Time Series Analysis of dataset downloaded

```
//Time Series Analysis of dataset downloaded from
https://www.kaggle.com/kanncaa1/time-series-prediction-tutorial-with-eda


//We have used 2 libraries for this, i.e. sparkts and flint


/*1. Load the Data

 It contains 2 files:
i) Aerial Bombing Operations in WW2
ii) Weather Conditions in WW2(This data set has 2 subset in it. First one
includes weather station locations like country, latitude and longitude.
Second one includes measured min, max and mean temperatures from weather
stations.)
*/




//importing sparkts and required java time classes
import java.time.{LocalDateTime, ZoneId, ZonedDateTime}

import com.cloudera.sparkts._
import com.cloudera.sparkts.stats.TimeSeriesStatisticalTests

import java.time.{LocalDateTime, ZoneId, ZonedDateTime}
import com.cloudera.sparkts._
import com.cloudera.sparkts.stats.TimeSeriesStatisticalTests
```

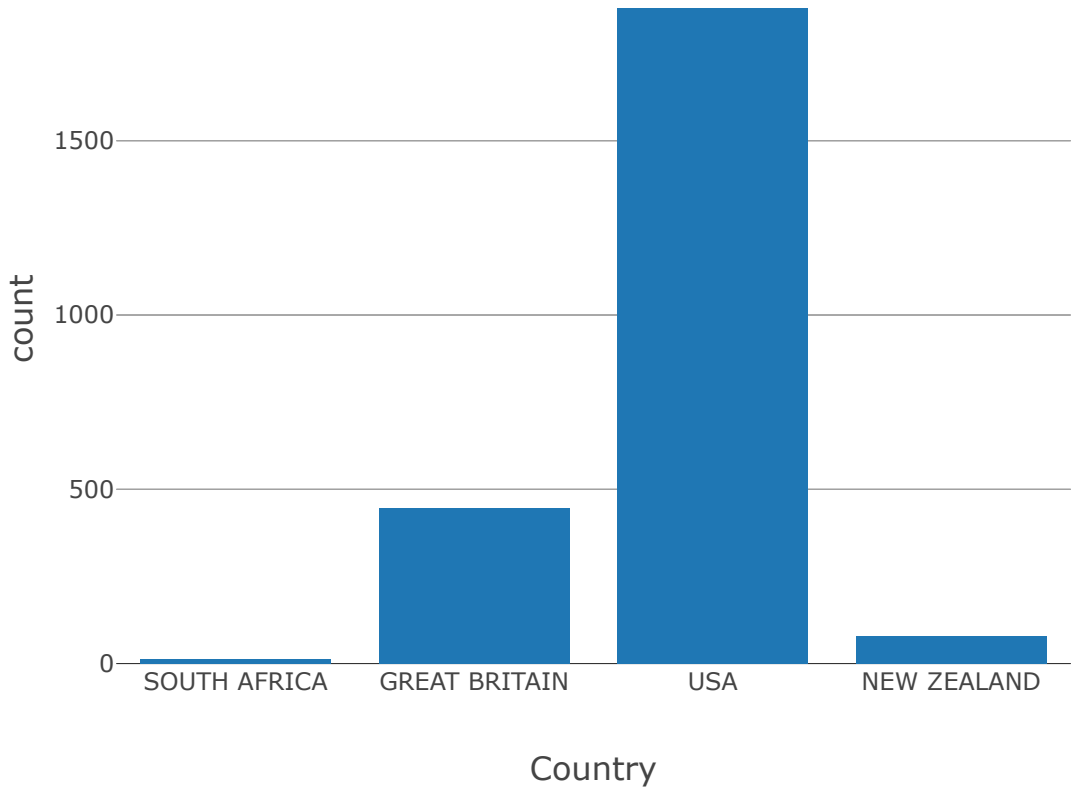
```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._

//bombing data
var aerial = spark.read.
                    format("csv")
                    .option("header","true")
                    .option("inferSchema","true")
                    .option("timestampFormat","MM/dd/yyyy")
                    .option("mode", "PERMISSIVE")

.load("dbfs:/FileStore/shared_uploads/harsh.vijaywat@gmail.com/operations.cs
v")
//weather data that includes locations like country, latitude and longitude.
var weather_station_locations = spark.read.
                    format("csv")
                    .option("header","true")
                    .option("inferSchema","true")
                    .option("timestampFormat","MM/dd/yyyy")
                    .option("mode", "PERMISSIVE")

.load("dbfs:/FileStore/shared_uploads/harsh.vijaywat@gmail.com/Weather_Stati
on_Locations.csv")

val weatherSchema = StructType(Array(StructField("STA", StringType),
StructField("Date", TimestampType), StructField("MeanTemp", DoubleType)))
//Second weather data that includes measured min, max and mean temperatures
var weather = spark.read.
                    format("csv")
                    .option("header","true")
                    .schema(weatherSchema)
                    .option("timestampFormat","yyyy/MM/dd")
                    .option("mode", "PERMISSIVE")

.load("dbfs:/FileStore/shared_uploads/harsh.vijaywat@gmail.com/Summary_of_We
ather.csv")
```

