# Assignment # 5, MIE 1628 Cloud Data Analytics

Student Name: Nazib Chowdhury

Student ID: # 1004694235

## 1 Part A:

### 1.1 A.1:

1. Ingest -> Azure Data Factory

2. Data Store -> Azure Data Lake

3. Prepare and Transform Data -> Azure Databricks

4. Model and Serve Data -> Azure Synapse Analytics

### 1.2 A.2:

**Explain how Stream Analytics works in Azure. Mention at least two common use cases or applications for this service**

Azure Stream Analytics is tailored for real time streaming data analytics tasks. So it can consume data in real time, very quickly, and process it according to our given functions and rapidly send the results to databases, online dashboards or apps in real time.

**Workflow of Azure Stream Analytics**

1. Input from Sources:
   typically input from IoT devices, with ingestion triggered by Azure Event Hubs, and then transmitted to Azure Blob Storage

2. Processing of Data:
   The data is automatically processed with SQL like language to undergo some transformations that we want before storing it.

3. Storing and Analytics:
   The output data from the processing is finally sent to store in SQL Database, or sent for visualization using tools like Power BI for real time data analytics.

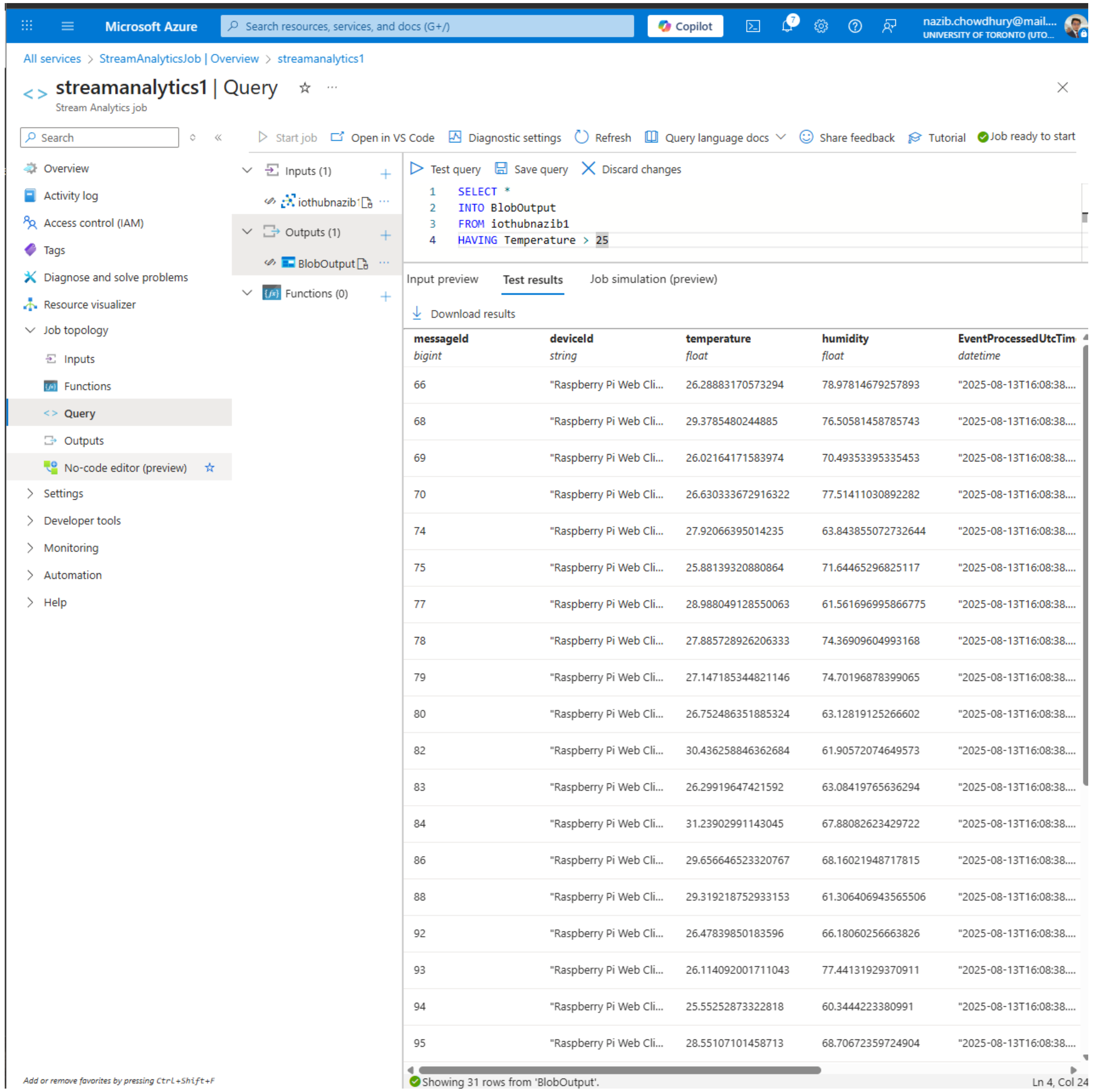
**Common Use cases of this application:**

1. **IoT Device Data:**
   IoT device data collection happens in real time and with a very fast frequency, often around (10-30 Hz) aligning with the frequency of the sensor. For more higher frequency physical processes like vibration, the data collection frequency can be even higher. For this reason, IoT device monitoring requires stream analytics to be done in real time. Because in the event of machine failure from say overheating, the symptoms arise very fast

and does not allow for much time before the system breaks down. Therefore, IoT device monitoring require real time analytics

2. **Real Time Fraud Detection:**
   Fraud detection is another useful case of stream analytics. Fraudulent operations take place very fast, and hence these activities like fraudulent website cookies, pop-ups, emails etc and even fraudulent transaction requests need to be monitored in real time.

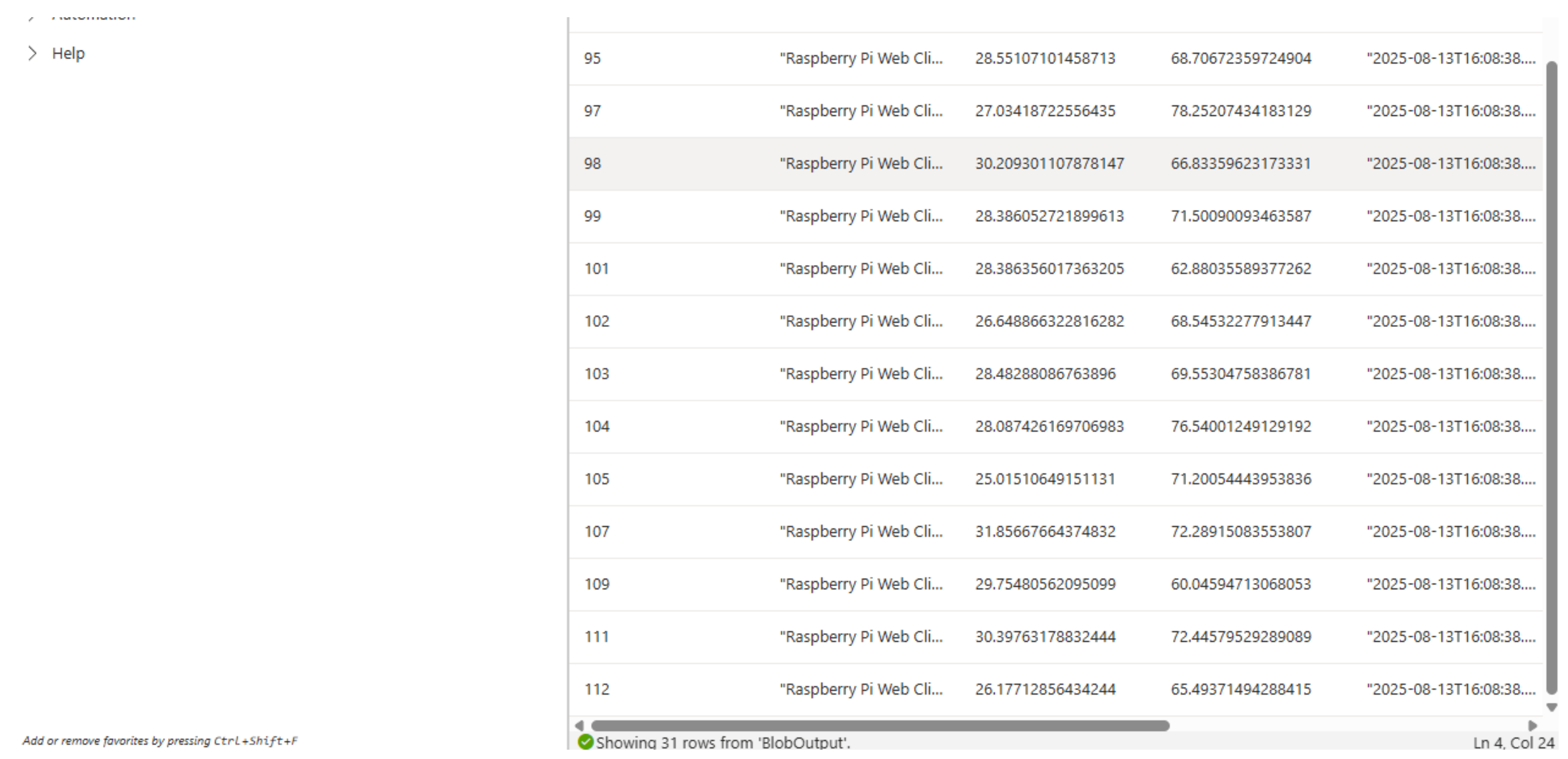
## 1.3 A.3



```
SELECT *
INTO BlobOutput
FROM iothubnazib1
HAVING Temperature > 25
```

| messageId<br>*bigint* | deviceId<br>*string* | temperature<br>*float* | humidity<br>*float* | EventProcessedUtcTim<br>*datetime* |
|---|---|---|---|---|
| 66 | "Raspberry Pi Web Cli... | 26.28883170573294 | 78.97814679257893 | "2025-08-13T16:08:38.... |
| 68 | "Raspberry Pi Web Cli... | 29.3785480244885 | 76.50581458785743 | "2025-08-13T16:08:38.... |
| 69 | "Raspberry Pi Web Cli... | 26.02164171583974 | 70.49353395335453 | "2025-08-13T16:08:38.... |
| 70 | "Raspberry Pi Web Cli... | 26.630333672916322 | 77.51411030892282 | "2025-08-13T16:08:38.... |
| 74 | "Raspberry Pi Web Cli... | 27.92066395014235 | 63.843855072732644 | "2025-08-13T16:08:38.... |
| 75 | "Raspberry Pi Web Cli... | 25.88139320880864 | 71.64465296825117 | "2025-08-13T16:08:38.... |
| 77 | "Raspberry Pi Web Cli... | 28.988049128550063 | 61.561696995866775 | "2025-08-13T16:08:38.... |
| 78 | "Raspberry Pi Web Cli... | 27.885728926206333 | 74.36909604993168 | "2025-08-13T16:08:38.... |
| 79 | "Raspberry Pi Web Cli... | 27.147185344821146 | 74.70196878399065 | "2025-08-13T16:08:38.... |
| 80 | "Raspberry Pi Web Cli... | 26.752486351885324 | 63.12819125266602 | "2025-08-13T16:08:38.... |
| 82 | "Raspberry Pi Web Cli... | 30.436258846362684 | 61.90572074649573 | "2025-08-13T16:08:38.... |
| 83 | "Raspberry Pi Web Cli... | 26.29919647421592 | 63.08419765636294 | "2025-08-13T16:08:38.... |
| 84 | "Raspberry Pi Web Cli... | 31.23902991143045 | 67.88082623429722 | "2025-08-13T16:08:38.... |
| 86 | "Raspberry Pi Web Cli... | 29.656646523320767 | 68.16021948717815 | "2025-08-13T16:08:38.... |
| 88 | "Raspberry Pi Web Cli... | 29.319218752933153 | 61.306406943565506 | "2025-08-13T16:08:38.... |
| 92 | "Raspberry Pi Web Cli... | 26.47839850183596 | 66.18060256663826 | "2025-08-13T16:08:38.... |
| 93 | "Raspberry Pi Web Cli... | 26.114092001711043 | 77.44131929370911 | "2025-08-13T16:08:38.... |
| 94 | "Raspberry Pi Web Cli... | 25.55252873322818 | 60.3444223380991 | "2025-08-13T16:08:38.... |
| 95 | "Raspberry Pi Web Cli... | 28.55107101458713 | 68.70672359724904 | "2025-08-13T16:08:38.... |

Showing 31 rows from 'BlobOutput'. Ln 4, Col 24



| | | | | |
|---|---|---|---|---|
| 95 | "Raspberry Pi Web Cli... | 28.55107101458713 | 68.70672359724904 | "2025-08-13T16:08:38.... |
| 97 | "Raspberry Pi Web Cli... | 27.03418722556435 | 78.25207434183129 | "2025-08-13T16:08:38.... |
| 98 | "Raspberry Pi Web Cli... | 30.209301107878147 | 66.83359623173331 | "2025-08-13T16:08:38.... |
| 99 | "Raspberry Pi Web Cli... | 28.386052721899613 | 71.50090093463587 | "2025-08-13T16:08:38.... |
| 101 | "Raspberry Pi Web Cli... | 28.386356017363205 | 62.88035589377262 | "2025-08-13T16:08:38.... |
| 102 | "Raspberry Pi Web Cli... | 26.648866322816282 | 68.54532277913447 | "2025-08-13T16:08:38.... |
| 103 | "Raspberry Pi Web Cli... | 28.48288086763896 | 69.55304758386781 | "2025-08-13T16:08:38.... |
| 104 | "Raspberry Pi Web Cli... | 28.087426169706983 | 76.54001249129192 | "2025-08-13T16:08:38.... |
| 105 | "Raspberry Pi Web Cli... | 25.01510649151131 | 71.20054443953836 | "2025-08-13T16:08:38.... |
| 107 | "Raspberry Pi Web Cli... | 31.85667664374832 | 72.28915083553807 | "2025-08-13T16:08:38.... |
| 109 | "Raspberry Pi Web Cli... | 29.75480562095099 | 60.04594713068053 | "2025-08-13T16:08:38.... |
| 111 | "Raspberry Pi Web Cli... | 30.39763178832444 | 72.44579529289089 | "2025-08-13T16:08:38.... |
| 112 | "Raspberry Pi Web Cli... | 26.17712856434244 | 65.49371494288415 | "2025-08-13T16:08:38.... |

Showing 31 rows from 'BlobOutput'. Ln 4, Col 24

# 2 Part B:

## 2.1 B.1: Problem Statement

### 2.1.1 Info on Hierarchical Time-Series Data:

Time series data usually refers to any data that changes value with respect to time. This could be the price of a commodity or stock, the demand for electricity, or the weather. Time series forecasting is widely used for the purpose of forecasting financial value, market demand, electricity demand, weather changes etc. For this reason, there is great interest in the research community to develop good forecasting algorithms for time series data.

One problem being investigated in regards to time series forecasting is making **coherent** predictions, i.e. predictions whose value adds up to a higher level of time series forecast. To understand this deeper, here is an explanation of **hierarchical time series data.**

Hierarchical time series refers to a set of different univariant time series data that are related to each other up the hierarchy through algebraic sums of one another. [1]

For example, if a time-series exist at the top-most hierarchy (level 0), and there are some more time-series existing at the immediate lower hierarchy (level 1), then the level 0 time series value at any point in time can be achieved by an algebraic sum of all its children time-series at its immediate lower level of hierarchy (level 0). A simple example is that for all sales data in every branch of a supermarket in a city, the sales data of all these stores must add up to the city-level sales data of that supermarket brand.

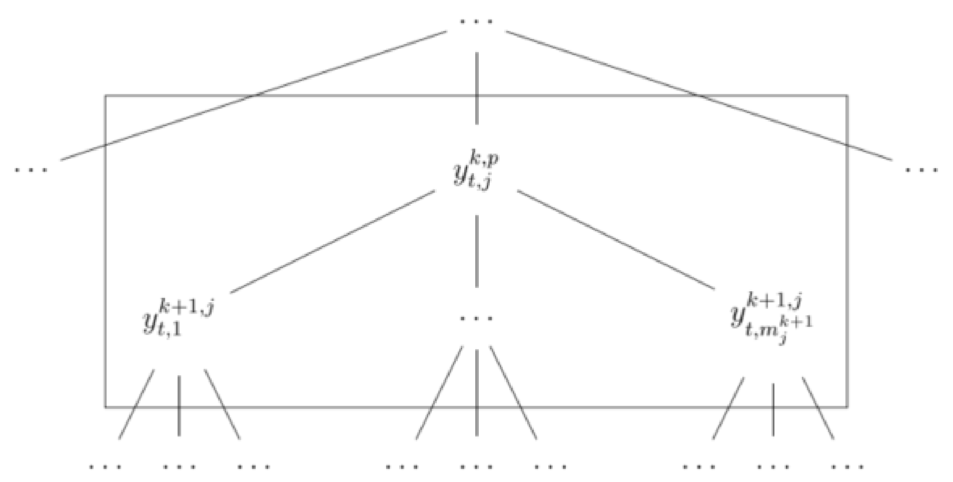


*Figure 2-1A hierarchical time series level at level k along with its children series in the immediate lower level k+1 [2]*

One of the interesting problems regarding hierarchical time series data, is while training machine learning algorithms to make a forecast on any time-series at a hierarchy, we can get better results by also including in the training, its children series at the immediate lower hierarchy. [2]

### 2.1.2 Problem Statement:

**Problem Statement:** In this project, I will train and evaluate the machine learning model based on architecture described in [2] to properly forecast hierarchical time-series data. I will use the dataset of Italian pasta sales used in [2] to train and evaluate this model.

### 2.1.3 Dataset Description:

**Dataset Name:** Hierarchical Sales Data

**Dataset website:** https://archive.ics.uci.edu/dataset/611/hierarchical+sales+data

**Dataset Download Website:** https://data.mendeley.com/datasets/njdkntcpc9/1

This dataset contains hierarchical sales data gathered from **an Italian grocery store**.

The dataset consists of 118 **daily time series** representing **the quantity of pasta products sold** from 01/01/2014 to 31/12/2018 of 4 national pasta brands. Besides univariate time series data, the quantity sold is integrated by information on the presence or the absence of a promotion. These time series can be naturally arranged to follow a 3-level hierarchical structure (see https://www.sciencedirect.com/science/article/pii/S0957417421005431). (Description copied from: [3])

**Dataset Characteristics =** Time-Series

**# Instances =** 1798

**# Features =** 237

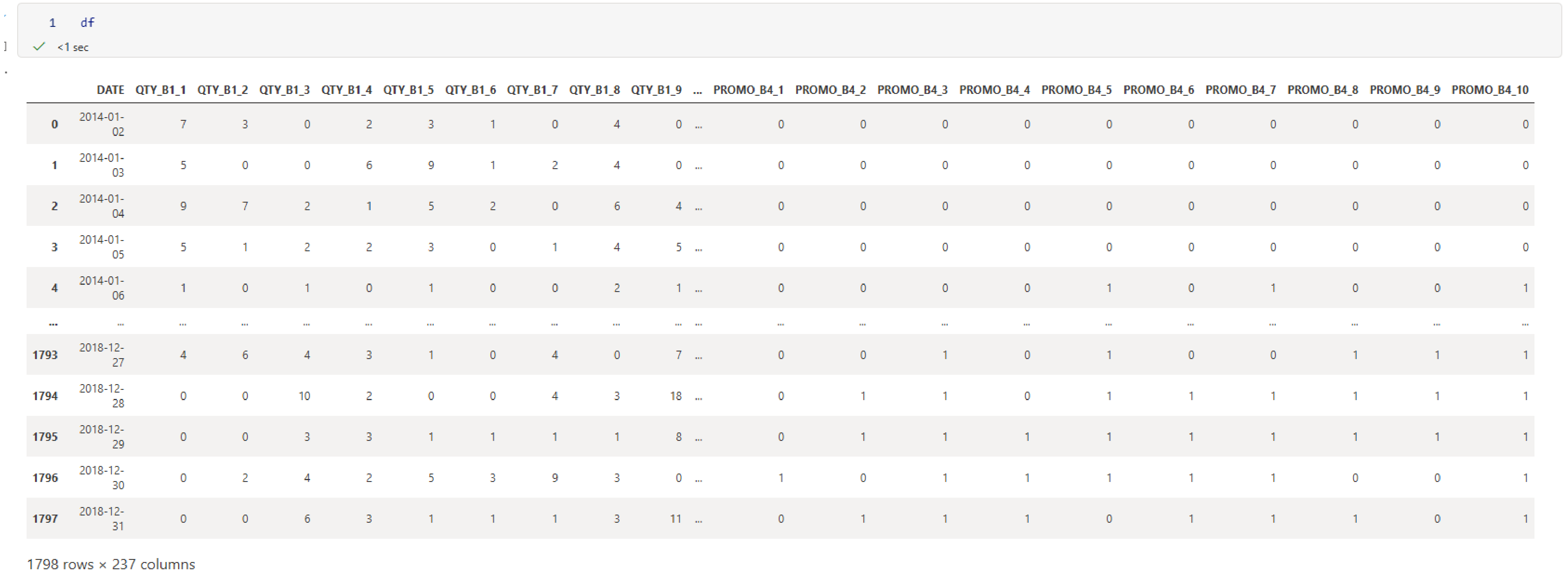
As described in the previous section, this dataset has 237 feature columns including the date column. Hence there are 236 features to train on. Among these 118 features are **daily time-series data** of sales of pasta brands. The remaining 118 features are **explanatory variables/ external regressors** , i.e., further information about the sales like presence of daily promotion.

