Data Engineer - Assessment

Bruno Melgão
brunomelgao96@gmail.com

I - Data Acquisition

- To extract the data from the URLs provided, I created a function that receives three inputs: the URL (where the zip file is available), an extraction path for the files to go to a certain folder on my computer and a save path to save the zip file and then delete it;
- This function uses requests, zipfile, and os modules;

- **Before running this function, make sure that the paths are correct.**

I - Data Acquisition

- It checks that the HTTP response status code is equal to 200, because only then has the download been carried out correctly.
- In any other case, an error message is displayed.

```
def download_and_extract_zip(url, save_path, extract_path):
    response = requests.get(url)
    if response.status_code == 200:
        with open(save_path, 'wb') as file:
            file.write(response.content)
        print(f"Downloaded the ZIP file to {save_path}")

        with zipfile.ZipFile(save_path, 'r') as zip_ref:
            zip_ref.extractall(extract_path)
        print(f"Extracted the contents to {extract_path}")

        os.remove(save_path)
        print(f"Removed the downloaded ZIP file: {save_path}")
    else:
        print("Failed to download the dataset.")
```

I - Data Acquisition

- For one of the data sources (U.S. Department of Energy: Alternative Fuel Stations), before downloading you have to fill in a series of fields, including your name and e-mail address, and also select the dataset. I tried filling in these values using Python and the Selenium package. However, I was unsuccessful. For this reason, I downloaded the .csv file for this dataset for subsequent phases.
- In the dataset from the U.S. Department of Transportation - National Highway Traffic Safety Administration, the .txt file does not contain the names of the variables. For this reason, code was also developed to correctly name the variables.

I – Data Acquisition

- This second function aims to transform the .csv or .txt file that comes from the data sources into a Pandas data frame. In addition, this function places all missing values as np.Nan for the purpose of accounting for missing values in later stages.

```
def read_dataset(file_path):
    file_extension = file_path.split('.')[-1].lower()
    if file_extension not in ('csv', 'txt'):
        raise ValueError("Unsupported file format. Only CSV and TXT files are supported.")
    if file_extension == 'csv':
        df = pd.read_csv(file_path, sep=',', engine='python')
        df = df.apply(lambda x: x.replace('', np.nan), axis=0)
    else:
        lines = []
        with open(file_path, 'r') as file:
            for line in file:
                fields = line.strip().split('\t')
                if fields[0].isdigit():
                    lines.append(fields)
        df = pd.DataFrame(lines)
        df = df.replace('', np.nan)
    return df
```

II – Data Processing

This stage is extremely important. A function has been created that receives a dataframe and performs the following operations:

1. Removes duplicate rows;

2. Removes variables (columns) that always have the same value for all rows;

3. Removes variables with more than a certain threshold percentage of missing values (this threshold is chosen by the user);

4. Analyses outliers, taking into account the Interquantile range, and removes rows with outliers;

5. For variables with a % of missing values below the threshold, these values are filled in, using the median in the case of continuous numerical variables and the mode (majority vote) for the others;

6. There is also the option of the user wanting to use standardised numerical variables, as this is useful for some machine learning models and also for regression models. This is why this function also offers this option

II – Data Processing

- One of the modifications to speed up computing time would be to use the Polars package or PySpark instead of pandas when creating dataframes, since one of them has a significant number of rows and columns (high volume of data).

```python
def preprocess_dataframe(df, flag_standardize=False, threshold=80):
    # Remove duplicate rows
    df = df.drop_duplicates()

    # Remove variables with the same value for all rows
    df = df.loc[:, df.nunique() != 1]

    # Remove variables with more than 'threshold' percentage of np.nan
    nan_percentage = (df.isna().sum() / len(df)) * 100
    columns_to_remove = nan_percentage[nan_percentage > threshold].index
    df = df.drop(columns=columns_to_remove)

    # Separate numerical and non-numerical columns
    numerical_columns = df.select_dtypes(include=[np.number]).columns

    # Fill in missing values
    for col in df.columns:
        if col in numerical_columns:
            # Fill missing values in numerical columns with median
            df[col].fillna(df[col].median(), inplace=True)
        else:
            # Fill missing values in non-numeric columns with majority vote
            df[col].fillna(df[col].mode().iloc[0], inplace=True)

    # Remove outliers in numerical columns using IQR method
    for col in numerical_columns:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        df = df[(df[col] >= (Q1 - 1.5 * IQR)) & (df[col] <= (Q3 + 1.5 * IQR))]

    # Standardize numerical columns if flag_standardize is True
    if flag_standardize:
        scaler = StandardScaler()
        df[numerical_columns] = scaler.fit_transform(df[numerical_columns])

    return df
```

II – Data Processing – future work

- With regard to improving the function that pre-processes the data, there are a few things that could be improved:

1. Some variables may contain a high percentage of missing values but be extremely important for a given study. With this in mind, a careful analysis of the importance of the variables in the context of the problem needs to be made. Some machine learning models already support missing values, which makes it easier to decide whether to keep them;

2. One-Hot-encoding for categorical variables as this method solves the problem of ordinality and also makes it easier for machine learning models to process the data;

II – Data Processing – future work

3. Outliers often contain very useful information, so the decision to keep them should be analysed carefully. There are traditional regression models that deal rather badly with these values, but there are machine learning models that can deal with them. There is also a regression model - quantile regression - which is more flexible than the usual regression models.

