COMP1878 Python Project Report

Name: Sean White ID: 001354814

Summary of Implemented Tasks

This is an overview of each task, broken down into the bullet-pointed subtasks, detailing the techniques used to implement them. Screenshots are provided of the relevant methods to enhance the explanation. As shown, 'Exception' and 'ValueError' are used throughout for error handling.

Task 1 – Class Creation

Created a 'DataPoint' class which retrieves and returns the met office weather forecast data from both the sitelist and localised forecast pages.

1. __init__ method takes the API key as input so the whole class in instantiated by this. The URL elements to be used throughout the class are loaded using the 'dotenv' and 'os' libraries which are established with 'self.'

2. Uses 'requests.get' to access met data, constructing the URL around 'sitelist'. If this works it checks the status code is 200 and uses 'json()' to read the data. Iterates through 'Location' in the 'sitelist' dictionary to return the 'id' of the given 'location_name'.

```
def get_location_id(self, location_name):
    """
    Call this function with a valid location name from the met office data for which it will retu
    corresponding location id.
    """
    try:
        response = requests.get(f'{self.Base}sitelist{self.Endpoint}{self.API}')
        if response.status_code == 200:
            data = response.json()
            for location in data['Locations']['Location']:
                if location['name'] == location_name:
                    return location['id']
            raise ValueError(f'Location "{location_name}" not found - check this is a valid name'
        else:
            print(f'Error: status code is {response.status_code}')
    except Exception as e:
        print(f'Error: {e}')
```

3. This method accesses the data in the same way as the previous method, however, iterates through the given IDs and uses them in URL to access site-specific pages. As can be seen below, the method must get through the different layers of the dictionary to reach 'Rep' which is the required data this is then added to the full 'pd.DataFrame' using

'pd.concat.' To make it more presentable I moved the location and day to the start of the df using '.insert' and specifying the index.

```
def get_forecast_data(self, location_ids):
    """
    This function will, using a list of location ids, return a dataframe with the 3hourly, 5-day forecast provided
    by the met office for each of the locations. There is no need for any user input to call this function.
    """
    forecast_data = pd.DataFrame()
    for location_id in location_ids:
        try:
            response = requests.get(f'{self.Base}{location_id}{self.Endpoint}{self.API}')
            if response.status_code == 200:
                data = response.json()
                location_name = data['SiteRep']['DV']['Location']['name']
                for day_data in data['SiteRep']['DV']['Location']['Period']:
                    forecast = day_data['Rep']
                    forecast_df = pd.DataFrame(forecast)
                    forecast_df.insert( loc: 0,  column: 'Location', location_name)
                    forecast_df.insert( loc: 1,  column: 'Day', day_data['value'])
                    forecast_data = pd.concat( objs: [forecast_data, forecast_df], ignore_index=True)
            else:
                print(f'error: status code is {response.status_code}')
                print('Location ID(s) may not be valid')
        except Exception as e:
            print(f'Error fetching data for location {location_id}: {e}')
            print('Try checking URL elements are correct')
    return forecast_data
```

Task 2 – Data Handling

Added a method to save data in csv format and methods to make the data more readable by altering headers and values removing redundant columns. Then created the 'main.py' file for importing this class.