Hence,

237 features = 1 date column + 118 daily time series + 118 external regressors

Each time series has 5 years of sales data, with granularity of 1 day. (Some sales were recorded only in working days, hence not all days of these 5 years are present)

Below, is the dataframe shown with column names



```
df
```

✓ <1 sec

| | DATE | QTY_B1_1 | QTY_B1_2 | QTY_B1_3 | QTY_B1_4 | QTY_B1_5 | QTY_B1_6 | QTY_B1_7 | QTY_B1_8 | QTY_B1_9 | ... | PROMO_B4_1 | PROMO_B4_2 | PROMO_B4_3 | PROMO_B4_4 | PROMO_B4_5 | PROMO_B4_6 | PROMO_B4_7 | PROMO_B4_8 | PROMO_B4_9 | PROMO_B4_10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 2014-01-02 | 7 | 3 | 0 | 2 | 3 | 1 | 0 | 4 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 2014-01-03 | 5 | 0 | 0 | 6 | 9 | 1 | 2 | 4 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 2014-01-04 | 9 | 7 | 2 | 1 | 5 | 2 | 0 | 6 | 4 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 2014-01-05 | 5 | 1 | 2 | 2 | 3 | 0 | 1 | 4 | 5 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | 2014-01-06 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 2 | 1 | ... | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **1793** | 2018-12-27 | 4 | 6 | 4 | 3 | 1 | 0 | 4 | 0 | 7 | ... | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| **1794** | 2018-12-28 | 0 | 0 | 10 | 2 | 0 | 0 | 4 | 3 | 18 | ... | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| **1795** | 2018-12-29 | 0 | 0 | 3 | 3 | 1 | 1 | 1 | 1 | 8 | ... | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **1796** | 2018-12-30 | 0 | 2 | 4 | 2 | 5 | 3 | 9 | 3 | 0 | ... | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| **1797** | 2018-12-31 | 0 | 0 | 6 | 3 | 1 | 1 | 1 | 3 | 11 | ... | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |

1798 rows × 237 columns

*Figure 2-2 Code snippet showing the column names and values of the dataset*

**Naming of Dataset Columns:**

- QTY_B'X'_'Y' - the quantity sold for brand 'X' item 'Y'
- PROMO_B'X'_'Y' - the promotion flag for brand 'X' and item 'Y')

### 2.1.4 Modelling Plan:

Summary of the data-preprocessing steps:

1. Identify the parent-child relations among the data columns, and group them together.
2. Concatenate the explanatory variables (promotion) data to the input time series data to create a single input vector
3. Create the 3-level hierarchical structure of the dataset.

4. Data-cleaning is unnecessary in this case as the dataset does not have missing values or irrelevant information.

The first step in data-preprocessing is to relate the relevant columns of the dataset to each other. The previous section outlines the different columns present in the dataset. Our first task is to group all the columns of pasta products (items Y) that fall under the same brand X. This will give us the parent-child relation of the hierarchical dataset.

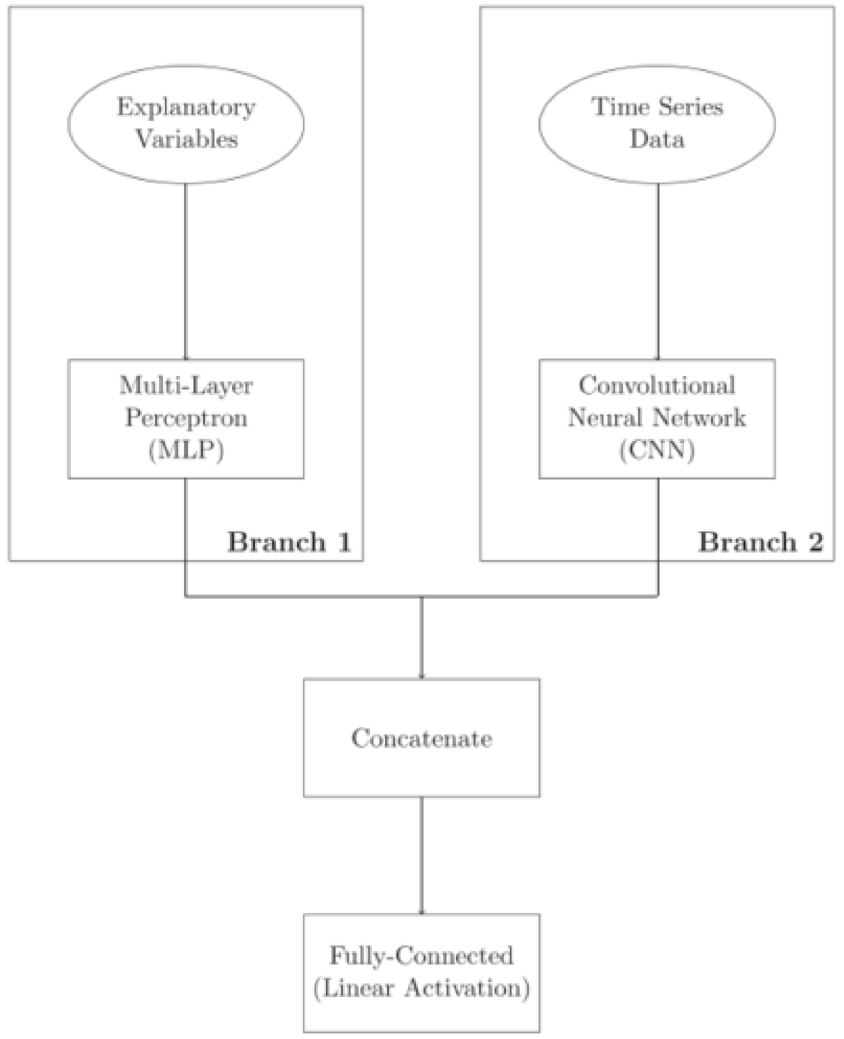

Explanatory Variables
Time Series Data
Multi-Layer Perceptron (MLP)
Convolutional Neural Network (CNN)
Branch 1
Branch 2
Concatenate
Fully-Connected (Linear Activation)

*Figure 2-3The ML model will train on concatenated data, one part of the data is the time series data, another part is the explanatory variables (presence or absence of promotion). This input concatenation will be done separately for each distinct brand of pasta [2]*

To train the ML model, we will concatenate the daily time series vector and explanatory variables vector to create a single input vector to train the ML forecasting model for each pasta brand. This concatenation of input vector will only contain the historic time series data of the series we are forecasting on, its immediate children in the hierarchy, and all explanatory variables (presence or absence of promotion of the included pasta brands or products). Hence for each different time series, the input vector will be different. We use the following naming convention described in the section – **Dataset Description**: to select the appropriate input vector for each time series forecast.

Lastly, we will create the hierarchical structure according to [2]

**2 level hierarchical structure implemented in** [2]**:**

1. Level-1: Sales data for each pasta brand (4 brands in total)

2. Level-2: Sales data for each item of the pasta brand. (Every item must fall into a child category of each of the above 4 mentioned branches.)

My hierarchy structure is different from [2] because I implement a 2-level hierarchy for simplicity whereas the paper [2] implements the 3 level hierarchy.

After doing all this data pre-processing and grouping of relevant data, we can move onto the next step of designing the ML architecture, and training the model

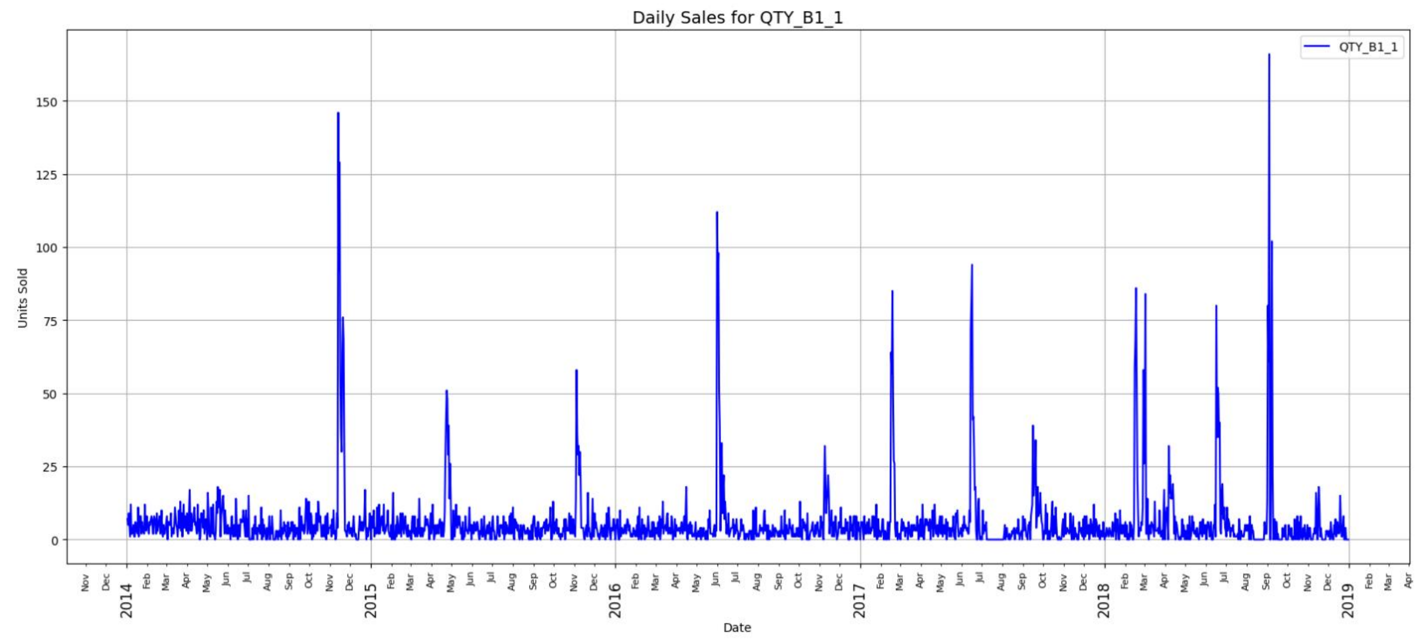
### 2.1.5 Data Pre-processing & Cleaning Requirements:

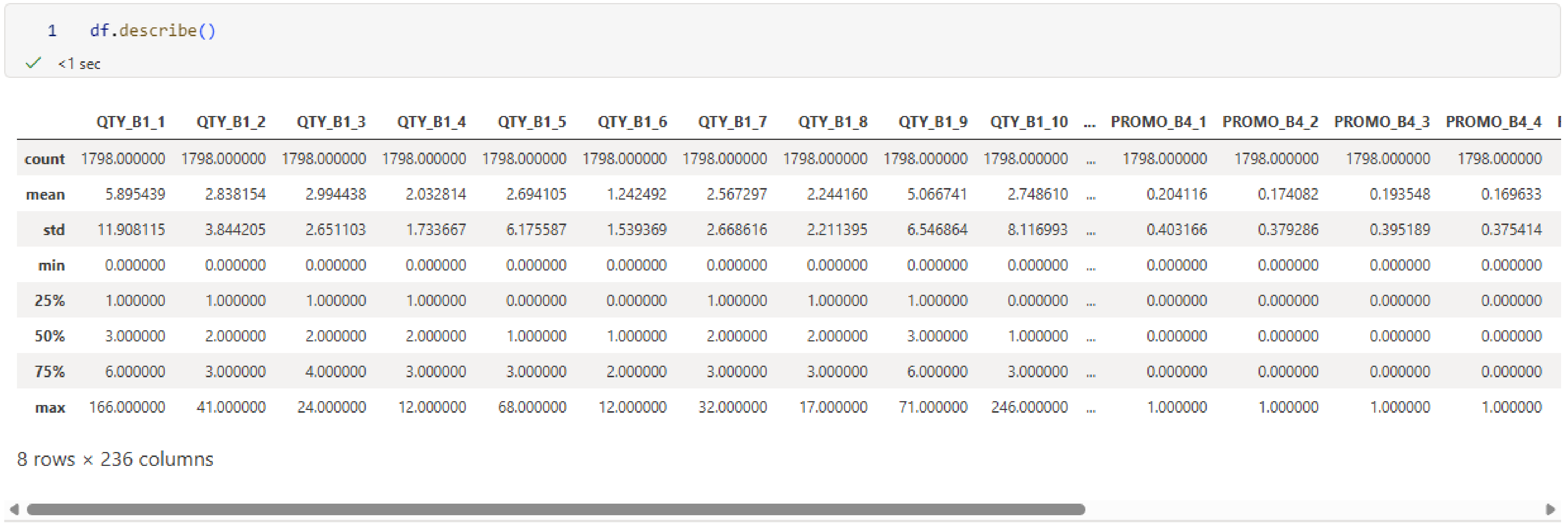
There might be some missing values in the dataset, these require to be filled by zero for proper time series predictions. Further description of the data cleaning and pre-processing work is mentioned in the following sections

## 2.2 Exploratory Data Analysis (EDA)

### 2.2.1 Daily Sales Time Series:

Here are some of the daily sales time series for some items across the entire time range (from 2014/01/01 to 2018/12/31)



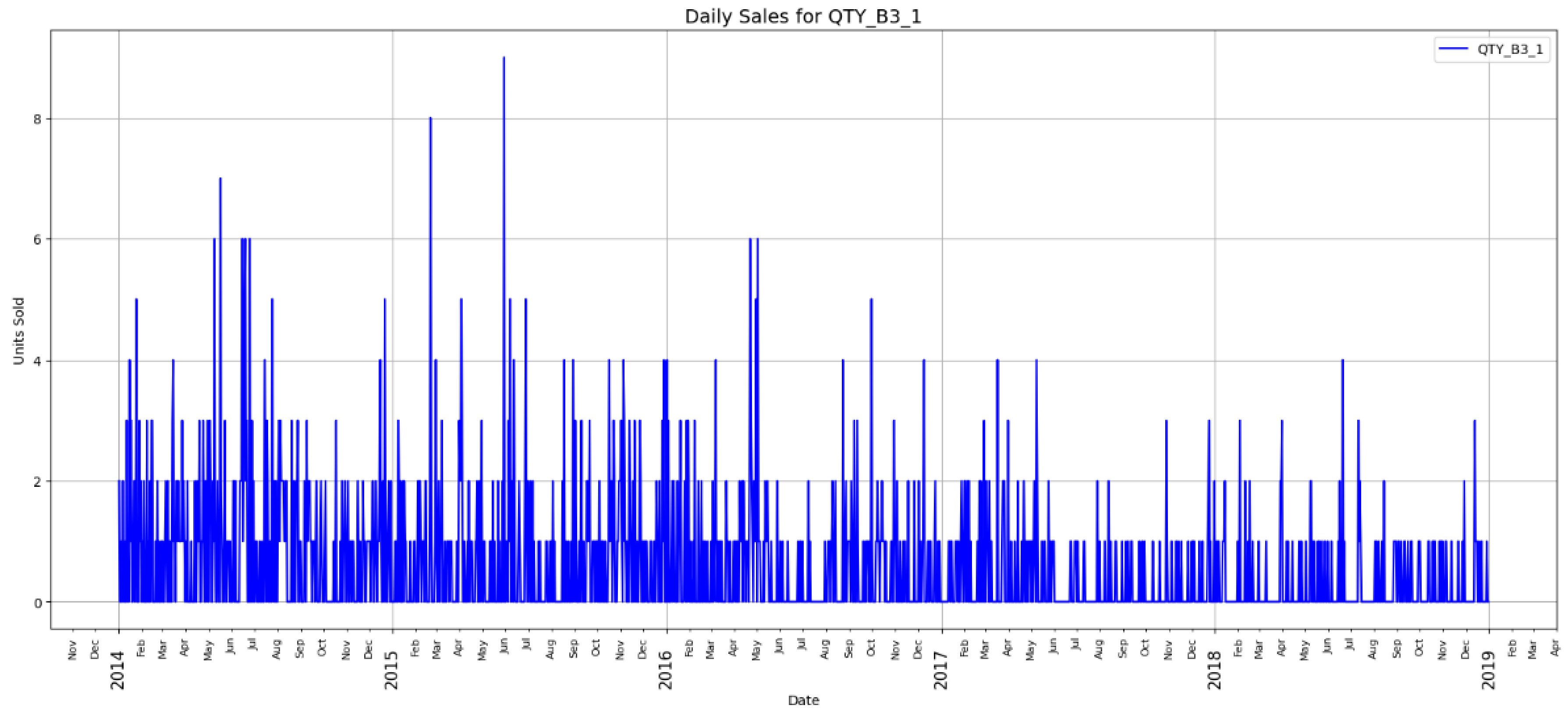


```
df.describe()
```

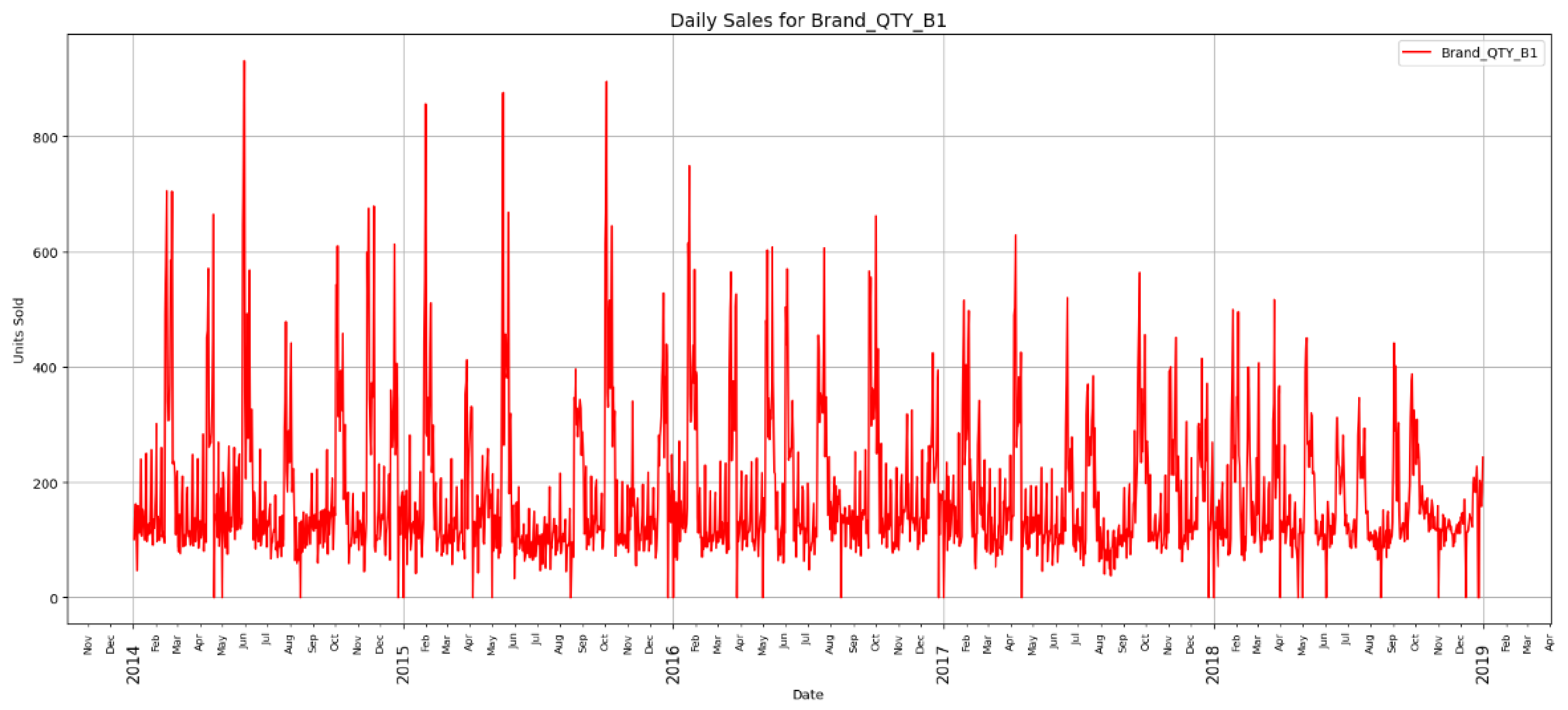
✓ <1 sec

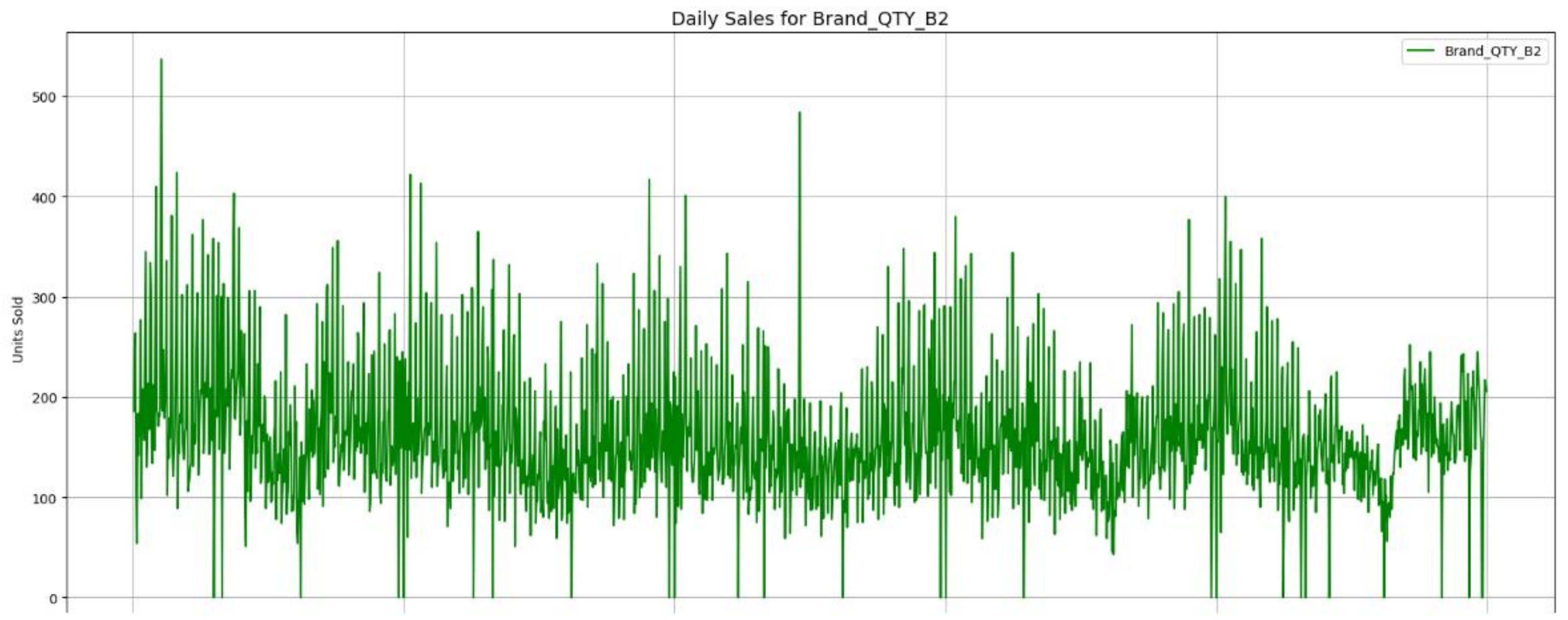
| | QTY_B1_1 | QTY_B1_2 | QTY_B1_3 | QTY_B1_4 | QTY_B1_5 | QTY_B1_6 | QTY_B1_7 | QTY_B1_8 | QTY_B1_9 | QTY_B1_10 | ... | PROMO_B4_1 | PROMO_B4_2 | PROMO_B4_3 | PROMO_B4_4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 1798.000000 | 1798.000000 | 1798.000000 | 1798.000000 | 1798.000000 | 1798.000000 | 1798.000000 | 1798.000000 | 1798.000000 | 1798.000000 | ... | 1798.000000 | 1798.000000 | 1798.000000 | 1798.000000 |
| **mean** | 5.895439 | 2.838154 | 2.994438 | 2.032814 | 2.694105 | 1.242492 | 2.567297 | 2.244160 | 5.066741 | 2.748610 | ... | 0.204116 | 0.174082 | 0.193548 | 0.169633 |
| **std** | 11.908115 | 3.844205 | 2.651103 | 1.733667 | 6.175587 | 1.539369 | 2.668616 | 2.211395 | 6.546864 | 8.116993 | ... | 0.403166 | 0.379286 | 0.395189 | 0.375414 |
| **min** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | ... | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| **25%** | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 0.000000 | 0.000000 | 1.000000 | 1.000000 | 1.000000 | 0.000000 | ... | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| **50%** | 3.000000 | 2.000000 | 2.000000 | 2.000000 | 1.000000 | 1.000000 | 2.000000 | 2.000000 | 3.000000 | 1.000000 | ... | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| **75%** | 6.000000 | 3.000000 | 4.000000 | 3.000000 | 3.000000 | 2.000000 | 3.000000 | 3.000000 | 6.000000 | 3.000000 | ... | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| **max** | 166.000000 | 41.000000 | 24.000000 | 12.000000 | 68.000000 | 12.000000 | 32.000000 | 17.000000 | 71.000000 | 246.000000 | ... | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

