Emotion Recognition in Texts using Deep Learning Techniques

TECHNICAL REPORT

From April 10th to June 10th

University of Portsmouth (UoP)

Thomas Chu

2nd year of a Bachelor's degree program in Computer Science

Dr. Alaa Mohasseb, Senior Lecturer

Ingrid Couturier, IUT Tutor

4th Semester, year 2022-2023

Table of Contents

Introduction

This report aims to present a project focused on developing highly accurate sentiment analysis model. The primary objective is to effectively analyze emotions conveyed in textual data while striving for unbiased results. The report serves as a comprehensive guide through the project, including an overview of the project workflow, instructions on building a deBERTa model, and fine-tuning a OpenAI model. Its purpose is to provide valuable insights for future enhancements and advancements for this project.

1 Prerequisites

1.1 Project Structure

The project follows a comprehensive structure for sentiment analysis, incorporating multiple modalities such as video, image, audio, and text. The input is obtained from a video source, which is then processed to extract components including audio, images, and text. Each modality undergoes data preparation to ensure compatibility with the subsequent prediction models. Specific prediction models are constructed for each modality and each generate emotion predictions. The predicted emotions from all modalities are then combined to produce a unified and comprehensive emotions output. This project structure uses the strengths of different modalities to enhance the accuracy and effectiveness of emotion prediction.

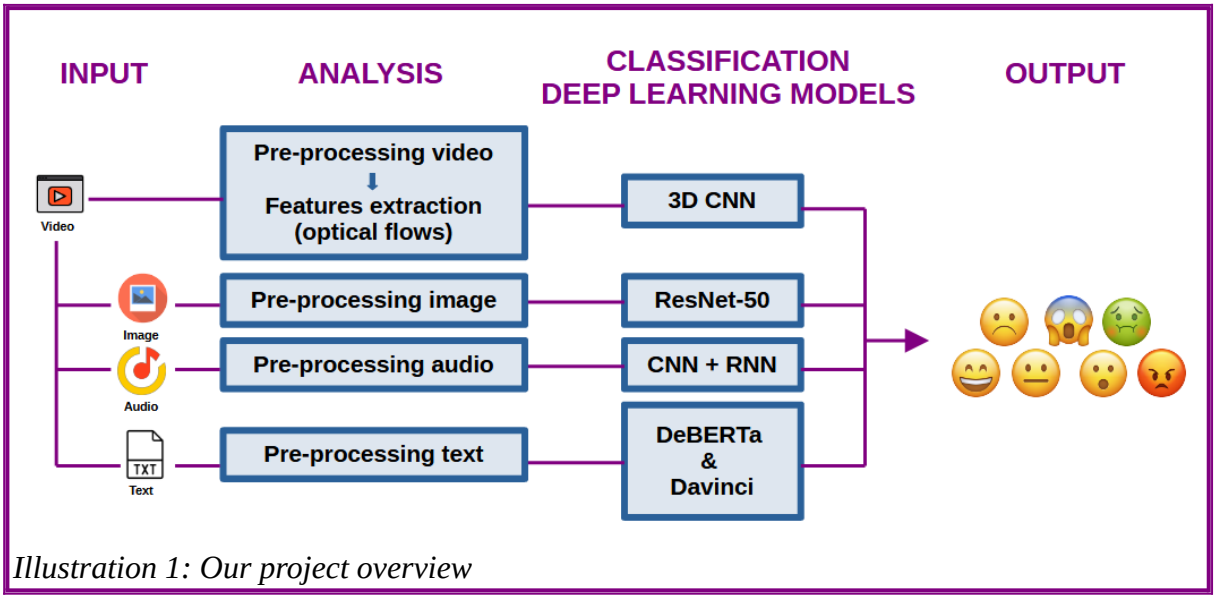

Illustration 1: Our project overview

My contribution in this project was specifically focused on the textual part. My work can be divided into four distinct parts, with each part serving a specific purpose:

- Creation of a deBERTa model
- Creation of a fine-tuned OpenAI model
- A comparative study between the two

1.2 Library and tool used

There is two options for this project: an easy approach and a more challenging one. The easy approach involves utilizing Google Colab, which provides a working environment that runs on Google's cloud servers and comes with (almost) all the setup you need to start coding. By following the code of the project, the developer can easily install any missing packages and ensures a smooth setup process for the project.

