```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types
import org.apache.spark.sql.Column
import org.apache.spark.sql.{DataFrame, Row, SaveMode}
import org.apache.spark.ml.linalg.{Matrix, Vectors}
import org.apache.spark.mllib.linalg.{Vector, Vectors}
import org.apache.spark.ml.stat.Correlation
import org.apache.spark.sql.Row
import spark.implicits._
import scala.util.Random
import org.apache.spark.sql.expressions.Window
import java.util.Date
import java.sql.Timestamp
import java.text.SimpleDateFormat
import java.time.{ZoneId, ZonedDateTime}
import org.apache.spark.sql.types.DoubleType
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.mllib.stat.Statistics
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.sql.functions.col
import org.apache.spark.sql.types.DoubleType
import org.apache.spark.sql.{Column, SparkSession}
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.regression.{RandomForestRegressionModel, RandomForestRegressor}
```


```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types
import org.apache.spark.sql.Column
import org.apache.spark.sql.{DataFrame, Row, SaveMode}
import org.apache.spark.ml.linalg.{Matrix, Vectors}
import org.apache.spark.mllib.linalg.{Vector, Vectors}
import org.apache.spark.ml.stat.Correlation
import org.apache.spark.sql.Row
import spark.implicits._
import scala.util.Random
import org.apache.spark.sql.expressions.Window
import java.util.Date
import java.sql.Timestamp
import java.text.SimpleDateFormat
import java.time.{ZoneId, ZonedDateTime}
import org.apache.spark.sql.types.DoubleType
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.mllib.stat.Statistics
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.sql.functions.col
```


```
// Import tables
val features_data = spark.table("features_data_set_csv").withColumnRenamed("Date", "Dates")
val sales_data = spark.table("sales_data_set_csv").withColumnRenamed("IsHoliday", "IsHolidays")
val stores_data = spark.table("stores_data_set_csv").withColumnRenamed("Store", "Store_num")

features_data: org.apache.spark.sql.DataFrame = [Store: int, Dates: string ... 10 more fields]
sales_data: org.apache.spark.sql.DataFrame = [Store: int, Dept: int ... 3 more fields]
stores_data: org.apache.spark.sql.DataFrame = [Store_num: int, Type: string ... 1 more field]
```

```
// Check table content
features_data.show(10)
sales_data.show(10)
stores_data.show(10)
```

```
+-----+----------+-----------+----------+---------+---------+---------+---------+---------+------------+----------+---------+
|Store|     Dates|Temperature|Fuel_Price|MarkDown1|MarkDown2|MarkDown3|MarkDown4|MarkDown5|         CPI|Unemployment|IsHoliday|
+-----+----------+-----------+----------+---------+---------+---------+---------+---------+------------+----------+---------+
|    1|05/02/2010|      42.31|     2.572|     null|     null|     null|     null|     null|211.09636|       8.106|    false|
|    1|12/02/2010|      38.51|     2.548|     null|     null|     null|     null|     null|211.24217|       8.106|     true|
|    1|19/02/2010|      39.93|     2.514|     null|     null|     null|     null|     null|211.28914|       8.106|    false|
|    1|26/02/2010|      46.63|     2.561|     null|     null|     null|     null|     null|211.31964|       8.106|    false|
|    1|05/03/2010|       46.5|     2.625|     null|     null|     null|     null|     null|211.35014|       8.106|    false|
|    1|12/03/2010|      57.79|     2.667|     null|     null|     null|     null|     null|211.38065|       8.106|    false|
|    1|19/03/2010|      54.58|      2.72|     null|     null|     null|     null|     null|211.21564|       8.106|    false|
|    1|26/03/2010|      51.45|     2.732|     null|     null|     null|     null|     null|211.01804|       8.106|    false|
|    1|02/04/2010|      62.27|     2.719|     null|     null|     null|     null|     null|210.82045|       7.808|    false|
|    1|09/04/2010|      65.86|      2.77|     null|     null|     null|     null|     null|210.62286|       7.808|    false|
+-----+----------+-----------+----------+---------+---------+---------+---------+---------+------------+----------+---------+
only showing top 10 rows

+-----+----+----------+------------+----------+
|Store|Dept|      Date|Weekly_Sales|IsHolidays|
+-----+----+----------+------------+----------+
|    1|   1|05/02/2010|     24924.5|     false|
|    1|   1|12/02/2010|    46039.49|      true|
```

```
// Check sizes
println((features_data.count(), features_data.columns.size))
println((sales_data.count(), sales_data.columns.size))
println((stores_data.count(), stores_data.columns.size))

(8190,12)
(421570,5)
(45,3)
```

Preprocessing and Data Cleaning

```
//Merging three tables
val store = sales_data.join(stores_data, sales_data("Store") === stores_data("Store_num"))
  .drop("Store")
  .drop("IsHolidays")
val sales = features_data.join(store, features_data("Dates") === store("Date") &&
                               features_data("Store") === store("Store_num"))
  .drop("Date")
  .withColumn("Date", to_date($"Dates", "dd/MM/yyyy"))
  .drop("Dates")

store: org.apache.spark.sql.DataFrame = [Dept: int, Date: string ... 4 more fields]
sales: org.apache.spark.sql.DataFrame = [Store: int, Temperature: float ... 15 more fields]
```

```
//Converting T/F values in the IsHoliday column into 1/0
//Transforming original DF into new DF grouped by week
val total_sales_hms = sales
  .groupBy("Date")
  .agg(sum("Weekly_Sales").as("total_sales"))
  .orderBy(col("Date"))
  .select("Date", "total_sales")
  .withColumn("Week", monotonically_increasing_id)
  .withColumn("Dates",concat(col("Date"), lit(" 00:00:00")))
  .drop("Date")
```

```
total_sales_hms: org.apache.spark.sql.DataFrame = [total_sales: double, Week: bigint ... 1 more field]
```

```
display(total_sales_hms)
```

	total_sales	Week	Dates
1	49750740.48903691	0	2010-02-05 00:00:00
2	48336677.65355355	1	2010-02-12 00:00:00
3	48276993.8119329	2	2010-02-19 00:00:00
4	43968571.08629376	3	2010-02-26 00:00:00
5	46871470.31031599	4	2010-03-05 00:00:00
6	45925396.47580373	5	2010-03-12 00:00:00
7	44988974.64017445	6	2010-03-19 00:00:00
8	44133961.00393582	7	2010-03-26 00:00:00
9	50423831.31450006	8	2010-04-02 00:00:00
10	47365290.4499662	9	2010-04-09 00:00:00
11	45183667.11180175	10	2010-04-16 00:00:00
12	44734452.60229535	11	2010-04-23 00:00:00
13	43705126.69820616	12	2010-04-30 00:00:00
14	48503243.52487117	13	2010-05-07 00:00:00
15	45330080.23340504	14	2010-05-14 00:00:00
16	45120108.02918173	15	2010-05-21 00:00:00
17	47757502.58156212	16	2010-05-28 00:00:00
18	50188543.14000124	17	2010-06-04 00:00:00
19	47826546.68921248	18	2010-06-11 00:00:00

Showing all 143 rows.

Interpolation (Data Population)

```
// Function for data population (interpolation)
def tsInterpolate(tsPattern: String) = udf{

  (ts1: String, ts2: String, amt1: Double, amt2: Double) =>
    import java.time.LocalDateTime
    import java.time.format.DateTimeFormatter

    val timeFormat = DateTimeFormatter.ofPattern(tsPattern)

    val perdaysTS = if (ts1 == ts2) Vector(ts1) else {
      val ldt1 = LocalDateTime.parse(ts1, timeFormat)
      val ldt2 = LocalDateTime.parse(ts2, timeFormat)
      Iterator.iterate(ldt1.plusDays(1))(_.plusDays(1)).
        takeWhile(! _.isAfter(ldt2)).
        map(_.format(timeFormat)).
        toVector
    }

    val perdaysAmt = for {
      i <- 1 to perdaysTS.size
    } yield amt1 + ((amt2 - amt1) * i / perdaysTS.size)

    perdaysTS zip perdaysAmt
}

// Populate the original data (becomes estimated daily data)
val df = total_sales_hms.select("Week", "total_sales","Dates")

val tsPattern = "yyyy-MM-dd HH:mm:ss"
val win = Window.orderBy(col("Week"))

val df_sales = df.
  withColumn("datePrev", when(row_number.over(win) === 1, $"Dates").
    otherwise(lag($"Dates", 1).over(win))).
  withColumn("salePrev", when(row_number.over(win) === 1, $"Dates").
    otherwise(lag($"total_sales", 1).over(win))).
  withColumn("interpolatedList",
    tsInterpolate(tsPattern)($"datePrev", $"Dates", $"salePrev", $"total_sales")
  ).
  withColumn("interpolated", explode($"interpolatedList")).
  select( $"interpolated._1".as("Dates"), round($"interpolated._2", 2).as("total_sales")
  )
```

```
tsInterpolate: (tsPattern: String)org.apache.spark.sql.expressions.UserDefinedFunction
df: org.apache.spark.sql.DataFrame = [Week: bigint, total_sales: double ... 1 more field]
tsPattern: String = yyyy-MM-dd HH:mm:ss
win: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@4b554242
df_sales: org.apache.spark.sql.DataFrame = [Dates: string, total_sales: double]
```

```
display(df_sales)
```

	Dates	total_sales
1	2010-02-06 00:00:00	49548731.51
2	2010-02-07 00:00:00	49346722.54
3	2010-02-08 00:00:00	49144713.56
4	2010-02-09 00:00:00	48942704.58
5	2010-02-10 00:00:00	48740695.61
6	2010-02-11 00:00:00	48538686.63
7	2010-02-12 00:00:00	48336677.65
8	2010-02-13 00:00:00	48328151.39
9	2010-02-14 00:00:00	48319625.13
10	2010-02-15 00:00:00	48311098.86
11	2010-02-16 00:00:00	48302572.6
12	2010-02-17 00:00:00	48294046.34
13	2010-02-18 00:00:00	48285520.08
14	2010-02-19 00:00:00	48276993.81
15	2010-02-20 00:00:00	47661504.85
16	2010-02-21 00:00:00	47046015.89
17	2010-02-22 00:00:00	46430526.93
18	2010-02-23 00:00:00	45815037.97

Showing all 994 rows.

Convert Timestamp into 3 Columns (Year, Month, Day)

```
// extract dates into year, month and day
val df_sale1 = df_sales.withColumn("year", year(to_timestamp($"Dates", "yyyy-MM-dd HH:mm:ss")))
                       .withColumn("month", month(to_timestamp($"Dates", "yyyy-MM-dd HH:mm:ss")))
                       .withColumn("day", dayofmonth(to_timestamp($"Dates", "yyyy-MM-dd HH:mm:ss")))
df_sale1.show
```

```
+-------------------+-------------+----+-----+---+
|              Dates|  total_sales|year|month|day|
+-------------------+-------------+----+-----+---+
|2010-02-06 00:00:00|4.954873151E7|2010|    2|  6|
|2010-02-07 00:00:00|4.934672254E7|2010|    2|  7|
|2010-02-08 00:00:00|4.914471356E7|2010|    2|  8|
|2010-02-09 00:00:00|4.894270458E7|2010|    2|  9|
|2010-02-10 00:00:00|4.874069561E7|2010|    2| 10|
|2010-02-11 00:00:00|4.853868663E7|2010|    2| 11|
|2010-02-12 00:00:00|4.833667765E7|2010|    2| 12|
|2010-02-13 00:00:00|4.832815139E7|2010|    2| 13|
|2010-02-14 00:00:00|4.831962513E7|2010|    2| 14|
|2010-02-15 00:00:00|4.831109886E7|2010|    2| 15|
|2010-02-16 00:00:00| 4.83025726E7|2010|    2| 16|
|2010-02-17 00:00:00|4.829404634E7|2010|    2| 17|
|2010-02-18 00:00:00|4.828552008E7|2010|    2| 18|
|2010-02-19 00:00:00|4.827699381E7|2010|    2| 19|
|2010-02-20 00:00:00|4.766150485E7|2010|    2| 20|
|2010-02-21 00:00:00|4.704601589E7|2010|    2| 21|
|2010-02-22 00:00:00|4.643052693E7|2010|    2| 22|
|2010-02-23 00:00:00|4.581503797E7|2010|    2| 23|
```

Time-Series Train-Test Split

```
// Function for timeseries train-test split
def ts_split(split_pct: Double, df: DataFrame) : (DataFrame, DataFrame) =  {
  /**
    * Returns two dataframe based on the given split percentage
    * The two dataframe can be used as train and test dataset
    *
    */
  val breakpoint = math.ceil(df.count() * split_pct)

  val df_id = df.withColumn("id", monotonically_increasing_id)
  val first = df_id.where($"id" < breakpoint).drop("id")
  val second = df_id.where($"id" >= breakpoint).drop("id")

  return (first, second)
}
```

```
ts_split: (split_pct: Double, df: org.apache.spark.sql.DataFrame)(org.apache.spark.sql.DataFrame, org.apache.spark.sql.DataFrame)
```

