National College of Ireland

Configuration Manual

MSc Research Project
Artificial Intelligence

Kiymet Elif Ari
Student ID: 23111488

School of Computing
National College of Ireland

Supervisor: Sheresh Zahoor

National College of Ireland
MSc Project Submission Sheet
School of Computing

Student Name:	Kiymet Elif Ari		
Student ID:	23111488		
Programme:	MSC Artificial Intelligence	**Year:**	2023 - 2024
Module:	Research Practicum		
Supervisor:	Sheresh Zahoor		
Submission Due Date:	12 August 2024		
Project Title:	Optimizing Job Recommendation Systems with AI: A Deep Dive into BERT and GPT Models		
Word Count:	420	**Page Count:**	5

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: Kiymet Elif Ari

Date: 06.08.2024

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

Kiymet Elif Ari
23111488

1. Introduction

This configuration manual provides the necessary steps to set up the environment and run the code for the research "Optimizing Job Recommendation Systems with AI: A Deep Dive into BERT and GPT Models". The research uses advanced NLP models like BERT and GPT to enhance job recommendation systems. The manual outlines the hardware and software requirements, data extraction process, required Python libraries, and instructions for running the code.

2. System Configuration

2.1. Hardware Requirements

The research is conducted at a local system with the specifications below.

1. Model Name: MacBook Pro
2. System OS: MacOS Sonoma (Version 14.5)
3. Processor: Apple M1
4. Memory: 8 GB

2.2. Software Requirements

1. Python 3.9: The code is written in Python 3.9.
2. Jupyter Notebook: Used for developing and running the project code.
3. Anaconda: Used for managing the Python environment.
4. Hugging Face Transformers: Library for NLP model implementation.
5. PyTorch: Deep learning framework used for model training and inference.

3. Data Extraction and Preparation

3.1. Dataset Description

The dataset used for this research consists of job postings scraped from an online platform. The dataset includes attributes below for Software Engineering job area.

```
 #   Column
---  ------
 0   company
 1   description
 2   descriptionHTML
 3   externalApplyLink
 4   id
 5   isExpired
 6   jobType
 7   jobType/0
 8   jobType/1
 9   jobType/2
 10  location
 11  positionName
 12  postedAt
 13  postingDateParsed
 14  rating
 15  reviewsCount
 16  salary
 17  scrapedAt
 18  searchInput/country
 19  searchInput/position
 20  url
 21  urlInput
```

Figure 1: Scrapped Dataset Attributes

3.2. Loading the Dataset

1. The provided CSV files are placed in a directory accessible by the Python environment.

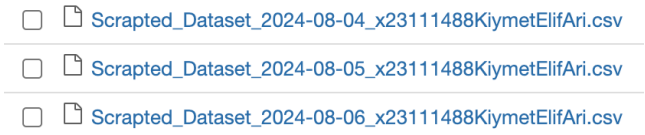

Figure 2: Provided CSV Files

2. The datasets is loaded using Pandas.

```
In [1]: import pandas as pd

        #Firstly, I loaded the dataset.
        file_path = 'Scrapted_Dataset_2024-08-04_x23111488KiymetElifAri.csv'
        data = pd.read_csv(file_path)

        print(data.head())
        print(data.info())
```

Figure 3: Data Loading

4. Required Python Libraries

The necessary Python libraries are installed using pip.

1. pandas
2. numpy
3. torch
4. transformers
5. scikit-learn
6. matplotlib
7. seaborn

5. Implementation

5.1. Data Preprocessing

1. Column Removal: Unnecessary columns are removed.
2. Handling Missing Values: Missing values in the salary column are filled with 'Not Provided'.
3. Text Normalization: Text columns are normalized by converting to lowercase and stripping excess whitespace.
4. Feature Encoding: One-hot encoding is applied to categorical features and numerical features.
5. Datetime Converting: Date columns are converted to datetime objects.

