We used a Jupyter Notebook running on an 11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz CPU on a personal computer to carry out the data collection and preprocessing. This choice was made because the execution time was reasonable for these two steps in the methodology adopted for this thesis.

1 Prediction of weather impacted flight delay

1.1 Importing libraries

[ ]:
```
pip install selenium
```

[ ]:
```
import pandas as pd
import numpy as np

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

from datetime import datetime, timedelta
import concurrent.futures
import matplotlib.pyplot as plt
import seaborn as sns
import torch
import os
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from sklearn.feature_selection import SelectKBest, mutual_info_classif
```

As the data collection section takes time, you can skip it until the reproducibility section, as the data collected is already present in the technical work repository in csv format.

1.2 Data collection

1.2.1 Collect flight data from BTS

To collect BTS flight data, go to the following web page:

https://transtats.bts.gov/DL_SelectFields.aspx?gnoyr_VQ=FGJ&QO_fu146_anzr=b0-gvzr

Download the data by ticking the boxes below:

FlightDate; Tail_Number; Flight_Number_Reporting_Airline; Origin; Dest; CRSDepTime; DepTime; DepDelay; DepDel15; TaxiOut; WheelsOff; WheelsOn; TaxiIn; CRSArrTime; ArrTime; ArrDelay; ArrDel15; Cancelled; CancellationCode; Diverted; CRSElapsedTime; ActualElapsedTime; AirTime; Distance; CarrierDelay; WeatherDelay; NASDelay; SecurityDelay; LateAircraftDelay

[ ]:
```python
def merge_jfk_flight_data(year1, year2, directory, filename):
    """
    Merge flight data for JFK airport from several CSV files into a␣
 ↪DataFrame and save it as a CSV file.

    Args:
        year1 (int): The start year of flight data collection.
        year2 (int): The end year of flight data collection.
        directory (str): Path to directory containing all flight data CSV␣
 ↪files.
        filename (str): Filename where the merged flight data will be␣
 ↪saved.

    Returns:
        None
    """

    # List for storing DataFrames
    dataframes_list = []

    # List of months
    months =␣
 ↪["JAN","FEB","MAR","APR","MAY","JUN","JUL","AUG","SEP","OCT","NOV","DEC"]

    # Loop for each year and month
    for year in range(year1, year2+1):

        # Flight data from years during COVID-19 are not merged
        if year in (2019, 2020):
            continue

        for month in months:
            # Name of CSV file to be uploaded
            Flight_Data_filename = os.path.join(directory,␣
 ↪f"T_ONTIME_REPORTING_{month}{year}.csv")

            if os.path.exists(Flight_Data_filename):
                # Upload the CSV file
```

```
                df_flight = pd.read_csv(Flight_Data_filename)

                # Filter for JFK airport
                df_jfk = df_flight[df_flight['ORIGIN'] == 'JFK']

                # Add to main DataFrame
                dataframes_list.append(df_jfk)

            else:
                print(f"File not found : {Flight_Data_filename}")

    if dataframes_list:

        # Combine all DataFrames
        df_combined = pd.concat(dataframes_list, ignore_index=True)

        # Save the combined DataFrame as a CSV file
        df_combined.to_csv(filename, index=False)

        print("The combined data was successfully saved.")
    else:
        print("No files were found and processed.")
```

[ ]:
```
merge_jfk_flight_data(2010,2023,"Chandrakumar_s419255_Data_MScCSTE","JFK_Flight_Data_20
 ↪csv")
```

1.2.2 Collect weather data from Weather Underground

Functions used for converting units of measurement for variables

[ ]:
```
def fahrenheit_to_celsius(f):
    """
    Converts the temperature from Fahrenheit to Celsius.

    Args:
        f (int): Temperature in Fahrenheit.

    Returns:
        int: Temperature converted to Celsius.
    """
    if f == 0:
        return 0
    return round((f - 32) * 5.0 / 9.0)
```

```python
def inches_to_mm(inches):
    """
    Converts inches to millimetres.

    Args:
        inches (float): Measurement in inches.

    Returns:
        int: Measurement in millimetres and rounded.
    """
    return round(inches * 25.4)

def inches_mercury_to_hpa(inches):
    """
    Converts pressure from inches of mercury to hectopascals.

    Args:
        inches (float): Pressure in inches of mercury.

    Returns:
        float: Pressure in hectopascals and rounded.
    """
    return round(inches * 33.8639, 2)

def mph_to_kmh(mph):
    """
    Converts speed from miles per hour to kilometers per hour.

    Args:
        mph (float): Speed in miles per hour.

    Returns:
        int: Speed in kilometers per hour and rounded.
    """
    return round(mph * 1.609344)

def weather_data_into_dataframe(data, date):
    """
    Converts raw weather data into a structured DataFrame.

    Args:
        data (list of lists): Raw weather data where each sublist␣
↪represents a row of data.
        date (str): The date in YYYY-MM-DD format related to the data.
```

```
    Returns:
        pd.DataFrame: A DataFrame with processed weather data.
    """

    # Define DataFrame column headers
    headers = ["DateTime", "Temperature (°C)", "Dew Point (°C)",␣
↪"Humidity (%)", "Wind",
               "Wind Speed (km/h)", "Wind Gust (km/h)", "Pressure (hPa)",␣
↪"Precip. (mm)", "Condition"]

    df = pd.DataFrame(data, columns=headers)

    # Remove data unit symbols from all data
    df['Temperature (°C)'] = df['Temperature (°C)'].str.replace('°F', '').
↪astype(int)
    df['Dew Point (°C)'] = df['Dew Point (°C)'].str.replace('°F', '').
↪astype(int)
    df['Humidity (%)'] = df['Humidity (%)'].str.replace('°%', '').
↪astype(int)
    df['Wind Speed (km/h)'] = df['Wind Speed (km/h)'].str.replace('°mph',␣
↪'').astype(float)
    df['Wind Gust (km/h)'] = df['Wind Gust (km/h)'].str.replace('°mph',␣
↪'').astype(float)
    df['Pressure (hPa)'] = df['Pressure (hPa)'].str.replace('°in', '').
↪astype(float)
    df['Precip. (mm)'] = df['Precip. (mm)'].str.replace('°in', '').
↪astype(float)

    # Apply conversion functions to data columns
    df["Temperature (°C)"] = df["Temperature (°C)"].
↪apply(fahrenheit_to_celsius)
    df["Dew Point (°C)"] = df["Dew Point (°C)"].
↪apply(fahrenheit_to_celsius)
    df["Wind Speed (km/h)"] = df["Wind Speed (km/h)"].apply(mph_to_kmh)
    df["Wind Gust (km/h)"] = df["Wind Gust (km/h)"].apply(mph_to_kmh)
    df["Pressure (hPa)"] = df["Pressure (hPa)"].
↪apply(inches_mercury_to_hpa)
    df["Precip. (mm)"] = df["Precip. (mm)"].apply(inches_to_mm)

    # For weather data collected for one day, it is possible to have the␣
↪data for the day before or after.
```

```
    observation_date = datetime.strptime(date, "%Y-%m-%d")

    # Create a list to store adjusted datetimes
    dates = []
    current_date = observation_date
    day_transition = 0

    # Iterate on the data to adjust the datetimes
    for i, time in enumerate(df["DateTime"]):

        if i == 0 and "PM" in time:
            # If the first entry contains PM, adjust to the previous date
            current_date -= timedelta(days=1)

        elif day_transition == 0 and "AM" in time:
            # If the time value contains AM for the first entry or for␣
↪just
            # after the first entry, reset the date to the current date.
            day_transition = 1
            current_date = observation_date

        elif day_transition == 1 and "AM" in time and "PM" in␣
↪df["DateTime"][i - 1]:
            # If we change from PM to AM, adjust the date by one day
            current_date += timedelta(days=1)

        # Add corrected datetimes to the list
        dates.append(current_date.strftime("%Y-%m-%d") + " " + time)

    # Update the DateTime column with the new values
    df["DateTime"] = pd.to_datetime(dates)

    return df
```

Collection of weather data for a specific date

```
[ ]: def collect_weather_data_for_date(date):
    """
    Collects weather data for a specific date from the Weather␣
↪Underground site.

    Args:
        date (str): Date in YYYY-MM-DD format.

```

```
    Returns:
        pd.DataFrame: A DataFrame containing the weather data for the␣
↪specific date.
    """

    # URL to collect weather data for the specified date
    url = 'https://www.wunderground.com/history/daily/us/ny/new-york-city/
↪KJFK/date/' + date
    #print(url)

    # Configuration of selenium package objects to access the page via␣
↪the URL
    options = webdriver.FirefoxOptions()
    options.add_argument('--headless')
    driver = webdriver.Firefox(options=options)

    driver.get(url)

    # Wait for the weather data table to appear
    wait = WebDriverWait(driver, 60)
    table = wait.until(EC.presence_of_element_located((By.XPATH, "//
↪table[contains(@class, 'mat-table cdk-table mat-sort␣
↪ng-star-inserted')]")))

    rows = []
    table_rows = table.find_elements(By.XPATH, ".//tbody/
↪tr[contains(@class,'mat-row cdk-row ng-star-inserted')]")

    # Extraction of weather data from each row of the table
    for table_row in table_rows:
        data = []
        table_columns = table_row.find_elements(By.XPATH, ".//td")
        for table_column in table_columns:
            value = table_column.get_attribute('textContent')
            data.append(value)
        rows.append(data)

    driver.quit()

    # Converting weather data into DataFrame
    df = weather_data_into_dataframe(rows, date)

    return df
```

