Predicting Human-Preferred Responses in Large Language Model Chatbots

Problem Statement: The problem is to predict which responses users will prefer in a head-to-head battle between chatbots powered by large language models (LLMs). The dataset has conversations from the Chatbot Arena, where different LLMs generate answers to user prompts.

Data Description:

- prompt - where the user provides some prompts in ARENA for models to respond.
- response_a- response from model_a
- response_b - response from model_b
- Winner- which model is a winner?
- model_a - specifies the model used
- model_b - the model used ,
- language - specifies which language is the prompt and responses are.

```
!pip install –U transformers accelerate peft bitsandbytes evaluate
```

Show hidden output

+ Code + Text

```
import os
import random
import numpy as np
import pandas as pd
import torch
from torch.utils.data import Dataset
from transformers import (
    GemmaTokenizerFast,
    Gemma2ForSequenceClassification,
    Trainer,
    TrainingArguments,
    EarlyStoppingCallback,
    DataCollatorWithPadding,
    TrainerCallback
)
from datasets import Dataset as HFDataset
from peft import LoraConfig, get_peft_model, TaskType, prepare_model_for_kbit_training
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import evaluate
```

Function to set all seeds for reproducibility

```
def seed_everything(seed=42):
    """
    Set seeds for reproducibility

    Args:
        seed: Seed value
    """
    os.environ['PYTHONHASHSEED'] = str(seed)
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.benchmark = True
    torch.backends.cudnn.deterministic = True
```

Set Hugging Face token

```
# Set Hugging Face token
os.environ["HF_TOKEN"] = "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"
```

Load_Data

1. Loads the data
2. Select the percent of the subsample I want to select to train and test

3. Make sure subsample has original class distribution

```
# Function to load and sample data with preserved class distribution
def load_data(file_path, sample_percentage=100, seed=42):
    """
    Load the dataset and sample according to provided percentage while preserving class distribution

    Args:
        file_path: Path to the dataset CSV/Parquet
        sample_percentage: Percentage of data to use (1-100)
        seed: Random seed

    Returns:
        Sampled DataFrame
    """


    df = pd.read_parquet(file_path)


    if sample_percentage < 100:
        # Stratified sampling to preserve class distribution
        if 'winner' in df.columns:
            # Use stratified sampling
            df = df.groupby('winner', group_keys=False).apply(
                lambda x: x.sample(frac=sample_percentage/100, random_state=seed)
            )
        else:
            # If no label column, use regular sampling
            df = df.sample(frac=sample_percentage/100, random_state=seed)

    print(f"Loaded {len(df)} samples ({sample_percentage}% of original data)")

    # Print class distribution if available
    if 'winner' in df.columns:
        class_dist = df['winner'].value_counts(normalize=True) * 100
        print("Class distribution:")
        for cls, pct in class_dist.items():
            print(f"  {cls}: {pct:.2f}%")

    return df
```

Split_data

Splits the data into train, validation, and test maintaining the original class distributions

```
# Function to split the data
def split_data(df, train_size=0.7, val_size=0.1, test_size=0.2, seed=42):
    """
    Split data into train, validation and test sets

    Args:
        df: Input DataFrame
        train_size: Proportion for training
        val_size: Proportion for validation
        test_size: Proportion for testing
        seed: Random seed

    Returns:
        train_df, val_df, test_df
    """
    # First split: Train + Val vs Test
    train_val_df, test_df = train_test_split(df, test_size=test_size, random_state=seed,
                                             stratify=df['winner'] if 'winner' in df.columns else None)

    # Second split: Train vs Val
    relative_val_size = val_size / (train_size + val_size)
    train_df, val_df = train_test_split(train_val_df, test_size=relative_val_size, random_state=seed,
                                        stratify=train_val_df['winner'] if 'winner' in train_val_df.columns else None)

    print(f"Train set: {len(train_df)} samples ({len(train_df)/len(df)*100:.2f}%)")
    print(f"Validation set: {len(val_df)} samples ({len(val_df)/len(df)*100:.2f}%)")
    print(f"Test set: {len(test_df)} samples ({len(test_df)/len(df)*100:.2f}%)")

    # Print class distribution for each split
    if 'winner' in df.columns:
        for name, split_df in [("Train", train_df), ("Validation", val_df), ("Test", test_df)]:
            class_dist = split_df['winner'].value_counts(normalize=True) * 100
            print(f"{name} class distribution:")
```

```
            for cls, pct in class_dist.items():
                print(f"  {cls}: {pct:.2f}%")

    return train_df, val_df, test_df
```