Time-Series Seasonality Differencing and Feature Engineering for Different Prediction Horizons

```
val w = Window.orderBy(col("Dates"))
val weekly_win = Window.orderBy(col("Dates")).rowsBetween(-7, 0)
val biweekly_win = Window.orderBy(col("Dates")).rowsBetween(-14, 0)
val triweekly_win = Window.orderBy(col("Dates")).rowsBetween(-21, 0)
val monthly_win = Window.orderBy(col("Dates")).rowsBetween(-28, 0)
val week5_win = Window.orderBy(col("Dates")).rowsBetween(-35, 0)
val week6_win = Window.orderBy(col("Dates")).rowsBetween(-42, 0)
val week7_win = Window.orderBy(col("Dates")).rowsBetween(-49, 0)

// Prediction horizon: 1 day
val df_1day1 = df_sale1
  .withColumn("lag_1year", lag("total_sales", 365, 0).over(w))
  .where($"lag_1year">0)
  .withColumn("diff_sales", abs($"total_sales" - $"lag_1year"))
  .withColumn("lag_1day", lag("diff_sales", 1, 0).over(w))
  .withColumn("lag_5days", lag("diff_sales", 5, 0).over(w))
  .withColumn("lag_1week", lag("diff_sales", 7, 0).over(w))
  .withColumn("lag_2weeks", lag("diff_sales", 14, 0).over(w))
  .withColumn("lag_3weeks", lag("diff_sales", 21, 0).over(w))
  .where($"lag_3weeks">0)
  .withColumn("ma_1week", avg("lag_1day").over(weekly_win))
  .withColumn("ma_2weeks", avg("lag_1day").over(biweekly_win))
  .withColumn("ma_3weeks", avg("lag_1day").over(triweekly_win))
  .withColumn("ma_4weeks", avg("lag_1day").over(monthly_win))
  .withColumn("std_1week", stddev("lag_1day").over(weekly_win))
  .withColumn("std_2weeks", stddev("lag_1day").over(biweekly_win))
  .withColumn("std_3weeks", stddev("lag_1day").over(triweekly_win))
  .withColumn("std_4weeks", stddev("lag_1day").over(monthly_win))
  .where($"std_4weeks">0)
  .withColumn("label", lead("diff_sales", 1, 0).over(w))
  .na.drop
  .drop("Dates", "total_sales")
//   .withColumn("Days", monotonically_increasing_id)

val season_diff_1day = df_1day1.select("lag_1year")

val df_1day = df_1day1.drop("lag_1year")

//Prediction horizon: 1 week
val df_1week1 = df_sale1
  .withColumn("lag_1year", lag("total_sales", 365, 0).over(w))
  .where($"lag_1year">0)
  .withColumn("diff_sales", abs($"total_sales" - $"lag_1year"))
  .withColumn("lag_1week", lag("diff_sales", 7, 0).over(w))
  .withColumn("lag_2weeks", lag("diff_sales", 14, 0).over(w))
  .withColumn("lag_3weeks", lag("diff_sales", 21, 0).over(w))
  .withColumn("lag_4weeks", lag("diff_sales", 28, 0).over(w))
  .where($"lag_4weeks">0)
  .withColumn("ma_1week", avg("lag_1week").over(weekly_win))
  .withColumn("ma_2weeks", avg("lag_1week").over(biweekly_win))
  .withColumn("ma_3weeks", avg("lag_1week").over(triweekly_win))
  .withColumn("ma_4weeks", avg("lag_1week").over(monthly_win))
  .withColumn("std_1week", stddev("lag_1week").over(weekly_win))
  .withColumn("std_2weeks", stddev("lag_1week").over(biweekly_win))
  .withColumn("std_3weeks", stddev("lag_1week").over(triweekly_win))
  .withColumn("std_4weeks", stddev("lag_1week").over(monthly_win))
  .where($"std_4weeks">0)
  .withColumn("label", lead("diff_sales", 7, 0).over(w))
  .na.drop
  .drop("Dates", "total_sales")

val season_diff_1week = df_1week1.select("lag_1year")

val df_1week = df_1week1.drop("lag_1year")

// Prediction horizon: 2 weeks
val df_2weeks1 = df_sale1
  .withColumn("lag_1year", lag("total_sales", 365, 0).over(w))
  .where($"lag_1year">0)
  .withColumn("diff_sales", abs($"total_sales" - $"lag_1year"))
  .withColumn("lag_2weeks", lag("diff_sales", 14, 0).over(w))
  .withColumn("lag_3weeks", lag("diff_sales", 21, 0).over(w))
  .withColumn("lag_4weeks", lag("diff_sales", 28, 0).over(w))
  .withColumn("lag_5weeks", lag("diff_sales", 35, 0).over(w))
  .where($"lag_5weeks">0)
  .withColumn("ma_2weeks", avg("lag_2weeks").over(biweekly_win))
  .withColumn("ma_3weeks", avg("lag_2weeks").over(triweekly_win))
  .withColumn("ma_4weeks", avg("lag_2weeks").over(monthly_win))
  .withColumn("ma_5weeks", avg("lag_2weeks").over(week5_win))
  .withColumn("std_2weeks", stddev("lag_2weeks").over(biweekly_win))
  .withColumn("std_3weeks", stddev("lag_2weeks").over(triweekly_win))
```

```
  .withColumn("std_4weeks", stddev("lag_2weeks").over(monthly_win))
  .withColumn("std_5weeks", stddev("lag_2weeks").over(week5_win))
  .where($"std_5weeks">0)
  .withColumn("label", lead("diff_sales", 14, 0).over(w))
  .na.drop
  .drop("Dates", "total_sales")

val season_diff_2weeks = df_2weeks1.select("lag_1year")

val df_2weeks = df_2weeks1.drop("lag_1year")

// Prediction horizon: 3 weeks
val df_3weeks1 = df_sale1
  .withColumn("lag_1year", lag("total_sales", 365, 0).over(w))
  .where($"lag_1year">0)
  .withColumn("diff_sales", abs($"total_sales" - $"lag_1year"))
  .withColumn("lag_3weeks", lag("diff_sales", 21, 0).over(w))
  .withColumn("lag_4weeks", lag("diff_sales", 28, 0).over(w))
  .withColumn("lag_5weeks", lag("diff_sales", 35, 0).over(w))
  .withColumn("lag_6weeks", lag("diff_sales", 42, 0).over(w))
  .where($"lag_6weeks">0)
  .withColumn("ma_3weeks", avg("lag_3weeks").over(triweekly_win))
  .withColumn("ma_4weeks", avg("lag_3weeks").over(monthly_win))
  .withColumn("ma_5weeks", avg("lag_3weeks").over(week5_win))
  .withColumn("ma_6weeks", avg("lag_3weeks").over(week6_win))
  .withColumn("std_3weeks", stddev("lag_3weeks").over(triweekly_win))
  .withColumn("std_4weeks", stddev("lag_3weeks").over(monthly_win))
  .withColumn("std_5weeks", stddev("lag_3weeks").over(week5_win))
  .withColumn("std_6weeks", stddev("lag_3weeks").over(week6_win))
  .where($"std_6weeks">0)
  .withColumn("label", lead("diff_sales", 21, 0).over(w))
  .na.drop
  .drop("Dates", "total_sales")

val season_diff_3weeks = df_3weeks1.select("lag_1year")

val df_3weeks = df_3weeks1.drop("lag_1year")

// Prediction horizon: 1 month
val df_1month1 = df_sale1
  .withColumn("lag_1year", lag("total_sales", 365, 0).over(w))
  .where($"lag_1year">0)
  .withColumn("diff_sales", abs($"total_sales" - $"lag_1year"))
  .withColumn("lag_4weeks", lag("diff_sales", 28, 0).over(w))
  .withColumn("lag_5weeks", lag("diff_sales", 35, 0).over(w))
  .withColumn("lag_6weeks", lag("diff_sales", 42, 0).over(w))
  .withColumn("lag_7weeks", lag("diff_sales", 49, 0).over(w))
  .where($"lag_7weeks">0)
  .withColumn("ma_4weeks", avg("lag_4weeks").over(monthly_win))
  .withColumn("ma_5weeks", avg("lag_4weeks").over(week5_win))
  .withColumn("ma_6weeks", avg("lag_4weeks").over(week6_win))
  .withColumn("ma_7weeks", avg("lag_4weeks").over(week7_win))
  .withColumn("std_4weeks", stddev("lag_4weeks").over(monthly_win))
  .withColumn("std_5weeks", stddev("lag_4weeks").over(week5_win))
  .withColumn("std_6weeks", stddev("lag_4weeks").over(week6_win))
  .withColumn("std_7weeks", stddev("lag_4weeks").over(week7_win))
  .where($"std_7weeks">0)
  .withColumn("label", lead("diff_sales", 28, 0).over(w))
  .na.drop
  .drop("Dates", "total_sales")

val season_diff_1month = df_1month1.select("lag_1year")

val df_1month = df_1month1.drop("lag_1year")
```

```
w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@1902da1a
weekly_win: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@1d00b113
biweekly_win: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@2a63769d
triweekly_win: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@33bd20ab
monthly_win: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@3ce0eecc
week5_win: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@24f0486a
week6_win: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@151615f7
week7_win: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@4ff4c786
df_1day1: org.apache.spark.sql.DataFrame = [year: int, month: int ... 17 more fields]
season_diff_1day: org.apache.spark.sql.DataFrame = [lag_1year: double]
df_1day: org.apache.spark.sql.DataFrame = [year: int, month: int ... 16 more fields]
df_1week1: org.apache.spark.sql.DataFrame = [year: int, month: int ... 16 more fields]
season_diff_1week: org.apache.spark.sql.DataFrame = [lag_1year: double]
df_1week: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
df_2weeks1: org.apache.spark.sql.DataFrame = [year: int, month: int ... 16 more fields]
season_diff_2weeks: org.apache.spark.sql.DataFrame = [lag_1year: double]
df_2weeks: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
df_3weeks1: org.apache.spark.sql.DataFrame = [year: int, month: int ... 16 more fields]
season_diff_3weeks: org.apache.spark.sql.DataFrame = [lag_1year: double]
df_3weeks: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
df_1month1: org.apache.spark.sql.DataFrame = [year: int, month: int ... 16 more fields]
```

```
// Split train-test sets from dataframes (80/20)
val (train_set_1day_a, test_set_1day) = ts_split(0.8, df_1day)
val (train_set_1week_a, test_set_1week) = ts_split(0.8, df_1week)
val (train_set_2weeks_a, test_set_2weeks) = ts_split(0.8, df_2weeks)
val (train_set_3weeks_a, test_set_3weeks) = ts_split(0.8, df_3weeks)
val (train_set_1month_a, test_set_1month) = ts_split(0.8, df_1month)
```

```
train_set_1day_a: org.apache.spark.sql.DataFrame = [year: int, month: int ... 16 more fields]
test_set_1day: org.apache.spark.sql.DataFrame = [year: int, month: int ... 16 more fields]
train_set_1week_a: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
test_set_1week: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
train_set_2weeks_a: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
test_set_2weeks: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
train_set_3weeks_a: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
test_set_3weeks: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
train_set_1month_a: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
test_set_1month: org.apache.spark.sql.DataFrame = [year: int, month: int ... 15 more fields]
```

```
// Create gap between train and test sets to ensure no data leakage
val train_set_1day = train_set_1day_a.limit(464)
val train_set_1week = train_set_1week_a.limit(445)
val train_set_2weeks = train_set_2weeks_a.limit(426)
val train_set_3weeks = train_set_3weeks_a.limit(406)
val train_set_1month = train_set_1month_a.limit(387)
```

```
train_set_1day: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [year: int, month: int ... 16 more fields]
train_set_1week: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [year: int, month: int ... 15 more fields]
train_set_2weeks: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [year: int, month: int ... 15 more fields]
train_set_3weeks: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [year: int, month: int ... 15 more fields]
train_set_1month: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [year: int, month: int ... 15 more fields]
```

```
// Split the seasonal differences using the same ratio
// Only use "test_diff" later for reconstruction
val (train_diff_1day, test_diff_1day) = ts_split(0.8, season_diff_1day)
val (train_diff_1week, test_diff_1week) = ts_split(0.8, season_diff_1week)
val (train_diff_2weeks, test_diff_2weeks) = ts_split(0.8, season_diff_2weeks)
val (train_diff_3weeks, test_diff_3weeks) = ts_split(0.8, season_diff_3weeks)
val (train_diff_1month, test_diff_1month) = ts_split(0.8, season_diff_1month)
```

```
train_diff_1day: org.apache.spark.sql.DataFrame = [lag_1year: double]
test_diff_1day: org.apache.spark.sql.DataFrame = [lag_1year: double]
train_diff_1week: org.apache.spark.sql.DataFrame = [lag_1year: double]
```

```
test_diff_1week: org.apache.spark.sql.DataFrame = [lag_1year: double]
train_diff_2weeks: org.apache.spark.sql.DataFrame = [lag_1year: double]
test_diff_2weeks: org.apache.spark.sql.DataFrame = [lag_1year: double]
train_diff_3weeks: org.apache.spark.sql.DataFrame = [lag_1year: double]
test_diff_3weeks: org.apache.spark.sql.DataFrame = [lag_1year: double]
train_diff_1month: org.apache.spark.sql.DataFrame = [lag_1year: double]
test_diff_1month: org.apache.spark.sql.DataFrame = [lag_1year: double]
```

Convert Features into Dense Vectors

```
// Vectorize the features

// Prediction horizon: 1 day
val vectorAssembler_1day = new VectorAssembler()
  .setInputCols(Array("year","month","day","lag_1day","lag_5days","lag_1week","lag_2weeks","lag_3weeks",
                      "ma_1week","ma_2weeks","ma_3weeks","ma_4weeks","std_1week","std_2weeks",
                     "std_3weeks","std_4weeks","diff_sales"))
  .setOutputCol("features")

val train_1day = vectorAssembler_1day.transform(train_set_1day)
  .drop("year","month","day","lag_1day","lag_5days","lag_1week","lag_2weeks","lag_3weeks",
        "ma_1week","ma_2weeks","ma_3weeks","ma_4weeks","std_1week","std_2weeks",
        "std_3weeks","std_4weeks","diff_sales")

val test_1day = vectorAssembler_1day.transform(test_set_1day)
  .drop("year","month","day","lag_1day","lag_5days","lag_1week","lag_2weeks","lag_3weeks",
        "ma_1week","ma_2weeks","ma_3weeks","ma_4weeks","std_1week","std_2weeks",
        "std_3weeks","std_4weeks","diff_sales")

// Prediction horizon: 1 week
val vectorAssembler_1week = new VectorAssembler()
  .setInputCols(Array("year","month","day","lag_1week","lag_2weeks","lag_3weeks","lag_4weeks","ma_1week",
                      "ma_2weeks","ma_3weeks","ma_4weeks","std_1week","std_2weeks","std_3weeks","std_4weeks",
                      "diff_sales"))
  .setOutputCol("features")

val train_1week = vectorAssembler_1week.transform(train_set_1week)
  .drop("year","month","day","lag_1week","lag_2weeks","lag_3weeks","lag_4weeks","ma_1week",
        "ma_2weeks","ma_3weeks","ma_4weeks","std_1week","std_2weeks","std_3weeks","std_4weeks",
       "diff_sales")

val test_1week = vectorAssembler_1week.transform(test_set_1week)
  .drop("year","month","day","lag_1week","lag_2weeks","lag_3weeks","lag_4weeks","ma_1week",
        "ma_2weeks","ma_3weeks","ma_4weeks","std_1week","std_2weeks","std_3weeks","std_4weeks",
        "diff_sales")

// Prediction horizon: 2 weeks
val vectorAssembler_2weeks = new VectorAssembler()
  .setInputCols(Array("year","month","day","lag_2weeks","lag_3weeks","lag_4weeks","lag_5weeks",
                      "ma_2weeks","ma_3weeks","ma_4weeks","ma_5weeks","std_2weeks","std_3weeks",
                      "std_4weeks","std_5weeks","diff_sales"))
  .setOutputCol("features")

val train_2weeks = vectorAssembler_2weeks.transform(train_set_2weeks)
  .drop("year","month","day","lag_2weeks","lag_3weeks","lag_4weeks","lag_5weeks","ma_2weeks",
        "ma_3weeks","ma_4weeks","ma_5weeks","std_2weeks","std_3weeks","std_4weeks","std_5weeks",
        "diff_sales")

val test_2weeks = vectorAssembler_2weeks.transform(test_set_2weeks)
  .drop("year","month","day","lag_2weeks","lag_3weeks","lag_4weeks","lag_5weeks","ma_2weeks",
        "ma_3weeks","ma_4weeks","ma_5weeks","std_2weeks","std_3weeks","std_4weeks","std_5weeks",
        "diff_sales")

// Prediction horizon: 3 weeks
val vectorAssembler_3weeks = new VectorAssembler()
  .setInputCols(Array("year","month","day","lag_3weeks","lag_4weeks","lag_5weeks","lag_6weeks",
                      "ma_3weeks","ma_4weeks","ma_5weeks","ma_6weeks","std_3weeks","std_4weeks",
                      "std_5weeks","std_6weeks","diff_sales"))
  .setOutputCol("features")

val train_3weeks = vectorAssembler_3weeks.transform(train_set_3weeks)
  .drop("year","month","day","lag_3weeks","lag_4weeks","lag_5weeks","lag_6weeks","ma_3weeks",
        "ma_4weeks","ma_5weeks","ma_6weeks","std_3weeks","std_4weeks","std_5weeks","std_6weeks",
        "diff_sales")

val test_3weeks = vectorAssembler_3weeks.transform(test_set_3weeks)
  .drop("year","month","day","lag_3weeks","lag_4weeks","lag_5weeks","lag_6weeks","ma_3weeks",
        "ma_4weeks","ma_5weeks","ma_6weeks","std_3weeks","std_4weeks","std_5weeks","std_6weeks",
        "diff_sales")

// Prediction horizon: 1 month
val vectorAssembler_1month = new VectorAssembler()
  .setInputCols(Array("year","month","day","lag_4weeks","lag_5weeks","lag_6weeks","lag_7weeks",
                      "ma_4weeks","ma_5weeks","ma_6weeks","ma_7weeks","std_4weeks","std_5weeks",
                      "std_6weeks","std_7weeks","diff_sales"))
  .setOutputCol("features")

val train_1month = vectorAssembler_1month.transform(train_set_1month)
  .drop("year","month","day","lag_4weeks","lag_5weeks","lag_6weeks","lag_7weeks","ma_4weeks",
        "ma_5weeks","ma_6weeks","ma_7weeks","std_4weeks","std_5weeks","std_6weeks","std_7weeks",
        "diff_sales")

val test_1month = vectorAssembler_1month.transform(test_set_1month)
  .drop("year","month","day","lag_4weeks","lag_5weeks","lag_6weeks","lag_7weeks","ma_4weeks",
        "ma_5weeks","ma_6weeks","ma_7weeks","std_4weeks","std_5weeks","std_6weeks","std_7weeks",
        "diff_sales")
```

```
vectorAssembler_1day: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler: uid=vecAssembler_379495da5b16, handleInvalid=error, numInputCols=17
train_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector]
test_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector]
vectorAssembler_1week: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler: uid=vecAssembler_90d7efcc70a8, handleInvalid=error, numInputCols=16
train_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector]
test_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector]
vectorAssembler_2weeks: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler: uid=vecAssembler_de7e8fbe08d1, handleInvalid=error, numInputCols=16
train_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector]
test_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector]
vectorAssembler_3weeks: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler: uid=vecAssembler_63259d30c297, handleInvalid=error, numInputCols=16
train_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector]
test_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector]
vectorAssembler_1month: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler: uid=vecAssembler_6da3c6a53487, handleInvalid=error, numInputCols=16
train_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector]
test_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector]
```

