Forecasting Retail Sales for Walmart at a large scale

IDS 561 – ANALYTICS FOR BIG DATA

ANIRUDHA BALKRISHNA, SANJAY MADESHA & SHUBHAM KHODE

Contents

Introduction 1

Objective 1

Data 1

Techniques 1

Data Preparation 2

Identifying Pattern & Trends 3

Multiple Time Series Forecasting 5

Results 8

Conclusion and Future Scope 10

Tasks 11

References 11

Introduction

The Retail Sector faces the fiercest of competition when compared to other sectors. In order to stay ahead, companies in the retail segment strive to differentiate themselves by providing different products and services to attract as many consumers as possible. Traditionally, the employees were trained to be attentive and polite and were encouraged to acquire deep knowledge so that they can serve their customer needs. Feedback from customers helps in defining future marketing strategies and store operations that impact the overall supply chain management. The emergence of Big Data in recent years has enhanced the efficiency of retailers in the last few years while also increasing competitions. In hopes to get ahead of competition, retailers are using Big Data analytics to understand what customers wants and how they can fulfil their needs.

Objective

Big Data analytics in retail sector is enabling companies to create customer recommendations based on their purchase history thereby resulting in personalized shopping experiences and improved customer service. In our project, we have attempted to use big data analytics and solve problems in the retail industry.

Our specific objective is to obtain and explore data generated by a set of Walmart retail stores. Therefore, we attempt to –

- Analyse past store sales data and identify key trends as well as factors that affect sales.
- Develop an accurate forecasting tool to predict their stores' sales for all the weeks in a year.

Both these objectives will help us to provide insights into what factors are critical to accurately forecast store sales for Walmart

Data

We are using publicly available sales data for 45 Walmart stores across the United States. The primary data consists of weekly sales per store per department. Additionally, the data also has details on store size, temperature, fuel price, promotional markdowns and other macro variables like CPI, unemployment rate, etc.

For our purpose, we are using a sample dataset available on Kaggle which is 3.22 MB in size with more than 420K+ rows for training and 100K+ rows of validation data. The entire dataset is split across four files, which is merged into the main dataset as a part of the data cleaning and pre-processing process.

Techniques

To achieve the objectives, we first need to organise the data that is spread over multiple files. Once this is accomplished, we need to understand the data and analyse the same. Lastly, we will be required to use the data to generate a demand forecast.

Data Preparation

Walmart, being one of the largest retailers, has a huge number of stores with different sizes that are spread across the country. Since intuitively the size store is directly related to its sales, we first categorize stores into three groups based on their size. The information related to store size can be found in 'stores.csv', and we define the store size bins as follows –

Small	Less than 75,000
Medium	Between 75,000 to 150,000
Large	Greater than 150,000

We further notice that the granularity of sales data is per store per department. In our sample 'train.csv' data, we have weekly sales data spread across 45 stores and 88 departments. To make forecasting easier and improve its accuracy, we need to perform aggregation, since aggregated data is inherently less noisy than low-level data.

After setting up the spark Environment, we have achieved the following –

1. Merging

As stated previously, the dataset is spread across 4 files, so in order to analyse all the data, these separate datasets need to be merged. Therefore, to prepare the final training data, we first merge the train data with the features data on the common keys – Stores and Date to extract additional features. This processed data is then merged again with stores data to get additional information regarding the stores such as Type and Size.

```
[ ]  1  merged_data.show()
```

Dept	Weekly_Sales	Store	Date	Temperature	Fuel_Price	MarkDown1	MarkDown2	MarkDown3	MarkDown4	MarkDown5	CPI	Unemployment	IsHoliday	Type	Size	Bucket
1	24924.5	1	2010-02-05	42.31	2.572	NA	NA	NA	NA	NA	211.0963582	8.106	FALSE	A	151315	Large
1	46039.49	1	2010-02-12	38.51	2.548	NA	NA	NA	NA	NA	211.2421698	8.106	TRUE	A	151315	Large
1	41595.55	1	2010-02-19	39.93	2.514	NA	NA	NA	NA	NA	211.2891429	8.106	FALSE	A	151315	Large
1	19403.54	1	2010-02-26	46.63	2.561	NA	NA	NA	NA	NA	211.3196429	8.106	FALSE	A	151315	Large
1	21827.9	1	2010-03-05	46.5	2.625	NA	NA	NA	NA	NA	211.3501429	8.106	FALSE	A	151315	Large
1	21043.39	1	2010-03-12	57.79	2.667	NA	NA	NA	NA	NA	211.3806429	8.106	FALSE	A	151315	Large

Once the final data is ready, using spark SQL and cast function the column datatypes are type casted into appropriate datatypes and NA values are removed.