In [2]:
```
#There are unnecessary columns. I dropped them.
columns_to_drop = ['descriptionHTML', 'externalApplyLink', 'urlInput']
data.drop(columns=columns_to_drop, inplace=True)

#I filled missing 'salary' value.
data['salary'] = data['salary'].fillna('Not Provided')

#I normalized text data. I converted them to lowercase and stripped of excess whitespace.
text_columns = ['company', 'description', 'location', 'positionName']
for col in text_columns:
    data[col] = data[col].str.lower().str.strip()

#I simplified job types into a single column.
data['jobType'] = data.apply(lambda x: x['jobType/0'] if pd.notna(x['jobType/0']) else
                                       (x['jobType/1'] if pd.notna(x['jobType/1']) else
                                        x['jobType/2']), axis=1)
data.drop(columns=['jobType/0', 'jobType/1', 'jobType/2'], inplace=True)

#I converted 'postingDateParsed' and 'scrapedAt' to datetime objects.
data['postingDateParsed'] = pd.to_datetime(data['postingDateParsed'], errors='coerce')
data['scrapedAt'] = pd.to_datetime(data['scrapedAt'], errors='coerce')

#I checked the dataset after cleaning.
print(data.head())
print(data.info())
```

In [18]:
```
#Categorical and Numerical Feature Encoding

from sklearn.preprocessing import OneHotEncoder, LabelEncoder, StandardScaler
from sklearn.compose import ColumnTransformer

#I applied One-Hot Encoding for 'jobType'.
onehot_encoder = OneHotEncoder(sparse_output=False)
job_type_encoded = onehot_encoder.fit_transform(data[['jobType']])
job_type_encoded_df = pd.DataFrame(job_type_encoded, columns=onehot_encoder.get_feature_names_out(['jobType']))

#I applied Label Encoding for 'location'.
label_encoder = LabelEncoder()
data['location_encoding'] = label_encoder.fit_transform(data['location'])

#I scaled for 'rating' and 'reviewsCount'.
scaler = StandardScaler()
data[['rating', 'reviewsCount']] = scaler.fit_transform(data[['rating', 'reviewsCount']].fillna(0))

#I combined the one-hot encoded columns back into the main DataFrame.
data = pd.concat([data, job_type_encoded_df], axis=1)
data.drop(['jobType'], axis=1, inplace=True)

print(data.head())
```

Figure 4: Data Preprocessing Steps

5.2. Feature Engineering

A Job Desirability Score is calculated based on normalized ratings, review counts, and job type weights and it is converted into binary labels.

In [19]:
```python
#Conceptualizing the Job Desirability Score

import numpy as np

#I normalized 'rating' and 'reviewsCount'.
max_rating = data['rating'].max()
max_reviews = data['reviewsCount'].max()

data['norm_rating'] = data['rating'] / max_rating
data['norm_reviews'] = data['reviewsCount'] / max_reviews

#I assigned higher weights to full-time jobs.
data['jobType_weight'] = data['jobType_Full-time'] * 1.0  # Weight for full-time jobs
data['jobType_weight'] += data['jobType_Part-time'] * 0.5  # Lower weight for part-time jobs

#I calculated Job Desirability Score.
weights = {
    'norm_rating': 0.4,  # 40% weight
    'norm_reviews': 0.4,  # 40% weight
    'jobType_weight': 0.2  # 20% weight
}

data['job_desirability_score'] = (
    data['norm_rating'] * weights['norm_rating'] +
    data['norm_reviews'] * weights['norm_reviews'] +
    data['jobType_weight'] * weights['jobType_weight']
)

print(data[['norm_rating', 'norm_reviews', 'jobType_weight', 'job_desirability_score']].head())
```

In [39]:
```python
#I converted job_desirability_score into binary labels.
threshold = data['job_desirability_score'].median()
data['desirability_label'] = (data['job_desirability_score'] > threshold).astype(int)
```

Figure 5: Feature Engineering

5.3. Model Training

1. BERT: Model is trained, using the prepared dataset.

In [59]:
```python
import torch
from torch.utils.data import DataLoader, Dataset
from transformers import BertTokenizer, BertForSequenceClassification
from transformers import get_linear_schedule_with_warmup
from torch.optim import AdamW
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
import re

#I loaded tokenizer and model.
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
classification_model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
```

Figure 6: BERT Model Training

2. GPT: Model is trained, using the prepared dataset.

In [63]:
```python
import torch
from torch.utils.data import DataLoader, Dataset
from transformers import GPT2Tokenizer, GPT2ForSequenceClassification
from transformers import get_linear_schedule_with_warmup
from torch.optim import AdamW
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
import re
import numpy as np

#I loaded tokenizer and model.
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
classification_model = GPT2ForSequenceClassification.from_pretrained('gpt2', num_labels=2)

tokenizer.pad_token = tokenizer.eos_token
classification_model.config.pad_token_id = tokenizer.eos_token_id
```

