Program to analyse weather data for different regions in Europe

Author: Jacques Thurling

Date: 2024/12/23

Table of Contents

1. Abstract
2. Task 1 - Data Table implementation
3. Task 2 - Candlestick Chart implementation
4. Task 3 - Graph Chart Implementation
5. Task 4 - Statistical Prediction implementation
6. Conclusion
7. References

Abstract

Used the initial program given to us for the Midterms and used that as a baseline to implement the task 1-4, followed the same patterns as the rest of the program. For task 1 created a basic data table in the terminal showing the opening, closing, highest and lowest temps for a region on a yearly timescale. Task 2, a candlestick chart was implemented, using the same temp information of the previous task, this is done by mapping the weather entry data to the candlestick data and display the monthly temperature as a candlestick. Task 3, shows the filtered data from a region, a beginning year and a ending year date and displays the points as a normal graph. Task 4, uses the SARIMA time-series model to predict the next 5

years of average temperature and displays them on the same chart that's done in Task 3.

Task 1 - Data Table Implementation

Task 1 focuses on the Data Table implementation, for the implementation, we use the input and tokeniser from the initial program given.

```
std::cout << "Enter the region (FR): " << std::endl;
std::string input;
std::getline(std::cin, input);

// Split input into tokens (though only one token expected)
std::vector<std::string> tokens = CSVReader::tokenise(input,
```

From this we can map the input from the user to a `WeatherEntryType` , which returns an `ENUM` value. After this we print out the data table headers.

We iterate over the years for the given region, printing the timestamp, opening, closing, highest and lowest temperature while iterating, until we hit a year with no temperature inputs and break out of the loop.

If there is an error during any of the steps during the implementation, we catch the error and log the error message to the console.

```
*/
void MerkelMain::printWeatherStats() {
  // Prompt user for region input
  std::cout << "Enter the region (FR): " << std::endl;
  std::string input;
  std::getline(std::cin, input);

  // Split input into tokens (though only one token expected)
  std::vector<std::string> tokens = CSVReader::tokenise(input, ',');

  // Container for weather entries
  std::vector<WeatherEntry> temp;

  try {
    // Convert input string to region enum type
    WeatherEntryType region = WeatherEntry::mapFromInputToRegion(tokens[0]);

    // Start from 1980 as base year
    int year = 1980;

    // Print header for data table
    std::cout << "Date                     Open       High      Low      Closing"
              << std::endl;
    // Set decimal precision for temperature values
    std::cout << std::fixed << std::setprecision(3);

    // Process data year by year until no more data is available
    do {
      // Get weather entries for current year and region
      temp = std::get<std::vector<WeatherEntry>>(
          weather.getWeatherEntries(region, std::to_string(year)));

      // Exit loop if no data found for current year
      if (temp.size() == 0) {
        break;
      }

      // Calculate temperature statistics for the year
      double lowestTemp = Weather::getLowestTemp(temp);
      double highestTemp = Weather::getHighestTemp(temp);
      double closingTemp = Weather::getClosingTemp(temp);
      double openingTemp = Weather::getOpeningTemp(temp);

      // Print statistics for current year
      std::cout << temp.begin()->timeframe << "    " << openingTemp << "      "
                << highestTemp << "    " << lowestTemp << "    " << closingTemp
                << std::endl;

      // Move to next year
      year++;
    } while (temp.size() > 0);

  } catch (const std::exception &e) {
    // Handle any errors during data processing
    std::cout << "MerkelMain::printWeatherStats error when mapping and "
                 "retrieving entries"
              << std::endl;
  }
}
```

Figure 1 - MerkelMain implementation for the data table.

Figure 1 shows the entire implementation for the data table. Once the entire program has run the output is shown in ***Figure 2***

```
1: Print help
2: Print Weather data for region
3: Print Candlestick chart for region and year
4: Print Graph for date range and region
5: Predict future temperatures
6: Continue
==============
Type in 1-6
2
You chose: 2
Enter the region (FR):
at
Date                    Open        High      Low       Closing
1980-01-01T00:00:00Z    6.049       29.132    -14.507   6.049
1981-01-01T00:00:00Z    7.192       29.419    -16.219   7.192
1982-01-01T00:00:00Z    7.567       28.395    -15.084   7.567
1983-01-01T00:00:00Z    8.054       32.416    -18.098   8.054
1984-01-01T00:00:00Z    6.836       32.659    -13.338   6.836
1985-01-01T00:00:00Z    6.577       29.166    -22.705   6.577
1986-01-01T00:00:00Z    7.121       29.785    -20.601   7.121
1987-01-01T00:00:00Z    6.600       28.054    -25.301   6.600
1988-01-01T00:00:00Z    7.689       31.313    -13.019   7.689
1989-01-01T00:00:00Z    8.067       28.968    -10.969   8.067
1990-01-01T00:00:00Z    8.067       29.145    -11.347   8.067
1991-01-01T00:00:00Z    6.926       30.448    -15.978   6.926
1992-01-01T00:00:00Z    8.043       32.326    -13.679   8.043
1993-01-01T00:00:00Z    7.390       30.092    -17.096   7.390
1994-01-01T00:00:00Z    8.698       31.458    -12.935   8.698
1995-01-01T00:00:00Z    7.331       31.620    -14.468   7.331
1996-01-01T00:00:00Z    5.976       27.225    -20.283   5.976
1997-01-01T00:00:00Z    7.322       27.528    -12.095   7.322
1998-01-01T00:00:00Z    7.847       31.945    -15.665   7.847
1999-01-01T00:00:00Z    7.760       29.667    -16.172   7.760
2000-01-01T00:00:00Z    8.764       32.539    -16.425   8.764
2001-01-01T00:00:00Z    7.675       30.489    -17.771   7.675
2002-01-01T00:00:00Z    8.421       30.305    -16.596   8.421
2003-01-01T00:00:00Z    8.114       33.745    -17.103   8.114
2004-01-01T00:00:00Z    7.290       28.997    -15.015   7.290
2005-01-01T00:00:00Z    6.979       31.314    -19.122   6.979
```