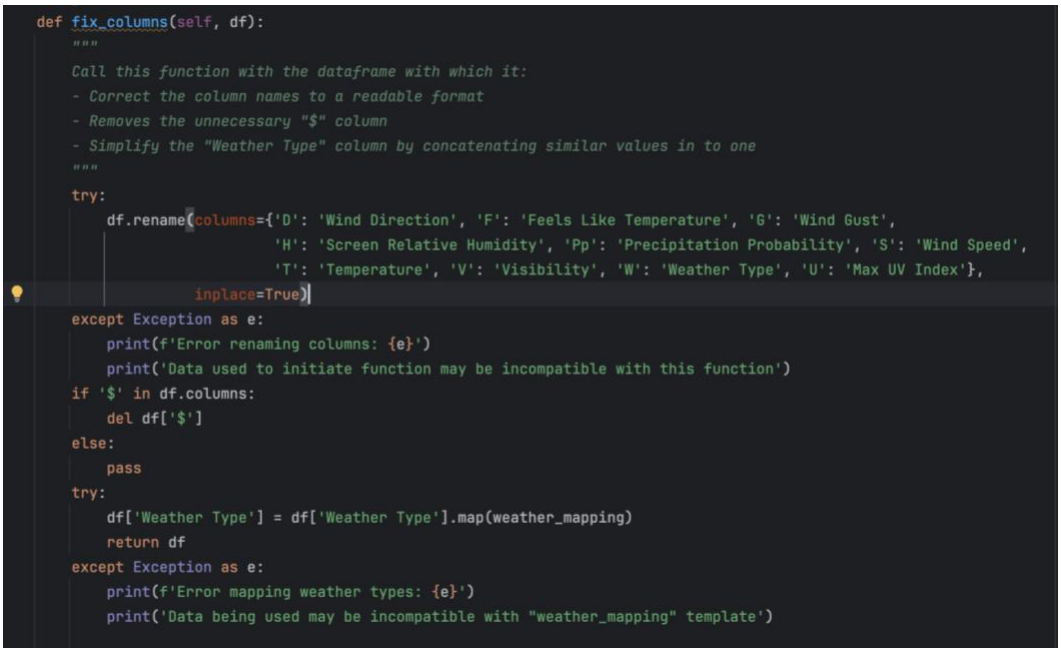
1. 'save_to_csv' method uses '.to_csv' to create the csv and 'os.path.join' to create a file path from a file name and folder path. Before this it uses 'os.path.exists' and 'os.makedirs' to check if the given folder path exists and create the folder if not.

```
def save_to_csv(self, data, file_name, folder_path):
    """
    Saves data as a csv file in the given folder. Please call this function with the following:
    - A pandas dataframe
    - A name for the csv file
    - A path to the folder you'd like to store the csv file in
    """
    try:
        if not os.path.exists(folder_path):
            os.makedirs(folder_path)

        file_path = os.path.join(folder_path, file_name)
        data.to_csv(file_path, index=False)
        print(f'{file_name} successfully saved to {folder_path}')
    except Exception as e:
        print(f'Error saving data to CSV: {e}')
        print('Check folder path exists & file name ends in ".csv"')
```

2. Used ‘df.rename’ with the ‘key:value’ template provided to change the columns headings succinctly. The ‘fix_columns’ method takes the df as input so it can be used independently of the class.

3. Used ‘del’ and indexed ‘$’ to remove the column. I felt it had a similar functionality within the class as renaming the headings, so it’s included in ‘fix_columns.’ In the case that there’s no ‘$’ column an ‘if’ condition passes over this part to make the function more broadly useable.



```
def fix_columns(self, df):
    """
    Call this function with the dataframe with which it:
    - Correct the column names to a readable format
    - Removes the unnecessary "$" column
    - Simplify the "Weather Type" column by concatenating similar values in to one
    """
    try:
        df.rename(columns={'D': 'Wind Direction', 'F': 'Feels Like Temperature', 'G': 'Wind Gust',
                           'H': 'Screen Relative Humidity', 'Pp': 'Precipitation Probability', 'S': 'Wind Speed',
                           'T': 'Temperature', 'V': 'Visibility', 'W': 'Weather Type', 'U': 'Max UV Index'},
                  inplace=True)
    except Exception as e:
        print(f'Error renaming columns: {e}')
        print('Data used to initiate function may be incompatible with this function')
    if '$' in df.columns:
        del df['$']
    else:
        pass
    try:
        df['Weather Type'] = df['Weather Type'].map(weather_mapping)
        return df
    except Exception as e:
        print(f'Error mapping weather types: {e}')
        print('Data being used may be incompatible with "weather_mapping" template')
```

4. Put the ‘weather_dict’ and ‘weather_mapping’ in a utilities file which is imported for use in this method as they were too large to store within. I then mapped the ‘weather_mapping’ values to the ‘Weather Type’ column.