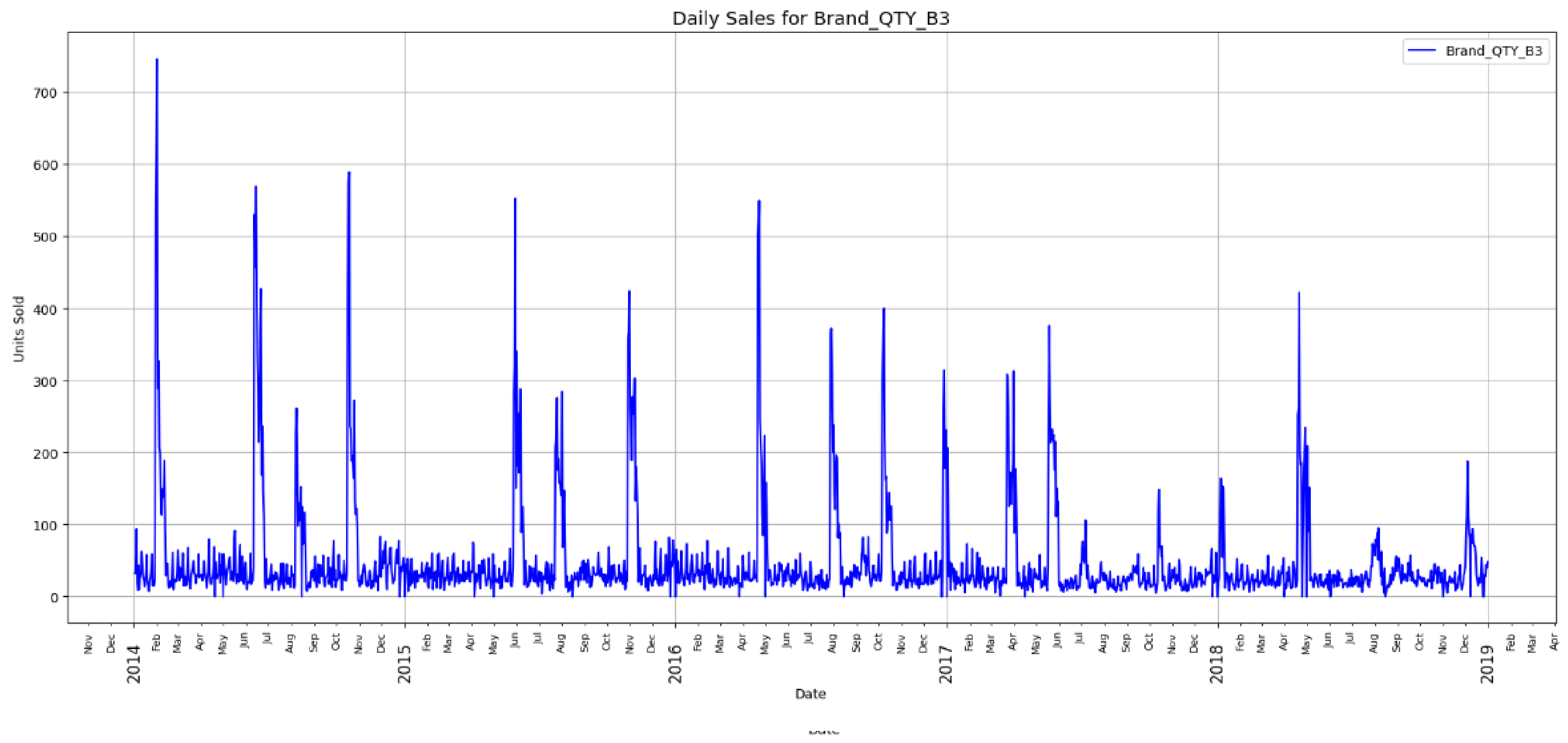
8 rows × 236 columns

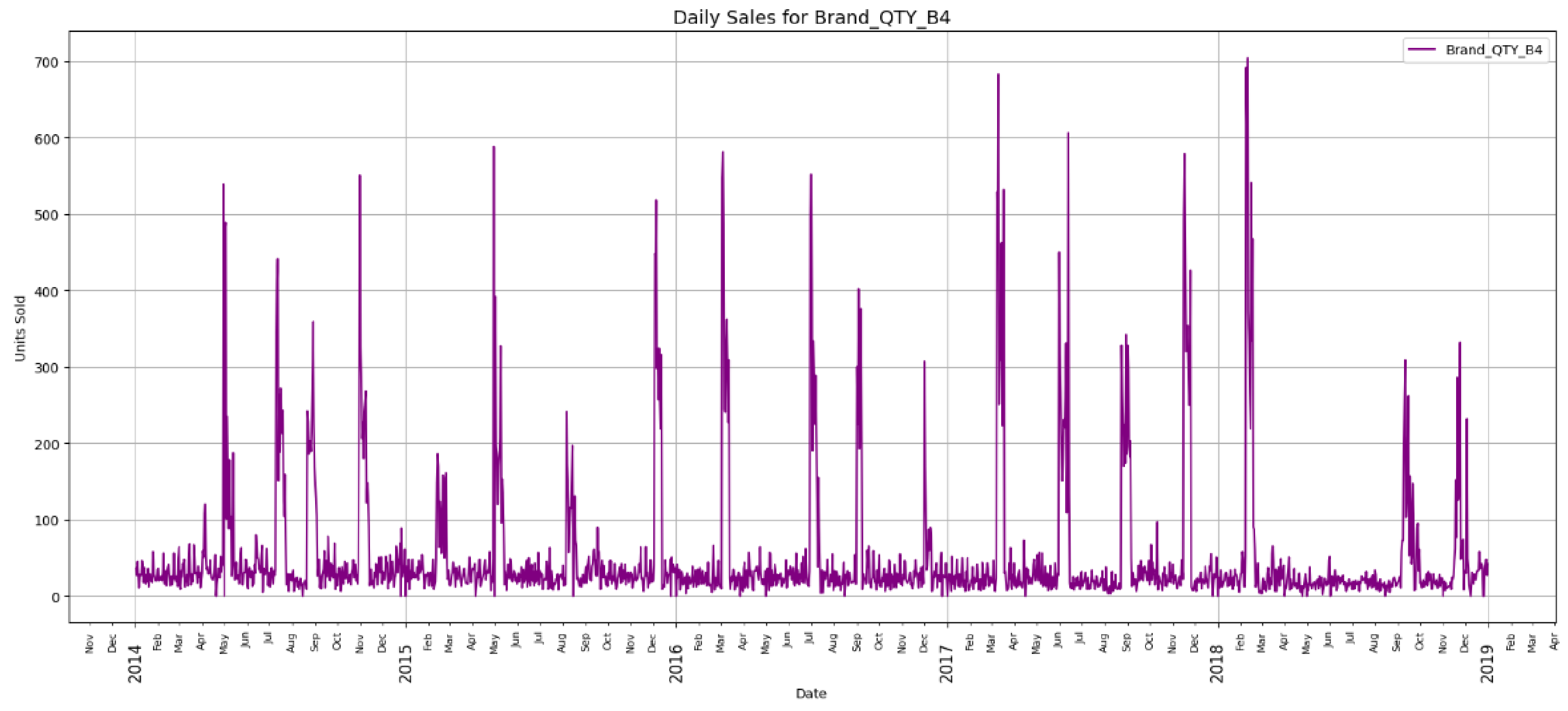


For all brands.

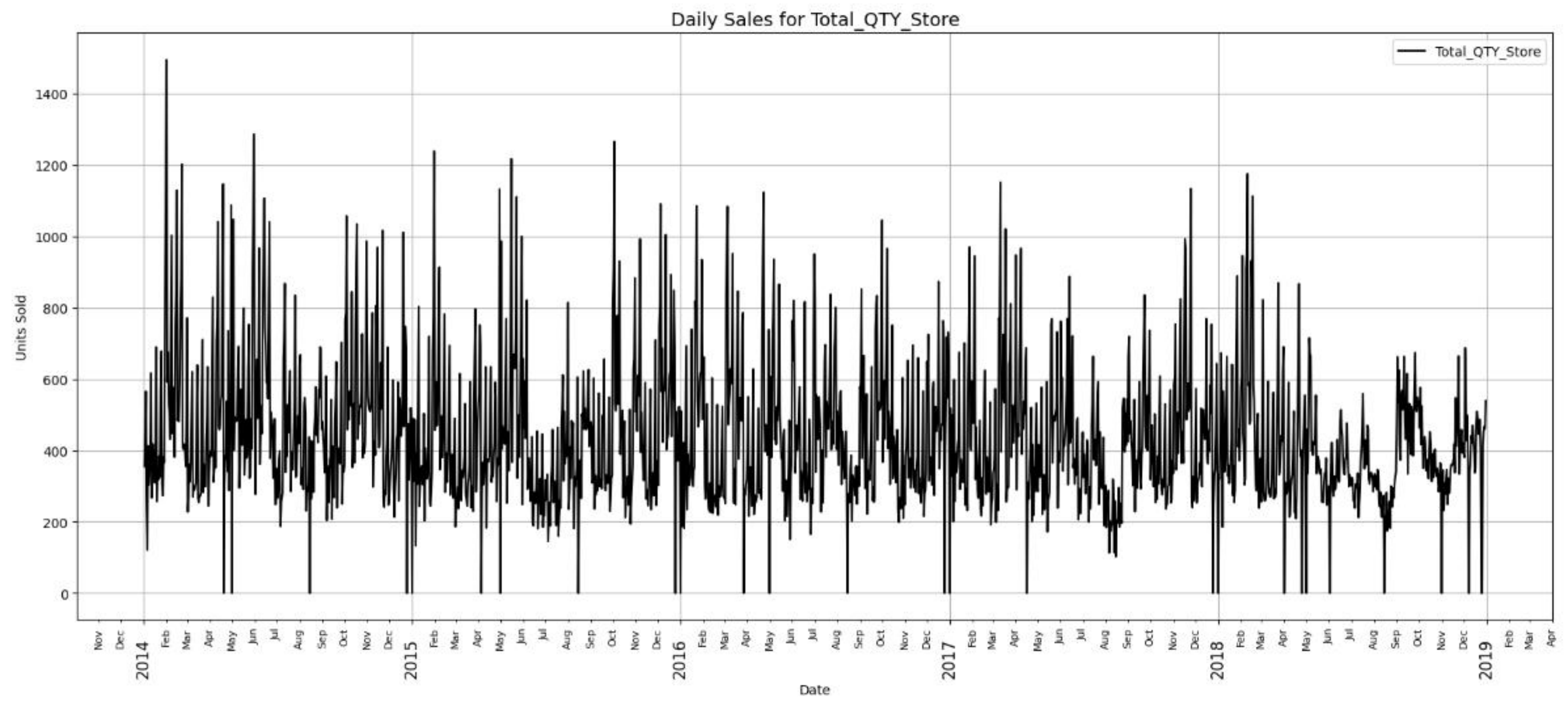

Daily Sales for Brand_QTY_B1
Brand_QTY_B1
Units Sold
Date
2014
2015
2016
2017
2018
2019


Daily Sales for Brand_QTY_B2
Brand_QTY_B2
Units Sold


Daily Sales for Brand_QTY_B3
Brand_QTY_B3
Units Sold
Date
2014
2015
2016
2017
2018
2019

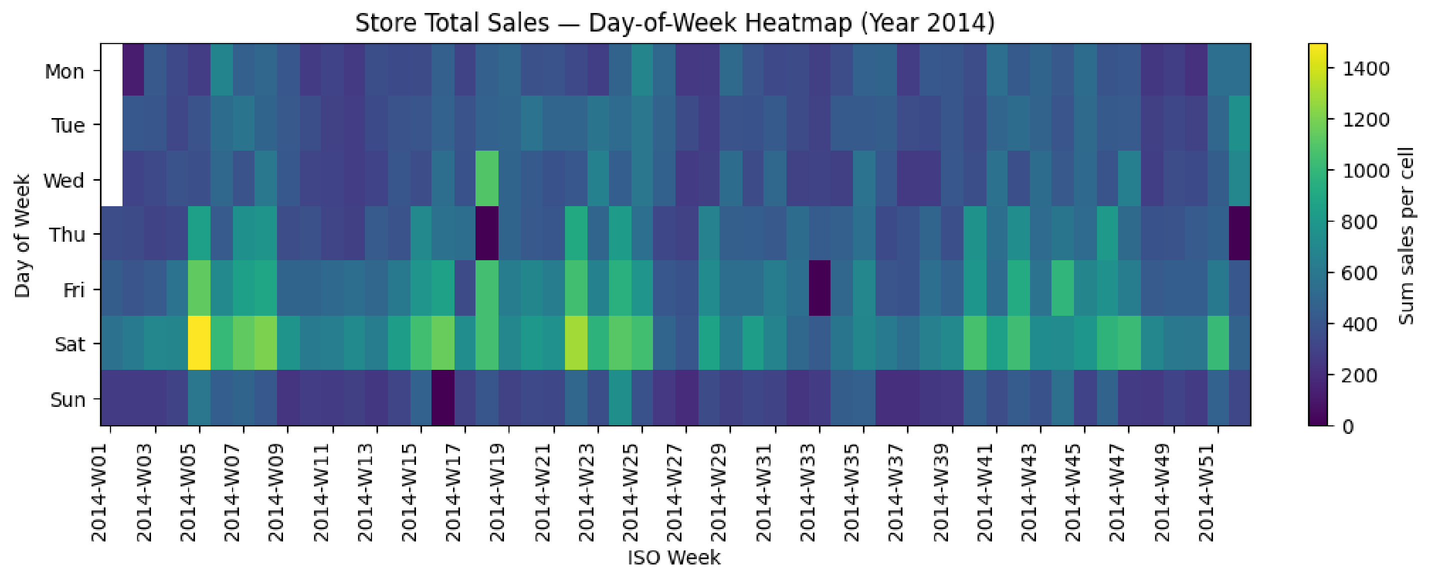

Daily Sales for Brand_QTY_B4
Brand_QTY_B4
Units Sold
Date
2014
2015
2016
2017
2018
2019

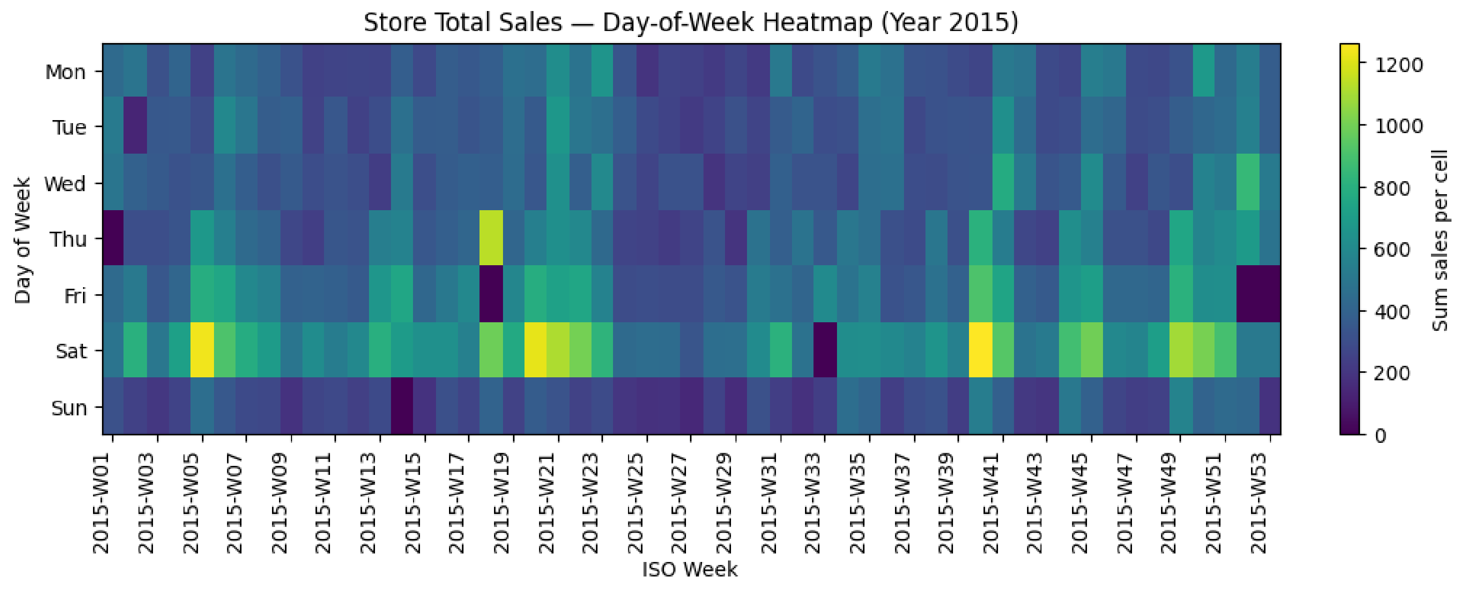
Total Sales:

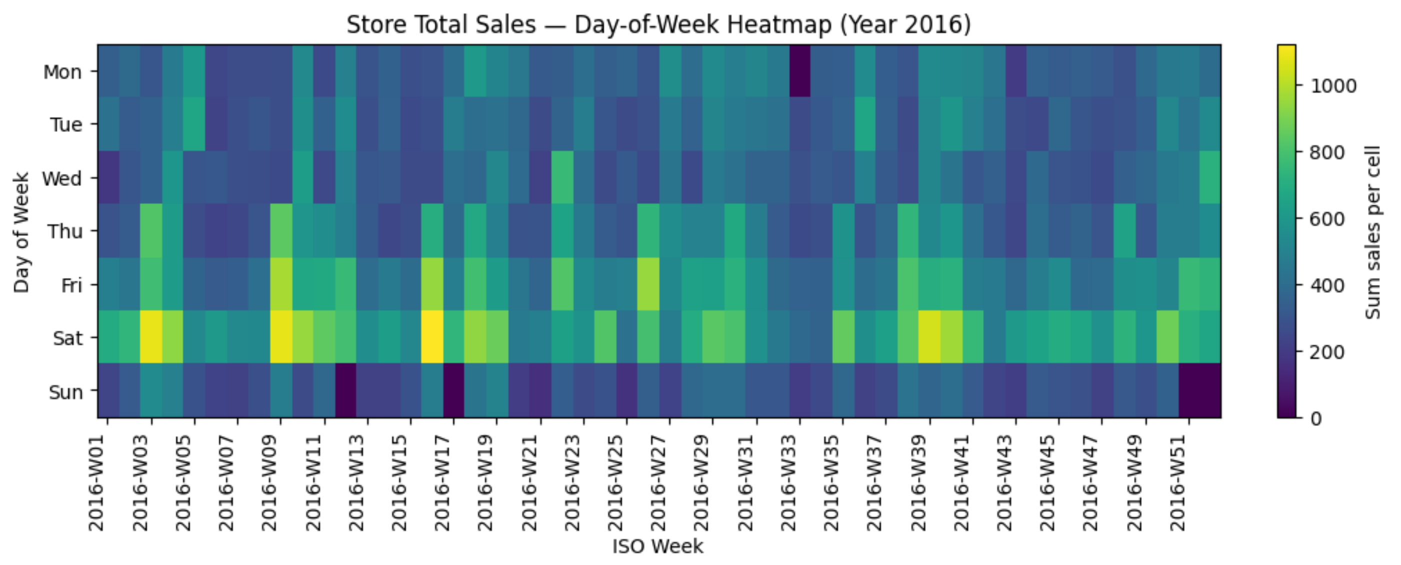

Daily Sales for Total_QTY_Store
Total_QTY_Store
Units Sold
Date
0
200
400
600
800
1000
1200
1400
2014
2015
2016
2017
2018
2019