Feature Importance (5 different prediction window dataframes)

```
// train a new random forest model
val rf = new RandomForestRegressor()

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_1day)

// Make predictions.
```

```
val predictions = model.transform(test_1day)

// extract feature importance from rf model
val importances = model // this would be your trained model
  .stages(0)
  .asInstanceOf[RandomForestRegressionModel]
  .featureImportances

val features = train_set_1day.columns.filterNot(Set("Date", "label"))

val printImportance = importances.toArray zip features

println("--------------------------------------------")
printImportance.sortBy(-_._1).foreach(x => println(x._2 + " -> " + x._1))
println("--------------------------------------------")
```

```
--------------------------------------------
std_4weeks -> 0.48060663788531005
diff_sales -> 0.20541242033194065
lag_3weeks -> 0.05996095489797325
ma_2weeks -> 0.029115174255551858
month -> 0.026363733331084928
std_1week -> 0.0254821371096213
lag_1day -> 0.0239961907817596 26
ma_1week -> 0.02299645822053426
ma_3weeks -> 0.022664348038718793
std_2weeks -> 0.020879240449779866
day -> 0.019281562052471505
std_3weeks -> 0.018497651159507272
ma_4weeks -> 0.017157458672051297
lag_2weeks -> 0.012598757073451606
lag_5days -> 0.009642874413557783
lag_1week -> 0.004671448769726507
year -> 6.729525772282224E-4
--------------------------------------------
rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_73dd29f69ef6
pipeline: org.apache.spark.ml.Pipeline = pipeline_b0cc7823e867
```

```
// train a new random forest model
val rf = new RandomForestRegressor()

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_1week)

// Make predictions.
val predictions = model.transform(test_1week)

// extract feature importance from rf model
val importances = model // this would be your trained model
  .stages(0)
  .asInstanceOf[RandomForestRegressionModel]
  .featureImportances

val features = train_set_1week.columns.filterNot(Set("Date", "label"))

val printImportance = importances.toArray zip features

println("--------------------------------------------")
printImportance.sortBy(-_._1).foreach(x => println(x._2 + " -> " + x._1))
println("--------------------------------------------")
```

```
--------------------------------------------
ma_3weeks -> 0.14297118005146908
std_4weeks -> 0.11531770637488058
ma_4weeks -> 0.10983726573915151
std_3weeks -> 0.08312879700344797
ma_2weeks -> 0.07877991485282489
std_1week -> 0.07132787062176611
day -> 0.0656406692045842
std_2weeks -> 0.06134166428920447
lag_2weeks -> 0.045426546768668856
lag_1week -> 0.0419079827888783
ma_1week -> 0.0377768375249901
diff_sales -> 0.03698123003983979
lag_3weeks -> 0.036123782021939894
lag_4weeks -> 0.031870373446634834
month -> 0.026683155693142144
year -> 0.014885023578577324
--------------------------------------------
rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_16ef4f97297e
pipeline: org.apache.spark.ml.Pipeline = pipeline_7188d97736a9
model: org.apache.spark.ml.PipelineModel = pipeline_7188d97736a9
```

```
// train a new random forest model
val rf = new RandomForestRegressor()

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_2weeks)

// Make predictions.
val predictions = model.transform(test_2weeks)

// extract feature importance from rf model
val importances = model // this would be your trained model
  .stages(0)
  .asInstanceOf[RandomForestRegressionModel]
  .featureImportances

val features = train_set_2weeks.columns.filterNot(Set("Date", "label"))

val printImportance = importances.toArray zip features

println("--------------------------------------------")
printImportance.sortBy(-_._1).foreach(x => println(x._2 + " -> " + x._1))
println("--------------------------------------------")
```

```
--------------------------------------------
std_2weeks -> 0.12014393168914063
std_4weeks -> 0.11224311104748723
month -> 0.11149736753773483
std_3weeks -> 0.1010823833998608
std_5weeks -> 0.0943050119063537
ma_3weeks -> 0.07536959152335206
lag_2weeks -> 0.0741888280443994
lag_5weeks -> 0.06874577445768497
ma_4weeks -> 0.04604026893232798
diff_sales -> 0.04301176435641 03
ma_5weeks -> 0.04292575164265888
day -> 0.04100093498460487
lag_3weeks -> 0.024817185814592174
ma_2weeks -> 0.01927680473618801
year -> 0.013071106755248236
lag_4weeks -> 0.012280183171955896
--------------------------------------------
```

```
rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_708a85aa1c97
pipeline: org.apache.spark.ml.Pipeline = pipeline_a6859af0b329
model: org.apache.spark.ml.PipelineModel = pipeline_a6859af0b329
```

```
// train a new random forest model
val rf = new RandomForestRegressor()

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_3weeks)

// Make predictions.
val predictions = model.transform(test_3weeks)

// extract feature importance from rf model
val importances = model // this would be your trained model
  .stages(0)
  .asInstanceOf[RandomForestRegressionModel]
  .featureImportances

val features = train_set_3weeks.columns.filterNot(Set("Date", "label"))

val printImportance = importances.toArray zip features

println("-----------------------------------------")
printImportance.sortBy(-_._1).foreach(x => println(x._2 + " -> " + x._1))
println("-----------------------------------------")
```

```
-----------------------------------------
ma_5weeks -> 0.13760178957748456
std_5weeks -> 0.12346492078430375
month -> 0.12264478979745258
day -> 0.08979584121057345
diff_sales -> 0.074325864511961
std_6weeks -> 0.07180492238726471
lag_6weeks -> 0.06952304944491863
lag_4weeks -> 0.04831830808648697
ma_4weeks -> 0.048018921649250125
ma_3weeks -> 0.04295242716980618
std_3weeks -> 0.04203999859962835
std_4weeks -> 0.03863833689847181
ma_6weeks -> 0.02940899557792176
lag_5weeks -> 0.027140143955199095
year -> 0.01939556239743649
lag_3weeks -> 0.014926127950544578
-----------------------------------------
rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_955aa5a165b3
pipeline: org.apache.spark.ml.Pipeline = pipeline_350eac7578ca
model: org.apache.spark.ml.PipelineModel = pipeline_350eac7578ca
```

```
// train a new random forest model
val rf = new RandomForestRegressor()

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_1month)

// Make predictions.
val predictions = model.transform(test_1month)

// extract feature importance from rf model
val importances = model // this would be your trained model
  .stages(0)
  .asInstanceOf[RandomForestRegressionModel]
  .featureImportances

val features = train_set_1month.columns.filterNot(Set("Date", "label"))

val printImportance = importances.toArray zip features

println("-----------------------------------------")
printImportance.sortBy(-_._1).foreach(x => println(x._2 + " -> " + x._1))
println("-----------------------------------------")
```

```
-----------------------------------------
ma_7weeks -> 0.17412154735552611
month -> 0.10868264617311027
diff_sales -> 0.09121351572812682
std_6weeks -> 0.08713885371589519
ma_6weeks -> 0.08292119712732847
std_7weeks -> 0.0723971257137345
std_4weeks -> 0.06638631355713298
ma_4weeks -> 0.06032910040294176
lag_6weeks -> 0.05535892082906385
ma_5weeks -> 0.05247840018408913
day -> 0.03516064153188055
std_5weeks -> 0.03248197169560493
year -> 0.029019676012196134
lag_7weeks -> 0.025055989537776793
lag_4weeks -> 0.017500266594483 41
lag_5weeks -> 0.009753833841374184
-----------------------------------------
rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_6b5518f7342d
pipeline: org.apache.spark.ml.Pipeline = pipeline_7c07b48690da
model: org.apache.spark.ml.PipelineModel = pipeline_7c07b48690da
```

Simple & Moving Average (SMA)

```
// def getLastRow(df: DataFrame): DataFrame = {
//   val with_id = df.withColumn("_id", monotonically_increasing_id())
//   val i = with_id.select(max("_id")).first()(0)
// //   val i = with_id.count().toInt - 1
//   return with_id.where(col("_id") === i).drop("_id")
// }
```

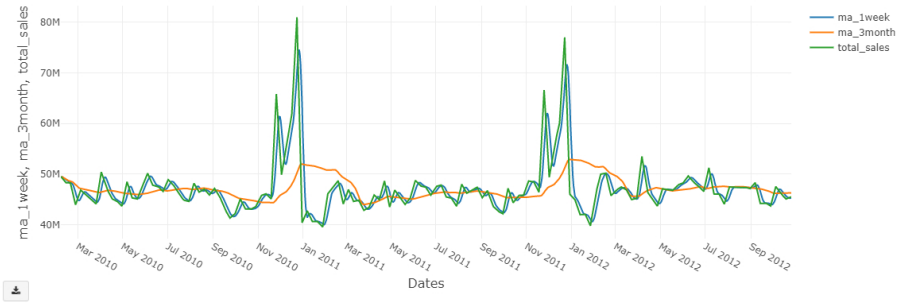
```
// def SMA_predict(df: DataFrame, prediction_len: Int, win_num: Int): DataFrame = {
//   /**
//   * df contains cols: Dates & total_sales
//   */
//   val clean_df: DataFrame = df.select("Dates","total_sales")

//   val w = Window.orderBy(col("Dates")).rowsBetween(-win_num, -1)

// //   val newRow1 = Seq((clean_df.tail(1)(0)(0) + 1, 0)).toDF("Dates","total_sales")
//   val newRow1 = getLastRow(clean_df).withColumn("Dates", date_add($"Dates", 1)).withColumn("total_sales", lit(0))

//   val df_apended = clean_df.union(newRow1)
```

```
//   var ma_1: DataFrame = df_apended.withColumn("ma_1", avg("total_sales").over(w))

//   ma_1 = ma_1.withColumn("total_sales", when(df_apended.col("total_sales") === 0, ma_1.tail(2)(0)(2)).
//             otherwise(df_apended.col("total_sales")))

//   println(df_apended.count().toInt)

//   if(df_apended.count().toInt >= prediction_len) {
//     return ma_1
//   } else {
//     SMA_predict(ma_1, prediction_len, win_num)
//   }

// }
```

```
val week_w = Window.orderBy(col("Dates")).rowsBetween(-7, -1)
val biweekly_w = Window.orderBy(col("Dates")).rowsBetween(-14, -1)
val triweekly_w = Window.orderBy(col("Dates")).rowsBetween(-21, -1)
val monthly_w = Window.orderBy(col("Dates")).rowsBetween(-28, -1)
val trimonth_w = Window.orderBy(col("Dates")).rowsBetween(-84, -1)
```

```
week_w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@21712a30
biweekly_w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@41b499b5
triweekly_w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@4711620e
monthly_w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@7590e6af
trimonth_w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@5d186287
```

```
// Create a comparison dataframe to show 1-week SMA versus 3-month SMA
val compare_ma = df_sale1
  .withColumn("ma_1week", avg("total_sales").over(week_w))
  .withColumn("ma_3month", avg("total_sales").over(trimonth_w))

display(compare_ma)
```




```
// Splitting the original data to get test set before calculating SMA
val (train_set_wma, test_set_wma) = ts_split(0.8, total_sales_hms)

train_set_wma: org.apache.spark.sql.DataFrame = [total_sales: double, Week: bigint ... 1 more field]
test_set_wma: org.apache.spark.sql.DataFrame = [total_sales: double, Week: bigint ... 1 more field]
```

```
val df_sales_wid = df_sales
  .withColumn("Day", monotonically_increasing_id)

display(df_sales_wid)
```

	Dates	total_sales	Day
1	2010-02-06 00:00:00	49548731.51	0
2	2010-02-07 00:00:00	49346722.54	1
3	2010-02-08 00:00:00	49144713.56	2
4	2010-02-09 00:00:00	48942704.58	3
5	2010-02-10 00:00:00	48740695.61	4
6	2010-02-11 00:00:00	48538686.63	5
7	2010-02-12 00:00:00	48336677.65	6
8	2010-02-13 00:00:00	48328151.39	7

Showing all 994 rows.

```
// Splitting the interpolated data to get test set before calculating SMA
val (train_set_dma, test_set_dma) = ts_split(0.8, df_sales_wid)

train_set_dma: org.apache.spark.sql.DataFrame = [Dates: string, total_sales: double ... 1 more field]
test_set_dma: org.apache.spark.sql.DataFrame = [Dates: string, total_sales: double ... 1 more field]
```

```
// Using interpolated data to calculate SMA
val ma_1week = test_set_dma
  .withColumn("ma_1week", avg("total_sales").over(week_w))
  .where("id > 802")
  .drop("Dates")

ma_1week.
  withColumn("diff_abs", abs($"ma_1week" - $"total_sales")).
  withColumn("demo", (abs($"ma_1week") + abs($"total_sales")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / ma_1week.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|1.4477|
+------+

ma_1week: org.apache.spark.sql.DataFrame = [total_sales: double, Day: bigint ... 1 more field]
```

```
val ma_2weeks = test_set_dma
  .withColumn("ma_2weeks", avg("total_sales").over(biweekly_w))
  .where("id > 809")
  .drop("Dates")

ma_2weeks.
  withColumn("diff_abs", abs($"ma_2weeks" - $"total_sales")).
  withColumn("demo", (abs($"ma_2weeks") + abs($"total_sales")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / ma_2weeks.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
```

```
| SMAPE|
+------+
|2.1174|
+------+

ma_2weeks: org.apache.spark.sql.DataFrame = [total_sales: double, Day: bigint ... 1 more field]
```

```
val ma_3weeks = test_set_dma
  .withColumn("ma_3weeks", avg("total_sales").over(triweekly_w))
  .where("id > 816")
  .drop("Dates")

ma_3weeks.
  withColumn("diff_abs", abs($"ma_3weeks" - $"total_sales")).
  withColumn("demo", (abs($"ma_3weeks") + abs($"total_sales")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / ma_3weeks.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|2.5212|
+------+

ma_3weeks: org.apache.spark.sql.DataFrame = [total_sales: double, Day: bigint ... 1 more field]
```

```
val ma_1month = test_set_dma
  .withColumn("ma_1month", avg("total_sales").over(monthly_w))
  .where("id > 823")
  .drop("Dates")

ma_1month.
  withColumn("diff_abs", abs($"ma_1month" - $"total_sales")).
  withColumn("demo", (abs($"ma_1month") + abs($"total_sales")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / ma_1month.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|2.5934|
+------+

ma_1month: org.apache.spark.sql.DataFrame = [total_sales: double, Day: bigint ... 1 more field]
```

```
val twoweek_w = Window.orderBy(col("Week")).rowsBetween(-2, -1)
val threeweek_w = Window.orderBy(col("Week")).rowsBetween(-3, -1)
val onemonth_w = Window.orderBy(col("Week")).rowsBetween(-4, -1)

twoweek_w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@3ec6ce32
threeweek_w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@1c632d6
onemonth_w: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@64bb1378
```

```
// Using original data to calculate SMA
val ma_2w = test_set_wma
  .drop("Dates")
  .withColumn("ma_2weeks", avg("total_sales").over(twoweek_w))
  .where("Week > 115")

ma_2w.
  withColumn("diff_abs", abs($"ma_2weeks" - $"total_sales")).
  withColumn("demo", (abs($"ma_2weeks") + abs($"total_sales")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / ma_2w.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|3.2855|
+------+

ma_2w: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [total_sales: double, Week: bigint ... 1 more field]
```

```
val ma_3w = test_set_wma
  .drop("Dates")
  .withColumn("ma_3weeks", avg("total_sales").over(threeweek_w))
  .where("Week > 116")

ma_3w.
  withColumn("diff_abs", abs($"ma_3weeks" - $"total_sales")).
  withColumn("demo", (abs($"ma_3weeks") + abs($"total_sales")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / ma_3w.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|3.5817|
+------+

ma_3w: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [total_sales: double, Week: bigint ... 1 more field]
```

```
val ma_1m = test_set_wma
  .drop("Dates")
  .withColumn("ma_1month", avg("total_sales").over(onemonth_w))
  .where("Week > 117")

ma_1m.
  withColumn("diff_abs", abs($"ma_1month" - $"total_sales")).
  withColumn("demo", (abs($"ma_1month") + abs($"total_sales")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / ma_1m.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|3.3261|
+------+

ma_1m: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [total_sales: double, Week: bigint ... 1 more field]
```