The more complex approach requires setting up a local environment. To proceed, the developer need to have Python installed on their machines, a functional Jupyter Notebook, and install all the setup beforehand.

Several important libraries and tools were utilized in this project:

- **numpy**: This library provides efficient data manipulation capabilities.
- **pandas**: This library is crucial for data analysis, and data manipulation. It enables effective preprocessing and organization of textual data before feeding it into the sentiment analysis models.
- **text_hammer**: This library offers a ranges of methods for text preprocessing.
- **sentencepiece**: This is an efficient tokenization library.
- **sklearn**: The scikit-learn library is a powerful toolkit for machine learning tasks.
- **tensorflow**: TensorFlow is a machine learning library that provides a framework for developing and training neural networks.
- **openai**: The openai library allows us to utilize the OpenAI platform to work on OpenAI models.
- **Wandb**: The wandb library (Weights and Biases) is a powerful tool for experiment tracking and collaboration. It enables easy logging of experiment configurations, metrics and visualizations for our models.

2 Technical presentation of the project

2.1 Project architecture

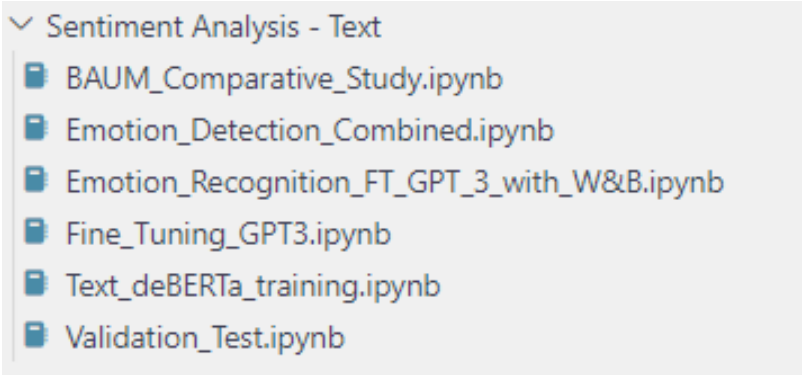

Illustration 2: Project architecture of my work

Jupyter Notebooks served as the primary tool for documenting and organizing code. Due to their inherent ease of use and excellent support for data analysis and machine learning tasks, they proved ideal for implementing the project's objectives. The datasets and trained models were stored in a shared drive, enabling seamless collaboration among team members. Each notebook file has a specific purpose and perform a distinct task within the project:

- **Text_deBERTa_training**: This notebook was dedicated to the creation, training and evaluation of a deBERTa model specifically designed for sentiment analysis.
- **Fine_Tuning_GPT3**: This notebook loads a Davinci model from OpenAI, model which is fine-tuned for sentiment analysis.
- **Emotion_Recognition_FT_GPT_3_with_W&B**: This notebook does the same as *Fine_Tuning_GPT3*, but it utilizes the wandb tool to compare hyperparameters and overall results of different models.
- **BAUM_Comparative_Study**: Using the BAUM dataset, this notebook was dedicated to evaluating and comparing the performance of the deBERTa model and the fine-tuned Davinci model.

- **Validation_Test**: This notebook involved evaluating the performance of the deBERTa model and the fine-tuned Davinci model using additional datasets sourced from Kaggle.
- **Emotion_Detection_Combined**: This notebook aimed to combine the trained models thus far, leveraging their capabilities to detect emotions in videos and live streams.