5. The separate script is the ‘main.py’ file, containing a function which only runs if not imported. The API is retrieved with the ‘dotenv’ and ‘os’ libraries from the ‘.env’ file. The DataPoint class is imported, and the 30 location IDs are given to create the raw data with ‘get_forecast_data’ which is saved in the ‘raw’ folder in ‘data’. This data is then used as input for the ‘fix_columns’ method which is saved in the ‘processed’ folder with a different name.

```python
def main():
    #Retrieve API
    dotenv_path = find_dotenv()
    load_dotenv(dotenv_path)
    api_key = os.environ.get('API')
    #Instantiate DataPoint with API
    data = DataPoint(api_key)

    #Call get_location_id with 'Southampton' and print resulting ID
    location = data.get_location_id('Southampton')
    print(f'Location ID for Southampton: {location}')

    #Sample 30 location IDs
    location_ids = ['310012', '310009', '310004', '310042', '310046', '310013', '351905', '310069', '310016',
                    '14', '26', '33', '3006', '3068', '3075', '3081', '3002', '3005', '310024', '310022',
                    '310025', '310047', '3908', '310035', '310026', '310048', '310031', '310037', '310011',
                    '310006']

    #Call get_forecast_data with IDs. Print result then save to csv
    forecast_data = data.get_forecast_data(location_ids)
    print(forecast_data)
    data.save_to_csv(forecast_data, file_name='Raw_Data.csv', folder_path='/Users/seanwhite/OneDrive - University of Greenwich/Documents/Advanced Programming
    #Call fix_columns with result of get_forecast_data. Print result then save to csv
    processed_data = data.fix_columns(forecast_data)
    print(processed_data)
    data.save_to_csv(processed_data, file_name='Processed_Data.csv', folder_path='/Users/seanwhite/OneDrive - University of Greenwich/Documents/Advanced Prog
```

```python
    #Instantiate WeatherClassifier with API and location_ids list
    weather = WeatherClassifier(api_key, location_ids)
    #Produce classification report for RandomForestClassifier
    weather.train_and_print_report(RandomForestClassifier, name: 'Random Forest Classifier')
    #Produce classification report for GradientBoostingClassifier
    weather.train_and_print_report(GradientBoostingClassifier, name: 'Gradient Boosting Classifier')


#File won't run if being imported by another
if __name__ == '__main__':
    main()
else:
    print('main.py file is not being used correctly - should only be run directly not imported elsewhere')
```

Task 3 – Training a Weather Classifier

Created a new class for this task as it has new functionalities. Once completed this class is imported and called in ‘main.py’. The processed data is retrieved by importing and calling functions from DataPoint which is then formatted for training/testing.

1. Features are assigned using ‘.drop’ to remove ‘Weather Type’ from the given df, similarly to assign the target I indexed ‘Weather Type’.
2. Encoded the features columns using ’LabelEncoder()’ from ‘sklearn.preprocessing’ to enhance performance of models then used ‘train_test_split’ on ‘X’ and ‘y’ as required with random state set to 0 to help mitigate the effect of having live data input on the models every time they’re run.

```python
def format_data(self):
    """
    Formats the data for use on classification models. Uses LabelEncoder and
    train_test_split to produce 'X' and 'y' for 'train' and 'test' data to be used in the following method.
    """
    try:
        if 'Weather Type' not in self.df:
            raise ValueError('Dataframe does not contain a Weather Type column')
        else:
            X = self.df.drop('Weather Type', axis=1)
            y = self.df['Weather Type']

        if len(X.select_dtypes(include=['object']).columns) == 0:
            raise ValueError('Dataframe does not contain non-numeric columns')
        else:
            label_encoder = LabelEncoder()
            for column in X.select_dtypes(include=['object']).columns:
                X[column] = label_encoder.fit_transform(X[column])
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split( *arrays: X, y, test_size=0.25, random_state=0)
        return self.X_train, self.X_test, self.y_train, self.y_test
    except Exception as e:
        print(f'Error occurred while formatting data: {e}')
        print('Try checking the API key is valid')
```

