Glassdoor Feedback Analysis Project

Jessica Jazmin Castillo Rios

Table of Contents

1 Project Overview 2

2 Dataset and Scraping 2

3 Preprocessing 5

4 Model Training 6

5 Evaluation 7

6 Sentiment Analysis 8

7 MLOps with MLflow 9

8 Execution Guide 10

9 Architecture Diagram 11

1 Project Overview

This project aims to extract, analyze, and classify employee reviews from Glassdoor. The reviews are classified into positive (Pro) and negative (Con) categories using natural language processing and a Logistic Regression model.

2 Dataset and Scraping

Source

Reviews are scraped directly from `glassdoor.com` using **Selenium** and **BeautifulSoup**. Up to 100 reviews per company can be extracted.

```

# Standard and third-party libraries needed for scraping, analysis, and data manipulation
import time
import json
import pandas as pd
from bs4 import BeautifulSoup
from langdetect import detect
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

# Selenium and WebDriver setup
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

# Initialize the VADER sentiment analyzer
analyzer = SentimentIntensityAnalyzer()

# WebDriver initialization
def init_driver():
    options = Options()

    # NOTE: Headless is disabled to allow manual CAPTCHA solving if needed
    # options.add_argument("--headless=new")

    options.add_argument("--disable-gpu")
    options.add_argument("--window-size=1920,1080")

    # Set a custom user-agent to simulate a real browser
    options.add_argument(
        "user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 "
        "KHTML, like Gecko Chrome/123.0.0.0 Safari/537.36"
    )

    # Options to reduce Selenium detection
    options.add_experimental_option("excludeSwitches", ["enable-automation"])
    options.add_experimental_option("useAutomationExtension", False)

    # Initialize Chrome WebDriver with the above options
    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()), options=options)

    # Hide the "webdriver" flag in JavaScript
    driver.execute_script("Object.defineProperty(navigator, 'webdriver', {get: () => undefined})")

    return driver

# Load cookies from a file to avoid manual login
def load_cookies(driver, cookie_file="glassdoor_cookies.json"):
    print("Loading saved cookies...")
    driver.get("https://www.glassdoor.com/")
    time.sleep(3)

    with open(cookie_file, "r") as f:
        cookies = json.load(f)
        for cookie in cookies:
            if "sameSite" in cookie:
```

```
                cookie["sameSite"] = 'Strict'  # Selenium compatibility adjustment
            driver.add_cookie(cookie)

    # Refresh the page with applied cookies
    driver.refresh()
    time.sleep(4)

# Classify sentiment using VADER
def classify_sentiment(text):
    score = analyzer.polarity_scores(text)["compound"]
    if score > 0.02:
        return "Pro"
    elif score < -0.02:
        return "Con"
    else:
        return "Neutral"

# Scrape company reviews from Glassdoor
def scrape_company_reviews(company_slug, max_reviews=100):
    print(f"Scraping reviews for {company_slug}...")
    driver = init_driver()
    load_cookies(driver)

    reviews = []
    page = 1

    while len(reviews) < max_reviews:
        # Build the URL for the current page
        url = f"https://www.glassdoor.com/Reviews/{company_slug.replace('.htm', f'_P{page}.htm')}"
        driver.get(url)
        time.sleep(5)

        soup = BeautifulSoup(driver.page_source, "html.parser")

        # Find PROS and CONS review sections
        pros = soup.find_all("span", attrs={"data-test": "review-text-PROS"})
        cons = soup.find_all("span", attrs={"data-test": "review-text-CONS"})

        if not pros and not cons:
            print(f"Page {page} does not contain visible reviews. Stopping scraping.")
            break

        # Pair PROS and CONS by index
        total = min(len(pros), len(cons))
        for i in range(total):
            if len(reviews) >= max_reviews:
                break
            try:
                pros_text = pros[i].get_text(strip=True)
                lang_pros = detect(pros_text)
                reviews.append({
                    "company": company_slug.split("-Reviews")[0],
                    "review": pros_text,
                    "language": lang_pros,
                    "type": "Pro"
                })
            except:
                pass

            if len(reviews) >= max_reviews:
                break
            try:
                cons_text = cons[i].get_text(strip=True)
                lang_cons = detect(cons_text)
                reviews.append({
                    "company": company_slug.split("-Reviews")[0],
                    "review": cons_text,
                    "language": lang_cons,
                    "type": "Con"
                })
            except:
```

```
                pass

        print(f"Page {page}: {total * 2} reviews processed (Total: {len(reviews)})")
        page += 1

    driver.quit()
    print(f"Total reviews extracted for {company_slug}: {len(reviews)}")
    return reviews

# Scrape multiple companies and save results to CSV
def scrape_multiple(company_slugs, output_csv="data/glassdoor_reviews.csv"):
    all_reviews = []
    for slug in company_slugs:
        all_reviews.extend(scrape_company_reviews(slug))

    df = pd.DataFrame(all_reviews)
    df.to_csv(output_csv, index=False)
    print(f"Reviews saved to {output_csv}")
```