Figure 2 - Final output for Task 1 inputs

Task 2 - Candlestick Chart implementation

For the candlestick implementation we use the same input implementation from Task 1, we get the user input in the format of `at,1990` , which would correlate to the `REGION,YEAR` , which gets passed into the tokeniser, we use the tokens to return a vector of vectors of `WeatherEntryType` , each sub-vector is the monthly weather temperature entries, which gets their own opening, closing, highest and lowest for the candlestick.

```cpp
void MerkelMain::printCandlesticksChart() {
  // Prompt user for input in format: region,year
  std::cout << "Enter the region and year (FR,1990): " << std::endl;
  std::string input;
  std::getline(std::cin, input);

  // Split input string into tokens using comma as delimiter
  std::vector<std::string> tokens = CSVReader::tokenise(input, ',');

  // Container for monthly temperature entries
  std::vector<std::vector<WeatherEntry>> monthly_entries;

  try {
    // Convert region string to enum type
    WeatherEntryType region = WeatherEntry::mapFromInputToRegion(tokens[0]);

    // Retrieve monthly weather data for specified region and year
    monthly_entries = std::get<std::vector<std::vector<WeatherEntry>>>(
        weather.getWeatherEntries(region, tokens[1],
                                  WeatherFilterOptions::monthly));

  } catch (const std::exception &e) {
    // Handle any errors during data retrieval
    std::cout << "MerkelMain::printWeatherStats error when mapping and "
                 "retrieving entries"
              << std::endl;
    throw e;
  }

  // Container for candlestick data
  std::vector<Candlestick> candlesticks;

  // Process each month's data into candlestick format
  for (int i = 0; i < monthly_entries.size(); i++) {
    // Calculate temperature metrics for the month
    double lowestTemp = Weather::getLowestTemp(monthly_entries[i]);
    double highestTemp = Weather::getHighestTemp(monthly_entries[i]);
    double closingTemp = monthly_entries[i].end()->temp;
    double openingTemp = monthly_entries[i].begin()->temp;

    // Create candlestick object with temperature data
    Candlestick candlestick{openingTemp, closingTemp, highestTemp, lowestTemp};

    // Add to collection
    candlesticks.push_back(candlestick);
  }

  // Display the candlestick chart
  Candlestick::printCandleStickChart(candlesticks);
}
```

Figure 3 - The initial implementation for Candlestick chart

Once the initial mapping is done we send the data though to the `Candlestick` class which is responsible for printing the candlestick chart on an `(x,y)` plane, where it’s broken down into the character spaces for the grid. A example of the grid is shown below:

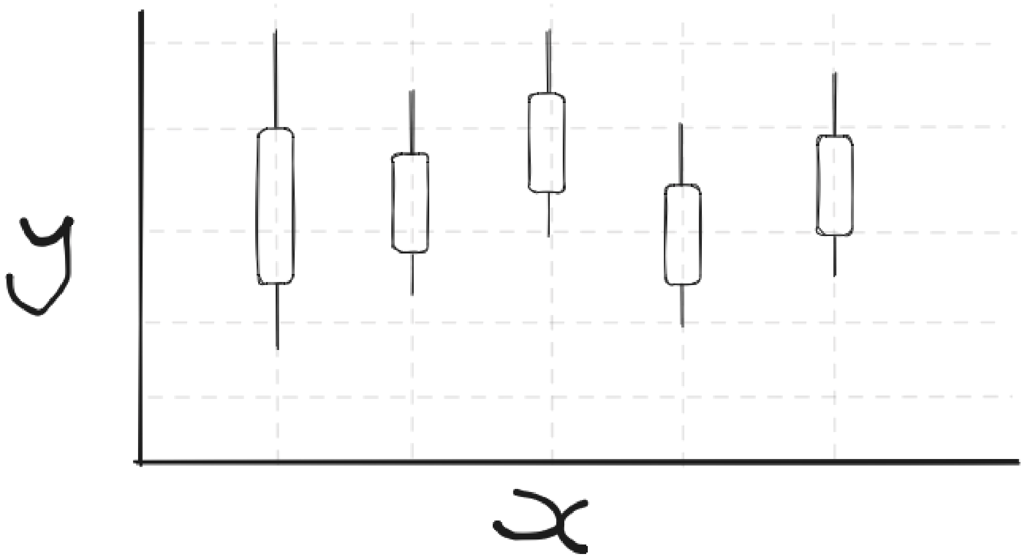

Figure 4 - Example for the candlestick chart from planning, which is going to be implemented for this task

