University of St Andrews

School of Computer Science
KY16 9SX

CS5052

SparkDataApp

A console-based PySpark data analysis tool

Student ID: [redacted]

March 19, 2025
Total Words: 1490 *(Source: Overleaf)*

1 Design and End-To-End Solution

For building the PySpark[1] console application, the choice was made to implement a main menu, showing all available options such as *query*, *save* or *exit*. For better formatting and output design, the *rich*[2] library was used. At each start-up, a *SparkSession* is created and will be used to perform data operations throughout the active console session. The *SparkSession* is explicitly stopped when closing the application through the main menu option *exit*. The decision was made that there should be a specified folder for the raw data and the processed data, to distinguish between the files that have been saved after doing operations on it. Starting from this point, the start workflow is as follows:

1. The user starts the app, the data directories are scanned, and all files found are displayed.
2. After choosing a file to load, initial analysis is done to check for missing values.
3. The user is shown a summary of missing values and asked if those should be replaced by default values, which is done if selected.
4. The main menu pops up and shows all possible options.

The reason for allowing the option to impute missing values is that queries and visualisations will ignore missing values by checking *.isNull()* on the data. By selecting to impute values with constants at the start, the data will now be treated as a normal entry and will show up in the visualisations and summaries.

The menu options shown are shown below, where *filter data* contains additional features to extract smaller subsets (certain years, regions, etc.) from the data set. Tab.1 shows the implemented features and where they can be accessed in the code. Furthermore, after doing any of the options that modify the original dataframe, after performing the query, the user is offered to save the query result as the current dataframe, overwriting the original one. This allows multiple filter options on a dataset to be performed and then saved from the main menu to the local folders.

[1] https://spark.apache.org/docs/latest/api/python/index.html
[2] https://github.com/Textualize/rich

```
Available commands:
l - Load data
q - Query data
v - Visualize data
f - Filter data
s - Save data
h - Help
x - Quit
```

To show the query data in a well-formatted fashion, I created a new formatted table using the *Table* class in the *rich* library. If the data are too complex for display in a table, I use matplotlib[3] to save a plot in the *plots/* directory. The terminal will display the name of the file and where it was saved to make the usage more intuitive.

	Feature	Implementation (report section)
Part 1	Read dataset	`load_data()` (2.1)
	Store dataset	`save_data()` (2.1)
	Search by LA	`handle_local_authority_query()` (2.2)
	Search by school type	`handle_school_type_query()` (2.2)
	Unauthorised absences	`handle_unauthorized_absences_query()` (2.2)
Part 2	Compare local authorities	`handle_local_authority_query()` (3.1)
	Regional performance	`analyse_regional_attendance()` (3.2)
Part 3	Explore links	`analyse_absence_patterns()` (4)

Table 1: Table containing the features implemented in the PySpark application, with corresponding methods from the codebase.

1.1 Challenges and Solutions

One issue was to implement an application that can handle dynamic data. To achieve this, all queries and visualisations fetch the data from the respective columns and display options with respect to the fetched data. This means that we can provide a dataset with a specific time period, which can be created through the *filter* menu option and saved locally.

Furthermore, it may seem convenient to use the **_percent*-columns for calculation of averages, but this would give each row equal weight regardless of the number of total sessions or enrolments. It was chosen to use the calculation specified in the data guidance file that uses the absolute session numbers to calculate the averages.

The absence data set contains hierarchical data at four geographic levels: National, Regional, Local Authority, and School. Each higher geographic level (National, Regional, LA) includes both individual school type breakdowns (Primary, Secondary, Special) and a *Total* row. When analysing these data, care must be taken to avoid double counting by either excluding *Total* rows and using only school-type breakdowns, or using only *Total* rows for aggregate figures. Furthermore, the appropriate geographical level needs to be selected as well, since each represents complete data.

The base figures are consistent across all levels when properly counted.

