National College of Ireland

Configuration Manual

MSc Research Project
Artificial Intelligence for Business

Barbaros Sonmez
Student ID: x23169788

School of Computing
National College of Ireland

Supervisor: Faithful Onwuegbuche

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Barbaros Sonmez
Student ID:	x23169788
Programme:	Artificial Intelligence for Business
Year:	2024
Module:	MSc Research Project
Supervisor:	Faithful Onwuegbuche
Submission Due Date:	12/08/2024
Project Title:	Configuration Manual
Word Count:	1170
Page Count:	26

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	11th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Barbaros Sonmez
x23169788

1 Introduction

This configuration manual outlines the software and hardware requirements to reproduce the study. This guide defines the coding methods needed to reproduce this study, from setting up the environment to examining the model results.

2 Environmental Setup

This section provides a list of all the tools and software required for the effective execution of the project.

2.1 Hardware Requirements

The hardware specifications for this project includes a 64-bit Windows 10 operating system and 16 GB RAM. The processor is Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz 2.40 GHz.

2.2 Software Requirements

Development environment is Google Colab Pro (Paid) edition. Salesforce Developer Edition (Lightning) with limited storage. Hugging Face Development Platform with paid serverless endpoint service. Google Drive account connected to Google Colab. Python is the programming language that is utilised for the development of the models. APEx and javascript languages is used for Salesforce development operations.

3 Libraries

Required Installations:

!pip install command was used to install: datasets, transformers, seqeval, emoji, contractions. And !apt install command was used to install git-lfs

Required Imports:

numpy as np, pandas as pd, re, Path, matplotlib.pyplot as plt, WordCloud, STOPWORDS, import emoji, classification_report, roc_curve, auc, nltk, wordnet, WordNetLemmatizer, contractions, os, string, json, import drive, TfidfVectorizer, SVC, OneVsRestClassifier, accuracy_score, roc_auc_score, pipeline, RobertaForSequenceClassification, DistilBertForSequenceClassification, TFRobertaModel, RobertaTokenizer, DistilBertTokenizer, Trainer, TrainingArguments, Dataset, DatasetDict, Features, ClassLabel, Sequence,

Value, load_metric, torch, notebook_login, ElectraForSequenceClassification, ElectraTokenizer.

4 Dataset

The GoEmotion dataset contains a collection of 58k well chosen comments collected from Reddit, which include human classifications to 27 categories of emotions Demszky et al. (2020). The dataset was imported from a Hugging Face repo, then seperated train, validation, and test part, then assigned to different dataframes using the code in the Figure 1.

```
splits = {'train': 'simplified/train-00000-of-00001.parquet', 'validation': 'simplified/validation-00000-of-00001.parquet', 'test': 'simplified/test-00000-of-00001.parquet'}
df_train = pd.read_parquet("hf://datasets/google-research-datasets/go_emotions/" + splits["train"])
df_valid = pd.read_parquet("hf://datasets/google-research-datasets/go_emotions/" + splits["validation"])
df_test = pd.read_parquet("hf://datasets/google-research-datasets/go_emotions/" + splits["test"])

df_train = pd.DataFrame(df_train)
df_valid = pd.DataFrame(df_valid)
df_test = pd.DataFrame(df_test)
```

Figure 1: The Dataset Import.

5 Data Preprocessing

ekman_mapping.json and emotions.txt files were imported google colab using the code in the Figure 2.

```
[ ] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive

[ ] with open('/content/drive/MyDrive/GoEmotions/ekman_mapping.json') as file:
        ekman_mapping = json.load(file)

[ ] ekman_mapping

    Show hidden output

[ ] emotion_file = open("/content/drive/MyDrive/GoEmotions/emotions.txt", "r")
    emotion_list = emotion_file.read()
    emotion_list = emotion_list.split("\n")
```

Figure 2: The Necessary Files Import.

The dataset column label turned into emotion column using the code in the Figure 3.

The emotions was mapped using ekman_mapping list and mapped emotions column created using the code in the Figure 4.

The mapped emotions were employed to create target classes using the code in the Figure 5.

The unnecessary columns for the project were deleted classes using the code in the Figure 6.

```
def idx2class(idx_list):
    arr = []
    for i in idx_list:
        arr.append(emotion_list[int(i)])
    return arr
```

```
[ ] df_train['emotions'] = df_train['labels'].apply(idx2class)
    df_valid['emotions'] = df_valid['labels'].apply(idx2class)
    df_test['emotions'] = df_test['labels'].apply(idx2class)
```

Figure 3: The Label into Emotions Transformation.

```
def emotion_mapping(emotion_list):
    map_list = []

    for i in emotion_list:
        if i in ekman_mapping['anger']:
            map_list.append('anger')
        if i in ekman_mapping['disgust']:
            map_list.append('disgust')
        if i in ekman_mapping['fear']:
            map_list.append('fear')
        if i in ekman_mapping['joy']:
            map_list.append('joy')
        if i in ekman_mapping['sadness']:
            map_list.append('sadness')
        if i in ekman_mapping['surprise']:
            map_list.append('surprise')
        if i == 'neutral':
            map_list.append('neutral')

    return map_list
```

```
[ ] df_train['mapped emotions'] = df_train['emotions'].apply(emotion_mapping)
    df_valid['mapped emotions'] = df_valid['emotions'].apply(emotion_mapping)
    df_test['mapped emotions'] = df_test['emotions'].apply(emotion_mapping)
```

Figure 4: The Emotions into Mapped Emotions Transformation.

```
df_train['anger'] = np.zeros((len(df_train),1))
df_train['disgust'] = np.zeros((len(df_train),1))
df_train['fear'] = np.zeros((len(df_train),1))
df_train['joy'] = np.zeros((len(df_train),1))
df_train['sadness'] = np.zeros((len(df_train),1))
df_train['surprise'] = np.zeros((len(df_train),1))
df_train['neutral'] = np.zeros((len(df_train),1))

df_valid['anger'] = np.zeros((len(df_valid),1))
df_valid['disgust'] = np.zeros((len(df_valid),1))
df_valid['fear'] = np.zeros((len(df_valid),1))
df_valid['joy'] = np.zeros((len(df_valid),1))
df_valid['sadness'] = np.zeros((len(df_valid),1))
df_valid['surprise'] = np.zeros((len(df_valid),1))
df_valid['neutral'] = np.zeros((len(df_valid),1))

df_test['anger'] = np.zeros((len(df_test),1))
df_test['disgust'] = np.zeros((len(df_test),1))
df_test['fear'] = np.zeros((len(df_test),1))
df_test['joy'] = np.zeros((len(df_test),1))
df_test['sadness'] = np.zeros((len(df_test),1))
df_test['surprise'] = np.zeros((len(df_test),1))
df_test['neutral'] = np.zeros((len(df_test),1))
```

```
for i in ['anger', 'disgust', 'fear', 'joy', 'sadness', 'surprise','neutral']:
    df_train[i] = df_train['mapped emotions'].apply(lambda x: 1 if i in x else 0)
    df_valid[i] = df_valid['mapped emotions'].apply(lambda x: 1 if i in x else 0)
    df_test[i] = df_test['mapped emotions'].apply(lambda x: 1 if i in x else 0)
```

Figure 5: The Target Classes Creation.

```
df_train.drop(["labels", "id", "emotions", "mapped emotions", "neutral"], axis=1, inplace=True)
df_valid.drop(["labels", "id", "emotions", "mapped emotions", "neutral"], axis=1, inplace=True)
df_test.drop(["labels", "id", "emotions", "mapped emotions", "neutral"], axis=1, inplace=True)
```

Figure 6: The Unnnecessary Columns Deletion.