Collection of weather data for a specific month and year

```
[ ]: def collect_weather_data(month, year):
        """
        Collects weather data for a specific month and year.

        Args:
            month (int): Month for which weather data is to be collected.
            year (int): Year for which weather data is to be collected.

        Returns:
            pd.DataFrame:  A DataFrame containing the weather data for the specific month and year.
        """
        # List of months
        months = ["JAN", "FEB", "MAR", "APR", "MAY", "JUN", "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"]

        # Define the start and end dates for each month
        start_date = datetime(year, month, 1)

        if month == 12:
            end_date = datetime(year + 1, 1, 1) - timedelta(days=1)
        else:
            end_date = datetime(year, month + 1, 1) - timedelta(days=1)

        # Store all dates for the month
        dates = [(start_date + timedelta(days=i)).strftime('%Y-%m-%d') for i in range((end_date - start_date).days + 1)]

        weather_data = []

        # Collect weather data for each date using multithreading to speed up the process
        with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
            futures = [executor.submit(collect_weather_data_for_date, date) for date in dates]
            for future in concurrent.futures.as_completed(futures):
                df_weather = future.result()
                if not df_weather.empty:
                    weather_data.append(df_weather)

        # Combine the weather data for each day of the month in a DataFrame
        df_weather_month = pd.concat(weather_data, ignore_index=True)
```

```
    df_weather_month = df_weather_month.sort_values(by="DateTime")

    # Save weather data in a CSV file
    output_filename = f'Chandrakumar_s419255_Data_MScCSTE/
 ↪WEATHER_DATA_KJFK_{months[month-1]}{year}.csv'
    df_weather_month.to_csv(output_filename, index=False)

    print(f'Data for {year}-{month:02} saved in {output_filename}')

    return df_weather_month
```

[ ]:
```
# Execution of weather data collection
years = range(2009, 2010) # Year values can be changed
dataframe = []

for year in years:
    # Weather data from years during COVID-19 are not collected
    if year in (2019, 2020):
        continue

    for i in range(12, 13): # Month values can be changed
        dataframe.append(collect_weather_data(i, year))

# It is better to collect weather data 6 months at a time to prevent the␣
 ↪code from crashing.
```

[ ]:
```
def merge_jfk_weather_data(year1, year2, directory, filename):
    """
    Merge weather data for JFK airport from several CSV files into a␣
 ↪DataFrame and save it as a CSV file.

    Args:
        year1 (int): The start year of weather data collection.
        year2 (int): The end year of weather data collection.
        directory (str): Path to directory containing all weather data␣
 ↪CSV files.
        filename (str): Filename where the merged weather data will be␣
 ↪saved.

    Returns:
        None
    """

    # List for storing DataFrames
```

```
    dataframes_list = []

    # List of months
    months =␣
 ↪["JAN","FEB","MAR","APR","MAY","JUN","JUL","AUG","SEP","OCT","NOV","DEC"]

    # Loop for each year and month
    for year in range(year1, year2+1):

        # Weather data from years during COVID-19 are not merged
        if year in (2019, 2020):
            continue

        for month in months:
            # Name of CSV file to be uploaded
            Weather_Data_filename = os.path.join(directory,␣
 ↪f"WEATHER_DATA_KJFK_{month}{year}.csv")

            if os.path.exists(Weather_Data_filename):
                # Upload the CSV file
                df_weather = pd.read_csv(Weather_Data_filename)

                # Add to main DataFrame
                dataframes_list.append(df_weather)

            else:
                print(f"File not found : {Weather_Data_filename}")

    if dataframes_list:

        # Combine all DataFrames
        df_combined = pd.concat(dataframes_list, ignore_index=True)

        # Save the combined DataFrame as a CSV file
        df_combined.to_csv(filename, index=False)

        print("The combined data was successfully saved.")
    else:
        print("No files were found and processed.")
```

```
[ ]: merge_jfk_weather_data(2009,2023,"Chandrakumar_s419255_Data_MScCSTE","JFK_Weather_Data_
 ↪csv")
```

1.3 Merging flight and weather data

[ ]:
```
def convert_to_time_format(val):
    """
    Converts a Time value into HH:MM format.

    Args:
        val (float): Time value in military format or NaN.

    Returns:
        str: Time in HH:MM format or None if the value is NaN.
    """
    # Check if the value is NaN
    if pd.isnull(val):
        return None

    hours = int(val) // 100
    minutes = int(val) % 100

    # If the hours are equal to 24, set them to 0
    if hours == 24:
        hours = 0

    # Format Time in HH:MM using two digits for hours and minutes
    return f"{hours:02d}:{minutes:02d}"

def convert_to_datetime(date, time):
    """
    Combines a date and a time.

    Args:
        date (str): Date value into YYYY-MM-DD format.
        time (str): Time value into HH:MM format or None.

    Returns:
        pd.Timestamp: Datetime object corresponding to the time and date␣
 ↪combination.
    """
    # Check if the value is NaN
    if pd.isnull(time):
        return None

    # Combine date and time
    return pd.to_datetime(f"{date} {time}")
```

```python
def merge_flight_weather_data(filename1, filename2, timing):
    """
    Merges flight data with weather data into a DataFrame and save it as␣
 ↪a CSV file.

    Args:
        filename1 (str): Path to CSV file containing flight data.
        filename2 (str): Path to CSV file containing weather data.
        timing (int): Timing of weather data collected before the␣
 ↪scheduled flight departure.

    Returns:
        pd.DataFrame: a DataFrame merged with flight data and associated␣
 ↪weather data.
    """

    # Check if the files exist
    if os.path.exists(filename1) and os.path.exists(filename2):

        df_flight = pd.read_csv(filename1)
        df_flight['FL_DATE'] = pd.to_datetime(df_flight['FL_DATE']).dt.
 ↪date

        # Convert time to HH:MM format
        df_flight['CRS_DEP_TIME_1'] = df_flight.apply(lambda row:␣
 ↪convert_to_time_format(row['CRS_DEP_TIME']), axis=1)

        df_weather = pd.read_csv(filename2)

        # Create or convert a datetime column for flight and weather data
        df_flight["FlightDateTime"] = df_flight.apply(lambda row:␣
 ↪convert_to_datetime(row['FL_DATE'], row['CRS_DEP_TIME_1']), axis=1)
        df_weather["DateTime"] = pd.to_datetime(df_weather["DateTime"])

        df_flight[f"FlightDateTime_minus_{timing}h"] =␣
 ↪df_flight["FlightDateTime"] - pd.Timedelta(hours=timing)

        # Sort DataFrames by datetime
        df_flight = df_flight.
 ↪sort_values(f"FlightDateTime_minus_{timing}h")
        df_weather = df_weather.sort_values('DateTime')
```

```
        # Merge flight and weather data according to the given timing
        if timing == 0:
            # No tolerance required
            flight_weather_data = pd.merge_asof(df_flight, df_weather,␣
↪left_on=f"FlightDateTime_minus_{timing}h",
                                                right_on='DateTime',␣
↪direction='backward')
        else:
            # With tolerance
            flight_weather_data = pd.merge_asof(df_flight, df_weather,␣
↪left_on=f"FlightDateTime_minus_{timing}h",
                                                right_on='DateTime',␣
↪direction='backward',tolerance=pd.Timedelta(hours=timing))

        # Deletes columns used just for merging
        flight_weather_data.drop(columns=["FlightDateTime",␣
↪f"FlightDateTime_minus_{timing}h", "DateTime","CRS_DEP_TIME_1"],␣
↪inplace=True)

        output_file = f"JFK_Flight_Weather_Data_2010_2023_{timing}h.csv"
        flight_weather_data.to_csv(output_file, index=False)
```

[ ]:
```
filename1 = "JFK_Flight_Data_2010_2023.csv"
filename2 = "JFK_Weather_Data_2010_2023.csv"
timings = [0,2,4,8,16,24,48]

for timing in timings:
    merge_flight_weather_data(filename1, filename2, timing)
```

1.4 Reproducibility

[ ]:
```
# Fixed random seed for reproducibility of results
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
torch.cuda.manual_seed(RANDOM_SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
print(RANDOM_SEED)
```

The cells in the Jupiter Notebook file must be run in chronological order to ensure reproducible results.

1.5 Data Preprocessing

The timing of the weather data can be changed (0h, 2h, 4h, 8h, 16h, 24h or 48h before flight departure)

[ ]:
```
timing = 0
# timing = 2
# timing = 4
# timing = 8
# timing = 16
# timing = 24
# timing = 48
```

[ ]:
```
df = pd.read_csv(f"JFK_Flight_Weather_Data_2010_2023_{timing}h.csv")
df
```

[ ]:
```
# Replace military times 2400.0 with 0.0 to indicate midnight
for column in␣
↪["DEP_TIME","WHEELS_OFF","WHEELS_ON","CRS_ARR_TIME","ARR_TIME"]:
    df.loc[df[column] == 2400.0, column] = 0.0
```

1.5.1 Handling Missing Values

[ ]:
```
# Display a summary of the DataFrame
df.info()
```

[ ]:
```
# Determine the number of missing values for each column in the DataFrame
df.isnull().sum()
```

Removing unnecessary data

[ ]:
```
# Keep only flights that are not diverted or cancelled
df = df[(df['DIVERTED'] == 0) & (df['CANCELLED'] == 0)].
↪reset_index(drop=True)

# Keep only rows where the Condition column contains no missing values
df = df[df["Condition"].notna()].reset_index(drop=True)

# Remove unnecessary columns from the DataFrame
df = df.drop(['DIVERTED','CANCELLED','CANCELLATION_CODE'], axis=1)
```