- Total Schools: 21,214

[3] https://matplotlib.org/

- Total Enrolments: 7,105,958
- School Types: Primary (16,780), Secondary (3,406), Special (1,028)

To filter the dataset for the different queries, the following code structure is used:

```
base_df = df.filter(
    (F.col("geographic_level") == "<LEVEL>") &
    (F.col("school_type") == "<SCHOOL TYPE>")
)
```

2 Part 1

2.1 Load / Save

To implement the read functionality, a default data location *data/*is specified. Whenever the app is started, the directory is scanned for suitable *csv* files or directories containing files saved by PySpark (discussed in *save data*). If data is found, the user is presented an option of which file to load. The loaded file is analysed after loading for any missing values using this code segment:

```
df = spark.read.csv(str(file_path), header=True, inferSchema=True)

null_counts = {
    col: int(df.filter(F.col(col).isNull()).count())
    for col in df.columns
}
```

The user can then choose to fill in the missing values or ignore the warning. If the warning is ignored, missing values will be excluded from calculations, and thus the user may miss valuable information but work only with available data.

The user can also choose to save the current dataframe loaded in the PySpark session to the local data folder, where *csv, parquet, json* can be chosen. *CSV* is the default type, and after having chosen it, the files are saved as partitioned using PySparks methods, as seen in the following code snipped.

```
df.write.format(file_format) \
    .option("header", "true") \
    .option("quote", "\"") \
    .option("escape", "\"") \
    .mode(mode) \
    .save(str(save_path))
```

In a distributed system, we would have an advantage of cutting the file into multiple partitions, as it could be saved on multiple clusters.

2.2 Query

To query the enrolments of each local authority (LA) by year, the following code snippet does the aggregation of the table. For this query, *geographic_level* is filtered by *Local Authority* and *school_type* as *Total*.

```
pivot_df = df.groupBy("la_name").pivot("time_period").agg(
    F.sum("enrolments")
).orderBy("la_name")
```

Additional summary statistics are shown with the average number of enrolments per selected LA to allow a comparison of the number for each LA in each year.

For querying the school type, the user is asked to select a specific type and a time period, and if the user chooses to see the breakdown of specific types of authorised absences in a specific time period, then the dataset will be filtered for a single year and only the selected time period will be shown. For this query, the geographical level of *School* is used, since it directly contains the school-level data that we want to query. The code used to query the values is as follows:

```
for year in years:
    year_data = df.filter(F.col("time_period") == year)

    stats = year_data.agg(
        F.sum("sess_authorised").alias("total_absences"),
        F.sum("enrolments").alias("total_enrolments")
    ).collect()[0]
```

For the last query, retrieving all unauthorised absences in a certain year, broken down by region name or LA name, the data are filtered for the geographic level of either *Region* or *Local Authority* and enrolments, total unauthorised absences, and absence sessions per student are shown.

3 Part 2

3.1 Compare LAs

To compare two LAs, the set is filtered for local authority and the *Total* school type, and an aggregation of key metrics (enrolments and absence counts) as well as a calculation of absolute and percentage differences is performed. The functionality to perform this comparison is implemented as follows:

```
metrics = comparison_data.groupBy("la_name").agg(
    F.sum("enrolments").alias("total_enrolments"),
    F.sum("sess_authorised").alias("total_authorised"),
    F.sum("sess_unauthorised").alias("total_unauthorised"),
    F.sum("sess_possible").alias("total_possible")
).collect()
```

3.2 Regional Performance

The next part is an analysis of the regional performance in the UK. First, the data is filtered by geographic level *Regional*, and the *Total* school type. It is then aggregated to calculate the average attendance per region and year using absolute numbers to correctly account for different region sizes.

```
attendance_expr = 100 * (1 - F.sum("sess_overall") / F.sum("sess_possible"))
absence_expr = 100 * F.sum("sess_overall") / F.sum("sess_possible")

region_year_stats = (
    df_england.groupBy("region_name", "time_period")
    .agg(
        attendance_expr.alias("avg_attendance"),
        absence_expr.alias("avg_absence")
    )
    .orderBy("region_name", "time_period")
)
```

The results showed that all regions have improved attendance rates on average per year as seen in Fig.1, but have begun to decline after peaking in 2013/14.

