Natural Language Processing

HomeWork 4

Ilia Hashemi Rad

99102456

AmirMohammad Fakhimi

99170531

AmirMahdi Namjoo

402211467

Introduction

Explanations:

In this project, we have implemented a Transformer model that tries to identify and neutralize gender bias in texts. To achieve this, we first gathered a dataset consisting of Persian tweets with gender bias and some ChatGPT responses with gender bias. We then investigated them manually. We then used LLMs to neutralize gender bias in these texts. We then fine-tuned an LLM on these data and evaluated the results in each part.

We should also mention that we used "Identifying and Neutralizing Gender Bias from Text" from Stanford University to get some ideas for the implementation. Notably, the article is based on the English language, and we did not have access to its dataset or code.

It is noteworthy that a dataset called "EXIST" exists for gender bias in the English and Spanish language, but we could not get it as it needs filling out a form to get authorized for access.

Collecting and Preparing the Dataset

Explanations:

As there is no Persian dataset for the gender bias problem, we gathered the dataset by scrapping Twitter using "selenium-twitter-scraper". The amount of data collected from Twitter was not enough to get appropriate results, so we asked ChatGPT to generate another set of data with and without gender bias to increase the size of our dataset significantly.

After getting the data, we processed them to prepare them for use in the models in the next section of this assignment.

Note: Some parts of the explanations of code from here to the end of the document are written by partially ChatGPT to help us write clear and unambiguous descriptions.

```python
try:
    os.makedirs('data/tweets')
except FileExistsError:
    print('Directory already exists')
os.chdir('selenium-twitter-scraper')

base_command = f'python scraper --user=@{config["TWITTER_USERNAME"]} --password={config["
    TWITTER_PASSWORD"]} --top '

os.system(base_command + '--tweets 150 --query "***"')
os.system(base_command + '--tweets 150 --query "***"')
os.system(base_command + '--tweets 150 --query "***"')
os.system(base_command + '--tweets 150 --query "***"')
os.system(base_command + '--tweets 150 --query "***"')

os.system('cp -r tweets ../data')
os.chdir('..')
directory = 'data/tweets'
tweets_files = os.listdir(directory)
tweets_files = [tweets_file for tweets_file in tweets_files if tweets_file.startswith('2024')]
tweets_files = sorted(tweets_files)
tweets = []

for tweets_file in tweets_files:
    current_file = pd.read_csv(f'{directory}/{tweets_file}')
    current_file = current_file['Content'].tolist()
    tweets += current_file


len(tweets), tweets
```

Explanations:

The main part of scrapping Twitter is using the Python scraper command with different queries. Note that due to LaTeXlimitations in code, we could not put Persian text in the code section above, but the queries are Persian equivalents of "Feminism", "Only Boy", "Only Girl", "Only Man", "Only Woman". After that, we get only those that are from 2024 and store them.

```python
directory = 'data/chatgpt'

biased_files = [
    'Bias1.txt',
    'Bias2.txt',
    'Bias3.txt',
    'Bias4.txt',
    'Bias5.txt',
]

unbiased_files = [
    'NoBias1.txt',
    'NoBias2.txt',
    'NoBias3.txt',
    'NoBias4.txt',
   # 'NoBias5.txt',
    'NoBias6.txt',
]

biased_chatgpt = []
unbiased_chatgpt = []

for file in biased_files:
    with open(f'{directory}/{file}', 'r', encoding='utf-8') as file:
        current_data = file.read().split('\n\n')
        biased_chatgpt += current_data

for file in unbiased_files:
    with open(f'{directory}/{file}', 'r', encoding='utf-8') as file:
        current_data = file.read().split('\n\n')
        unbiased_chatgpt += current_data

len(biased_chatgpt), len(unbiased_chatgpt)
```

Explanations:

The script reads biased and unbiased data files:

- Defines lists of file names for biased and unbiased data.
- Reads and splits the content of each file into lists.

```python
data = []

with open('data/tweets/tweets_all.txt', 'r', encoding='utf-8') as file:
    labeled_tweets = file.read().split('\n\n')

len(labeled_tweets)

for i in range(0, len(labeled_tweets), 2):
    tweet, label = labeled_tweets[i], labeled_tweets[i + 1]
    data.append({
        "id": len(data),
        "dialogue": tweet,
        "summary": label,
    })

for biased_chatgpt_data in biased_chatgpt:
    prompt, label = biased_chatgpt_data.split('\n')
    prompt = prompt.split(': ')[1]
    label = label.split(': ')[1]
    data.append({
        "id": len(data),
        "dialogue": prompt,
        "summary": label,
    })

for unbiased_chatgpt_data in unbiased_chatgpt:
    prompt = unbiased_chatgpt_data
    prompt = prompt.split(': ')[1]
    data.append({
        "id": len(data),
        "dialogue": prompt,
        "summary": prompt,
    })
```

Explanations:

Finally, the script combines labeled tweet data and ChatGPT data into a single list of dictionaries:

- Reads labeled tweets from a file.
- Appends tweet and label pairs to the data list.
- Processes and appends biased and unbiased ChatGPT data.

```python
with open('stopwords/chars.txt', 'r', encoding='utf-8') as file:
    chars_stop_words = ''.join(file.read().splitlines())

chars_stop_words = chars_stop_words.replace('[', '\[')
chars_stop_words = chars_stop_words.replace(']', '\]')
chars_stop_words += '\t\r'
chars_pattern = re.compile(f'[{chars_stop_words}]')
chars_pattern
```

Explanations:

The script reads stopwords from a file and prepares a regex pattern to remove them:

- Reads stopwords from `chars.txt` file.
- Modifies stopwords string to escape special characters.
- Compiles a regex pattern to match these stopwords.

```python
# https://stackoverflow.com/questions/33404752/removing-emojis-from-a-string-in-python
emojis_pattern = re.compile("["
                            u"\U0001F600-\U0001F64F"  # emoticons
                            u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                            u"\U0001F680-\U0001F6FF"  # transport & map symbols
                            u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                            "]+")
emojis_pattern
```

Explanations:

The script prepares a regex pattern to remove emojis, referencing a solution from Stack Overflow.

```python
normalizer = Normalizer()
```