We have a `width` and `height` variables which dictates the size of the grid on which the graph is going to be rendered. We also have an `temp` variables which are the values that's going to be shown on the `y-axis` . We also have a tolerance to make sure the candlestick chart renders relatively correctly.

```
void Candlestick::printCandleStickChart(
    std::vector<Candlestick> &candlesticks) {
  // Chart height in characters
  unsigned int height = 25;
  // Chart width in characters
  unsigned int width = 90;

  // Initial temperature values for y-axis
  int temp = 50;

  std::cout << std::endl;

  // Temperature tolerance for drawing candlesticks
  int tolerance = 3;
```

Figure 5 - Opening of the `printCandleStickChart` function

Once we iterate over the rows, we map the index to the proper temperature via the `mapYCoordsFromIndex` , after that we show each notch `┤` when the `index % 2 == 0` as we iterate over the rows, once we get to the bottom of the graph we start rendering the `x-axis` . As we iterate over the columns, when `j % 10 == 0` we print the candlestick and the `x-axis` notch `┬` .

For the complete code ***Figure 5***, ***Figure 6*** and ***Figure 7*** all cover the `Candlestick::printCandleStickChart()`

```cpp
// Iterate through each row of the chart
for (int i = 0; i < height; i++) {
  // Map row index to temperature values
  temp = mapYCoordFromIndex(i);

  // Iterate though each column
  for (int j = 0; j < width; j++) {
    if (j == 0 && i == height - 1) {
      // Bottom left corner
      std::cout << "     └";
    } else if (j == 0) {
      if ((i % 2) == 0) {
        // Prepare the y-axis
        if (temp < -10) {
          std::cout << temp << " ┤";
        } else if (temp < 10 && temp > -1) {
          std::cout << "  " << temp << " ┤";
        } else if (temp < -1 && temp > -10) {
          std::cout << " " << temp << " ┤";
        } else {
          std::cout << " " << temp << " ┤";
        }
      } else {
        // Y-axis line
        std::cout << "     │";
      }
    } else if (i == height - 1) {
      // Populate the x-axis
      if (j % 10 == 0) {
        std::cout << "┬";
      } else {
        std::cout << "─";
      }
    } else {
      if (j % 10 == 0) {
        Candlestick candlestick = candlesticks[(j / 10) - 1];

        // Draw candlestick components based on temperature relationships
        if (candlestick.highestTemp == candlestick.lowestTemp) {
          // No variation in temperatures
          std::cout << " ";
        } else if (candlestick.openingTemp < candlestick.closingTemp) {
          // Bullish candlestick (closing > opening)
          if (temp < candlestick.highestTemp &&
              temp > candlestick.closingTemp + tolerance) {
            std::cout << "│";
          } else if (temp < candlestick.closingTemp + tolerance &&
                     temp > candlestick.openingTemp - tolerance) {
            if (candlestick.closingTemp - candlestick.openingTemp < 1) {
              std::cout << "┼";
            } else {
              std::cout << "█";
            }
          } else if (temp < candlestick.openingTemp &&
                     temp > candlestick.lowestTemp) {
            std::cout << "│";
          } else {
            std::cout << " ";
          }
        } else if (candlestick.closingTemp < candlestick.openingTemp) {
          // Bearish candlestick (closing < opening)
          if (temp < candlestick.highestTemp &&
              temp > candlestick.openingTemp) {
            std::cout << "│";
          } else if (temp < candlestick.openingTemp &&
                     temp > candlestick.closingTemp) {
            std::cout << "█";
          } else if (temp < candlestick.closingTemp &&
                     temp > candlestick.lowestTemp) {
            std::cout << "│";
          } else {
            std::cout << " ";
          }
        } else {
          std::cout << " ";
        }

      } else {
        std::cout << " ";
      }
    }
  }
  std::cout << std::endl;
}
}
```

Figure 6 - Rest of the `printCandleStickChart` function

```
// Maps a y-coordinate index to a temperature value
// This creates the temperature scale on the y-axis
double Candlestick::mapYCoordFromIndex(int index) {
  return 40 - ((80 * index) / 25);
}
```

Figure 7 - Map the index in the loop to the Y-Coords for display

For the complete output of the candlestick chart this can be shown on the ***Figure 8***