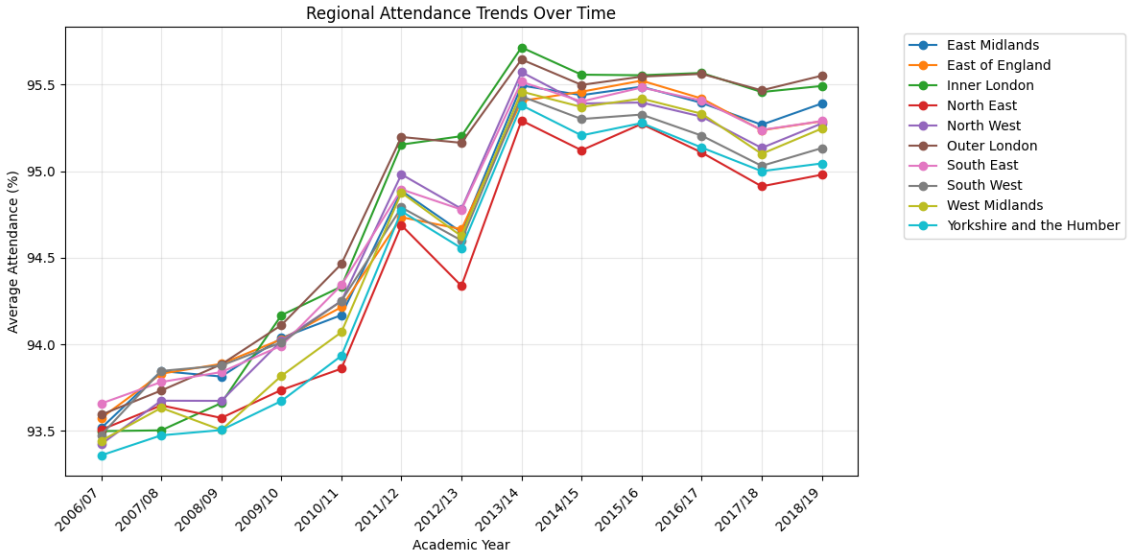

Figure 1: Regional attendance rates over all recorded years.

Overall, no region has worsened on average over the inspected time period, but there are some differences when looking at each region in detail. Tab.2 shows that *Outer and Inner London* have the best improvements in general, but the region with the highest highest average in attendance is *Outer London* with an average attendance of 94.88%. The worst region is *North East* with a 94.46% attendance rate. Fig.4 shows the absence rates that support these findings.

