Time Series Analytics with Spark

Simon Ouellette
Faimdata

SPARK SUMMIT EAST 2017

What is spark-timeseries?

https://github.com/sryza/spark-timeseries

- Open source time series library for Apache Spark 2.0
- Sandy Ryza
  - Advanced Analytics with Spark: Patterns for Learning from Data at Scale
  - Senior Data Scientist at Clover Health
- Started in February 2015

Who am I?

http://faimdata.com

- Chief Data Science Officer at Faimdata
- Contributor to spark-timeseries since September 2015
- Participated in early design discussions (March 2015)
- Been an active user for ~2 years

Survey: Who uses time series?

Design Question #1: How do we structure multivariate time series?

Columnar or Row-based?

Columnar representation

```
TimeSeriesRDD(
     DateTimeIndex,
     RDD[Vector]
)
```

DateTime Index	Vector for Series 1	Vector for Series 2
2:30:01	4.56	78.93
2:30:02	4.57	79.92
2:30:03	4.87	79.91
2:30:04	4.48	78.99

Row-based representation

```
RDD[(ZonedDateTime, Vector)]
```

Vectors	Date/ Time	Series 1	Series 2
Vector 1	2:30:01	4.56	78.93
Vector 2	2:30:02	4.57	79.92
Vector 3	2:30:03	4.87	79.91
Vector 4	2:30:04	4.48	78.99

Columnar vs Row-based

More efficient in columnar representation:

- Lagging
- Differencing
- Rolling operations
- Feature generation
- Feature selection
- Feature transformation

More efficient in row-based representation:

- Regression
- Clustering
- Classification
- Etc.

Example: lagging operation

Row-based representation

- Time complexity: O(N)

(assumes pre-sorted RDD)

- For each row, we need to get values from previous **k** rows

Columnar representation

- Time complexity: O(K)
- For each column to lag, we truncate most recent **k** values, and truncate the DateTimeIndex's oldest **k** values.

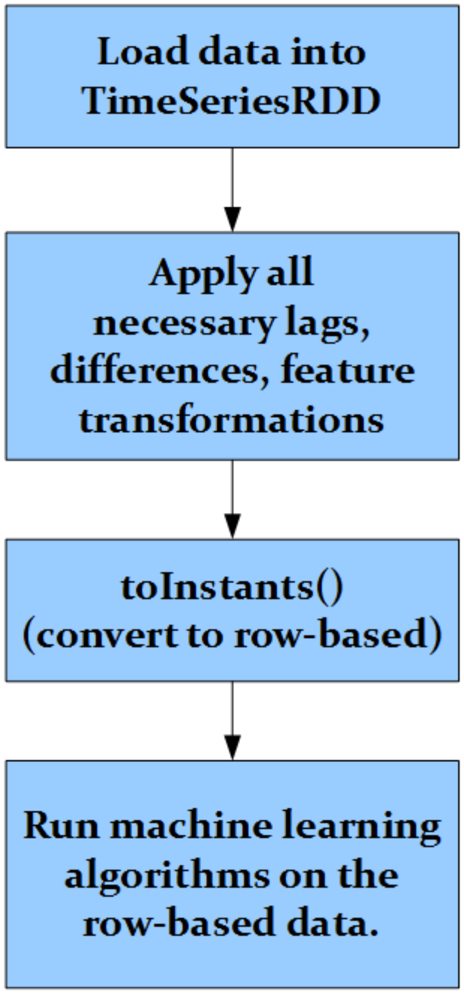
Example: regression

- We're estimating: $$y_t = \alpha + \sum_I \sum_J \beta_{ij} x_{i(t-j)}$$
- The lagged values are typically part of each row, because they are pre-generated as new features.
- **Stochastic Gradient Descent**: we iterate on examples and estimate error gradient to adjust weights, which means that we care about rows, not columns.
- To avoid shuffling, the partitioning must be done such that all elements of a row are together in the same partition (so the gradient can be computed locally).