The different emoji patterns were created for effective emoji preprocessing using the code in the Figure 7.

```
# Precompile regular expressions
EMOJIS = sorted(emoji.EMOJI_DATA, key=len, reverse=True)
EMOJI_PATTERN = re.compile('|'.join(re.escape(u) for u in EMOJIS))
MULTI_EMOJI_PATTERN = re.compile(r"({})\1+".format(EMOJI_PATTERN.pattern))
REPEATED_CHAR_PATTERN = re.compile(r'(.)\1{2,}')

emoticon_dict = {
    ':)': '<smile_face>',
    ':-)': '<smile_face>',
    ':(': '<sad_face>',
    ':-(': '<sad_face>',
    ':D': '<big_smile>',
    ';)': '<wink>',
    ':-P': '<tongue_out>',
    ':/': '<unsure_face>',
    '<3': '<heart>',
    '¯_(ツ)_/¯': '<shrug>',
}
```

Figure 7: The Emoji Patterns Creation.

The created emoji patterns were used for emoji preprocessing using the code in the Figure 8.

Lowercase transformation, stop word removal, lemmatization, contraction expansion, improved tokenization using the code in the Figure 9.

After the preprocessing the first five raws of train dataframe are shown in the Figure 10.

6 EDA

The colour map creation and calculation of the target class percentage were performed using the code in the Figure 11.

The target classes distribution were calculated using the code in the Figure 12.

The target classes distribution is presented in the Figure 13.

The word clouds were created using the code in the Figure 14.

The word clouds are shown in the Figure 15.

The box plots were calculated using the code in the Figure 16.

The box plot of comment lengths by sentiment label provides a visual representation of the relationship between the length of the text and the sentiment expressed. The box plot shows the typical range of comment lengths associated with each sentiment, allowing to spot significant deviations or outliers. The box plot of comment lengths are shown in the below Figure 17.

7 SVM

The dataframes was transformed into SVM algorithm inputs using the code in the Figure 18.

```python
def preprocess_text(text):
    # Replace sequences of the same emoji with a single emoji
    text = MULTI_EMOJI_PATTERN.sub(r"\1", text)
    # Limit repeated characters to three repetitions
    text = REPEATED_CHAR_PATTERN.sub(r'\1\1\1', text)
    # Replace user mentions with a special token
    text = re.sub(r'u/\w+', '<user>', text)
    # Replace prices
    text = re.sub(r'\$\d{1,3}(,\d{3})*(\.\d+)?', '<PRICE>', text)
    # Replace times
    text = re.sub(r'\b\d{1,2}:\d{2}(:\d{2})?\b', '<TIME>', text)
    # Replace numbers except 1 or 2 digit integers
    text = re.sub(r'\d{1,3}(,\d{3})*(\.\d+)?|\d+(\.\d+)?',
                  lambda m: '<NUM>' if len(m.group(0))>2 else m.group(0), text)
    for emoticon, name in emoticon_dict.items():
        text = text.replace(emoticon, name)
    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text).strip()
    return text
```

```python
df_train['text'] = df_train['text'].apply(preprocess_text)
df_valid['text'] = df_valid['text'].apply(preprocess_text)
df_test['text'] = df_test['text'].apply(preprocess_text)
```

Figure 8: The Emoji Preprocessing.

```python
def preprocess(sentence):
    sentence = sentence.lower()
    stop_words = set(stopwords.words('english'))
    # initialize the WordNetLemmatizer
    lemmatizer = WordNetLemmatizer()
    sentence = re.sub('[^A-z]', ' ', sentence)
    negative = ['not', 'neither', 'nor', 'but', 'however', 'although', 'nonetheless', 'despite', 'except', 'even though', 'yet']
    stop_words = [z for z in stop_words if z not in negative]
    #lemmatization
    preprocessed_tokens = [lemmatizer.lemmatize(contractions.fix(temp.lower())) for temp in sentence.split() if temp not in stop_words]
    return ' '.join([x for x in preprocessed_tokens]).strip()

def text_preprocessing_pipeline(text):
    text = preprocess(text)
    return text
```

```python
df_train['text'] = df_train['text'].apply(lambda x: preprocess(x))
df_valid["text"] = df_valid["text"].apply(lambda x: preprocess(x))
df_test["text"] = df_test["text"].apply(text_preprocessing_pipeline)
```

Figure 9: The Text Preprocessing.

```python
df_train.head()
```

	text	anger	disgust	fear	joy	sadness	surprise
0	favourite food anything cook	0	0	0	0	0	0
1	everyone think he laugh screwing people instea...	0	0	0	0	0	0
2	fuck bayless isoing	1	0	0	0	0	0
3	make feel threatened	0	0	1	0	0	0
4	dirty southern wanker	1	0	0	0	0	0

Figure 10: The First Five Raws of Training Dataframe After Preprocessing.

```python
# Define a color map for each label
colors = {
    'anger': '#e6194b',
    'disgust': '#3cb44b',
    'fear': '#ffe119',
    'joy': '#0082c8',
    'sadness': '#911eb4',
    'surprise': '#f58231',
}
# Function to compute label counts for multi-label data
def compute_label_counts(ds):
    return ds.drop(columns=['text']).count(axis=0)
# Get the counts of each label in each set
train_counts = compute_label_counts(df_train)
val_counts = compute_label_counts(df_valid)
test_counts = compute_label_counts(df_test)
# Generate the pie charts
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
datasets = [
    (train_counts, "Train Set"),
    (val_counts, "Validation Set"),
    (test_counts, "Test Set")
]
```

Figure 11: The Dataframes Colour Map.

```python
for i, (counts, title) in enumerate(datasets):
    labels = counts.index
    sizes = counts.values
    colors_list = [colors[label] for label in labels]
    axs[i].pie(sizes, labels=labels, colors=colors_list, autopct='%1.1f%%', startangle=90)
    axs[i].axis('equal')
    axs[i].set_title(title)
    axs[i].text(0.5, -0.05, f'Total: {int(sum(sizes))}', size=10, ha='center', transform=axs[i].transAxes)

plt.tight_layout()
plt.show()
```

Figure 12: The Target Classes Distribution Calculation.

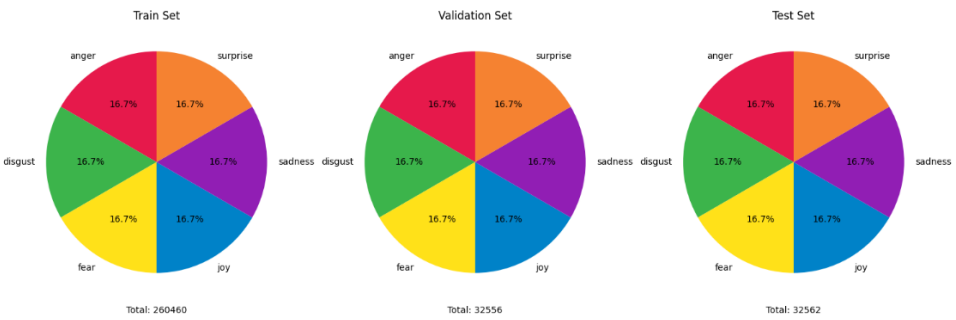

Figure 13: The Target Classes Distribution.

```python
# Given the multi-label nature, we need to iterate over each emotion column to get the text associated with it.
fig, axes = plt.subplots(2, 3, figsize=(18, 12))  # 2x3 grid of plots

for i, emotion in enumerate(emotions):
    ax = axes[i//3, i%3]

    # Filter out comments with the current emotion label
    emotion_comments = df_train[df_train[emotion] == 1]['text']

    # Concatenate all comments for the current emotion
    all_text = ' '.join(emotion_comments)

    # Generate word cloud
    wordcloud = WordCloud(stopwords=stopwords, width=800, height=800,
                          background_color='white', min_font_size=10).generate(all_text)

    ax.imshow(wordcloud, interpolation='bilinear')
    ax.axis('off')
    ax.set_title(emotion.upper(), fontsize=36, color='red', fontweight='bold')

plt.tight_layout()
plt.show()
```

Figure 14: The Word Clouds Creation.