2. Bucketing

For bucketing, we have used the 'when()' function along with 'lit()' function. The bucketing is done in the Stores data file. This file essentially has data on Walmart stores along with their sizes. The 'when' function helps us to specify the condition and the lit function specifies the category. Additionally, we have grouped the dataset by dates and calculated the average sale by week.

```
6  medium_stores_data.show(5)
```

```
+----------+-------+-------+------+------+-------+------+-------+-------+------+------+-------+------+------+-----------------+
|      Date|     10|     12|    15|    17|     18|    21|     22|     23|    25|    29|     35|    45|     9|             Mean|
+----------+-------+-------+------+------+-------+------+-------+-------+------+------+-------+------+------+-----------------+
|2010-02-12|2175995|1117831|682415|841921|1187849|809289|1022543|1380858|583337|529647|1168790|656962|552644|977698.5384615385|
|2010-09-24|1655007| 851887|548509|852845| 950832|671656| 902745|1099025|607788|465309| 758042|678201|452878|807286.4615384615|
|2012-03-09|1917452|1113175|545087|860194|1084858|755054| 991092|1292692|643684|504718| 796323|776932|574925|912014.3076923077|
|2010-08-20|1983159| 948418|637060|783897|1141829|848843|1017015|1439871|724470|531608|1033688|708539|499299|945976.6153846154|
|2012-08-17|1827764|1004971|579706|844895|1048101|751931| 981239|1510097|728435|416851| 887947|722472|535120|         910733.0|
+----------+-------+-------+------+------+-------+------+-------+-------+------+------+-------+------+------+-----------------+
```

In order to achieve this, we have pivoted the dataset by the stores under a category (Size Bucket). As shown in the picture above, we have created a pivot for the “Medium” Category stores and identified their weekly sales.

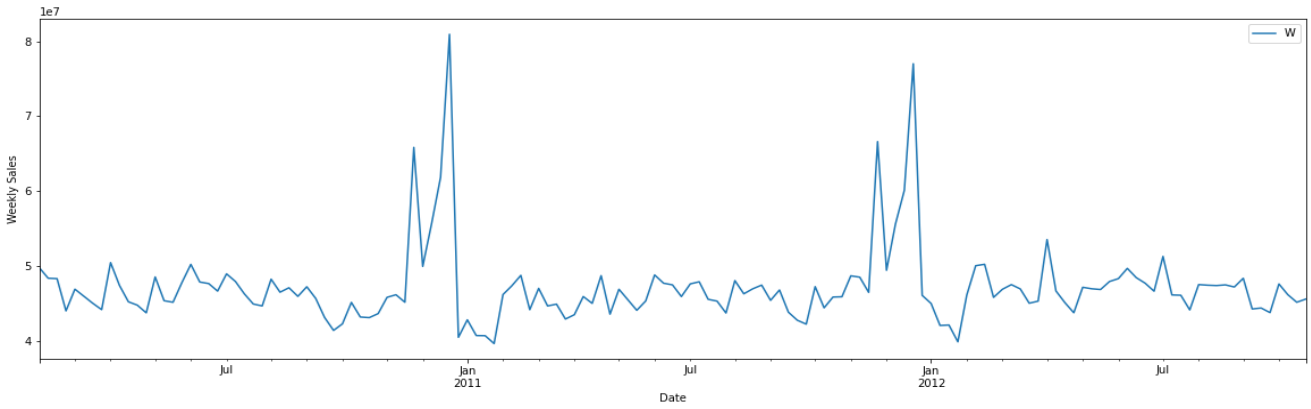
Identifying Pattern & Trends

Our first objective is to analyse past store sales data and identify key trends as well as factors that affect sales. In ‘features.csv’ file, there are a few good indicators like CPI, fuel prices, unemployment, temperature, holiday, etc. We intend to see which of these factors maybe impactful to the forecast sales. To do this, we need to merge the ‘features.csv’ with ‘train.csv’ and perform bivariate analysis.

Lastly, since the nature of data is time series, we plan to decompose it into components - level, trend, seasonality, random (noise). Knowing all the components in a time series data would help us to get a fine understanding of data and making more accurate model for better forecasting values.