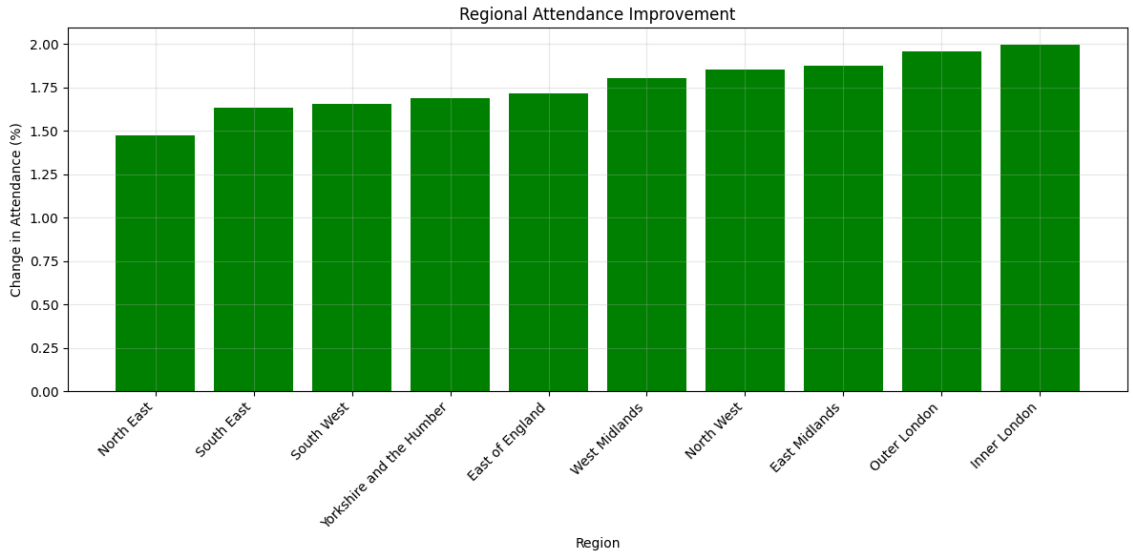

Figure 2: Comparison of overall attendance changes.

4 Part 3

This part focusses on exploring the data for a relation between school type, pupil absences, and the location of the school. To perform this analysis, the geographical location *Regional* is chosen, as it offers a good broad structure of the country. This time, the data is filtered to include the school types *Primary, Secondary and Special*, instead of *Total*. The average absence rates by school type, region, and year are calculated using the following PySpark methods, where the total values are also saved to allow weighted calculations later on.

```
result = base_df.groupBy("school_type", "region_name", "time_period").agg(
    (100.0 * F.sum("sess_overall") / F.sum("sess_possible"))\
        .alias("avg_absence_rate"),
    F.sum("sess_overall").alias("total_sessions"),
    F.sum("sess_possible").alias("total_possible_sessions")
).orderBy("time_period", "school_type", "region_name")
```

Fig.3 shows the resulting data. It is evident that the highest absence rate belongs to the school type *special* with an average of 10.6%. State-funded primary schools are the best in attendance, with only 4.6% absence rate. Notably, *Special Schools* in *Inner London* have the highest absence rate of all combinations, followed by the sane schooll type in the *West Midlands*.

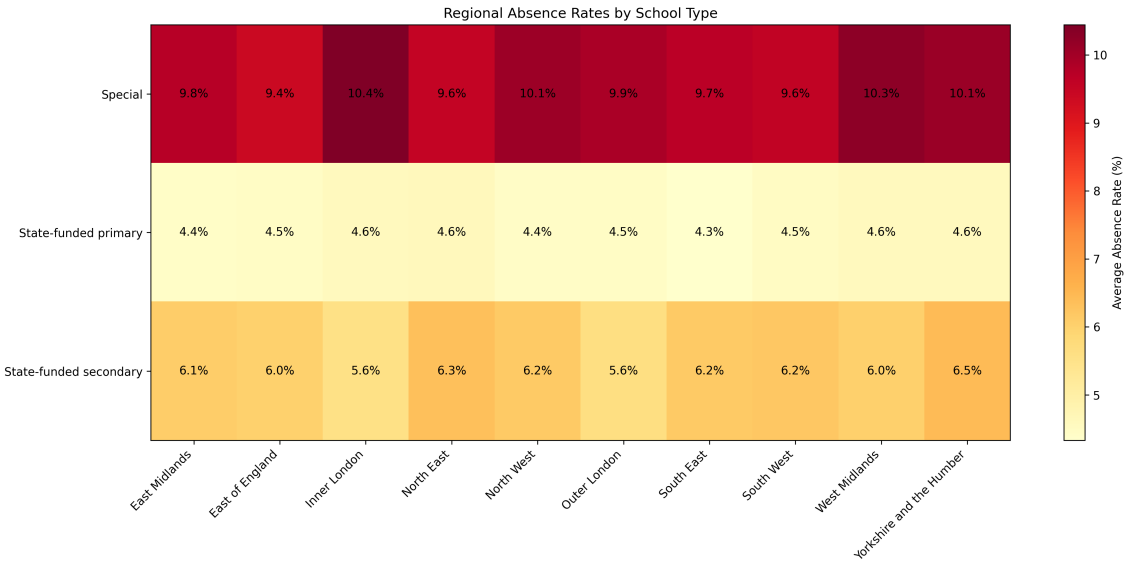

Figure 3: Comparison of region, school type and absence rate.