Replace missing values with specific ones

[ ]:
```
# Replace the missing values by 0 for the columns associated with the␣
↪departure/arrival delay,
```

```
# as it can be seen that the missing values are present when the␣
↪scheduled departure/arrival time
# is the same as the actual departure/arrival time.
for column in ['ARR_DELAY','ARR_DEL15','DEP_DELAY','DEP_DEL15']:
    df.loc[df[column].isna(), column] = 0.0

# Replace missing values with 0 for delay columns, as missing values are␣
↪present when flights
# arrive on time.
for column in ['CARRIER_DELAY', 'WEATHER_DELAY', 'NAS_DELAY',␣
↪'SECURITY_DELAY','LATE_AIRCRAFT_DELAY']:
    df.loc[df['ARR_DEL15'] == 0.0, column] = 0.0

# Replace the missing values by CALM for the Wind column, as the missing␣
↪values are present
# when the wind speed or wind gust is equal to 0.
df.loc[df["Wind"].isna(), "Wind"] = "CALM"
```

Convert DataFrame object columns

[ ]:
```
# Convert specified DataFrame columns to string format
df['ORIGIN'] = df['ORIGIN'].convert_dtypes()
df['DEST'] = df['DEST'].convert_dtypes()
df['OP_UNIQUE_CARRIER'] = df['OP_UNIQUE_CARRIER'].convert_dtypes()
df['Wind'] = df['Wind'].convert_dtypes()
df['Condition'] = df['Condition'].convert_dtypes()
df['TAIL_NUM'] = df['TAIL_NUM'].convert_dtypes()
```

[ ]:
```
# Convert FL_DATE column in the DataFrame to datetime format
df['FL_DATE'] = pd.to_datetime(df['FL_DATE'])
```

1.5.2 Target variable creation

[ ]:
```
# Sets the STATUS column to 0 to indicate that the flight is on-time
df['STATUS'] = 0

# Sets the STATUS column to 1 for delayed flights with no weather delay.
df.loc[(df['WEATHER_DELAY'] == 0.0) & (df['ARR_DEL15'] == 1.0), 'STATUS']␣
↪= 1

# Sets the STATUS column to 2 for flights delayed due to weather␣
↪conditions
df.loc[(df['WEATHER_DELAY'] > 0.0) & (df['ARR_DEL15'] == 1.0), 'STATUS']␣
↪= 2
```

```
df
```

1.5.3 Exploratory Data Analysis (EDA)

[ ]:
```
df = df.sort_values(by=['FL_DATE','DEST', 'CRS_DEP_TIME']).
↪reset_index(drop=True)
df
```

[ ]:
```
df.describe()
```

Distribution of different flight categories

[ ]:
```
distribution = df["STATUS"].value_counts()

status = {0: 'on-time', 1: 'delayed (non-weather)', 2: 'delayed␣
↪(weather)'}


plt.figure(figsize=(10, 6))
bars = plt.bar(distribution.index, distribution.values, color=['blue',␣
↪'orange', 'red'], width=0.4)

# Axis configuration
plt.xlabel('Flight Category', fontweight='bold', fontsize=14)
plt.ylabel('Number of flights', fontweight='bold', fontsize=14)

# Define grid
plt.grid(axis='y', linestyle='', alpha=0.7)

# Define tick labels for the x-axis and y-axis
plt.xticks(ticks=distribution.index, labels=[status[i] for i in␣
↪distribution.index], rotation=0, fontsize=12)
plt.yticks(fontsize=13)
y_ticks = plt.gca().get_yticks()
plt.gca().set_yticklabels([f'{int(y)}' for y in y_ticks])

# Add values above the bars
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2.0, yval, int(yval),␣
↪va='bottom', ha='center', fontsize=11)

plt.show()
```

Different types of delay

```
[ ]: # Calculation of the number of samples for each type of delay
carrier_delay = (df['CARRIER_DELAY'] > 0).sum()
security_delay = (df['SECURITY_DELAY'] > 0).sum()
weather_delay = (df['WEATHER_DELAY'] > 0).sum()
late_aircraft_delay = (df['LATE_AIRCRAFT_DELAY'] > 0).sum()
NAS_delay = (df['NAS_DELAY'] > 0).sum()

# Creation of the DataFrame for delays
delay = pd.DataFrame({'Delay Type': ['Air carrier', 'Security',␣
↪'Weather', 'Late-arriving aircraft', 'NAS'],
    'Count': [carrier_delay, security_delay, weather_delay,␣
↪late_aircraft_delay, NAS_delay]})

plt.figure(figsize=(10, 6))
bars = sns.barplot(x='Delay Type', y='Count', data=delay)

# Axis configuration
plt.xlabel('Type of delay', fontweight='bold', fontsize=14)
plt.ylabel('Number of flights', fontweight='bold', fontsize=14)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)

# Define grid
plt.grid(axis='y', linestyle='', alpha=0.7)

# Adding values above the bars
for bar in bars.patches:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2.0, yval, int(yval),␣
↪va='bottom', ha='center', fontsize=11)

plt.show()
```

Distribution of arrival flight delays

```
[ ]: plt.figure(figsize=(10, 6))
sns.histplot(df['ARR_DELAY'], bins=50)

# Axis configuration
plt.xlabel('Arrival delay (minutes)', fontweight='bold', fontsize=14)
plt.ylabel('Number of flights', fontweight='bold', fontsize=14)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)
```

```
plt.show()
```

[ ]:
```
plt.figure(figsize=(10, 6))
sns.scatterplot(x='DISTANCE', y='ARR_DELAY', data=df)

# Axis configuration
plt.xlabel('Distance', fontweight='bold', fontsize=14)
plt.ylabel('Arrival delay (minutes)', fontweight='bold', fontsize=14)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)

plt.show()
```

Distribution of weather delays

[ ]:
```
columns = ['Temperature (°C)', 'Dew Point (°C)', 'Humidity (%)','Wind',␣
↪'Wind Speed (km/h)', 'Wind Gust (km/h)', 'Pressure (hPa)','Precip.␣
↪(mm)']
for column in columns:
    plt.figure(figsize=(10, 6))

    sns.scatterplot(x=df[column], y=df['WEATHER_DELAY'])

    # Axis configuration
    plt.xlabel(column, fontweight='bold', fontsize=14)
    plt.ylabel('Weather Delays (minutes)', fontweight='bold', fontsize=14)
    plt.xticks(fontsize=13)
    plt.yticks(fontsize=13)

    plt.show()
```

Influence of the weather on flight delays with weather score

[ ]:
```
def classify_temperature(temp):
    """
    Classifies temperatures with a score according to their influence on␣
↪flight delays.
    The higher the score, the greater the influence of temperature on␣
↪flight delays.

    Args:
        temp (int): Temperature in degrees Celsius.

    Returns:
```

```
        int: Temperature score
    """
    if 10 <= temp <= 20:
        return 0
    elif 0 <= temp < 10 or 20 < temp <= 30:
        return 1
    else:
        return 2

def classify_dew_point(dp):
    """
    Classifies dew point with a score according to their influence on␣
 ↪flight delays.
    The higher the score, the greater the influence of dew point on␣
 ↪flight delays.

    Args:
        dp (int): Dew point in degrees Celsius.

    Returns:
        int: Dew point score
    """
    if 10 <= dp <= 20:
        return 0
    elif 0 <= dp < 10 or 20 < dp <= 25:
        return 1
    else:
        return 2

def classify_humidity(hum):
    """
    Classifies humidity with a score according to their influence on␣
 ↪flight delays.
    The higher the score, the greater the influence of humidity on flight␣
 ↪delays.

    Args:
        hum (int): Humidity in percent.

    Returns:
        int: Humidity score
    """
    if 40 <= hum <= 60:
        return 0
```

```python
    elif 30 <= hum < 40 or 60 < hum <= 70:
        return 1
    else:
        return 2

def classify_wind_speed(ws):
    """
    Classifies wind speed with a score according to their influence on flight delays.
    The higher the score, the greater the influence of wind speed on flight delays.

    Args:
        ws (int): Wind speed in km/h.

    Returns:
        int: Wind speed score
    """
    if ws < 10:
        return 0
    elif 10 <= ws <= 20:
        return 1
    else:
        return 2

def classify_wind_gust(wg):
    """
    Classifies wind gust with a score according to their influence on flight delays.
    The higher the score, the greater the influence of wind gust on flight delays.

    Args:
        wg (int): Wind gust in km/h.

    Returns:
        int: Wind gust score
    """
    if 20 < wg < 30:
        return 0
    elif 10 <= wg < 20 or 30 < wg <= 40:
        return 1
    else:
        return 2
```

```python
def classify_pressure(press):
    """
    Classifies pressure with a score according to their influence on␣
    ↪flight delays.
    The higher the score, the greater the influence of pressure on flight␣
    ↪delays.

    Args:
        press (float): Pressure in hpa.

    Returns:
        int: Pressure score
    """
    if 1012 <= press <= 1015:
        return 0
    elif 1000 <= press < 1012 or 1015 < press <= 1025:
        return 1
    else:
        return 2

def classify_precip(precip):
    """
    Classifies precipation with a score according to their influence on␣
    ↪flight delays.
    The higher the score, the greater the influence of precipation on␣
    ↪flight delays.

    Args:
        precip (int): Precipation in mm.

    Returns:
        int: Precipation score
    """
    if precip == 0:
        return 0
    elif 0 < precip <= 10:
        return 1
    else:
        return 2

def classify_condition(cond):
    """
```

```
    Classifies weather condition with a score according to their␣
↪influence on flight delays.
    The higher the score, the greater the influence of weather condition␣
↪on flight delays.

    Args:
        cond (string): Weather condition.

    Returns:
        int: Weather condition score
    """
    if cond in ['Fair', 'Fair / Windy', 'Partly Cloudy', 'Partly Cloudy /␣
↪Windy']:
        return 0

    elif cond in ['Mostly Cloudy', 'Mostly Cloudy / Windy']:
        return 1

    elif cond in ['Cloudy', 'Cloudy / Windy']:
        return 2

    elif cond in ['Light Rain', 'Light Rain / Windy', 'Light Drizzle',␣
↪'Light Drizzle / Windy',
                  'Drizzle', 'Haze', 'Mist', 'Smoke', 'Patches of Fog']:
        return 3

    elif cond in ['Rain', 'Rain / Windy']:
        return 4

    elif cond in ['Light Snow', 'Light Snow / Windy', 'Light Freezing␣
↪Rain', 'Light Freezing Drizzle',
                  'Light Sleet', 'Rain and Sleet','Drizzle and Fog',␣
↪'Fog', 'Fog / Windy', 'Patches of Fog / Windy',
                  'Shallow Fog', 'Snow and Sleet', 'Snow and Sleet /␣
↪Windy', 'Haze / Windy', 'Mist / Windy',
                  'Smoke / Windy']:
        return 5

    elif cond in ['Snow', 'Snow / Windy', 'Light Snow and Sleet', 'Light␣
↪Snow and Sleet / Windy',
                  'Wintry Mix', 'Wintry Mix / Windy', 'Rain and Snow',␣
↪'Rain and Snow / Windy',
```

```
                'Unknown Precipitation', 'Sleet', 'Sleet / Windy',␣
↪'Blowing Snow / Windy','Blowing Snow']:
        return 6

    elif cond in ['Heavy Rain', 'Heavy Rain / Windy', 'Squalls / Windy',␣
↪'Heavy Drizzle', 'Rain / Freezing Rain',
                'Snow / Freezing Rain', 'Light Snow / Freezing␣
↪Rain','Heavy Snow', 'Heavy Snow / Windy',
                'Thunder in the Vicinity', 'Rain / Freezing Rain /␣
↪Windy']:
        return 7

    elif cond in ['Light Rain with Thunder', 'Heavy T-Storm','Heavy␣
↪T-Storm / Windy', 'T-Storm', 'T-Storm / Windy',
                'Thunder', 'Thunder / Windy','Thunder and Hail /␣
↪Windy']:
        return 8
```