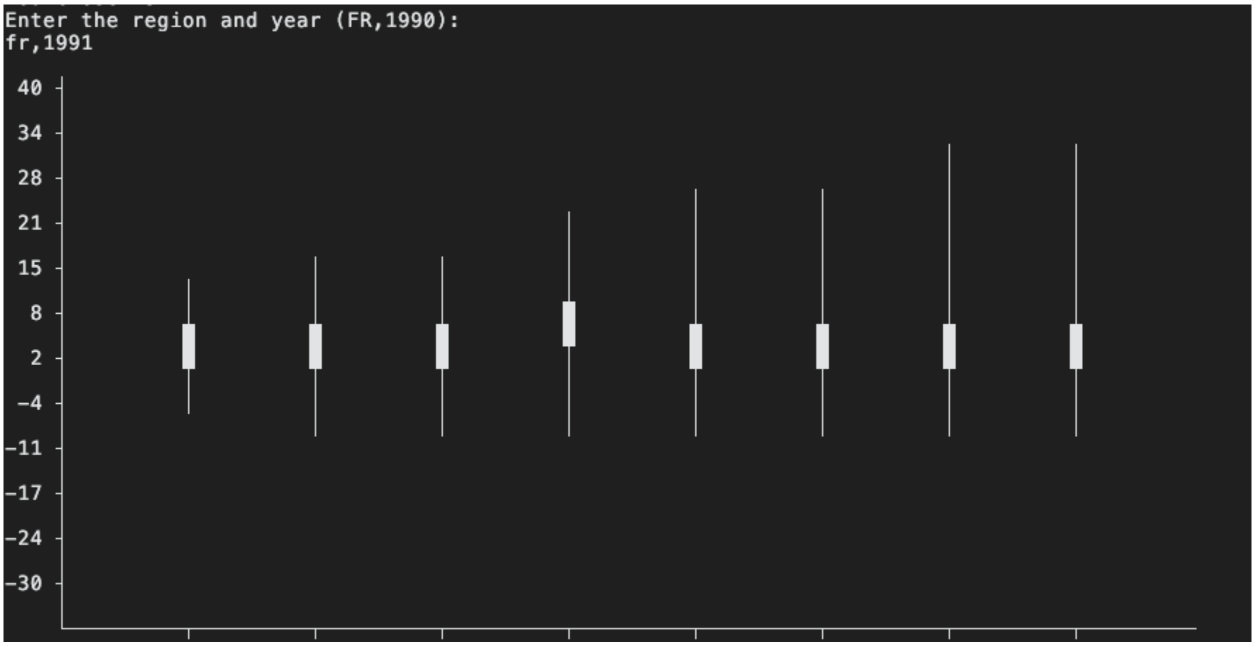

Figure 8 - Implemented Candlestick Chart

Task 3 - Graph chart implementation

For the graph chart implementation, we use the same input implementation from the previous tasks. From there we test to see if the input was a valid date range, by checking if the second year minus first year i.e. `(endingYear - beginningYear) > 0` is not negative. Then we iterate over the year difference and get the weather entries for each year, once we get the yearly entries, we pass it over to the static `printGraph` function in the `ChartRenderer` class.

```cpp
/**
 * Displays temperature data as a graph for a specified date range
 * Format: region,start_year,end_year (e.g. FR,1990,2001)
 */
void MerkelMain::printFilteredChart() {
  // Prompt user for input
  std::cout << "Enter the region, start year and end year e.g. FR,1990,2001: "
            << std::endl;
  std::string input;
  std::getline(std::cin, input);

  // Split input string into tokens using comma as delimiter
  std::vector<std::string> tokens = CSVReader::tokenise(input, ',');

  try {
    // Calculate the year range
    int year_difference = std::stoi(tokens[2]) - std::stoi(tokens[1]);

    // Convert region string to enum type
    WeatherEntryType region = WeatherEntry::mapFromInputToRegion(tokens[0]);

    // Validate year range is positive
    if (year_difference < 1) {
      std::cout << "Please choose valid years" << std::endl;
      return;
    }

    // Container for weather data across multiple years
    std::vector<std::vector<WeatherEntry>> weatherDataYearlyEntries;

    // Collect weather data for each year in the range
    for (int i = 0; i <= year_difference; i++) {
      int year = std::stoi(tokens[1]) + i;

      // Get weather entries for current year and region
      std::vector<WeatherEntry> temp = std::get<std::vector<WeatherEntry>>(
          weather.getWeatherEntries(region, std::to_string(year)));
      weatherDataYearlyEntries.push_back(temp);
    }

    // Render the graph using collected data
    ChartRenderer::printGraph(weatherDataYearlyEntries);

  } catch (const std::exception &e) {
    // Handle any errors during processing
    std::cout << "printFilteredChart - there has been an error" << std::endl;
  }
}
```

Figure 9 - Initial implementation of the Graph chart

The graph chart follows the same implementation as done in the Candlestick Chart, with a few differences, first is that we use a step function to pinpoint the index of the relevant temperature start and end in the vector passed through `(the parameter)` i.e.:

```
WeatherEntry start = data_to_render[floor(j / 10)];
WeatherEntry end = data_to_render[floor(j / 10) + 1];
```

After which, we perform linear interpolation between the two points, `start` and `end` as shown above, so we can determine the y value for the x index value when iterating between those indices. The returned y then gets mapped to the relatively correct position on the chart.

The reason for using linear interpolation, is to render the correct line between two data points.