3. Chose ‘GradientBoostingClassifier’ as the second model – also from ‘sklearn.ensemble’. I initially tried SVC, however, the GBC proved far more accurate. Using the inherited split data, the model was built with ‘model.fit’ and metrics produced with ‘model.predict’ and ‘classification_report’ from ‘sklearn.metrics’. Also used pandas ‘to_excel’ to save the classification table in excel format, allowing the layout to be made more presentable. Missing values are handled with ‘zero_division=np.nan’ using numpy to avoid mistaking them for values of 0.

```python
def train_and_print_report(self, classifier, name):
    """
    Call this function with a valid classification model and the name of the classifier used. The function then builds and
    runs the model on the data, returning accuracy metrics for the models performance. It also returns an Excel copy
    of the performance metrics table in the "reports" folder of this repository.
    """
    try:
        model = classifier(random_state=0)
        model.fit(self.X_train, self.y_train)

        predictions = model.predict(self.X_test)

        print(f'{name} Report:')
        print(classification_report(self.y_test, predictions, zero_division=np.nan))
        df_report = pd.DataFrame(classification_report(self.y_test, predictions, output_dict=True))
        file_path = os.path.join('/Users/seanwhite/OneDrive - University of Greenwich/Documents/Advanced '
                                 'Programming/Coursework/CourseworkReport&Code/reports/figures/',
                                 f'{name}_report.xlsx')
        df_report.to_excel(file_path, index=False)
        return df_report
    except Exception as e:
        print(f'Error running classifier: {e}')
        print(f'Check "{name}" is a valid classification model')
```

Documentation

The function ‘main()’ in ‘main.py’ provides the location name/IDs and classifier models to call the classes and methods of the imported files. For this file I didn’t use docstrings as this is just a standalone function so is not callable with the 'help()' function - instead it is explained with comments. Below are the imported classes and their functions.

DataPoint

Found in the 'data_upload.py' file, the data accessed is used to prepare and save forecasts in appropriate formats. Specifically, it will: find a location ID given a name input, create a data frame of live weather forecast given any number of location IDs, save the created data frame as a csv file and clean this data to make it more readable.

- __init__
    - Initializes the class establishing the API and retrieving the URL elements to be used in subsequent methods.
- get_location_id
    - Given a location name, return the location ID from met office sitelist.
- get_forecast_data
    - Given a list of location IDs, creates a data frame consisting of the live feed from each location's three-hourly five-day forecast from the met office.
- save_to_csv
    - Given a pandas data frame, a file name and path to a folder, saves the data in csv format with the file name and in the folder specified.
- fix_columns
    - Given a pandas data frame, this method restyles it to make it interpretable and presentable based on the format of the met office raw data.

WeatherClassifier

Found in the 'weather_classifier.py' file, this class runs classification models for the forecast data and produces a table of metrics on the performance of each model. It's initiated with an API which is used to access pre-prepared data from the DataPoint class. It also requires the IDs of the location on which to build the classifier models.

- __init__
    - This class establishes the data to be used on the classifier by accessing the DataPoint class and establishes the other features that will be used throughout the class.
- format_data
    - No input needed for this method as it inherits the data from the class which it encodes and splits to a format suitable for applying classification models.
- train_and_print_report

- Takes any valid classifier model and the name of the model. The data is trained on the given model and a classification report is produced based on performance metrics which is saved to an excel file under the given name.