3 Preprocessing

Preprocessing includes:

- Lowercasing text, removing punctuation and digits.
- Language detection using `langdetect`.
- Removing stopwords based on the detected language (English or Spanish).

```

import pandas as pd
import re
import string
from langdetect import detect
import nltk
from nltk.corpus import stopwords

# Check if stopwords are already downloaded; if not, download them
try:
    nltk.data.find("corpora/stopwords")
except LookupError:
    nltk.download("stopwords")

# Load stopword lists for English and Spanish
stopwords_en = set(stopwords.words("english"))
stopwords_es = set(stopwords.words("spanish"))

# Function to clean the text:
# converts to lowercase, removes punctuation and digits
def clean_text(text):
    text = text.lower()  # Lowercase
    text = re.sub(f"[{re.escape(string.punctuation)}]", "", text)  # Remove punctuation
    text = re.sub(r"\d+", "", text)  # Remove digits
    return text.strip()  # Trim whitespace

# Function to detect the language of a text
def detect_language(text):
    try:
        return detect(text)
    except:
        return "unknown"  # If detection fails, return "unknown"

# Function to remove stopwords based on the detected language
def remove_stopwords(text, lang):
    tokens = text.split()  # Split the text into simple tokens
    stops = stopwords_en if lang == "en" else stopwords_es  # Choose the appropriate stopword set
    filtered = [word for word in tokens if word not in stops]  # Remove stopwords
    return " ".join(filtered)  # Join the filtered words back into a string

# Function to apply all preprocessing steps to a DataFrame
def preprocess_dataframe(df):
    print("[INFO] Preprocessing reviews...")

    # Basic text cleaning
    df["cleaned"] = df["review"].apply(clean_text)

    # Language detection for each review
    df["language"] = df["review"].apply(detect_language)

    # Remove stopwords based on detected language
    df["processed"] = df.apply(lambda row: remove_stopwords(row["cleaned"], row["language"]), axis
    =1)

    return df
```

4 Model Training

The model used is a **Logistic Regression** pipeline with TF-IDF vectorization.

```

# Import necessary libraries for text vectorization, model training, and evaluation
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Function to train a text classification model using Logistic Regression
# df: DataFrame with preprocessed data
# text_column: column containing the processed text
# label_column: column containing the labels ("Pro", "Con")
def train_classifier(df, text_column="processed", label_column="type"):
    print("Training classification model...")

    # Filter the DataFrame to keep only "Pro" and "Con" classes
    df = df[df[label_column].isin(["Pro", "Con"])]

    # Split the data into training and testing sets (80%-20%)
    # Ensures at least 2 examples in the test set
    X_train, X_test, y_train, y_test = train_test_split(
        df[text_column],
        df[label_column],
        test_size=max(2, int(len(df) * 0.2)),
        random_state=42
    )

    # Show class distribution in the test set
    print("Class distribution in test set:", y_test.value_counts())

    # Define a pipeline: TF-IDF vectorization followed by Logistic Regression classifier
    pipeline = Pipeline([
        ("tfidf", TfidfVectorizer()),
        ("clf", LogisticRegression(max_iter=500, class_weight="balanced"))  # max_iter=500 to
    ensure convergence
    ])

    # Train the pipeline with the training data
    pipeline.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = pipeline.predict(X_test)

    # Generate a classification report with per-class metrics
    report = classification_report(y_test, y_pred, output_dict=True)

    print("Classification completed.")

    # Return the trained model, the report, and the evaluation data
    return pipeline, report, X_test, y_test, y_pred
```