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
aerial: org.apache.spark.sql.DataFrame = [Mission ID: int, Mission Date: tim
estamp ... 44 more fields]
weather_station_locations: org.apache.spark.sql.DataFrame = [WBAN: int, NAM
E: string ... 6 more fields]
weatherSchema: org.apache.spark.sql.types.StructType = StructType(StructFiel
d(STA,StringType,true), StructField(Date,TimestampType,true), StructField(Me
anTemp,DoubleType,true))
```

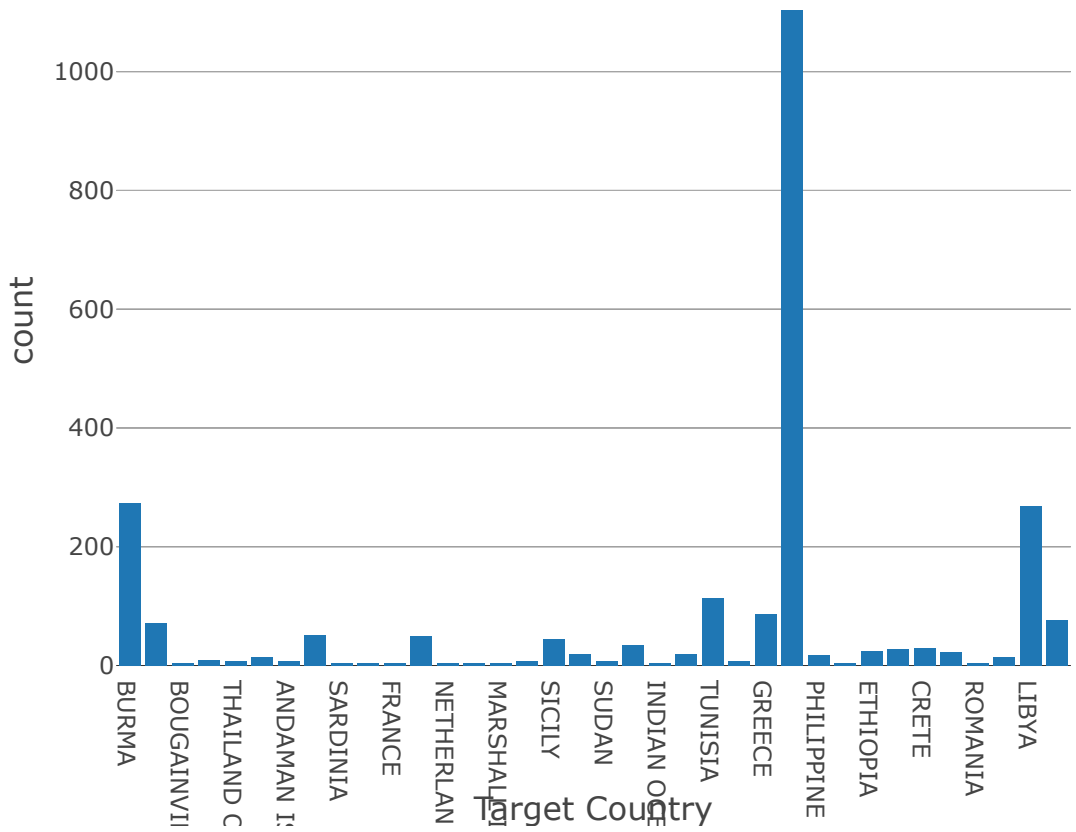
```
weather: org.apache.spark.sql.DataFrame = [STA: string, Date: timestamp ...
1 more field]

/*Data Description

1. Aerial bombing Data description:
Mission Date: Date of mission
Theater of Operations: Region in which active military operations are in
progress; "the army was in the field awaiting action"; Example: "he served
in the Vietnam theater for three years"
Country: Country that makes mission or operation like USA
Air Force: Name or id of air force unity like 5AF
Aircraft Series: Model or type of aircraft like B24
Callsign: Before bomb attack, message, code, announcement, or tune that is
broadcast by radio.
Takeoff Base: Takeoff airport name like Ponte Olivo Airfield
Takeoff Location: takeoff region Sicily
Takeoff Latitude: Latitude of takeoff region
Takeoff Longitude: Longitude of takeoff region
Target Country: Target country like Germany
Target City: Target city like Berlin
Target Type: Type of target like city area
Target Industry: Target industy like town or urban
Target Priority: Target priority like 1 (most)
Target Latitude: Latitude of target
Target Longitude: Longitude of target

2. Weather Condition data description:
Weather station location:
WBAN: Weather station number
NAME: weather station name
STATE/COUNTRY ID: acronym of countries
Latitude: Latitude of weather station
Longitude: Longitude of weather station

3. Weather:
STA: eather station number (WBAN)
Date: Date of temperature measurement
MeanTemp: Mean temperature */
```

```
/* Data Cleaning

Aerial Bombing data includes a lot of NaN value. Instead of using them,
drop some NaN values. It does not only remove the uncertainty but it also
ease visualization process.
Drop countries that are NaN
Drop if target longitude is NaN
Drop if takeoff longitude is NaN
Drop unused features
*/


//drop if target longitude, takeoff longitude ,Country are NaN
aerial = aerial.na.drop("any", Array("Country", "Target Latitude", "Takeoff
Longitude"))
//drop unused features
aerial = aerial.drop("Mission ID", "Unit ID", "Target ID", "Altitude
(Hundreds of Feet)", "Attacking Aircraft", "Bombing Aircraft", "Aircraft
Returned", "Aircraft Failed", "Aircraft Damaged", "Aircraft Lost", "High
Explosives", "High Explosives Type", "Mission Type", "High Explosives Weight
(Pounds)", "High Explosives Weight (Tons)", "Incendiary Devices",
"Incendiary Devices Type", "Incendiary Devices Weight (Pounds)", "Incendiary
Devices Weight (Tons)", "Fragmentation Devices", "Fragmentation Devices
Type", "Fragmentation Devices Weight (Pounds)", "Fragmentation Devices
Weight (Tons)", "Total Weight (Pounds)", "Total Weight (Tons)", "Time Over
Target", "Bomb Damage Assessment", "Source ID" )
//drop  takeoff latitude=4248, takeoff longitude=1355
aerial = aerial.filter("`Takeoff Longitude` != 1355 AND `Takeoff Latitude`
!= 4248")
//select only important columns
weather_station_locations = weather_station_locations.select("WBAN", "NAME",
"STATE/COUNTRY ID", "Latitude", "Longitude")
//select only important columns
weather = weather.select("STA", "Date", "MeanTemp")
//drop nan values in weather
weather = weather.na.drop