Testing

The test.py file used primarily pytest assertions to test the functional methods from the DataPoint and WeatherClassifier classes. Each of the six methods are tested to show them functioning as normal and to demonstrate the error handling embedded in the code by giving false input. All tests passed as shown in the test plan table below which contains a description of each. A more detailed description can be found under the table which is broken down into the approach for each function.

Test	Test Name	Test Description	Pass/Fail
1	test_get_location_id	Test legitimate location name returns expected ID	Pass
2			Pass
3			Pass
4	test_false_id	Test invalid location name returns appropriate error	Pass
5			Pass
6			Pass
7	test_get_forecast_data	Check valid location IDs return a filled pandas df with expected columns	Pass
8			Pass
9			Pass
10	test_false_forecast_data	Check with invalid location id that an empty dataframe is returned	Pass
11	test_save_to_csv	Test file path exists and csv translates as readable df	Pass
12			Pass
13	test_save_to_csv_error	Test error raised if input contains an invalid folder path	Pass
14	test_fix_columns	Test output has expected column names	Pass
15	test_fix_columns_2	Test error raised if no 'W' column to map 'Weather Type'	Pass
16	test_fix_columns_3	Test function runs without error if no '$' column	Pass
17	test_format_data	Test all returned features have been filled	Pass
18	test_format_missing_weather_type	Test error raised when no 'Weather Type' column in df	Pass
19	test_format_numeric_columns	Test error raised when no non-numeric columns in df	Pass
20	test_train_and_print_report	Test classification model (Logistic Regression) produces a filled df	Pass
21		Test classification model (SGD Classifier) produces a filled df	Pass
22	test_false_train_report	Test non-classification model (Linear Regression) doesn't work	Pass
23		Test non-classification model (SGD Regression) doesn't work	Pass

- get_location_id
    - There are two functions – one which uses ‘pytest.mark.parametrize’ to feed valid location names and assert the expected ID, and the other that uses the same approach to assert the function will flag it as an invalid name.
- get_forecast_data

    - This also uses parametrize in the first function which inputs valid location IDs to assert the output will be a pd.DataFrame with 'isinstance', a list of the 'output.values' will be 'len' > 0 and assert what a list of the '.columns' will be.
    - The second gives a false location name and asserts the values will be equal to 0 and 'output is '.empty.'

- save_to_csv
    - The first parametrize to input sample data and assert the given file path '.is_file()' and that the created csv, when read by pandas is readable with 'data.equals().'
    - The second passes an invalid folder path and uses 'capsys.readouterr()' to assert the error which prompts to check the file name and folder.