Current solution

- Core representation is columnar.
- Utility functions to go to/from row-based.
- **Reasoning**: spark-timeseries operations are mostly time-related, i.e. columnar. Row-based operations are about relationships between the variables (ML/statistical), thus external to spark-timeseries.

Typical time series analytics workflow:



Survey: Who uses univariate time series that don't fit inside a single executor's memory?

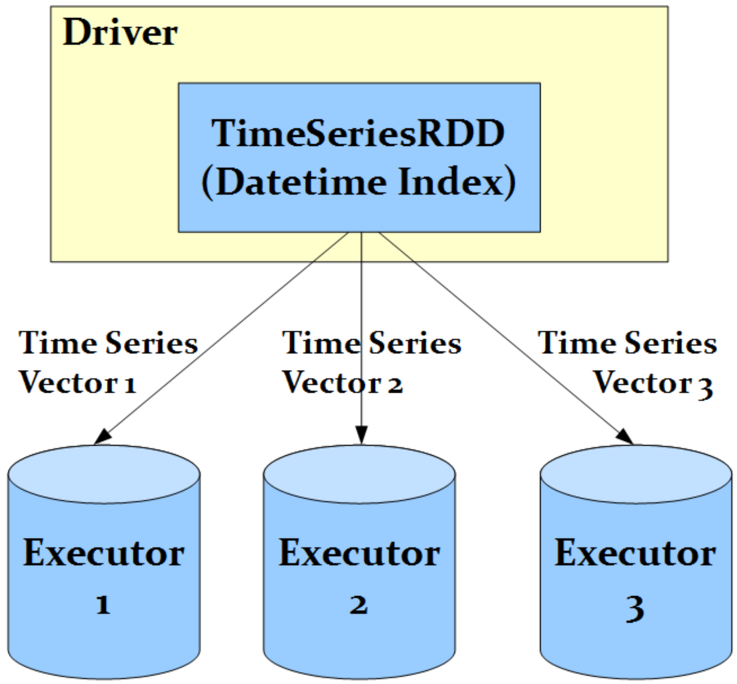
(or multivariate of which a single variable's time series doesn't fit)

Design Question #2: How do we partition the multi-variate time series for distributed processing?

Across features, or across time?

Current design

Assumption: a single time series must fit inside executory memory!



Current design

Assumption: a single time series must fit inside executory memory!

```
TimeSeriesRDD (
    DatetimeIndex,
    RDD[(K, Vector)]
)

IrregularDatetimeIndex (
    Array[Long],        // Other limitation: Scala arrays = 2^32 elements
    java.time.ZoneId
)
```

Future improvements

- Creation of a new TimeSeriesRDD-like class that will be longitudinally (i.e. across time) partitioned rather than horizontally (i.e. across features).

- Keep both types of partitioning, on a case-by-case basis.

Design Question #3: How do we lag, difference, etc.?

Re-sampling, or index-preserving?

Option #1: re-sampling

Before

Irregular Time	y value at t	x value at t
1:30:05	51.42	4.87
1:30:07.86	52.37	4.99
1:30:07.98	53.22	4.95
1:30:08.04	55.87	4.97
1:30:12	54.84	5.12
1:30:14	49.88	5.10

After (1 second lag)

Uniform Time	y value at t	x value at (t – 1)
1:30:06	51.42	4.87
1:30:07	51.42	4.87
1:30:08	53.22	4.87
1:30:09	55.87	4.95
1:30:10	55.87	4.97
1:30:11	55.87	4.97
1:30:12	54.84	4.97
1:30:13	54.84	5.12
1:30:14	49.88	5.12

Option #2: index preserving

Before

Irregular Time	y value at t	x value at t
1:30:05	51.42	4.87
1:30:07.86	52.37	4.99
1:30:07.98	53.22	4.95
1:30:08.04	55.87	4.97
1:30:12	54.84	5.12
1:30:14	49.88	5.10