aerial: org.apache.spark.sql.DataFrame = [Mission Date: timestamp, Theater o
f Operations: string ... 16 more fields]
aerial: org.apache.spark.sql.DataFrame = [Mission Date: timestamp, Theater o
f Operations: string ... 16 more fields]
aerial: org.apache.spark.sql.DataFrame = [Mission Date: timestamp, Theater o
f Operations: string ... 16 more fields]
weather_station_locations: org.apache.spark.sql.DataFrame = [WBAN: int, NAM
E: string ... 3 more fields]
weather: org.apache.spark.sql.DataFrame = [STA: string, Date: timestamp ...
1 more field]
weather: org.apache.spark.sql.DataFrame = [STA: string, Date: timestamp ...
1 more field]
```

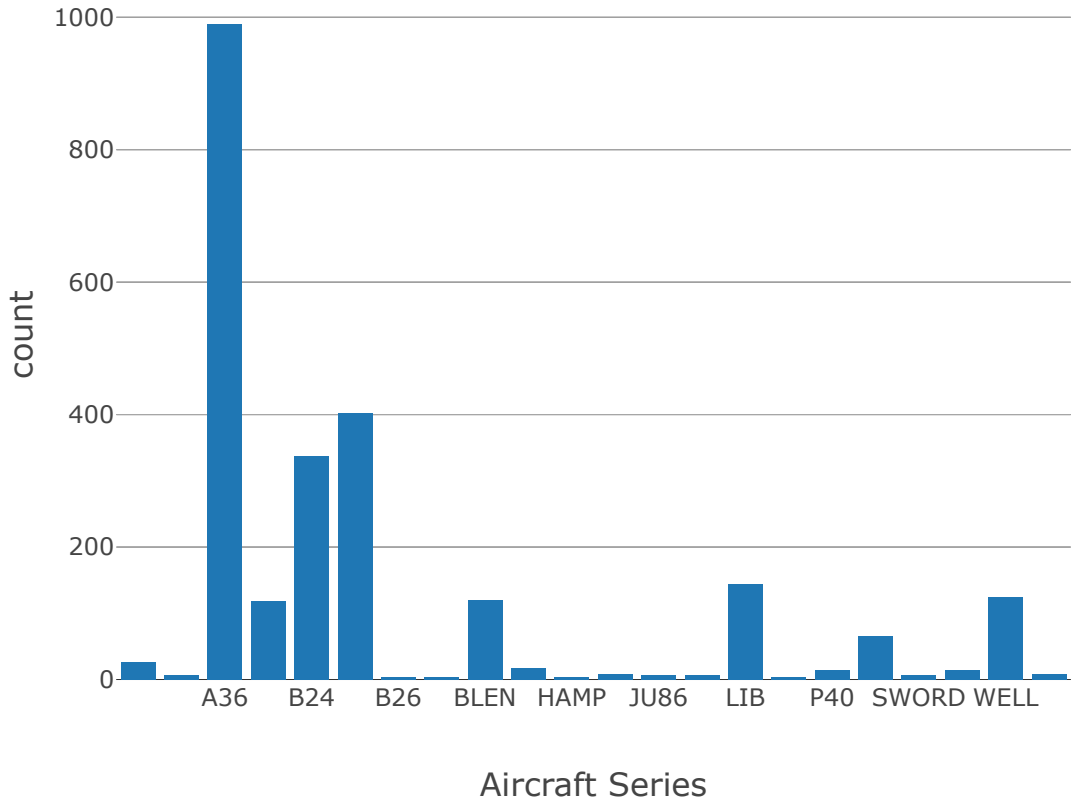
```
//Data Visualization


//How many country which attacks
display(aerial.groupBy("Country").count())
```




```
//Top target countries
display(aerial.groupBy("Target Country").count())
```

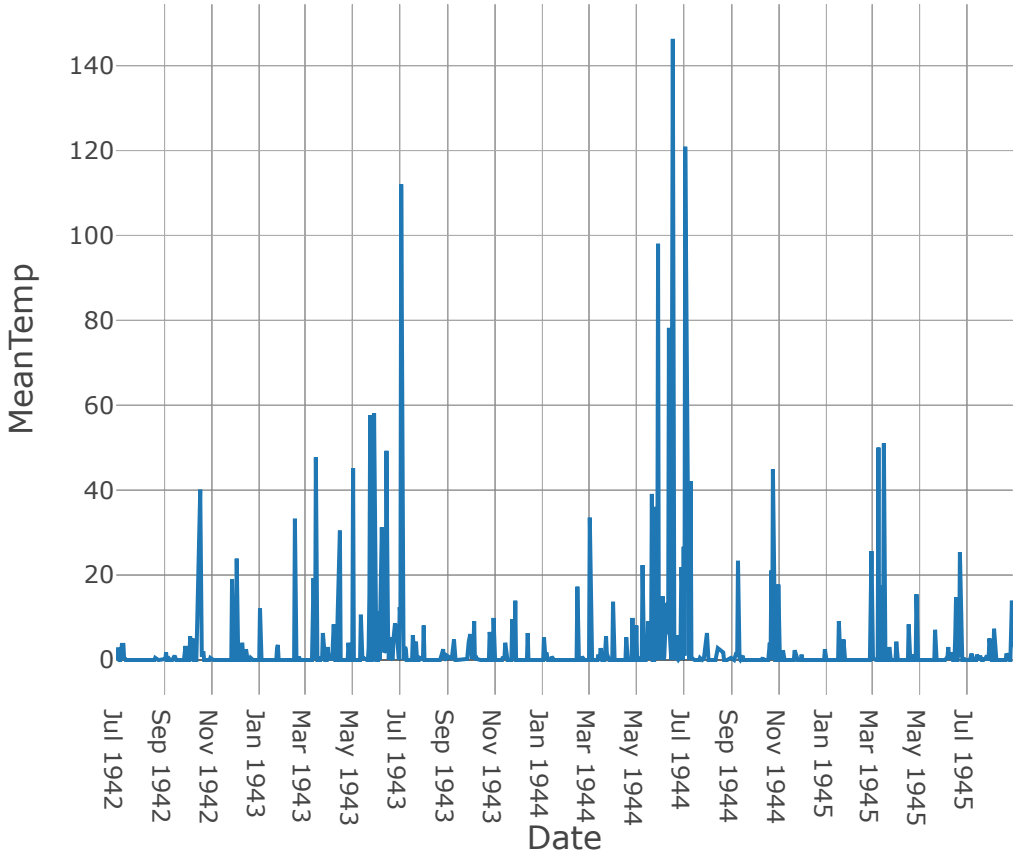



```
//Top 10 aircraft series
display(aerial.groupBy("Aircraft Series").count().orderBy("Aircraft
Series"))
```




```
//MeanTemp as function of date
display(weather.select("Date", "MeanTemp"))
```

Showing sample based on the first 1000 rows.




```
weather.show(10)
```

```
+-----+-------------------+--------+
|  STA|               Date|MeanTemp|
+-----+-------------------+--------+
|10001|1942-07-01 00:00:00|   1.016|
|10001|1942-07-02 00:00:00|     0.0|
|10001|1942-07-03 00:00:00|    2.54|
|10001|1942-07-04 00:00:00|    2.54|
|10001|1942-07-05 00:00:00|     0.0|
|10001|1942-07-06 00:00:00|     0.0|
|10001|1942-07-08 00:00:00|   3.556|
|10001|1942-07-10 00:00:00|   3.556|
|10001|1942-07-11 00:00:00|     0.0|
|10001|1942-07-12 00:00:00|   0.508|
+-----+-------------------+--------+
only showing top 10 rows
```

```
//Time Series Prediction with ARIMA
```

/*Stationarity of a Time Series

There are three basic criterion for a time series to understand whether it is stationary series or not.
Statistical properties of time series such as mean, variance should remain constant over time to call time series is stationary
constant mean
constant variance
autocovariance that does not depend on time. autocovariance is covariance between time series and lagged time series.

Lets visualize and check seasonality trend of our time series. */

/*We can check stationarity using the following methods:

1.Plotting Rolling Statistics: We have a window lets say window size is 6 and then we find rolling mean and variance to check stationary.
2.Dickey-Fuller Test: The test results comprise of a Test Statistic and some Critical Values for difference confidence levels. If the test statistic is less than the critical value, we can say that time series is stationary. */