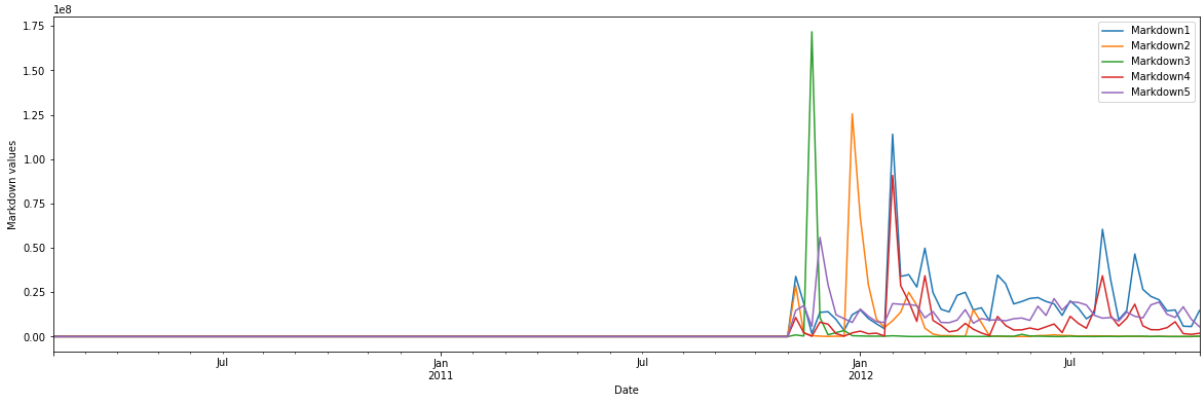
To start off we calculate the number of distinct stores and departments present in our dataset which is found to be 45 and 81 respectively. We then move on to perform univariant analysis of the columns present in our dataset.

1. Univariate analysis on Weekly sales column: — plotting it over date



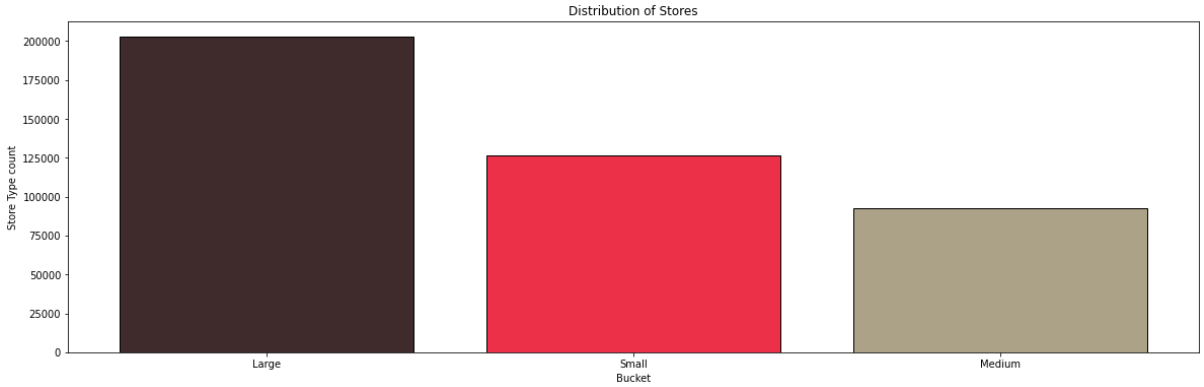
From the plot above, it can be inferred that weekly sales undergo a sharp increase towards the year end possibly due to presence of holiday season during the same time.

2. **Univariate analysis on markdown1, markdown2, markdown3, markdown4, markdown5 column**



From the plot above it can be inferred that, till December 2011 there were no Markdown values and then a spike can be seen at the start of January 2012 for all Markdown values with Markdown 1 having the biggest spike. However, Markdown 3 seem to be markdown which has the highest value consistently when compared over the course of the remaining time period.