2.2 Training a deBERTa model

2.2.1 Data pre-processing

Before getting into the model building, a very important task need to be done. When it comes to unstructured data like data, cleaning and preprocessing the data is crucial to achieve good results. I used **text_hammer** which is a great tool for common text preprocessing like lower casing, contraction removal, special removal and more:

```
def text_preprocessing(df,col_name):
    column = col_name
    df[column] = df[column].progress_apply(lambda x:str(x).lower())
    df[column] = df[column].progress_apply(lambda x: th.cont_exp(x))
    df[column] = df[column].progress_apply(lambda x: th.remove_emails(x))
    df[column] = df[column].progress_apply(lambda x: th.remove_html_tags(x))
    df[column] = df[column].progress_apply(lambda x: th.remove_special_chars(x))
    df[column] = df[column].progress_apply(lambda x: th.remove_accented_chars(x))
    df[column] = df[column].progress_apply(lambda x: th.remove_stopwords(x))
    df[column] = df[column].progress_apply(lambda x: th.remove_urls(x))
    return(df)
```

Illustration 3: One of my preprocessing method

2.2.2 Representation

Next, I created a representation for text that could be fed into a deep learning model. The deep learning algorithms take as input a sequence of text to learn the structure of text just like a human does. Since Machine cannot understand words they expect their data in numerical form. To represent our text data as a series of numbers, we need to fully understand the **Keras Tokenizer** function.

In simple words, a tokenizer is a utility function that splits a sentence into words. The tokenizer tokenizes (or splits) the text into tokens (or words) while only keeping the most important information:

```
def tokenize_data(data, tokenizer):
    tokenized_data = tokenizer(
        text=data.sentences.tolist(),
        add_special_tokens=True,
        max_length=22,
        truncation=True,
        padding=True,
        return_tensors='tf',
        return_token_type_ids=False,
        return_attention_mask=True,
        verbose=True
    )
    return tokenized_data

x_train = tokenize_data(data_train, tokenizer)
x_test = tokenize_data(data_test, tokenizer)
```

Illustration 4: Tokenizing my text data

I directly used the tokenizer from the deBERTa models available for free through **huggingface**. This model outperforms BERT and RoBERTa on a good majority of NLP tasks with 80GB training data:

```
from transformers import AutoTokenizer,TFDebertaModel
tokenizer = AutoTokenizer.from_pretrained('microsoft/deberta-base')
deberta = TFDebertaModel.from_pretrained('microsoft/deberta-base')
```

Illustration 5: Loading the deBERTa model and its tokenizer

2.2.3 Setting up the model

Now the data is fully tokenized with input sequences and attention masks (numerical data for the model to process). These inputs are passed through an embedding layer to obtain contextualized words representations. The condensed representation is passed through dense layers with activation function to capture complex patterns and relationships within the data. Dropout regularization is applied to prevent overfitting[1], enhancing the model's generalization ability.

```
max_len = 22
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense

input_ids = Input(shape=(max_len,), dtype=tf.int32, name="input_ids")
input_mask = Input(shape=(max_len,), dtype=tf.int32, name="attention_mask")

embeddings = deberta(input_ids,attention_mask = input_mask)[0]
out = tf.keras.layers.GlobalMaxPool1D()(embeddings)
out = Dense(128, activation='relu')(out)
out = tf.keras.layers.Dropout(0.1)(out)
out = Dense(32,activation = 'relu')(out)

y = Dense(7,activation = 'sigmoid')(out)

model = tf.keras.Model(inputs=[input_ids, input_mask], outputs=y)
model.layers[2].trainable = True
```

Illustration 6: Neural Network Architecture with deBERTa Embeddings

The specific choices made in the layering and algorithms used in this code are based on empirical observations and our experience. The decision to use these particular algorithms and configurations has been determined through trial and error to achieve satisfactory performance in accuracy.

1 Overfitting occurs when a model becomes too specific or overly complex, resulting in poor generalization to unseen data. It happens when the models learns to fit the training data too closely.

2.2.4 Model compilation

Now we need to set the model's training configuration. The Adam optimizer was chosen based on its popularity and effectiveness in optimizing deep learning models. The learning rate of 1e-05 is often found to be a suitable value for Adam optimization. The additional parameters (epsilon and clipnorm) are set to default to ensure stable and controlled optimization. The EarlyStopping callback from Keras helps a lot in preventing overfitting by monitoring the loss and stopping training if the loss does not improve after one additional epoch (patience set to 1).

```
optimizer = Adam(
    learning_rate=1e-05,
    epsilon=1e-08,
    clipnorm=1.0)

# earlyStopping from Keras
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=1)

# Set loss and metrics
loss =CategoricalCrossentropy(from_logits = True)
metric = CategoricalAccuracy('balanced_accuracy'),

# Compile the model
model.compile(
    optimizer = optimizer,
    loss = loss,
    metrics = metric)
```