Random Forest Models Predictions, Reconstructions, and SMAPE

```
def rolling_cv_tuning(initial_train_obs: Int, shift: Int, assembled_df: DataFrame, hyper_params: Map[String, String]): Double = {
  /**
    * Returns the average rmse for the whole rolling process
```

```
    * For example, if we have 143 rows in total,
    * we set the initial train as 43, and 10 more rolling forward, then
    * train first 43, test 44 - 53
    * train first 53, test 54 - 64
    * ...
    * train first 133, test 134 - 143
    * the function will roll 9 times in this case, and we calculate the rmse for each time / 9, and return the value
    */

  val total_rows = assembled_df.count()
  val rolling_space = total_rows - initial_train_obs
  val fold_num = (rolling_space / shift).toInt


  var total_rmse = 0.0

  for (i <- 1 to fold_num) {

    // define rolling frame
    var total_select = initial_train_obs + shift + i
    var train_pct = initial_train_obs.toFloat / total_select

    // train test split
    val (train_set, test_set) = ts_split(train_pct, assembled_df.limit(total_select.toInt))

    /********************REPLACE TO OTHER MODEL START *********************/
    // Initiate model object
    val rf = new RandomForestRegressor().
      setMaxBins(hyper_params("MaxBins").toInt).
      setNumTrees(hyper_params("NumTrees").toInt)

    // Setup Pipeline.
    val pipeline = new Pipeline().setStages(Array(rf))

    // Train model
    val model = pipeline.fit(train_set)

    // Make predictions.
    val predictions = model.transform(test_set)

    // Select (prediction, true label) and compute test error.
    val evaluator = new RegressionEvaluator()
      .setMetricName("rmse")

    val rmse = evaluator.evaluate(predictions)

    // trained model
    val rfModel = model.stages(0).asInstanceOf[RandomForestRegressionModel]
    /********************REPLACE TO OTHER MODEL END *********************/

    total_rmse += rmse

    println(s"Hyper params: $hyper_params; rolling times: $i ($total_select/$total_rows); rmse: $rmse")
  }

  return total_rmse / fold_num

}

rolling_cv_tuning: (initial_train_obs: Int, shift: Int, assembled_df: org.apache.spark.sql.DataFrame, hyper_params: Map[String,String])Double


// Setting up for rolling cv
val param_grid = Map(
  "NumTrees" -> Seq("5","10","15"),
  "MaxBins" -> Seq("28","30","32")
)

// transform arrays into lists with values paired with map key
val pairedWithKey = param_grid.map { case (k,v) => v.map(i => k -> i).toList }

val accumulator = pairedWithKey.head.map(x => Vector(x))

val params_combination = pairedWithKey.tail.foldLeft(accumulator)( (acc, elem) =>
  for { x <- acc; y <- elem } yield x :+ y
)

param_grid: scala.collection.immutable.Map[String,Seq[String]] = Map(NumTrees -> List(5, 10, 15), MaxBins -> List(28, 30, 32))
pairedWithKey: scala.collection.immutable.Iterable[List[(String, String)]] = List(List((NumTrees,5), (NumTrees,10), (NumTrees,15)), List((MaxBins,28), (MaxBins,30), (MaxBins,32)))
accumulator: List[scala.collection.immutable.Vector[(String, String)]] = List(Vector((NumTrees,5)), Vector((NumTrees,10)), Vector((NumTrees,15)))
params_combination: List[scala.collection.immutable.Vector[(String, String)]] = List(Vector((NumTrees,5), (MaxBins,28)), Vector((NumTrees,5), (MaxBins,30)), Vector((NumTrees,5), (MaxBins,32)), Vector((NumTrees,10), (MaxBins,28)), Vector((NumTrees,10), (MaxBins,30)), Vector((NumTrees,10), (MaxBins,32)), Vector((NumTrees,15), (MaxBins,28)), Vector((NumTrees,15), (MaxBins,30)), Vector((NumTrees,15), (MaxBins,32)))
```

Prediction Window = 1 day

```
//1 day rolling cv
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(64, 200, train_1day, hyper_params) //train1day has 464 rows, use 64 as train and every 200 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("----------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("----------------------------------------------------")
```

```
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 1 (264/464); rmse: 620859.0764262772
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 2 (464/464); rmse: 857982.2917266006
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 1 (264/464); rmse: 560581.3740698603
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 2 (464/464); rmse: 814843.2950235651
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 1 (264/464); rmse: 609296.9982348494
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 2 (464/464); rmse: 905068.6811920457
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 1 (264/464); rmse: 494907.80525327433
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 2 (464/464); rmse: 849945.4559216195
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 1 (264/464); rmse: 494641.3242002952
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 2 (464/464); rmse: 759748.9666059434
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 1 (264/464); rmse: 479925.5526462603
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 2 (464/464); rmse: 728874.2480037136
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 1 (264/464); rmse: 500672.02343810216
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 2 (464/464); rmse: 772947.8542815064
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 1 (264/464); rmse: 480671.9334075607
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 2 (464/464); rmse: 806515.7239681443
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 1 (264/464); rmse: 485204.08841444645
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 2 (464/464); rmse: 770674.9187805974
----------------------------------------------------
Best Hyper Parameter is:
Some((604399.9003249869,Map(NumTrees -> 10, MaxBins -> 32)))
```

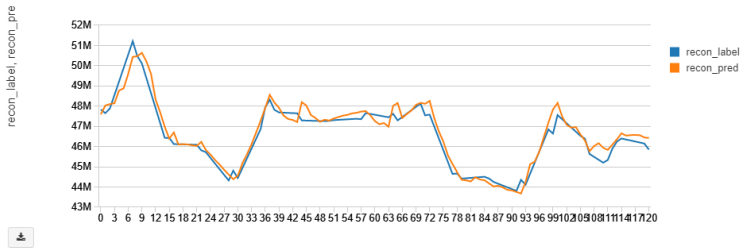
```
// Using best parameters from CV
// Best hyperparameters found: NumTrees = 10, MaxBins = 32
val rf = new RandomForestRegressor()
```

```
    .setFeaturesCol("features")
    .setNumTrees(10)
    .setMaxBins(32)

// Setup Pipeline
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_1day)

// Make predictions.
val predictions = model.transform(test_1day)

// Reconstruction of predictions with seasonal differences
val diff_1day = test_diff_1day.withColumn("id", monotonically_increasing_id())
val pred_1day = predictions.withColumn("id", monotonically_increasing_id())
val merged_1day = pred_1day.join(diff_1day, diff_1day.col("id") === pred_1day.col("id"), "left_outer").drop("id")
val recon_predictions_1day = merged_1day.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_1day.
//   limit(120).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1day.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|0.6322|
+------+
```

```
rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_b64e5c3b5ced
pipeline: org.apache.spark.ml.Pipeline = pipeline_80d2a3b65ba2
model: org.apache.spark.ml.PipelineModel = pipeline_80d2a3b65ba2
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1day: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
display(recon_predictions_1day)
```



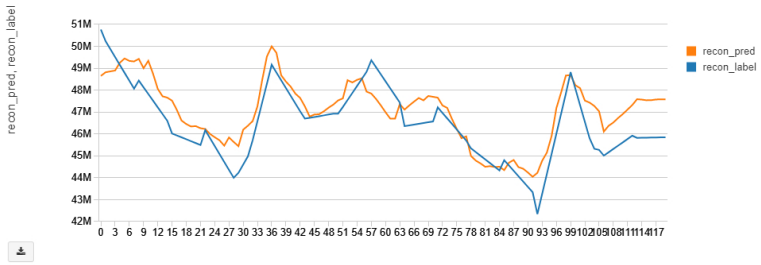
Prediction Window = 1 week

```
// 1 week rolling cv
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(65, 190, train_1week, hyper_params) //trainlweek has 445 rows, use 65 as train and every 190 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("------------------------------------------------")
```

```
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 1 (255/445); rmse: 1626974.9174286106
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 2 (445/445); rmse: 1655955.2154380255
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 1 (255/445); rmse: 1624233.0207597497
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 2 (445/445); rmse: 1708655.5801201407
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 1 (255/445); rmse: 1520737.765905241
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 2 (445/445); rmse: 1710527.9479596484
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 1 (255/445); rmse: 1630032.38157273
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 2 (445/445); rmse: 1633162.5329673588
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 1 (255/445); rmse: 1513333.590161624
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 2 (445/445); rmse: 1639287.2683525262
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 1 (255/445); rmse: 1468493.6879179259
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 2 (445/445); rmse: 1603034.319464999
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 1 (255/445); rmse: 1584928.8369680813
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 2 (445/445); rmse: 1657170.957480294
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 1 (255/445); rmse: 1536626.040784858
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 2 (445/445); rmse: 1602869.162096915
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 1 (255/445); rmse: 1650624.9049826586
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 2 (445/445); rmse: 1660664.3179522834
------------------------------------------------
Best Hyper Parameter is:
Some((604399.9003249869,Map(NumTrees -> 10, MaxBins -> 32)))
```

```
// Using best parameters from CV
// Best hyperparameters found: NumTrees = 10, MaxBins = 32
val rf = new RandomForestRegressor()
    .setFeaturesCol("features")
    .setNumTrees(10)
    .setMaxBins(32)

// Setup Pipeline
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_1week)

// Make predictions.
val predictions = model.transform(test_1week)

// Reconstruction of predictions with seasonal differences
val diff_1week = test_diff_1week.withColumn("id", monotonically_increasing_id())
val pred_1week = predictions.withColumn("id", monotonically_increasing_id())
val merged_1week = pred_1week.join(diff_1week, diff_1week.col("id") === pred_1week.col("id"), "left_outer").drop("id")
```

```
val recon_predictions_1week = merged_1week.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_1week.
//   limit(113).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1week.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
|1.8343|
+------+

rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_fbf2f545df91
pipeline: org.apache.spark.ml.Pipeline = pipeline_f51db3666d8f
model: org.apache.spark.ml.PipelineModel = pipeline_f51db3666d8f
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1week: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]

display(recon_predictions_1week)
```



Prediction Window = 2 weeks

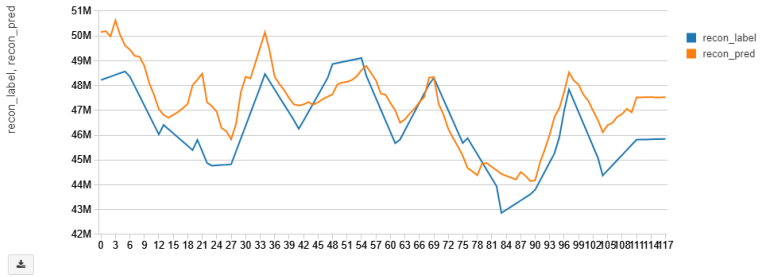
```
// 2 weeks rolling cv
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(66, 180, train_2weeks, hyper_params) //train2weeks has 426 rows, use 66 as train and every 180 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("------------------------------------------------")
```

```
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 1 (246/426); rmse: 2032996.0503821538
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 2 (426/426); rmse: 1838340.8104556703
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 1 (246/426); rmse: 2191152.188600683
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 2 (426/426); rmse: 1966826.603888683
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 1 (246/426); rmse: 2297145.5981749785
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 2 (426/426); rmse: 2023931.515368753
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 1 (246/426); rmse: 1942923.4221352485
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 2 (426/426); rmse: 1800295.2516126814
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 1 (246/426); rmse: 2104564.140242831
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 2 (426/426); rmse: 1896566.7272125285
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 1 (246/426); rmse: 2208697.0685660676
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 2 (426/426); rmse: 1955308.8539766665
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 1 (246/426); rmse: 2134589.0167023167
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 2 (426/426); rmse: 1917168.7944642229
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 1 (246/426); rmse: 2235609.4116415083
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 2 (426/426); rmse: 1966338.6331891166
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 1 (246/426); rmse: 2142918.2009635773
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 2 (426/426); rmse: 1937630.3321122394
------------------------------------------------
Best Hyper Parameter is:
Some((604399.9003249869,Map(NumTrees -> 10, MaxBins -> 32)))
```

```
// Using best parameters from CV
// Best hyperparameters found: NumTrees = 10, MaxBins = 32
val rf = new RandomForestRegressor()
    .setFeaturesCol("features")
    .setNumTrees(10)
    .setMaxBins(32)

// Setup Pipeline
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_2weeks)

// Make predictions.
val predictions = model.transform(test_2weeks)

// Reconstruction of predictions with seasonal differences
val diff_2weeks = test_diff_2weeks.withColumn("id", monotonically_increasing_id())
val pred_2weeks = predictions.withColumn("id", monotonically_increasing_id())
val merged_2weeks = pred_2weeks.join(diff_2weeks, diff_2weeks.col("id") === pred_2weeks.col("id"), "left_outer").drop("id")
val recon_predictions_2weeks = merged_2weeks.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_2weeks.
//   limit(104).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_2weeks.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
```

```
|2.3636|
+------+

rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_dc619575cb77
pipeline: org.apache.spark.ml.Pipeline = pipeline_a1803a098d64
model: org.apache.spark.ml.PipelineModel = pipeline_a1803a098d64
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_2weeks: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
display(recon_predictions_2weeks)
```



Prediction Window = 3 weeks

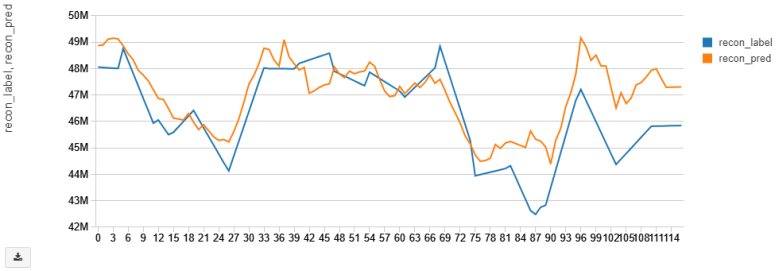
```
// 3 weeks rolling cv
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(60, 173, train_3weeks, hyper_params) //train3weeks has 406 rows, use 60 as train and every 173 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("------------------------------------------------")
```

```
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 1 (233/406); rmse: 1115504.9406377166
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 2 (406/406); rmse: 1535782.0316430603
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 1 (233/406); rmse: 1105164.2764877197
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 2 (406/406); rmse: 1528492.7466532318
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 1 (233/406); rmse: 1108105.211926003
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 2 (406/406); rmse: 1534134.4008604405
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 1 (233/406); rmse: 1078664.262309281
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 2 (406/406); rmse: 1436895.1768710448
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 1 (233/406); rmse: 1053052.7050330648
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 2 (406/406); rmse: 1440302.3802055777
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 1 (233/406); rmse: 1075363.0156288454
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 2 (406/406); rmse: 1442199.136047611
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 1 (233/406); rmse: 1076536.083586833
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 2 (406/406); rmse: 1469019.1180838281
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 1 (233/406); rmse: 1081900.3755640795
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 2 (406/406); rmse: 1460872.401785882
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 1 (233/406); rmse: 1061357.0648553052
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 2 (406/406); rmse: 1444741.8417268263
------------------------------------------------
Best Hyper Parameter is:
Some((1246677.5426193213,Map(NumTrees -> 10, MaxBins -> 30)))
```

```
// Using best parameters from CV
// Best hyperparameters found: NumTrees = 10, MaxBins = 30
val rf = new RandomForestRegressor()
    .setFeaturesCol("features")
    .setNumTrees(10)
    .setMaxBins(30)

// Setup Pipeline
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_3weeks)

// Make predictions.
val predictions = model.transform(test_3weeks)

// Reconstruction of predictions with seasonal differences
val diff_3weeks = test_diff_3weeks.withColumn("id", monotonically_increasing_id())
val pred_3weeks = predictions.withColumn("id", monotonically_increasing_id())
val merged_3weeks = pred_3weeks.join(diff_3weeks, diff_3weeks.col("id") === pred_3weeks.col("id"), "left_outer").drop("id")
val recon_predictions_3weeks = merged_3weeks.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_3weeks.
//   limit(96).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_3weeks.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|2.0982|
+------+

rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_5b05fe442cc6
pipeline: org.apache.spark.ml.Pipeline = pipeline_0c409cc38806
model: org.apache.spark.ml.PipelineModel = pipeline_0c409cc38806
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_3weeks: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
display(recon_predictions_3weeks)
```