Figure 15: The Word Clouds.

```python
# Melt the DataFrame to long format
melted_df = df_train.melt(id_vars=['text'],  # Remove 'new_length' as it's not in the DataFrame
                    value_vars=emotions,
                    var_name='emotion',
                    value_name='label_value')

# Filter rows where the label is 1
filtered_df = melted_df[melted_df['label_value'] == 1]

# Calculate the length of each comment
filtered_df['text_length'] = filtered_df['text'].apply(len)

# Plot boxplot using the new 'text_length' column
ax = filtered_df.boxplot(column='text_length', by='emotion', grid=False, showfliers=False)

plt.suptitle('Boxplot of Comment Lengths by Emotion', fontsize=16)
plt.title('')
plt.xlabel('Emotion', fontsize=12)
plt.ylabel('Comment Length', fontsize=12)
plt.xticks(rotation=45)
plt.show()
```

Figure 16: The Box Plots Calculation.

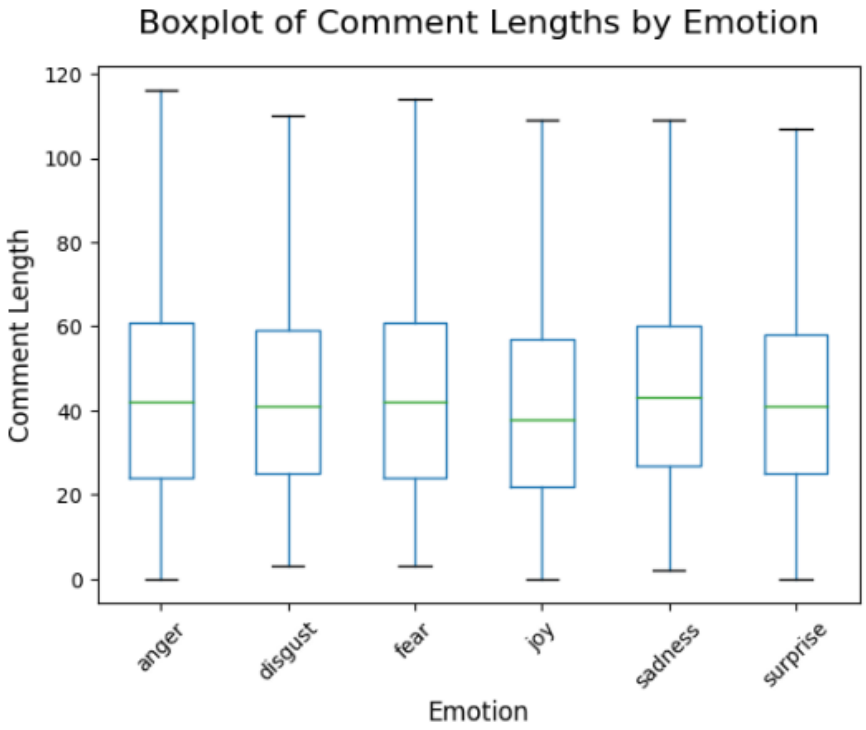

Figure 17: Box Plot of Comment Lengths.

```
X_train = df_train['text']
y_train = df_train[emotions]
X_valid = df_valid['text']
y_valid = df_valid[emotions]
X_test = df_test['text']
y_test = df_test[emotions]

# Tokenization and TF-IDF
vectorizer = TfidfVectorizer()
X_train_vec = vectorizer.fit_transform(X_train)
X_valid_vec = vectorizer.transform(X_valid)
X_test_vec = vectorizer.transform(X_test)
```

```
[ ] y_train = y_train.values
    y_valid = y_valid.values
    y_test = y_test.values
```

Figure 18: The Dataframes Transformation for SVM Input.

The target classes weight calculation and SVM algorithm employment was performed using the code in the Figure 19.

```
# Calculate class weights for each emotion separately
class_weight_dict = {}
for i, emotion in enumerate(emotions):
    class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(y_train[:, i]), y=y_train[:, i])
    class_weight_dict[emotion] = {i: weight for i, weight in enumerate(class_weights)}

# Create a list of class weights for each emotion
class_weights = [class_weight_dict[emotion] for emotion in emotions]

# Use a list comprehension to create a list of SVC instances with separate class weights for each emotion
estimators = [SVC(kernel='linear', class_weight=class_weights[i]) for i in range(len(emotions))]

# Create a OneVsRestClassifier with the list of estimators
# Older versions of scikit-learn use estimator= not estimators=
model = OneVsRestClassifier(estimator=SVC(kernel='linear')) # Pass a single estimator
model.fit(X_train_vec, y_train)
```

Figure 19: The Class Weight Calculation and SVM Employment.

The performance scores calculation code and performance scores of SVM are described in the below Figure 20.

The ROC graph of SVM was plotted using the code in the Figure 21.

The ROC graph of SVM algorithm is presented in Figure 22.

The micro avg ROC graph of SVM was plotted using the code in the Figure 23.

The micro avg ROC graph of SVM algorithm is described in Figure 24.

```
# Predict and Evaluate
y_pred = model.predict(X_test_vec)
print("SVM Performance as a Baseline:")
print(classification_report(df_test[emotions], y_pred, target_names=emotions))
```

```
SVM Performance as a Baseline:
              precision    recall  f1-score   support

       anger       0.74      0.20      0.31       726
     disgust       0.82      0.22      0.35       123
        fear       0.86      0.38      0.52        98
         joy       0.84      0.71      0.77      2104
     sadness       0.73      0.36      0.48       379
    surprise       0.72      0.12      0.21       677

   micro avg       0.82      0.47      0.59      4107
   macro avg       0.78      0.33      0.44      4107
weighted avg       0.79      0.47      0.55      4107
 samples avg       0.34      0.34      0.34      4107
```