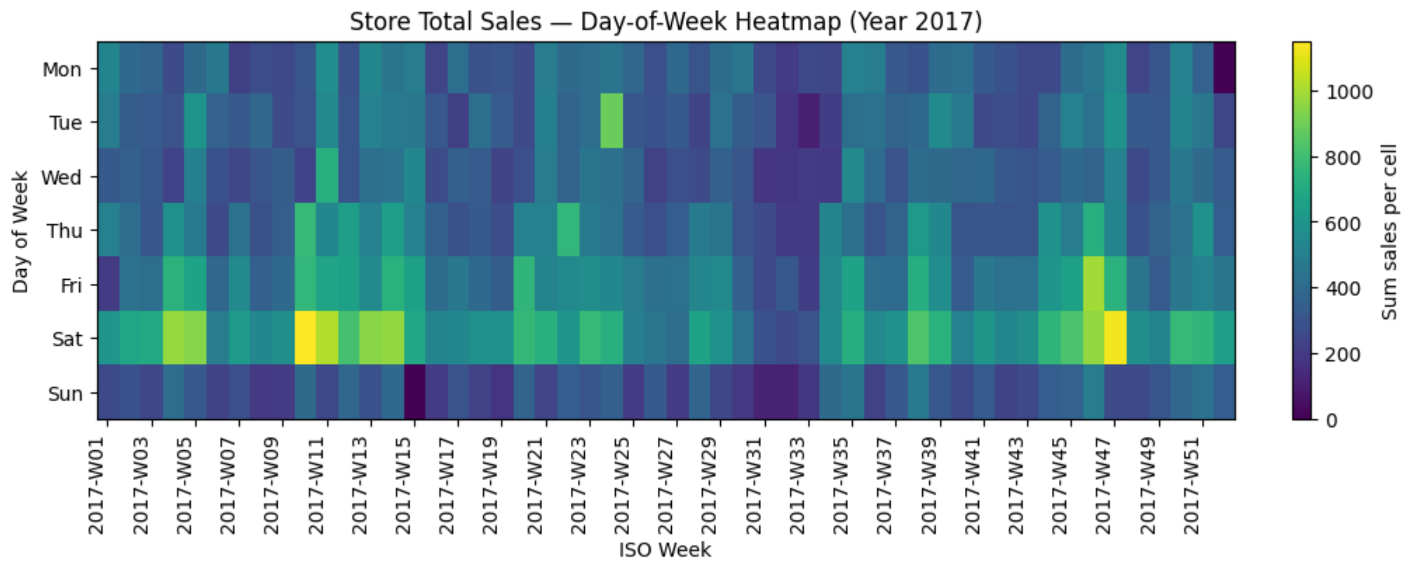
### 2.2.2 Heatmap, Sales Across Weeks

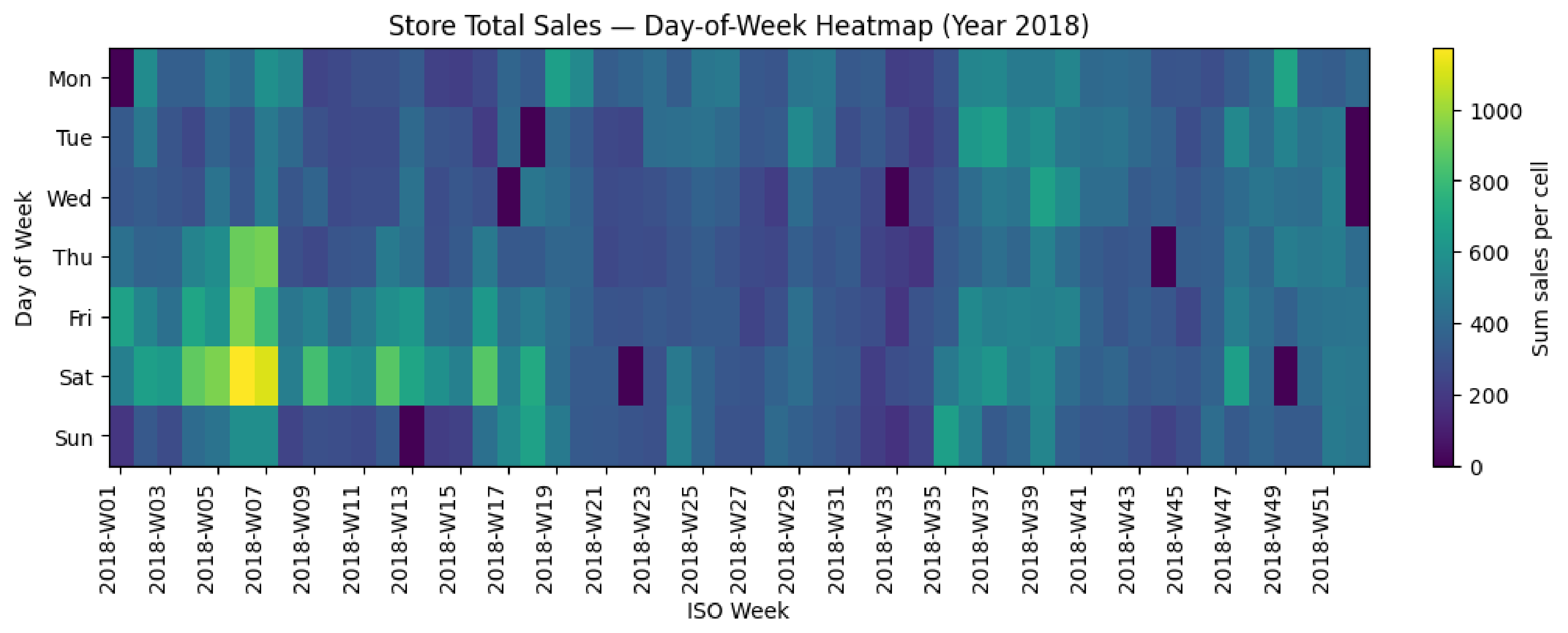
The heatmaps represent the total sale each day of the week for each year. It helps identify recurring sales patterns in week days. The weeks were numbered using ISO system which starts number of each week from the first Thursday of that year.






Store Total Sales — Day-of-Week Heatmap (Year 2016)
Mon
Tue
Wed
Thu
Fri
Sat
Sun
Day of Week
2016-W01
2016-W03
2016-W05
2016-W07
2016-W09
2016-W11
2016-W13
2016-W15
2016-W17
2016-W19
2016-W21
2016-W23
2016-W25
2016-W27
2016-W29
2016-W31
2016-W33
2016-W35
2016-W37
2016-W39
2016-W41
2016-W43
2016-W45
2016-W47
2016-W49
2016-W51
ISO Week
0
200
400
600
800
1000
Sum sales per cell

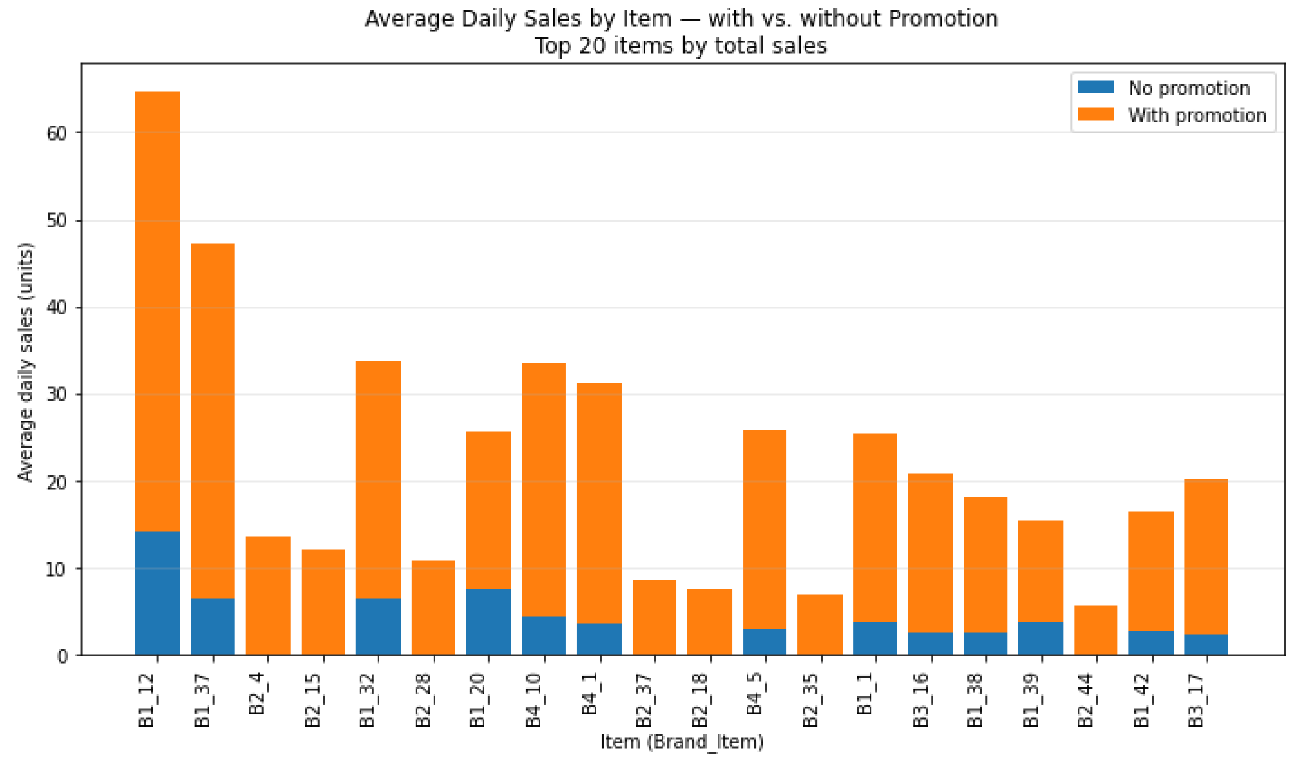

Store Total Sales — Day-of-Week Heatmap (Year 2017)
Mon
Tue
Wed
Thu
Fri
Sat
Sun
Day of Week
2017-W01
2017-W03
2017-W05
2017-W07
2017-W09
2017-W11
2017-W13
2017-W15
2017-W17
2017-W19
2017-W21
2017-W23
2017-W25
2017-W27
2017-W29
2017-W31
2017-W33
2017-W35
2017-W37
2017-W39
2017-W41
2017-W43
2017-W45
2017-W47
2017-W49
2017-W51
ISO Week
0
200
400
600
800
1000
Sum sales per cell



The heatmaps clearly indicate that there are high sales in most Saturdays and some Fridays. Some portions of the heatmap is blank, indicating the dataset does not have sales data for some days which might be holidays when the store was closed.

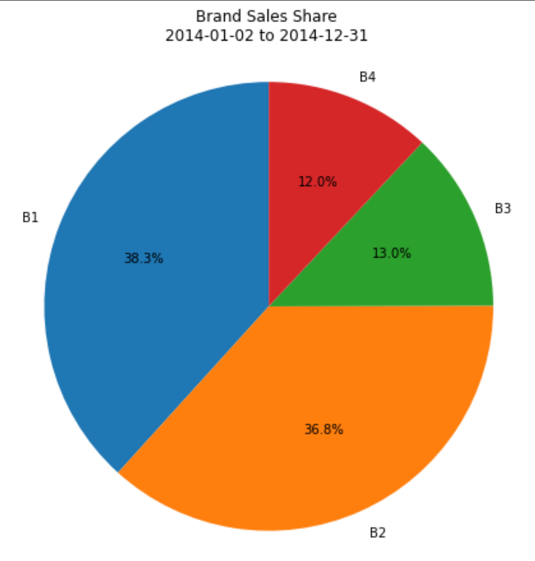
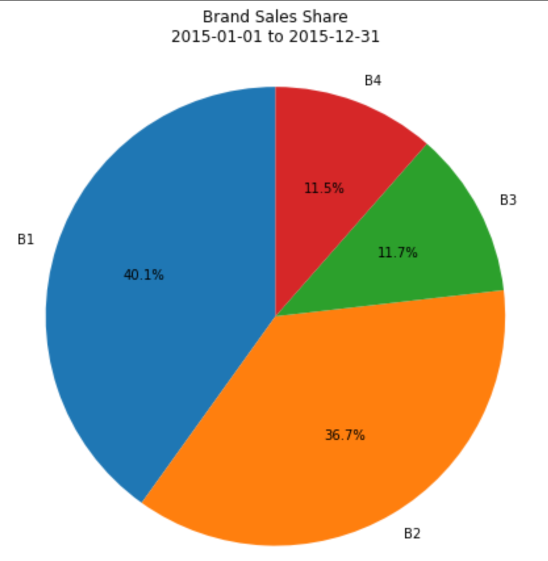
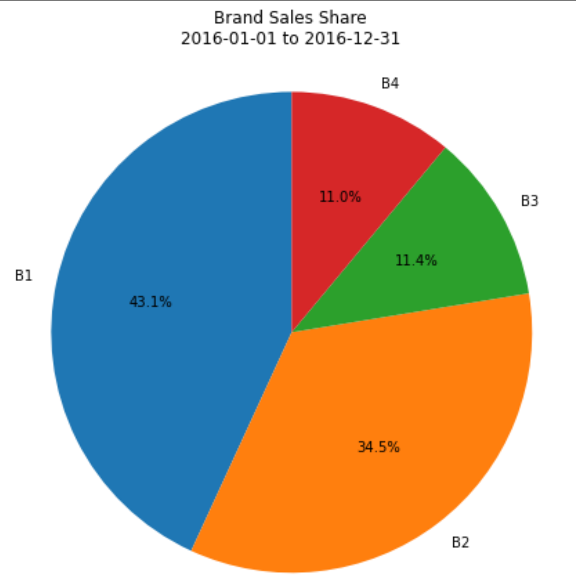
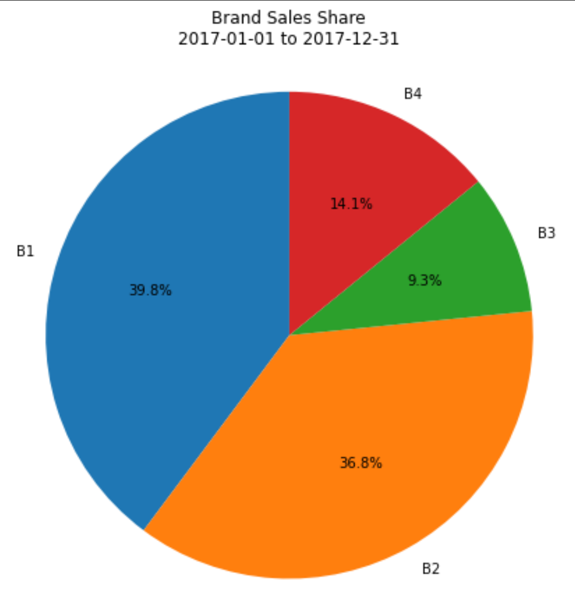
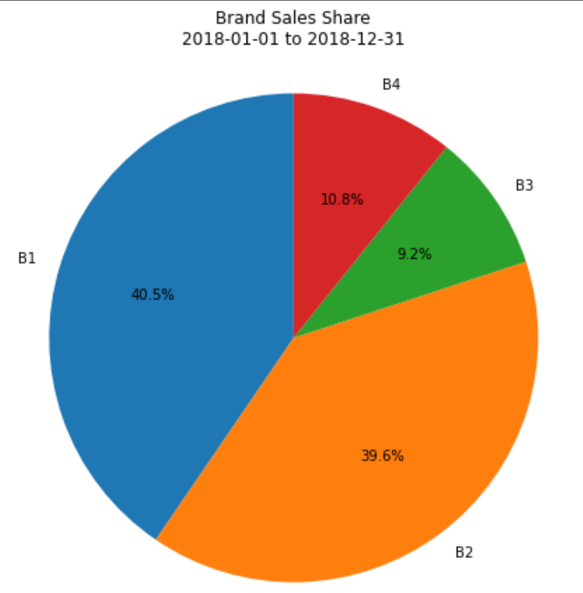
### 2.2.3 Impact of Promotion

The plot shows the impact of promotion on top 20 items sold. The average daily sales were compared for this plot for each item. The top 20 displayed were selected on the basis of total sales in all 5 years, but the chart y-axis values are of average daily sales of each item.



### 2.2.4 Brand Sales Share by Year:

The pie charts describe the brand sales share in the store for each year

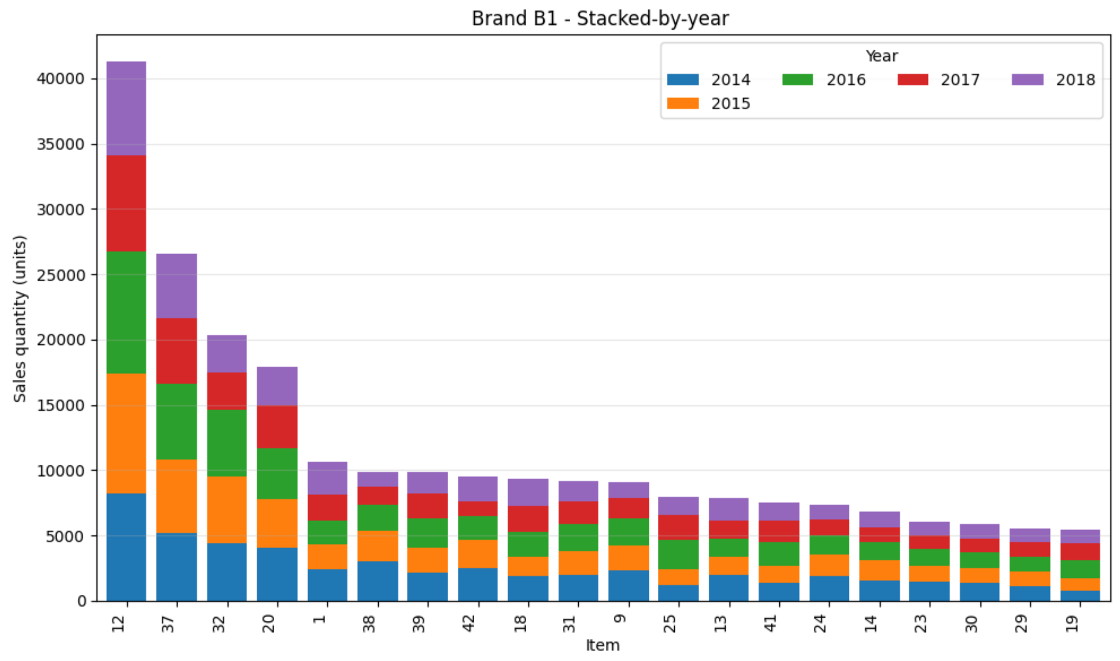
| Year 2014 | Year 2015 | Year 2016 | Year 2017 | Year 2018 |
|---|---|---|---|---|
|  |  |  |  |  |

All the pie charts show that brand B1 has the major share in sales followed by brand B2. The sales shares are pretty consistent throughout the years.

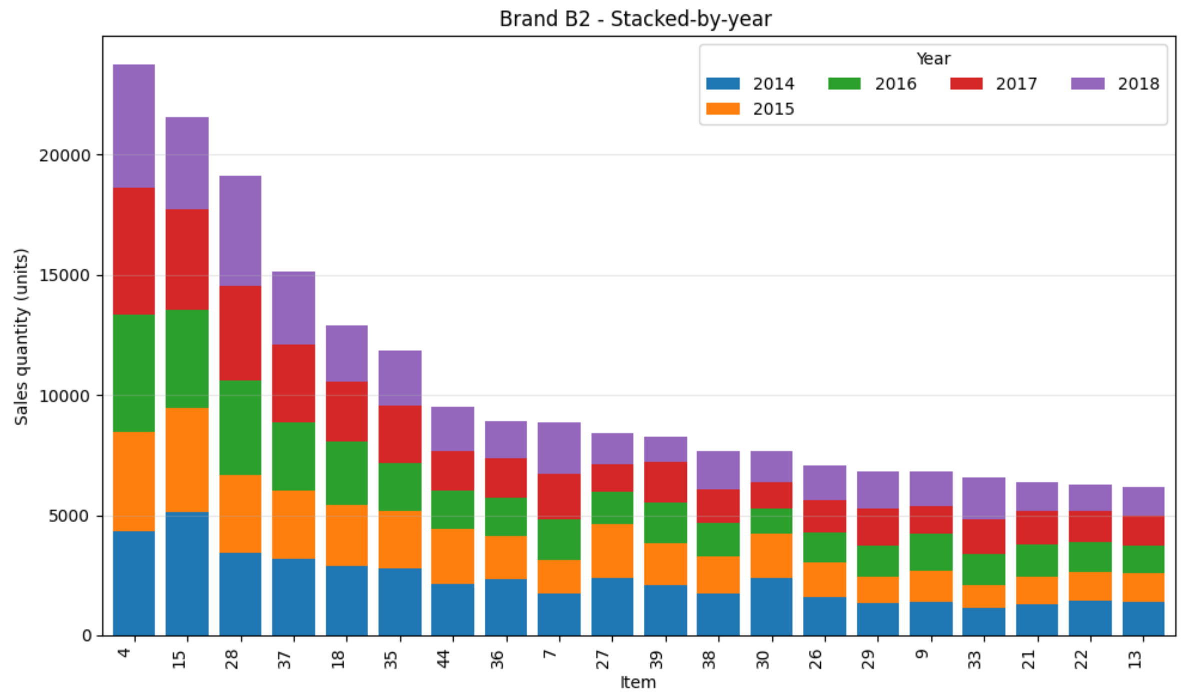
### 2.2.5 Item Sales Share for each brand

Here are some plots showing the top 20 items sold from each brand, and stacked by year.