```
//importing flint packages/classes  for rolling statistics
import com.twosigma.flint.timeseries.TimeSeriesRDD
import scala.concurrent.duration._  // for defined value of DAYS
val tsRdd = TimeSeriesRDD.fromDF(weather)(isSorted = false, timeUnit = DAYS, timeColumn = "Date")
```

```
import com.twosigma.flint.timeseries.TimeSeriesRDD
import scala.concurrent.duration._
tsRdd: com.twosigma.flint.timeseries.TimeSeriesRDD = com.twosigma.flint.timeseries.TimeSeriesRDDImpl@238674dc
```

```
tsRdd.toDF.show(10)
```

```
+-------------------+-----+--------+
|               time|  STA|MeanTemp|
+-------------------+-----+--------+
|1940-01-01 00:00:00|10701|    7.62|
|1940-01-01 00:00:00|22504|   2.286|
|1940-01-01 00:00:00|22508|   0.254|
|1940-01-02 00:00:00|10701|   15.24|
|1940-01-02 00:00:00|22504|   5.334|
|1940-01-02 00:00:00|22508|   10.16|
|1940-01-03 00:00:00|10701|    25.4|
|1940-01-03 00:00:00|22504|     0.0|
|1940-01-04 00:00:00|10701|     0.0|
|1940-01-04 00:00:00|22504|   7.112|
+-------------------+-----+--------+
```

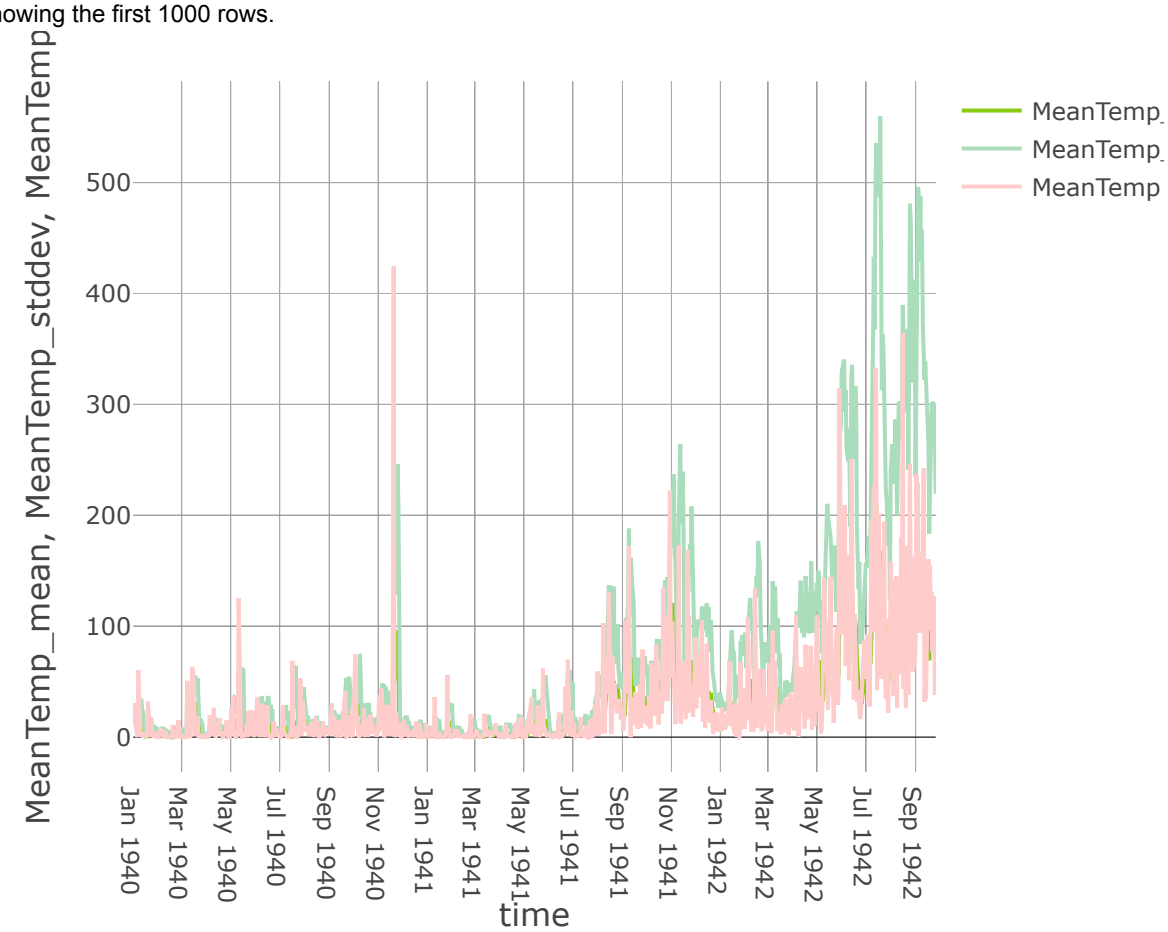
```
only showing top 10 rows
```

```
//Calculation of rolling mean and rolling std of MeanTemp
import com.twosigma.flint.timeseries._
val result = tsRdd.summarizeWindows(
 Windows.pastAbsoluteTime("6days"),
 Summarizers.mean("MeanTemp")).summarizeWindows(
 Windows.pastAbsoluteTime("6days"),
 Summarizers.stddev("MeanTemp"))
```

```
import com.twosigma.flint.timeseries._
result: com.twosigma.flint.timeseries.TimeSeriesRDD = com.twosigma.flint.tim
eseries.TimeSeriesRDDImpl@4c38ca5a
```