Figure 20: Box Plot of Comment Lengths.

```
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")

# Calculate AUC for each class
auc_scores = roc_auc_score(y_test, y_pred, average=None)

# Plot AUC curve for each class
plt.figure(figsize=(10, 6))
for i, emotion in enumerate(emotions):
  fpr, tpr, _ = roc_curve(y_test[:, i], y_pred[:, i])
  roc_auc = auc(fpr, tpr)
  plt.plot(fpr, tpr, label=f'{emotion} (AUC = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

Figure 21: The ROC Graph Plot Code of SVM.

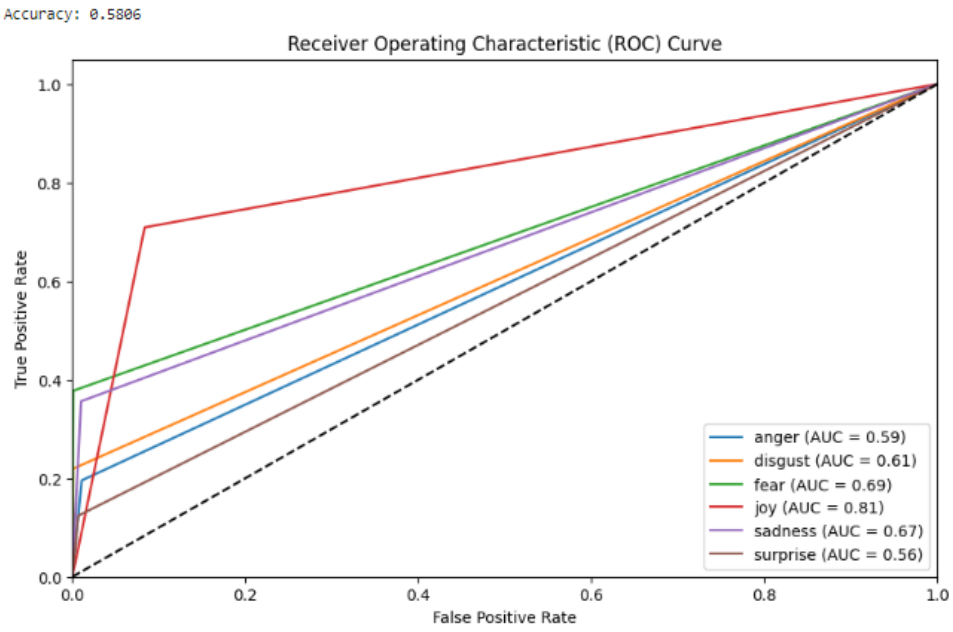

Figure 22: The ROC Graph of SVM.

```
# Calculate micro-average AUC
micro_avg_auc = roc_auc_score(y_test, y_pred, average='micro')

# Plot micro-average AUC curve
plt.figure(figsize=(10, 6))
fpr, tpr, _ = roc_curve(y_test.ravel(), y_pred.ravel())
plt.plot(fpr, tpr, label=f'Micro-average (AUC = {micro_avg_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Micro-average Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

Figure 23: The Micro AVG ROC Graph Plot Code of SVM.

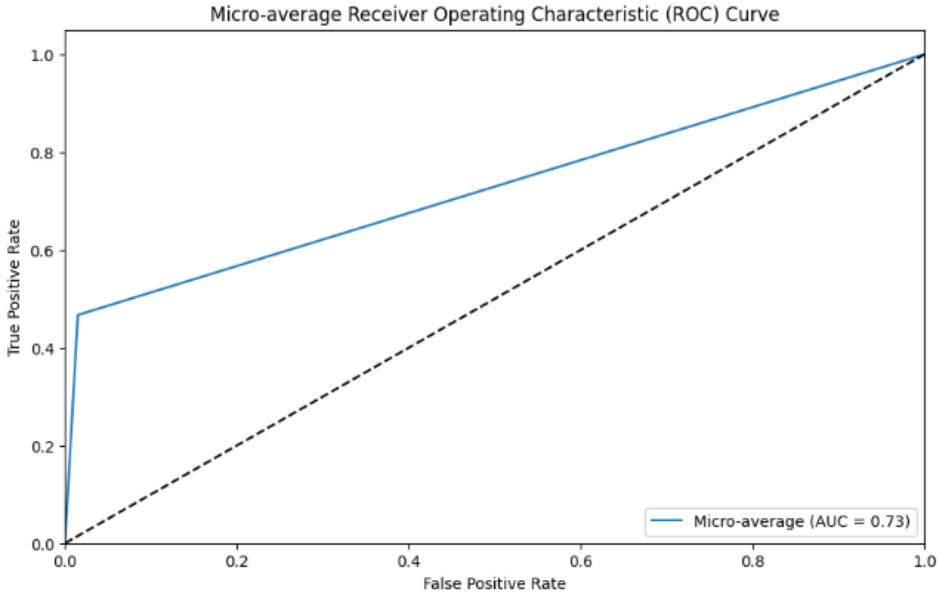

Figure 24: The Micro AVG ROC Graph of SVM.

8 RoBERTa

The RoBERTa tokenizer and dataset creation was done using the code in the Figure 25.

```
[ ]  # Load the tokenizer
     tokenizer = RobertaTokenizer.from_pretrained('distilroberta-base')
```

```
tokenizer_config.json: 100%  25.0/25.0 [00:00<00:00, 2.20kB/s]
vocab.json: 100%  899k/899k [00:00<00:00, 1.31MB/s]
merges.txt: 100%  456k/456k [00:00<00:00, 878kB/s]
tokenizer.json: 100%  1.36M/1.36M [00:00<00:00, 1.94MB/s]
config.json: 100%  480/480 [00:00<00:00, 41.2kB/s]
```

```
# Tokenization function
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", max_length=128, truncation=True)
```

```
[ ]  # Convert dataframes to datasets
     train_dataset = Dataset.from_pandas(df_train)
     val_dataset = Dataset.from_pandas(df_valid)
     test_dataset = Dataset.from_pandas(df_test)
```

Figure 25: The RoBERTa Tokenizer and Dataset Creation Code.

The RoBERTa tokenization was applied to the datasets using the code in the Figure 26.

It is neccessary to transform label classes into numpy arrays. The label classes were formatted for multi-label classification using the code in the Figure 27.

The column names were deleted, the datasets were transformed into torch format, the model was created and model configuration was set to multi label classification using the code in the Figure 28.

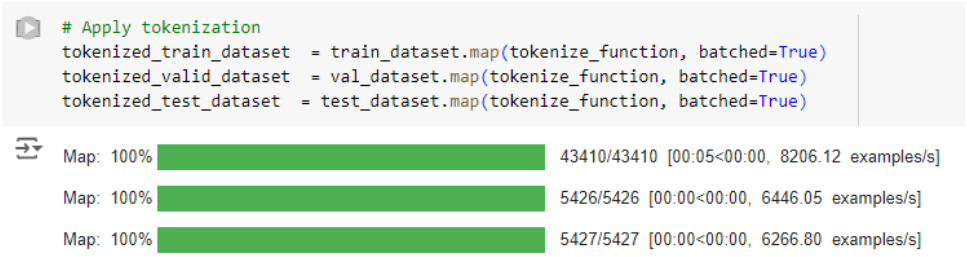

Figure 26: The RoBERTa Tokenization Application to Dataset.

Figure 27: The Label Classes Transformation for RoBERTa.

Figure 28: The Datasets Formatting and Model Creation Code.

The training args and compute metric function were defined using the code in the Figure 29.

```python
# Define training arguments
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=4,
    weight_decay=0.01,
    fp16=True
)
```

```python
# Define the metric
def compute_metrics(p):
    metric = load_metric("accuracy")
    predictions = torch.sigmoid(torch.tensor(p.predictions)).numpy()
    preds = (predictions > 0.5).astype(int)
    # Convert predictions and references to lists of single values
    preds_flat = preds.flatten().tolist()  # Flatten and convert to list
    references_flat = p.label_ids.flatten().tolist()
    return metric.compute(predictions=preds_flat, references=references_flat)
```

Figure 29: The Training Args and Compute Function Code.

The training was performed using the code in the Figure 30.

```python
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset_final,
    eval_dataset=val_dataset_final,
    compute_metrics=compute_metrics,
    tokenizer=tokenizer,
)
```

```python
# Train the model
trainer.train()
```

[10856/10856 16:27, Epoch 4/4]

Epoch	Training Loss	Validation Loss	Accuracy
1	0.251900	0.240167	0.912858
2	0.219100	0.227568	0.916114
3	0.195300	0.222929	0.916022
4	0.181300	0.230773	0.914916

```
TrainOutput(global_step=10856, training_loss=0.21863051803127842, metrics={'train_runtime': 987.7993, 'train_samples_per_second': 175.785,
'train_steps_per_second': 10.99, 'total_flos': 5750819977236480.0, 'train_loss': 0.21863051803127842, 'epoch': 4.0})
```

Figure 30: The Training of RoBERTa.

