2. Documentation on Dataset Preprocessing, Model Training, and Evaluation Report

2.1. Dataset Preprocessing

2.1.1. Data Loading

- The dataset is loaded from a CSV file (`final\_labels.csv`).

- Required columns:

- `body`: The text of the comment.

- `level\_1`: The first-level label (e.g., `toxic`, `neutral`).

- `level\_2`: The second-level label (e.g., `hate\_speech`, `offensive\_language`).

- `split`: Indicates whether the data belongs to the `train` or `test` set.

- `parent\_body`: The text of the parent comment (used to provide context).

Code:

df = pd.read\_csv("/content/final\_labels.csv")

2.1.2. Handling Missing Values

- Missing values in the required columns (`body`, `level\_1`, `level\_2`, `split`, `parent\_body`) are dropped to ensure clean data for training and evaluation.

Code:

df = df[['body', 'level\_1', 'level\_2', 'split', 'parent\_body']].dropna()

2.1.3. Context Creation

- The `context` column is created by concatenating the `parent\_body` and `body` with a `[SEP]` token.

- If `parent\_body` is missing, it is replaced with `"[No Context]"`.

Code:

if 'parent\_body' not in df.columns:

df['parent\_body'] = "[No Context]"

df['context'] = df['parent\_body'] + " [SEP] " + df['body']

2.1.4. Label Encoding

- Labels for `level\_1` and `level\_2` are encoded into numerical values using `LabelEncoder`.

Code:

from sklearn.preprocessing import LabelEncoder

# Encode level\_1 labels

level\_1\_mapping = {label: idx for idx, label in enumerate(df['level\_1'].unique())}

df['level\_1\_label'] = df['level\_1'].map(level\_1\_mapping).astype(int)

# Encode level\_2 labels

level\_2\_mapping = {label: idx for idx, label in enumerate(df['level\_2'].unique())}

df['level\_2\_label'] = df['level\_2'].map(level\_2\_mapping).astype(int)

2.1.5. Train-Test Split

- The dataset is split into training and testing sets using an 80-20 split.

Code:

from sklearn.model\_selection import train\_test\_split

train\_df, test\_df = train\_test\_split(df, test\_size=0.2, random\_state=42)

2.2. Model Training

2.2.1. Dataset Creation

- A custom `MisogynyDataset` class is used to tokenize and format the data for the model.

Code:

class MisogynyDataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, texts, labels\_1, labels\_2, tokenizer, max\_length=256):

self.texts = texts.tolist()

self.labels\_1 = labels\_1.tolist()

self.labels\_2 = labels\_2.tolist()

self.tokenizer = tokenizer

self.max\_length = max\_length

def \_\_getitem\_\_(self, idx):

encoding = self.tokenizer(

self.texts[idx],

truncation=True,

padding='max\_length',

max\_length=self.max\_length,

return\_tensors='pt'

)

return {

"input\_ids": encoding["input\_ids"].squeeze(),

"attention\_mask": encoding["attention\_mask"].squeeze(),

"labels\_1": torch.tensor(self.labels\_1[idx], dtype=torch.long),

"labels\_2": torch.tensor(self.labels\_2[idx], dtype=torch.long),

}

2.2.2. Model Initialization

- The `roberta-large` model and tokenizer are loaded.

- The hierarchical classifier is initialized with the base model and two classifiers.

Code:

from transformers import AutoTokenizer, AutoModel

model\_name = "roberta-large"

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

class HierarchicalClassifier(nn.Module):

def \_\_init\_\_(self, model\_name, num\_labels\_1, num\_labels\_2):

super(HierarchicalClassifier, self).\_\_init\_\_()

self.base\_model = AutoModel.from\_pretrained(model\_name)

self.dropout = nn.Dropout(0.3)

self.classifier\_1 = nn.Linear(self.base\_model.config.hidden\_size, num\_labels\_1)

self.classifier\_2 = nn.Linear(self.base\_model.config.hidden\_size, num\_labels\_2)

model = HierarchicalClassifier(model\_name, len(level\_1\_mapping), len(level\_2\_mapping))

2.2.3. Training Arguments

- Training arguments are configured using `TrainingArguments`.