```
//visualtisation of rolling stats
display(result.toDF)
```

Aggregated (by sum) in the backend.
Showing the first 1000 rows.



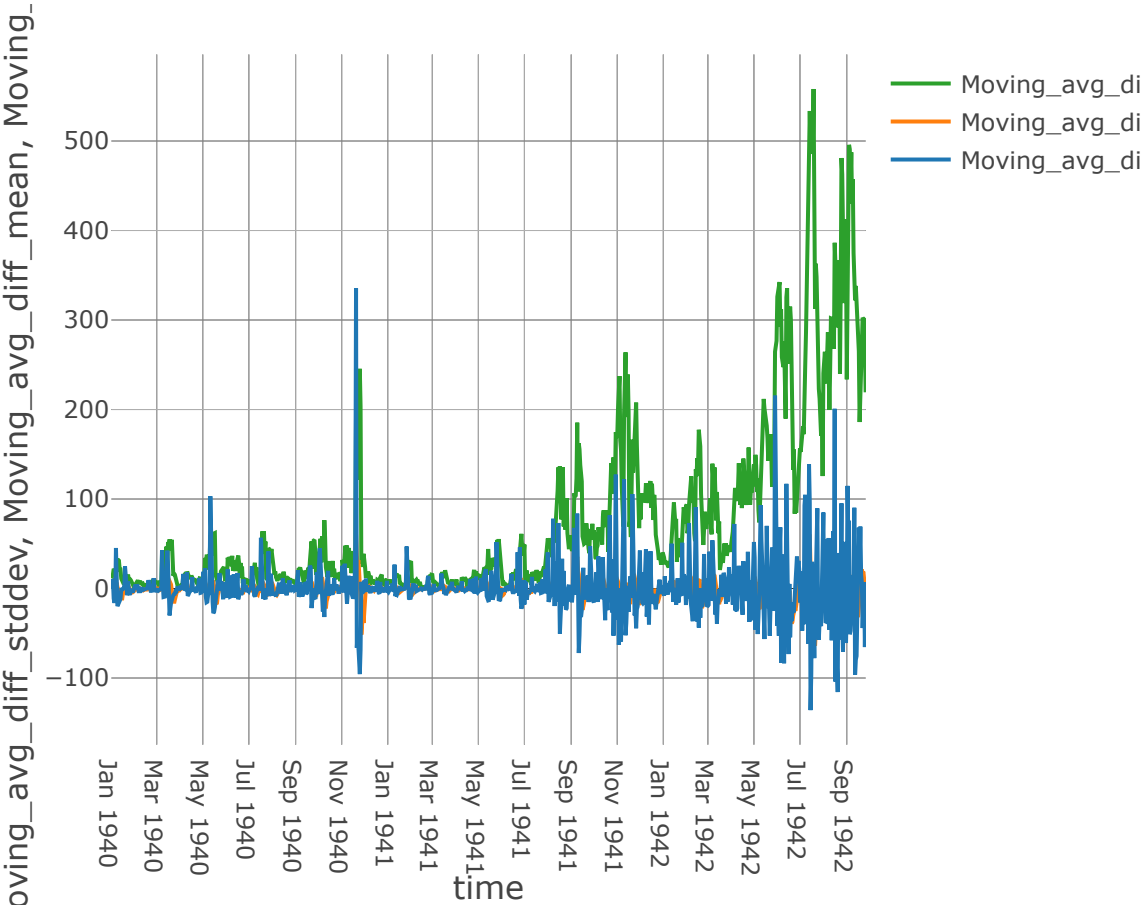
```
/*Our first criteria for stationary is constant mean. So we fail because
mean is not constant as you can see from plot(black line) above . (no
stationary)
Second one is constant variance. We fail this too because std  is not
constant.
As a result, we sure that our time series is not stationary.
Lets make time series stationary at the next part.
*/



result.toDF.show(10)
```

```
+-------------------+-----+--------+------------------+-----------------+
|               time|  STA|MeanTemp|     MeanTemp_mean|  MeanTemp_stddev|
+-------------------+-----+--------+------------------+-----------------+
|1940-01-01 00:00:00|10701|    7.62|3.3866666666666667|3.804351368279924|
|1940-01-01 00:00:00|22504|   2.286|3.3866666666666667|3.804351368279924|
|1940-01-01 00:00:00|22508|   0.254|3.3866666666666667|3.804351368279924|
|1940-01-02 00:00:00|10701|   15.24| 6.815666666666667|5.451047520125528|
|1940-01-02 00:00:00|22504|   5.334| 6.815666666666667|5.451047520125528|
|1940-01-02 00:00:00|22508|   10.16| 6.815666666666667|5.451047520125528|
|1940-01-03 00:00:00|10701|    25.4|           8.28675|8.644466962828222|
|1940-01-03 00:00:00|22504|     0.0|           8.28675|8.644466962828222|
|1940-01-04 00:00:00|10701|     0.0|  6.88109090909091|7.793665857021912|
|1940-01-04 00:00:00|22504|   7.112|  6.88109090909091|7.793665857021912|
+-------------------+-----+--------+------------------+-----------------+
only showing top 10 rows
```

```
/*Make a Time Series Stationary?

First solve trend(constant mean) problem
Most popular method is moving average.
Moving average: We have window that take the average over the past 'n'
sample. 'n' is window size.



//Moving average method
val o = result.toDF.withColumn("Moving_avg_diff", col("MeanTemp")-
col("MeanTemp_mean"))
val tsRdd2 = TimeSeriesRDD.fromDF(o)(isSorted = false, timeUnit = DAYS,
timeColumn = "time")

o: org.apache.spark.sql.DataFrame = [time: timestamp, STA: string ... 4 more
fields]
tsRdd2: com.twosigma.flint.timeseries.TimeSeriesRDD = com.twosigma.flint.tim
eseries.TimeSeriesRDDImpl@3849d15
```

```
val result2 = tsRdd2.summarizeWindows(
 Windows.pastAbsoluteTime("6days"),
 Summarizers.mean("Moving_avg_diff")).summarizeWindows(
 Windows.pastAbsoluteTime("6days"),
 Summarizers.stddev("Moving_avg_diff"))

result2: com.twosigma.flint.timeseries.TimeSeriesRDD = com.twosigma.flint.timeseries.TimeSeriesRDDImpl@25e95692

display(result2.toDF.select("time","Moving_avg_diff","Moving_avg_diff_mean","Moving_avg_diff_stddev"))
//Constant mean criteria: mean looks like constant as you can see from plot(orange line) above . (yes stationary)
```

Aggregated (by sum) in the backend.
Showing the first 1000 rows.



/*Forecasting a Time Series

For prediction(forecasting) we will use ts_diff time series that is result of moving avg method.
Also prediction method is ARIMA that is Auto-Regressive Integrated Moving Averages.
AR: Auto-Regressive (p): AR terms are just lags of dependent variable. For example lets say p is 3, we will use x(t-1), x(t-2) and x(t-3) to predict x(t)
I: Integrated (d): These are the number of nonseasonal differences. For example, in our case we take the first order difference. So we pass that variable and put d=0
MA: Moving Averages (q): MA terms are lagged forecast errors in prediction equation.
(p,d,q) is parameters of ARIMA model.
In order to choose p,d,q parameters we will use two different plots.
Autocorrelation Function (ACF): Measurement of the correlation between time series and lagged version of time series.
Partial Autocorrelation Function (PACF): This measures the correlation between the time series and lagged version of time series but after eliminating the variations already explained by the intervening comparisons.*/

//we need array of double to calculate acf and pacf (due to scala APIs)