The evaluation was performed using the code in the Figure 31.

The RoBERTa ROC graph was plotted using the code in the Figure 32.

The ROC graph of RoBERTa algorithm is shown in Figure 33.

The RoBERTa accuracy scores for each label was plotted using the code in the Figure 34.

The accuracy scores of RoBERTa algorithm is presented in Figure 35.

The model was saved to Google Drive and pushed to Hugging Face Hub for further Endpoint creation using the code in the Figure 36.

```
# Evaluate the model
results = trainer.evaluate(test_dataset_final)
print(results)
```

```
[340/340 00:06]
{'eval_loss': 0.23452453315258026, 'eval_accuracy': 0.9153307536392113, 'eval_runtime': 7.6149,
```

Figure 31: The Evaluation of RoBERTa.

```
# Calculate ROC AUC and plot ROC curves
label_names = ['anger', 'disgust', 'fear', 'joy', 'sadness', 'surprise']
num_labels = len(label_names)
fpr = {}
tpr = {}
roc_auc = {}
```

```
[ ]  plt.figure(figsize=(12, 8))

     for i, label_name in enumerate(label_names):
         fpr[i], tpr[i], _ = roc_curve(labels[:, i], predictions[:, i])
         roc_auc[i] = roc_auc_score(labels[:, i], predictions[:, i])
         plt.plot(fpr[i], tpr[i], label=f'{label_name} (AUC = {roc_auc[i]:.2f})')

     plt.plot([0, 1], [0, 1], 'k--')
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.title('Receiver Operating Characteristic (ROC) Curve for Multi-Label Classification')
     plt.legend(loc='lower right')
     plt.show()
```

Figure 32: The ROC Graph Plotting of RoBERTa.

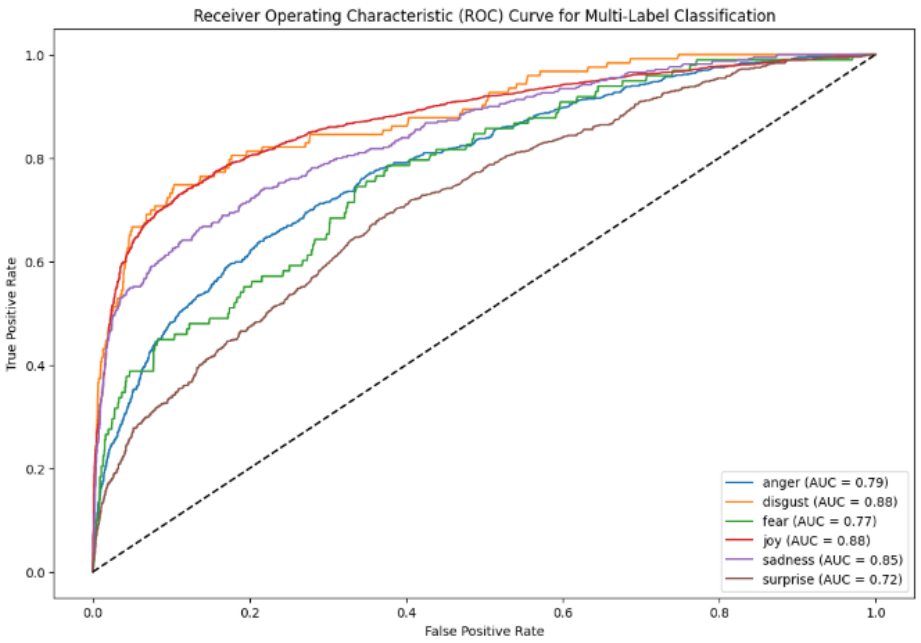

Figure 33: The ROC Graph of RoBERTa.

```
# Calculate AUC for each label
auc_scores = []
for i in range(labels.shape[1]):
    auc = roc_auc_score(labels[:, i], predictions[:, i])
    auc_scores.append(auc)
```

```
# Calculate the average AUC
average_auc = np.mean(auc_scores)
print(f"Average AUC: {average_auc}")
```

```
Average AUC: 0.8465447314513376
```

```
# Plot AUC scores
plt.figure(figsize=(10, 5))
plt.bar(range(len(auc_scores)), auc_scores, tick_label=['anger', 'disgust', 'fear', 'joy', 'sadness', 'surprise'])
plt.xlabel('Labels')
plt.ylabel('AUC Score')
plt.title('AUC Score per Label')
plt.show()
```

Figure 34: The Accuracy Scores for Each Label of RoBERTa.

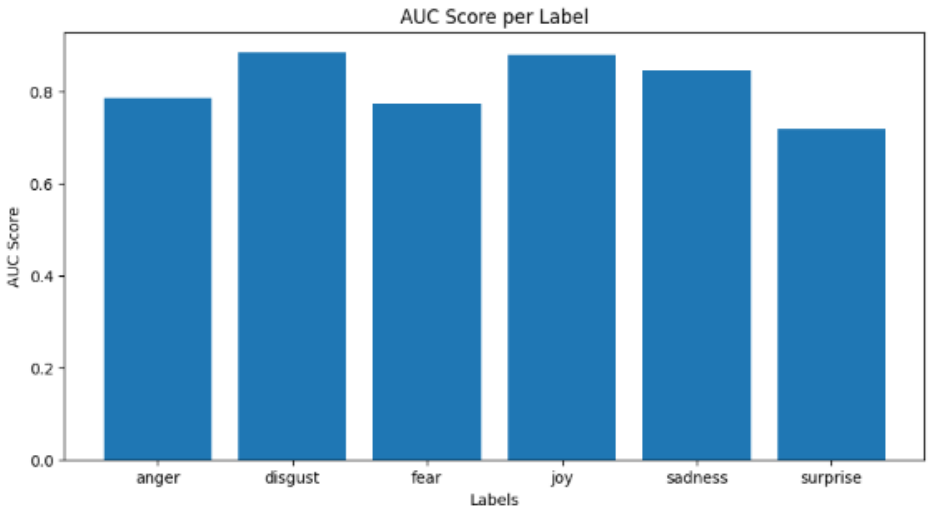

Figure 35: The Accuracy Scores of RoBERTa.

```
# Save model to drive
trainer.save_model("/content/drive/My Drive/RobertaModel")
```

```
# Replace these with your actual model and tokenizer
model = trainer.model
tokenizer = trainer.tokenizer
```

```
# Choose a model name for the Hugging Face Hub
model_name = "roberta-emotion"
```

```
# Push the model and tokenizer to the Hub
model.push_to_hub(model_name)
tokenizer.push_to_hub(model_name)
```

```
model.safetensors: 100% 329M/329M [00:24<00:00, 21.4MB/s]
README.md: 100% 5.17k/5.17k [00:00<00:00, 446kB/s]
CommitInfo(commit_url='https://huggingface.co/barx2os/roberta-emotion/commit/ca3153133681faf8bb4cd55a9144c37c73937e70', commit_message='Upload tokenizer',
commit_description='', oid='ca3153133681faf8bb4cd55a9144c37c73937e70', pr_url=None, pr_revision=None, pr_num=None)
```

Figure 36: The RoBERTa Model Save and Push Code.

