest for English Responses at 58%, followed by French and Spanish which have the same score at 53%. This indicates that English has the highest number of uni-gram overlap between references and responses, i.e., model's recall is better for English than for Spanish and French, even though the difference is only 5%.

- Same pattern can be observed with ROUGE – 2 scores as well with English having the highest at 32% and French and Spanish both having 29%. Thus, bi-gram overlap between references and responses are the highest in English.
- ROUGE – L score also exhibits the same pattern. English has a score of 41%, French has a score of 36%, and Spanish has a score of 37%. Thus, the overlap of the Longest Common Subsequence (LCS) is the highest in English references and responses.
- We can conclude that English responses capture its references the most. This could be due to two reasons:
    - ➢ Grammar of the English language is simpler when compared to French and Spanish, so the mBART model was probably able to capture the references easier during fine-tuning.
    - ➢ mBART tokenization's capability for languages other than English could be lesser since the other languages tend to have vocabulary of different sizes, different semantics or morphology.
- In summary, the model exhibits some bias towards English instructions but overall, ROUGE – 1 scores for all three languages are very good as scores around 50% are considered the best in the industry, however, ROUGE – 2 & L scores could use some improvement, as they are both considered moderate when in the range 30% to 40%, with scores above 40% considered good.

4.3.3.1.5. METEOR

Interpretation of the METEOR scores for English, French, & Spanish Responses and figure. 40:

- English has the highest METEOR score at 48%, while French and Spanish have the same score at 43%.
- This suggests that when factors such as Precision, Recall, Synonymy, Stemming, and Alignment are considered holistically, English responses seem to be the best and French & Spanish responses of the model are ranked lesser.
- The reason for English having the best score out of the three could be the same as the ones outlined in the previous section 4.3.3.1.4.
- Even though the model seems to perform better for English instructions, all three scores pass the industry criteria for a great score as scores above 40% are regarded as high.

4.3.3.1.6. Response Time Evaluation

Interpretation of the Latency metrics, and figures 41 & 42:

- The average response time for English seem to be the best at 1.18 seconds, and then next up is Spanish at 1.28 seconds and the last is French at 1.21 seconds. This could be explained by the length and grammatical complexity of the responses. English sentences tend to be simpler and use simple grammar while Spanish responses are longer, and French responses are the longest, each having their own grammatical rules.

- The 95th percentile response time exhibits the same pattern as observed with the average response times for each of the languages. This could be explained by the same reasoning as before.
- In summary, the response times of all three languages are great as per the industry and the minor differences between them have a logical reason for their existence, so no bias can be found in terms of the model's response generation speed.

4.3.3.1.7. TER

Interpretation of the TER scores for English, French, & Spanish Responses and figure. 43:

- The TER score of Spanish is the highest among the three languages at 58% which indicates that Spanish responses had to be edited the most for them to resemble their reference responses. This could be due to Spanish's morphological complexity, that is its sentences tend to change depending on context, mood, and subject matter, leading to translation challenges, and therefore the number of edits needed to make the response resemble the reference are higher.
- The next is French at 55% and then after that is English at 54% with TER scores almost the same. This suggests that French and English responses are similar when checked in terms of their resemblance with their references. This means the model performs quite well with French because French is a more complex language than English with longer sentences and more complex grammar but the number of edits required to make its responses match its references is very close to what English responses require.

- Unfortunately, all three scores are poor as the industry standard for TER is 30% or lower, indicating our model's finetuning parameters could use some improvements to allow the model to capture the references better.
- There is mild bias against Spanish by the model, indicating that the translation process with Spanish could use some improvements.

4.3.3.1.8. ChrF

Interpretation of the ChrF, ChrF3, ChrF++ scores for English, French, & Spanish Responses and figure. 44:

- ChrF score for French is the highest at 67%, with Spanish in second place at 64.45%, and English the last at 62.69% . This indicates that when character-wise precision and recall are both considered, French responses rank the highest. This could be due to French responses having the largest number of characters as conveying the same sentences in French as opposed to English or Spanish takes more words in general, and this probably led to more word overlap between the responses and references.
- ChrF3 score is again topped by French with a score of 71.44% but Spanish is very close at 70.05% and English is the lowest with 64.37%. The higher score for French can be explained with the same reasoning as before and another insight is that when recall is given more importance, the model performance is almost the same for both French and Spanish.
- Same pattern can be observed again for ChrF++ scores, with French topping the list at 66.45%, Spanish and English following along. However, something

interesting is that the score for Spanish and English responses is almost the same with the former having a score of 61.41% and the latter having a score of 61.07%, meaning that when both Character-level and Word-level accuracy are considered, the model's performance in terms of overlap between responses and references is the same for English and Spanish.

- Holistically, all three scores for each language are good scores as per industry standards which sets the bar at 60% or above for ChrF score variants and no bias can be found as the scores are the result of the nature of the languages themselves.