Prediction Window = 1 month

```
// 1 month rolling cv
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(57, 165, train_1month, hyper_params) //train1month has 387 rows, use 57 as train and every 165 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")
```

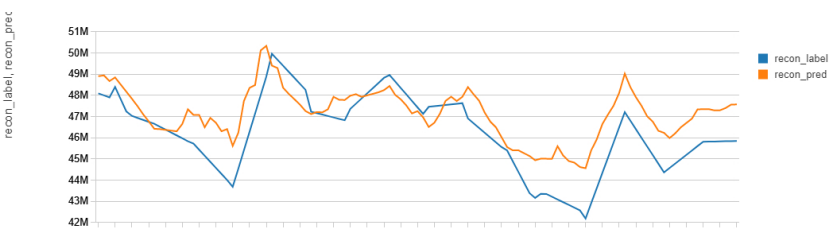
```
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 1 (222/387); rmse: 1159546.6602385603
Hyper params: Map(NumTrees -> 5, MaxBins -> 28); rolling times: 2 (387/387); rmse: 1526076.7099672586
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 1 (222/387); rmse: 1122838.9037959273
Hyper params: Map(NumTrees -> 5, MaxBins -> 30); rolling times: 2 (387/387); rmse: 1500109.7446752638
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 1 (222/387); rmse: 1110729.0687340582
Hyper params: Map(NumTrees -> 5, MaxBins -> 32); rolling times: 2 (387/387); rmse: 1499796.6365522065
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 1 (222/387); rmse: 1082351.9674277033
Hyper params: Map(NumTrees -> 10, MaxBins -> 28); rolling times: 2 (387/387); rmse: 1490172.5704682488
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 1 (222/387); rmse: 1113978.523655678
Hyper params: Map(NumTrees -> 10, MaxBins -> 30); rolling times: 2 (387/387); rmse: 1501554.8941543547
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 1 (222/387); rmse: 1083058.9504865713
Hyper params: Map(NumTrees -> 10, MaxBins -> 32); rolling times: 2 (387/387); rmse: 1486557.509180841
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 1 (222/387); rmse: 1125069.5345799893
Hyper params: Map(NumTrees -> 15, MaxBins -> 28); rolling times: 2 (387/387); rmse: 1509942.821196458
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 1 (222/387); rmse: 1087929.164532966
Hyper params: Map(NumTrees -> 15, MaxBins -> 30); rolling times: 2 (387/387); rmse: 1490559.4320660967
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 1 (222/387); rmse: 1134376.8155980576
Hyper params: Map(NumTrees -> 15, MaxBins -> 32); rolling times: 2 (387/387); rmse: 1515116.5705763726
-------------------------------------------------
Best Hyper Parameter is:
Some((1284808.2298337063,Map(NumTrees -> 10, MaxBins -> 32)))
```

```
// Using best parameters from CV
// Best hyperparameters found: NumTrees = 10, MaxBins = 32
val rf = new RandomForestRegressor()
    .setFeaturesCol("features")
    .setNumTrees(10)
    .setMaxBins(32)

// Setup Pipeline
val pipeline = new Pipeline().setStages(Array(rf))

// Train model
val model = pipeline.fit(train_1month)

// Make predictions.
val predictions = model.transform(test_1month)

// Reconstruction of predictions with seasonal differences
val diff_1month = test_diff_1month.withColumn("id", monotonically_increasing_id())
val pred_1month = predictions.withColumn("id", monotonically_increasing_id())
val merged_1month = pred_1month.join(diff_1month, diff_1month.col("id") === pred_1month.col("id"), "left_outer").drop("id")
val recon_predictions_1month = merged_1month.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_1month.
//   limit(87).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1month.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|2.4604|
+------+
```

```
rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_1a2439fe4077
pipeline: org.apache.spark.ml.Pipeline = pipeline_6da9b6a9301f
model: org.apache.spark.ml.PipelineModel = pipeline_6da9b6a9301f
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1month: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
display(recon_predictions_1month)
```



0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78 81 84 87 90 93 96 99 102105108111114

Linear Regression Models Predictions, Reconstructions, and SMAPE

```
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.regression.LinearRegressionModel
def rolling_cv_tuning(initial_train_obs: Int, shift: Int, assembled_df: DataFrame, hyper_params: Map[String, String]): Double = {

  val total_rows = assembled_df.count()
  val rolling_space = total_rows - initial_train_obs
  val fold_num = (rolling_space / shift).toInt


  var total_rmse = 0.0

  for (i <- 1 to fold_num) {

    // define rolling frame
    var total_select = initial_train_obs + shift * i
    var train_pct = initial_train_obs.toFloat / total_select

    // train test split
    val (train_set, test_set) = ts_split(train_pct, assembled_df.limit(total_select.toInt))

    /********************REPLACE TO OTHER MODEL START *********************/
    // Initiate model object
    val lr = new LinearRegression()
      .setRegParam(hyper_params("RegParam").toFloat)
      .setElasticNetParam(hyper_params("ElasticNetParam").toFloat)
      .setLabelCol("label")
      .setFeaturesCol("features")

    // Setup Pipeline.
    val pipeline = new Pipeline().setStages(Array(lr))

    // Train model
    val model = pipeline.fit(train_set)

    // Make predictions.
    val predictions = model.transform(test_set)

    // Select (prediction, true label) and compute test error.
    val evaluator = new RegressionEvaluator()
      .setMetricName("rmse")

    val rmse = evaluator.evaluate(predictions)

    // trained model
    val lrModel = model.stages(0).asInstanceOf[LinearRegressionModel]
    /********************REPLACE TO OTHER MODEL END *********************/

    total_rmse += rmse

    println(s"Hyper params: $hyper_params; rolling times: $i ($total_select/$total_rows); rmse: $rmse")
  }

  return total_rmse / fold_num

}
```

```
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.regression.LinearRegressionModel
rolling_cv_tuning: (initial_train_obs: Int, shift: Int, assembled_df: org.apache.spark.sql.DataFrame, hyper_params: Map[String,String])Double
```

```
// Setting up for rolling cv
val param_grid = Map(
  "RegParam" -> Seq("0","0.1","0.5"),
  "ElasticNetParam" -> Seq("0","0.1","0.5")
)

// transform arrays into lists with values paired with map key
val pairedWithKey = param_grid.map { case (k,v) => v.map(i => k -> i).toList }

val accumulator = pairedWithKey.head.map(x => Vector(x))

val params_combination = pairedWithKey.tail.foldLeft(accumulator)( (acc, elem) =>
  for { x <- acc; y <- elem } yield x :+ y
)
```

```
param_grid: scala.collection.immutable.Map[String,Seq[String]] = Map(RegParam -> List(0, 0.1, 0.5), ElasticNetParam -> List(0, 0.1, 0.5))
pairedWithKey: scala.collection.immutable.Iterable[List[(String, String)]] = List(List((RegParam,0), (RegParam,0.1), (RegParam,0.5)), List((ElasticNetParam,0), (ElasticNetParam,0.1), (ElasticNetParam,0.5)))
accumulator: List[scala.collection.immutable.Vector[(String, String)]] = List(Vector((RegParam,0)), Vector((RegParam,0.1)), Vector((RegParam,0.5)))
params_combination: List[scala.collection.immutable.Vector[(String, String)]] = List(Vector((RegParam,0), (ElasticNetParam,0)), Vector((RegParam,0), (ElasticNetParam,0.1)), Vector((RegParam,0), (ElasticNetParam,0.5)), Vector((RegParam,0.
1), (ElasticNetParam,0)), Vector((RegParam,0.1), (ElasticNetParam,0.1)), Vector((RegParam,0.1), (ElasticNetParam,0.5)), Vector((RegParam,0.5), (ElasticNetParam,0)), Vector((RegParam,0.5), (ElasticNetParam,0.1)), Vector((RegParam,0.5), (Ela
sticNetParam,0.5)))
```

Prediction Window = 1 day

```
//1 day rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(64, 200, train_1day, hyper_params) //trainlday has 464 rows, use 64 as train and every 200 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")
```

```
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 1 (264/464); rmse: 5533072.012463968
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 2 (464/464); rmse: 4690550.726433968
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 1 (264/464); rmse: 5533072.012463968
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 2 (464/464); rmse: 4690550.726433968
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 1 (264/464); rmse: 5533072.012463968
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 2 (464/464); rmse: 4690550.726433968
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 1 (264/464); rmse: 6867715.245802469
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 2 (464/464); rmse: 6043765.306474306
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 1 (264/464); rmse: 1928471.2935608719
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 2 (464/464); rmse: 1875698.1505265783
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 1 (264/464); rmse: 2075276.1600215451
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 2 (464/464); rmse: 1966781.0504926462
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 1 (264/464); rmse: 6863676.577312397
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 2 (464/464); rmse: 6040384.562850681
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 1 (264/464); rmse: 2075257.4787815535
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 2 (464/464); rmse: 1966768.7907371665
```

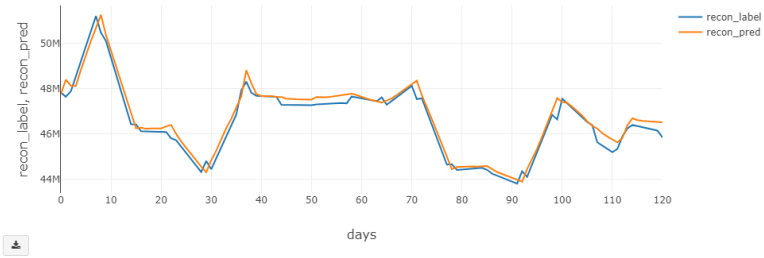
```
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 1 (264/464); rmse: 2075262.472327159
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 2 (464/464); rmse: 1966719.1272153175
--------------------------------------------------
Best Hyper Parameter is:
Some((1902084.7220437252,Map(RegParam -> 0.1, ElasticNetParam -> 0.1)))
```

```
//1 day best cv settings
//best hyperparameters found: RegParam = 0.1, ElasticNetParam = 0.1
val lr = new LinearRegression()
      .setRegParam(0.1)
      .setElasticNetParam(0.1)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(lr))

// Train model
val model = pipeline.fit(train_1day)

// Make predictions.
val predictions = model.transform(test_1day)
// Reconstruction of predictions with seasonal differences
val diff_1day = test_diff_1day.withColumn("id", monotonically_increasing_id())
val pred_1day = predictions.withColumn("id", monotonically_increasing_id())
val merged_1day = pred_1day.join(diff_1day, diff_1day.col("id") === pred_1day.col("id"), "left_outer").drop("id")
val recon_predictions_1day = merged_1day.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_1day.
//   limit(120).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1day.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|0.5562|
+------+
```

```
lr: org.apache.spark.ml.regression.LinearRegression = linReg_4ef17f32e3c9
pipeline: org.apache.spark.ml.Pipeline = pipeline_4b44af41446a
model: org.apache.spark.ml.PipelineModel = pipeline_4b44af41446a
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1day: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
//1 day prediction horizon
display(recon_predictions_1day)
```



Prediction Window = 1 week

```
//1 week rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(65, 190, train_1week, hyper_params) //train1week has 445 rows, use 65 as train and every 190 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("--------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("--------------------------------------------------")
```

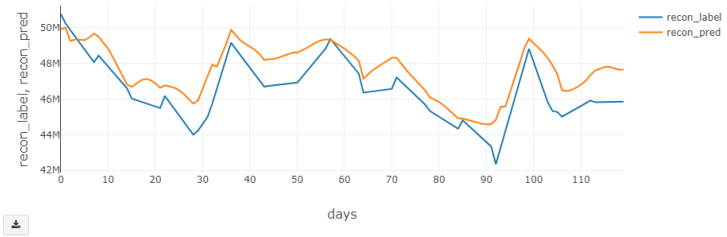
```
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 1 (255/445); rmse: 2.912203394643107E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 2 (445/445); rmse: 2.37255572326513 94E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 1 (255/445); rmse: 2.912203394643107E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 2 (445/445); rmse: 2.37255572326513 94E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 1 (255/445); rmse: 2.912203394643107E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 2 (445/445); rmse: 2.37255572326513 94E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 1 (255/445); rmse: 3.1017546188208353E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 2 (445/445); rmse: 3.301334427802443E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 1 (255/445); rmse: 1.3958009819878759E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 2 (445/445); rmse: 1.1083859288961306E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 1 (255/445); rmse: 1.3958013474273004E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 2 (445/445); rmse: 1.1247185563229848E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 1 (255/445); rmse: 3.099419046695359 4E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 2 (445/445); rmse: 3.29876164728273E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 1 (255/445); rmse: 1.3957799350224935E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 2 (445/445); rmse: 1.1247006793170353E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 1 (255/445); rmse: 1.1713489868719868E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 2 (445/445); rmse: 1.2786400532825679E7
--------------------------------------------------
Best Hyper Parameter is:
Some((1.2249945200772773E7,Map(RegParam -> 0.5, ElasticNetParam -> 0.5)))
```

```
//1 week best cv settings
//best hyperparameters found: RegParam = 0.5 ,ElasticNetParam = 0.5
val lr = new LinearRegression()
      .setRegParam(0.5)
      .setElasticNetParam(0.5)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(lr))
```

```
// Train model
val model = pipeline.fit(train_1week)

// Make predictions.
val predictions = model.transform(test_1week)
// Compute sMAPE
// Reconstruction of predictions with seasonal differences
val diff_1week = test_diff_1week.withColumn("id", monotonically_increasing_id())
val pred_1week = predictions.withColumn("id", monotonically_increasing_id())
val merged_1week = pred_1week.join(diff_1week, diff_1week.col("id") === pred_1week.col("id"), "left_outer").drop("id")
val recon_predictions_1week = merged_1week.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_1week.
//   limit(113).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1week.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|2.4552|
+------+

lr: org.apache.spark.ml.regression.LinearRegression = linReg_e75dd4a9e1ce
pipeline: org.apache.spark.ml.Pipeline = pipeline_2d39b17bffe4
model: org.apache.spark.ml.PipelineModel = pipeline_2d39b17bffe4
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1week: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
//1 week prediction horizon
display(recon_predictions_1week)
```