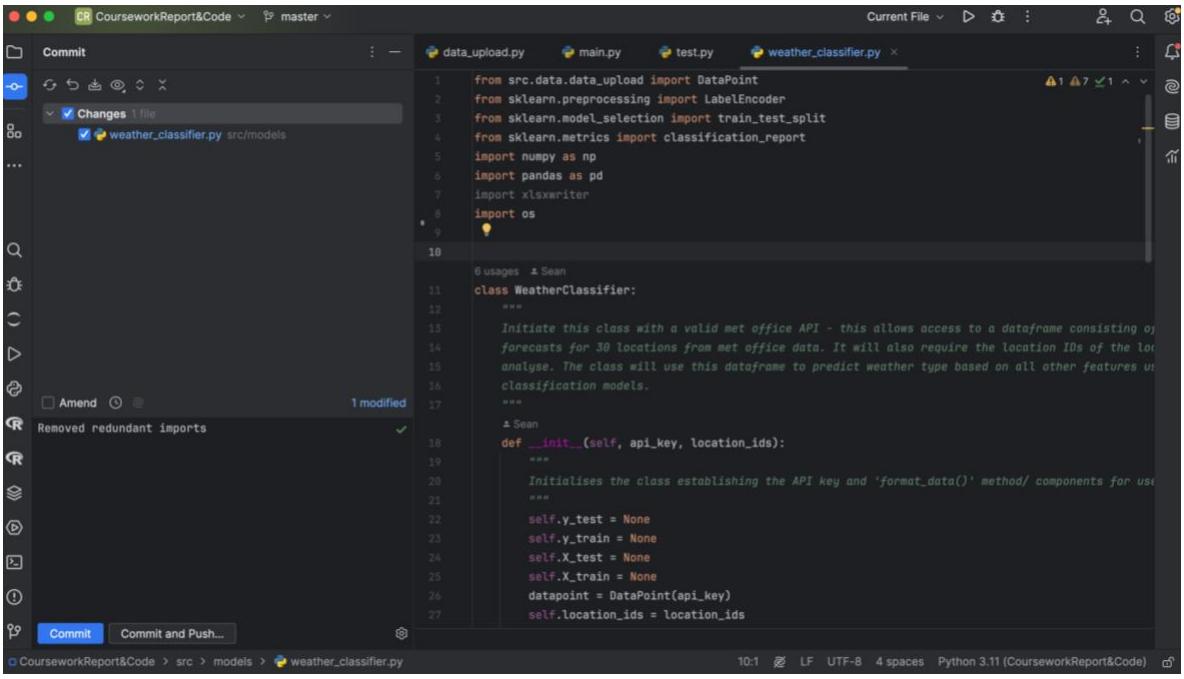
- fix_columns
    - For these tests 'sample_forecast_data' with 'pytest.fixture' is created which mimics the columns names of forecast data and the size of the 'weather_mapping' dictionary. The 'W' column in mapped to the 'weather_dict' to make it truer to the real data.
    - The First test asserts the expected columns.
    - The second removes the 'W' column and uses capsys to assert the error message.
    - The third removes the '$' column to demonstrate that the function will run regardless.

- format_data
    - For these tests 'pytest.fixture' calls the WeatherClassifier class with two valid IDs.
    - The first test asserts that the 'X' and 'y', 'test' and 'train' data 'is not None.'
    - The second removes 'Weather Type' and uses 'apsys to assert the error.
    - The third leaves the df with only numeric columns to assert the error using capsys.

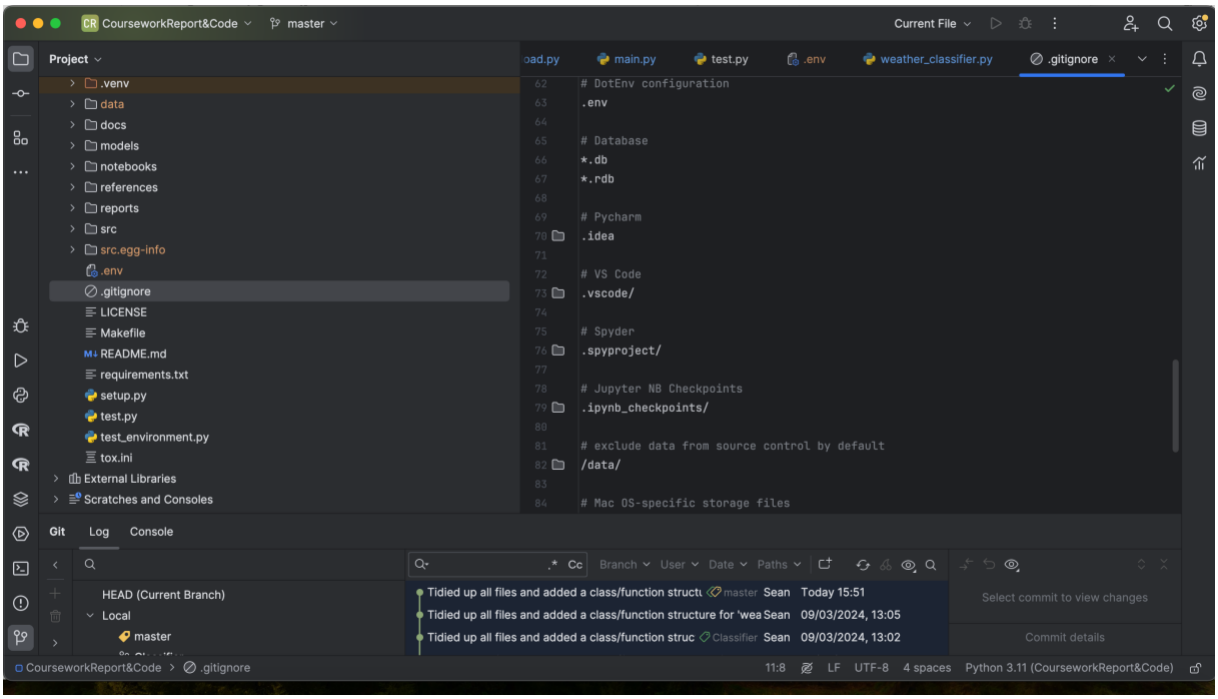
- train_and_print_report
    - Uses parametrize to input two classifier models for the first function and two regression models for the second to assert that the first produces a 'pd.DataFrame' with values > 0, and the second produces the expected outcome of 'None' and not a df. The df is the classification report.