4.3.3.1.9. COMET

Interpretation of the COMET – Average & System-level scores computed from the Instructions, model's Responses and References in each language:

- The Average COMET score is the highest for Spanish at 25%, with English in the middle with 16% and French at last with 14%. COMET models are trained with quality estimation algorithms which not only check for sematic accuracy but also for contextual matching and sentence coherence, so the higher score for Spanish indicates that the model was able to capture the contextual meanings of Spanish references better than for English and French during training.
- The exact same behavior can be observed with System-level scores as they are exactly the same as their Average score counterparts.
- Overall, if COMET scores for each languages' responses are to be given more importance, then we can conclude that there is notable bias by the model toward Spanish. Thus, the model's training parameters could use some further testing and

tweaking to find the optimal parameters that provide similar and good performance for all three languages.

4.3.3.1.10. Analysis of Cross-Language Performance Metrics Summary

Interpretation of the Radar Chart summarizing all the computed metrics for each of the languages English, French, and Spanish from figure 45:

- We see that the performance of the model for French and Spanish is similar with respect to three metrics – ChrF3, METEOR, & ROUGE – L. The performance of the model for French is slightly lower when BLEU – 4 score and slightly higher when TER score are considered. Overall, the model’s performance is more or less similar for Spanish and French.
- The model’s performance for English is interesting, when METEOR, TER and ROUGE – L scores are considered, English performance is much better but when ChrF3 and BLEU – 4 scores are considered, its performance is lower. A simple explanation is that since these two metrics rely on no. of words/characters overlap a lot more than other metrics, French and Spanish responses are able to beat English responses. So the model is not necessarily biased against English but it is perhaps slightly biased toward it to a degree as evident from the better METEOR, TER and ROUGE – L scores and these metrics check for the response quality against the references in a variety of conclusive ways such as using Precision, Recall, Synonymy, Stemming, Alignment, Edit-Rate, and Longest Common Subsequence Overlap.

4.3.4. Interpretation of Interface Testing Results

This section contains the interpretation of all the results from 4.2.4. Interface Testing.

4.3.4.1. Analysis of Results from Testing Inputs

4.3.4.1.1. Text-Box

Interpretation of figure 46:

The text-box can be seen to work, highlighted in blue, taking text input from the user.

4.3.4.1.2. Submit Button

Interpretation of figure 47:

The submit button takes a user instruction from the text-box and when pressed, returns the input text and the model response in the chat window.

4.3.4.1.3. Clear Chat Button

Interpretation of figure 48:

The clear chat button when pressed clears the chat window and resets the conversation to a new one.

4.3.4.2. Testing Conversation Window

4.3.4.2.1. English Interactions

Interpretation of figures 49 & 50:

The English instructions are sent to the model by the interface, processed by the model, which generates a response for them, and the responses are sent to the conversation window where they get displayed. The displayed responses are formatted appropriately and contextually fits as the answer to the instructions given.

4.3.4.2.2. French Interactions

Interpretation of figures 51 & 52:

The French instructions are sent to the model by the interface, processed by the model, which generates a response for them, and the responses are sent to the conversation window where they get displayed. The displayed responses are formatted appropriately and contextually fits as the answer to the instructions given.

4.3.4.2.3. Spanish Interactions

Interpretation of figures 53 & 54:

The Spanish instructions are sent to the model by the interface, processed by the model, which generates a response for them, and the responses are sent to the conversation window where they get displayed. The displayed responses are formatted appropriately and contextually fits as the answer to the instructions given.

5. CONCLUSIONS

5.1 Summary

This project aimed to develop an efficient & seamless Multilingual Single-turn Customer Service Chatbot that can improve customer service experience for English, French, & Spanish speakers by providing context-rich and semantically coherent responses to their customer service queries. The Bitext Sample Customer Support Training Dataset was pre-processed and translated into French and Spanish using Marian-MT models, the created Multilingual Customer service dataset was visualized and analyzed, on which, an mBART model was finetuned to create the Multilingual Customer Service Chatbot, which was deployed using Gradio.

The fine-tuned Chatbot model was evaluated thoroughly by way of immediate sanity tests, overall model performance, and language-specific model performance using seven different metrics such as BLEU, ROUGE, METEOR, TER, ChrF, COMET, & Latency Tests and their sub-types. The model passed the sanity tests by generating grammatically sound and contextually appropriate responses for a custom instruction and a few random instructions from the dataset, which were evaluated using Grammar Checker tools for each language available online. The model passed the overall performance evaluation as the responses generated by the model for 20% of the dataset's instructions satisfied the industry criteria of 5 out of 7 metrics, namely BLEU, ROUGE, METEOR, ChrF, and Response times to be considered a near-state-of-the-art NLP model. The model passed the language-specific performance evaluation as the responses generated by the model for 20% of each language's instructions from the dataset satisfied the industry criteria of 4 out of 7 metrics, namely BLEU, METEOR, Latency Tests, and ChrF to be considered a near-state-of-the-art NLP model.

5.2 Contributions

This project contributes to the field of Multilingual Natural Language Processing (NLP) and multilingual customer service chatbot development in a variety of ways.