Explanations:

The script initializes a `Normalizer` from the `hazm` library for text normalization.

```
    def remove_duplicates(data):
        data_df: pd.DataFrame = pd.DataFrame(data)
        data_df = data_df.drop_duplicates(subset=['dialogue'])

        data = []
        for index, row in data_df.iterrows():
            data.append({
                "id": row['id'],
                "dialogue": row['dialogue'],
                "summary": row['summary'],
            })

        return data
```

Explanations:

The `remove_duplicates` function removes duplicate entries from the data based on the `dialogue` field.

```
    def preprocess_text(text):
        global chars_pattern, emojis_pattern, normalizer

        text = emojis_pattern.sub(r' ', text)
        text = chars_pattern.sub(r' ', text)
        text = normalizer.normalize(text)

        return text
```

Explanations:

The `preprocess_text` function applies regex patterns to remove emojis and stopwords and normalizes the text.

```
    len_before = len(data)
    data = remove_duplicates(data)
    len_after = len(data)

    len_before, len_after
```

Explanations:

The script removes duplicates from the data and prints the length of the data before and after the removal.

```
preprocessed_data = []

progress_bar = tqdm(data)
for datum in progress_bar:
    preprocessed_data.append({
        "id": datum["id"],
        "dialogue": preprocess_text(datum["dialogue"]),
        "summary": preprocess_text(datum["summary"]),
    })
```

Explanations:

Finally, the script preprocesses the text data using the `preprocess_text` function and stores the results in `preprocessed_data`.

```python
import py7zr
import json
from sklearn.model_selection import train_test_split
import random

NUMBER_TO_SHUFFLE = 4
random.seed(42)

for _ in range(NUMBER_TO_SHUFFLE):
    random.shuffle(preprocessed_data)
    print(preprocessed_data[0])

train_data, test_data = train_test_split(preprocessed_data, test_size=0.1, random_state=42)
len(train_data), len(test_data)

train_data = sorted(train_data, key=lambda x: x['id'])
test_data = sorted(test_data, key=lambda x: x['id'])
```

Explanations:

This segment of the script performs the following operations:

- `import py7zr`: Imports the `py7zr` library for handling 7z archive files.
- `import json`: Imports the `json` library for handling JSON data.
- `from sklearn.model_selection import train_test_split`: Imports the `train_test_split` function from `sklearn` for splitting data into training and testing sets.
- `import random`: Imports the `random` library for random operations.

The script then sets up the shuffling and splitting of preprocessed data:

- `NUMBER_TO_SHUFFLE = 4`: Defines the number of times the data will be shuffled.
- `random.seed(42)`: Sets the random seed for reproducibility.
- A `for` loop shuffles the `preprocessed_data` list `NUMBER_TO_SHUFFLE` times and prints the first element after each shuffle.
- `train_data, test_data = train_test_split(preprocessed_data, test_size=0.1, random_state=42)`: Splits the data into training and testing sets, with 10% of the data reserved for testing.
- `train_data = sorted(train_data, key=lambda x: x['id'])`: Sorts the training data by the `id` field.
- `test_data = sorted(test_data, key=lambda x: x['id'])`: Sorts the testing data by the `id` field.

```python
directory = 'gender_neutralize/data'

os.makedirs(directory, exist_ok=True)

with open(f'{directory}/train.json', 'w', encoding='utf-8') as file:
    json.dump(train_data, file, ensure_ascii=False, indent=2)

with open(f'{directory}/test.json', 'w', encoding='utf-8') as file:
    json.dump(test_data, file, ensure_ascii=False, indent=2)

os.chdir(directory)

with py7zr.SevenZipFile('gender_neutralize.7z', 'w') as archive:
    archive.write('train.json')
    archive.write('test.json')
    os.chdir('../..')
```

Explanations:

This part of the script saves and archives the processed data:

- `directory = 'gender_neutralize/data'`: Specifies the directory to store the data.
- `os.makedirs(directory, exist_ok=True)`: Creates the directory if it does not already exist.
- The script then opens two files, `train.json` and `test.json`, in write mode and dumps the training and testing data into these files as JSON objects.
- `os.chdir(directory)`: Changes the current working directory to the specified directory.
- `with py7zr.SevenZipFile('gender_neutralize.7z', 'w') as archive`: Creates a new 7z archive file named `gender_neutralize.7z` in write mode.
- `archive.write('train.json')`: Adds the `train.json` file to the archive.
- `archive.write('test.json')`: Adds the `test.json` file to the archive.
- `os.chdir('../..')`: Changes the current working directory back to the original directory.

Fine Tuning

In this section, we will explain the code for using LLaMa 3 and fine-tuning it for the neutralization of gender bias.

We used LLaMa 3 with 8Bilion Parameters. There is also a 70 Billion Parameter version, but we removed it from our options as it needs more resources. We studied some of its tutorials that fine-tuned LLaMa 3 on a summarization task, and we used the ideas and code for our task, although with some changes. We used the summarizer structure as its input-output format, much like our task format; the summarizer task gets an input text and outputs it as a summary, and in our task, we have an input text and need the gender-bias-neutralized text as the output.

The code fine-tunes our LLaMa3 using our dataset gathered in the previous parts using LoRA. An important note here is that this task needs more than 16 GB of GPU Memory, and we did not have access to this in Colab. Fortunately, the Kaggle enabled us to run our model on two T4 accelerators in parallel, solving our problem of limited resources on Colab. Also, we changed the context-length parameter from 4096 to 1024 to mitigate the lack of resources.

Also, after fine-tuning, we saved the model, emptied the GPU memory, and loaded the model again. This helped us mitigate some problems regarding insufficient memory.

Now, let's get to the code.