After (1 second lag)

Irregular Time	y value at t	x value at (t – 1)
1:30:05	51.42	N/A
1:30:07.86	52.37	4.87
1:30:07.98	53.22	4.87
1:30:08.04	55.87	4.87
1:30:12	54.84	4.97
1:30:14	49.88	5.12

Current functionality

- Option #1: resample() function for lagging/differencing by upsampling/downsampling.
  - Custom interpolation function (used when downsampling)

- Conceptual problems:
  - Information loss and duplication (downsampling)
  - Bloating (upsampling)

Current functionality

- Option #2: functions to lag/difference irregular time series based on arbitrary time intervals. (preserves index)

- Same thing: custom interpolation function can be passed for when downsampling occurs.

Overview of current API

High-level objects

- TimeSeriesRDD
- TimeSeries
- TimeSeriesStatisticalTests
- TimeSeriesModel
- DatetimeIndex
- UnivariateTimeSeries

TimeSeriesRDD

- collectAsTimeSeries
- filterStartingBefore, filterStartingAfter, slice
- filterByInstant
- quotients, differences, lags
- fill: fills NaNs by specified interpolation method (*linear, nearest, next, previous, spline, zero*)
- mapSeries
- seriesStats: min, max, average, std. deviation
- toInstants, toInstantsDataFrame
- resample
- rollSum, rollMean
- saveAsCsv, saveAsParquetDataFrame

TimeSeriesStatisticalTests

- Stationarity tests:
  - Augmented Dickey-Fuller (adftest)
  - KPSS (kpsstest)

- Serial auto-correlation tests:
  - Durbin-Watson (dwtest)
  - Breusch-Godfrey (bgtest)
  - Ljung-Box (lbtest)

- Breusch-Pagan heteroskedasticity test (bptest)
- Newey-West variance estimator (neweyWestVarianceEstimator)

TimeSeriesModel

- AR, ARIMA
- ARX, ARIMAX (i.e. with exogenous variables)
- Exponentially weighted moving average
- Holt-winters method (triple exp. smoothing)
- GARCH(1,1), ARGARCH(1,1,1)

Others

- Java bindings
- Python bindings
- YAHOO financial data parser

Code example #1

Time	Y	X
12:45:01	3.45	25.0
12:46:02	4.45	30.0
12:46:58	3.45	40.0
12:47:45	3.00	35.0
12:48:05	4.00	45.0

Y is stationary
X is integrated of order 1

Code example #1

```
val ts = TimeSeriesRDD.timeSeriesRDDFromCsv("mydata.csv", sc)

val newIndex = ts.index.islice(1, ts.index.size)

val tsTransformed = ts.mapSeries(vec => {
  val result = TimeSeriesStatisticalTests.adftest(vec, 0, "c")
  if (result._2 > 0.05) differencesAtLag(vec, 1) else vec
}, newIndex).lags(2, Map(("y" -> true)))

val instantsAsLPs = tsTransformed.toInstants().map(row =>
  LabeledPoint(row._2(0), Vectors.dense(row._2.toArray.drop(1))))

val algo = new LassoWithSGD().setIntercept(true)
algo.optimizer.setRegParam(0.5)
val model = algo.run(instantsAsLPs)
```

Code example #1

Time	y	d(x)	Lag1(y)	Lag2(y)	Lag1(d(x))	Lag2(d(x))
12:45:01	3.45					
12:46:02	4.45	5.0	3.45			
12:46:58	3.45	10.0	4.45	3.45	5.0	
12:47:45	3.00	-5.0	3.45	4.45	10.0	5.0
12:48:05	4.00	10.0	3.00	3.45	-5.0	10.0