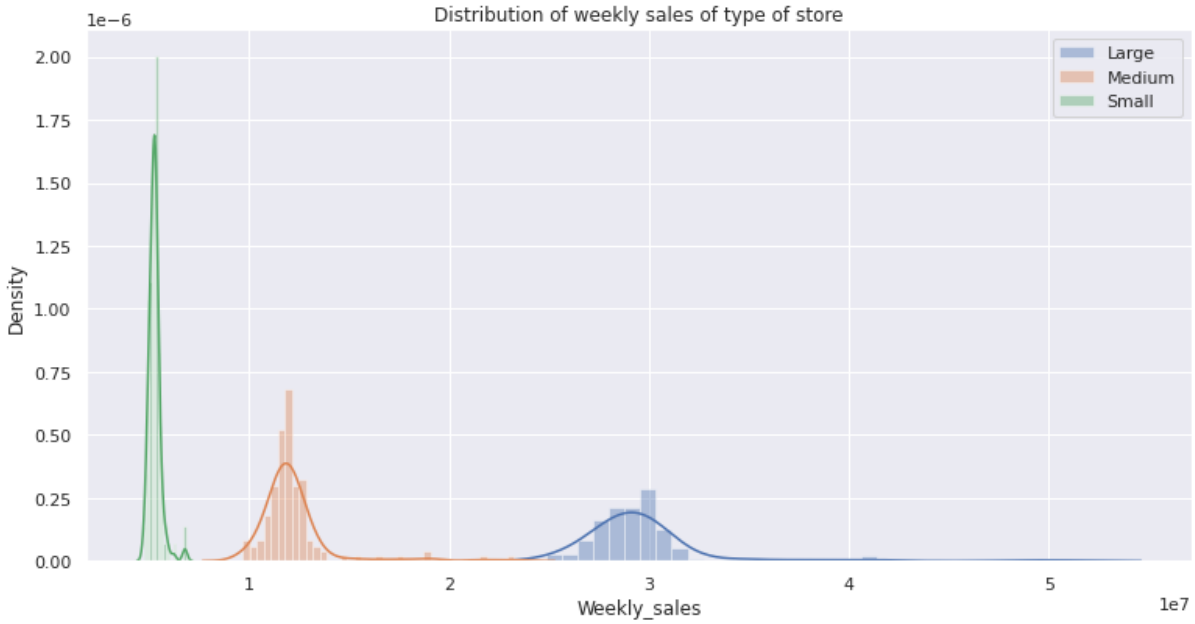
3. **Univariate analysis on Bucket column**



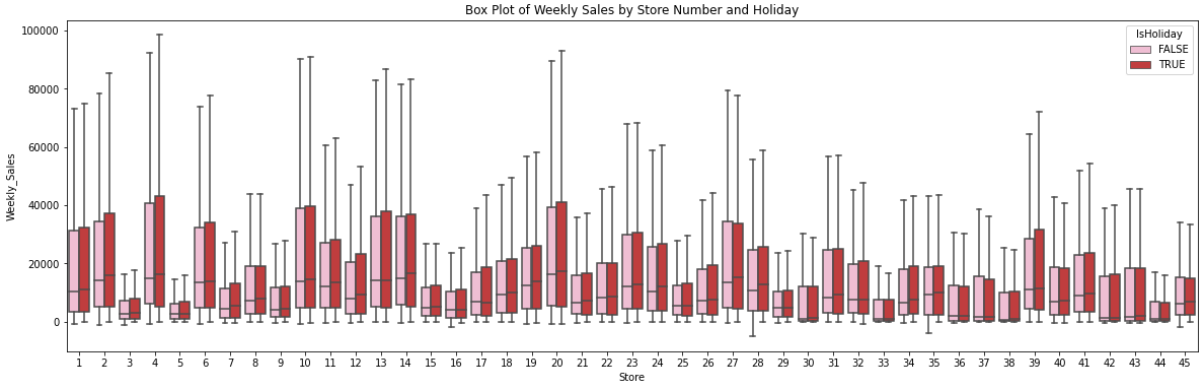
The distribution of different types of stores that were created based on their sales – Large, Medium and Small can be seen above. Large and Small stores form the majority indicating the performance of the stores in the dataset to be at extremes side of the scales that is either good or poor with only minimal number of stores qualifying as medium.

4. **Analysis of 'Bucket' column with weekly sales**

It is very easy to separate the values of weekly sales values of all type of store as the values are completely in different scales. The small stores have less variance in their weekly and their mean is below 0.5*10^7. The medium and small stores have comparatively larger variance with mean of medium stores located at 1.5*10^7 and mean of small stores located at 2.8*10^7