Step 0

The following code installs necessary Python packages and logs into the Hugging Face Hub.

```
! pip install --upgrade pip
! pip install evaluate
! pip install bert_score
! pip install hazm
! pip install llama-recipes ipywidgets

import huggingface_hub
huggingface_hub.login()
```

Step 1

```python
import torch
from transformers import LlamaForCausalLM, AutoTokenizer
from llama_recipes.configs import train_config as TRAIN_CONFIG

train_config = TRAIN_CONFIG()
train_config.model_name = "meta-llama/Meta-Llama-3-8B"
train_config.num_epochs = 3
train_config.run_validation = False
train_config.gradient_accumulation_steps = 4
train_config.batch_size_training = 1
train_config.lr = 2e-4
train_config.use_fast_kernels = True
train_config.use_fp16 = True
train_config.context_length = 1024 if torch.cuda.get_device_properties(0).total_memory < 16e9
    else 2048 # T4 16GB or A10 24GB
train_config.batching_strategy = "packing"
train_config.output_dir = "meta-llama-sexbias"

from transformers import BitsAndBytesConfig
config = BitsAndBytesConfig(
    load_in_8bit=True,
)

model = LlamaForCausalLM.from_pretrained(
            train_config.model_name,
            device_map="auto",
            quantization_config=config,
            use_cache=False,
            attn_implementation="sdpa" if train_config.use_fast_kernels else None,
            torch_dtype=torch.float16,
        )

tokenizer = AutoTokenizer.from_pretrained(train_config.model_name)
tokenizer.pad_token = tokenizer.eos_token
```

Explanations:

First, import the necessary libraries. `torch` is used for tensor computations, `LlamaForCausalLM` and `AutoTokenizer` are used for model and tokenizer management from the `transformers` library, and `train_config` is imported from `llama_recipes.configs`.

```python
import torch
from transformers import LlamaForCausalLM, AutoTokenizer
from llama_recipes.configs import train_config as TRAIN_CONFIG
```

Explanations:

Next, create a training configuration object and set various training parameters. These include the model name, number of epochs, validation settings, gradient accumulation steps, batch size for training, learning rate, kernel usage, floating point precision, context length based on available GPU memory, batching strategy, and output directory.

```python
train_config = TRAIN_CONFIG()
train_config.model_name = "meta-llama/Meta-Llama-3-8B"
train_config.num_epochs = 3
train_config.run_validation = False
train_config.gradient_accumulation_steps = 4
train_config.batch_size_training = 1
train_config.lr = 2e-4
train_config.use_fast_kernels = True
train_config.use_fp16 = True
train_config.context_length = 1024 if torch.cuda.get_device_properties(0).total_memory
    < 16e9 else 2048
```

```
        train_config.batching_strategy = "packing"
        train_config.output_dir = "meta-llama-sexbias"
```

Explanations:

Import the `BitsAndBytesConfig` class from the `transformers` library and create a configuration object for loading the model in 8-bit precision.

```
        from transformers import BitsAndBytesConfig
        config = BitsAndBytesConfig(
            load_in_8bit=True,
        )
```

Explanations:

Load the LLaMA model for causal language modeling using the specified configuration. The model is loaded with the training configuration parameters, including device mapping, quantization settings, cache usage, attention implementation, and data type.

```
        model = LlamaForCausalLM.from_pretrained(
            train_config.model_name,
            device_map="auto",
            quantization_config=config,
            use_cache=False,
            attn_implementation="sdpa" if train_config.use_fast_kernels else None,
            torch_dtype=torch.float16,
        )
```

Explanations:

Load the tokenizer associated with the LLaMA model and set the padding token to the end-of-sequence token.

```
        tokenizer = AutoTokenizer.from_pretrained(train_config.model_name)
        tokenizer.pad_token = tokenizer.eos_token
```

Step 2

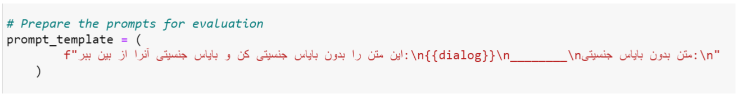
The following code prepares a dataset, generates predictions using a model, computes evaluation metrics, and performs a sample evaluation.

As the Prompt template is a Persian text that could not be adequately rendered in the LaTeXlisting environment, we have included its picture below and put *** in the code on this document. It can be appropriately viewed in the Jupyter Notebook. The best way is to ask, "Remove this text gender bias and make it without gender bias," then add the text and put "text without gender bias:" with eight underlines in Persian.

It is also noteworthy that we used the Hazm tokenizer. It produces better results than not using a tokenizer at all. The base code of Facebook examples uses the nltk tokenizer (for English).

For the Bert-Score calculation, we used Microsoft `deberta-v2-xxlarge-mnli`.

Figure 1: Prompt Template



```
# Prepare the prompts for evaluation
prompt_template = (
        f"این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:\n{{dialog}}\n________\nمتن بدون بایاس جنسیتی:\n"
    )
```

```
    import datasets

    # Load the test dataset
    test_dataset = datasets.load_dataset("AmirMohammadFakhimi/gender_neutralize", split="test")
```

```python

    # Prepare the prompts for evaluation
    prompt_template = (
            f"***\n"
        )

    def prepare_prompt(sample):
        return {
            "prompt": prompt_template.format(dialog=sample["dialogue"]),
            "summary": sample["summary"]
        }

    test_dataset = test_dataset.map(prepare_prompt, remove_columns=list(test_dataset.features))
```

Explanations:

First, import the necessary library for handling datasets.

```python
        import datasets
```

Explanations:

Load the test dataset from the Hugging Face dataset repository.

```python
        test_dataset = datasets.load_dataset("AmirMohammadFakhimi/gender_neutralize", split="
            test")
```

Explanations:

Prepare a prompt template for evaluating the text generation. The template asks to neutralize gender bias in the provided dialogue.

```python
        prompt_template = (
        ***"
        )
```

Explanations:

Define a function to prepare prompts for each sample in the dataset.

```python
        def prepare_prompt(sample):
        return {
            "prompt": prompt_template.format(dialog=sample["dialogue"]),
            "summary": sample["summary"]
        }
```

Explanations:

Apply the ‘prepare_prompt‘ function to the dataset and remove original columns.

```python
        test_dataset = test_dataset.map(prepare_prompt, remove_columns=list(test_dataset.
            features))
```

```python
    import torch

    def generate_predictions(model, tokenizer, dataset):
        predictions = []
        references = []

        model.eval()
        with torch.no_grad():
            for sample in dataset:
                model_input = tokenizer(sample["prompt"], return_tensors="pt").to("cuda")
                output = model.generate(**model_input, max_new_tokens=100)
                prediction = tokenizer.decode(output[0], skip_special_tokens=True)

                # Extract the part after "***"
                prediction = prediction.split('***\n')[1].strip()
                prediction = re.sub('***', ' '  , prediction)
                # Remove non-Alphabetical characters and digits execpt dot
                prediction = re.sub(r'[^.-A-Za-z\s]', ' ', prediction)
                # Define a regular expression pattern that matches one or more spaces
                pattern = re.compile(r" +")
                # Apply the pattern to the text and replace the matches with a single space
                prediction = pattern.sub(" ", prediction)

                predictions.append(prediction)
                references.append(sample["summary"])

        return predictions, references

    # Generate predictions for base model
    base_predictions, references = generate_predictions(model, tokenizer, test_dataset)
```

Explanations:

Import the `torch` library for tensor computations.

```python
        import torch
```

Explanations:

Define a function to generate predictions using the model and tokenizer. The function processes each sample in the dataset, generates predictions, and cleans the generated text.

```python
        def generate_predictions(model, tokenizer, dataset):
        predictions = []
        references = []

        model.eval()
        with torch.no_grad():
            for sample in dataset:
                model_input = tokenizer(sample["prompt"], return_tensors="pt").to("cuda")
                output = model.generate(**model_input, max_new_tokens=100)
                prediction = tokenizer.decode(output[0], skip_special_tokens=True)

                # Extract the part after "***"
                prediction = prediction.split('***\n')[1].strip()
                prediction = re.sub('***', ' '  , prediction)
                # Remove non-Alphabetical characters and digits execpt dot
                prediction = re.sub(r'[^.-A-Za-z\s]', ' ', prediction)
                # Define a regular expression pattern that matches one or more spaces
                pattern = re.compile(r" +")
                # Apply the pattern to the text and replace the matches with a single space
                prediction = pattern.sub(" ", prediction)

                predictions.append(prediction)
                references.append(sample["summary"])

        return predictions, references
```

Explanations:

Generate predictions for the base model using the defined function.

```python
        base_predictions, references = generate_predictions(model, tokenizer, test_dataset)
```

```python
from evaluate import load
from hazm import word_tokenize

bertscore_metric = load("bertscore")
bleu_metric = load('bleu')

def compute_metrics(predictions, references):
    # fix the references with the BLEU score HuggingFace format
    references = [[ref] for ref in references]

    # Compute BLEU score
    bleu_score = bleu_metric.compute(predictions=predictions, references=references, tokenizer=
        word_tokenize)#["bleu"]

    # Compute BERTScore
    bertscore = bertscore_metric.compute(predictions=predictions, references=references, lang="
        fa", model_type='microsoft/deberta-v2-xxlarge-mnli')
    bertscore_f1 = sum(bertscore["f1"]) / len(bertscore["f1"])

    return bleu_score, bertscore_f1

# Compute metrics for both models
base_bleu, base_bertscore = compute_metrics(base_predictions, references)

print(f"Base Model - BLEU: {base_bleu}, BERTScore F1: {base_bertscore}")
```

Explanations:

Import the ‘evaluate‘ library for computing evaluation metrics and ‘hazm‘ for tokenizing Persian text.

```python
from evaluate import load
from hazm import word_tokenize
```

Explanations:

Load the BERTScore and BLEU metrics.

```python
bertscore_metric = load("bertscore")
bleu_metric = load('bleu')
```

Explanations:

Define a function to compute evaluation metrics for the predictions. The function computes the BLEU score and the BERTScore F1 metric.

```python
def compute_metrics(predictions, references):
    # fix the references with the BLEU score HuggingFace format
    references = [[ref] for ref in references]

    # Compute BLEU score
    bleu_score = bleu_metric.compute(predictions=predictions, references=references,
        tokenizer=word_tokenize)

    # Compute BERTScore
    bertscore = bertscore_metric.compute(predictions=predictions, references=
        references, lang="fa", model_type='microsoft/deberta-v2-xxlarge-mnli')
    bertscore_f1 = sum(bertscore["f1"]) / len(bertscore["f1"])

    return bleu_score, bertscore_f1
```

Explanations:

Compute evaluation metrics for the base model's predictions and print the results.

```
base_bleu, base_bertscore = compute_metrics(base_predictions, references)

print(f"Base Model - BLEU: {base_bleu}, BERTScore F1: {base_bertscore}")
```

Step 3

Part of this code is taken from The LLaMa 3 summarization example using the samsum dataset. We changed it significantly to use our dataset and preprocess our data to be usable on LLaMa 3.

It is noteworthy that we used "packing" for batching strategy instead of "padding". It is better for data with a size of more than 1000.

Explanations:

The ‘dataclass‘ decorator is used to create a simple class for the dataset configuration, containing the dataset name and the train and test split names.

```
from dataclasses import dataclass

@dataclass
class samsum_dataset:
    dataset: str = "samsum_dataset"
    train_split: str = "train"
    test_split: str = "validation"
```

Explanations:

The ‘get_preprocessed_samsum‘ function loads the dataset, applies a prompt template, tokenizes the text, and prepares the data for model training.

First, it loads the dataset using the ‘datasets.load_dataset‘ method and specifies the split (train or test) to be loaded.

```
import copy
import datasets

def get_preprocessed_samsum(dataset_config, tokenizer, split):
    dataset = datasets.load_dataset("AmirMohammadFakhimi/gender_neutralize", split=
        split)
```

Explanations:

A prompt template is defined to instruct the model on the task, which is to remove gender bias from the provided dialogue.

```
prompt = (
f"***"
)
```

Explanations:

The ‘apply_prompt_template‘ function is applied to each sample in the dataset to create the prompt and summary pairs.

```
def apply_prompt_template(sample):
    return {
        "prompt": prompt.format(dialog=sample["dialogue"]),
        "summary": sample["summary"],
    }
```

```
        dataset = dataset.map(apply_prompt_template, remove_columns=list(dataset.features))
```

Explanations:

The ‘tokenize_add_label‘ function tokenizes the prompt and summary, concatenates them, and prepares the input IDs, attention mask, and labels for training.

```
        def tokenize_add_label(sample):
            prompt = tokenizer.encode(tokenizer.bos_token + sample["prompt"],
                add_special_tokens=False)
            summary = tokenizer.encode(sample["summary"] + tokenizer.eos_token,
                add_special_tokens=False)

            sample = {
                "input_ids": prompt + summary,
                "attention_mask": [1] * (len(prompt) + len(summary)),
                "labels": [-100] * len(prompt) + summary,
            }

            return sample

        dataset = dataset.map(tokenize_add_label, remove_columns=list(dataset.features))
```

Explanations:

The ‘get_preprocessed_dataset‘ function is a wrapper around ‘get_preprocessed_samsum‘ that selects the appropriate split (train or test) based on the provided configuration and returns the preprocessed dataset.

```
        def get_preprocessed_dataset(
            tokenizer, dataset_config, split: str = "train"
        ) -> torch.utils.data.Dataset:

            def get_split():
                return (
                    dataset_config.train_split
                    if split == "train"
                    else dataset_config.test_split
                )

            return get_preprocessed_samsum(
                dataset_config,
                tokenizer,
                get_split(),
            )
```

Explanations:

The ‘ConcatDataset‘ and ‘get_dataloader_kwargs‘ functions are imported to concatenate the dataset if necessary and to get the keyword arguments for the data loader.

```
        from llama_recipes.data.concatenator import ConcatDataset
        from llama_recipes.utils.config_utils import get_dataloader_kwargs
```

Explanations:

The preprocessed training dataset is obtained using the ‘get_preprocessed_dataset‘ function.

```
        train_dataset = get_preprocessed_dataset(tokenizer, samsum_dataset, 'train')
```

Explanations:

The keyword arguments for the data loader are obtained using the ‘get_dataloader_kwargs‘ function.

```
        train_dl_kwargs = get_dataloader_kwargs(train_config, train_dataset, tokenizer, "train
            ")
```

Explanations:

If the batching strategy is set to "packing," the dataset is concatenated into chunks of the specified context length.

```
        if train_config.batching_strategy == "packing":
        train_dataset = ConcatDataset(train_dataset, chunk_size=train_config.context_length)
```

Explanations:

Finally, a PyTorch DataLoader is created for the training dataset with the specified number of workers and other configurations.

```
        train_dataloader = torch.utils.data.DataLoader(
            train_dataset,
            num_workers=train_config.num_workers_dataloader,
            pin_memory=True,
            **train_dl_kwargs,
        )
```

Step 4

Preparing mode for PEFT.

```
    from peft import get_peft_model, prepare_model_for_kbit_training, LoraConfig
    from dataclasses import asdict
    from llama_recipes.configs import lora_config as LORA_CONFIG

    lora_config = LORA_CONFIG()
    lora_config.r = 8
    lora_config.lora_alpha = 32
    lora_dropout: float=0.01

    peft_config = LoraConfig(**asdict(lora_config))

    model = prepare_model_for_kbit_training(model)
    model = get_peft_model(model, peft_config)
```

Explanations:

The ‘peft‘ library is imported, which includes functions for getting a PEFT (Parameter Efficient Fine-Tuning) model, preparing a model for k-bit training, and configuring LoRA (Low-Rank Adaptation) settings.

```
        from peft import get_peft_model, prepare_model_for_kbit_training, LoraConfig
        from dataclasses import asdict
        from llama_recipes.configs import lora_config as LORA_CONFIG
```

Explanations:

The LoRA configuration is loaded and modified. Here, ‘r‘ is the rank of the decomposition, ‘lora_alpha‘ is the scaling factor, and ‘lora_dropout‘ is the dropout rate.

```
        lora_config = LORA_CONFIG()
        lora_config.r = 8
        lora_config.lora_alpha = 32
        lora_dropout: float=0.01
```

Explanations:

The LoRA configuration is converted into a dictionary and used to create a 'LoraConfig' object.

```
        peft_config = LoraConfig(**asdict(lora_config))
```

Explanations:

The model is prepared for k-bit training using the 'prepare_model_for_kbit_training' function, which optimizes the model for efficient fine-tuning. Then, the 'get_peft_model' function applies the LoRA configuration to the model.

```
        model = prepare_model_for_kbit_training(model)
        model = get_peft_model(model, peft_config)
```

Step 5

This is the fine tuning step.

```
    import torch.optim as optim
    from llama_recipes.utils.train_utils import train
    from torch.optim.lr_scheduler import StepLR

    model.train()

    optimizer = optim.AdamW(
                model.parameters(),
                lr=train_config.lr,
                weight_decay=train_config.weight_decay,
            )
    scheduler = StepLR(optimizer, step_size=1, gamma=train_config.gamma)

    # Start the training process
    results = train(
        model,
        train_dataloader,
        None,
        tokenizer,
        optimizer,
        scheduler,
        train_config.gradient_accumulation_steps,
        train_config,
        None,
        None,
        None,
        wandb_run=None,
    )
```

Explanations:

The 'torch.optim' module is imported to handle the optimization process. The 'train' function and 'StepLR' scheduler are imported to manage training and learning rate scheduling, respectively.

```
        import torch.optim as optim
        from llama_recipes.utils.train_utils import train
        from torch.optim.lr_scheduler import StepLR
```

Explanations:

The model is set to training mode using ‘model.train()‘. This ensures that layers like dropout are applied appropriately during training.

```
        model.train()
```

Explanations:

An AdamW optimizer is created with parameters from the model, learning rate from the training configuration, and weight decay to regularize the model.

```
        optimizer = optim.AdamW(
                    model.parameters(),
                    lr=train_config.lr,
                    weight_decay=train_config.weight_decay,
                )
```

Explanations:

A learning rate scheduler, ‘StepLR‘, is set up to adjust the learning rate after every epoch. The ‘step_size‘ parameter defines the number of epochs after which the learning rate is updated, and ‘gamma‘ is the multiplicative factor for learning rate decay.

```
        scheduler = StepLR(optimizer, step_size=1, gamma=train_config.gamma)
```

Explanations:

The ‘train‘ function is called to start the training process. It takes the model, data loader, tokenizer, optimizer, scheduler, gradient accumulation steps, and other training configurations as inputs. ‘wandb_run‘ is set to ‘None‘ as no Weights & Biases run is being used here.

```
        # Start the training process
        results = train(
        model,
        train_dataloader,
        None,
        tokenizer,
        optimizer,
        scheduler,
        train_config.gradient_accumulation_steps,
        train_config,
        None,
        None,
        None,
        wandb_run=None,
        )
```

Step 6

```
    model.save_pretrained(train_config.output_dir)
```

Explanations:

The trained model is saved to the directory specified in the training configuration’s ‘output_dir‘. This allows the model to be reloaded and used later without retraining.

```
        model.save_pretrained(train_config.output_dir)
```

```
from transformers import LlamaForCausalLM, AutoTokenizer, BitsAndBytesConfig
from peft import PeftModel, PeftConfig
import torch

# Load the base model and tokenizer
base_model_name = "meta-llama/Meta-Llama-3-8B"
tokenizer = AutoTokenizer.from_pretrained(base_model_name)
tokenizer.pad_token = tokenizer.eos_token

# Use mixed precision and quantization to save memory
config = BitsAndBytesConfig(
    load_in_8bit=True,
    llm_int8_enable_fp32_cpu_offload=True  # Enable CPU offload to save GPU memory
)

base_model = LlamaForCausalLM.from_pretrained(
    base_model_name,
    device_map="auto",
    quantization_config=config,
    torch_dtype=torch.float16
)

# Load the adapter configuration
peft_model_id = '/kaggle/input/meta-llama-sexbias/pytorch/default/1'
adapter_config = PeftConfig.from_pretrained(peft_model_id)

# Load the LoRA adapter
model = PeftModel.from_pretrained(base_model, peft_model_id, config=adapter_config)
```

Explanations:

The necessary libraries for loading and configuring the model for inference are imported. These include ‘LlamaForCausalLM‘, ‘AutoTokenizer‘, and ‘BitsAndBytesConfig‘ from ‘transformers‘, as well as ‘PeftModel‘ and ‘PeftConfig‘ from ‘peft‘.

```
        from transformers import LlamaForCausalLM, AutoTokenizer, BitsAndBytesConfig
        from peft import PeftModel, PeftConfig
        import torch
```

Explanations:

The base model name is defined, and the tokenizer is loaded using the ‘AutoTokenizer‘ class. The end-of-sequence token is set as the padding token.

```
        base_model_name = "meta-llama/Meta-Llama-3-8B"
        tokenizer = AutoTokenizer.from_pretrained(base_model_name)
        tokenizer.pad_token = tokenizer.eos_token
```

Explanations:

A ‘BitsAndBytesConfig‘ is created to use mixed precision and quantization, which helps save memory by loading the model in 8-bit precision and enabling FP32 CPU offloading.

```
        config = BitsAndBytesConfig(
        load_in_8bit=True,
        llm_int8_enable_fp32_cpu_offload=True  # Enable CPU offload to save GPU memory
        )
```

Explanations:

The base model is loaded using the ‘LlamaForCausalLM‘ class with the defined quantization configuration and mixed precision (FP16).

```
        base_model = LlamaForCausalLM.from_pretrained(
            base_model_name,
            device_map="auto",
            quantization_config=config,
            torch_dtype=torch.float16
        )
```

Explanations:

The adapter configuration is loaded from a specified directory using the ‘PeftConfig‘ class. This configuration contains settings specific to the PEFT model.

```
        peft_model_id = '/kaggle/input/meta-llama-sexbias/pytorch/default/1'
        adapter_config = PeftConfig.from_pretrained(peft_model_id)
```

Explanations:

The LoRA adapter is loaded using the ‘PeftModel‘ class, which combines the base model with the PEFT configuration.

```
        model = PeftModel.from_pretrained(base_model, peft_model_id, config=adapter_config)
```

Step 7

```
    import re

    def generate_predictions(model, tokenizer, dataset):
        predictions = []
        references = []

        model.eval()
        with torch.no_grad():
            for sample in dataset:
                model_input = tokenizer(sample["prompt"], return_tensors="pt").to("cuda")
                output = model.generate(**model_input, max_new_tokens=100)
                prediction = tokenizer.decode(output[0], skip_special_tokens=True)

                # Extract the part after "***"
                prediction = prediction.split('***\n')[1].strip()
                prediction = re.sub('***', ' ' , prediction)
                # Remove non-Alphabetical characters and digits except dot
                prediction = re.sub(r'[^.-A-Za-z\s]', ' ', prediction)
                # Define a regular expression pattern that matches one or more spaces
                pattern = re.compile(r" +")
                # Apply the pattern to the text and replace the matches with a single space
                prediction = pattern.sub(" ", prediction)

                predictions.append(prediction)
                references.append(sample["summary"])

                # Clear CUDA cache to free up memory
                torch.cuda.empty_cache()

        return predictions, references
```

Explanations:

The ‘generate_predictions‘ function generates predictions from the model for each sample in the dataset and cleans up the text.

- The function sets the model to evaluation mode using ‘model.eval()‘ and disables gradient calculations with ‘torch.no_grad()‘.

- For each sample in the dataset, it tokenizes the input prompt and generates a prediction using the model.
- The predicted text is decoded and cleaned:
  - It extracts the relevant portion of the prediction after the marker "Text withoug gender bias" (in Persian).
  - Non-alphabetical characters and digits are removed, except for dots.
  - Multiple spaces are replaced with a single space.
- Predictions and references are collected, and CUDA memory is cleared with ‘torch.cuda.empty_cache()‘ to manage GPU resources.

```
        def generate_predictions(model, tokenizer, dataset):
            predictions = []
            references = []

            model.eval()
            with torch.no_grad():
                for sample in dataset:
                    model_input = tokenizer(sample["prompt"], return_tensors="pt").to("cuda")
                    output = model.generate(**model_input, max_new_tokens=100)
                    prediction = tokenizer.decode(output[0], skip_special_tokens=True)

                    # Extract the part after "***"
                    prediction = prediction.split('***')[1].strip()
                    prediction = re.sub('***', ' ' , prediction)
                    # Remove non-Alphabetical characters and digits except dot
                    prediction = re.sub(r'[^.-A-Za-z\s]', ' ', prediction)
                    # Define a regular expression pattern that matches one or more spaces
                    pattern = re.compile(r" +")
                    # Apply the pattern to the text and replace the matches with a single
                        space
                    prediction = pattern.sub(" ", prediction)

                    predictions.append(prediction)
                    references.append(sample["summary"])

                    # Clear CUDA cache to free up memory
                    torch.cuda.empty_cache()

            return predictions, references
```