For the entire implementation, it is shown in ***Figure 10:***

```cpp
// Prints a line graph visualization of weather data
void ChartRenderer::printGraph(
    std::vector<std::vector<WeatherEntry>> yearly_entries) {
  std::vector<WeatherEntry> data_to_render;

  // Use more memory for rendering - keep data separate
  for (std::vector<WeatherEntry> entries : yearly_entries) {
    std::string time = entries.begin()->timeframe;

    // Process and prepare data for rendering
    // Extract mean temperatures for each timeframe
    double mean_temp = Weather::getClosingTemp(entries);

    // Create new entry with processed data
    WeatherEntry new_entry{mean_temp, time, entries.begin()->region};

    data_to_render.push_back(new_entry);
  }

  // Set chart dimensions
  unsigned int height = 25;
  unsigned int width = 90;

  std::cout << data_to_render.size() << std::endl;

  // Debug output - print data size and temperatures
  for (int i = 0; i < data_to_render.size(); i++) {
    std::cout << data_to_render[i].temp << std::endl;
  }

  std::cout << std::endl;

  double temp = 50;

  // Iterate through each row of the chart
  for (int i = 0; i < height; i++) {

    // Calculate temperature for current row
    temp = round(mapYCoordFromIndex((double)i) * 10.0) / 10.0;
    std::cout << std::fixed << std::setprecision(1);

    // Iterate through each column
    for (int j = 0; j < width; j++) {
      if (j == 0 && i == height - 1) {
        std::cout << "       └";
      } else if (j == 0) {
        if ((i % 2) == 0) {
          // Prepare the y-axis
          if (temp < -10) {
            std::cout << temp << " ┤";
          } else if (temp < 10 && temp > -1) {
            std::cout << "  " << temp << " ┤";
          } else if (temp < -1 && temp > -10) {
            std::cout << " " << temp << " ┤";
          } else {
            std::cout << " " << temp << " ┤";
          }
        } else {
          std::cout << "       |";
        }
      } else if (i == height - 1) {
        // Populate the x-axis
        // Draw x-axis with tick marks
        if (j % 10 == 0) {
          std::cout << "┬";
        } else {
          std::cout << "─";
        }
      } else {
        // Plot data points and connecting lines
        WeatherEntry start = data_to_render[floor(j / 10)];
        WeatherEntry end = data_to_render[floor(j / 10) + 1];

        if (j % 10 == 0) {
          // Draw data points
          if (start.temp > mapYCoordFromIndex(i + 1) && start.temp < temp) {
            std::cout << "*";
          } else {
            std::cout << " ";
          }
        } else {
          // Draw connecting lines between data points
          double linSpace = linearSpace(start.temp, end.temp, j % 10);

          if (linSpace > mapYCoordFromIndex(i + 1) && linSpace < temp) {
            std::cout << "*";
          } else {
            std::cout << " ";
          }
        }
      }
    }
    std::cout << std::endl;
  }
}
```

Figure 10 - Implementation Graph chart renderer

For how we implement linear implementation, the concept for how we are handling the `x,y` values, ***Figure 11***, shows how we are going to implement the idea. The image also shows the different points that we are looking for between the two x values, this shifts as we get different indices from the initial step function.

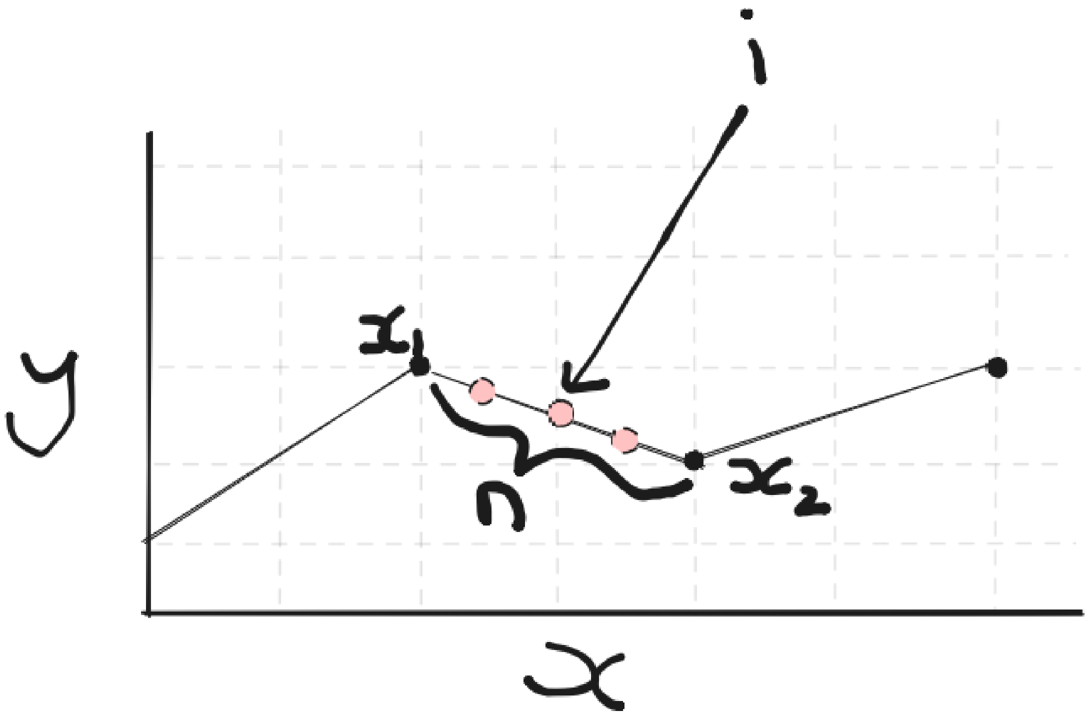

Figure 11 - Linear interpolation example for the graph line between two points

To implement linear interpolation, we use the following formula:

$$x_1 + \frac{i}{n-1}(x_2 - x_1)$$

Where n is the amount of points on the line between x_1 and x_2 and where i is the certain point on the line. Each different interpolated point is drawn using a `*` which acts as a continuation of the line.