9 Electra

The Electra tokenizer and model was created and model configuration was set to multi label classification using the code in the Figure 37.

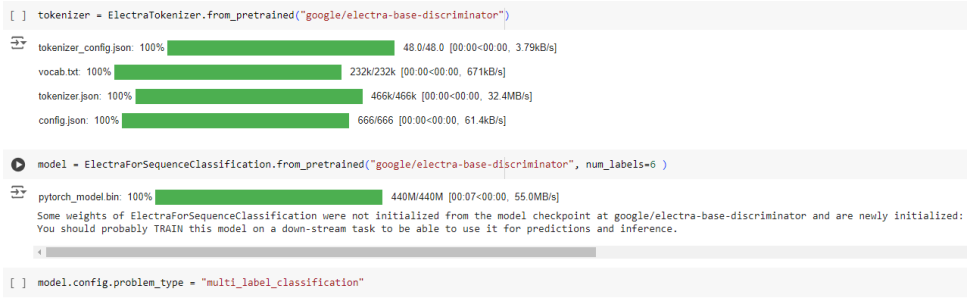

Figure 37: The Electra Tokenizer and Model Creation.

The datasets were created and tokenization was employed to the datasets using the code in the Figure 38.

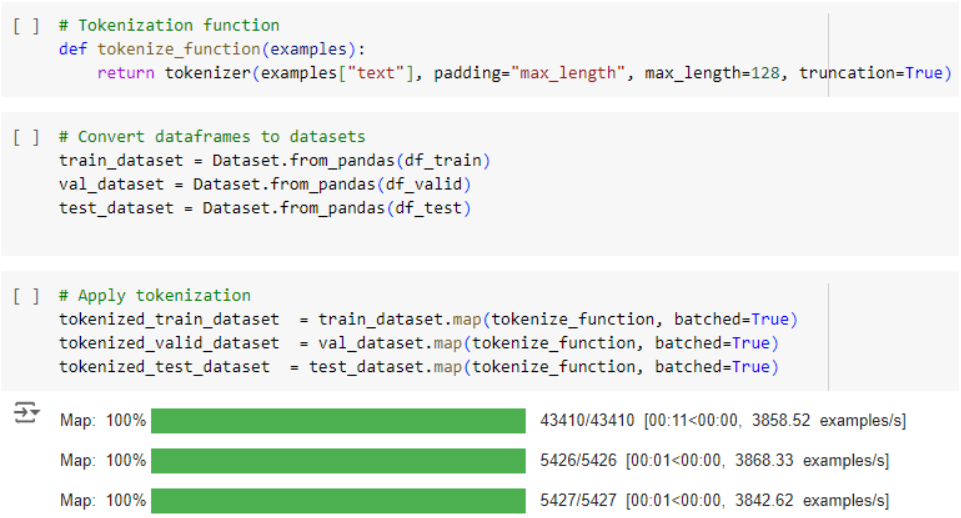

Figure 38: The Datasets Creation and Tokenization.

It is neccessary to transform label classes into numpy arrays. The label classes were formatted for multi-label classification using the code in the Figure 39.

The column names were deleted, the datasets were transformed into torch format using the code in the Figure 40.

The training args and compute metric function were defined using the code in the Figure 41.

The training was performed using the code in the Figure 42.

The evaluation was performed using the code in the Figure 43.

The Electra ROC graph was plotted using the code in the Figure 44.

The ROC graph of Electra algorithm is shown in Figure 45.

The Electra accuracy scores for each label was plotted using the code in the Figure 46.

The accuracy scores of Electra algorithm is presented in Figure 47.

```
def format_labels(examples):
    labels = np.array([
        [
            float(examples["anger"][i]),
            float(examples["disgust"][i]),
            float(examples["fear"][i]),
            float(examples["joy"][i]),
            float(examples["sadness"][i]),
            float(examples["surprise"][i])
        ]
        for i in range(len(examples["text"]))
    ])
    return {"labels": labels}
```

```
[ ] # Apply the format_labels function to the datasets
    formatted_train_dataset = tokenized_train_dataset.map(format_labels, batched=True)
    formatted_valid_dataset = tokenized_valid_dataset.map(format_labels, batched=True)
    formatted_test_dataset = tokenized_test_dataset.map(format_labels, batched=True)
```

```
Map: 100% 43410/43410 [00:00<00:00, 104014.89 examples/s]
Map: 100% 5426/5426 [00:00<00:00, 81509.89 examples/s]
Map: 100% 5427/5427 [00:00<00:00, 83364.72 examples/s]
```

Figure 39: The Label Classes Transformation for Electra.

```
# Remove columns that are not needed
train_dataset_final = formatted_train_dataset.remove_columns(['text', 'anger', 'disgust', 'fear', 'joy', 'sadness', 'surprise'])
val_dataset_final = formatted_valid_dataset.remove_columns(['text', 'anger', 'disgust', 'fear', 'joy', 'sadness', 'surprise'])
test_dataset_final = formatted_test_dataset.remove_columns(['text', 'anger', 'disgust', 'fear', 'joy', 'sadness', 'surprise'])
```

```
[ ] # Set the format for PyTorch
    train_dataset_final.set_format(type='torch')
    val_dataset_final.set_format(type='torch')
    test_dataset_final.set_format(type='torch')
```

Figure 40: The Datasets Formatting for Electra.

```
# Define training arguments
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=4,
    weight_decay=0.01,
    fp16=True
)
```

```
[ ] # Define the metric
    def compute_metrics(p):
        metric = load_metric("accuracy")
        predictions = torch.sigmoid(torch.tensor(p.predictions)).numpy()
        preds = (predictions > 0.5).astype(int)
        # Convert predictions and references to lists of single values
        preds_flat = preds.flatten().tolist()  # Flatten and convert to list
        references_flat = p.label_ids.flatten().tolist()
        return metric.compute(predictions=preds_flat, references=references_flat)
```

Figure 41: The Training Args and Compute Function Code.

Figure 42: The Training of Electra.

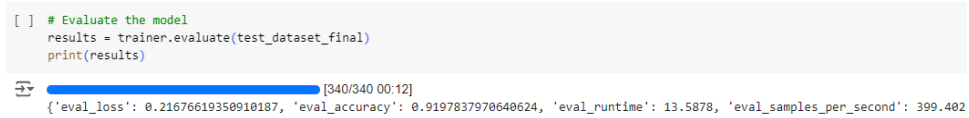

Figure 43: The Evaluation of Electra.

```
[ ]  # Calculate ROC AUC and plot ROC curves
     label_names = ['anger', 'disgust', 'fear', 'joy', 'sadness', 'surprise']
     num_labels = len(label_names)
     fpr = {}
     tpr = {}
     roc_auc = {}
```

```
     plt.figure(figsize=(12, 8))

     for i, label_name in enumerate(label_names):
         fpr[i], tpr[i], _ = roc_curve(labels[:, i], predictions[:, i])
         roc_auc[i] = roc_auc_score(labels[:, i], predictions[:, i])
         plt.plot(fpr[i], tpr[i], label=f'{label_name} (AUC = {roc_auc[i]:.2f})')

     plt.plot([0, 1], [0, 1], 'k--')
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.title('Receiver Operating Characteristic (ROC) Curve for Multi-Label Classification')
     plt.legend(loc='lower right')
     plt.show()
```

Figure 44: The ROC Graph Plotting of Electra.