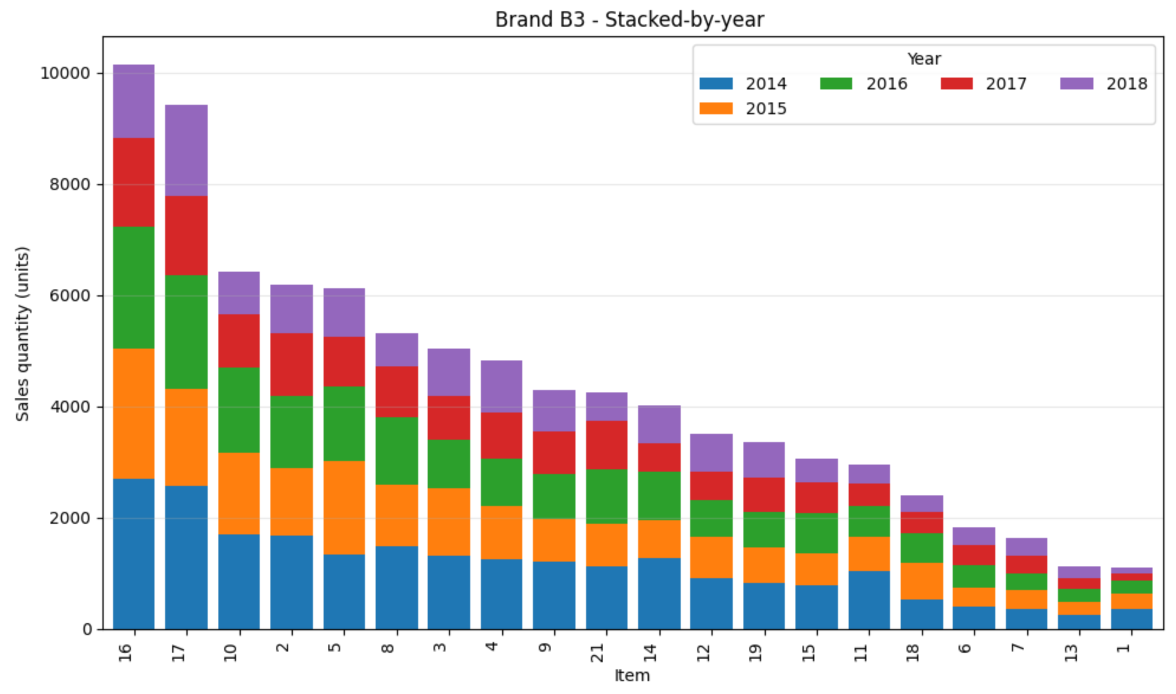
…



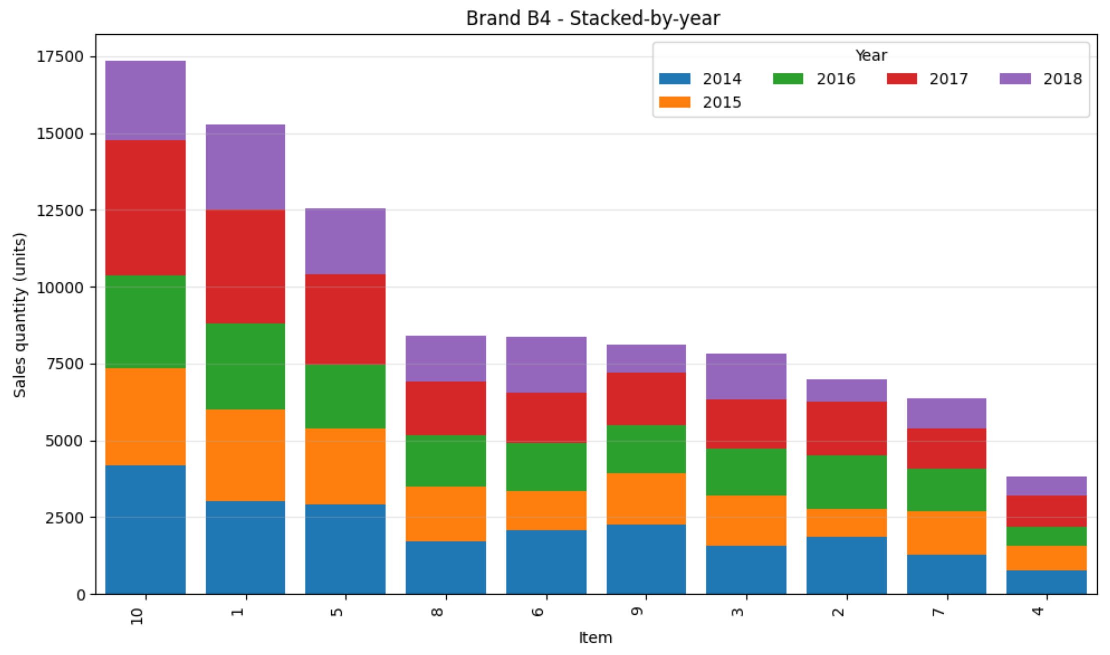
…



…



…



The plots obviously point out that some items of each brand have far greater amount in sales than others

## 2.3 Data Pre-processing & Cleaning:

The following steps of Data Cleaning and Pre-processing were done

### 2.3.1 Cleaning Data

1. Convert Time column to date-time
2. Clearing out missing values (store closed)
3. Filling those missing values with 0

### 2.3.2 Data Pre-process

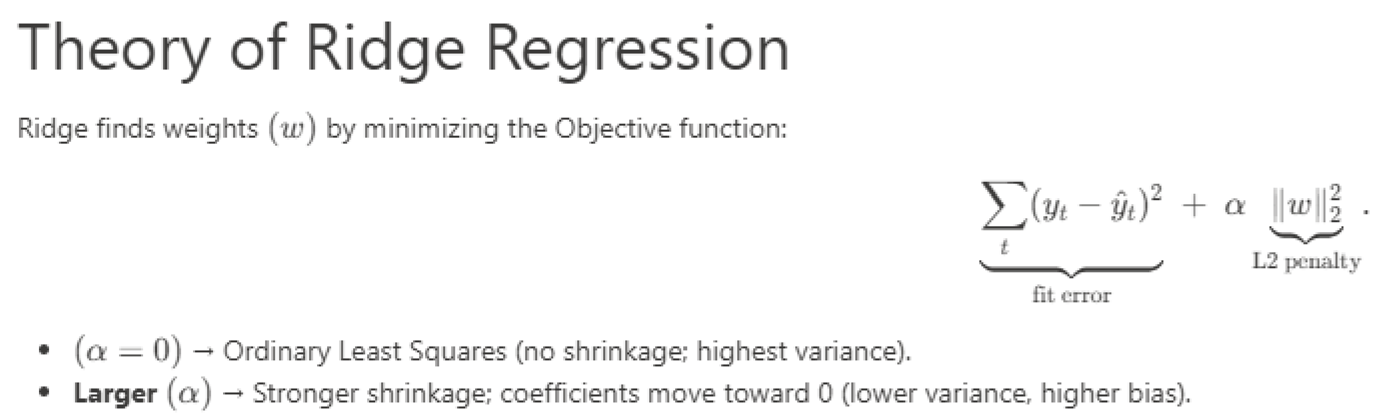
1. Parsing to separate the brands and items
2. Defining functions to find sum between dates
3. (Augmenting the dataset) Adding new columns for brand level sales, total store sales
4. Scaling: Normalization on the data

## 2.4 ML Model 1:

The first ML model we trained was a simple **Ridge Regression Algorithm.** The theory of ridge regression is described in the following section.

To keep initial training simple, we applied this ML algorithm **only to Brand B1 sales** and **not in any other time series** of the dataset. Hence this ML model was trained without any relation to the other time series of this dataset, treating Brand B1 sales as an independent univariate time series data.

### 2.4.1 Ridge Regression: Theory



Theory of Ridge Regression

Ridge finds weights $(w)$ by minimizing the Objective function:

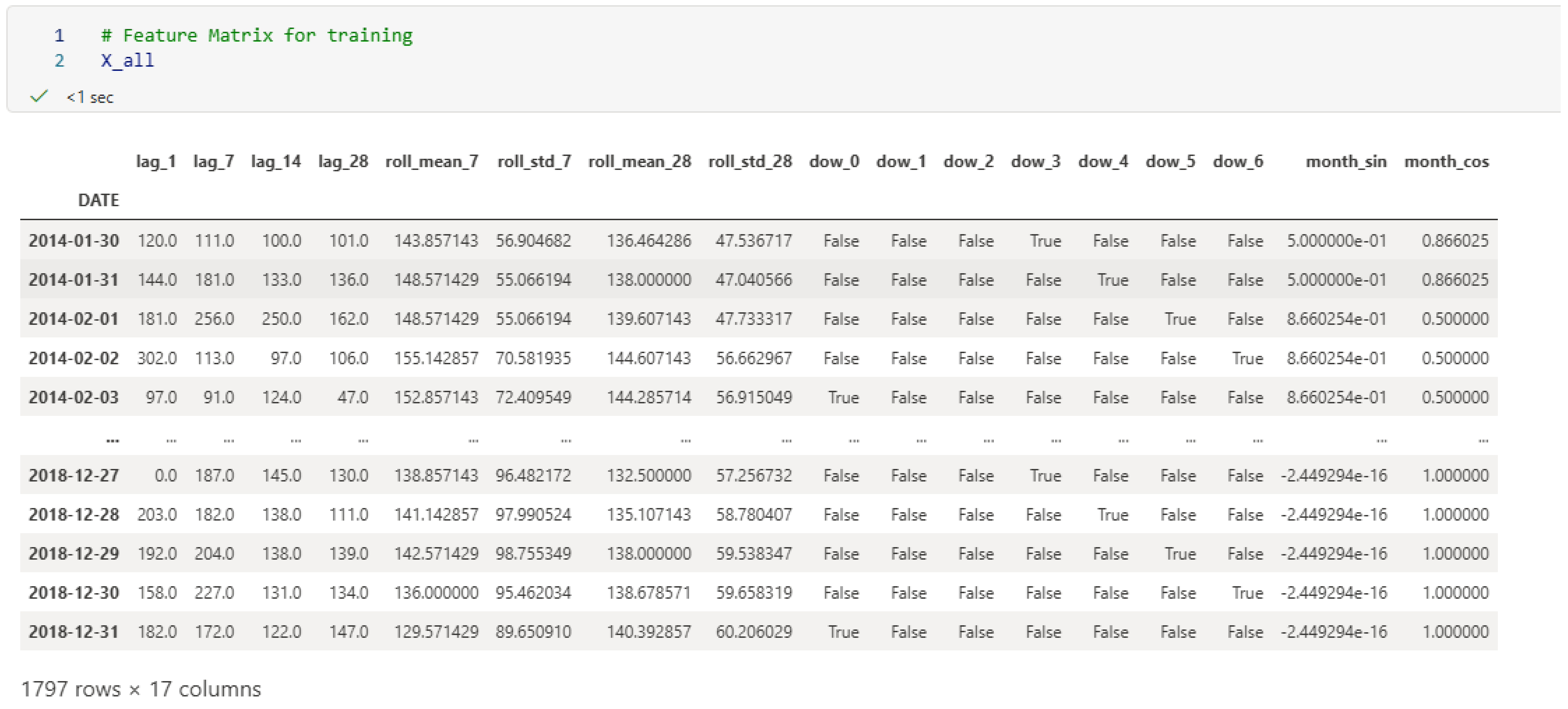
$$\underbrace{\sum_t (y_t - \hat{y}_t)^2}_{\text{fit error}} + \alpha \underbrace{\|w\|_2^2}_{\text{L2 penalty}} .$$

- $(\alpha = 0)$ → Ordinary Least Squares (no shrinkage; highest variance).
- **Larger** $(\alpha)$ → Stronger shrinkage; coefficients move toward 0 (lower variance, higher bias).

Note: the above picture is taken from the ipython notebook.

Here the y_t term is the actual series, and hat(y_t) is the predicted time series. Alpha is a tuning parameter that we tune later

### 2.4.2 Feature Engineering



```
# Feature Matrix for training
X_all
```

✓ <1 sec

| | lag_1 | lag_7 | lag_14 | lag_28 | roll_mean_7 | roll_std_7 | roll_mean_28 | roll_std_28 | dow_0 | dow_1 | dow_2 | dow_3 | dow_4 | dow_5 | dow_6 | month_sin | month_cos |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **DATE** | | | | | | | | | | | | | | | | | |
| **2014-01-30** | 120.0 | 111.0 | 100.0 | 101.0 | 143.857143 | 56.904682 | 136.464286 | 47.536717 | False | False | False | True | False | False | False | 5.000000e-01 | 0.866025 |
| **2014-01-31** | 144.0 | 181.0 | 133.0 | 136.0 | 148.571429 | 55.066194 | 138.000000 | 47.040566 | False | False | False | False | True | False | False | 5.000000e-01 | 0.866025 |
| **2014-02-01** | 181.0 | 256.0 | 250.0 | 162.0 | 148.571429 | 55.066194 | 139.607143 | 47.733317 | False | False | False | False | False | True | False | 8.660254e-01 | 0.500000 |
| **2014-02-02** | 302.0 | 113.0 | 97.0 | 106.0 | 155.142857 | 70.581935 | 144.607143 | 56.662967 | False | False | False | False | False | False | True | 8.660254e-01 | 0.500000 |
| **2014-02-03** | 97.0 | 91.0 | 124.0 | 47.0 | 152.857143 | 72.409549 | 144.285714 | 56.915049 | True | False | False | False | False | False | False | 8.660254e-01 | 0.500000 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **2018-12-27** | 0.0 | 187.0 | 145.0 | 130.0 | 138.857143 | 96.482172 | 132.500000 | 57.256732 | False | False | False | True | False | False | False | -2.449294e-16 | 1.000000 |
| **2018-12-28** | 203.0 | 182.0 | 138.0 | 111.0 | 141.142857 | 97.990524 | 135.107143 | 58.780407 | False | False | False | False | True | False | False | -2.449294e-16 | 1.000000 |
| **2018-12-29** | 192.0 | 204.0 | 138.0 | 139.0 | 142.571429 | 98.755349 | 138.000000 | 59.538347 | False | False | False | False | False | True | False | -2.449294e-16 | 1.000000 |
| **2018-12-30** | 158.0 | 227.0 | 131.0 | 134.0 | 136.000000 | 95.462034 | 138.678571 | 59.658319 | False | False | False | False | False | False | True | -2.449294e-16 | 1.000000 |
| **2018-12-31** | 182.0 | 172.0 | 122.0 | 147.0 | 129.571429 | 89.650910 | 140.392857 | 60.206029 | True | False | False | False | False | False | False | -2.449294e-16 | 1.000000 |

1797 rows × 17 columns

*Figure 2-4 Feature Matrix for Ridge Regression*

The above picture shows the feature matrix for the input to the ML algorithm for ridge regression. Weights will be multiplied to all the features of this feature matrix, and then the optimal weights will be found during training of the ridge regression algorithm.

#### 2.4.2.1.1 Lag Features:

The Figure 2-4 terms lag_1, lag_7, lag_14, lag_28, each are respectively the sales values of 1 day before, 7 days before, 14 days before, and 28 days before the selected date in the row. Hence the number of rows has also reduced by 28 from 1825 to 1797 as seen in the above picture. This is to ensure that lag does not cause going behind to a value outside of the time range of the dataset. Hence a first few values have been omitted.

These lag features help look at the recent history of those days before and feed those features into the learning algorithm.

#### 2.4.2.1.2 Rolling Statistics

Similarly, the **Rolling statistics** roll_mean_7, roll_std_7, roll_mean_28, roll_std_28, are the mean and standard deviation for a rolling window of 7 days and 28 days behind the current point. This window statistics is also fed to the ridge regression algorithm for learning.

#### 2.4.2.1.3 Days of Week (Dow) Encoding

Days of the week (DOW) has been encoded in a **one-hot-encoding** format as shown from the Figure 2-4 Feature Matrix for Ridge Regression

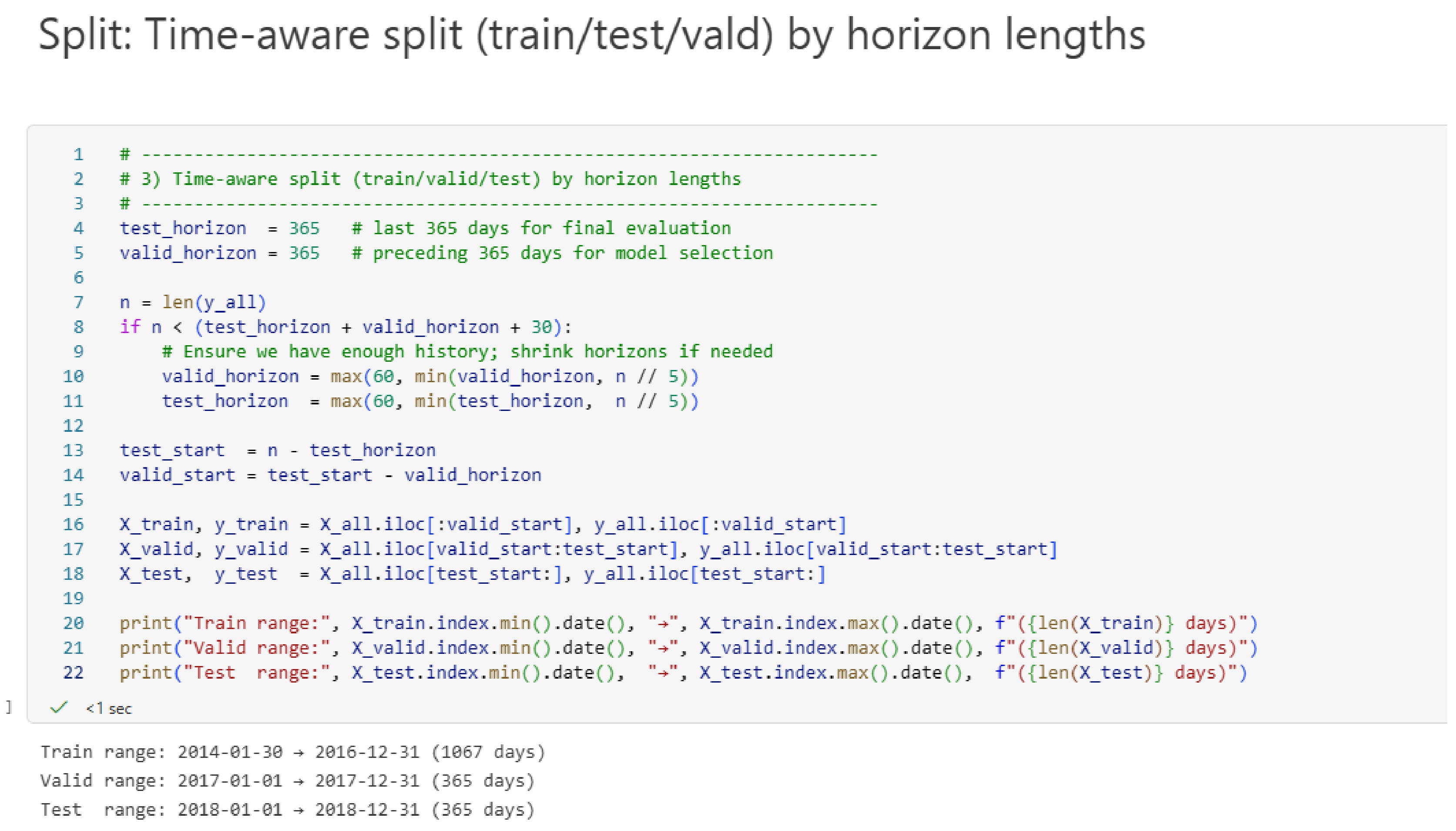
#### 2.4.2.1.4 Months of Year, cyclic encoding

Months of the year has been encoded in **cyclic encoding** to keep January, close to December. The reason this was done was because of the seasonal patterns similarity existing in January, December (cold months) vs June, July (hot months).

Whereas one-hot encoding was done for DOW because there might be sudden changes in customer behavior from Sunday to Monday, hence not ideal for cyclic encoding.

### 2.4.3 Train-Test Split

Next we did a train-test split in the dataset according to the following figure



Split: Time-aware split (train/test/vald) by horizon lengths

```python
# -----------------------------------------------------------------------
# 3) Time-aware split (train/valid/test) by horizon lengths
# -----------------------------------------------------------------------
test_horizon  = 365   # last 365 days for final evaluation
valid_horizon = 365   # preceding 365 days for model selection

n = len(y_all)
if n < (test_horizon + valid_horizon + 30):
    # Ensure we have enough history; shrink horizons if needed
    valid_horizon = max(60, min(valid_horizon, n // 5))
    test_horizon  = max(60, min(test_horizon,  n // 5))

test_start  = n - test_horizon
valid_start = test_start - valid_horizon

X_train, y_train = X_all.iloc[:valid_start], y_all.iloc[:valid_start]
X_valid, y_valid = X_all.iloc[valid_start:test_start], y_all.iloc[valid_start:test_start]
X_test,  y_test  = X_all.iloc[test_start:], y_all.iloc[test_start:]

print("Train range:", X_train.index.min().date(), "→", X_train.index.max().date(), f"({len(X_train)} days)")
print("Valid range:", X_valid.index.min().date(), "→", X_valid.index.max().date(), f"({len(X_valid)} days)")
print("Test  range:", X_test.index.min().date(),  "→", X_test.index.max().date(),  f"({len(X_test)} days)")
```

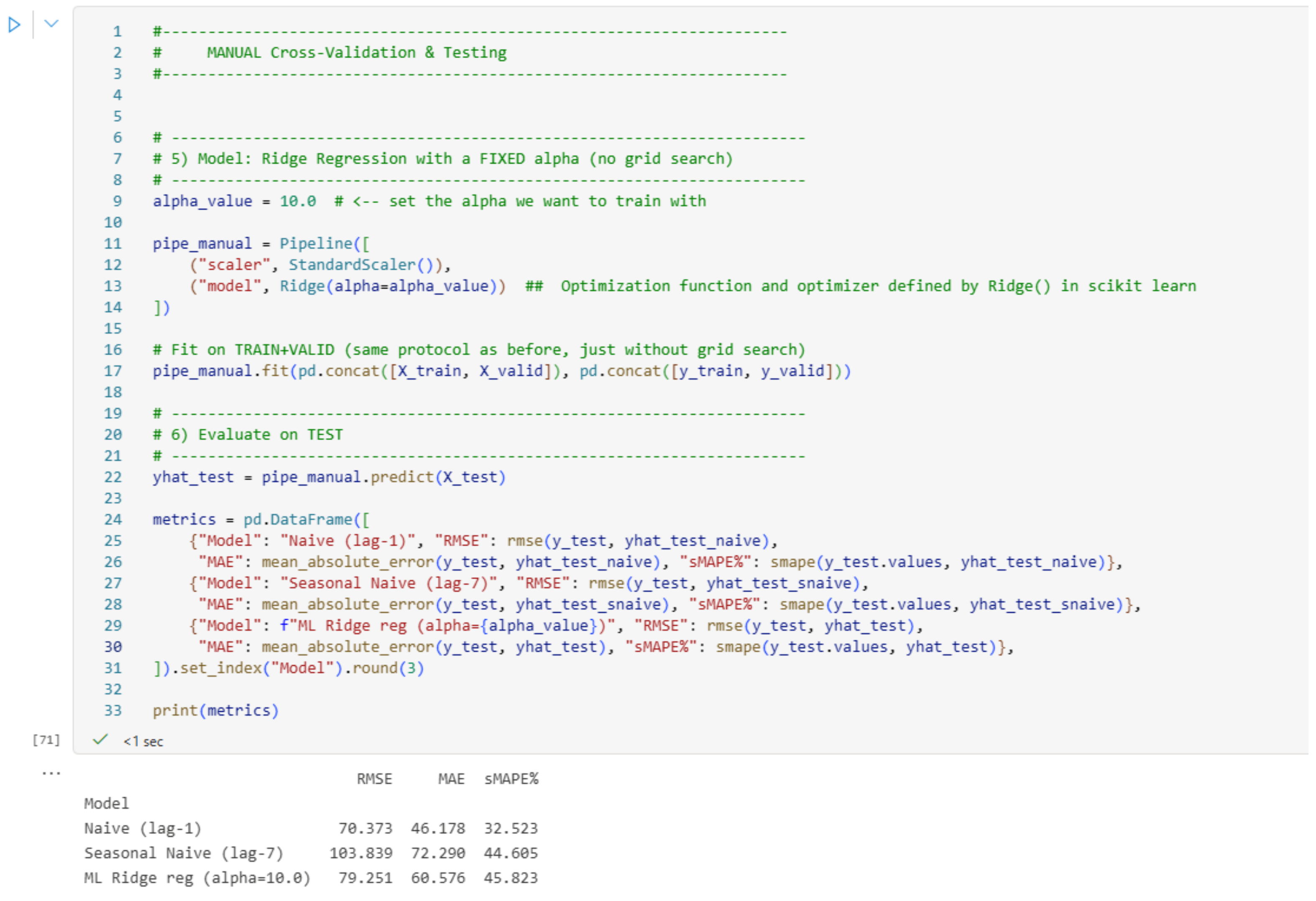
✓ <1 sec

```
Train range: 2014-01-30 → 2016-12-31 (1067 days)
Valid range: 2017-01-01 → 2017-12-31 (365 days)
Test  range: 2018-01-01 → 2018-12-31 (365 days)
```

*Figure 2-5Train-test split by horizon length*

We secified the test-horizon to be 365 day, and the validation horizon to be 365 days. That is it will take the last 365 + 365 days for test and validation and will use the remaining days to train the ridge regression model, seen from Figure 2-5

### 2.4.4 Training & Cross-Validation


```
#-------------------------------------------------------------------
#     MANUAL Cross-Validation & Testing
#-------------------------------------------------------------------


# ---------------------------------------------------------------------
# 5) Model: Ridge Regression with a FIXED alpha (no grid search)
# ---------------------------------------------------------------------
alpha_value = 10.0  # <-- set the alpha we want to train with

pipe_manual = Pipeline([
    ("scaler", StandardScaler()),
    ("model", Ridge(alpha=alpha_value))  ##  Optimization function and optimizer defined by Ridge() in scikit learn
])

# Fit on TRAIN+VALID (same protocol as before, just without grid search)
pipe_manual.fit(pd.concat([X_train, X_valid]), pd.concat([y_train, y_valid]))

# ---------------------------------------------------------------------
# 6) Evaluate on TEST
# ---------------------------------------------------------------------
yhat_test = pipe_manual.predict(X_test)

metrics = pd.DataFrame([
    {"Model": "Naive (lag-1)", "RMSE": rmse(y_test, yhat_test_naive),
     "MAE": mean_absolute_error(y_test, yhat_test_naive), "sMAPE%": smape(y_test.values, yhat_test_naive)},
    {"Model": "Seasonal Naive (lag-7)", "RMSE": rmse(y_test, yhat_test_snaive),
     "MAE": mean_absolute_error(y_test, yhat_test_snaive), "sMAPE%": smape(y_test.values, yhat_test_snaive)},
    {"Model": f"ML Ridge reg (alpha={alpha_value})", "RMSE": rmse(y_test, yhat_test),
     "MAE": mean_absolute_error(y_test, yhat_test), "sMAPE%": smape(y_test.values, yhat_test)},
]).set_index("Model").round(3)

print(metrics)
```
[71] ✓ <1 sec

```
                           RMSE     MAE  sMAPE%
Model
Naive (lag-1)            70.373  46.178  32.523
Seasonal Naive (lag-7)  103.839  72.290  44.605
ML Ridge reg (alpha=10.0)  79.251  60.576  45.823
```

*Figure 2-6Training Fit to Ridge Regression*

The above Figure 2-6 shows the training of the ridge regression model. The **ridge regression implementation from scikit-learn** is used in this training. This implementation uses an auto solver that uses the Cholesky decomposition to solve for optimal weights.