For the complete implementation of the formula, this is displayed in ***Figure 12:***

```
// Maps y-coordinate index to temperature value
// Creates the temperature scale on the y-axis
double ChartRenderer::mapYCoordFromIndex(double index) {
  return -0.56 * index + 13.0;
}

// Calculates intermediate points for line drawing between two temperatures
// Uses linear interpolation
double ChartRenderer::linearSpace(double y1, double y2, double i) {
  return y1 + (i / (10 - 1)) * (y2 - y1);
}
```

Figure 12 - Implementation of the mapping functions for the y-axis and the linear interpolation formula

For the output of the final solution, it's displayed in ***Figure 13*** below.

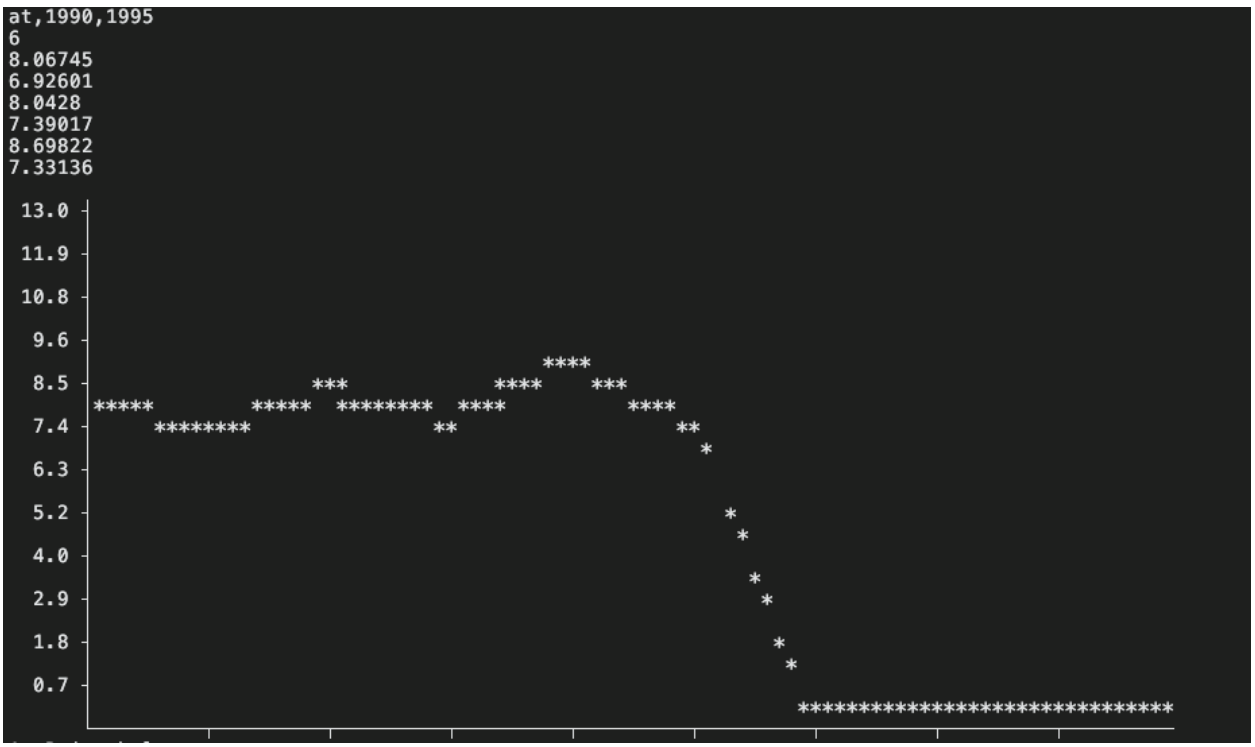

Figure 13 - Final output for the Graph chart implementation

Task 4 - Statistical Prediction implementation

Task 4 begins using the same input implementation as the previous tasks, we get all the data for the region, similar to how we get the data in task 1. Once we have the data, we push that data to our model, which is based on the SARIMA model, which is used for time-series analysis.

Below are our reasons for using the SARIMA:

- Best suited for your data as it specifically handles both trend and seasonal components
- Can capture the annual temperature cycles evident in your dataset
- Particularly effective for data with consistent yearly patterns
- Demonstrates superior accuracy with a lower Mean Squared Error compared to simpler methods