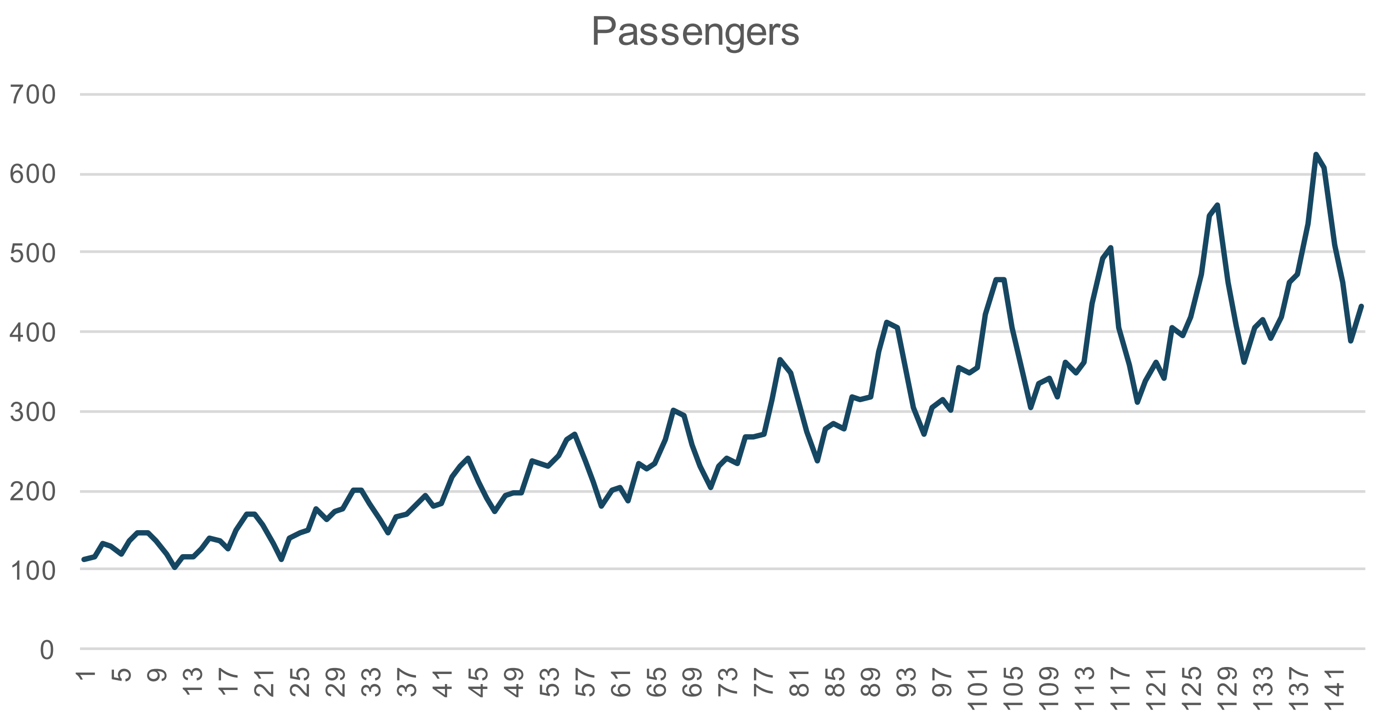
Code example #2

- We will use Holt-Winters to forecast some seasonal data.

- Holt-winters: exponential moving average applied to level, trend and seasonal component of the time series, then combined into global forecast.

$$
\begin{aligned}
l_x &= \alpha(y_x - s_{x-L}) + (1-\alpha)(l_{x-1} + b_{x-1}) && \text{level} \\
b_x &= \beta(l_x - l_{x-1}) + (1-\beta)b_{x-1} && \text{trend} \\
s_x &= \gamma(y_x - l_x) + (1-\gamma)s_{x-L} && \text{seasonal} \\
\hat{y}_{x+m} &= l_x + mb_x + s_{x-L+1+(m-1)mod(L)} && \text{forecast}
\end{aligned}
$$