Prediction Window = 2 weeks

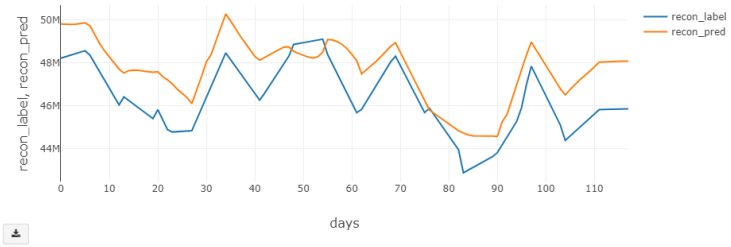
```
// 2 weeks rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(66, 180, train_2weeks, hyper_params) //train2weeks has 426 rows, use 66 as train and every 180 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")
```

```
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 1 (246/426); rmse: 4.101761186896301E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 2 (426/426); rmse: 3.617825804620921E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 1 (246/426); rmse: 4.101761186896301E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 2 (426/426); rmse: 3.617825804620921E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 1 (246/426); rmse: 4.101761186896301E7
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 2 (426/426); rmse: 3.617825804620921E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 1 (246/426); rmse: 4.528310733711382E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 2 (426/426); rmse: 3.95191021964172E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 1 (246/426); rmse: 4490290.7512706965
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 2 (426/426); rmse: 3943785.1335564437
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 1 (246/426); rmse: 4759192.251733424
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 2 (426/426); rmse: 4185647.1932266937
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 1 (246/426); rmse: 4.525904206404188E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 2 (426/426); rmse: 3.949861856086161E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 1 (246/426); rmse: 4759171.526772669
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 2 (426/426); rmse: 4185625.461447741
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 1 (246/426); rmse: 4224149.074380541
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 2 (426/426); rmse: 3681338.317918476
-------------------------------------------------
Best Hyper Parameter is:
Some((3952743.6961495085,Map(RegParam -> 0.5, ElasticNetParam -> 0.5)))
```

```
//2 week best cv settings
//best hyperparameters found: RegParam = 0.5, ElasticNetParam = 0.5
val lr = new LinearRegression()
      .setRegParam(0.5)
      .setElasticNetParam(0.5)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(lr))

// Train model
val model = pipeline.fit(train_2weeks)

// Make predictions.
val predictions = model.transform(test_2weeks)
// Compute sMAPE
// Reconstruction of predictions with seasonal differences
val diff_2weeks = test_diff_2weeks.withColumn("id", monotonically_increasing_id())
val pred_2weeks = predictions.withColumn("id", monotonically_increasing_id())
val merged_2weeks = pred_2weeks.join(diff_2weeks, diff_2weeks.col("id") === pred_2weeks.col("id"), "left_outer").drop("id")
val recon_predictions_2weeks = merged_2weeks.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_2weeks.
//   limit(104).
```

```
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_2weeks.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
|3.0125|
+------+

lr: org.apache.spark.ml.regression.LinearRegression = linReg_a14b88090d60
pipeline: org.apache.spark.ml.Pipeline = pipeline_c1f5146ff99f
model: org.apache.spark.ml.PipelineModel = pipeline_c1f5146ff99f
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_2weeks: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
// 2 weeks prediction horizon
display(recon_predictions_2weeks)
```



Prediction Window = 3 weeks

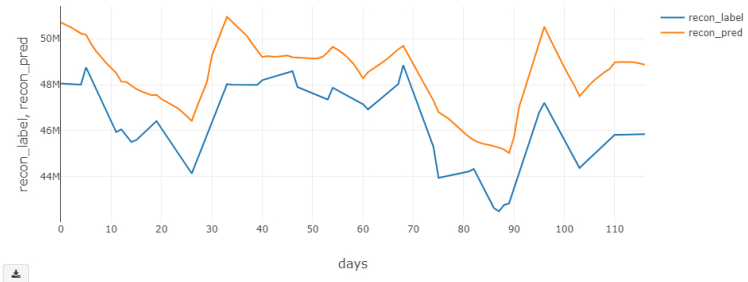
```
//3 weeks rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(60, 173, train_3weeks, hyper_params) //train3weeks has 406 rows, use 60 as train and every 173 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("------------------------------------------------")
```

```
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 1 (233/406); rmse: 4809257.446063449
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 2 (406/406); rmse: 4084532.277841278
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 1 (233/406); rmse: 4809257.446063449
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 2 (406/406); rmse: 4084532.277841278
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 1 (233/406); rmse: 4809257.446063449
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 2 (406/406); rmse: 4084532.277841278
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 1 (233/406); rmse: 8682838.66718464
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 2 (406/406); rmse: 7545833.671581693
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 1 (233/406); rmse: 1124544.6339932622
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 2 (406/406); rmse: 3198521.8145924374
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 1 (233/406); rmse: 1133699.8460338216
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 2 (406/406); rmse: 3233872.5414895597
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 1 (233/406); rmse: 8542441.45863956
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 2 (406/406); rmse: 7398190.509162538
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 1 (233/406); rmse: 1133699.8000009744
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 2 (406/406); rmse: 3233830.4842536086
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 1 (233/406); rmse: 1401330.7248013548
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 2 (406/406); rmse: 2322996.00965402
------------------------------------------------
Best Hyper Parameter is:
Some((1862163.3672276875,Map(RegParam -> 0.5, ElasticNetParam -> 0.5)))
```

```
//3 week best cv settings
//best hyperparameters found: RegParam = 0.5, ElasticNetParam = 0.5
val lr = new LinearRegression()
      .setRegParam(0.5)
      .setElasticNetParam(0.5)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(lr))

// Train model
val model = pipeline.fit(train_3weeks)

// Make predictions.
val predictions = model.transform(test_3weeks)
// Reconstruction of predictions with seasonal differences
val diff_3weeks = test_diff_3weeks.withColumn("id", monotonically_increasing_id())
val pred_3weeks = predictions.withColumn("id", monotonically_increasing_id())
val merged_3weeks = pred_3weeks.join(diff_3weeks, diff_3weeks.col("id") === pred_3weeks.col("id"), "left_outer").drop("id")
val recon_predictions_3weeks = merged_3weeks.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_3weeks.
//   limit(96).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_3weeks.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
|4.5267|
+------+

lr: org.apache.spark.ml.regression.LinearRegression = linReg_96857c506173
pipeline: org.apache.spark.ml.Pipeline = pipeline_f1b136e1ab7e
model: org.apache.spark.ml.PipelineModel = pipeline_f1b136e1ab7e
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_3weeks: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
```

```
pred_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
// 3 weeks prediction horizon
display(recon_predictions_3weeks)
```



Prediction Window = 1 month

```
//1 month rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(57, 165, train_1month, hyper_params) //train1month has 387 rows, use 57 as train and every 165 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")
```

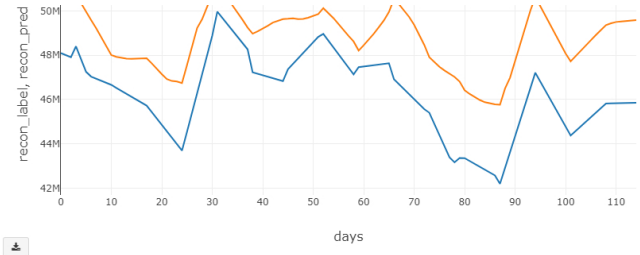
```
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 1 (222/387); rmse: 8734023.08804814
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0); rolling times: 2 (387/387); rmse: 8981644.323251298
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 1 (222/387); rmse: 8734023.08804814
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.1); rolling times: 2 (387/387); rmse: 8981644.323251298
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 1 (222/387); rmse: 8734023.08804814
Hyper params: Map(RegParam -> 0, ElasticNetParam -> 0.5); rolling times: 2 (387/387); rmse: 8981644.323251298
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 1 (222/387); rmse: 9785693.585711066
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0); rolling times: 2 (387/387); rmse: 1.484204622551107E7
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 1 (222/387); rmse: 7101804.927184899
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.1); rolling times: 2 (387/387); rmse: 6914711.79720566
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 1 (222/387); rmse: 7016881.7671507755
Hyper params: Map(RegParam -> 0.1, ElasticNetParam -> 0.5); rolling times: 2 (387/387); rmse: 6785089.453951162
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 1 (222/387); rmse: 9382911.071718205
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0); rolling times: 2 (387/387); rmse: 1.3202975391361753E7
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 1 (222/387); rmse: 7016669.382083226
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.1); rolling times: 2 (387/387); rmse: 6784866.105937593
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 1 (222/387); rmse: 7282447.848357575
Hyper params: Map(RegParam -> 0.5, ElasticNetParam -> 0.5); rolling times: 2 (387/387); rmse: 6963676.744125272
-------------------------------------------------
Best Hyper Parameter is:
Some((6900767.744010409,Map(RegParam -> 0.5, ElasticNetParam -> 0.1)))
```

```
//1 month best cv settings
//best hyperparameters found: RegParam = 0.5, ElasticNetParam = 0.1
val lr = new LinearRegression()
      .setRegParam(0.5)
      .setElasticNetParam(0.1)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(lr))

// Train model
val model = pipeline.fit(train_1month)

// Make predictions.
val predictions = model.transform(test_1month)
// Reconstruction of predictions with seasonal differences
val diff_1month = test_diff_1month.withColumn("id", monotonically_increasing_id())
val pred_1month = predictions.withColumn("id", monotonically_increasing_id())
val merged_1month = pred_1month.join(diff_1month, diff_1month.col("id") === pred_1month.col("id"), "left_outer").drop("id")
val recon_predictions_1month = merged_1month.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_1month.
//   limit(87).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1month.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|5.3629|
+------+
```

```
lr: org.apache.spark.ml.regression.LinearRegression = linReg_44a5706f4ff4
pipeline: org.apache.spark.ml.Pipeline = pipeline_7a30dc51cda5
model: org.apache.spark.ml.PipelineModel = pipeline_7a30dc51cda5
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1month: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
// 1 month prediction horizon
display(recon_predictions_1month)
```



Decision Tree Models Predictions, Reconstructions, and SMAPE

```
import org.apache.spark.ml.regression.DecisionTreeRegressor
import org.apache.spark.ml.regression.DecisionTreeRegressionModel

def rolling_cv_tuning(initial_train_obs: Int, shift: Int, assembled_df: DataFrame, hyper_params: Map[String, String]): Double = {

  val total_rows = assembled_df.count()
  val rolling_space = total_rows - initial_train_obs
  val fold_num = (rolling_space / shift).toInt


  var total_rmse = 0.0

  for (i <- 1 to fold_num) {

    // define rolling frame
    var total_select = initial_train_obs + shift * i
    var train_pct = initial_train_obs.toFloat / total_select

    // train test split
    val (train_set, test_set) = ts_split(train_pct, assembled_df.limit(total_select.toInt))

    /*******************REPLACE TO OTHER MODEL START *********************/
    // Initiate model object
    val dt = new DecisionTreeRegressor()
      .setMaxDepth(hyper_params("MaxDepth").toInt)
      .setMaxBins(hyper_params("MaxBins").toInt)
      .setLabelCol("label")
      .setFeaturesCol("features")

    // Setup Pipeline.
    val pipeline = new Pipeline().setStages(Array(dt))

    // Train model
    val model = pipeline.fit(train_set)

    // Make predictions.
    val predictions = model.transform(test_set)

    // Select (prediction, true label) and compute test error.
    val evaluator = new RegressionEvaluator()
      .setMetricName("rmse")

    val rmse = evaluator.evaluate(predictions)

    // trained model
    val dtModel = model.stages(0).asInstanceOf[DecisionTreeRegressionModel]
    /*******************REPLACE TO OTHER MODEL END *********************/

    total_rmse += rmse

    println(s"Hyper params: $hyper_params; rolling times: $i ($total_select/$total_rows); rmse: $rmse")
  }

  return total_rmse / fold_num

}
```

```
import org.apache.spark.ml.regression.DecisionTreeRegressor
import org.apache.spark.ml.regression.DecisionTreeRegressionModel
rolling_cv_tuning: (initial_train_obs: Int, shift: Int, assembled_df: org.apache.spark.sql.DataFrame, hyper_params: Map[String,String])Double
```

```
// Setting up for rolling cv
val param_grid = Map(
  "MaxDepth" -> Seq("5","7","10"),
  "MaxBins" -> Seq("20","25","30")
)

// transform arrays into lists with values paired with map key
val pairedWithKey = param_grid.map { case (k,v) => v.map(i => k -> i).toList }

val accumulator = pairedWithKey.head.map(x => Vector(x))

val params_combination = pairedWithKey.tail.foldLeft(accumulator)( (acc, elem) =>
  for { x <- acc; y <- elem } yield x :+ y
)
```

```
param_grid: scala.collection.immutable.Map[String,Seq[String]] = Map(MaxDepth -> List(5, 7, 10), MaxBins -> List(20, 25, 30))
pairedWithKey: scala.collection.immutable.Iterable[List[(String, String)]] = List(List((MaxDepth,5), (MaxDepth,7), (MaxDepth,10)), List((MaxBins,20), (MaxBins,25), (MaxBins,30)))
accumulator: List[scala.collection.immutable.Vector[(String, String)]] = List(Vector((MaxDepth,5)), Vector((MaxDepth,7)), Vector((MaxDepth,10)))
params_combination: List[scala.collection.immutable.Vector[(String, String)]] = List(Vector((MaxDepth,5), (MaxBins,20)), Vector((MaxDepth,5), (MaxBins,25)), Vector((MaxDepth,5), (MaxBins,30)), Vector((MaxDepth,7), (MaxBins,20)), Vector((MaxDepth,7), (MaxBins,25)), Vector((MaxDepth,7), (MaxBins,30)), Vector((MaxDepth,10), (MaxBins,20)), Vector((MaxDepth,10), (MaxBins,25)), Vector((MaxDepth,10), (MaxBins,30)))
```

Prediction Window = 1 day

```
// 1 day rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(64, 200, train_1day, hyper_params) //train1day has 464 rows, use 64 as train and every 200 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")
```

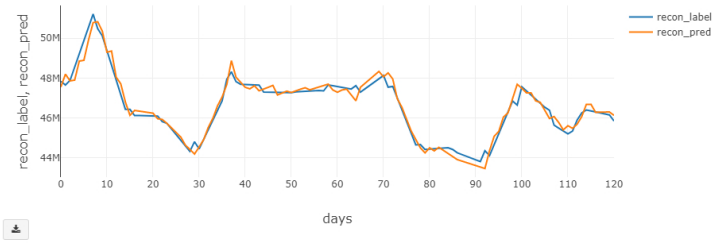
```
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 1 (264/464); rmse: 513285.5844499546
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 2 (464/464); rmse: 911753.7618601996
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 1 (264/464); rmse: 581663.2069489224
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 2 (464/464); rmse: 874428.626511769
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 1 (264/464); rmse: 507244.3449370127
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 2 (464/464); rmse: 899194.7157377745
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 1 (264/464); rmse: 538367.9637294344
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 2 (464/464); rmse: 922883.8155846416
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 1 (264/464); rmse: 614947.4379434711
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 2 (464/464); rmse: 896092.9087471683
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 1 (264/464); rmse: 512133.69896521996
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 2 (464/464); rmse: 912643.6062029466
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 1 (264/464); rmse: 545409.464234261
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 2 (464/464); rmse: 923466.3038981276
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 1 (264/464); rmse: 611479.3568314001
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 2 (464/464); rmse: 897145.0625249345
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 1 (264/464); rmse: 516702.4732766299
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 2 (464/464); rmse: 912972.2891411036
-------------------------------------------------
Best Hyper Parameter is:
Some((703219.5303373936,Map(MaxDepth -> 5, MaxBins -> 30)))
```

```
//1 day best cv settings
//best hyperparameters found: MaxDepth = 5, MaxBins = 30
val dt = new DecisionTreeRegressor()
  .setLabelCol("label")
  .setFeaturesCol("features")
  .setMaxDepth(5)
  .setMaxBins(30)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(dt))

// Train model
val model = pipeline.fit(train_1day)

// Make predictions.
val predictions = model.transform(test_1day)

// Reconstruction of predictions with seasonal differences
val diff_1day = test_diff_1day.withColumn("id", monotonically_increasing_id())
val pred_1day = predictions.withColumn("id", monotonically_increasing_id())
val merged_1day = pred_1day.join(diff_1day, diff_1day.col("id") === pred_1day.col("id"), "left_outer").drop("id")
val recon_predictions_1day = merged_1day.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_1day.
//   limit(120).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1day.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|0.5103|
+------+
```

```
dt: org.apache.spark.ml.regression.DecisionTreeRegressor = dtr_67fec2860b10
pipeline: org.apache.spark.ml.Pipeline = pipeline_9b6bbd1e2c3b
model: org.apache.spark.ml.PipelineModel = pipeline_9b6bbd1e2c3b
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1day: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
// 1 day prediction horizon
display(recon_predictions_1day)
```



Prediction Window = 1 week

```
//1 week rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(65, 190, train_1week, hyper_params) //train1week has 445 rows, use 65 as train and every 190 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")
```