For a initial implementation in `MerkelMain`, reference ***Figure 14***

```cpp
/**
 * Makes temperature predictions using historical data and displays forecast
 * Uses a prediction model to estimate next 5 temperature values
 */
void MerkelMain::printPrediction() {
  // Prompt user for region input
  std::cout << "Enter the region (FR): " << std::endl;
  std::string input;
  std::getline(std::cin, input);

  // Parse input into tokens
  std::vector<std::string> tokens = CSVReader::tokenise(input, ',');

  // Initialize containers for weather data and candlestick patterns
  std::vector<WeatherEntry> temp;
  std::vector<Candlestick> data;

  // Start year for historical data collection
  int year = 1980;

  try {
    // Convert input string to region enum
    WeatherEntryType region = WeatherEntry::mapFromInputToRegion(tokens[0]);

    std::cout << "Making Predictions" << std::endl;

    // Set decimal precision for temperature output
    std::cout << std::fixed << std::setprecision(3);

    // Collect historical temperature data year by year
    do {
      // Get weather entries for current year and region
      temp = std::get<std::vector<WeatherEntry>>(
          weather.getWeatherEntries(region, std::to_string(year)));

      // Break if no data available for current year
      if (temp.size() == 0) {
        break;
      }

      // Calculate temperature metrics for the year
      double lowestTemp = Weather::getLowestTemp(temp);
      double highestTemp = Weather::getHighestTemp(temp);
      double closingTemp = Weather::getClosingTemp(temp);
      double openingTemp = Weather::getOpeningTemp(temp);

      // Create candlestick object from temperature data
      Candlestick candle{openingTemp, closingTemp, highestTemp, lowestTemp};

      // Add to historical data collection
      data.push_back(candle);

      year++;
    } while (temp.size() > 0);

  } catch (const std::exception &e) {
    std::cout << "MerkelMain::printWeatherStats error when mapping and "
                 "retrieving entries"
              << std::endl;
  }

  // Create and train prediction model using historical data
  Prediction model{data};
  model.fit();

  // Generate 5-year temperature forecast
  std::vector<double> forecast = model.predict(5);
  std::cout << "Next 5 temps: " << std::endl;

  // Prepare data structure for visualization
  std::vector<std::vector<WeatherEntry>> chart;

  // Convert forecast data to WeatherEntry format
  for (int i = 0; i < forecast.size(); i++) {
    std::vector<WeatherEntry> predictions;

    // Create WeatherEntry for each predicted temperature
    WeatherEntry prediction{forecast[i],
                            std::to_string(year) + "-01-01T00:00:00Z",
                            WeatherEntry::mapFromInputToRegion(tokens[0])};

    predictions.push_back(prediction);
    chart.push_back(predictions);
  }

  // Display forecast as graph
  ChartRenderer::printGraph(chart);
}
```

Figure 14 - Initial entry point for the prediction graph

```
/**
 * Constructor: Initializes prediction model with historical candlestick data
 * @param candlestickData Vector of historical temperature data points
 */
Prediction::Prediction(std::vector<Candlestick> candlestickData) {
  for (const Candlestick& entry : candlestickData) {
    data.push_back(entry.closingTemp);
  }
}
```

Figure 15 - Constructor for initialising the data

For the SARIMA model, we use the following formula:

$$(1 - \phi_1 B)(1 - \Phi_1 B^s)(1 - B)y_t = (1 + \theta_1 B)(1 + \Theta_1 B^2)\varepsilon_t$$

Where y_t is the temperature value at time t, B is the backshift operator (shift data back one period), ε_t is Error term at time t, s is the seasonal period (1 for yearly data).

$(1 - \phi_1 B)$ is the Autoregressive term, where ϕ_1 is the ***AR*** coefficient, this represents how our current value correlates to our previous values. This is used to find the non-seasonal values.

$(1 - \Phi_1 B^s)$ is used for our seasonal values, where Φ_1 is the seasonal ***AR*** coefficient, models our yearly patterns.

$(1 - B)$ is our regular differencing and $(1 - B^s)$ is seasonal differencing, which helps us remove trends and seasonal patterns.

The backshift function, takes a vector of doubles and returns a new vector of doubles, the purpose of this function is to shift the elements of the input vector to the right by a value of `k` this then allows us to use the last item in the vector for analysis.

```cpp
/**
 * Shifts data backwards by k positions, filling with zeros
 * @param clonedData Input data vector to shift
 * @param k Number of positions to shift
 * @return Vector with shifted data
 */
std::vector<double> Prediction::backshift(std::vector<double> clonedData, int k) {

  std::vector<double> result;

  if (k < clonedData.size()) {
    // Add zeroes to the beginning
    result.insert(result.begin(), k, 0.0);

    // Add data[:-k]
    result.insert(result.end(), clonedData.begin(), clonedData.end() - k);
  } else {
    // If K >= data.size(), return a vector of zeroes
    result.resize(clonedData.size(), 0.0);
  }

  return result;
}
```

Figure 16 - Selecting previous temperature values

When we calculate ***AR*** component, we backshift by one value and then we iterate over the data and multiply that value by our non-seasonal ϕ element and save that to the new result vector.

```cpp
/**
 * Calculates the autoregressive (AR) component of the prediction
 * @param clonedData Input data vector
 * @return Vector containing AR components
 */
std::vector<double> Prediction::arComponent(std::vector<double> clonedData) {
  std::vector<double> shifted = backshift(clonedData);

  std::vector<double> result;

  for (int i = 0; i < clonedData.size(); i++) {
    result.push_back(shifted[i] * phi);
  }

  return result;
}
```

Figure 17 - Calculates how correlated the prediction with be to the previous values non-seasonal

The same principle applies to the seasonal ***AR*** values, so we backshift and then apply the Φ seasonal component to each item in the vector.

```
/**
 * Calculates the seasonal autoregressive component of the prediction
 * @param clonedData Input data vector
 * @return Vector containing seasonal AR components
 */
std::vector<double> Prediction::seasonalArComponent(std::vector<double> clonedData) {
  std::vector<double> shifted = backshift(clonedData, 1);

  std::vector<double> result;

  for (int i = 0; i < clonedData.size(); i++) {
    result.push_back(shifted[i] * PHI);
  }

  return result;
}
```