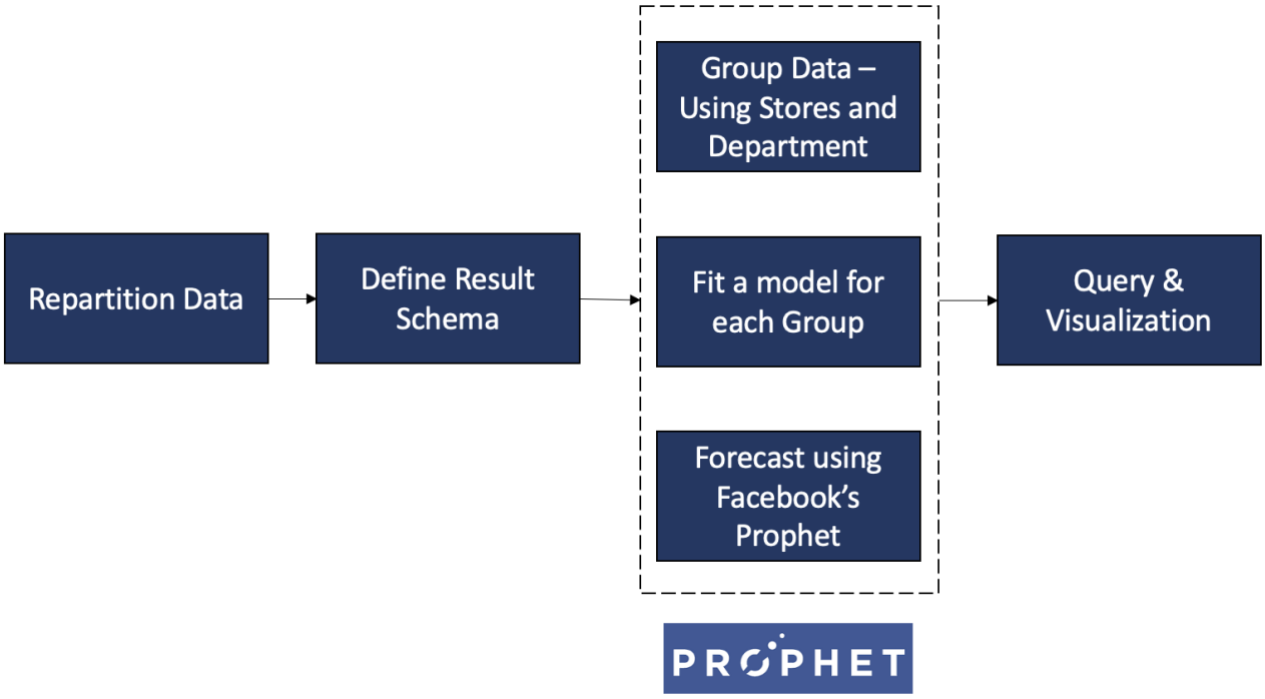
5. Analysis of weekly sales vs store with hue as holiday



From the plot it can be observed that for all the 45 stores, the weekly sales are higher whenever there is a holiday, as compared to non-holiday week. Thus, emphasizing the importance of holidays as a feature in determining the weekly sales.

Multiple Time Series Forecasting

Retailers can now develop more dependable sales projections because of advancements in time series forecasting. The difficulty is to create these estimates in a timely manner and at a granularity that allows the company to make exact inventory adjustments. More and more businesses are discovering that by combining Apache Spark™ and Facebook Prophet, they can overcome the scalability and accuracy limitations of previous solutions.



Our goal is to build hundreds of models and forecasts for each store-department combination, rather than a single forecast for the entire dataset, which would be extremely time-consuming to do as an sequential operation. We leverage power of distributed programming Spark, where individual worker nodes in a cluster can train a subset of models in parallel with other worker nodes, drastically lowering the time it takes to train the full collection of time-series models.

Spark is a lazy evaluation framework, which means it will not do anything until we perform an action. Instead, it will construct a DAG (Directed Acyclic Graph), which is a preliminary execution plan. If we call *explain()* function, we see that all the data is in a single partition.

```
[ ]  1 train.explain()

    == Physical Plan ==
    FileScan csv [Store#244,Dept#245,Date#246,Weekly_Sales#247] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)
```

To partition the data into multiple partitions, we use a SQL statement on top of it to repartition data on per store-department group. Since multi-time series forecasting is an iterative process, we cache data so that we don't have to fetch it over and again. Each worker needs to have access to the subset of data it requires to do its work. We bring together all the time series data for those key values onto a given worker node by grouping the data on key values (in our case - combinations of store and department). This allows these workers to process subsets of the data in parallel, cutting down on the time it takes us to complete our tasks.

```
sql_statement = '''
  SELECT
    Store as store,
    Dept as dept,
    CAST(Date as date) as ds,
    SUM(Weekly_Sales) as y
  FROM train
  GROUP BY store, dept, ds
  ORDER BY store, dept, ds
  '''

train = (
  spark
    .sql( sql_statement )
    .repartition(sc.defaultParallelism, ['store', 'dept'])
  ).cache()
```