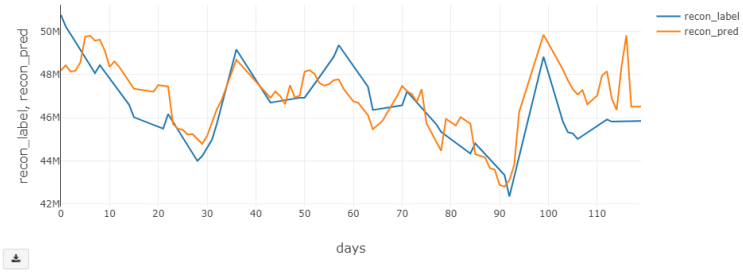
```
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 1 (255/445); rmse: 1849506.777300293
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 2 (445/445); rmse: 1952464.0839697993
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 1 (255/445); rmse: 1670071.8927188322
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 2 (445/445); rmse: 1864044.4240888797
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 1 (255/445); rmse: 1942724.4210595957
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 2 (445/445); rmse: 1972019.9857668884
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 1 (255/445); rmse: 1835924.2251680098
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 2 (445/445); rmse: 1968340.1805602352
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 1 (255/445); rmse: 1667901.527659499
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 2 (445/445); rmse: 1870810.7871175914
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 1 (255/445); rmse: 1974659.3762590883
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 2 (445/445); rmse: 2003463.9170115897
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 1 (255/445); rmse: 1832240.7471404963
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 2 (445/445); rmse: 1964442.0658798576
```

```
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 1 (255/445); rmse: 1666807.8388545294
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 2 (445/445); rmse: 1860597.0690116843
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 1 (255/445); rmse: 1986668.8927004575
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 2 (445/445); rmse: 2004819.8469928901
--------------------------------------------------
Best Hyper Parameter is:
Some((1763702.4539331067,Map(MaxDepth -> 10, MaxBins -> 25)))
```

```
//1 week best cv settings
//best hyperparameters found: MaxDepth -> 10, MaxBins -> 25
val dt = new DecisionTreeRegressor()
  .setLabelCol("label")
  .setFeaturesCol("features")
  .setMaxDepth(10)
  .setMaxBins(25)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(dt))

// Train model
val model = pipeline.fit(train_1week)

// Make predictions.
val predictions = model.transform(test_1week)

// Reconstruction of predictions with seasonal differences
val diff_1week = test_diff_1week.withColumn("id", monotonically_increasing_id())
val pred_1week = predictions.withColumn("id", monotonically_increasing_id())
val merged_1week = pred_1week.join(diff_1week, diff_1week.col("id") === pred_1week.col("id"), "left_outer").drop("id")
val recon_predictions_1week = merged_1week.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_1week.
//   limit(113).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1week.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
|SMAPE|
+------+
|2.128|
+------+

dt: org.apache.spark.ml.regression.DecisionTreeRegressor = dtr_ba45258cbf00
pipeline: org.apache.spark.ml.Pipeline = pipeline_42e1697d9608
model: org.apache.spark.ml.PipelineModel = pipeline_42e1697d9608
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1week: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
// 1 week prediction horizon
display(recon_predictions_1week)
```



Prediction Window = 2 weeks

```
// 2 week rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(66, 180, train_2weeks, hyper_params) //train2weeks has 426 rows, use 66 as train and every 180 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("--------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("--------------------------------------------------")
```

```
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 1 (246/426); rmse: 2714688.9595595673
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 2 (426/426); rmse: 2254383.397492016
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 1 (246/426); rmse: 2697096.8430075366
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 2 (426/426); rmse: 2263045.5187261216
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 1 (246/426); rmse: 2620529.2689621006
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 2 (426/426); rmse: 2199475.9712490453
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 1 (246/426); rmse: 2694682.7641937514
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 2 (426/426); rmse: 2264272.119201302
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 1 (246/426); rmse: 2694832.4337504855
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 2 (426/426); rmse: 2271571.7148701777
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 1 (246/426); rmse: 2612918.284682655
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 2 (426/426); rmse: 2183792.4655912044
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 1 (246/426); rmse: 2699353.936871085
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 2 (426/426); rmse: 2260284.751785886
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 1 (246/426); rmse: 2696536.578118588
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 2 (426/426); rmse: 2274261.5350220874
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 1 (246/426); rmse: 2614652.938674883
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 2 (426/426); rmse: 2183033.1265034927
--------------------------------------------------
Best Hyper Parameter is:
Some((2398355.3751369296,Map(MaxDepth -> 7, MaxBins -> 30)))
```

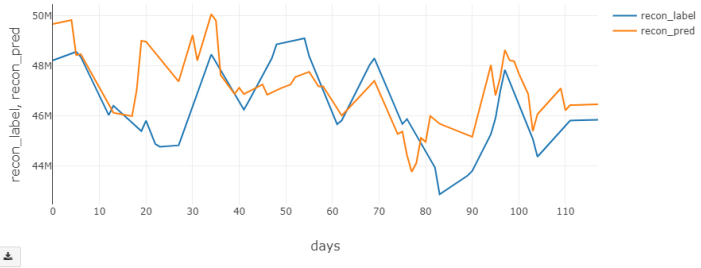
```
// 2 weeks best cv settings
// best hyperparameters found:MaxDepth -> 7, MaxBins -> 30
val dt = new DecisionTreeRegressor()
```

```
  .setLabelCol("label")
  .setFeaturesCol("features")
  .setMaxDepth(7)
  .setMaxBins(30)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(dt))

// Train model
val model = pipeline.fit(train_2weeks)

// Make predictions.
val predictions = model.transform(test_2weeks)

// Reconstruction of predictions with seasonal differences
val diff_2weeks = test_diff_2weeks.withColumn("id", monotonically_increasing_id())
val pred_2weeks = predictions.withColumn("id", monotonically_increasing_id())
val merged_2weeks = pred_2weeks.join(diff_2weeks, diff_2weeks.col("id") === pred_2weeks.col("id"), "left_outer").drop("id")
val recon_predictions_2weeks = merged_2weeks.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_2weeks.
//   limit(104).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_2weeks.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|2.5853|
+------+
```

```
dt: org.apache.spark.ml.regression.DecisionTreeRegressor = dtr_e1272083e7e
pipeline: org.apache.spark.ml.Pipeline = pipeline_cd41cde210be
model: org.apache.spark.ml.PipelineModel = pipeline_cd41cde210be
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_2weeks: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
// 2 weeks prediction horizon
display(recon_predictions_2weeks)
```



Prediction Window = 3 weeks

```
// 3 week rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(60, 173, train_3weeks, hyper_params) //train3weeks has 406 rows, use 60 as train and every 173 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")
```

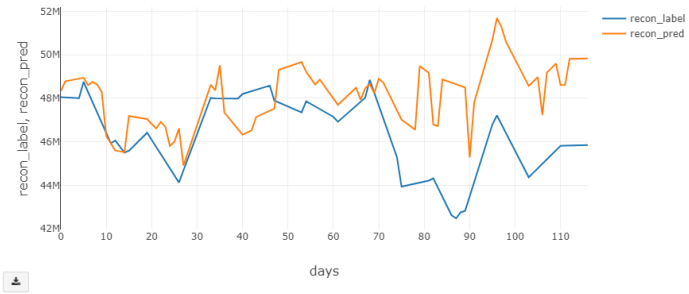
```
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 1 (233/406); rmse: 1199202.5806906673
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 2 (406/406); rmse: 1532512.843073247
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 1 (233/406); rmse: 1266471.3647186894
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 2 (406/406); rmse: 1505170.6386685888
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 1 (233/406); rmse: 1233395.3204438135
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 2 (406/406); rmse: 1542982.7757104973
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 1 (233/406); rmse: 1184516.914212802
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 2 (406/406); rmse: 1538308.637144782
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 1 (233/406); rmse: 1269829.189395284
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 2 (406/406); rmse: 1519418.89425426
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 1 (233/406); rmse: 1208806.023783766
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 2 (406/406); rmse: 1523333.1805032704
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 1 (233/406); rmse: 1189317.890879421
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 2 (406/406); rmse: 1537831.5572623226
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 1 (233/406); rmse: 1265708.0893758317
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 2 (406/406); rmse: 1513683.8607907186
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 1 (233/406); rmse: 1194305.6464969213
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 2 (406/406); rmse: 1515794.230379794
-------------------------------------------------
Best Hyper Parameter is:
Some((1355049.9384383578,Map(MaxDepth -> 10, MaxBins -> 30)))
```

```
//3 week  best cv settings
//best hyperparameters found:MaxDepth -> 10, MaxBins -> 30
val dt = new DecisionTreeRegressor()
  .setLabelCol("label")
  .setFeaturesCol("features")
  .setMaxDepth(10)
  .setMaxBins(30)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(dt))

// Train model
val model = pipeline.fit(train_3weeks)

// Make predictions.
```

```
val predictions = model.transform(test_3weeks)

// Reconstruction of predictions with seasonal differences
val diff_3weeks = test_diff_3weeks.withColumn("id", monotonically_increasing_id())
val pred_3weeks = predictions.withColumn("id", monotonically_increasing_id())
val merged_3weeks = pred_3weeks.join(diff_3weeks, diff_3weeks.col("id") === pred_3weeks.col("id"), "left_outer").drop("id")
val recon_predictions_3weeks = merged_3weeks.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
recon_predictions_3weeks.
// limit(96).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_3weeks.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|4.4684|
+------+

dt: org.apache.spark.ml.regression.DecisionTreeRegressor = dtr_276a78c096b1
pipeline: org.apache.spark.ml.Pipeline = pipeline_83c5c3ea4bf9
model: org.apache.spark.ml.PipelineModel = pipeline_83c5c3ea4bf9
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_3weeks: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

```
// 3 weeks prediction horizon
display(recon_predictions_3weeks)
```



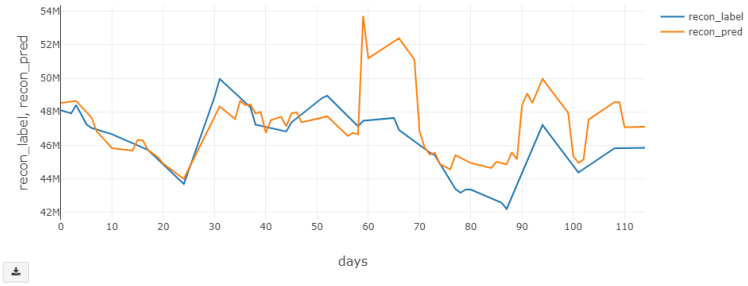
Prediction Window = 1 month

```
//1month rolling cv + tuning
var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  val avg_rmse = rolling_cv_tuning(57, 165, train_1month, hyper_params) //train1month has 387 rows, use 57 as train and every 165 for cv

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}
// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")
```

```
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 1 (222/387); rmse: 1072602.0542204771
Hyper params: Map(MaxDepth -> 5, MaxBins -> 20); rolling times: 2 (387/387); rmse: 1521489.7763248696
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 1 (222/387); rmse: 1073211.7138925483
Hyper params: Map(MaxDepth -> 5, MaxBins -> 25); rolling times: 2 (387/387); rmse: 1490831.8794757624
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 1 (222/387); rmse: 1148326.4020893045
Hyper params: Map(MaxDepth -> 5, MaxBins -> 30); rolling times: 2 (387/387); rmse: 1522017.6786206134
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 1 (222/387); rmse: 1014877.107655046
Hyper params: Map(MaxDepth -> 7, MaxBins -> 20); rolling times: 2 (387/387); rmse: 1497697.3527624656
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 1 (222/387); rmse: 1028459.4878591116
Hyper params: Map(MaxDepth -> 7, MaxBins -> 25); rolling times: 2 (387/387); rmse: 1482176.0999556913
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 1 (222/387); rmse: 1090455.401658053
Hyper params: Map(MaxDepth -> 7, MaxBins -> 30); rolling times: 2 (387/387); rmse: 1498624.1123454452
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 1 (222/387); rmse: 1016557.6021465559
Hyper params: Map(MaxDepth -> 10, MaxBins -> 20); rolling times: 2 (387/387); rmse: 1495719.0290720402
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 1 (222/387); rmse: 1036976.1335293652
Hyper params: Map(MaxDepth -> 10, MaxBins -> 25); rolling times: 2 (387/387); rmse: 1484681.0435539752
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 1 (222/387); rmse: 1084689.3146357497
Hyper params: Map(MaxDepth -> 10, MaxBins -> 30); rolling times: 2 (387/387); rmse: 1496892.9711915753
-------------------------------------------------
Best Hyper Parameter is:
Some((1255317.7939074014,Map(MaxDepth -> 7, MaxBins -> 25)))
```

```
//1 month  best cv settings
//best hyperparameters found: MaxDepth -> 7, MaxBins -> 25
val dt = new DecisionTreeRegressor()
  .setLabelCol("label")
  .setFeaturesCol("features")
  .setMaxDepth(7)
  .setMaxBins(25)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(dt))

// Train model
val model = pipeline.fit(train_1month)

// Make predictions.
val predictions = model.transform(test_1month)

// Reconstruction of predictions with seasonal differences
val diff_1month = test_diff_1month.withColumn("id", monotonically_increasing_id())
val pred_1month = predictions.withColumn("id", monotonically_increasing_id())
val merged_1month = pred_1month.join(diff_1month, diff_1month.col("id") === pred_1month.col("id"), "left_outer").drop("id")
val recon_predictions_1month = merged_1month.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("days",monotonically_increasing_id())

// Compute sMAPE
```

```
recon_predictions_1month.
//   limit(87).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1month.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
|3.1903|
+------+

dt: org.apache.spark.ml.regression.DecisionTreeRegressor = dtr_78fce4c6da48
pipeline: org.apache.spark.ml.Pipeline = pipeline_7cfad423c96f
model: org.apache.spark.ml.PipelineModel = pipeline_7cfad423c96f
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1month: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]


// 1 month prediction horizon
display(recon_predictions_1month)
```



Gradient Boosted Tree Models Predictions, Reconstructions, and SMAPE

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.{GBTRegressionModel, GBTRegressor}
import org.apache.spark.sql.DataFrame
import org.apache.spark.ml.feature.VectorAssembler

def rolling_cv_tuning(initial_train_obs: Int, shift: Int, assembled_df: DataFrame, hyper_params: Map[String, String]): Double = {
  /**
    * Returns the average rmse for the whole rolling process
    * For example, we have 143 rows in total,
    * we set the initial train as 43, and 10 more rolling forward, then
    * train first 43, test 44 - 53
    * train first 53, test 54 - 64
    * ...
    * train first 133, test 134 - 143
    * the function will roll 9 times in this case, and we calculate the rmse for each time / 9, and return the value
    */

  val total_rows = assembled_df.count()
  val rolling_space = total_rows - initial_train_obs
  val fold_num = (rolling_space / shift).toInt


  var total_rmse = 0.0

  for (i <- 1 to fold_num) {

    // define rolling frame
    var total_select = initial_train_obs + shift * i
    var train_pct = initial_train_obs.toFloat / total_select

    // train test split
    val (train_set, test_set) = ts_split(train_pct, assembled_df.limit(total_select.toInt))

    // Initiate model object

    val gbt = new GBTRegressor()
      .setMaxIter(hyper_params("Iteration").toInt)
      .setMaxBins(hyper_params("MaxBins").toInt)
      .setMaxDepth(hyper_params("MaxDepth").toInt)

    // Setup Pipeline.
    val pipeline = new Pipeline().setStages(Array(gbt))

    // Train model
    val model = pipeline.fit(train_set)

    // Make predictions.
    val predictions = model.transform(test_set)

    // Select (prediction, true label) and compute test error.
    val evaluator = new RegressionEvaluator()
      .setMetricName("rmse")

    val rmse = evaluator.evaluate(predictions)

    // trained model
    val gbtModel = model.stages(0).asInstanceOf[GBTRegressionModel]

    total_rmse += rmse

    println(s"Hyper params: $hyper_params; rolling times: $i ($total_select/$total_rows); rmse: $rmse")
  }

  return total_rmse / fold_num

}

import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.{GBTRegressionModel, GBTRegressor}
import org.apache.spark.sql.DataFrame
import org.apache.spark.ml.feature.VectorAssembler
rolling_cv_tuning: (initial_train_obs: Int, shift: Int, assembled_df: org.apache.spark.sql.DataFrame, hyper_params: Map[String,String])Double
```