Calculating the weather score

[ ]:
```
def calculate_weather_score(row):
    """
    Calculates a score based on various weather conditions.

    Args:
        row (pd.Series): A row in the DataFrame containing the weather␣
↪data for a flight.

    Returns:
        float: Normalised score for weather based on several criteria.
    """
    score = 0

    # Application of coefficients according to the importance of the␣
↪scores
    score += (classify_temperature(row['Temperature (°C)'])/2) * 2
    score += (classify_dew_point(row['Dew Point (°C)'])/2) * 2
    score += (classify_humidity(row['Humidity (%)'])/2) * 4.5
    score += (classify_wind_speed(row['Wind Speed (km/h)'])/2) * 3
    score += (classify_wind_gust(row['Wind Gust (km/h)'])/2) * 4
    score += (classify_pressure(row['Pressure (hPa)'])/2) * 3
    score += (classify_precip(row['Precip. (mm)'])/2) * 4
    score += (classify_condition(row['Condition'])/8) * 5
```

```
    # Global weather score normalisation
    return score/27.5

df['Weather_Score'] = df.apply(calculate_weather_score, axis=1)
```

[ ]:
```
plt.figure(figsize=(10, 6))

bins = [i/10 for i in range(0, 11)]
number, bin, bars = plt.hist([df[df['STATUS'] == 2]['Weather_Score']],␣
 ↪bins=bins)

# Axis configuration
plt.xlabel('Weather Score', fontweight='bold', fontsize=14)
plt.ylabel('Number of Flights Delayed \n Due to Weather',␣
 ↪fontweight='bold', fontsize=14)
plt.xticks([i/10 for i in range(0, 11)], fontsize=13)
plt.yticks(fontsize=13)

# Define grid
plt.grid(True, linestyle='', alpha=0.7)

# Adding values above the bars
for bar in bars.patches:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2.0, yval, int(yval),␣
 ↪va='bottom', ha='center', fontsize=11)

plt.show()
```

[ ]:
```
plt.figure(figsize=(10, 6))

sns.scatterplot(x=df["Weather_Score"], y=df['WEATHER_DELAY'])

# Axis configuration
plt.xlabel("Weather_Score", fontweight='bold', fontsize=14)
plt.ylabel('Weather Delays (minutes)', fontweight='bold', fontsize=14)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)

plt.show()
```

1.5.4 Data Encoding

[ ]:
```python
def encode(df):
    """
    Encodes categorical and string columns using Label Encoding.

    Args:
        df (pd.DataFrame): A DataFrame containing columns to encode.

    Returns:
        pd.DataFrame: A DataFrame with encoded columns.
    """
    # Select columns of type category, object, string or datetime.
    columnsToEncode = list(df.select_dtypes(include=['category',␣
 ↪'object', 'string', 'datetime']))
    for feature in columnsToEncode:
        # Initialise the LabelEncoder
        le = LabelEncoder()
        print(feature)
        # Encode the column with LabelEncoder
        df[feature] = le.fit_transform(df[feature])
    return df
```

[ ]:
```python
df = encode(df)
df
```

1.5.5 Data Normalisation

[ ]:
```python
# List of columns to be normalised
features = ['FL_DATE', 'OP_UNIQUE_CARRIER', 'TAIL_NUM',␣
 ↪'OP_CARRIER_FL_NUM',
       'ORIGIN', 'DEST', 'CRS_DEP_TIME', 'DEP_TIME', 'DEP_DELAY',␣
 ↪'DEP_DEL15',
       'TAXI_OUT', 'WHEELS_OFF', 'WHEELS_ON', 'TAXI_IN', 'CRS_ARR_TIME',
       'ARR_TIME', 'ARR_DELAY', 'ARR_DEL15', 'CRS_ELAPSED_TIME',
       'ACTUAL_ELAPSED_TIME', 'AIR_TIME', 'DISTANCE', 'CARRIER_DELAY',
       'WEATHER_DELAY', 'NAS_DELAY', 'SECURITY_DELAY',␣
 ↪'LATE_AIRCRAFT_DELAY',
       'Temperature (°C)', 'Dew Point (°C)', 'Humidity (%)', 'Wind',
       'Wind Speed (km/h)', 'Wind Gust (km/h)', 'Pressure (hPa)',
       'Precip. (mm)', 'Condition']

# Creation of a MinMaxScaler object
scaler = MinMaxScaler()
```

```
df[features] = scaler.fit_transform(df[features])
df
```

1.5.6 Data Split

[ ]:
```
# List of features
features = ['FL_DATE', 'OP_UNIQUE_CARRIER', 'TAIL_NUM',␣
 ↪'OP_CARRIER_FL_NUM',
       'ORIGIN', 'DEST', 'CRS_DEP_TIME', 'DEP_TIME', 'DEP_DELAY',␣
 ↪'DEP_DEL15',
       'TAXI_OUT', 'WHEELS_OFF', 'WHEELS_ON', 'TAXI_IN', 'CRS_ARR_TIME',
       'ARR_TIME', 'ARR_DELAY', 'ARR_DEL15', 'CRS_ELAPSED_TIME',
       'ACTUAL_ELAPSED_TIME', 'AIR_TIME', 'DISTANCE', 'CARRIER_DELAY',
       'WEATHER_DELAY', 'NAS_DELAY', 'SECURITY_DELAY',␣
 ↪'LATE_AIRCRAFT_DELAY',
       'Temperature (°C)', 'Dew Point (°C)', 'Humidity (%)', 'Wind',
       'Wind Speed (km/h)', 'Wind Gust (km/h)', 'Pressure (hPa)',
       'Precip. (mm)', 'Condition']

target = "STATUS"

# Divide data into training, validation and test sets
X_train, X_test, y_train, y_test = train_test_split(df[features],␣
 ↪df[target], test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,␣
 ↪test_size=0.3, random_state=42)

# DataFrame for training data
df_train = pd.DataFrame(X_train, columns=features)
df_train['STATUS'] = y_train

# DataFrame for validation data
df_val = pd.DataFrame(X_val, columns=features)
df_val['STATUS'] = y_val

# DataFrame for test data
df_test = pd.DataFrame(X_test, columns=features)
df_test['STATUS'] = y_test
```

[ ]:
```
df_train
```

[ ]:
```
df_val
```

```
[ ]: df_test
```

1.5.7 Sampling Techniques

```
[ ]: df_train['STATUS'].value_counts()
```

```
[ ]: # Creation of an instance of SMOTE with a specific random string for␣
      ↪reproducibility
     smote = SMOTE(random_state=RANDOM_SEED)

     # Application of SMOTE to resample training data in order to balance the␣
      ↪data
     X_train_resampled, y_train_resampled = smote.fit_resample(X_train,␣
      ↪y_train)

     # Creation of a DataFrame with the resampled data
     df_train_resampled = pd.DataFrame(X_train_resampled, columns=features)
     df_train_resampled['STATUS'] = y_train_resampled
     df_train_resampled
```

```
[ ]: df_train_resampled['STATUS'].value_counts()
```

1.5.8 Feature selection

```
[ ]: # List of relevant columns to be retained in the DataFrame before␣
      ↪selection
     relevant_columns = ['FL_DATE','CRS_DEP_TIME', 'DEP_TIME', 'DEP_DELAY',␣
      ↪'DEP_DEL15','TAXI_OUT', 'WHEELS_OFF', 'CRS_ARR_TIME',
                         'CRS_ELAPSED_TIME', 'DISTANCE','Temperature (°C)',␣
      ↪'Dew Point (°C)', 'Humidity (%)', 'Wind',
                         'Wind Speed (km/h)', 'Wind Gust (km/h)', 'Pressure␣
      ↪(hPa)','Precip. (mm)', 'Condition',
                         'STATUS']

     df_train_resampled = df_train_resampled[relevant_columns]
     df_train_resampled
```

```
[ ]: plt.figure(figsize=(20, 16))

     # Correlation matrix
     corr_matrix = df_train_resampled.corr()
     print(corr_matrix)
```

```python
# Heat map of the correlation matrix
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.show()
```