Truncate

smart_truncate will truncate the text based on the max_token set for the prompt and responses, but it will not truncate the end part; it will preserve the start and end but trim the middle. For example, if max_token for prompt is set to 400, then it will preserve the first 200 and last 200 words in the text.

```
# Function to perform smart truncation on longer text
def smart_truncate(text, tokenizer, max_tokens):
    """
    Performs smart truncation by keeping start and end of text if it exceeds token limit

    Args:
        text: Text to truncate
        tokenizer: Tokenizer to use
        max_tokens: Maximum number of tokens allowed

    Returns:
        Truncated text
    """
    encoded = tokenizer(text, return_offsets_mapping=True, add_special_tokens=False)
    if len(encoded['input_ids']) <= max_tokens:
        return text

    # Keep first and last parts
    first_part_tokens = max_tokens // 2
    last_part_tokens = max_tokens - first_part_tokens - 1  # -1 for the (snip) placeholder

    # Get character indices from token offsets
    if first_part_tokens > 0:
        _, first_end_idx = encoded['offset_mapping'][first_part_tokens-1]
    else:
        first_end_idx = 0

    if last_part_tokens > 0 and len(encoded['offset_mapping']) > last_part_tokens:
        last_start_idx, _ = encoded['offset_mapping'][-last_part_tokens]
    else:
        last_start_idx = len(text)

    # Combine first and last parts with (snip) in between
    truncated_text = text[:first_end_idx] + "\n(snip)\n" + text[last_start_idx:]
    return truncated_text
```

preprocess_data

- Checking missing Values
- Calling smart_truncate
- Encoding the winner column

```
# Function to preprocess dataset with smart truncation
def preprocess_data(df, tokenizer, prompt_max_tokens=150, response_max_tokens=350):
    """
    Preprocess data with smart truncation

    Args:
        df: Input DataFrame
        tokenizer: Tokenizer
        prompt_max_tokens: Maximum tokens for prompts
        response_max_tokens: Maximum tokens for responses

    Returns:
        Processed DataFrame
    """
    df = df.copy()

    # Fill NA values
    for col in ['prompt', 'response_a', 'response_b']:
        df[col] = df[col].fillna('')

    # Apply smart truncation to each column
    print("Applying smart truncation...")
    for col in ['prompt', 'response_a', 'response_b']:
```

```
        max_tokens = prompt_max_tokens if col == 'prompt' else response_max_tokens
        df[col] = df[col].apply(lambda x: smart_truncate(x, tokenizer, max_tokens))

    # Encode winner labels
    if 'winner' in df.columns:
        df['winner_encoded'] = df['winner'].map({'model_a': 0, 'model_b': 1})

    return df
```

Tokenize

- tokenize_with_special_tokens()- tokenize's the prompt, response_a and response_b seperately after they are truncated.
- As LLMs expect the same length of text for every row, some of the input will be padded, but those can be ignored by the model with an attention mask.All tokens, attention mask,winner label are stored in dictionary.

```
# Function to tokenize inputs with special tokens
def tokenize_with_special_tokens(df, tokenizer, max_length=512):
    """
    Tokenize inputs with special tokens

    Args:
        df: Input DataFrame
        tokenizer: Tokenizer
        max_length: Maximum sequence length

    Returns:
        Dictionary with input_ids, attention_mask and labels
    """
    # Create texts with special tokens
    prompts = ["<prompt>: " + p for p in df['prompt']]
    response_a = ["\n\n<response_a>: " + r for r in df['response_a']]
    response_b = ["\n\n<response_b>: " + r for r in df['response_b']]

    # Combine all parts into a single text
    texts = [p + ra + rb for p, ra, rb in zip(prompts, response_a, response_b)]

    # Tokenize
    tokenized = tokenizer(texts, max_length=max_length, truncation=True)

    # Create dataset dictionary
    dataset_dict = {
        'input_ids': tokenized['input_ids'],
        'attention_mask': tokenized['attention_mask']
    }

    # Add labels if available
    if 'winner_encoded' in df.columns:
        dataset_dict['labels'] = df['winner_encoded'].tolist()

    return dataset_dict
```