Fig.4 shows the total averages of each region, with the standard deviation shown in the diagram. This shows that *Inner London* has one of the lowest absence rates despite the highest ranking of special schools. The total school numbers in Sec.1.1 show that special schools only amount to 5% of all schools, hence not much influencing the calculated average.

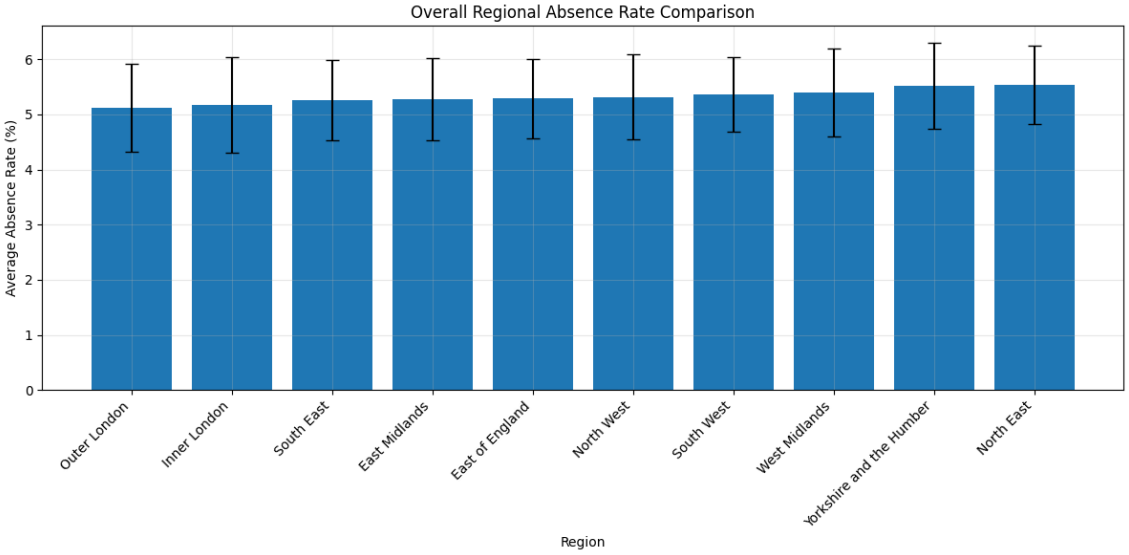

Figure 4: Comparison of regional absences rates over the recorded time periods.

Reflective Summary

The development of this PySpark console application has provided insights into both the technical implementation of building a console app as well as data analysis methodologies. Furthermore, I have had a good experience and learning curve when getting to know PySpark. Being someone who used Pandas in the past, PySpark offered a good entry and comparable functions to allow an easy transition between the technologies.

A key learning outcome was the importance of exploratory data analysis to get to know a data set before working with it. Another important aspect was to see how Apache Spark is used in practice through API interfaces, in this case, PySpark. I also gained insights into manipulating large datasets using the methods provided by PySpark, as well as handling save and load operations. Furthermore, I also experienced how to use data analysis to find patterns and trends in datasets and to create visually appealing ways of showing the results to a potential user. This highlights the importance of selecting the right data, defining query operations, and choosing the right way of showing the aggregated data. Building a console application that might be used by a third party also highlighted the importance of appropriate input validation, error handling, and application design. It needed some out-of-the-box thinking to build a robust application that continues running while displaying helpful error messages to a potential user. Furthermore, The project showed that writing dependable code, with meaningful variable names and a clear structure is essential to keep track of all features when the program grows.

In summary, this practical has allowed me to get first-hand experience on working with Apache Spark, exploring its advantages and features, as well as creating a usable console application for data analysis.