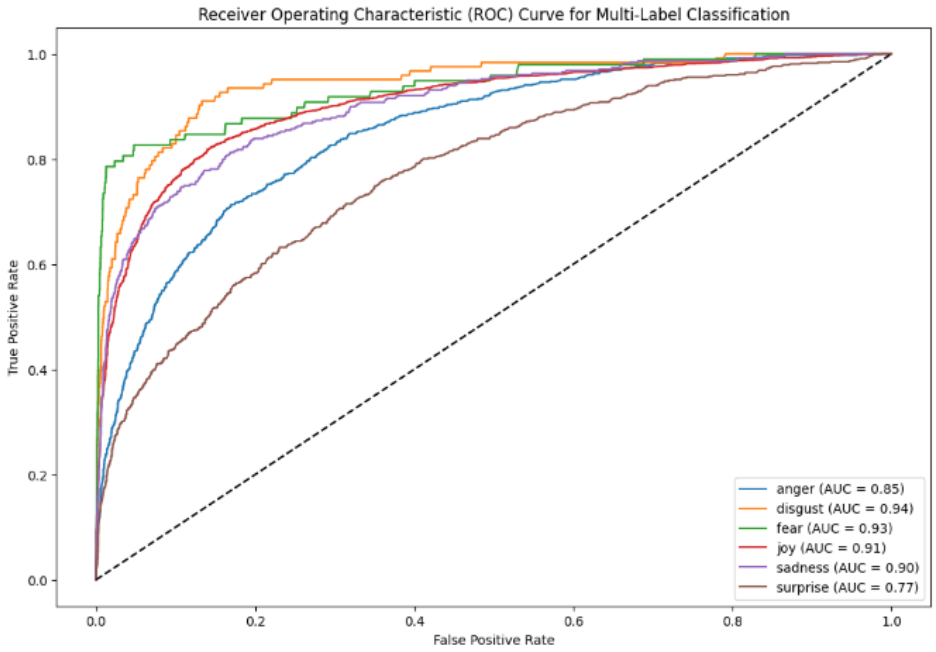

Figure 45: The ROC Graph of Electra.

```
# Calculate AUC for each label
auc_scores = []
for i in range(labels.shape[1]):
    auc = roc_auc_score(labels[:, i], predictions[:, i])
    auc_scores.append(auc)
```

```
# Calculate the average AUC
average_auc = np.mean(auc_scores)
print(f"Average AUC: {average_auc}")
```

```
Average AUC: 0.8852423354650941
```

```
# Plot AUC scores
plt.figure(figsize=(10, 5))
plt.bar(range(len(auc_scores)), auc_scores, tick_label=['anger', 'disgust', 'fear', 'joy', 'sadness', 'surprise'])
plt.xlabel('Labels')
plt.ylabel('AUC Score')
plt.title('AUC Score per Label')
plt.show()
```

Figure 46: The Accuracy Scores for Each Label of Electra.

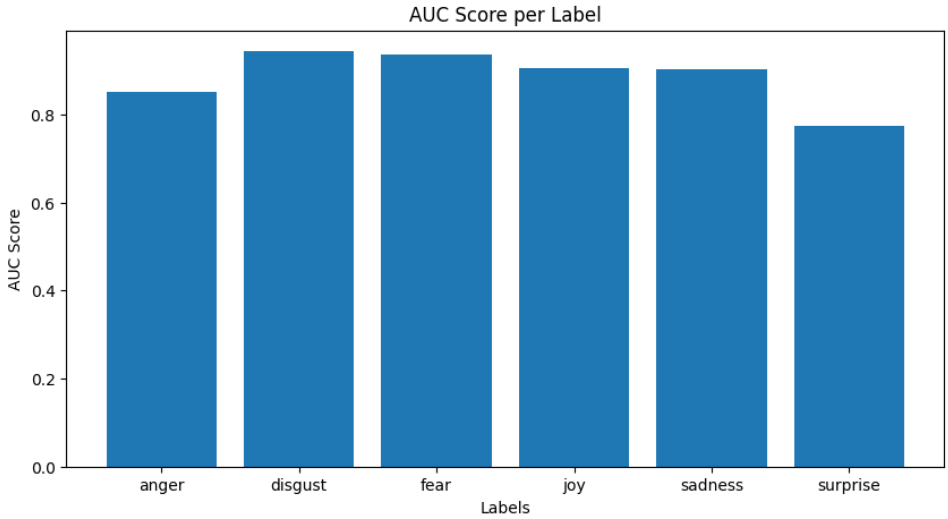

Figure 47: The Accuracy Scores of Electra.

The model was saved to Google Drive and pushed to Hugging Face Hub for further Endpoint creation using the code in the Figure 48.

```
# Save model to drive
trainer.save_model("/content/drive/My Drive/ElectraModel")

# Replace these with your actual model and tokenizer
model = trainer.model
tokenizer = trainer.tokenizer

# Choose a model name for the Hugging Face Hub
model_name = "electra-emotion"

# Push the model and tokenizer to the Hub
model.push_to_hub(model_name)
tokenizer.push_to_hub(model_name)
```

```
model.safetensors: 100% 438M/438M [00:24<00:00, 36.4MB/s]
README.md: 100% 5.17k/5.17k [00:00<00:00, 397kB/s]
CommitInfo(commit_url='https://huggingface.co/barx2os/electra-emotion/commit/d2573e6f3ff7aa3e89e263a208ed559d0c55fd48', commit_message='Upload tokenizer',
commit_description='', oid='d2573e6f3ff7aa3e89e263a208ed559d0c55fd48', pr_url=None, pr_revision=None, pr_num=None)
```

Figure 48: The Electra Model Save and Push Code.

10 AI API and Salesforce Configuration

The Hugging Face Endpoint Configuration is presented in Figure 49.

For configuring Salesforce, an UI was created, UI is presented in Figure 50.

A button (Sentiment Email) was created in UI for triggering a flow action. The flow action executes a script (SentimentEmailFlowController.apxc). The script is shown in Figure 51.

SentimentEmailFlowController uses a utility class (SentimentAnalysisUtil.apxc) for the sake of modular code structure. Util class includes all functionality such as call out methods, email sending methods, tailored email responses, and parsing JSON . The script is shown in Figure 52, Figure 53, Figure 54, and Figure 55.

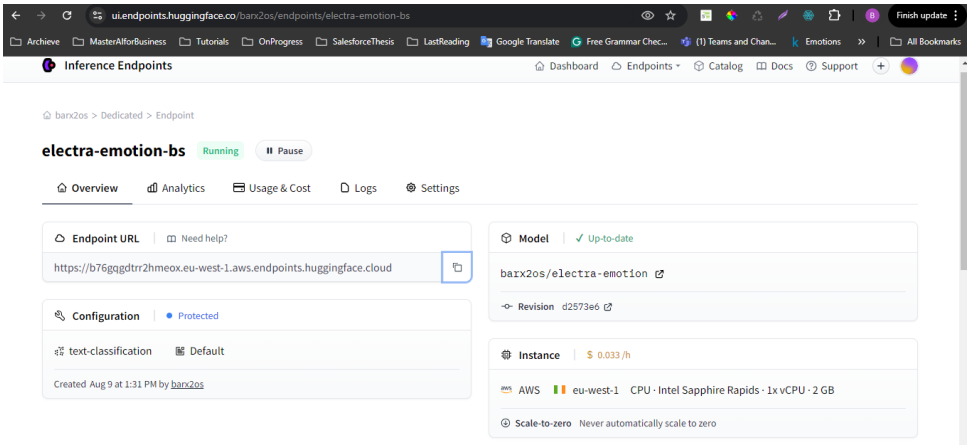

Figure 49: The Endpoint Configuration.