Previously, the *explain()* function merely returned an RDD function, but now we can see that it has performed extensive hash partitioning.

```
[ ]  train.explain()

== Physical Plan ==
InMemoryTableScan [store#279, dept#280, ds#281, y#282]
   +- InMemoryRelation [store#279, dept#280, ds#281, y#282], StorageLevel(disk, memory, deserialized, 1 replicas)
         +- Exchange hashpartitioning(store#279, dept#280, 2), REPARTITION_BY_NUM, [id=#225]
            +- *(3) Sort [store#279 ASC NULLS FIRST, dept#280 ASC NULLS FIRST, ds#281 ASC NULLS FIRST], true, 0
               +- Exchange rangepartitioning(store#279 ASC NULLS FIRST, dept#280 ASC NULLS FIRST, ds#281 ASC NULL
                  +- *(2) HashAggregate(keys=[store#244, dept#245, _groupingexpression#288], functions=[sum(Weekly
                     +- Exchange hashpartitioning(store#244, dept#245, _groupingexpression#288, 200), ENSURE_REQU
                        +- *(1) HashAggregate(keys=[store#244, dept#245, _groupingexpression#288], functions=[par
                           +- *(1) Project [Store#244, Dept#245, Weekly_Sales#247, cast(Date#246 as date) AS _gro
                              +- FileScan csv [Store#244,Dept#245,Date#246,Weekly_Sales#247] Batched: false, Datal
```

Next challenge was to think about how we can provide data to *FBProphet* after aggregating it at the store-department level. As we want to create a model for each store-department combination, we need to send a store-department subset from the dataset we just created, train a model on it, and get back forecast. We'd expect the forecast to be returned as a dataset with a format similar to this, where we keep the store and department identifiers for which the forecast was created and limit the output to only the *FBProphet* model's relevant subset of fields. For this, we first define the schema for our return result –

```
result_schema = StructType([
  StructField('ds',DateType()),
  StructField('store',IntegerType()),
  StructField('dept',IntegerType()),
  StructField('y',FloatType()),
  StructField('yhat',FloatType()),
  StructField('yhat_upper',FloatType()),
  StructField('yhat_lower',FloatType())
  ])
```

A custom function is defined that instantiates, configures, and fits our model to the data it has received. The model generates a forecast, which is then returned as the function's result in a format defined in above schema.

```python
def forecast_store_dept( history_pd: pd.DataFrame ) -> pd.DataFrame:

  model = Prophet(
    interval_width=0.95,
    daily_seasonality=False,
    weekly_seasonality=True,
    yearly_seasonality=True,
    holidays = holidays
    )

  model.fit( history_pd )

  future_pd = model.make_future_dataframe(
    periods=52,
    freq='w',
    include_history=True
    )

  forecast_pd = model.predict( future_pd )
```

To call our custom function, we group historical (training) data around store-department and use *applyInPandas()* function. Additionally, we tag along today's date as 'training_date' for future data management purposes.

Since we have data in multiple partitions, we perform coalesce operation (opposite of repartitioning) to take all required results and create one huge file that can be easily queried.

```python
from pyspark.sql.functions import current_date

results = (
  train
    .groupBy('store', 'dept')
      .applyInPandas(forecast_store_dept, schema=result_schema)
    .withColumn('training_date', current_date() )
    )

results.createOrReplaceTempView('forecasts')
```