Huggingface Datasets

Converting the dictionary I created earlier while tokenizing are transforming into HuggingFace Datasets

```
# Function to create HuggingFace datasets
def create_hf_datasets(train_df, val_df, test_df, tokenizer, max_length=512):
    """
    Create HuggingFace datasets from DataFrames

    Args:
        train_df, val_df, test_df: DataFrames for each split
        tokenizer: Tokenizer
        max_length: Maximum sequence length

    Returns:
        HuggingFace datasets
    """
    train_dict = tokenize_with_special_tokens(train_df, tokenizer, max_length)
    val_dict = tokenize_with_special_tokens(val_df, tokenizer, max_length)
    test_dict = tokenize_with_special_tokens(test_df, tokenizer, max_length)

    hf_train_dataset = HFDataset.from_dict(train_dict)
    hf_val_dataset = HFDataset.from_dict(val_dict)
    hf_test_dataset = HFDataset.from_dict(test_dict)
```

```
    return hf_train_dataset, hf_val_dataset, hf_test_dataset
```

Initializing the Tokenizer

init_tokenizer()- Initializes the tokenizer which is GemmaTokenizerFast

```
def init_tokenizer(model_path,use_auth_token=True):
    """
    Initialize the GemmaTokenizerFast

    Args:
        model_path: Path to the model/tokenizer

    Returns:
        tokenizer
    """
    tokenizer = GemmaTokenizerFast.from_pretrained(
        model_path,
        add_eos_token=True,
        padding_side="right",
        use_auth_token=use_auth_token
         # Add this to specify we're loading from local files
    )
    return tokenizer
```

Initializing the model

init_model()- initializing the model with the following

- Initializing the model from Hugging Face Transformers
- Using prepare_model_for_kbit_training to quantize the model.
- As part of PEFT(Parameter Efficient Finetuning) i am using Low Rank Adaption technique where i applying to layers 6 to 26(First 6 layers are being freezed)

```
# Function to initialize model with LoRA and layer freezing
def init_model(model_name="google/gemma-2-2b", num_labels=2, quantize=True, num_layers=26, freeze_layers=6,use_auth_token=Tr
    """
    Initialize Gemma2ForSequenceClassification with LoRA parameters and layer freezing

    Args:
        model_path: Path to the model
        num_labels: Number of output classes
        quantize: Whether to use quantization
        num_layers: Total number of layers in the model
        freeze_layers: Number of layers to freeze

    Returns:
        LoRA-configured model
    """
    # Load the base model
    model = Gemma2ForSequenceClassification.from_pretrained(
        model_name,
        num_labels=num_labels,
        torch_dtype=torch.bfloat16 if torch.cuda.is_available() else torch.float32,
        device_map="auto",
        use_cache=False,  # Important for training
        use_auth_token=use_auth_token,
        problem_type="single_label_classification",

    )

    # Prepare model for k-bit training if quantizing
    if quantize:
        model = prepare_model_for_kbit_training(model)

    # LoRA configuration with layer targeting
    lora_config = LoraConfig(
        task_type=TaskType.SEQ_CLS,
        r=32,  # Rank
        lora_alpha=64,  # Alpha parameter for LoRA scaling
        lora_dropout=0.05,  # Dropout probability for LoRA layers
        target_modules=["q_proj", "k_proj", "v_proj"],  # Target attention mechanisms
        layers_to_transform=[i for i in range(num_layers) if i >= freeze_layers],  # Only apply to non-frozen layers
        bias='none',
    )
```

```
    # Apply LoRA
    model = get_peft_model(model, lora_config)
    model.print_trainable_parameters()

    return model
```