Version Control

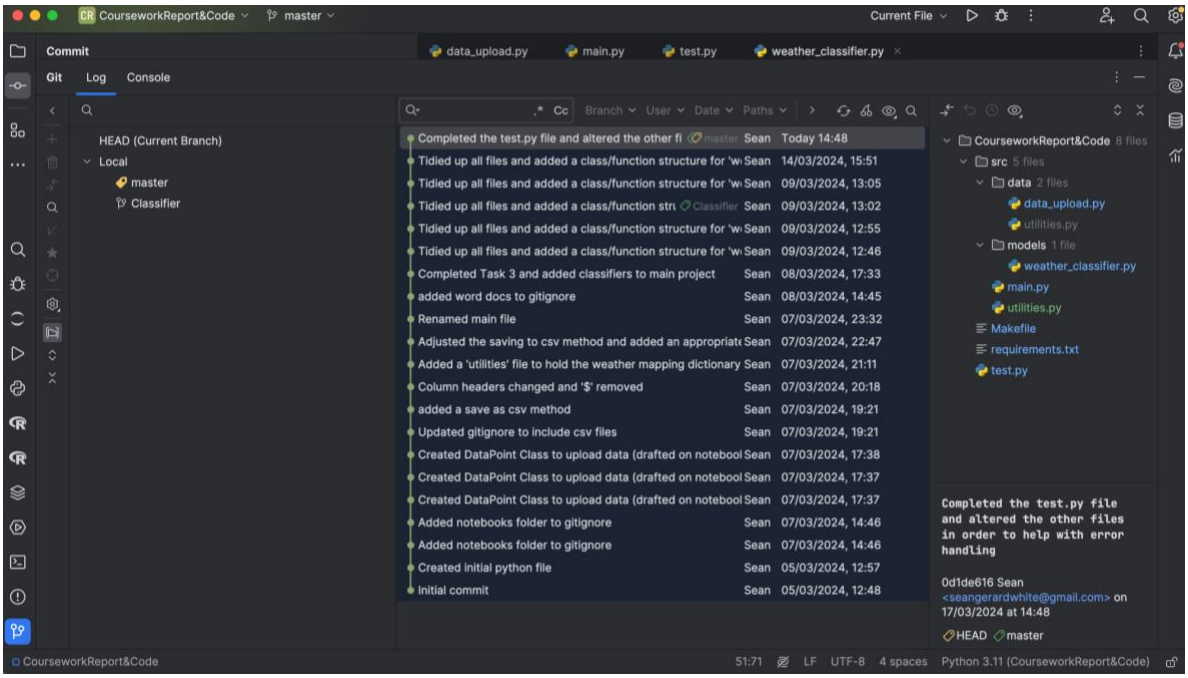
For this project I initiated the git repository and operated it on Pycharm as it felt simpler in this scenario where the whole project is being created using Pycharm rather than using gitbash.