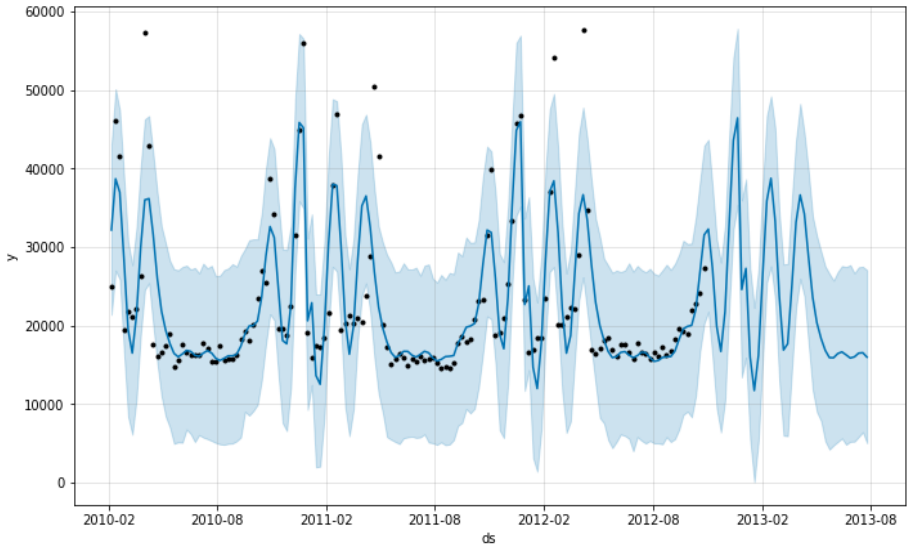
```python
results.coalesce(1)
```

```
DataFrame[ds: date, store: int, dept: int, y: float, yhat: float, yhat_upp
```

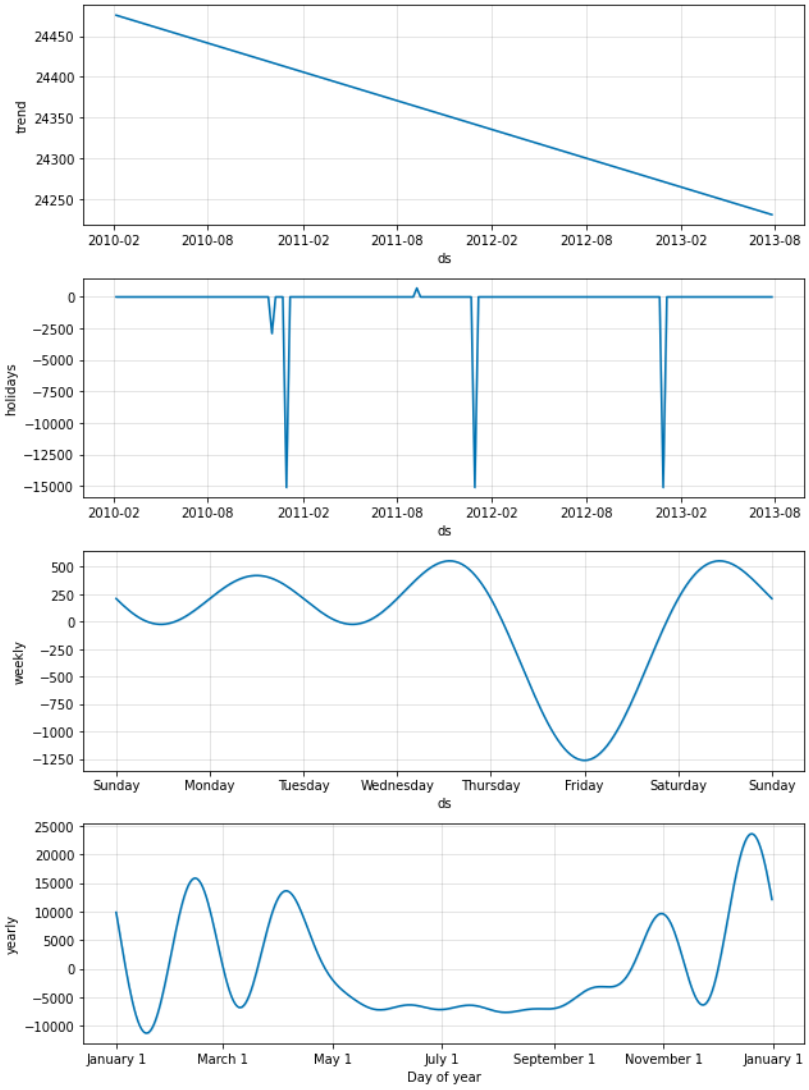
```python
test_sql = '''
SELECT
ds,
y
yhat,
yhat_upper,
yhat_lower
FROM forecasts
WHERE store = 1 AND dept = 1
'''

test_result = spark.sql(test_sql)
```