Code example #2



Code example #2

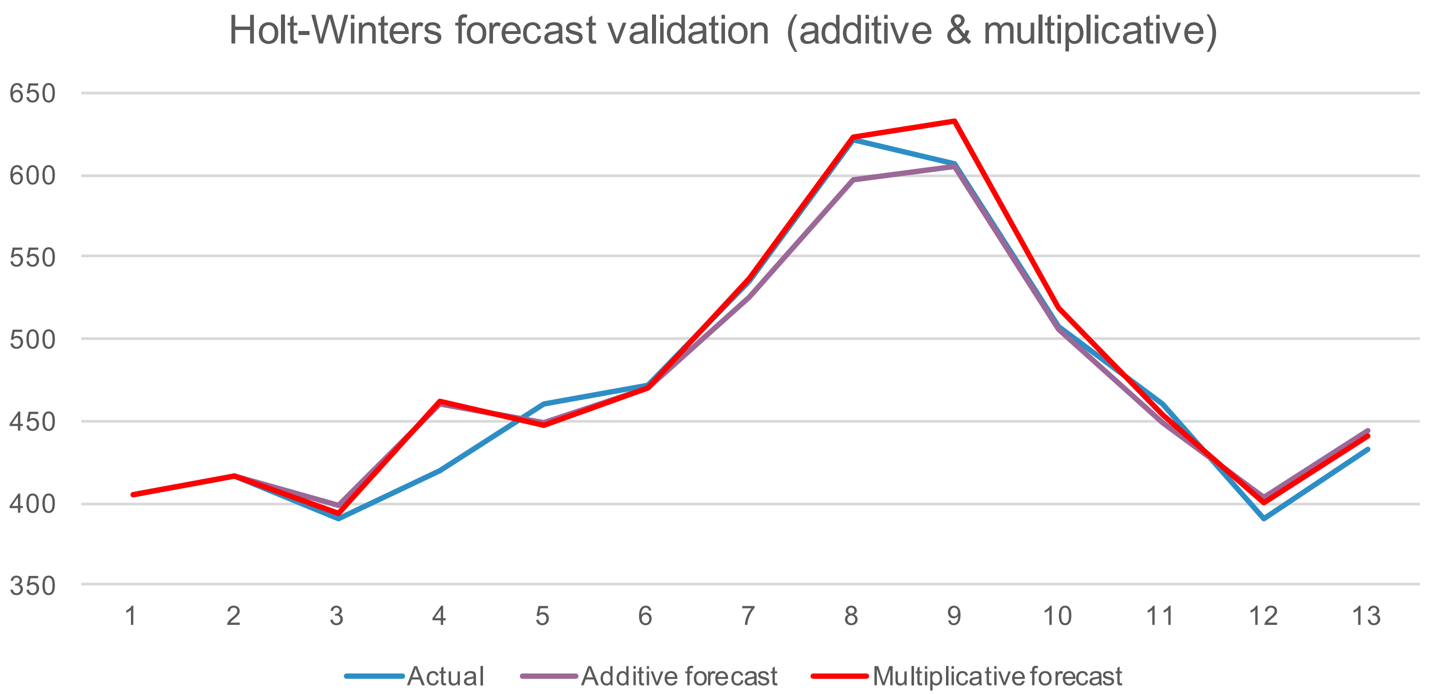
```
val period = 12
val model = HoltWinters.fitModel(tsAirPassengers, period, "additive", "BOBYQA")

val additive_forecasted = new DenseVector(new Array[Double](period))
model.forecast(tsAirPassengers, additive_forecasted)

val model2 = HoltWinters.fitModel(tsAirPassengers, period, "multiplicative", "BOBYQA")

val mult_forecasted = new DenseVector(new Array[Double](period))
model2.forecast(tsAirPassengers, mult_forecasted)
```

Code example #2



Thank You.

e-mail: souellette@faimdata.com

SPARK SUMMIT EAST 2017