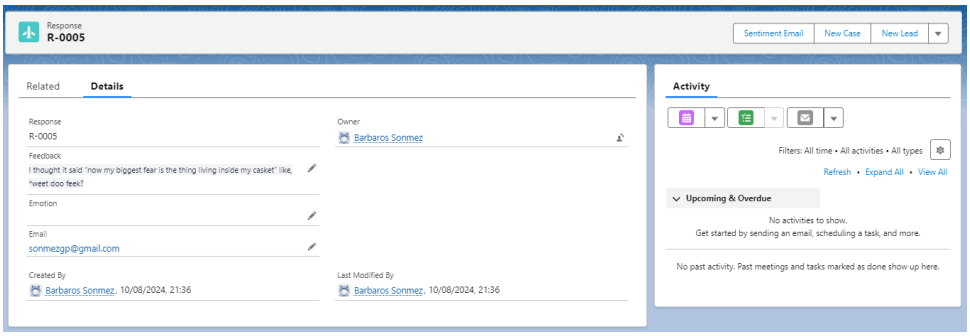

Figure 50: The Salesforce UI.

```
public with sharing class SentimentEmailFlowController {

    @InvocableMethod(label='Analyze Sentiment and Send Email' description='Analyze sentiment and send an email based on feedback.')
    public static void analyzeAndSend(List<Id> responseIds) {
        for (Id responseId : responseIds) {
            Response__c responseRecord = [SELECT Id, Feedback__c, Email__c, Emotion__c FROM Response__c WHERE Id = :responseId LIMIT 1];

            // Create an instance of the SentimentAnalysisUtil class
            SentimentAnalysisUtil util = new SentimentAnalysisUtil();

            // Extract fields from the retrieved record
            String feedbackText = util.stripHtmlTags(responseRecord.Feedback__c);
            String email = responseRecord.Email__c;

            // Get the emotion from the feedback text
            String emotion = util.getEmotion(feedbackText);
            System.debug('Emotion to be set: ' + emotion);

            // Update the emotion field on the feedback record
            responseRecord.Emotion__c = emotion;
            System.debug('Setting Emotion__c to: ' + responseRecord.Emotion__c);

            try {
                update responseRecord;
                System.debug('Successfully updated Emotion__c.');
            } catch (DmlException e) {
                System.debug('Error updating Emotion__c: ' + e.getMessage());
            }

            // Send an email based on the detected emotion
            util.sendEmotionEmail(email, emotion);
        }
    }
}
```

Figure 51: The SentimentEmailFlowController Script.

```
public class SentimentAnalysisUtil {

    // Define your specific API endpoint and token
    private static final String API_ENDPOINT = 'https://b76gqgdtrr2hmeox.eu-west-1.aws.endpoints.huggingface.cloud';
    private static final String API_TOKEN = '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';

    // Mapping labels from the model to actual emotion names
    private static final Map<String, String> LABEL_MAP = new Map<String, String>{
        'LABEL_0' => 'anger',
        'LABEL_1' => 'disgust',
        'LABEL_2' => 'fear',
        'LABEL_3' => 'joy',
        'LABEL_4' => 'sadness',
        'LABEL_5' => 'surprise'
    };

    public String getEmotion(String feedbackText) {
    String emotion = 'Unknown';
    HttpRequest req = new HttpRequest();
    req.setEndpoint(API_ENDPOINT);
    req.setMethod('POST');
    req.setHeader('Authorization', 'Bearer ' + API_TOKEN);
    req.setHeader('Content-Type', 'application/json');
    req.setHeader('Accept', 'application/json'); // Set the correct Accept header

    // Create JSON request body
    String requestBody = '{"inputs": "' + feedbackText + '"}';
    System.debug('Request body: ' + requestBody);
    req.setBody(requestBody);
```

Figure 52: The SentimentAnalysisUtil Script Part-1.

```
    // Send HTTP request
    Http http = new Http();
    try {
        HttpResponse res = http.send(req);
        System.debug('Request sent');
        System.debug('Response: ' + res.getBody()); // Log the full response for debugging

        if (res.getStatusCode() == 200) {
            // Process the response to extract the label and score
            List<Object> responseList = (List<Object>) JSON.deserializeUntyped(res.getBody());
            if (!responseList.isEmpty()) {
                Map<String, Object> responseMap = (Map<String, Object>) responseList.get(0); // Use get(0) instead of [0]
                System.debug('Response Map: ' + responseMap);

                if (responseMap.containsKey('label')) {
                    String label = (String) responseMap.get('label');
                    System.debug('Label received: ' + label);
                    if (LABEL_MAP.containsKey(label)) {  // Check if the label exists in LABEL_MAP
                        emotion = LABEL_MAP.get(label);
                        System.debug('Mapped emotion: ' + emotion);
                    } else {
                        System.debug('Label not found in LABEL_MAP: ' + label);
                    }
                } else {
                    System.debug('Response map does not contain a "label" key');
                }
            } else {
                System.debug('Response list is empty');
            }
        } else {
            System.debug('Failed with status code: ' + res.getStatusCode());
        }
    } catch (Exception e) {
        System.debug('Error calling Hugging Face API: ' + e.getMessage());
    }
```

Figure 53: The SentimentAnalysisUtil Script Part-2.

```
    System.debug('Final emotion: ' + emotion);
    return emotion;
}

    public void sendEmotionEmail(String email, String emotion) {
        if (email != null && email.contains('@')) {
            String subject = 'Your Feedback Emotion Analysis';
            String body = '';

            // Create email body based on the emotion detected
            switch on emotion {
                when 'anger' {
                    body = 'Thank you for your feedback. We noticed that your feedback expressed anger. We are sorry for any inconvenience and are here to help!';
                }
                when 'disgust' {
                    body = 'Thank you for your feedback. We noticed that your feedback expressed disgust. We value your opinion and will work to improve!';
                }
                when 'fear' {
                    body = 'Thank you for your feedback. We noticed that your feedback expressed fear. Please let us know how we can assist you!';
                }
                when 'joy' {
                    body = 'Thank you for your feedback. We noticed that your feedback expressed joy. We are glad you had a positive experience!';
                }
                when 'sadness' {
                    body = 'Thank you for your feedback. We noticed that your feedback expressed sadness. We are here to support you!';
                }
                when 'surprise' {
                    body = 'Thank you for your feedback. We noticed that your feedback expressed surprise. We hope to continue surprising you in positive ways!';
                }
                when else {
                    body = 'Thank you for your feedback. We appreciate your input!';
                }
            }
```

Figure 54: The SentimentAnalysisUtil Script Part-3.

```
            // Send email
            Messaging.SingleEmailMessage emailMessage = new Messaging.SingleEmailMessage();
            emailMessage.setToAddresses(new String[] { email });
            emailMessage.setSubject(subject);
            emailMessage.setPlainTextBody(body);
            Messaging.sendEmail(new Messaging.SingleEmailMessage[] { emailMessage });
        }
    }

    public String stripHtmlTags(String html) {
        if (String.isNotEmpty(html)) {
            // Regular expression to remove HTML tags
            return html.replaceAll('<[^>]*>', '');
        }
        return html;
    }
}
```

Figure 55: The SentimentAnalysisUtil Script Part-4.

References

Demszky, D., Movshovitz-Attias, D., Ko, J., Cowen, A., Nemade, G. and Ravi, S. (2020). GoEmotions: A Dataset of Fine-Grained Emotions, *58th Annual Meeting of the Association for Computational Linguistics (ACL)*.