5 Evaluation

Evaluation metrics include:

- Precision
- Recall
- F1-Score
- Confusion Matrix

```

# Import necessary libraries for plotting and evaluating classification models
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

# Function to evaluate a classification model
# Takes the true values (y_test), predicted values (y_pred),
# and the class labels to be shown in the confusion matrix
def evaluate_model(y_test, y_pred, labels=["Pro", "Con"]):
    # Print a report with metrics like precision, recall, and F1-score
    print("Model results:")
    print(classification_report(y_test, y_pred))

    # Compute the confusion matrix with the specified labels
    cm = confusion_matrix(y_test, y_pred, labels=labels)

    # Create a display object for visualizing the confusion matrix
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)

    # Plot the confusion matrix using a blue color map
    disp.plot(cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")

    # Show the plot
    plt.show()
```

6 Sentiment Analysis

VADER sentiment analysis is applied to reviews.

```

from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

# Initialize VADER sentiment analyzer
analyzer = SentimentIntensityAnalyzer()

def analyze_sentiment(text, lang):
    # Applies sentiment analysis using VADER, regardless of language.
    # Returns the sentiment label ('POS', 'NEG', 'NEU') and the compound score.
    score = analyzer.polarity_scores(text)
    label = "POS" if score["compound"] > 0.05 else "NEG" if score["compound"] < -0.05 else "NEU"
    return label, score["compound"]
```

7 MLOps with MLflow

MLflow is used to log:

- Model type and configuration
- Metrics per label (Pro/Con)
- Classification report as a JSON artifact

```

# Import necessary libraries for working with MLflow, file handling, and JSON serialization
import mlflow
import mlflow.sklearn
import json
import os

# Function to log a trained model and its evaluation metrics in MLflow
# model: trained sklearn model
# classification_report_dict: dictionary containing evaluation metrics (e.g., output of classification_report(output_dict=True))
# run_name: name of the experiment run in MLflow
def log_with_mlflow(model, classification_report_dict, run_name="glassdoor_model_run"):
    print("Registering experiment in MLflow...")

    # Define or create an MLflow experiment to store the run data
    mlflow.set_experiment("Glassdoor_Reviews_Analysis")

    # Start a new MLflow run with the specified name
    with mlflow.start_run(run_name=run_name):
        # Log the trained model in MLflow
        mlflow.sklearn.log_model(model, "model")

        # Log model type as a parameter
        mlflow.log_param("model_type", "LogisticRegression")

        # Loop through evaluation metrics for each class ("Pro", "Con") and log the values
        for label in ["Pro", "Con"]:
            for metric in ["precision", "recall", "f1-score"]:
                value = classification_report_dict.get(label, {}).get(metric)
                if value is not None:
                    mlflow.log_metric(f"{label}_{metric}", value)

        # Create a folder to store artifacts if it doesn't exist
        os.makedirs("mlruns_artifacts", exist_ok=True)

        # Save the classification report as a JSON file
        report_path = "mlruns_artifacts/classification_report.json"
        with open(report_path, "w") as f:
            json.dump(classification_report_dict, f, indent=4)

        # Log the JSON report as an artifact in MLflow
        mlflow.log_artifact(report_path)

        print("MLflow tracking completed.")
```

MLflow UI

To view the results, start the MLflow UI:

```
mlflow ui
```

Then, open your browser and navigate to `http://localhost:5000`.

8 Execution Guide

Follow these steps to execute the project:

1. **Set Up the Environment**:
   - Install the required dependencies:

```
pip install -r requirements.txt

```

2. **Run the Script**:
   - Execute the main script:

```
python src/main.py

```

3. **View Results**:
   - Check the console for metrics and visualizations.
   - Use the MLflow UI to explore logged experiments and artifacts.

4. **Upload Results to GitHub**:
   - Use the `upload_results.py` script to push results to a GitHub repository:

```
python upload_results.py

```

9 Architecture Diagram

The following diagram illustrates the architecture of the Glassdoor Feedback Analysis project. It shows the complete pipeline, starting from data extraction via web scraping, followed by preprocessing, model training and evaluation, and finally, experiment tracking using MLflow. This modular design ensures that each component is clearly separated, reusable, and easy to maintain or scale.