Figure 18 - Calculates how correlated the prediction with be to the previous values seasonal (yearly)

Once we have calculated the ***AR*** components we add them together and push them to the back of the initial vector and then we continue with the next time stage of the prediction.

```
/**
 * Generates future predictions based on historical data
 * @param steps Number of future steps to predict
 * @return Vector of predicted values
 */
std::vector<double> Prediction::predict(int steps){
  std::vector<double> predictions;
  std::vector<double> working_data = data;

  // Loop without index needed
  // Generate predictions for specified number of steps
  for (int _ = 0; _ < steps; _++) {
    double ar = arComponent(working_data).back();
    double seasonalAr = seasonalArComponent(working_data).back();

    double nextValue = ar + seasonalAr;
    predictions.push_back(nextValue);
    working_data.push_back(nextValue);
  }

  return predictions;
}
```

Figure 19 - Calculating the predictions

Trains our model by predicting values and setting our correlation and learning rate for the AR components.

```
/**
 * Trains the model by adjusting parameters to minimize prediction error
 * @param epochs Number of training iterations
 */
void Prediction::fit(int epochs) {
  for (int _ = 0; _ < epochs; _++) {
    std::vector<double> prediction = predict(1);

    double error = prediction[0] - data[data.size() - 1];

    phi -= learningRate * error * data[data.size() - 2];
    PHI -= learningRate * error * data[data.size() - 1];
  }
}
```

Figure 20 - “Training” our model to predict the time-series analysis

Once the predictions are complete we display those prediction on the same graph as task 3.

For the final output, this is displayed in Figure 21

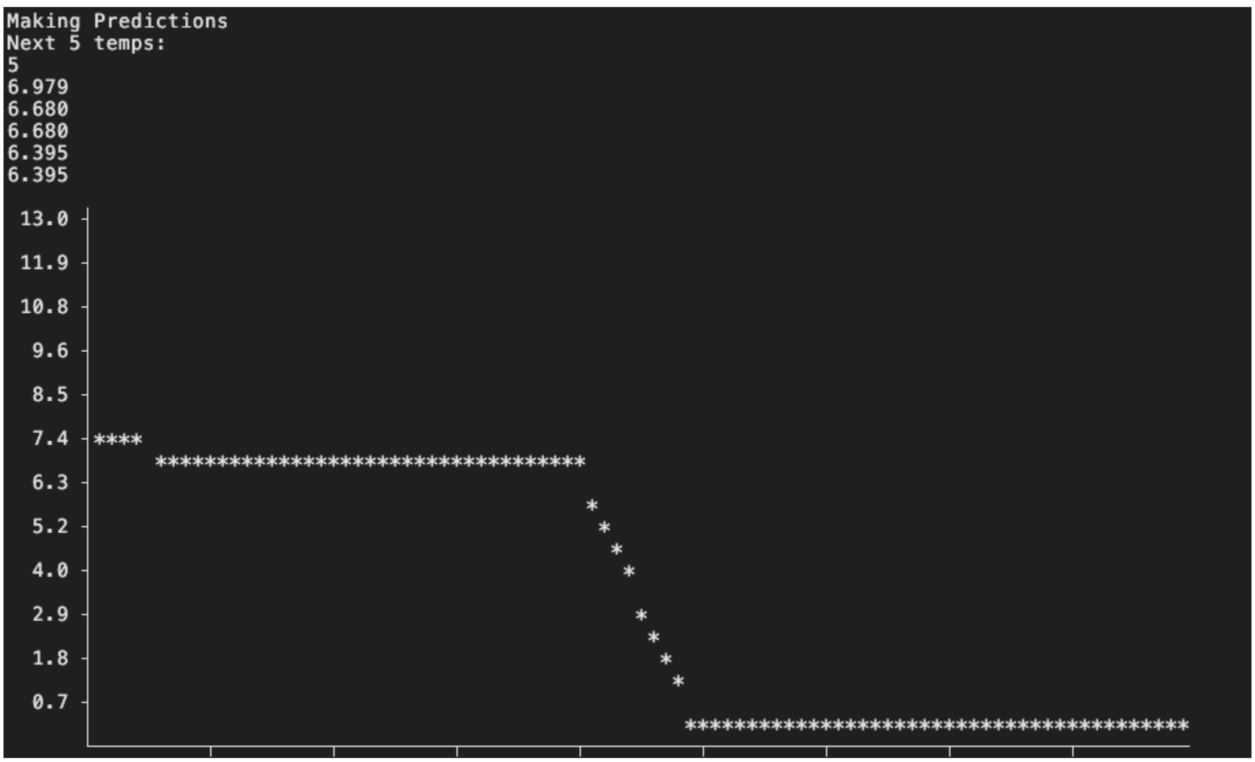

Figure 21 - Graph showing the final output of the prediction model.

Conclusion

We have implemented all tasks given, along with examples and explanations for why we chose a specific way of solving the problems and then showed the end result in the report.

References

Singh, S.K., Singh, R. & Sherigar, S. 2024, 'Temperature Forecasting and Analysis Using Linear and Timeseries Models', IRE Journals, vol. 7, no. 8, pp. 78-84