[ ]:
```python
# Separate features and target variable
X_feature_selection = df_train_resampled.drop(columns=['STATUS'])
y_feature_selection = df_train_resampled['STATUS']

# Use the mutual information score to determine the relationships between␣
↪features and the target variable
selector = SelectKBest(score_func=mutual_info_classif, k='all')
X_results = selector.fit_transform(X_feature_selection,␣
↪y_feature_selection)

# Display scores
mutual_info_scores = pd.DataFrame({'Feature': X_feature_selection.
↪columns, 'Score': selector.scores_})
print(mutual_info_scores.sort_values(by='Score', ascending=False))
```

[ ]:
```python
# Sort scores in descending order
mutual_info_scores = mutual_info_scores.sort_values(by='Score',␣
↪ascending=False)

# Horizontal bar chart with features and their mutual information scores
plt.figure(figsize=(10, 6))
plt.barh(mutual_info_scores["Feature"], mutual_info_scores["Score"])

# Axis configuration
plt.xlabel('Mutual information score', fontsize=14, fontweight='bold')
plt.ylabel('Features', fontsize=14, fontweight='bold')

# Mutual information score threshold
plt.axvline(x=0.15, color='red', linestyle='--', linewidth=2)

plt.show()
```

[ ]:
```python
# Features with a score of more than 0.15 are retained.
features = ['DEP_DELAY', 'WHEELS_OFF', 'TAXI_OUT', 'DEP_TIME', 'FL_DATE',␣
↪'CRS_ARR_TIME', 'DEP_DEL15',
            'CRS_ELAPSED_TIME', 'Pressure (hPa)','CRS_DEP_TIME',␣
↪'Humidity (%)','Temperature (°C)',
            'Dew Point (°C)','Wind Speed (km/
↪h)','DISTANCE','Wind','Condition']
```

```
# Correlation matrix for selected features
corr_matrix = df_train_resampled[features].corr()

plt.figure(figsize=(20, 16))

# Heat map of the correlation matrix
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f',␣
 ↪center=0, linewidths=.5)
plt.show()

# Selection of feature pairs with an absolute correlation greater than 0.8
high_corr = corr_matrix[((corr_matrix >= 0.8) & (corr_matrix < 1.00)) |␣
 ↪((corr_matrix <= -0.8) & (corr_matrix <= -1.00))]
high_corr = high_corr.dropna(how='all', axis=0).dropna(how='all', axis=1)

# Transformation of the correlation matrix into a DataFrame
high_corr_unstacked = high_corr.unstack().dropna().reset_index()
high_corr_unstacked.columns = ['Feature 1', 'Feature 2', 'Correlation␣
 ↪Coefficient']

# Store pairs of highly correlated features
high_corr_unstacked['Ordered Features'] = high_corr_unstacked.
 ↪apply(lambda row: tuple(sorted([row['Feature 1'], row['Feature 2']])),␣
 ↪axis=1)

# Delete duplicate pairs
high_corr_unstacked = high_corr_unstacked.drop_duplicates(subset='Ordered␣
 ↪Features').drop(columns='Ordered Features')

print(high_corr_unstacked)
```

[ ]:
```
# Features selected for training and evaluation of models
features = ['DEP_DELAY', 'WHEELS_OFF', 'TAXI_OUT', 'FL_DATE',␣
 ↪'CRS_ARR_TIME', 'DEP_DEL15',
            'CRS_ELAPSED_TIME', 'Pressure (hPa)','CRS_DEP_TIME',␣
 ↪'Humidity (%)','Temperature (°C)'
            ,'Wind Speed (km/h)','Wind','Condition']

target = 'STATUS'
```

[ ]:
```
df_train_resampled = df_train_resampled[features + [target]]
df_val = df_val[features + [target]]
df_test = df_test[features + [target]]
```

[ ]:
```
df_train_resampled
```

[ ]:
```
df_val
```

[ ]:
```
df_test
```

[ ]:
```
# Saving data
df_train_resampled.to_csv(f"Training_Dataset_{timing}h.csv",index=False)
df_val.to_csv(f"Validation_Dataset_{timing}h.csv",index=False)
df_test.to_csv(f"Testing_Dataset_{timing}h.csv",index=False)
```

Training and evaluation of the models were carried out using the Crescent 2 platform, which allows the use of a Nvidia T4 16GB GPU.

2 Implementation of DL models

2.1 LNN

```
# Importing Libraries

import subprocess

# Ensure all necessary libraries are installed
libraries = ["pandas", "torch", "scikit-learn", "numpy","torchdiffeq"]
for lib in libraries:
    subprocess.run(["pip", "install", lib], check=True)

import time
import pandas as pd
import numpy as np
import torch
import sklearn
import torchdiffeq
import torch.nn as nn
import torch.optim as optim
from torchdiffeq import odeint_adjoint as odeint

# Fixed random seed for reproducibility of results
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
torch.cuda.manual_seed(RANDOM_SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

print(RANDOM_SEED)


# Implementation of Deep Learning Model
```

```python
# Liquid Neural Network (LNN)

# The timing of the weather data can be changed (0h, 2h, 4h, 8h, 16h, 24h or 48h before flight departure)
timing = 0
# timing = 2
# timing = 4
# timing = 8
# timing = 16
# timing = 24
# timing = 48

# Features selected for training and evaluation of models
features = ['DEP_DELAY', 'WHEELS_OFF', 'TAXI_OUT', 'FL_DATE', 'CRS_ARR_TIME', 'DEP_DEL15',
            'CRS_ELAPSED_TIME', 'Pressure (hPa)','CRS_DEP_TIME', 'Humidity (%)','Temperature ( C )'
            ,'Wind Speed (km/h)','Wind','Condition']

target = 'STATUS'

# Load training, validation and test data
df_train = pd.read_csv(f"Training_Dataset_{timing}h.csv")
df_val = pd.read_csv(f"Validation_Dataset_{timing}h.csv")
df_test = pd.read_csv(f"Testing_Dataset_{timing}h.csv")

# Separation of features and target variable
df_train_resampled_X = df_train[features]
df_train_resampled_y = df_train[target]
df_val_resampled_X = df_val[features]
df_val_resampled_y = df_val[target]
df_test_X = df_test[features]
df_test_y = df_test[target]

# Use a GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

# Convert training, validation and test data into PyTorch tensors and transfer them to the GPU
X_train_LNN = torch.tensor(df_train_resampled_X.values, dtype=torch.float32).to(device)
X_val_LNN = torch.tensor(df_val_resampled_X.values, dtype=torch.float32).to(device)
X_test_LNN = torch.tensor(df_test_X.values, dtype=torch.float32).to(device)
y_train_LNN = torch.tensor(df_train_resampled_y.values, dtype=torch.long).view(-1).to(device)
y_val_LNN = torch.tensor(df_val_resampled_y.values, dtype=torch.long).view(-1).to(device)
y_test_LNN = torch.tensor(df_test_y.values, dtype=torch.long).view(-1).to(device)

class LiquidODEFunc(nn.Module):
```

```
    def __init__(self, liquid_size):
        """
        Initialises the ODE function for the LNN model.

        Args:
            liquid_size (int): Size of the liquid layer.

        Returns:
            None.
        """
        super(LiquidODEFunc, self).__init__()

        # Sequential neural network having two linear layers and a Tanh
    activation function
        self.NN = nn.Sequential(
            nn.Linear(liquid_size, liquid_size),
            nn.Tanh(),
            nn.Linear(liquid_size, liquid_size)
        )

    def forward(self, time, input_x):
        """
        Specifies the forward propagation function for computing the ODE.

        Args:
            time (Tensor): Instants of time.
            input_x (Tensor): Inputs to the ODE function.

        Returns:
            Tensor: Model Output after the forward propagation.
        """
        return self.NN(input_x)

class LiquidNeuralNetwork(nn.Module):
    def __init__(self, input_size, liquid_size, output_size, solver='
    dopri5'):
        """
        Initialises the LNN model.

        Args:
            input_size (int): Size of model inputs.
            liquid_size (int): Size of the liquid layer.
            output_size (int): Size of model outputs.
            solver (str): Solver method for ODEs.

        Returns:
            None.
        """
        super(LiquidNeuralNetwork, self).__init__()
        self.liquid_size = liquid_size
        # Input Layer
        self.input_layer = nn.Linear(input_size, liquid_size)
        # Dropout for regularisation
        self.dropout = nn.Dropout(0.5) # Put it in comments if it is not
```

```
used
        self.ode_func = LiquidODEFunc(liquid_size)
        # Output Layer
        self.output_layer = nn.Linear(liquid_size, output_size)
        self.solver = solver

    def forward(self, x, time_span):
        """
        Defines the forward propagation function for the LNN model.

        Args:
            x (Tensor): Model inputs.
            time_span (tuple): Time span for solving the ODE.

        Returns:
            Tensor: Network output after propagation of ODE states and the
    output layer.
        """
        # Input propagation through the input layer
        x = self.input_layer(x)
        # Dropout application
        x = self.dropout(x) # Put it in comments if it is not used
        t = torch.linspace(time_span[0], time_span[1], 10).to(x.device)
        # ODE resolution
        ode_output = torchdiffeq.odeint(self.ode_func, x, t, method=self.
solver)
        # Take the last state
        x = ode_output[-1]
        # Propagation of the last state through the output layer
        return self.output_layer(x)


def train_model(model, train_loader, val_loader, criterion, optimizer,
    num_epochs, device):
    """
    Train a model on training data and validate its performance on
    validation data.

    Args:
        model (nn.Module): Model to be trained.
        train_loader (DataLoader): DataLoader for training data.
        val_loader (DataLoader): DataLoader for validation data.
        criterion (nn.Module): Loss function used.
        optimizer (torch.optim.Optimizer): Optimizer used.
        num_epochs (int): Number of epochs.
        device (torch.device): The system on which the model is trained.

    Returns:
        None.
    """
    for epoch in range(num_epochs):
        # Put the model in training mode
        model.train()
```

```
        # Initialise the training loss for each epoch
        train_loss = 0.0

        # It re on training data
        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            # Reset optimizer gradients
            optimizer.zero_grad()

            # Forward propagation of the model
            outputs = model(inputs, (0, 1))

            # Calculate the loss
            loss = criterion(outputs, targets.view(-1))

            # Update model weights
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        # Calculates the average training loss for the epoch
        avg_train_loss = train_loss / len(train_loader)

        # Validate the model on validation data
        val_loss, val_accuracy = evaluate_model(model, val_loader,
    criterion, device)
        print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {avg_train_loss
    :.4f}, Val Loss: {val_loss:.4f}')


def evaluate_model(model, data_loader, criterion, device, print_metrics=
    False):
    """
    Evaluate a model on validation/test data.

    Args:
        model (nn.Module): Model to be evaluated..
        data_loader (DataLoader): DataLoader for validation/test data.
        criterion (nn.Module): Loss function used.
        optimizer (torch.optim.Optimizer): Optimizer used.
        device (torch.device): The system on which the model is trained.
        print_metrics (bool): If True, displays the confusion matrix and
    the classification report.

    Returns:
        Tuple: The average loss and global accuracy of the model.
    """

    # Put the model in evaluation mode
    model.eval()

    # Set total loss to zero
```

```
    total_loss = 0.0

    # Lists to store all true and predicted values
    all_targets = []
    all_preds = []

    with torch.no_grad():
        for inputs, targets in data_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            # Forward propagation of the model to obtain predictions
            outputs = model(inputs, (0, 1))

            # Calculate the loss
            loss = criterion(outputs, targets.view(-1))

            total_loss += loss.item()

            # Add targets and predictions to their lists
            all_targets.extend(targets.cpu().numpy())
            all_preds.extend(outputs.argmax(dim=1).cpu().numpy())

    # Calculate the average loss
    avg_loss = total_loss / len(data_loader)

    # Calculate the global accuracy
    accuracy = sklearn.metrics.accuracy_score(all_targets, all_preds)

    if print_metrics:
        # Display the confusion matrix
        print("\nConfusion Matrix:")
        print(sklearn.metrics.confusion_matrix(all_targets, all_preds))
        # Display the classification report
        print("\nClassification Report:")
        print(sklearn.metrics.classification_report(all_targets, all_preds
   ))
        # Display the global accuracy
        print(f"\nGlobal Accuracy: {accuracy:.4f}")

    return avg_loss, accuracy

# Hyperparameters
input_size = df_train_resampled_X.shape[1] # Fixed size of the input layer
output_size = 3  # Fixed size of the output layer
solver = 'dopri5' # Fixed value for the solver

# The values of the hyperparameters below can be changed to test other
   configurations
hidden_size = 350 # Set the size of the liquid layer
num_epochs = 50 # Set the number of epochs
batch_size = 1024 # Set the batch size
learning_rate = 0.0001 # Set the learning rate value
```

```python
print(f"{timing}h")
print(f"hidden_size = {hidden_size}")
print(f"epochs = {num_epochs}")
print(f"batch_size = {batch_size}")
print(f"learning_rate = {learning_rate}")
print(f"dropout = 0.5") # Put it in comments if Dropout is not used

# Setup
model = LiquidNeuralNetwork(input_size, hidden_size, output_size, solver).
    to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Data loaders
train_data_LNN = torch.utils.data.TensorDataset(X_train_LNN, y_train_LNN)
train_loader_LNN = torch.utils.data.DataLoader(train_data_LNN, batch_size=
    batch_size, shuffle=True)
val_data_LNN = torch.utils.data.TensorDataset(X_val_LNN, y_val_LNN)
val_loader_LNN = torch.utils.data.DataLoader(val_data_LNN, batch_size=
    batch_size)
test_data_LNN = torch.utils.data.TensorDataset(X_test_LNN, y_test_LNN)
test_loader_LNN = torch.utils.data.DataLoader(test_data_LNN, batch_size=
    batch_size)

# Train the model
start_time = time.time()
train_model(model, train_loader_LNN, val_loader_LNN, criterion, optimizer,
     num_epochs, device)
end_time = time.time()
total_train_time = end_time - start_time
print(f'Total training time: {total_train_time:.4f}s')

# Evaluate on test set
start_time = time.time()
test_loss, test_acc = evaluate_model(model,  test_loader_LNN, criterion,
    device, print_metrics=True)
end_time = time.time()
total_evaluation_time = end_time - start_time
print(f'Total evaluation time: {total_evaluation_time:.4f}s')
print(f'Test Loss: {test_loss:.4f}')
```