```
val cc =
result2.toDF.select("Moving_avg_diff").rdd.collect.map(_.getDouble(0))
```

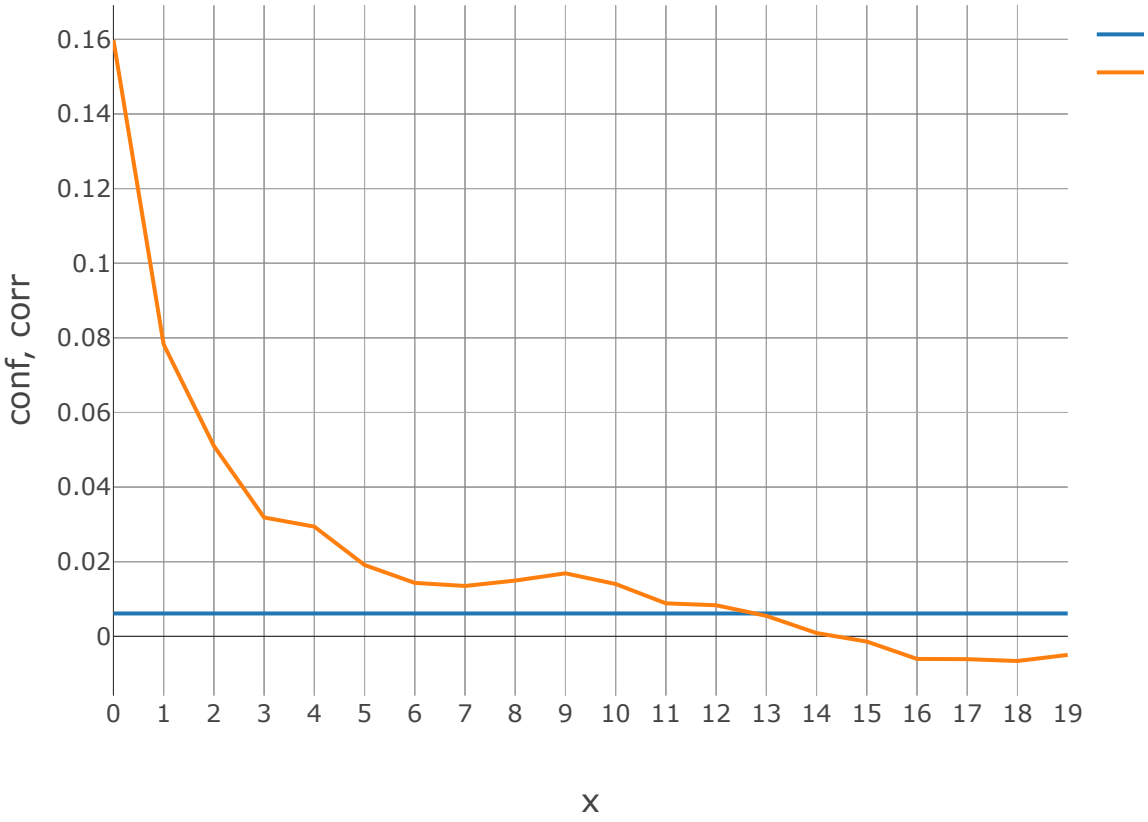
```
cc: Array[Double] = Array(4.233333333333333, -1.1006666666666667, -3.1326666
666666667, 8.424333333333333, -1.4816666666666674, 3.344333333333333, 17.113
25, -8.28675, -6.88109090909091, 0.2309090909090905, -4.59509090909091, -5.4
24714285714286, -5.424714285714286, -5.170714285714286, -4.746625, -4.74662
5, 15.536333333333333, 29.760333333333335, -9.186333333333334, 11.8956666666
66667, 8.847666666666665, -7.662333333333334, -5.122333333333334, -6.9003333
33333334, -5.948947368421053, -5.948947368421053, -5.440947368421053, -5.547
894736842106, -4.785894736842106, -4.531894736842106, -5.8420000000000005, -
5.8420000000000005, -5.8420000000000005, -5.8420000000000005, -2.72344444444
44446, 1.594555555555555, -0.5503333333333337, -0.5503333333333337, -0.55033
33333333337, -0.38847058823529446, -0.38847058823529446, -0.4603750000000003
7, 0.8096249999999996, -0.135466666666667, -0.389466666666667, 0.93662499999
99998, 15.668624999999999, 8.302624999999999, -2.7491764705882353, -2.241176
4705882353, 5.632823529411764, -2.4951764705882353, -2.4951764705882353, -2.
4951764705882353, -2.4951764705882353, -2.4951764705882353, -3.3584444444444
45, 2.7375555555555553, 8.579555555555556, 1.6227777777777774, -1.4252222222
222222, -3.697111111111111, -3.697111111111111, -2.1025555555555555, -0.8325
555555555555, -0.07055555555555548, -1.6086666666666667, -1.608666666666666
7, -1.6086666666666667, -0.5221111111111112, 0.23988888888888882, 1.08655555
55555555, -1.9614444444444445, -1.9614444444444445, -0.9595555555555557, -0.
9595555555555557, -0.9595555555555557, -0.5677647058823531, -0.5080000000000
```

```
//importing required dependencies
import org.apache.commons.math3.distribution.NormalDistribution
import com.cloudera.sparkts.models.Autoregression
import org.apache.spark.mllib.linalg._
//method for finding confident interval value (same for +ve and -ve)
def calcConfVal(conf: Double, n: Int): Double = {
    val stdNormDist = new NormalDistribution(0, 1)
    val pVal = (1 - conf) / 2.0
    stdNormDist.inverseCumulativeProbability(1 - pVal) / Math.sqrt(n)
  }
```

```
import org.apache.commons.math3.distribution.NormalDistribution
import com.cloudera.sparkts.models.Autoregression
import org.apache.spark.mllib.linalg._
calcConfVal: (conf: Double, n: Int)Double

def acf(data: Array[Double], maxLag: Int, conf: Double = 0.95) = {
    // calculate correlations and confidence bound
    val autoCorrs = UnivariateTimeSeries.autocorr(data, maxLag)
    val confVal = calcConfVal(conf, data.length)
 }

 def pacf(data: Array[Double], maxLag: Int, conf: Double = 0.95) = {
    // create AR(maxLag) model, retrieve coefficients and calculate
confidence bound
    val model = Autoregression.fitModel(new DenseVector(data), maxLag)
    val pCorrs = model.coefficients // partial autocorrelations are the
coefficients in AR(n) model
    val confVal = calcConfVal(conf, data.length)
 }
```

```
acf: (data: Array[Double], maxLag: Int, conf: Double)Unit
pacf: (data: Array[Double], maxLag: Int, conf: Double)Unit