Illustration 7: Configuring Adam Optimizer and Metrics for the deBERTa model

2.2.5 Model training

The training can now begin, to do that, we call the fit() function with specific parameters. The number of epochs indicates that the model will be trained for 10 iterations over the entire training dataset. The batch size of 32 indicates that the model will update its weights after processing 32 samples at a time. And the callback function is the EarlyStopping method that we defined earlier. Those are hyperparameters that we want to change when trying to better the accuracy of our model:

```
# use max len(test) instead of padded sequence
history = model.fit(
    x={'input_ids': x_train['input_ids'], 'attention_mask': x_train['attention_mask']},
    y=to_categorical(data_train.Emotion),
    validation_data=(
        {'input_ids': x_test['input_ids'], 'attention_mask':  x_test['attention_mask']},
        to_categorical(data_test.Emotion)
    ),
    epochs=10,
    batch_size=32,
    callbacks=[callback]
)
```

Illustration 8: Training the deBERTa model

After completing this training process, we can proceed to make predictions using the trained model. The outputs generated by the model will be aligned with the target column from the dataset used for training (in our case the column *Emotion*, from the BAUM dataset).

2.3 Fine-tuning an OpenAI model

Fine-tuning an OpenAI model offers a faster and relatively less complex approach. However, it is important to acknowledge that certain aspects of the process may be challenging to grasp fully. This is primarily due to the novelty of the technology and the limited availability of comprehensive documentation on the OpenAI website and the internet in general. This work may require further exploration and experimentation but the models we made are fully operational so I will explain them.

2.3.1 Prompt engineering

OpenAI models typically work with a two-column format: a prompt column and a completion column. The prompt provides context or instructions to guide the model's generation in the completion column. The challenge lies in finding the right balance between providing sufficient guidance to the model and avoiding over-specifying the desired output, which could limit the model's creativity. To do this, I used the OpenAI Playground that allows for interactive experimentation with a range of OpenAI models:

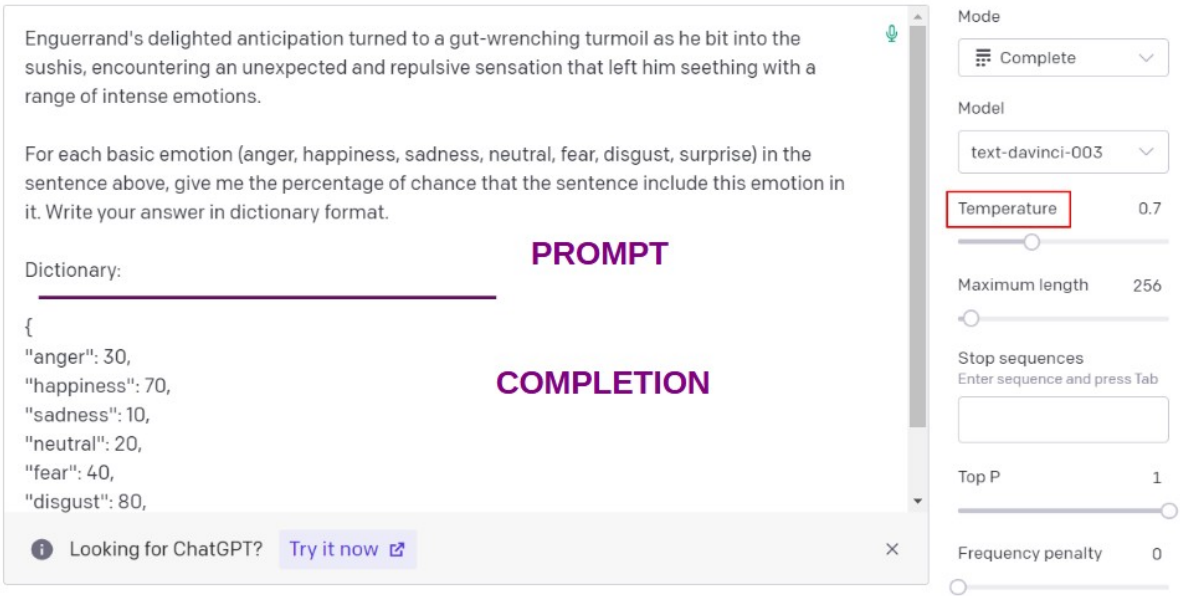

Illustration 9: Experimentation in the OpenAI Playground

After determining the most effective prompt format in the OpenAI Playground, we applied it to our dataset, ensuring our prompt and completion follow this specific format. Experimenting with the temperature parameter can be interesting, as it controls the randomness of the model's output, balancing creativity and coherence. It is also important to explore alternative models instead of *davinci*, considering their capabilities, performance, and cost. In our project, the *davinci-003* model demonstrated superior performance aligning well with our current requirements and yielding high-quality responses.

```
df['prompt'] = df['prompt'] + "\n\n For each basic emotion (anger, happiness, sadness, neutral, fear, disgust, surprise), gi
df['completion'] = df['completion'].apply(lambda x: " " + str(x) + " END")
df.head(3)
```

Illustration 10: Applying our format to the dataset

Adding " END" is very important for good generation according to the OpenAI documentation. It is an ending token that serves as a stop sequence during fine-tuning.

2.3.2 Model fine-tuning

Similar to the DeBERTa training process, preparing the data to fine-tune an OpenAI model is a straightforward task. OpenAI provides tools that simplify this process, allowing us to efficiently transform our DataFrame into a required JSONL file format. This file format serves as an input parameter for the fine-tuning process. Once it was done, we launched the OpenAI fine-tune method and successfully created our personalized fine-tuned Davinci model for sentiment analysis:

```
! openai api fine_tunes.follow -i ft-4LnA9z376nXjHfdvwosa6P24

[2023-05-19 09:07:02] Created fine-tune: ft-4LnA9z376nXjHfdvwosa6P24
[2023-05-19 09:08:27] Fine-tune costs $6.80
[2023-05-19 09:08:28] Fine-tune enqueued. Queue number: 0
[2023-05-19 09:08:29] Fine-tune started
[2023-05-19 09:13:57] Completed epoch 1/4
[2023-05-19 09:16:49] Completed epoch 2/4
[2023-05-19 09:19:41] Completed epoch 3/4
[2023-05-19 09:22:33] Completed epoch 4/4
[2023-05-19 09:23:18] Uploaded model: davinci:ft-personal-2023-05-19-09-23-18
[2023-05-19 09:23:20] Uploaded result file: file-lvhRDM0ZLcOGqy260SglDZki
[2023-05-19 09:23:20] Fine-tune succeeded

Job complete! Status: succeeded 🎉
```

Illustration 11: Desired output of a successful fine-tuning attempt

To predict emotions using this model, we can utilize the following code:

```python
import openai
openai.api_key = 'YOUR_API_KEY'

response = openai.Completion.create(
  model="davinci:ft-personal-2023-05-15-12-52-32",
  prompt=prompt,
  temperature=0.7,
  max_tokens=256,
  top_p=1,
  frequency_penalty=0,
  presence_penalty=0,
  stop=[" end"]
)

print(response['choices'][0]['text'])
```

Illustration 12: Code that generates an emotion using our fine-tuned model

The ***prompt*** parameter should contain the specific prompt text format from the prompt engineering process, tailored to our task. All the other parameters are the exact same parameters as the OpenAI Playground experiment from the very first step.

2.3.3 The Weights and Biases (wandb) tool

The wandb tool was a fine addition to our project, assisting us in achieving improved accuracy. We extensively documented our code to replicate the same visualizations and graphs as showcased in our wandb project overview.

Illustration 13: Our wandb project overview

One particularly valuable graph is the hyperparameter Sweep graph, which provides valuable insights for future enhancements and optimizations. The Sweeps system consists of a controller and one or more agents. The controller selects new hyperparameter combination, while agents retrieve these hyperparameters from the Weights & Biases server to conduct model training. Agents have the flexibility to run multiple processes across multiple machines, enabling parallelization and scalability of Sweeps.