Using Pycharm for git automates much of the git management process. Once the repository is set up, if a new file is created there is an automatic prompt to start tracking the file. Whenever sufficient changes are made there is a commit menu which shows all tracked file that have been changed since the last commit. These can be committed with a comment to describe the change as demonstrated above and each of these are archived in a log which provides a record of and access to all previous versions of the project.



Using cookiecutter meant the '.gitignore' file was added with the automated structuring of the folder. The '.env' file that is shown in the .gitignore above is where the API key as well as URL was kept. The '/data/' folder is also included. These have been ignored as they're not being altered - only referenced. Common file types are automatically added in the file by cookiecutter however, I did have to add some manually such as '*.docx' and '*.xlsx' to suit my specific case.



As seen above I had a branch called 'classifier' as well as 'master'. This was created when I started constructing the 'WeatherClassifier' class for task 3 as I was happy with the 'DataPoint'

class which this new one would be inheriting from and wanted to work on it independently. Once I was happy with the implementation of task 3, I merged the 'classifier' branch back into 'master' and imported it in 'main.py'.

Analysis

Both models yielded good results (generally ranging from 80s to low 90s in percentage accuracy) with the GBC usually outperforming the RFC. The exact metrics of performance changed as live forecasts are given as input data, however, the model's results stay consistent relative to each other. The below tables are a good representation of average output.

Random Forest Classification

	Precision	Recall	F1-Score	Support
Clear	0.4	0.25	0.307692	8
Cloudy	0.895397	0.995349	0.942731	215
Heavy rain	1	0.5	0.666667	4
Light rain	0.777778	0.518519	0.622222	27
Low visibility	1	1	1	1
Partly cloudy	0.769231	0.434783	0.555556	23
accuracy			0.874101	278
macro avg	0.807068	0.616442	0.682478	278
weighted avg	0.861161	0.874101	0.85753	278

Gradient Boosting Classification

	Precision	Recall	F1-Score	Support
Clear	0.5	0.375	0.428571	8
Cloudy	0.941964	0.981395	0.961276	215
Heavy rain	1	0.5	0.666667	4
Light rain	0.821429	0.851852	0.836364	27
Low visibility	1	1	1	1
Partly cloudy	0.705882	0.521739	0.6	23
accuracy			0.906475	278
macro avg	0.828213	0.704998	0.748813	278
weighted avg	0.899051	0.906475	0.899825	278

The biggest weakness in both models is their reliance on unstable data mainly in the way of having incomplete target variables from the 'Weather Type' column. Depending on the day certain types don't show up and there are unbalanced numbers of those that are included. As can be seen in the tables above, there are only six variables listed out of a possible twelve, of which some have such little frequency that they have no effect on the metrics. This has also resulted in a variance of about 10% in the accuracy when produced at different times. Altering

the ‘test size’ from 0.25 to perhaps 0.50 may help to mitigate this given the relatively small size of the population size provided.

Despite fluctuations, both models proved very effective as demonstrated the in the tables where they have accuracy scores of 87% and 91% with the Gradient Boosting Classifier scoring better on almost all metrics.

Self Assessment

The part of the assignment I found most challenging was the testing. I found it quite difficult to conceptualize the best ways to assess the quality of the individual functions where most of them are reliant on their class being initialized to run. Whilst I was able to produce effective tests for all major functions, I think the execution could have been better and the testing more thorough. I did find this part of the assignment useful, however in highlighting problem areas and helping make my code more robust. The code all runs effectively – delivering on all the given tasks and in some cases adding functionality beyond the specification. I think the report provides overview of the project, however there are some areas where I feel I’ve repeated myself when describing the classes.