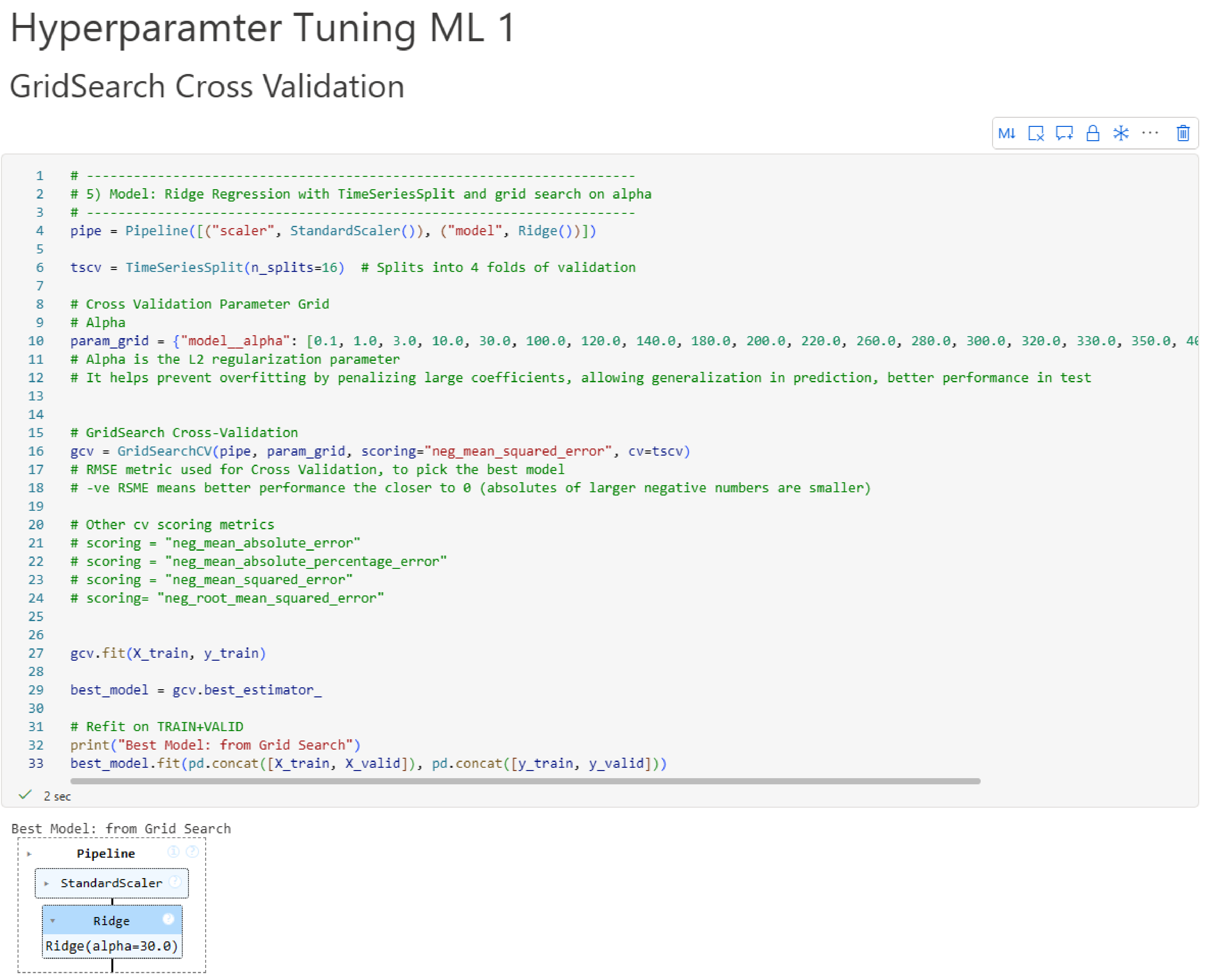
After training, the results are shown in Figure 2-6.

Results used are Root Mean Squared (RMSE), Mean Absolute Error (MAE), Symmetric Mean Absolute Percentage Error (sMAPE)

The ML Ridge regression model is compared with some other prediction methods, the **Naïve**, and **Seasonal Naïve** . Naïve and Seasonal Naïve does no calculations at all, Naïve just draws the lag_1 value (the immediate preceeding value) from the feature matrix, and Seasonal Naïve draws out the lag_7 value (the value 7 days behind this value) from the feature matrix. They are just used as markers to see how the ML model compares to these easy predictors.

### 2.4.5 Hyperparameter Tuning

Hyperparameter tuning was later done on this model using grid search.



Hyperparamter Tuning ML 1

GridSearch Cross Validation

```python
# ----------------------------------------------------------------------
# 5) Model: Ridge Regression with TimeSeriesSplit and grid search on alpha
# ----------------------------------------------------------------------
pipe = Pipeline([("scaler", StandardScaler()), ("model", Ridge())])

tscv = TimeSeriesSplit(n_splits=16)  # Splits into 4 folds of validation

# Cross Validation Parameter Grid
# Alpha
param_grid = {"model__alpha": [0.1, 1.0, 3.0, 10.0, 30.0, 100.0, 120.0, 140.0, 180.0, 200.0, 220.0, 260.0, 280.0, 300.0, 320.0, 330.0, 350.0, 40
# Alpha is the L2 regularization parameter
# It helps prevent overfitting by penalizing large coefficients, allowing generalization in prediction, better performance in test


# GridSearch Cross-Validation
gcv = GridSearchCV(pipe, param_grid, scoring="neg_mean_squared_error", cv=tscv)
# RMSE metric used for Cross Validation, to pick the best model
# -ve RSME means better performance the closer to 0 (absolutes of larger negative numbers are smaller)

# Other cv scoring metrics
# scoring = "neg_mean_absolute_error"
# scoring = "neg_mean_absolute_percentage_error"
# scoring = "neg_mean_squared_error"
# scoring= "neg_root_mean_squared_error"


gcv.fit(X_train, y_train)

best_model = gcv.best_estimator_

# Refit on TRAIN+VALID
print("Best Model: from Grid Search")
best_model.fit(pd.concat([X_train, X_valid]), pd.concat([y_train, y_valid]))
```

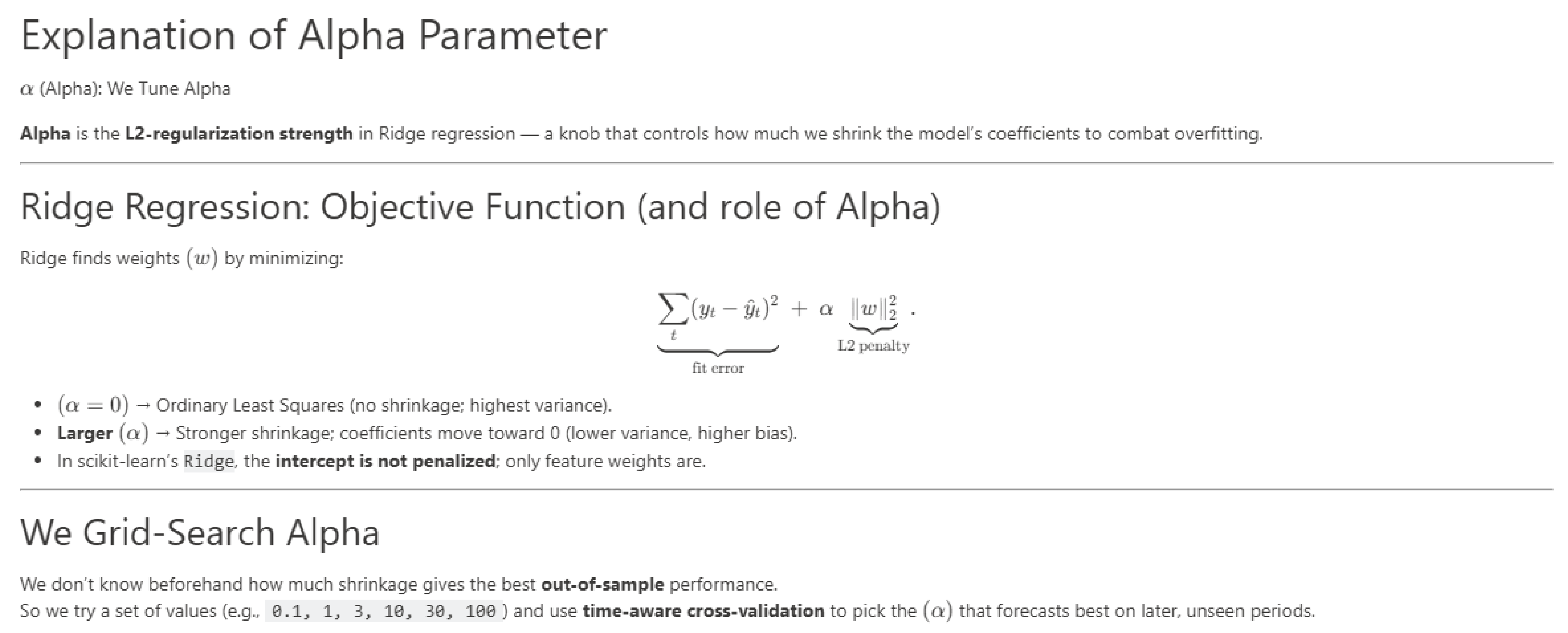
3] ✓ 2 sec

```
Best Model: from Grid Search
Pipeline
  StandardScaler
  Ridge
  Ridge(alpha=30.0)
```

*Figure 2-7 Hyperparameter tuning for ML Ridge Model*

The hyperparameter tuning was done on a grid search. The hyperparamter tuned here was the \alpha parameter.

The alpha was the L2 regularization parameter shown in Figure 2-8



## Explanation of Alpha Parameter

$\alpha$ (Alpha): We Tune Alpha

**Alpha** is the **L2-regularization strength** in Ridge regression — a knob that controls how much we shrink the model's coefficients to combat overfitting.

## Ridge Regression: Objective Function (and role of Alpha)

Ridge finds weights $(w)$ by minimizing:

$$\underbrace{\sum_t (y_t - \hat{y}_t)^2}_{\text{fit error}} + \alpha \underbrace{\|w\|_2^2}_{\text{L2 penalty}} .$$

- $(\alpha = 0)$ → Ordinary Least Squares (no shrinkage; highest variance).
- **Larger** $(\alpha)$ → Stronger shrinkage; coefficients move toward 0 (lower variance, higher bias).
- In scikit-learn's `Ridge`, the **intercept is not penalized**; only feature weights are.

## We Grid-Search Alpha

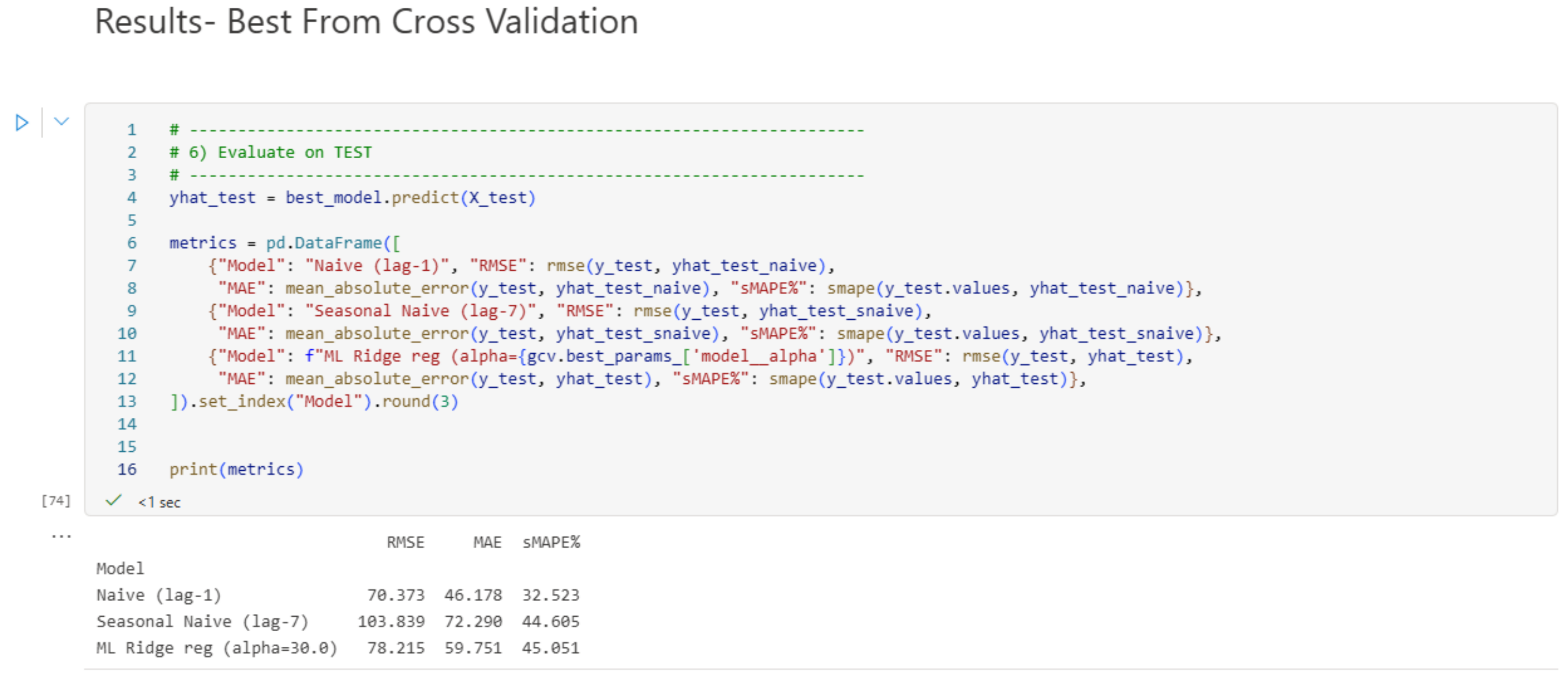
We don't know beforehand how much shrinkage gives the best **out-of-sample** performance.
So we try a set of values (e.g., `0.1, 1, 3, 10, 30, 100`) and use **time-aware cross-validation** to pick the $(\alpha)$ that forecasts best on later, unseen periods.

*Figure 2-8 Ridge Regression Alpha Paramter Tuning*

Higher values of alpha allows for generalization.

After implementing grid search we found the best model had alpha = 30.0 as shown in Figure 2-7

### 2.4.6 Results from Best model for ML model 1



Results- Best From Cross Validation

```python
# ----------------------------------------------------------------------
# 6) Evaluate on TEST
# ----------------------------------------------------------------------
yhat_test = best_model.predict(X_test)

metrics = pd.DataFrame([
    {"Model": "Naive (lag-1)", "RMSE": rmse(y_test, yhat_test_naive),
     "MAE": mean_absolute_error(y_test, yhat_test_naive), "sMAPE%": smape(y_test.values, yhat_test_naive)},
    {"Model": "Seasonal Naive (lag-7)", "RMSE": rmse(y_test, yhat_test_snaive),
     "MAE": mean_absolute_error(y_test, yhat_test_snaive), "sMAPE%": smape(y_test.values, yhat_test_snaive)},
    {"Model": f"ML Ridge reg (alpha={gcv.best_params_['model__alpha']})", "RMSE": rmse(y_test, yhat_test),
     "MAE": mean_absolute_error(y_test, yhat_test), "sMAPE%": smape(y_test.values, yhat_test)},
]).set_index("Model").round(3)


print(metrics)
```

[74] ✓ <1 sec

```
                           RMSE     MAE  sMAPE%
Model
Naive (lag-1)            70.373  46.178  32.523
Seasonal Naive (lag-7)  103.839  72.290  44.605
ML Ridge reg (alpha=30.0) 78.215  59.751  45.051
```

*Figure 2-9 Best Results from Cross Validation of ML model 1*

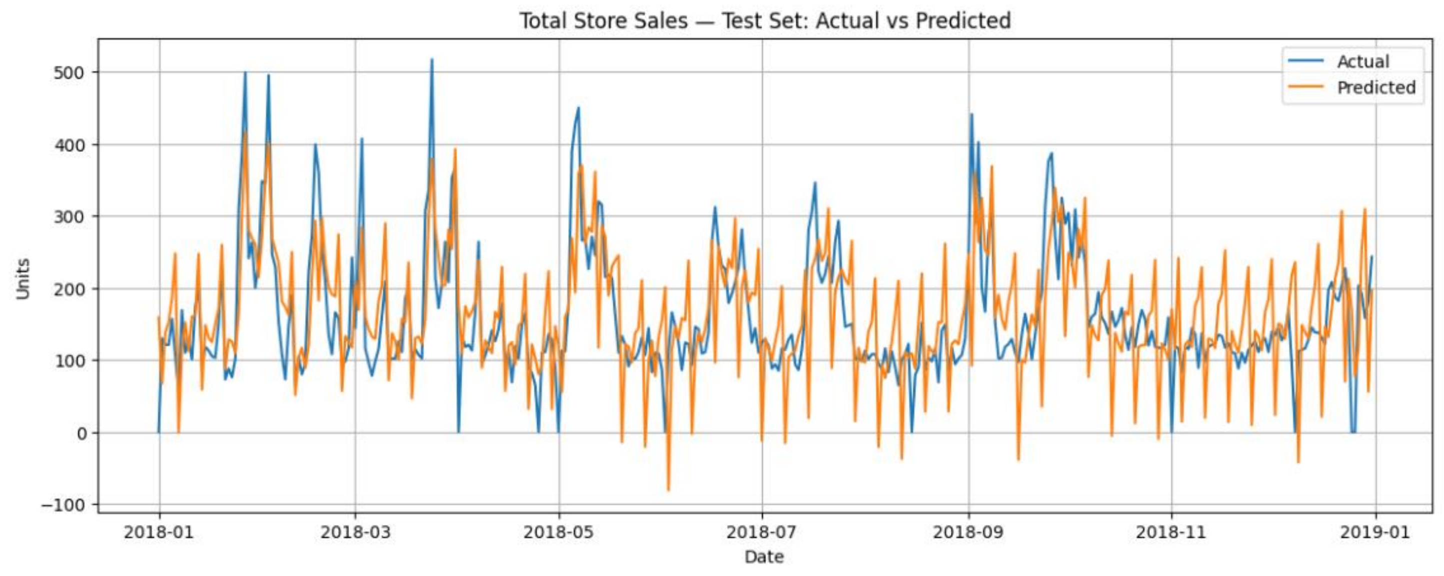


*Figure 2-10 Plot of B1 Brands total store sales, actual and predicted*

The forecast results for the trained ridge model is shown in Figure 2-10

## 2.5 ML Model 2:

### 2.5.1 Architecture for Hierarchy Exploitation:

This ML Model no. 2 exploits the hierarchical relationship between brand and its items. Hence the ML model will train together the time series for the brand and time series for every item in the brand as shown in Figure 2-1

This ML model fully uses the modelling plan described in section Modelling Plan:

The Time series hierarchy and Modelling plan is also presented below.