Figure 7: GPT Model Training

6. Running The Code

1. Jupyter Notebook: Directory is opened, containing the notebook files provided.
2. Executing: Cells in the notebooks is run sequentially to preprocess data, train the models, and evaluate the results.
3. Saving: The trained models are saved for using again in the future.

```
    #Validation
    classification_model.eval()
    val_loss = 0
    val_preds = []
    val_labels_list = []
    with torch.no_grad():
        for batch in val_loader:
            texts, labels = batch

            labels = labels.to(torch.long)

            inputs = tokenizer(list(texts), return_tensors='pt', max_length=512, truncation=True, padding='max_lengt
            inputs = {key: value.to(device) for key, value in inputs.items()}
            labels = labels.to(device)

            outputs = classification_model(**inputs, labels=labels)
            loss = outputs.loss
            val_loss += loss.item()

            logits = outputs.logits
            preds = torch.argmax(logits, dim=1).cpu().numpy()
            val_preds.extend(preds)
            val_labels_list.extend(labels.cpu().numpy())

            #I printed predictions and actual labels.
            print(f"Predictions: {preds}")
            print(f"Actual Labels: {labels.cpu().numpy()}")

    avg_val_loss = val_loss / len(val_loader)
    val_accuracy = accuracy_score(val_labels_list, val_preds)
    val_f1 = f1_score(val_labels_list, val_preds)
    val_precision = precision_score(val_labels_list, val_preds, zero_division=1)
    val_recall = recall_score(val_labels_list, val_preds, zero_division=1)

    print(f'Epoch {epoch+1}/{epochs}, Validation Loss: {avg_val_loss}')
    print(f'Validation Accuracy: {val_accuracy}, F1 Score: {val_f1}, Precision: {val_precision}, Recall: {val_recall

#I saved the model and tokenizer.
classification_model.save_pretrained('./model_save_directory')
tokenizer.save_pretrained('./model_save_directory')
```

Figure 8: BERT Model Validation and Evaluation

```
    #Validation
    classification_model.eval()
    val_loss = 0
    val_preds = []
    val_labels_list = []
    with torch.no_grad():
        for batch in val_loader:
            texts, labels = batch

            #I ensured labels are tensors of type long.
            labels = labels.to(torch.long)

            inputs = tokenizer(list(texts), return_tensors='pt', max_length=512, truncation=True, padding=True)
            inputs = {key: value.to(device) for key, value in inputs.items()}
            labels = labels.to(device)

            outputs = classification_model(**inputs, labels=labels)
            loss = outputs.loss
            val_loss += loss.item()

            logits = outputs.logits
            preds = torch.argmax(logits, dim=1).cpu().numpy()
            val_preds.extend(preds)
            val_labels_list.extend(labels.cpu().numpy())

            #I printed predictions and actual labels.
            print(f"Predictions: {preds}")
            print(f"Actual Labels: {labels.cpu().numpy()}")

    avg_val_loss = val_loss / len(val_loader)
    val_accuracy = accuracy_score(val_labels_list, val_preds)
    val_f1 = f1_score(val_labels_list, val_preds)
    val_precision = precision_score(val_labels_list, val_preds, zero_division=1)
    val_recall = recall_score(val_labels_list, val_preds, zero_division=1)

    print(f'Epoch {epoch+1}/{epochs}, Validation Loss: {avg_val_loss}')
    print(f'Validation Accuracy: {val_accuracy}, F1 Score: {val_f1}, Precision: {val_precision}, Recall: {val_recall

    #Early Stopping Check
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        early_stop_counter = 0
        # Save the best model
        classification_model.save_pretrained('./best_model_save_directory')
        tokenizer.save_pretrained('./best_model_save_directory')
    else:
        early_stop_counter += 1
        if early_stop_counter >= patience:
            print("Early stopping")
            break

#I saved the model and tokenizer.
classification_model.save_pretrained('./final_model_save_directory')
tokenizer.save_pretrained('./final_model_save_directory')
```

Figure 9: GPT Model Validation and Evaluation