2.2 LSTM

```python
# Importing Libraries

import subprocess

# Ensure all necessary libraries are installed
libraries = ["pandas", "torch", "scikit-learn", "numpy"]
for lib in libraries:
    subprocess.run(["pip", "install", lib], check=True)

import time
import itertools
```

```
import pandas as pd
import numpy as np
import torch
import sklearn
import torch.nn
import torch.optim

# Fixed random seed for reproducibility of results
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
torch.cuda.manual_seed(RANDOM_SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

print(RANDOM_SEED)


# Implementation of Deep Learning Model
# Long Short-Term Memory (LSTM)

# The timing of the weather data can be changed (0h, 2h, 4h, 8h, 16h, 24h
   or 48h before flight departure)
timing = 0
# timing = 2
# timing = 4
# timing = 8
# timing = 16
# timing = 24
# timing = 48

# Features selected for training and evaluation of models
features = ['DEP_DELAY', 'WHEELS_OFF', 'TAXI_OUT', 'FL_DATE', '
   CRS_ARR_TIME', 'DEP_DEL15',
            'CRS_ELAPSED_TIME', 'Pressure (hPa)','CRS_DEP_TIME', 'Humidity
    (%)','Temperature ( C )'
            ,'Wind Speed (km/h)','Wind','Condition']

target = 'STATUS'

# Load training, validation and test data
df_train = pd.read_csv(f"Training_Dataset_{timing}h.csv")
df_val = pd.read_csv(f"Validation_Dataset_{timing}h.csv")
df_test = pd.read_csv(f"Testing_Dataset_{timing}h.csv")

# Separation of features and target variable
df_train_resampled_X = df_train[features]
df_train_resampled_y = df_train[target]
df_val_resampled_X = df_val[features]
df_val_resampled_y = df_val[target]
df_test_X = df_test[features]
df_test_y = df_test[target]

# Use a GPU if available
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

class LSTM_Model(torch.nn.Module):
    def __init__(self, input_size, LSTM_size, output_size, num_layers,
   dropout):
        """
        Initialises the LSTM modle.

        Args:
            input_size (int): Size of model inputs.
            LSTM_size (int): Size of the LSTM layers.
            output_size (int): Size of model outputs.
            num_layers (int): Number of stacked LSTM layers.
            dropout (float): Dropout rate used.
        """
        super(LSTM_Model, self).__init__()
        self.LSTM_size = LSTM_size
        self.num_layers = num_layers

        # Definition of the LSTM layer
        self.lstm = torch.nn.LSTM(input_size, LSTM_size, num_layers,
   batch_first=True, dropout=dropout)

        # Output layer
        self.fc = torch.nn.Linear(LSTM_size, output_size)

    def forward(self, x):
        """
        Defines the forward propagation function for the LNN model.

        Args:
            x (Tensor): Model inputs.

        Returns:
            Tensor: Model output after the forward propagation.
        """
        # Initialise hidden state with zeros
        h0 = torch.zeros(self.num_layers, x.size(0), self.LSTM_size).to(x.
   device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.LSTM_size).to(x.
   device)

        out, _ = self.lstm(x, (h0, c0))

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out

def prepare_lstm_data(df_X, df_y, sequence_length, input_size, device):
    """
    Prepares data for the LSTM model

    Args:
```

```
        df_X (pd.DataFrame): A DataFrame containing the input features.
        df_y (pd.Series): A Serie contenaining target labels.
        sequence_length (int): Length of time sequences used.
        input_size (int): Size of model inputs.
        device (torch.device): The system to which the tensors are
    transferred.

    Returns:
        Tuple: Two PyTorch tensors for training and evaluating the LSTM
    model.
    """
    # Calculation of the number of possible sequences
    num_samples = df_X.shape[0]
    new_num_samples = num_samples // sequence_length

    # Preparing input data
    X_LSTM = torch.tensor(df_X.values[:new_num_samples * sequence_length],
     dtype=torch.float32)
    X_LSTM = X_LSTM.view(new_num_samples, sequence_length, input_size).to(
    device)

    # Preparation of target data
    y_LSTM = torch.tensor(df_y.values[:new_num_samples * sequence_length],
     dtype=torch.long)
    y_LSTM = y_LSTM.view(new_num_samples, sequence_length).to(device)

    return X_LSTM, y_LSTM

input_size = df_train_resampled_X.shape[1]
sequence_length = 1

# Prepare training, validation and test data
X_train_LSTM, y_train_LSTM = prepare_lstm_data(df_train_resampled_X,
    df_train_resampled_y, sequence_length, input_size, device)
X_val_LSTM, y_val_LSTM = prepare_lstm_data(df_val_resampled_X,
    df_val_resampled_y, sequence_length, input_size, device)
X_test_LSTM, y_test_LSTM = prepare_lstm_data(df_test_X, df_test_y,
    sequence_length, input_size, device)

def train_model(model, train_loader, val_loader, criterion, optimizer,
    num_epochs, device):
    """
    Train a model on training data and validate its performance on
    validation data.

    Args:
        model (nn.Module): Model to be trained.
        train_loader (DataLoader): DataLoader for training data.
        val_loader (DataLoader): DataLoader for validation data.
        criterion (nn.Module): Loss function used.
        optimizer (torch.optim.Optimizer): Optimizer used.
        num_epochs (int): Number of epochs.
        device (torch.device): The system on which the model is trained.
```

```
    Returns:
        None.
    """
    for epoch in range(num_epochs):
        # Put the model in training mode
        model.train()

        # Initialise the training loss for each epoch
        train_loss = 0.0

        # It re on training data
        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            # Reset optimizer gradients
            optimizer.zero_grad()

            # Forward propagation of the model
            outputs = model(inputs)

            # Calculate the loss
            loss = criterion(outputs, targets.view(-1))

            # Update model weights
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        # Calculates the average training loss for the epoch
        avg_train_loss = train_loss / len(train_loader)

        # Validate the model on validation data
        val_loss, val_accuracy = evaluate_model(model, val_loader,
   criterion, device)
        print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {avg_train_loss
   :.4f}, Val Loss: {val_loss:.4f}')


def evaluate_model(model, data_loader, criterion, device, print_metrics=
   False):
    """
    Evaluate a model on validation/test data.

    Args:
        model (nn.Module): Model to be evaluated..
        data_loader (DataLoader): DataLoader for validation/test data.
        criterion (nn.Module): Loss function used.
        optimizer (torch.optim.Optimizer): Optimizer used.
        device (torch.device): The system on which the model is trained.
        print_metrics (bool): If True, displays the confusion matrix and
   the classification report.

    Returns:
```

```
        Tuple: The average loss and global accuracy of the model.
    """

    # Put the model in evaluation mode
    model.eval()

    # Set total loss to zero
    total_loss = 0.0

    # Lists to store all true and predicted values
    all_targets = []
    all_preds = []

    with torch.no_grad():
        for inputs, targets in data_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            # Forward propagation of the model to obtain predictions
            outputs = model(inputs)

            # Calculate the loss
            loss = criterion(outputs, targets.view(-1))

            total_loss += loss.item()

            # Add targets and predictions to their lists
            all_targets.extend(targets.cpu().numpy())
            all_preds.extend(outputs.argmax(dim=1).cpu().numpy())

    # Calculate the average loss
    avg_loss = total_loss / len(data_loader)

    # Calculate the global accuracy
    accuracy = sklearn.metrics.accuracy_score(all_targets, all_preds)

    if print_metrics:
        # Display the confusion matrix
        print("\nConfusion Matrix:")
        print(sklearn.metrics.confusion_matrix(all_targets, all_preds))
        # Display the classification report
        print("\nClassification Report:")
        print(sklearn.metrics.classification_report(all_targets, all_preds
   ))
        # Display the global accuracy
        print(f"\nGlobal Accuracy: {accuracy:.4f}")

    return avg_loss, accuracy

# Hyperparameters
input_size = df_train_resampled_X.shape[1] # Fixed size of the input layer
output_size = 3  # Fixed size of the output layer
dropout = 0.5 # Fixed dropout rate
batch_size = 1024 # Fixed batch size
```

```python
# Use the same range of hyperparameter values used for each grid search
    performed to ensure reproducibility of results
# Grid search for the LSTM model has been split into 2 parts to avoid
    having a long execution time for this optimisation algorithm

# Uncomment one of the 2 configurations to carry out the grid search

# Grid search 1
'''
num_epochs_list = [20, 50]
num_layers_list = [2, 5, 10]
learning_rate_list = [0.001, 0.0001]
hidden_size_list = [50, 100]
'''
# Grid search 2
num_epochs_list = [20, 50]
num_layers_list = [2, 5, 10]
learning_rate_list = [0.001, 0.0001]
hidden_size_list = [150]



# Generate all possible combinations of hyperparameters tested during the
    grid search
all_combinations = list(itertools.product(num_epochs_list, num_layers_list
    , learning_rate_list, hidden_size_list))

best_accuracy = 0.0
best_params = {}

print(f"{timing}h")

# Grid search
for num_epochs, num_layers, learning_rate, hidden_size in all_combinations
    :
  print(f"Testing configuration: batch_size={batch_size}, num_epochs={
    num_epochs}, num_layers={num_layers}, learning_rate={learning_rate},
    dropout={dropout}, hidden_size={hidden_size}")

  # Setup
  model = LSTM_Model(input_size, hidden_size, output_size, num_layers,
    dropout).to(device)
  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

  # Data loaders
  train_dataset_LSTM = torch.utils.data.TensorDataset(X_train_LSTM,
    y_train_LSTM)
  train_loader_LSTM = torch.utils.data.DataLoader(train_dataset_LSTM,
    batch_size=batch_size, shuffle=True)
  val_dataset_LSTM = torch.utils.data.TensorDataset(X_val_LSTM, y_val_LSTM
    )
  val_loader_LSTM = torch.utils.data.DataLoader(val_dataset_LSTM,
    batch_size=batch_size)
```

```
  test_dataset_LSTM = torch.utils.data.TensorDataset(X_test_LSTM,
   y_test_LSTM)
  test_loader_LSTM = torch.utils.data.DataLoader(test_dataset_LSTM,
   batch_size=batch_size)

  # Train the model
  start_time = time.time()
  train_model(model, train_loader_LSTM, val_loader_LSTM, criterion,
   optimizer, num_epochs, device)
  end_time = time.time()
  total_train_time = end_time - start_time
  print(f'Total training time: {total_train_time:.4f}s')

  # Evaluate on validation set
  start_time = time.time()
  test_loss, test_accuracy = evaluate_model(model, test_loader_LSTM,
   criterion, device, print_metrics=True)
  end_time = time.time()
  total_evaluation_time = end_time - start_time
  print(f'Total evaluation time: {total_evaluation_time:.4f}s')
  print(f'Test Loss: {test_loss:.4f}')

  # Updating the best results
  if test_accuracy > best_accuracy:
      best_accuracy = test_accuracy
      best_params = {
          "batch_size": batch_size,
          "num_epochs": num_epochs,
          "num_layers": num_layers,
          "learning_rate": learning_rate,
          "dropout": dropout,
          "hidden_size": hidden_size
      }

print(f"Best configuration: {best_params} with accuracy: {best_accuracy:.4
   f}")
```