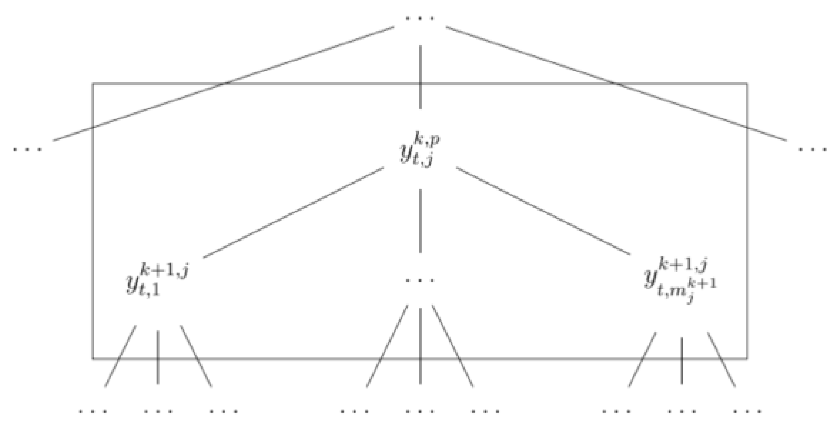


*Figure 2-11Time Series Hierarchy followed for ML Model 2*

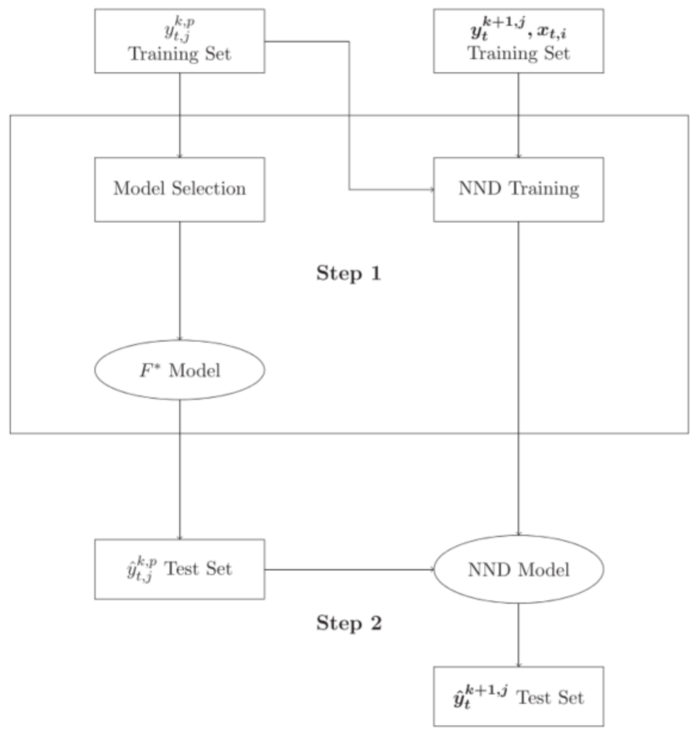

$y_{t,j}^{k,p}$ Training Set
$\boldsymbol{y}_t^{k+1,j}, \boldsymbol{x}_{t,i}$ Training Set
Model Selection
NND Training
Step 1
$F^*$ Model
$\hat{y}_{t,j}^{k,p}$ Test Set
NND Model
Step 2
$\hat{\boldsymbol{y}}_t^{k+1,j}$ Test Set

*Figure 2-12 Neural Network Disaggregation (NND) architecture followed in paper [2]*

The Neural Network Dissaggregation (NND) architecture shown in Figure 2-12 and adapted from [2] is implemented in this ML Model 2.

But one difference from this model described in [2] and my implementation is that **I only used hierarchy of brand level to item level**. The **final hierarchy of total store level sales was not used** to keep the code simple.

## 2.5.2 Feature Engineering

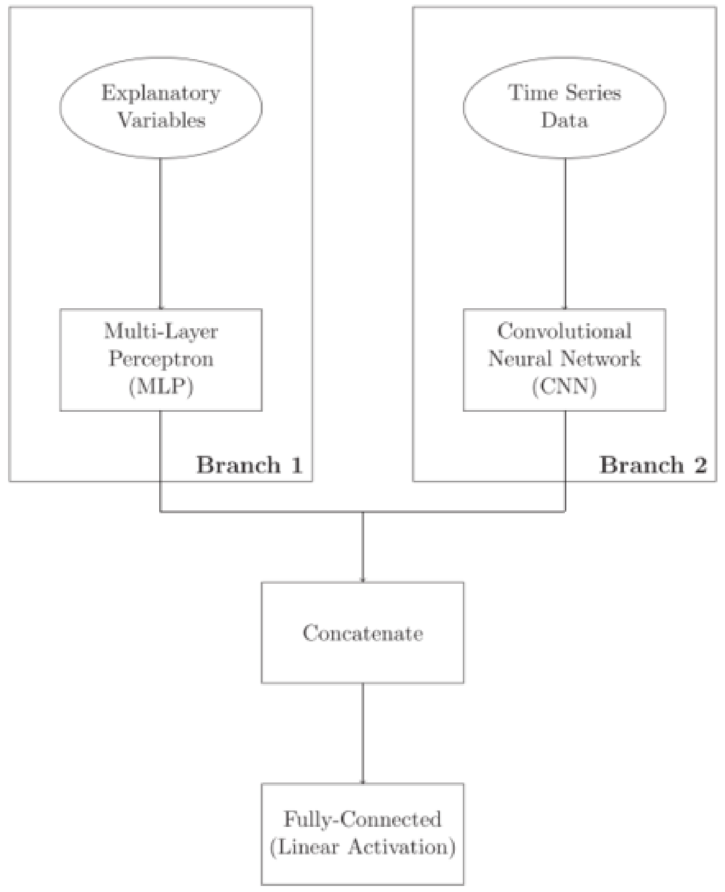

Explanatory Variables
Time Series Data
Multi-Layer Perceptron (MLP)
Convolutional Neural Network (CNN)
Branch 1
Branch 2
Concatenate
Fully-Connected (Linear Activation)

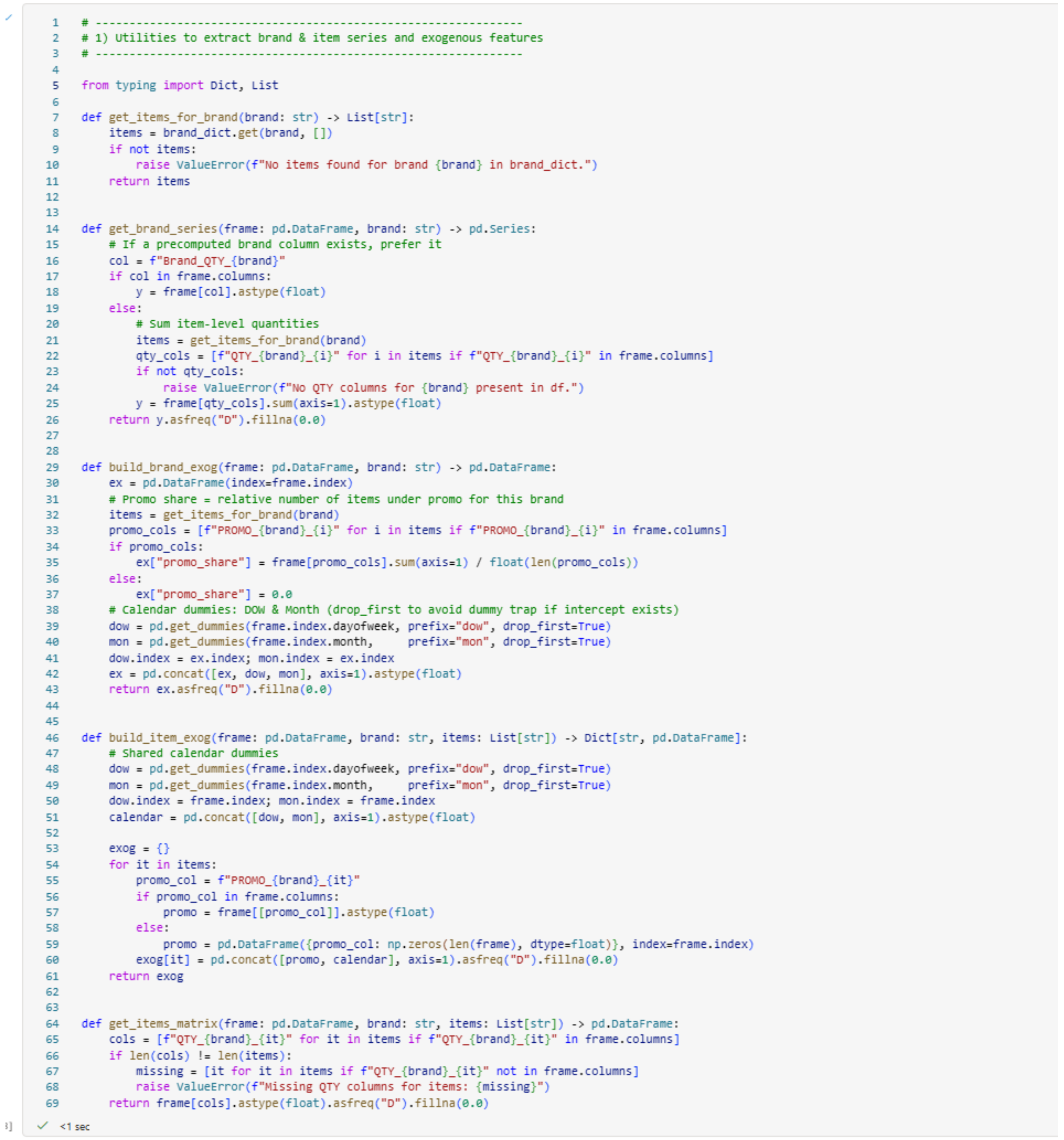
*Figure 2-13 Input Variable Concatenation plan for ML Model 2. Adapted from [2]*

For this NND architecture to work and train on both the top level time series and its contributing children time series, we have to concatenate the training data (previous historical data of the time series) and the **exogeneous / explanatory variables** (in this case, the presence or absence of promotion)

These exogeneous variables were extracted from every brand and every item using the code shown in Figure 2-14

Meanwhile Figure 2-2 shows the original dataset which contains the exogenous variables for each brand. They were separated using the code from Figure 2-14

Exogenous Variable Extraction for Brand

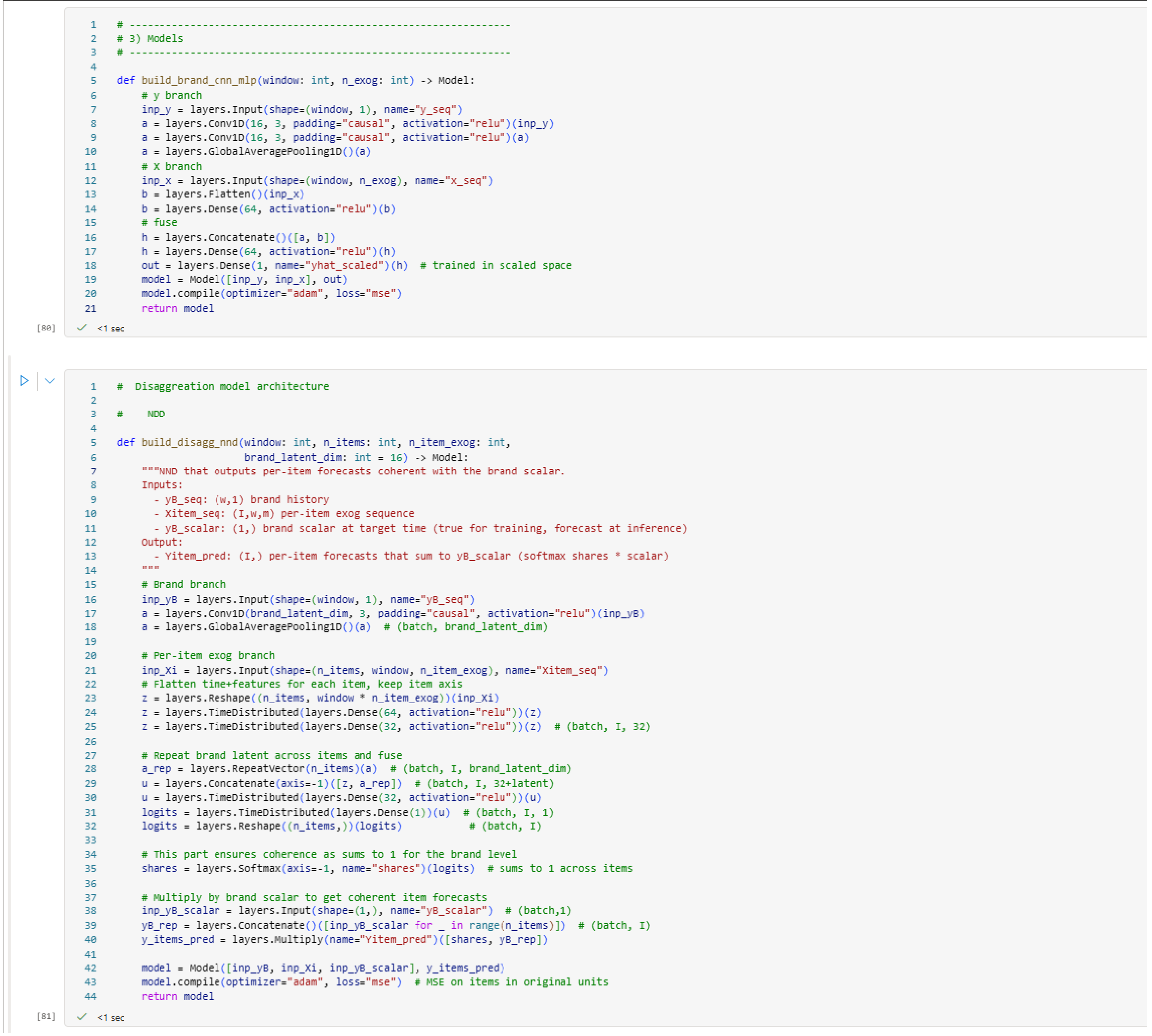


```python
# ---------------------------------------------------------------
# 1) Utilities to extract brand & item series and exogenous features
# ---------------------------------------------------------------

from typing import Dict, List

def get_items_for_brand(brand: str) -> List[str]:
    items = brand_dict.get(brand, [])
    if not items:
        raise ValueError(f"No items found for brand {brand} in brand_dict.")
    return items


def get_brand_series(frame: pd.DataFrame, brand: str) -> pd.Series:
    # If a precomputed brand column exists, prefer it
    col = f"Brand_QTY_{brand}"
    if col in frame.columns:
        y = frame[col].astype(float)
    else:
        # Sum item-level quantities
        items = get_items_for_brand(brand)
        qty_cols = [f"QTY_{brand}_{i}" for i in items if f"QTY_{brand}_{i}" in frame.columns]
        if not qty_cols:
            raise ValueError(f"No QTY columns for {brand} present in df.")
        y = frame[qty_cols].sum(axis=1).astype(float)
    return y.asfreq("D").fillna(0.0)


def build_brand_exog(frame: pd.DataFrame, brand: str) -> pd.DataFrame:
    ex = pd.DataFrame(index=frame.index)
    # Promo share = relative number of items under promo for this brand
    items = get_items_for_brand(brand)
    promo_cols = [f"PROMO_{brand}_{i}" for i in items if f"PROMO_{brand}_{i}" in frame.columns]
    if promo_cols:
        ex["promo_share"] = frame[promo_cols].sum(axis=1) / float(len(promo_cols))
    else:
        ex["promo_share"] = 0.0
    # Calendar dummies: DOW & Month (drop_first to avoid dummy trap if intercept exists)
    dow = pd.get_dummies(frame.index.dayofweek, prefix="dow", drop_first=True)
    mon = pd.get_dummies(frame.index.month,     prefix="mon", drop_first=True)
    dow.index = ex.index; mon.index = ex.index
    ex = pd.concat([ex, dow, mon], axis=1).astype(float)
    return ex.asfreq("D").fillna(0.0)


def build_item_exog(frame: pd.DataFrame, brand: str, items: List[str]) -> Dict[str, pd.DataFrame]:
    # Shared calendar dummies
    dow = pd.get_dummies(frame.index.dayofweek, prefix="dow", drop_first=True)
    mon = pd.get_dummies(frame.index.month,     prefix="mon", drop_first=True)
    dow.index = frame.index; mon.index = frame.index
    calendar = pd.concat([dow, mon], axis=1).astype(float)

    exog = {}
    for it in items:
        promo_col = f"PROMO_{brand}_{it}"
        if promo_col in frame.columns:
            promo = frame[[promo_col]].astype(float)
        else:
            promo = pd.DataFrame({promo_col: np.zeros(len(frame), dtype=float)}, index=frame.index)
        exog[it] = pd.concat([promo, calendar], axis=1).asfreq("D").fillna(0.0)
    return exog


def get_items_matrix(frame: pd.DataFrame, brand: str, items: List[str]) -> pd.DataFrame:
    cols = [f"QTY_{brand}_{it}" for it in items if f"QTY_{brand}_{it}" in frame.columns]
    if len(cols) != len(items):
        missing = [it for it in items if f"QTY_{brand}_{it}" not in frame.columns]
        raise ValueError(f"Missing QTY columns for items: {missing}")
    return frame[cols].astype(float).asfreq("D").fillna(0.0)
```

✓ <1 sec

*Figure 2-14 Code for Exogenous variable extraction of both items and brands*

### 2.5.3 Code for ML Architecture



```python
# ---------------------------------------------------------------
# 3) Models
# ---------------------------------------------------------------

def build_brand_cnn_mlp(window: int, n_exog: int) -> Model:
    # y branch
    inp_y = layers.Input(shape=(window, 1), name="y_seq")
    a = layers.Conv1D(16, 3, padding="causal", activation="relu")(inp_y)
    a = layers.Conv1D(16, 3, padding="causal", activation="relu")(a)
    a = layers.GlobalAveragePooling1D()(a)
    # X branch
    inp_x = layers.Input(shape=(window, n_exog), name="x_seq")
    b = layers.Flatten()(inp_x)
    b = layers.Dense(64, activation="relu")(b)
    # fuse
    h = layers.Concatenate()([a, b])
    h = layers.Dense(64, activation="relu")(h)
    out = layers.Dense(1, name="yhat_scaled")(h)  # trained in scaled space
    model = Model([inp_y, inp_x], out)
    model.compile(optimizer="adam", loss="mse")
    return model
```

[80] ✓ <1 sec

```python
#  Disaggreation model architecture

#    NDD

def build_disagg_nnd(window: int, n_items: int, n_item_exog: int,
                     brand_latent_dim: int = 16) -> Model:
    """NND that outputs per-item forecasts coherent with the brand scalar.
    Inputs:
      - yB_seq: (w,1) brand history
      - Xitem_seq: (I,w,m) per-item exog sequence
      - yB_scalar: (1,) brand scalar at target time (true for training, forecast at inference)
    Output:
      - Yitem_pred: (I,) per-item forecasts that sum to yB_scalar (softmax shares * scalar)
    """
    # Brand branch
    inp_yB = layers.Input(shape=(window, 1), name="yB_seq")
    a = layers.Conv1D(brand_latent_dim, 3, padding="causal", activation="relu")(inp_yB)
    a = layers.GlobalAveragePooling1D()(a)  # (batch, brand_latent_dim)

    # Per-item exog branch
    inp_Xi = layers.Input(shape=(n_items, window, n_item_exog), name="Xitem_seq")
    # Flatten time+features for each item, keep item axis
    z = layers.Reshape((n_items, window * n_item_exog))(inp_Xi)
    z = layers.TimeDistributed(layers.Dense(64, activation="relu"))(z)
    z = layers.TimeDistributed(layers.Dense(32, activation="relu"))(z)  # (batch, I, 32)

    # Repeat brand latent across items and fuse
    a_rep = layers.RepeatVector(n_items)(a)  # (batch, I, brand_latent_dim)
    u = layers.Concatenate(axis=-1)([z, a_rep])  # (batch, I, 32+latent)
    u = layers.TimeDistributed(layers.Dense(32, activation="relu"))(u)
    logits = layers.TimeDistributed(layers.Dense(1))(u)  # (batch, I, 1)
    logits = layers.Reshape((n_items,))(logits)          # (batch, I)

    # This part ensures coherence as sums to 1 for the brand level
    shares = layers.Softmax(axis=-1, name="shares")(logits)  # sums to 1 across items

    # Multiply by brand scalar to get coherent item forecasts
    inp_yB_scalar = layers.Input(shape=(1,), name="yB_scalar")  # (batch,1)
    yB_rep = layers.Concatenate()([inp_yB_scalar for _ in range(n_items)])  # (batch, I)
    y_items_pred = layers.Multiply(name="Yitem_pred")([shares, yB_rep])

    model = Model([inp_yB, inp_Xi, inp_yB_scalar], y_items_pred)
    model.compile(optimizer="adam", loss="mse")  # MSE on items in original units
    return model
```

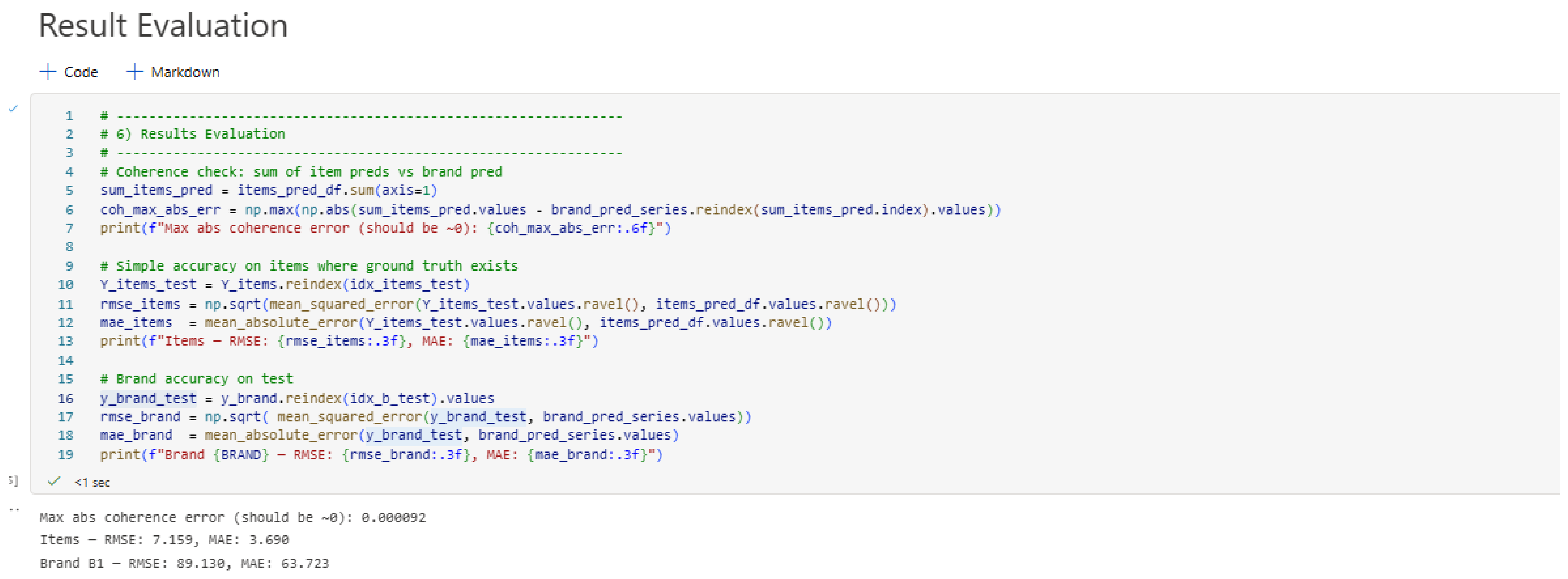
[81] ✓ <1 sec

*Figure 2-15 Code for NDD architecture*

The code for the NDD architecture is shown in Figure 2-15. Here, in the cell # 2, and line # 34, we can see the enforcement of the **coherence constraint** (ie, sum of all child time series must equal its parent time series predictions). This ensures that the model learns relations such that sum of all its child series will equal to the prediction in the parent series. This helps the model learn better.