Results

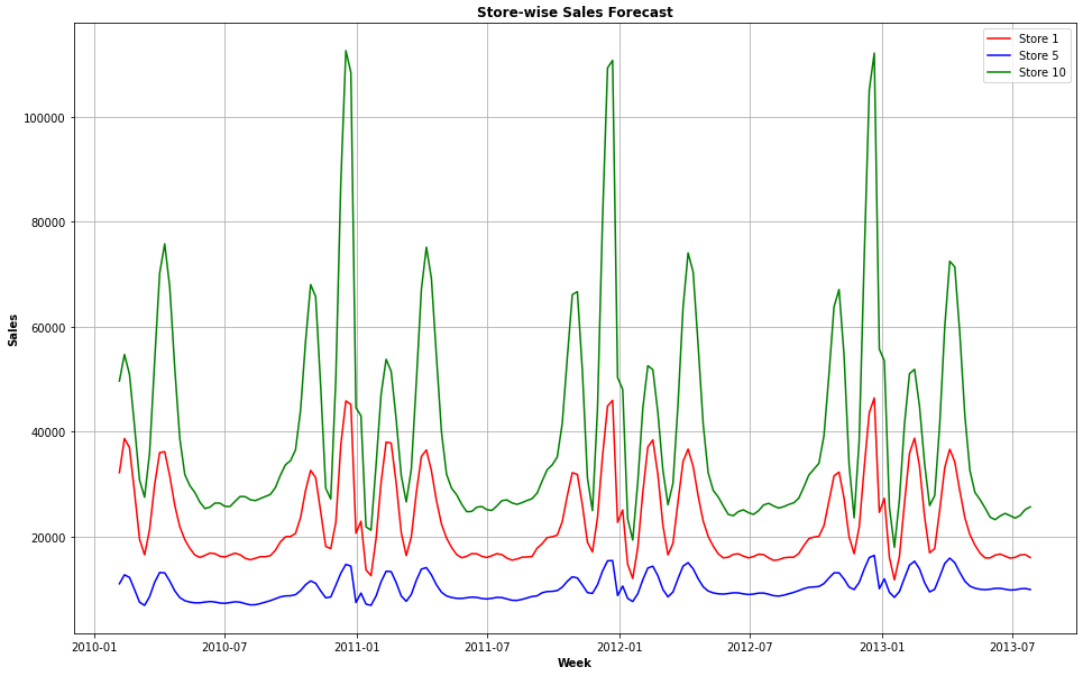
We run a test query for a single store #1 and a single department #1 and generate forecast over test data. To see how our actual and predicted data line up as well as a forecast for the future, we call the *plot()* function from model.



FBProphet also allows us to visualize how the general trend and seasonality varies, along with the effect of holidays on sales by calling *plot_components()* function from model.



Analysts can view the customised forecasts for each store and department via a SQL query. We've displayed the predicted sales for store #1, #5, and #10 for department #1 in the graph below. As you can see, sales estimates differ for each store, but the overall pattern is consistent across all of them, as we would expect.



Conclusion and Future Scope

To summarize, we merged various data files to create one single repository and analysed different features that affect weekly sales. We were able to understand that the weekly sales follow a trend, they are typically higher towards the end of a year. Also, promotions may not be a very important feature to consider but, the effect of holidays can be clearly observed and must be considered for a better forecast.

Finally, we use Facebook's Prophet library to forecast sales for the coming year. We employ partitioning to make sure that parts of data are analysed reducing the strain on computing resources. Using the forecast, we can understand sales patterns for the future for different stores and departments.

In the future, parts of this solution can be automated such that if we require a forecast, we enter a "query" and the results are displayed along with their Tableau visualizations (trends). In order to achieve this, the Py-Spark solution must be connected to a SQL server. This will enable a user to fetch data using SQL queries. Then, this SQL server must be connected to a Tableau File that will show the existing trend and future forecast. In order to visualise real time data, a Window Scheduler type function can be used. The scheduler will make sure that the data is refreshed at a specific time interval. If we assume the interval is 1-minute, every 1-minute data will be updated which will in turn update the SQL database and Tableau visualization.

Tasks

Role	Member
Data Cleaning, Exploratory Data Analysis	Sanjay Madesha
Setting up environment, Merging Data & Bucketing store data	Anirudha Balkrishna
Time Series Analysis, Decomposition & Forecasting Sales	Shubham Khode
Literature Review	All

Table - Task Allocation

References

[1] McKinsey Report on Big Data in Retail

[2] Journal of Big Data, Springer July 2020

[3] Annals of Data Science, March 2015

[4] IEEE, December 2017

[5] Facebook Prophet API

[6] Fine Grained Time Series Forecasting at Scale (Databricks)

[7] Multiple Time Series Forecasting using Prophet