//calculating acf
val autoC = UnivariateTimeSeries.autocorr(cc, 20)

autoC: Array[Double] = Array(0.15985960653767617, 0.07822934497499003, 0.051
04178272023124, 0.03185898579582731, 0.02940081553604699, 0.0191147184851530
9, 0.014325336877323201, 0.01350030016098633, 0.014950189714590315, 0.016887
683052156863, 0.014020513417277041, 0.0088403512962757, 0.00833189049362899
5, 0.005490817537416041, 8.82023584502792E-4, -0.0014000519138274822, -0.006
05184528400493, -0.006115838711134724, -0.006609667118626515, -0.00497244817
8312112)
```

```
//calculating confident interval value for 5%
val confVal2 = calcConfVal(0.95, cc.length)

confVal2: Double = 0.006128269705725016

//case class for reconverting Array[Double] to Dataframe for visualisation
case class acfclass(x:Double, corr:Double, conf:Double)  // x is lag value,
corr is autocorrelation, conf is confidence interval value

defined class acfclass

//collection of acfclass objects
val f = for(i <- 0 to 19)yield{
  acfclass(i, autoC(i) ,confVal2 )

}

f: scala.collection.immutable.IndexedSeq[acfclass] = Vector(acfclass(0.0,0.1
5985960653767617,0.006128269705725016), acfclass(1.0,0.07822934497499003,0.0
06128269705725016), acfclass(2.0,0.05104178272023124,0.006128269705725016),
acfclass(3.0,0.03185898579582731,0.006128269705725016), acfclass(4.0,0.02940
081553604699,0.006128269705725016), acfclass(5.0,0.01911471848515309,0.00612
8269705725016), acfclass(6.0,0.014325336877323201,0.006128269705725016), acf
class(7.0,0.01350030016098633,0.006128269705725016), acfclass(8.0,0.01495018
9714590315,0.006128269705725016), acfclass(9.0,0.016887683052156863,0.006128
269705725016), acfclass(10.0,0.014020513417277041,0.006128269705725016), acf
class(11.0,0.0088403512962757,0.006128269705725016), acfclass(12.0,0.0083318
90493628995,0.006128269705725016), acfclass(13.0,0.005490817537416041,0.0061
28269705725016), acfclass(14.0,8.82023584502792E-4,0.006128269705725016), ac
fclass(15.0,-0.0014000519138274822,0.006128269705725016), acfclass(16.0,-0.0
0605184528400493,0.006128269705725016), acfclass(17.0,-0.006115838711134724,
0.006128269705725016), acfclass(18.0,-0.006609667118626515,0.006128269705725
016), acfclass(19.0,-0.004972448178312112,0.006128269705725016))

//plot acf graph
display(f.toSeq.toDF)
```

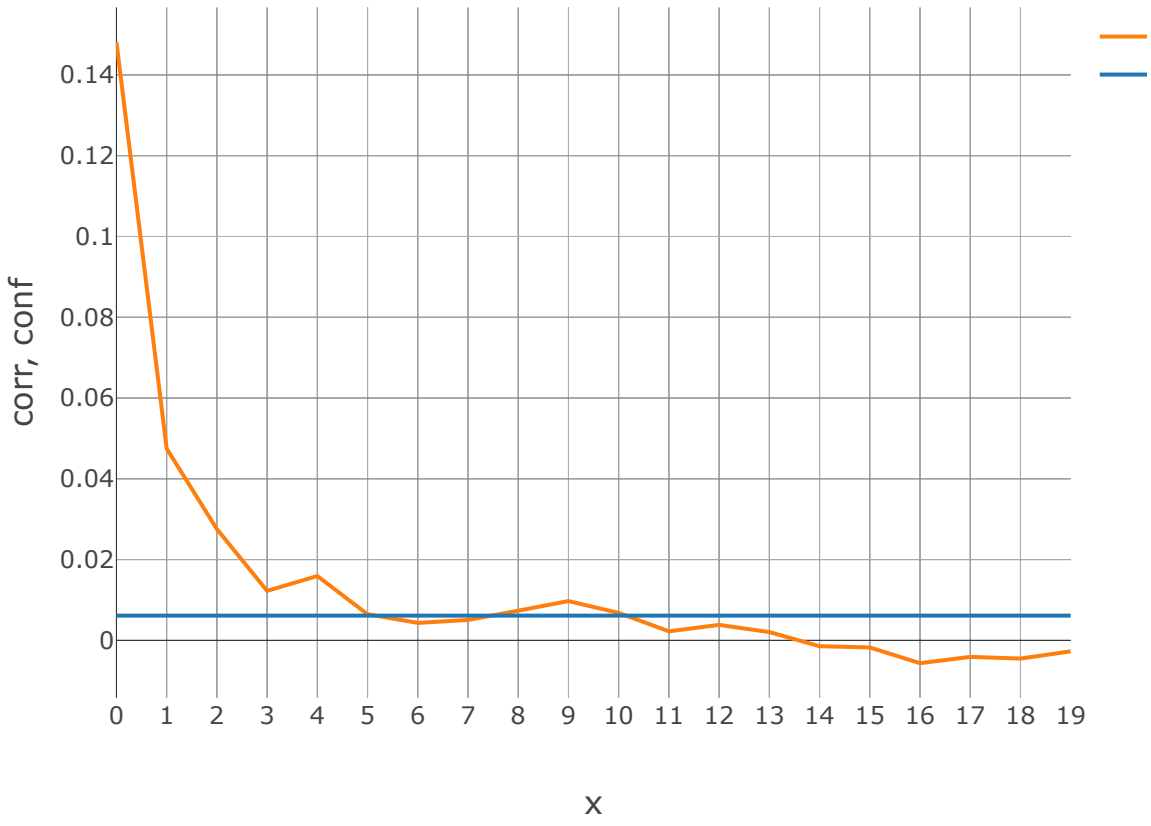



```
//acf graph crosses the confident line for the first time at lag value x=
13, therefore q=13


//calculating pacf
// create AR(maxLag) model, retrieve coefficients and calculate confidence
bound
val model = Autoregression.fitModel(new DenseVector(cc), 20)
    val pCorrs = model.coefficients // partial autocorrelations are the
coefficients in AR(n) model

model: com.cloudera.sparkts.models.ARModel = com.cloudera.sparkts.models.ARM
odel@3d8e1ceb
pCorrs: Array[Double] = Array(0.14814005189294893, 0.04748073410040342, 0.02
7557210554358398, 0.012252914228101056, 0.015928687733959314, 0.006466256757
204596, 0.004326543518545098, 0.005051619760605817, 0.007353556735416871, 0.
00973015466197282, 0.006802798298257134, 0.0022449303174158753, 0.0038286787
543913565, 0.002053598790637832, -0.001450916900002485, -0.0017555517648898
6, -0.005649625633282044, -0.004087619613910831, -0.004504395672759216, -0.0
02715647009149274)
```

```
val f2 = for(i <- 0 to 19)yield{
  acfclass(i, pCorrs(i) ,confVal2 )