2.3 MLP

```
# Importing Libraries

import subprocess

# Ensure all necessary libraries are installed
libraries = ["pandas", "torch", "scikit-learn", "numpy"]
for lib in libraries:
    subprocess.run(["pip", "install", lib], check=True)

import time
import itertools
import pandas as pd
import numpy as np
import torch
import sklearn
```

```python
import torch.nn
import torch.optim

# Fixed random seed for reproducibility of results
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
torch.cuda.manual_seed(RANDOM_SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

print(RANDOM_SEED)

# Implementation of Deep Learning Model
# MultiLayer Perceptron (MLP)

# The timing of the weather data can be changed (0h, 2h, 4h, 8h, 16h, 24h
    or 48h before flight departure)
timing = 0
# timing = 2
# timing = 4
# timing = 8
# timing = 16
# timing = 24
# timing = 48

# Features selected for training and evaluation of models
features = ['DEP_DELAY', 'WHEELS_OFF', 'TAXI_OUT', 'FL_DATE', '
    CRS_ARR_TIME', 'DEP_DEL15',
            'CRS_ELAPSED_TIME', 'Pressure (hPa)','CRS_DEP_TIME', 'Humidity
     (%)','Temperature ( C )'
            ,'Wind Speed (km/h)','Wind','Condition']

target = 'STATUS'

# Load training, validation and test data
df_train = pd.read_csv(f"Training_Dataset_{timing}h.csv")
df_val = pd.read_csv(f"Validation_Dataset_{timing}h.csv")
df_test = pd.read_csv(f"Testing_Dataset_{timing}h.csv")

# Separation of features and target variable
df_train_resampled_X = df_train[features]
df_train_resampled_y = df_train[target]
df_val_resampled_X = df_val[features]
df_val_resampled_y = df_val[target]
df_test_X = df_test[features]
df_test_y = df_test[target]

# Use a GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

# Convert training, validation and test data into PyTorch tensors and
    transfer them to the GPU
```

```
X_train_MLP = torch.tensor(df_train_resampled_X.values, dtype=torch.float32).to(device)
X_val_MLP = torch.tensor(df_val_resampled_X.values, dtype=torch.float32).to(device)
X_test_MLP = torch.tensor(df_test_X.values, dtype=torch.float32).to(device)
y_train_MLP = torch.tensor(df_train_resampled_y.values, dtype=torch.long).view(-1).to(device)
y_val_MLP = torch.tensor(df_val_resampled_y.values, dtype=torch.long).view(-1).to(device)
y_test_MLP = torch.tensor(df_test_y.values, dtype=torch.long).view(-1).to(device)

class MLP_Model(torch.nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size, dropout=0.5):
        """
        Initialises the MLP model.

        Args:
            input_size (int): Size of model inputs.
            hidden_sizes (list of int): List of hidden layer sizes.
            output_size (int): Size of model outputs.
            dropout (float, optional): Dropout rate used.

        Returns:
            None.
        """
        super(MLP_Model, self).__init__()
        # List for storing layers
        layers = []
        # Size of the previous layer
        # It is initialised to the inputs size
        prev_size = input_size

        # Creation of hidden layers with a linear layer, a LeakyRelu activation function and a dropout layer
        for hidden_size in hidden_sizes:
            layers.append(torch.nn.Linear(prev_size, hidden_size))
            layers.append(torch.nn.LeakyReLU())
            layers.append(torch.nn.Dropout(dropout))
            # Update the previous layer size
            prev_size = hidden_size

        # Output layer
        layers.append(torch.nn.Linear(prev_size, output_size))

        self.mlp = torch.nn.Sequential(*layers)

    def forward(self, x):
        """
        Defines the forward propagation function for the MLP model.

        Args:
```

```
            x (Tensor): Model inputs.

        Returns:
            Tensor: Model output after the foward propagation.
        """
        return self.mlp(x)

def train_model(model, train_loader, val_loader, criterion, optimizer,
   num_epochs, device):
    """
    Train a model on training data and validate its performance on
   validation data.

    Args:
        model (nn.Module): Model to be trained.
        train_loader (DataLoader): DataLoader for training data.
        val_loader (DataLoader): DataLoader for validation data.
        criterion (nn.Module): Loss function used.
        optimizer (torch.optim.Optimizer): Optimizer used.
        num_epochs (int): Number of epochs.
        device (torch.device): The system on which the model is trained.

    Returns:
        None.
    """
    for epoch in range(num_epochs):
        # Put the model in training mode
        model.train()

        # Initialise the training loss for each epoch
        train_loss = 0.0

        # It re on training data
        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            # Reset optimizer gradients
            optimizer.zero_grad()

            # Forward propagation of the model
            outputs = model(inputs)

            # Calculate the loss
            loss = criterion(outputs, targets.view(-1))

            # Update model weights
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        # Calculates the average training loss for the epoch
        avg_train_loss = train_loss / len(train_loader)
```

```
        # Validate the model on validation data
        val_loss, val_accuracy = evaluate_model(model, val_loader,
    criterion, device)
        print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {avg_train_loss
    :.4f}, Val Loss: {val_loss:.4f}')


def evaluate_model(model, data_loader, criterion, device, print_metrics=
    False):
    """
    Evaluate a model on validation/test data.

    Args:
        model (nn.Module): Model to be evaluated..
        data_loader (DataLoader): DataLoader for validation/test data.
        criterion (nn.Module): Loss function used.
        optimizer (torch.optim.Optimizer): Optimizer used.
        device (torch.device): The system on which the model is trained.
        print_metrics (bool): If True, displays the confusion matrix and
    the classification report.

    Returns:
        Tuple: The average loss and global accuracy of the model.
    """

    # Put the model in evaluation mode
    model.eval()

    # Set total loss to zero
    total_loss = 0.0

    # Lists to store all true and predicted values
    all_targets = []
    all_preds = []

    with torch.no_grad():
        for inputs, targets in data_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            # Forward propagation of the model to obtain predictions
            outputs = model(inputs)

            # Calculate the loss
            loss = criterion(outputs, targets.view(-1))

            total_loss += loss.item()

            # Add targets and predictions to their lists
            all_targets.extend(targets.cpu().numpy())
            all_preds.extend(outputs.argmax(dim=1).cpu().numpy())

    # Calculate the average loss
    avg_loss = total_loss / len(data_loader)
```

```
    # Calculate the global accuracy
    accuracy = sklearn.metrics.accuracy_score(all_targets, all_preds)

    if print_metrics:
        # Display the confusion matrix
        print("\nConfusion Matrix:")
        print(sklearn.metrics.confusion_matrix(all_targets, all_preds))
        # Display the classification report
        print("\nClassification Report:")
        print(sklearn.metrics.classification_report(all_targets, all_preds
   ))
        # Display the global accuracy
        print(f"\nGlobal Accuracy: {accuracy:.4f}")

    return avg_loss, accuracy

# Hyperparameters
input_size = df_train_resampled_X.shape[1] # Fixed size of the input layer
output_size = 3  # Fixed size of the output layer
dropout = 0.5 # Fixed dropout rate
num_layers = 3 # Fixed number of layers

# Use the same range of hyperparameter values used for each grid search
   performed to ensure reproducibility of results
# Grid search for the MLP model has been split into 4 parts to avoid
   having a long execution time for this optimisation algorithm

# Uncomment one of the 4 configurations to carry out the grid search

# Grid search 1
'''
hidden_sizes_list = [[128, 64, 32], [256, 128, 64], [512, 256, 128]]
num_epochs_list = [10, 20, 30]
learning_rate_list = [0.001, 0.0001]
batch_size = 1024
'''
# Grid search 2
'''
hidden_sizes_list = [[128, 64, 32], [256, 128, 64], [512, 256, 128]]
num_epochs_list = [10, 20, 30]
learning_rate_list = [0.001, 0.0001]
batch_size = 512
'''
# Grid search 3
'''
hidden_sizes_list = [[128, 64, 32], [256, 128, 64], [512, 256, 128]]
num_epochs_list = [50]
learning_rate_list = [0.001, 0.0001]
batch_size = 512
'''
# Grid search 4
hidden_sizes_list = [[128, 64, 32], [256, 128, 64], [512, 256, 128]]
num_epochs_list = [50]
learning_rate_list = [0.001, 0.0001]
```

```python
batch_size = 1024


# Generate all possible combinations of hyperparameters tested during the
    grid search
all_combinations = list(itertools.product(num_epochs_list,
    learning_rate_list, hidden_sizes_list))

best_accuracy = 0.0
best_params = {}

print(f"{timing}h")

# Grid search
for num_epochs, learning_rate, hidden_sizes in all_combinations:
  print(f"Testing configuration: batch_size={batch_size}, num_epochs={
   num_epochs}, learning_rate={learning_rate}, dropout={dropout},
   hidden_size={hidden_sizes}")

  # Setup
  model = MLP_Model(input_size, hidden_sizes, output_size, dropout).to(
   device)
  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

  # Data loaders
  train_dataset_MLP = torch.utils.data.TensorDataset(X_train_MLP,
   y_train_MLP)
  train_loader_MLP = torch.utils.data.DataLoader(train_dataset_MLP,
   batch_size=batch_size, shuffle=True)
  val_dataset_MLP = torch.utils.data.TensorDataset(X_val_MLP, y_val_MLP)
  val_loader_MLP = torch.utils.data.DataLoader(val_dataset_MLP, batch_size
   =batch_size)
  test_dataset_MLP = torch.utils.data.TensorDataset(X_test_MLP, y_test_MLP
   )
  test_loader_MLP = torch.utils.data.DataLoader(test_dataset_MLP,
   batch_size=batch_size)

  # Train the model
  start_time = time.time()
  train_model(model, train_loader_MLP, val_loader_MLP, criterion,
   optimizer, num_epochs, device)
  end_time = time.time()
  total_train_time = end_time - start_time
  print(f'Total training time: {total_train_time:.4f}s')

  # Evaluate on validation set
  start_time = time.time()
  test_loss, test_accuracy = evaluate_model(model, test_loader_MLP,
   criterion, device, print_metrics=True)
  end_time = time.time()
  total_evaluation_time = end_time - start_time
  print(f'Total evaluation time: {total_evaluation_time:.4f}s')
  print(f'Test Loss: {test_loss:.4f}')
```

```python
    # Updating the best results
    if test_accuracy > best_accuracy:
        best_accuracy = test_accuracy
        best_params = {
            "num_epochs": num_epochs,
            "learning_rate": learning_rate,
            "hidden_sizes": hidden_sizes,
            "dropout": dropout,
            "batch_size": batch_size
        }

print(f"Best configuration: {best_params} with accuracy: {best_accuracy:.4
    f}")
```