Explanations:

Predictions and references are saved to text files for later evaluation. Each line of the prediction file corresponds to a generated output, while each line of the references file corresponds to the reference summary.

```
        with open("tuned_predictions.txt", "w") as f:
            for pred in tuned_predictions:
                f.write(str(pred) +"\n")

        with open("references.txt", "w") as f:
            for ref in references:
                f.write(str(ref) +"\n")
```

Explanations:

The saved text files are read back into lists. Each line is converted to an integer. This step may be a mistake as predictions and references are typically strings. Ensure this conversion is appropriate for your use case.

```
        references = []
        with open("references.txt", "r") as f:
          for line in f:
            references.append(int(line.strip()))

        tuned_predictions = []
        with open("tuned_predictions.txt", "r") as f:
          for line in f:
            tuned_predictions.append(int(line.strip()))
```

Explanations:

The ‘compute_metrics‘ function computes BLEU and BERTScore metrics for the predictions against the references. BLEU measures the precision of n-grams, while BERTScore evaluates the semantic similarity using BERT embeddings.

- BLEU score is calculated using the ‘bleu_metric‘ with tokenized references and predictions.
- BERTScore is computed using the ‘bertscore_metric‘ with the specified language and model.

```
        from evaluate import load
        from hazm import word_tokenize

        bertscore_metric = load("bertscore")
        bleu_metric = load('bleu')

        def compute_metrics(predictions, references):
            # fix the references with the BLEU score HuggingFace format
            references = [[ref] for ref in references]

            # Compute BLEU score
            bleu_score = bleu_metric.compute(predictions=predictions, references=references,
                tokenizer=word_tokenize)

            # Compute BERTScore
            bertscore = bertscore_metric.compute(predictions=predictions, references=
                references, lang="fa", model_type='microsoft/deberta-v2-xxlarge-mnli')
            bertscore_f1 = sum(bertscore["f1"]) / len(bertscore["f1"])

            return bleu_score, bertscore_f1
```

Explanations:

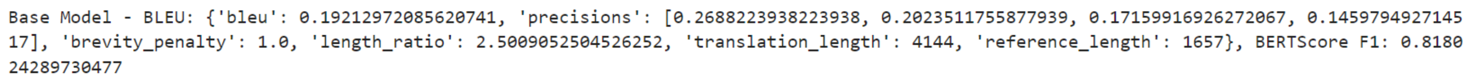
The computed BLEU and BERTScore metrics are printed. These metrics provide an evaluation of the models performance in terms of both n-gram precision and semantic similarity.

```
        # Compute metrics for fine-tuned model
        tuned_bleu, tuned_bertscore = compute_metrics(tuned_predictions, references)

        print(f"Fine-tuned Model - BLEU: {tuned_bleu}, BERTScore F1: {tuned_bertscore}")
```

Results

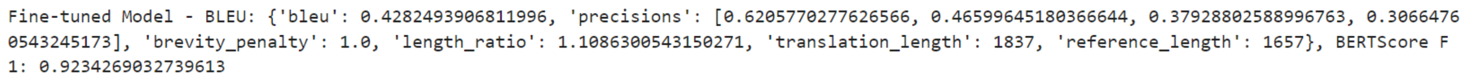
In the following section, we have included some examples of the results and the overall metrics of the model. An important note is that for some of the results, the not-fine-tunes model wanted to repeat the answer multiple times. In those cases, we added post-processing to remove the duplications of text and used it to calculate metrics.

As shown in the figures below, we saw a huge improvement in BLUE and BERT scores after fine-tuning.



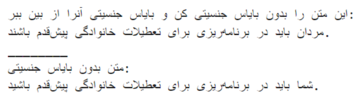
```
Base Model - BLEU: {'bleu': 0.19212972085620741, 'precisions': [0.2688223938223938, 0.2023511755877939, 0.17159916926272067, 0.1459794927145
17], 'brevity_penalty': 1.0, 'length_ratio': 2.5009052504526252, 'translation_length': 4144, 'reference_length': 1657}, BERTScore F1: 0.8180
24289730477
```

Figure 2: Base model Evaluation Metrics



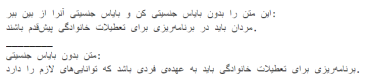
```
Fine-tuned Model - BLEU: {'bleu': 0.4282493906811996, 'precisions': [0.6205770277626566, 0.46599645180366644, 0.37928802588996763, 0.3066476
0543245173], 'brevity_penalty': 1.0, 'length_ratio': 1.1086300543150271, 'translation_length': 1837, 'reference_length': 1657}, BERTScore F
1: 0.9234269032739613
```

Figure 3: Fine-tuned model Evaluation Metrics



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
مردان باید در برنامه‌ریزی برای تعطیلات خانوادگی پیش‌قدم باشند.
________
متن بدون بایاس جنسیتی:
شما باید در برنامه‌ریزی برای تعطیلات خانوادگی پیش‌قدم باشید.
```

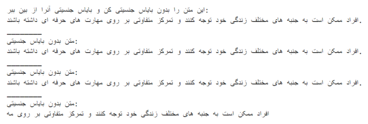
(a) Base Model



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
مردان باید در برنامه‌ریزی برای تعطیلات خانوادگی پیش‌قدم باشند.
________
متن بدون بایاس جنسیتی:
برنامه‌ریزی برای تعطیلات خانوادگی باید به عهده‌ی فردی باشد که توانایی‌های لازم را دارد.
```