}
```

```
f: scala.collection.immutable.IndexedSeq[acfclass] = Vector(acfclass(0.0,0.1
4814005189294893,0.006128269705725016), acfclass(1.0,0.04748073410040342,0.0
06128269705725016), acfclass(2.0,0.027557210554358398,0.006128269705725016),
acfclass(3.0,0.012252914228101056,0.006128269705725016), acfclass(4.0,0.0159
28687733959314,0.006128269705725016), acfclass(5.0,0.006466256757204596,0.00
6128269705725016), acfclass(6.0,0.004326543518545098,0.006128269705725016),
acfclass(7.0,0.005051619760605817,0.006128269705725016), acfclass(8.0,0.0073
53556735416871,0.006128269705725016), acfclass(9.0,0.0097301546619728 2,0.006
128269705725016), acfclass(10.0,0.006802798298257134,0.006128269705725016),
acfclass(11.0,0.0022449303174158753,0.006128269705725016), acfclass(12.0,0.0
038286787543913565,0.006128269705725016), acfclass(13.0,0.00205359879063783
2,0.006128269705725016), acfclass(14.0,-0.001450916900002485,0.0061282697057
25016), acfclass(15.0,-0.00175555176488986,0.006128269705725016), acfclass(1
6.0,-0.005649625633282044,0.006128269705725016), acfclass(17.0,-0.0040876196
13910831,0.006128269705725016), acfclass(18.0,-0.004504395672759216,0.006128
269705725016), acfclass(19.0,-0.002715647009149274,0.006128269705725016))
```

```
display(f2.toSeq.toDF)
```



```
//pacf graph crosses the confident line for the first time at lag value x=
5, therefore  p = 5


//Now lets use p=5, q=13, d=0  as parameters of ARIMA models and predict


//In scala, to apply ARIMA model we need vector therefore we convert column
data to vector
import org.apache.spark.mllib.linalg.Vector
import com.cloudera.sparkts.stats.TimeSeriesStatisticalTests
import org.apache.spark.mllib.linalg.Vectors
val ts =
Vectors.dense(weather.select("MeanTemp").rdd.collect.map(_.getDouble(0)))
```

```
import org.apache.spark.mllib.linalg.Vectors
ts: org.apache.spark.mllib.linalg.Vector = [1.016,0.0,2.54,2.54,0.0,0.0,3.55
6,3.556,0.0,0.508,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.
0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.254,0.0,0.0,0.508,0.254,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.254,0.254,1.778,0.0,0.0,0.254,0.0,0.0,0.0,1.016,0.
0,0.0,0.0,0.0,0.0,0.0,0.254,0.0,0.0,3.302,0.762,0.0,0.0,0.0,5.588,0.0,0.254,5.0
8,2.032,0.0,0.0,25.654,40.132,0.762,2.032,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.508,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,19.05,0.0,0.0,0.0,0.0,0.0,23.876,0.0,0.0,0.0,0.0,0.0,0.0,0.0,4.064,
0.0,0.0,0.0,0.0,0.0,2.54,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.508,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,12.192,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,2.794,3.556,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,33.274,0.0,0.0,0.0,0.0,0.0,
0.762,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,19.30
4,0.0,0.0,47.752,0.0,0.0,0.0,0.0,0.762,0.0,0.0,0.0,0.508,0.0,6.35,3.556,0.0,0.0,0.0,0.0,
0.0,3.048,0.0,0.0,0.762,0.0,0.0,8.382,0.0,0.0,0.0,0.0,0.0,10.16,27.178,30.48,0.
0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,4.064,0.0,0.0,0.0,0.0,0.0,3.556,45.212,0.0,0.0,0.0,0.2
54,0.0,0.0,0.0,0.0,0.0,0.0,0.0,10.668,0.0,0.508,0.0,0.0,0.0,2.794,15.494,57.658,0.25
4,0.0,0.0,0.254,58.166,0.762,0.0,0.0,0.0,11.43,0.0,31.242,17.78,26.924,1.01
6,0.0,2.54,26.67,6.604,5.334,0.254,4.826,7.62,8.636,7.874,6.604,0.0,12.446,1
0.16,90.932,13.462,11.938,0.0,0.508,1.778,2.54,0.0,0.0,0.0,0.0,0.254,0.0,5.8
```

```
//import required dependencies
import com.cloudera.sparkts.stats._
import com.cloudera.sparkts.models._
//fitting ARIMA model
val arimaModel = ARIMA.fitModel(5, 0, 13, ts)
```

```
import com.cloudera.sparkts.stats._
import com.cloudera.sparkts.models._
arimaModel: com.cloudera.sparkts.models.ARIMAModel = com.cloudera.sparkts.mo
dels.ARIMAModel@5a9e3f1f
```

```
//predicting 50 future values
arimaModel.forecast(ts, 50)
```

```
res40: org.apache.spark.mllib.linalg.Vector = [0.5437894491217253,0.83386149
26319519,0.8003718232876662,1.4931783656711775,1.625375177511836,1.441658563
8942383,1.5712270978965954,2.146600709106111,2.264114703199161,1.96791129358
83252,2.164053060205302,1.7882432129383803,1.7396980303137395,1.707344597860
8266,1.6631187669418332,1.648721971220636,1.628127223823239,1.61317940850284
84,1.5993171513222422,1.5876944736854839,1.5775547399869352,1.56884858581358
79,1.561325722616133,1.5548417903051786,1.5492476499496728,1.544423145166277
3,1.5402617191135235,1.5366724695645873,1.5335766453569342,1.530906441171672
6,1.5286033324926431,1.526616854851082,1.5249034767405831,1.523425653017329
1,1.5221509996463907,1.5210515849306858,1.5729947622984033,1.52922549967409
3,1.5481428792490517,1.6394827856928869,1.5980836974432324,1.594806857097427
4,1.5844918598531164,1.553883029050622,1.5465660277847535,1.542748790947846
1,1.5385976715252754,1.5353129042880223,1.5323779239194377,1.582772940338674
3,1.5905477330866533,1.9355958708760326,1.6391159613692743,1.753391264585601
6,1.7012631197017825,1.6012420009533583,1.6235553957818754,1.594166387054367
4,1.7906078767838771,1.6113017147196391,1.6814145253703832,1.619507126064078
3,1.5821626421005524,1.5806842193243427,1.6217188696642513,1.572184241817113
5,1.5848743585605398,2.253080974641941,1.8405909117742394,1.963510998291111
4,1.837008609358858,1.689437786459794,2.8372860821567403,1.8675556580188077,
2.3346163809155374,3.028683642960409,2.4256930531100673,2.468096168025685,2.
2889040460797823,7.339330519689777,11.282884144188024,6.584092125154323,8.53
```