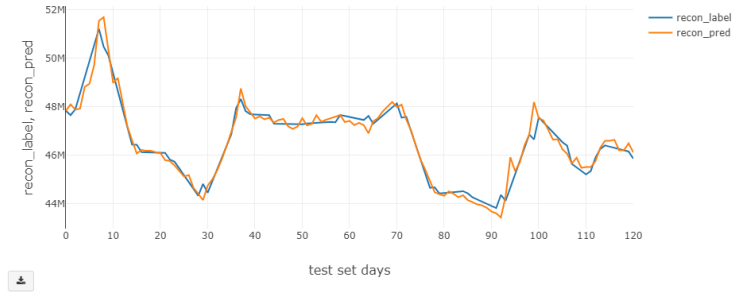
Prediction Window = 1 day

```
val param_grid = Map(
  "Iteration" -> Seq("5","7","10"),
  "MaxBins" -> Seq("23","25","27"),
  "MaxDepth" -> Seq("6","8","10")
)

// transform arrays into lists with values paired with map key
val pairedWithKey = param_grid.map { case (k,v) => v.map(i => k -> i).toList }

val accumulator = pairedWithKey.head.map(x => Vector(x))

val params_combination = pairedWithKey.tail.foldLeft(accumulator)( (acc, elem) =>
  for { x <- acc; y <- elem } yield x :+ y
)

var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  // how to set the initial value (first param), and shift value (second param)?
  // In train_1day, we have 464 total rows, I set 64 as the initial, and 200 for each rolling forward
  // this means we will roll 2 times in total for each combination of hyper-params: (464 - 64) / 200 = 2
  // *the smaller the shift, the more rolling times, and the longer time we need to run this function
  val avg_rmse = rolling_cv_tuning(64, 200, train_1day, hyper_params)

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}

// print best hyper-paramter and its average rmse
println("------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("------------------------------------------------")



// use best hyper parameter to train a new model, and apply to the test set
//Hyper params: Map(MaxDepth -> 6, MaxBins -> 23, Iteration -> 5); rolling times: 1 (264/464); rmse: 501392.40469460766
//Hyper params: Map(MaxDepth -> 6, MaxBins -> 23, Iteration -> 5); rolling times: 2 (464/464); rmse: 1005183.6056703655
// GBT model set-up using the tuned hyperparemeter.

val gbt = new GBTRegressor()
    .setFeaturesCol("features")
    .setMaxIter(5)
    .setMaxBins(23)
    .setMaxDepth(6)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(gbt))

// Train model
val model = pipeline.fit(train_1day)

// Make predictions.
val predictions = model.transform(test_1day)

// Reconstruction of predictions with seasonal differences
val diff_1day = test_diff_1day.withColumn("id", monotonically_increasing_id())
val pred_1day = predictions.withColumn("id", monotonically_increasing_id())
val merged_1day = pred_1day.join(diff_1day, diff_1day.col("id") === pred_1day.col("id"), "left_outer").drop("id")
val recon_predictions_1day = merged_1day.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_1day.
//   limit(120).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1day.count() * 100, 4) as "SMAPE").
  show()
```

```
+------+
| SMAPE|
+------+
|0.4785|
+------+
```

```
gbt: org.apache.spark.ml.regression.GBTRegressor = gbtr_6e86699089c5
pipeline: org.apache.spark.ml.Pipeline = pipeline_c7943b8b3374
model: org.apache.spark.ml.PipelineModel = pipeline_c7943b8b3374
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1day: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1day: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]
```

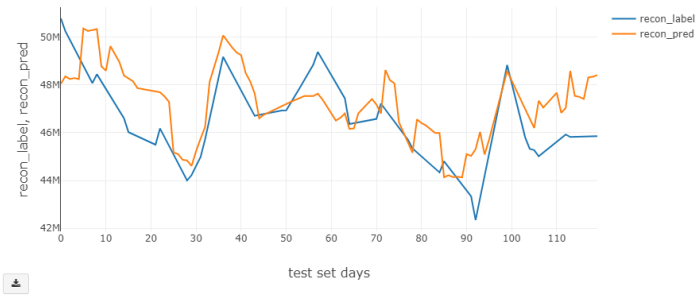
```
display(recon_predictions_1day)
```



Prediction Window = 1 week

```
val param_grid = Map(
  "Iteration" -> Seq("5","7","10"),
  "MaxBins" -> Seq("25","27","30"),
  "MaxDepth" -> Seq("4","6","8")
)
// transform arrays into lists with values paired with map key
val pairedWithKey = param_grid.map { case (k,v) => v.map(i => k -> i).toList }

val accumulator = pairedWithKey.head.map(x => Vector(x))
```

```
val params_combination = pairedWithKey.tail.foldLeft(accumulator)( (acc, elem) =>
  for { x <- acc; y <- elem } yield x :+ y
)

var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  // how to set the initial value (first param), and shift value (second param)?
  // In train_1week, we have 445 total rows, I set 65 as the initial, and 190 for each rolling forward
  // this means we will roll 2 times in total for each combination of hyper-params: (445 - 65) / 190 = 2
  // *the smaller the shift, the more rolling times, and the longer time we need to run this function
  val avg_rmse = rolling_cv_tuning(65, 190, train_1week, hyper_params)

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}

// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")



// use best hyper parameter to train a new model, and apply to the test set
// GBT model set-up
// Hyper params: Map(MaxDepth -> 6, MaxBins -> 25, Iteration -> 5); rolling times: 1 (255/445); rmse: 1546188.490863489
// Hyper params: Map(MaxDepth -> 6, MaxBins -> 25, Iteration -> 5); rolling times: 2 (445/445); rmse: 1826336.0053840075

val gbt = new GBTRegressor()
    .setFeaturesCol("features")
    .setMaxIter(5)
    .setMaxBins(25)
    .setMaxDepth(6)


// Train model
val model = pipeline.fit(train_1week)

// Make predictions.
val predictions = model.transform(test_1week)

// Reconstruction of predictions with seasonal differences
val diff_1week = test_diff_1week.withColumn("id", monotonically_increasing_id())
val pred_1week = predictions.withColumn("id", monotonically_increasing_id())
val merged_1week = pred_1week.join(diff_1week, diff_1week.col("id") === pred_1week.col("id"), "left_outer").drop("id")
val recon_predictions_1week = merged_1week.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_1week.
//   limit(113).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1week.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
|2.4828|
+------+

gbt: org.apache.spark.ml.regression.GBTRegressor = gbtr_63bd9a93bd0a
model: org.apache.spark.ml.PipelineModel = pipeline_c7943b8b3374
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1week: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1week: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]


display(recon_predictions_1week)
```



Prediction Window = 2 weeks

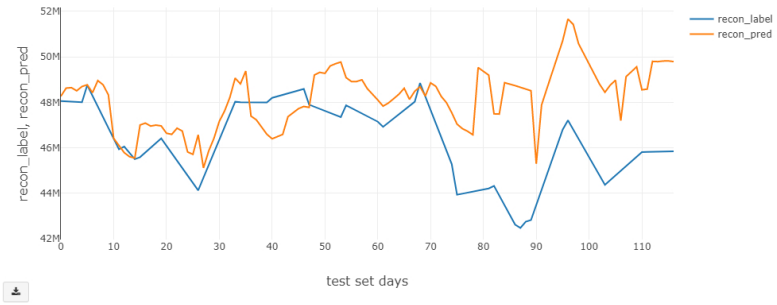
```
val param_grid = Map(
  "Iteration" -> Seq("5","7","10"),
  "MaxBins" -> Seq("25","27","30"),
  "MaxDepth" -> Seq("4","6","8")
)
// transform arrays into lists with values paired with map key
val pairedWithKey = param_grid.map { case (k,v) => v.map(i => k -> i).toList }

val accumulator = pairedWithKey.head.map(x => Vector(x))

val params_combination = pairedWithKey.tail.foldLeft(accumulator)( (acc, elem) =>
  for { x <- acc; y <- elem } yield x :+ y
)

var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

   // In train_2weeks, we have 426 total rows, I set 66 as the initial, and 180 for each rolling forward
  // this means we will roll 2 times in total for each combination of hyper-params: (426 - 66) / 180 = 2
  // *the smaller the shift, the more rolling times, and the longer time we need to run this function
  val avg_rmse = rolling_cv_tuning(66, 180, train_2weeks, hyper_params)
```

```
  result_collector = result_collector :+ (avg_rmse, hyper_params)

}

// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")



//Hyper params: Map(MaxDepth -> 4, MaxBins -> 27, Iteration -> 10); rolling times: 1 (246/426); rmse: 1755331.4325966472
//Hyper params: Map(MaxDepth -> 4, MaxBins -> 27, Iteration -> 10); rolling times: 2 (426/426); rmse: 1718890.8777430

val gbt = new GBTRegressor()
    .setFeaturesCol("features")
    .setMaxIter(4)
    .setMaxBins(27)
    .setMaxDepth(10)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(gbt))


// Train model
val model = pipeline.fit(train_2weeks)

// Make predictions.
val predictions = model.transform(test_2weeks)

// Reconstruction of predictions with seasonal differences
val diff_2weeks = test_diff_2weeks.withColumn("id", monotonically_increasing_id())
val pred_2weeks = predictions.withColumn("id", monotonically_increasing_id())
val merged_2weeks = pred_2weeks.join(diff_2weeks, diff_2weeks.col("id") === pred_2weeks.col("id"), "left_outer").drop("id")
val recon_predictions_2weeks = merged_2weeks.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_2weeks.
//   limit(104).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_2weeks.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
|1.8691|
+------+

gbt: org.apache.spark.ml.regression.GBTRegressor = gbtr_e9cb27a6d7af
pipeline: org.apache.spark.ml.Pipeline = pipeline_53944efcab17
model: org.apache.spark.ml.PipelineModel = pipeline_53944efcab17
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_2weeks: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_2weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]


display(recon_predictions_2weeks)
```

Prediction Window = 3 weeks

```
val param_grid = Map(
  "Iteration" -> Seq("5","7","10"),
  "MaxBins" -> Seq("25","27","30"),
  "MaxDepth" -> Seq("4","6","8")
)
// transform arrays into lists with values paired with map key
val pairedWithKey = param_grid.map { case (k,v) => v.map(i => k -> i).toList }

val accumulator = pairedWithKey.head.map(x => Vector(x))

val params_combination = pairedWithKey.tail.foldLeft(accumulator){ (acc, elem) =>
  for { x <- acc; y <- elem } yield x :+ y
)

var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  // how to set the initial value (first param), and shift value (second param)?
  // In train_3weeks, we have 406 total rows, I set 60 as the initial, and 173 for each rolling forward
  // this means we will roll 2 times in total for each combination of hyper-params: (406 - 60) / 173 = 2
  // *the smaller the shift, the more rolling times, and the longer time we need to run this function
  val avg_rmse = rolling_cv_tuning(60, 173, train_3weeks, hyper_params)

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}

// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")



//Hyper params: Map(MaxDepth -> 8, MaxBins -> 30, Iteration -> 5); rolling times: 1 (233/406); rmse: 1196415.9807007953
```

```
//Hyper params: Map(MaxDepth -> 8, MaxBins -> 30, Iteration -> 5); rolling times: 2 (406/406); rmse: 1529967.9947142801

val gbt = new GBTRegressor()
    .setFeaturesCol("features")
    .setMaxIter(5)
    .setMaxBins(30)
    .setMaxDepth(8)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(gbt))

// Train model
val model = pipeline.fit(train_3weeks)

// Make predictions.
val predictions = model.transform(test_3weeks)

// Reconstruction of predictions with seasonal differences
val diff_3weeks = test_diff_3weeks.withColumn("id", monotonically_increasing_id())
val pred_3weeks = predictions.withColumn("id", monotonically_increasing_id())
val merged_3weeks = pred_3weeks.join(diff_3weeks, diff_3weeks.col("id") === pred_3weeks.col("id"), "left_outer").drop("id")
val recon_predictions_3weeks = merged_3weeks.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_3weeks.
//   limit(96).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_3weeks.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
|4.4718|
+------+

gbt: org.apache.spark.ml.regression.GBTRegressor = gbtr_c64895a153f4
pipeline: org.apache.spark.ml.Pipeline = pipeline_a00f0ee3f5c2
model: org.apache.spark.ml.PipelineModel = pipeline_a00f0ee3f5c2
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_3weeks: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_3weeks: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]


display(recon_predictions_3weeks)
```



Prediction Window = 1 month

```
val param_grid = Map(
  "Iteration" -> Seq("5","7","10"),
  "MaxBins" -> Seq("25","27","30"),
  "MaxDepth" -> Seq("4","6","8")
)
// transform arrays into lists with values paired with map key
val pairedWithKey = param_grid.map { case (k,v) => v.map(i => k -> i).toList }

val accumulator = pairedWithKey.head.map(x => Vector(x))

val params_combination = pairedWithKey.tail.foldLeft(accumulator)( (acc, elem) =>
  for { x <- acc; y <- elem } yield x :+ y
)

var result_collector : List[(Double,Map[String, String])] = List()

for (comb <- params_combination) {

  val hyper_params = comb.groupBy(_._1).map { case (k,v) => (k,v.map(_._2).head)}

  // how to set the initial value (first param), and shift value (second param)?
  // In train_1month, we have 387 total rows, I set 57 as the initial, and 165 for each rolling forward
  // this means we will roll 2 times in total for each combination of hyper-params: (387 - 57) / 165 = 2
  // *the smaller the shift, the more rolling times, and the longer time we need to run this function
  val avg_rmse = rolling_cv_tuning(57, 165, train_1month, hyper_params)

  result_collector = result_collector :+ (avg_rmse, hyper_params)

}

// print best hyper-paramter and its average rmse
println("-------------------------------------------------")
println("Best Hyper Parameter is: ")
println(result_collector.sortBy(_._1).lift(0))
println("-------------------------------------------------")



//Hyper params: Map(MaxDepth -> 6, MaxBins -> 25, Iteration -> 5); rolling times: 1 (222/387); rmse: 997349.6312181305
//Hyper params: Map(MaxDepth -> 6, MaxBins -> 25, Iteration -> 5); rolling times: 2 (387/387); rmse: 1451847.3289248738
val gbt = new GBTRegressor()
    .setFeaturesCol("features")
    .setMaxIter(5)
    .setMaxBins(25)
    .setMaxDepth(6)

// Setup Pipeline.
val pipeline = new Pipeline().setStages(Array(gbt))

// Train model
val model = pipeline.fit(train_1month)

// Make predictions.
```

```
val predictions = model.transform(test_1month)

// Reconstruction of predictions with seasonal differences
val diff_1month = test_diff_1month.withColumn("id", monotonically_increasing_id())
val pred_1month = predictions.withColumn("id", monotonically_increasing_id())
val merged_1month = pred_1month.join(diff_1month, diff_1month.col("id") === pred_1month.col("id"), "left_outer").drop("id")
val recon_predictions_1month = merged_1month.withColumn("recon_label", $"label"+$"lag_1year")
  .withColumn("recon_pred", $"prediction"+$"lag_1year")
  .withColumn("test set days", monotonically_increasing_id())

// Compute SMAPE
recon_predictions_1month.
//   limit(87).
  withColumn("diff_abs", abs($"recon_pred" - $"recon_label")).
  withColumn("demo", (abs($"recon_pred") + abs($"recon_label")) / 2).
  withColumn("division", $"diff_abs" / $"demo").
  agg(round(sum($"division") / recon_predictions_1month.count() * 100, 4) as "SMAPE").
  show()

+------+
| SMAPE|
+------+
|3.2513|
+------+

gbt: org.apache.spark.ml.regression.GBTRegressor = gbtr_0cc1c94e872e
pipeline: org.apache.spark.ml.Pipeline = pipeline_57370a98503d
model: org.apache.spark.ml.PipelineModel = pipeline_57370a98503d
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more field]
diff_1month: org.apache.spark.sql.DataFrame = [lag_1year: double, id: bigint]
pred_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
merged_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 2 more fields]
recon_predictions_1month: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 5 more fields]


display(recon_predictions_1month)
```