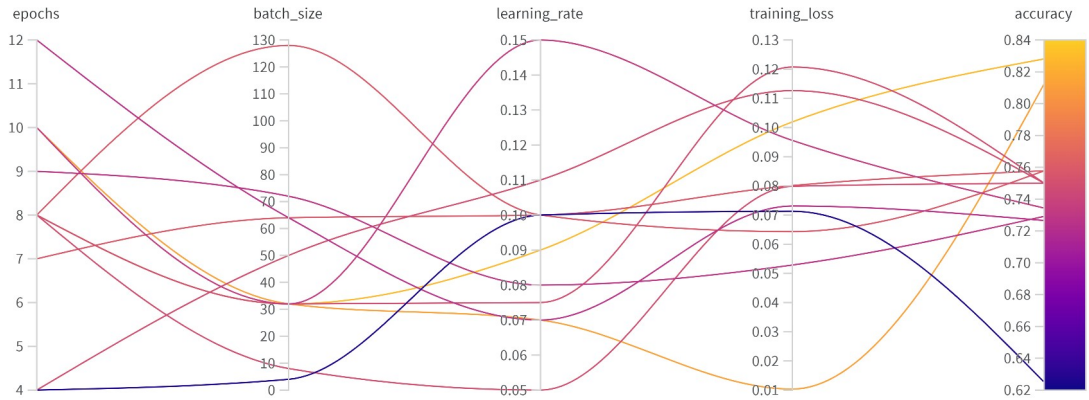

Illustration 14: Sweep from our Davinci fine-tuned models

Currently, our best performing Davinci model achieves an accuracy of 82,8% which is considered quite good. OpenAI's documentation indicates that achieving an accuracy of 85,5% for "Commonsense reasoning around everyday events" models (like ours), is a challenging threshold to surpass. These findings confirm our results and suggest that further research may not significantly enhance the accuracy beyond this established limit.

	GPT-4 Evaluated few-shot	GPT-3.5 Evaluated few-shot	LM SOTA Best external LM evaluated few-shot	SOTA Best external model (incl. benchmark-specific tuning)
MMLU [49] Multiple-choice questions in 57 subjects (professional & academic)	**86.4%** 5-shot	70.0% 5-shot	70.7% 5-shot U-PaLM [50]	75.2% 5-shot Flan-PaLM [51]
HellaSwag [52] Commonsense reasoning around everyday events	**95.3%** 10-shot	85.5% 10-shot	84.2% LLaMA (validation set) [28]	85.6 ALUM [53]
AI2 Reasoning Challenge (ARC) [54] Grade-school multiple choice science questions. Challenge-set.	**96.3%** 25-shot	85.2% 25-shot	85.2% 8-shot PaLM [55]	86.5% ST-MOE [18]
WinoGrande [56] Commonsense reasoning around pronoun resolution	**87.5%** 5-shot	81.6% 5-shot	85.1% 5-shot PaLM [3]	85.1% 5-shot PaLM [3]
HumanEval [43] Python coding tasks	**67.0%** 0-shot	48.1% 0-shot	26.2% 0-shot PaLM [3]	65.8% CodeT + GPT-3.5 [57]
DROP [58] (F1 score) Reading comprehension & arithmetic.	80.9 3-shot	64.1 3-shot	70.8 1-shot PaLM [3]	**88.4** QDGAT [59]
GSM-8K [60] Grade-school mathematics questions	**92.0%*** 5-shot chain-of-thought	57.1% 5-shot	58.8% 8-shot Minerva [61]	87.3% Chinchilla + SFT+ORM-RL, ORM

Illustration 15: Traditional benchmarks for machine learning OpenAI models

Conclusion

As a research project, it is important to acknowledge that the completion of this endeavor is an ongoing pursuit, as the possibilities for implementing additional features and incorporating emerging tools, particularly with OpenAI models, are constantly evolving. Presently, the project is fully functional, with the models generating coherent outputs. However, there is still room for improvement in terms of enhancing both accuracy and speed. This project served as a transformative entry point into the realm of Machine Learning, allowing me to explore and develop my skills as a data science student. By providing invaluable exposure to new tools and methodologies, this project has paved the way for my continued professional development.

Table of Illustrations