Code:

from transformers import TrainingArguments

training\_args = TrainingArguments(

output\_dir="./results",

evaluation\_strategy="epoch",

save\_strategy="epoch",

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

num\_train\_epochs=4,

weight\_decay=0.01,

logging\_dir="./logs",

logging\_steps=10,

load\_best\_model\_at\_end=True,

metric\_for\_best\_model="overall\_accuracy",

greater\_is\_better=True,

)

2.2.4. Trainer Setup

- The `Trainer` class is used to handle the training loop, evaluation, and saving of the model.

Code:

from transformers import Trainer

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=test\_dataset,

compute\_metrics=compute\_metrics

)

2.2.5. Training

- The model is trained using the `trainer.train()` method.

Code:

trainer.train()

2.3. Model Evaluation

2.3.1. Metrics Calculation

- Custom metrics are computed for evaluation:

- Overall Accuracy: A sample is correct only if both `level\_1` and `level\_2` predictions are correct.

- Average Precision, Recall, and F1-Score: Weighted averages for both levels.

Code:

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

def compute\_metrics(pred):

labels\_1 = pred.label\_ids[0]

labels\_2 = pred.label\_ids[1]

logits\_1, logits\_2 = pred.predictions

preds\_1 = np.argmax(logits\_1, axis=-1)

preds\_2 = np.argmax(logits\_2, axis=-1)

overall\_accuracy = np.mean((preds\_1 == labels\_1) & (preds\_2 == labels\_2))

precision\_1 = precision\_score(labels\_1, preds\_1, average='weighted')

precision\_2 = precision\_score(labels\_2, preds\_2, average='weighted')

avg\_precision = (precision\_1 + precision\_2) / 2

recall\_1 = recall\_score(labels\_1, preds\_1, average='weighted')

recall\_2 = recall\_score(labels\_2, preds\_2, average='weighted')

avg\_recall = (recall\_1 + recall\_2) / 2

f1\_1 = f1\_score(labels\_1, preds\_1, average='weighted')

f1\_2 = f1\_score(labels\_2, preds\_2, average='weighted')

avg\_f1 = (f1\_1 + f1\_2) / 2

return {

"overall\_accuracy": overall\_accuracy,

"avg\_precision": avg\_precision,

"avg\_recall": avg\_recall,

"avg\_f1": avg\_f1

}

2.3.2. Explainability\*\*

- SHAP values are computed to explain model predictions.

Code:

import shap

def explain\_prediction(text, model, tokenizer):

masker = shap.maskers.Text(tokenizer)

explainer = shap.Explainer(model\_forward, masker)

shap\_values = explainer(text)

return shap\_values

2.3.3. Moderation Suggestions

- Constructive alternatives are generated for toxic comments.

Code:

def suggest\_moderation(text, model, tokenizer):

inputs = tokenizer(text, return\_tensors="pt", truncation=True, padding=True)

device = next(model.parameters()).device

inputs = {k: v.to(device) for k, v in inputs.items()}

\_, logits\_1, \_ = model(\*\*inputs)

prediction = torch.argmax(logits\_1, dim=1).item()

suggestions = {

0: "This comment is neutral.",

1: "Your comment could be more positive. Consider saying: 'I appreciate diverse perspectives and believe we can improve together.'",

2: "Your comment appears harsh. A more constructive alternative might be: 'I value different opinions and think there is room for improvement in our approach.'"

}

return suggestions.get(prediction, "Unable to classify.")

2.4. Summary

- The dataset is preprocessed to include context and encode labels.

- The model is trained using a hierarchical classifier with two output heads.

- Evaluation metrics include overall accuracy, average precision, recall, and F1-score.

- SHAP values are used for explainability, and moderation suggestions are provided for toxic comments.