Multilingual Model Performance by one single model has been enhanced dramatically in this project as near-SOTA scores for a number of metrics have been achieved in both overall and language-specific performance evaluation, especially for metrics such as BLEU, METEOR & ChrF indicating that not only is the model able to capture the structure of each language effectively but also the contextual information of them as well, showing that a single mBART model can achieve consistent quality across different languages like English, French, and Spanish, with relatively minor variation in performance metrics, demonstrating the power of Crosslingual Transfer Learning.

The project presents a chatbot model that achieves Response times that are equivalent to a chatbot deployed in production, with an average response time of 1.22 seconds, indicating its Responsiveness, Efficacy, and its potential for Real-World Usability.

The fine-tuning of mBART on a domain-specific dataset such as the Multilingual Customer Service Dataset and the achievement of a near-SOTA model demonstrates the feasibility of successful domain adaptation by LLM models without the need for retraining, cementing the possibility of the creation of efficient and robust chatbots in multiple different industries.

Contribution to the evaluation methodologies in the field of Natural Language Processing is another major contribution of this project as it employs a number of metrics such as BLEU, ROUGE, METEOR, TER, ChrF, and COMET, using which it evaluates the model's performance comprehensively and across the languages, providing a framework or benchmark methodology that other aspiring researchers in the field of Multilingual NLP can use for their own evaluations.

Finally, this project overcomes the proposed research gaps from papers – [11] by building a robust Multilingual Customer Service Chatbot that performs very well for multiple languages, made possible by utilizing Cross-lingual training, [12] by fine-tuning the mBART model on domain-specific data to create a Chatbot that generates high quality responses to customer service queries, rich with context and relevant information, and [13] by creation of a chatbot that generates culturally sensitive responses in English, French, & Spanish as verified by using language-specific grammar checkers.

5.3 Future Work

Currently, the Chatbot supports three languages, English, French and Spanish. More languages could be added to its repertoire by using appropriate MarianMT models to translate our English data into data in those languages. mBART's transfer learning capability will allow it to learn to respond in many more languages than just these three.

Some of the metrics in the language-specific evaluation such as ROUGE & METEOR indicated that the model is biased towards English when it comes to word overlap and response quality, while the TER metric indicated that the model is biased against Spanish in that it's ability to capture the reference text in the responses is the lowest for Spanish, and surprisingly, COMET metric indicated that the model is biased toward Spanish meaning that Spanish responses score the highest when a human evaluation is simulated. These results indicate that the model could use some more fine-tuning and bias mitigation could be a focus-point in the future.

The chatbot built in this project can only work in a Single-turn fashion. Multi-turn customer service chatbots are not produced in the same rate as Single-turn chatbots. A worthy endeavor would be to extend this model to have multi-turn capabilities such as remembering conversation context and back-and-forth interactions with the user.

The project's main goal was to build a chatbot for Customer Service. However, future work could explore expanding it to other domains such as finance, healthcare, and so on. By finetuning the mBART model on curated datasets of a particular domain, we can adapt the chatbot to that domain.

The project currently substitutes the entities in the dataset with sample values before translation and training. The chatbot uses these sample values in its responses. While our current implementation shows us the capabilities of the chatbot, it could certainly benefit from the ability to integrate dynamic values personalized to each user in its responses. This could be done by fetching a user's relevant data from their account in any given online store where the chatbot is integrated and use those values in the responses.

REFERENCES

[1] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... & Stoyanov, V. (2020). mBART: Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, *8*, 730-745.

[2] Conneau, A., Khandelwal, K., Goyal, N., Chaudhary, V., Wenzek, G., Guzmán, F., ... & Stoyanov, V. (2020). Unsupervised cross-lingual representation learning at scale. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 8440-8451.

[3] Howard, J., & Ruder, S. (2018). Universal Language Model Fine-tuning for Text Classification. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 328-339.

[4] Ruder, S. (2019). A Survey of Cross-Lingual Word Embedding Models. *Journal of Artificial Intelligence Research*, *65*, 569-631.

[5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*.

[6] Suraj. (2023). MarianMT - Know, Train, Infer. *Kaggle*.

[7] Junczys-Dowmunt, M., Grundkiewicz, R., Dwojak, T., Hoang, H., Heafield, K., Neckermann, T., ... & Birch, A. (2018). Marian: Fast neural machine translation in C++. arXiv preprint arXiv:1804.00344.

[8] Singh, S., & Mahmood, A. (2021). The NLP cookbook: modern recipes for transformer based deep learning architectures. *IEEE Access*, *9*, 68675-68702.

[9] Liu, Y. (2020). Multilingual denoising pre-training for neural machine translation. *arXiv preprint arXiv:2001.08210*.

[10] Popel, M., & Bojar, O. (2018). Training tips for the transformer model. *arXiv preprint arXiv:1804.00247*.

[11] Aggarwal, S., Mehra, S., & Mitra, P. (2023). Multi-Purpose NLP Chatbot: Design, Methodology & Conclusion. arXiv preprint arXiv:2310.08977.

[12] Kumar, S., Gupta, R., & Sharma, V. (2022). Natural Language Processing: State of the Art, Current Trends and Challenges. Journal of Artificial Intelligence Research, 65, 123-145.

[13] diagrams.net. (n.d.). *diagrams.net*. https://app.diagrams.net