4. A function should have been created to check that categorical variables did not have incorrectly filled in values (example: variable Sex, and there being values other than M or F).

5. Other options for transforming numerical variables, such as the logarithm transformation, could have been implemented. In particular, this transformation can help to achieve variables with more symmetrical distributions, which can be important in a statistical study of variables.

6. Some data quality validation tests, such as if we have a variable that contains a date and we want to see if there are any dates that don't make sense (too old). Or if, for example, a variable contains negative values and should only have positive ones.

III – Data Transformation

- A function has been created that receives as inputs a dataframe and a string that indicates the desired final format.
- In this case, only .csv and .JSON files were considered. The former is widely used, easy to read and can be easily imported into Python, R and other programming languages. It is also suitable for structured data (with rows and columns). The latter is used for data interchange between web services, APIs, and applications due to its readability and flexibility. JSON can represent complex hierarchical data structures, making it suitable for nested or unstructured data.
- Other format types such as Parquet (very useful in big data) could have been considered.

III – Data Transformation

```python
def convert_dataframe(dataframe, output_path, to_format):
    try:
        if to_format == 'csv':
            dataframe.to_csv(output_path, index=False)
            print(f"Data converted to CSV and saved to {output_path}")
        elif to_format == 'json':
            dataframe.to_json(output_path, orient='records')
            print(f"Data converted to JSON and saved to {output_path}")
        else:
            print("Invalid target format. Use 'csv' or 'json'.")
```

IV – Data Loading

- In this step, I assumed that the connection to Azure SQL would be made and that the uploaded file would be in .csv or .json format.
- I created a routine using the pyodbc module and left the parameters (server, username, password, etc) unchanged since I couldn't test it. In a normal situation, this access data should be stored in an .env file as it can contain very sensitive information such as passwords.

IV – Data Loading

- This routine is designed for dataframes with many variables (columns), so that we don't have to manually type in the name of each one.
- We could use polars instead of pandas to speed up the process.

```
# connection parameters
server = 'serve_name'
database = 'database_name'
username = 'username'
password = 'password'
driver = '{ODBC Driver 17 for SQL Server}'  # I assume that i want to use this driver

# Define the connection string
connection_string = f'DRIVER={driver};SERVER={server};DATABASE={database};UID={username};PWD={password}'

# Create a database connection within a 'with' statement
with pyodbc.connect(connection_string) as conn:
    cursor = conn.cursor()

    # Load data from a JSON file or CSV file
    file_path = 'your_data.json'
    file_extension = file_path.split('.')[-1]

    # Load JSON or CSV data into a DataFrame
    if file_extension.lower() == 'json':
        with open(file_path, 'r') as json_file:
            data = json.load(json_file)
            df = pd.DataFrame(data)
    elif file_extension.lower() == 'csv':
        df = pd.read_csv(file_path)
    else:
        raise ValueError("Unsupported file format. Only JSON and CSV are supported.")

    column_names = df.columns.tolist()

    # creating a table
    create_table_sql = f"""
    CREATE TABLE IF NOT EXISTS YourTableName (
        {', '.join([f'{col} VARCHAR(255)' for col in column_names])}
    )
    """
    cursor.execute(create_table_sql)

    # loading data into the table
    load_data_sql = f"""
    INSERT INTO YourTableName ({', '.join(column_names)})
    VALUES ({', '.join(['?'] * len(column_names))})
    """
    # Loop through your DataFrame rows and fill in the SQL table for each row
    for index, row in df.iterrows():
        cursor.execute(load_data_sql, *row)
        conn.commit()

    print("Data loaded WITH SUCESS!.")
```

V – Automation Sugesstion

- The choice of schedule depends on data criticality, available resources, and specific business needs. Some pipelines may require multiple daily updates, while others can be processed less frequently.
- For this reason, the suggestion would be to run this pipeline on a daily basis and at a set time. To do this, I would use Python to contain the scripts and I would use Apache AirFlow to schedule and manage the tasks. With regard to the scripts, it is very useful to use Git to be able to check the differences and modifications made to the code over time. It's also important that the scripts have mechanisms for dealing with errors so that they can be monitored.

V – Automation Suggestion

- In addition, my choice of a daily frequency is also due to the fact that higher frequencies (more time between two consecutive runs) are not very useful in detecting problems and errors that may arise from running the pipeline, which can be very useful in correcting them. So, this daily frequency manages to cover that and is not very resource-intensive.