Train Model

Using the trainer API to train the model and passing all the hyperparameters, data, and metrics I want the model to get evaluated. Saves the best model that has the best Validation Accuracy

```
def train_model(train_dataset, val_dataset, output_dir, model, learning_rate=5e-5, batch_size=1, num_epochs=3,
                grad_accum_steps=4, eval_steps=None, save_steps=200):
    """
    Train the model with given parameters

    Args:
        model: The model to train
        train_dataset, val_dataset: Training and validation datasets
        output_dir: Directory to save model artifacts
        learning_rate: Learning rate
        batch_size: Batch size
        num_epochs: Number of training epochs
        grad_accum_steps: Gradient accumulation steps
        eval_steps: Evaluation steps (if None, evaluate every epoch)
        save_steps: Model saving steps

    Returns:
        Trained model and training metrics
    """
    # Compute metrics function
    def compute_metrics(eval_pred):
        predictions, labels = eval_pred
        predictions = np.argmax(predictions, axis=1)
        return {'accuracy': accuracy_score(y_true=labels, y_pred=predictions)}

    # Set up training arguments
    training_args = TrainingArguments(
        output_dir=output_dir,
        learning_rate=learning_rate,
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        num_train_epochs=num_epochs,
        weight_decay=0.01,
        # Evaluate during training to track progress
        eval_strategy="steps",
        eval_steps=eval_steps if eval_steps else 500,  # Default to evaluate every 100 steps
        # Save checkpoints
        save_strategy="steps",
        save_steps=save_steps,
        save_total_limit=3,
        # Load best model at end
        load_best_model_at_end=True,
        metric_for_best_model="accuracy",
        greater_is_better=True,
        # Logging settings
        logging_dir=f"{output_dir}/logs",
        logging_strategy="steps",
        logging_steps=500,
        logging_first_step=True,  # Log the first step to see initial metrics
        # Report training loss
        report_to="tensorboard",
        # Precision settings
        fp16=True,
        bf16=False,
        # Other settings
        gradient_accumulation_steps=grad_accum_steps,
        warmup_ratio=0.1,  # Increased from 0.01 to help with training stability
        lr_scheduler_type="cosine",  # Changed from linear to cosine for better learning rate decay
        # Add label smoothing to help with overconfidence
        label_smoothing_factor=0.1,
        # Group by length for more efficient batching
        group_by_length=True,
        # Add some noise to avoid overfitting
        optim="adamw_torch_fused" if torch.cuda.is_available() else "adamw_torch",
    )

    # Initialize trainer with early stopping
```

```
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        compute_metrics=compute_metrics,
        callbacks=[
            EarlyStoppingCallback(early_stopping_patience=3),
            # Add a custom callback to track training accuracy
            TrainCallback(),
        ],
        data_collator=DataCollatorWithPadding(tokenizer=tokenizer),  # Dynamic padding
    )

    # Train the model
    trainer.train()

    # Save the best model
    trainer.save_model(f"{output_dir}/best_model")

    return trainer

# Custom callback to track training metrics
class TrainCallback(TrainerCallback):
    """Custom callback to track training metrics"""

    def on_log(self, args, state, control, logs=None, **kwargs):
        """Called each time trainer logs metrics"""
        logs = logs or {}

        # Extract training accuracy if available
        if 'loss' in logs:
            print(f"Step {state.global_step}: Training Loss: {logs['loss']}")

        # Print evaluation metrics when they exist
        if 'eval_accuracy' in logs:
            print(f"Step {state.global_step}: Eval Accuracy: {logs['eval_accuracy']}, Eval Loss: {logs.get('eval_loss', 'N/A
```

Testing

Tests the Finetuned model with test data

```
# Function to evaluate on test set
def evaluate_model(trainer, test_dataset):
    """
    Evaluate model on test set

    Args:
        trainer: Trained Trainer object
        test_dataset: Test dataset

    Returns:
        Evaluation metrics
    """
    # Evaluate on test set
    test_results = trainer.evaluate(test_dataset)
    print(f"Test set results: {test_results}")
    return test_results
```

Below all are Funcations calls to above Functions

```
# Example usage:
# 1. Set seed for reproducibility
seed_everything(42)
```

```
# 2. Load and sample data
data = load_data("/content/train.parquet", sample_percentage=100)
```

```
Loaded 48439 samples (100% of original data)
Class distribution:
  model_b: 50.54%
  model_a: 49.46%
```

```
# 3. Split the data
train_df, val_df, test_df = split_data(data)
```

```
Train set: 33907 samples (70.00%)
Validation set: 4844 samples (10.00%)
Test set: 9688 samples (20.00%)
Train class distribution:
  model_b: 50.54%
  model_a: 49.46%
Validation class distribution:
  model_b: 50.54%
  model_a: 49.46%
Test class distribution:
  model_b: 50.54%
  model_a: 49.46%
```

```
model_name = "google/gemma-2-2b"  # HuggingFace model name
tokenizer = init_tokenizer(model_name, use_auth_token=True)
```

```
/usr/local/lib/python3.11/dist-packages/transformers/tokenization_utils_base.py:1932: FutureWarning: The `use_auth_token
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens),
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  46.4k/46.4k [00:00<00:00, 5.50MB/s]
tokenizer.model: 100%  4.24M/4.24M [00:00<00:00, 64.0MB/s]
tokenizer.json: 100%  17.5M/17.5M [00:00<00:00, 84.2MB/s]
special_tokens_map.json: 100%  636/636 [00:00<00:00, 81.6kB/s]
```

```
# 5. Preprocess data with smart truncation
train_df = preprocess_data(train_df, tokenizer)
val_df = preprocess_data(val_df, tokenizer)
test_df = preprocess_data(test_df, tokenizer)
```

```
Applying smart truncation...
Applying smart truncation...
Applying smart truncation...
```

```
model = init_model(
    model_name,
    num_labels=2,
    quantize=True,
    num_layers=26,
    freeze_layers=6,
    use_auth_token=True
)
```

```
/usr/local/lib/python3.11/dist-packages/transformers/modeling_utils.py:4056: FutureWarning: The `use_auth_token` argumen
  warnings.warn(
config.json: 100%  818/818 [00:00<00:00, 106kB/s]
model.safetensors.index.json: 100%  24.2k/24.2k [00:00<00:00, 2.90MB/s]
Fetching 3 files: 100%  3/3 [00:27<00:00, 11.42s/it]
model-00001-of-00003.safetensors: 100%  4.99G/4.99G [00:26<00:00, 349MB/s]
model-00002-of-00003.safetensors: 100%  4.98G/4.98G [00:27<00:00, 470MB/s]
model-00003-of-00003.safetensors: 100%  481M/481M [00:01<00:00, 318MB/s]
Loading checkpoint shards: 100%  3/3 [00:03<00:00,  1.15it/s]
Some weights of Gemma2ForSequenceClassification were not initialized from the model checkpoint at google/gemma-2-2b and
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
trainable params: 7,049,728 || all params: 2,621,396,224 || trainable%: 0.2689
```

```
# 6. Create HuggingFace datasets
hf_train_dataset, hf_val_dataset, hf_test_dataset = create_hf_datasets(
    train_df, val_df, test_df, tokenizer, max_length=512
)
```