2.4 Sample scheduler script

```bash
#!/bin/bash
##
## GPU submission script for PBS on CR2
## -------------------------------------------
##
## Follow the 6 steps below to configure your job
##
## STEP 1:
##
## Enter a job name after the -N on the line below:
##
#PBS -N gpu__example_0
##
## STEP 2:
##
## Select the number of cpus/cores and GPUs required by modifying the #PBS
    -l select line below
##
## The Maximum value for ncpus is 8 and mpiprocs MUST be the same value as
    ncpus.
## The Maximum value for ngpus is 1
## e.g.  1 GPU and 8 CPUs : select=1:ncpus=8:mpiprocs=8;ngpus=1
##
#PBS -l select=1:ncpus=8:mpiprocs=8:ngpus=1:mem=64g
##
## STEP 3:
##
## The queue for GPU jobs is defined in the #PBS -q line below
##
#PBS -q gpu_T4
#
## The default walltime in the gpu_A100 queue is one day(24 hours)
## The maximum walltime in the gpu_A100 queue is five days(120 hours)
## In order to increase the walltime modify the #PBS -l walltime line
   below
## and remove one of the leading # characters
##
```

```
##PBS -l walltime=24:00:00
##
## STEP 4:
##
## Replace the hpc@cranfield.ac.uk email address
## with your Cranfield email address on the #PBS -M line below:
## Your email address is NOT your username
##
#PBS -m abe
#PBS -M majuran.chandrakumar.255@cranfield.ac.uk
##
## ====================================
## DO NOT CHANGE THE LINES BETWEEN HERE
## ====================================
#PBS -j oe
#PBS -v "CUDA_VISIBLE_DEVICES="
#PBS -W sandbox=PRIVATE
#PBS -k n
ln -s $PWD $PBS_O_WORKDIR/$PBS_JOBID
## Allocated gpu(s)
echo CUDA_VISIBLE_DEVICES=$CUDA_VISIBLE_DEVICES
## Change to working directory
cd $PBS_O_WORKDIR
## Calculate number of CPUs and GPUs
export cpus=`cat $PBS_NODEFILE | wc -l`
export gpus=`echo $CUDA_VISIBLE_DEVICES|awk -F"," '{print NF}'`
## ========
## AND HERE
## ========
##
## STEP 5:
##
## Load the production USE
module use /apps/modules/all
##  Load the default application environment
##
module load Anaconda3/2022.10
##
## STEP 6:
##
## Run gpu application
##
## Put correct parameters and cuda application in the line below:
##
python Chandrakumar_s419255_CodingFile_LNN_Model_MScCSTE.py # Change the
   name of the python file

## Tidy up the log directory
## DO NOT CHANGE THE LINE BELOW
## ============================
rm $PBS_O_WORKDIR/$PBS_JOBID
#
```