## 2.6 Results of ML Model 2

After training, the following results were obtained for the ML model 2 using the NND.

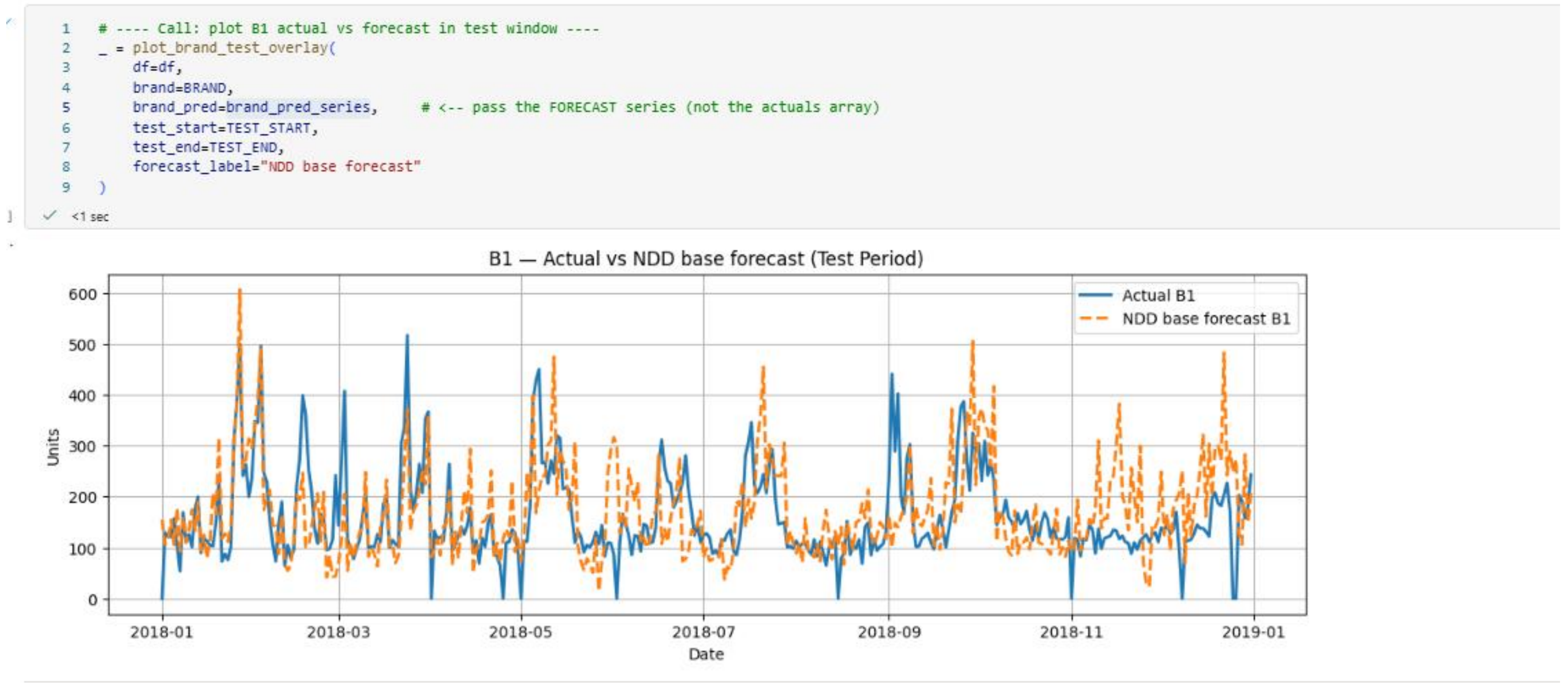


Result Evaluation

```python
# -------------------------------------------------------------
# 6) Results Evaluation
# -------------------------------------------------------------
# Coherence check: sum of item preds vs brand pred
sum_items_pred = items_pred_df.sum(axis=1)
coh_max_abs_err = np.max(np.abs(sum_items_pred.values - brand_pred_series.reindex(sum_items_pred.index).values))
print(f"Max abs coherence error (should be ~0): {coh_max_abs_err:.6f}")

# Simple accuracy on items where ground truth exists
Y_items_test = Y_items.reindex(idx_items_test)
rmse_items = np.sqrt(mean_squared_error(Y_items_test.values.ravel(), items_pred_df.values.ravel()))
mae_items  = mean_absolute_error(Y_items_test.values.ravel(), items_pred_df.values.ravel())
print(f"Items — RMSE: {rmse_items:.3f}, MAE: {mae_items:.3f}")

# Brand accuracy on test
y_brand_test = y_brand.reindex(idx_b_test).values
rmse_brand = np.sqrt( mean_squared_error(y_brand_test, brand_pred_series.values))
mae_brand  = mean_absolute_error(y_brand_test, brand_pred_series.values)
print(f"Brand {BRAND} — RMSE: {rmse_brand:.3f}, MAE: {mae_brand:.3f}")
```

```
Max abs coherence error (should be ~0): 0.000092
Items — RMSE: 7.159, MAE: 3.690
Brand B1 — RMSE: 89.130, MAE: 63.723
```

*Figure 2-16 Results, NND on B1 sales*



```
# ---- Call: plot B1 actual vs forecast in test window ----
_ = plot_brand_test_overlay(
    df=df,
    brand=BRAND,
    brand_pred=brand_pred_series,     # <-- pass the FORECAST series (not the actuals array)
    test_start=TEST_START,
    test_end=TEST_END,
    forecast_label="NDD base forecast"
)
```

*Figure 2-17 B1 Sales Plot*

The above figure show the forecast made by NND model after training. The forecast well matches the actual data. This is for B1 sales only.

## 2.7 Hyperparameter Tuning for ML Model 2 (NND)

Hyperparameter tuning was done on the B1 sales results to simplify the search.

Hyperparameter tuning was done on the following features:

1. Window size
2. Brand Latent Dimension

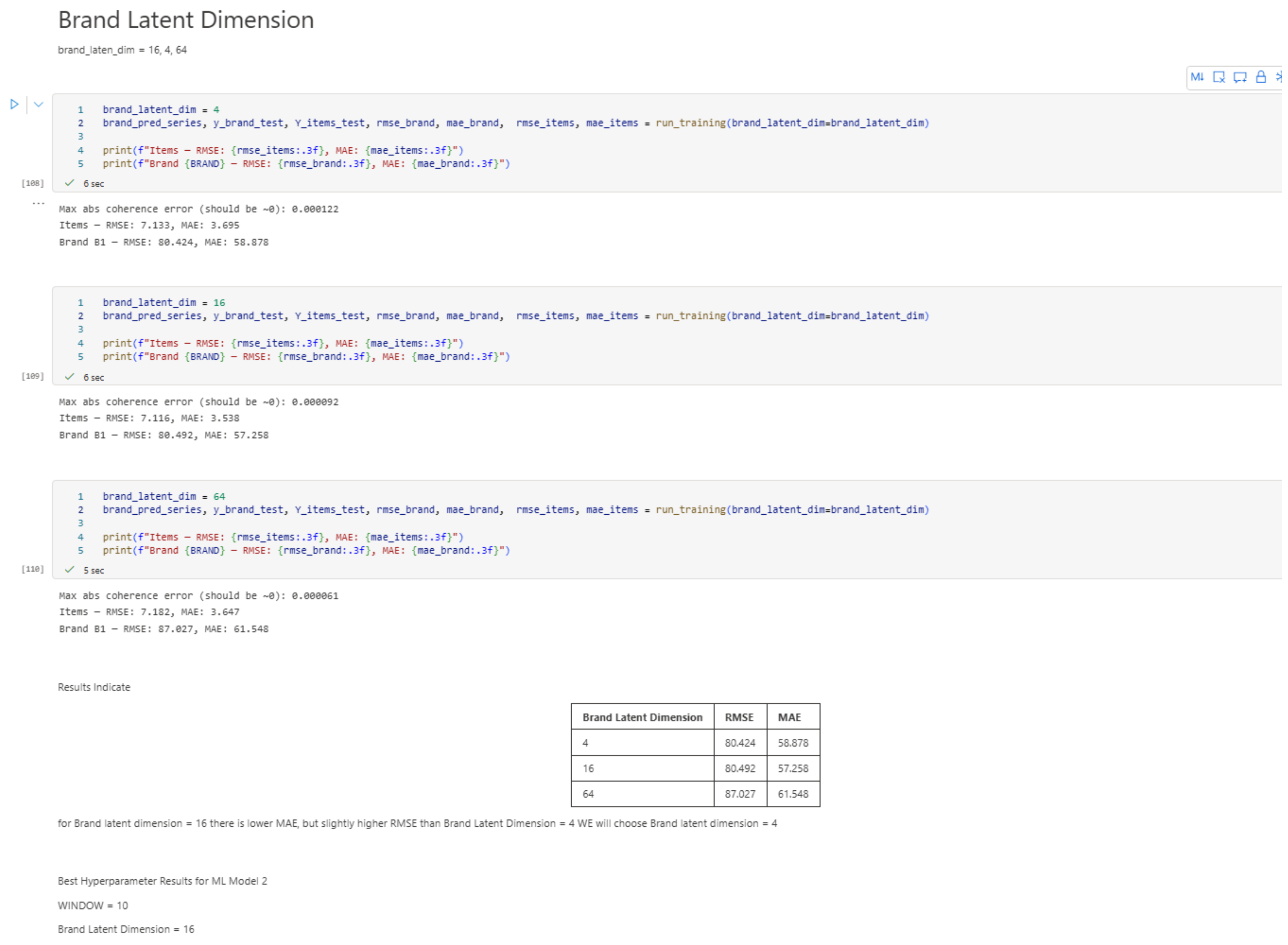
### 2.7.1 Window Size Tuning

Window size determines the window size to look for during the training process. Hence we tried three different window sizes, 10,  30, 60. Results are shown below.

| Window size | Root Mean Squared Error (RMSE) | Mean Absolute Error (MAE) |
|---|---|---|
| 10 | 85.252 | 61.74 |
| 30 | 88.382 | 61.856 |
| 60 | 93.346 | 68.156 |

Clearly Window =10 was the best performer. So we selected window = 10 for following calculations

### 2.7.2 Brand Latent Dimension Tuning



Brand Latent Dimension

brand_laten_dim = 16, 4, 64

```
brand_latent_dim = 4
brand_pred_series, y_brand_test, Y_items_test, rmse_brand, mae_brand,  rmse_items, mae_items = run_training(brand_latent_dim=brand_latent_dim)

print(f"Items – RMSE: {rmse_items:.3f}, MAE: {mae_items:.3f}")
print(f"Brand {BRAND} – RMSE: {rmse_brand:.3f}, MAE: {mae_brand:.3f}")
```

[108] 6 sec

```
Max abs coherence error (should be ~0): 0.000122
Items – RMSE: 7.133, MAE: 3.695
Brand B1 – RMSE: 80.424, MAE: 58.878
```

```
brand_latent_dim = 16
brand_pred_series, y_brand_test, Y_items_test, rmse_brand, mae_brand,  rmse_items, mae_items = run_training(brand_latent_dim=brand_latent_dim)

print(f"Items – RMSE: {rmse_items:.3f}, MAE: {mae_items:.3f}")
print(f"Brand {BRAND} – RMSE: {rmse_brand:.3f}, MAE: {mae_brand:.3f}")
```

[109] 6 sec

```
Max abs coherence error (should be ~0): 0.000092
Items – RMSE: 7.116, MAE: 3.538
Brand B1 – RMSE: 80.492, MAE: 57.258
```

```
brand_latent_dim = 64
brand_pred_series, y_brand_test, Y_items_test, rmse_brand, mae_brand,  rmse_items, mae_items = run_training(brand_latent_dim=brand_latent_dim)

print(f"Items – RMSE: {rmse_items:.3f}, MAE: {mae_items:.3f}")
print(f"Brand {BRAND} – RMSE: {rmse_brand:.3f}, MAE: {mae_brand:.3f}")
```

[110] 5 sec

```
Max abs coherence error (should be ~0): 0.000061
Items – RMSE: 7.182, MAE: 3.647
Brand B1 – RMSE: 87.027, MAE: 61.548
```

Results Indicate

| Brand Latent Dimension | RMSE | MAE |
|---|---|---|
| 4 | 80.424 | 58.878 |
| 16 | 80.492 | 57.258 |
| 64 | 87.027 | 61.548 |

for Brand latent dimension = 16 there is lower MAE, but slightly higher RMSE than Brand Latent Dimension = 4 WE will choose Brand latent dimension = 4

Best Hyperparameter Results for ML Model 2

WINDOW = 10

Brand Latent Dimension = 16

*Figure 2-18 Brand Latent Dimension Hyper-parameter search*

In the NND architecture the brand Latent Dimension is a the latent space where the model learns features relating to brand and item level time series relation. Hence changing the dimension of this latent space has the potential to change how the model learns. A higher latent space may mean model may learn more complex features but my fail to generalize

As seen from Figure 2-18 the latent dimension of 16 is the best choice

### 2.7.3 Best Hyperparamters

Best Hyperparameters

**Window = 10**

**Brand Latent Dimension = 16**

## 2.8 Results: All Brands

Using the best hyperparameter, the following results of forecast for all remaining brands are shown

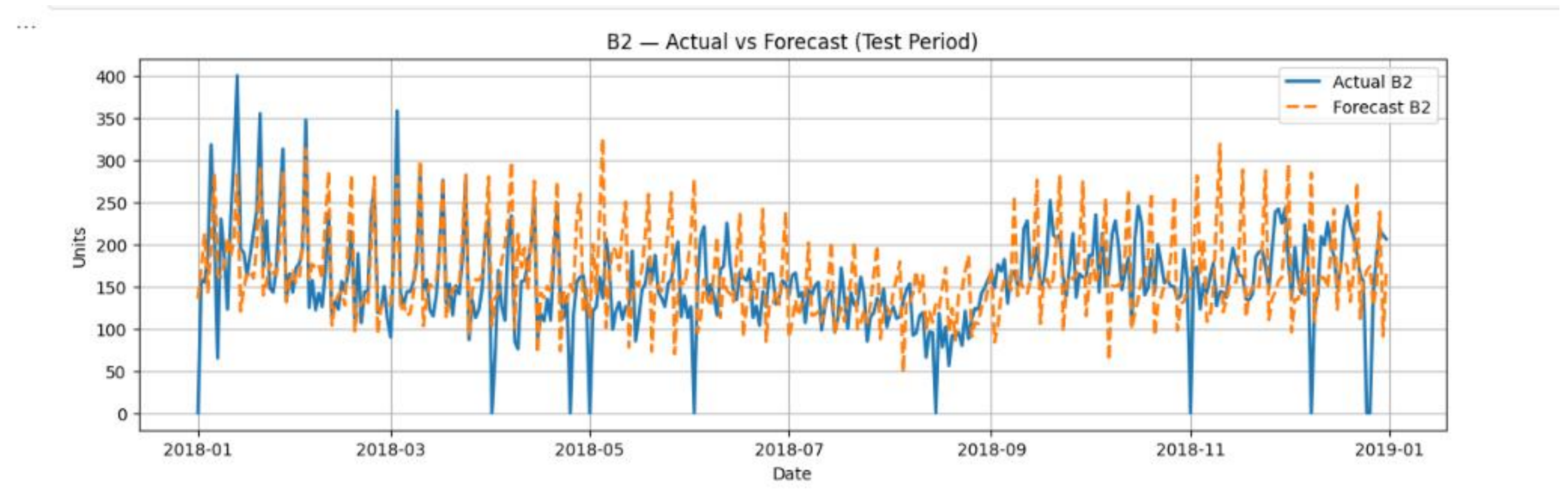


*Figure 2-19 Brand B2 Forecast*

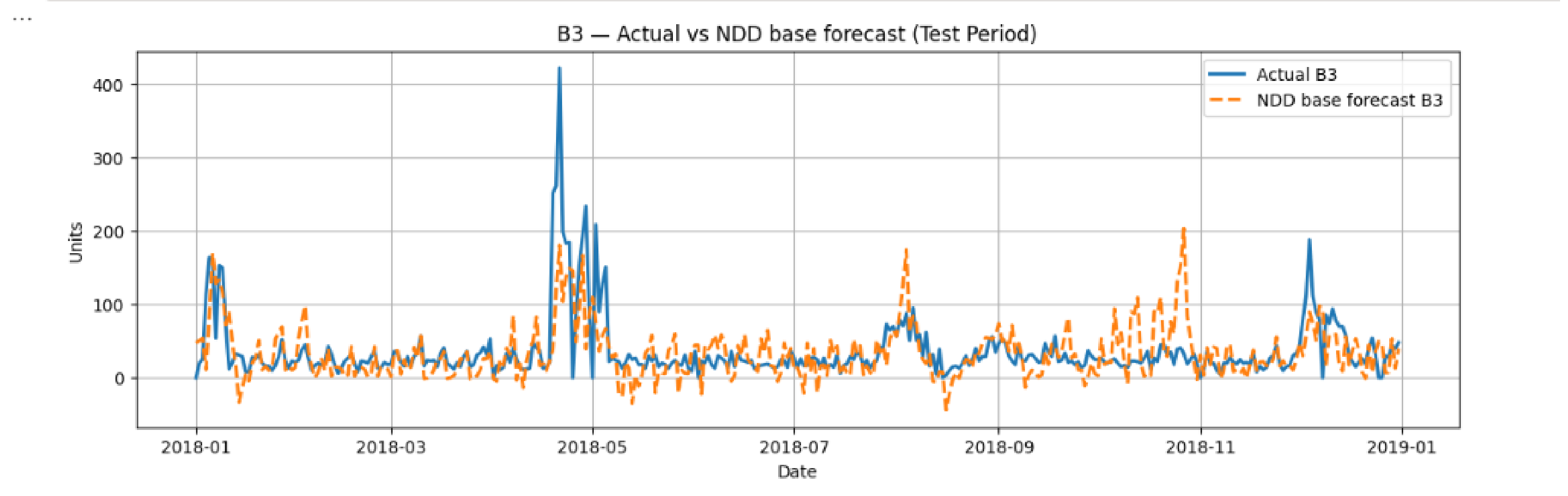


*Figure 2-20 Brand B3 Forecast*

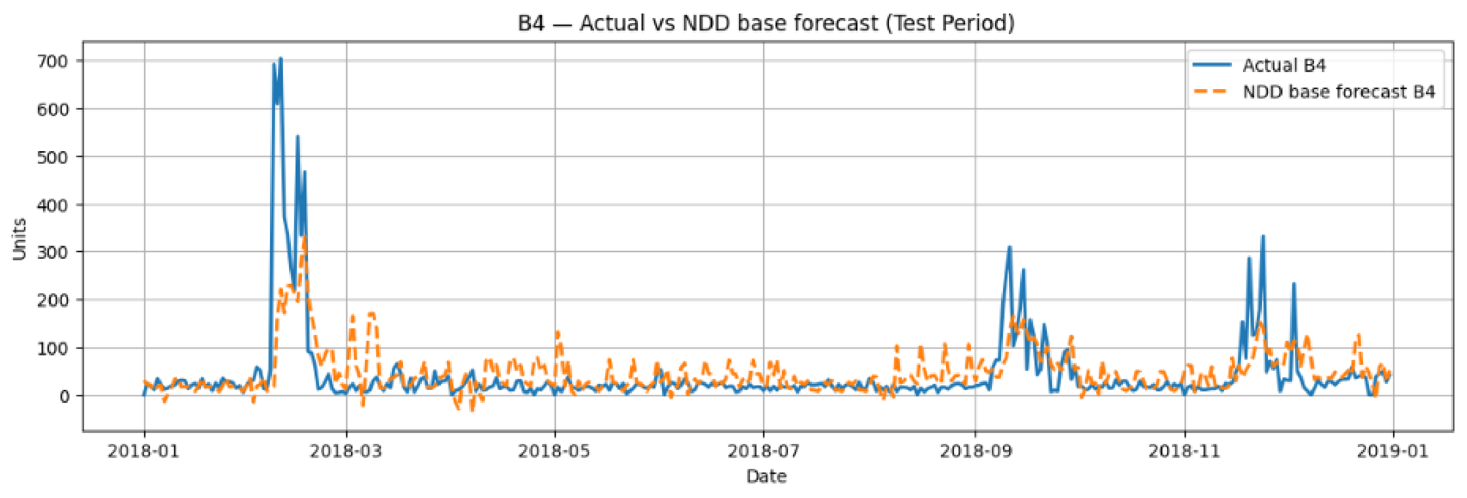


*Figure 2-21Brand B4 Forecast*

As we can see, all the figures show that the forecast is pretty accurate

## 2.9 Conclusions:

### 2.9.1 Comparing ML Model 1 and ML Model 2

Comparing ML model 1 (Ridge Regression with no information of hierarchy) vs. ML Model 2 (NND Model with hierarchy information) we can clearly see that the forecast of ML model 2 (Figure 2-17, Figure 2-19, Figure 2-20, Figure 2-21) are much better. The figures visually show better forecasts then the forecast of ML model 1 (Figure 2-10)

Hence the NND model was able to generate better forecasts by including information of the hierarchy in the brands to its items.

### 2.9.2 Future Work Scope

Although in this project, I was not able to use the full level of hierarchy, (I omitted the top level of store level sales that accumulate all brand sales) it can be left to future work to implement this 3 level hierarchy in the NND architecture and train all 3 hierarchies of time series simultaneously.

## 3 References

[1] R. &. A. G. Hyndman, "Forecasting: principles and practice.," OTexts, 2018.

[2] P. M. a. V. P. a. A. M. Sudoso, "A machine learning approach for forecasting hierarchical time series," *Expert Systems with Applications,* p. 115102, 2021.

[3] P. M. a. V. P. a. A. M. Sudoso, "UC Irvine Machine Learning Repository," [Online]. Available: https://archive.ics.uci.edu/dataset/611/hierarchical+sales+data.