Training Results:

Training and Validation Accuracy:0.5772

```
# 8. Train model
output_dir = "./model_output"
trainer = train_model(
    hf_train_dataset,
    hf_val_dataset,
    output_dir,
    model,
    learning_rate=2e-5,
    batch_size=1,
    num_epochs=1,
    grad_accum_steps=8,
    eval_steps=500,  # Evaluate every 500 steps
    save_steps=500,  # Save every 200 5teps

)
```

```
No label_names provided for model class `PeftModelForSequenceClassification`. Since `PeftModel` hides base models input
```

[2223/4238 1:08:41 < 1:02:19, 0.54 it/s, Epoch 0.52/1]

Step	Training Loss	Validation Loss	Accuracy
500	1.764000	0.743949	0.516722
1000	0.733500	0.777601	0.500619
1500	0.711500	0.822648	0.507019
2000	0.702700	0.684144	0.561932

```
Step 1: Training Loss: 3.8183
Step 500: Training Loss: 1.764
Step 500: Eval Accuracy: 0.5167217175887696, Eval Loss: 0.7439494132995605
Step 1000: Training Loss: 0.7335
Step 1000: Eval Accuracy: 0.5006193228736582, Eval Loss: 0.7776008248329163
Step 1500: Training Loss: 0.7115
Step 1500: Eval Accuracy: 0.5070189925681255, Eval Loss: 0.8226484656333923
Step 2000: Training Loss: 0.7027
Step 2000: Eval Accuracy: 0.5619322873658134, Eval Loss: 0.6841441988945007
```

[4238/4238 2:12:46, Epoch 0/1]

Step	Training Loss	Validation Loss	Accuracy
500	1.764000	0.743949	0.516722
1000	0.733500	0.777601	0.500619
1500	0.711500	0.822648	0.507019
2000	0.702700	0.684144	0.561932
2500	0.702800	0.683765	0.560900
3000	0.690600	0.686002	0.562139
3500	0.685500	0.678584	0.573080
4000	0.685000	0.678262	0.577209

```
Step 2500: Training Loss: 0.7028
Step 2500: Eval Accuracy: 0.5609000825763831, Eval Loss: 0.6837645173072815
Step 3000: Training Loss: 0.6906
Step 3000: Eval Accuracy: 0.5621387283236994, Eval Loss: 0.686001718044281
Step 3500: Training Loss: 0.6855
Step 3500: Eval Accuracy: 0.5730800990916598, Eval Loss: 0.6785843372344971
Step 4000: Training Loss: 0.685
Step 4000: Eval Accuracy: 0.5772089182493807, Eval Loss: 0.6782618761062622
```

Test Results

Test Accuracy:0.5614

```
# 9. Evaluate on test set
test_results = evaluate_model(trainer, hf_test_dataset)
```

[9688/9688 10:31]

```
Step 4238: Eval Accuracy: 0.5614161849710982, Eval Loss: 0.6821391582489014
Test set results: {'eval_loss': 0.6821391582489014, 'eval_accuracy': 0.5614161849710982, 'eval_runtime': 631.4887, 'eval
```