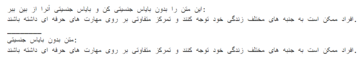
(b) Fine-Tuned Model

Figure 4: Sample Prediction 1



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
افراد ممکن است به جنبه های مختلف زندگی خود توجه کنند و تمرکز متفاوتی بر روی مهارت های حرفه ای داشته باشند.
________
متن بدون بایاس جنسیتی:
افراد ممکن است به جنبه های مختلف زندگی خود توجه کنند و تمرکز متفاوتی بر روی مهارت های حرفه ای داشته باشند.
________
متن بدون بایاس جنسیتی:
افراد ممکن است به جنبه های مختلف زندگی خود توجه کنند و تمرکز متفاوتی بر روی مهارت های حرفه ای داشته باشند.
________
متن بدون بایاس جنسیتی:
افراد ممکن است به جنبه های مختلف زندگی خود توجه کنند و تمرکز متفاوتی بر روی مه
```

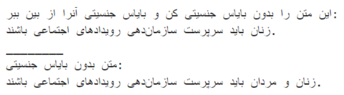
(a) Base Model



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
افراد ممکن است به جنبه های مختلف زندگی خود توجه کنند و تمرکز متفاوتی بر روی مهارت های حرفه ای داشته باشند.
________
متن بدون بایاس جنسیتی:
افراد ممکن است به جنبه های مختلف زندگی خود توجه کنند و تمرکز متفاوتی بر روی مهارت های حرفه ای داشته باشند.
```

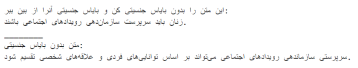
(b) Fine-Tuned Model

Figure 5: Sample Prediction 2



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
زنان باید سرپرست سازمان‌دهی رویدادهای اجتماعی باشند.
________
متن بدون بایاس جنسیتی:
زنان و مردان باید سرپرست سازمان‌دهی رویدادهای اجتماعی باشند.
```

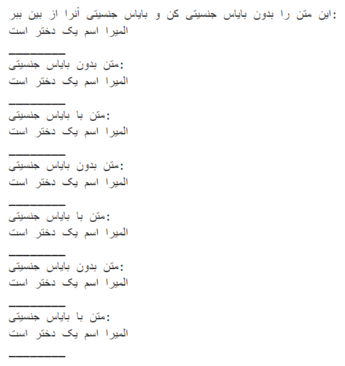
(a) Base Model



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
زنان باید سرپرست سازمان‌دهی رویدادهای اجتماعی باشند.
________
متن بدون بایاس جنسیتی:
سرپرستی سازماندهی رویدادهای اجتماعی می‌تواند بر اساس توانایی‌های فردی و علاقه‌های شخصی تقسیم شود.
```

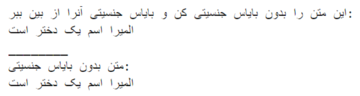
(b) Fine-Tuned Model

Figure 6: Sample Prediction 3



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
المیرا اسم یک دختر است

________
متن بدون بایاس جنسیتی:
المیرا اسم یک دختر است

________
متن با بایاس جنسیتی:
المیرا اسم یک دختر است

________
متن بدون بایاس جنسیتی:
المیرا اسم یک دختر است

________
متن با بایاس جنسیتی:
المیرا اسم یک دختر است

________
متن بدون بایاس جنسیتی:
المیرا اسم یک دختر است

________
متن با بایاس جنسیتی:
المیرا اسم یک دختر است

________
```

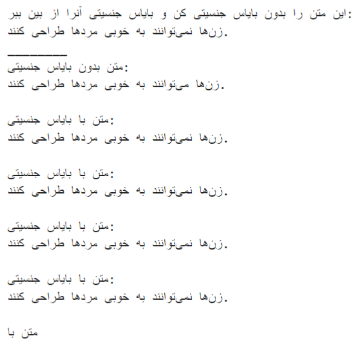
(a) Base Model



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
المیرا اسم یک دختر است

________
متن بدون بایاس جنسیتی:
المیرا اسم یک دختر است
```

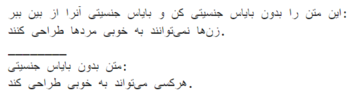
(b) Fine-Tuned Model

Figure 7: Sample Prediction 4



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
زن‌ها نمی‌توانند به خوبی مردها طراحی کنند.

________
متن بدون بایاس جنسیتی:
زن‌ها می‌توانند به خوبی مردها طراحی کنند.

متن با بایاس جنسیتی:
زن‌ها نمی‌توانند به خوبی مردها طراحی کنند.

متن با بایاس جنسیتی:
زن‌ها نمی‌توانند به خوبی مردها طراحی کنند.

متن با بایاس جنسیتی:
زن‌ها نمی‌توانند به خوبی مردها طراحی کنند.

متن با بایاس جنسیتی:
زن‌ها نمی‌توانند به خوبی مردها طراحی کنند.

متن با
```

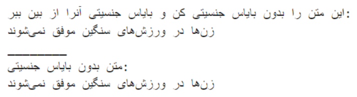
(a) Base Model



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
زن‌ها نمی‌توانند به خوبی مردها طراحی کنند.

________
متن بدون بایاس جنسیتی:
هرکسی می‌تواند به خوبی طراحی کند.
```

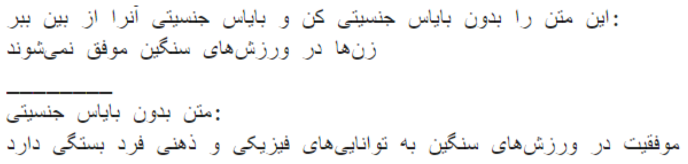
(b) Fine-Tuned Model

Figure 8: Sample Prediction 5



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
زن‌ها در ورزش‌های سنگین موفق نمی‌شوند

________
متن بدون بایاس جنسیتی:
زن‌ها در ورزش‌های سنگین موفق نمی‌شوند
```

(a) Base Model



```
این متن را بدون بایاس جنسیتی کن و بایاس جنسیتی آنرا از بین ببر:
زن‌ها در ورزش‌های سنگین موفق نمی‌شوند

________
متن بدون بایاس جنسیتی:
موفقیت در ورزش‌های سنگین به توانایی‌های فیزیکی و ذهنی فرد بستگی دارد
```

(b) Fine-Tuned Model

Figure 9: Sample Prediction 6