ASOF Joins

For those unfamiliar with the term, ASOF joins are temporal join operations based on inexact matching of timestamps. Within the context of Apache Spark, a join operation, loosely speaking, matches records from two data frames (let's call them `left` and `right`) based on some criteria. A temporal join implies matching records in `left` and `right` based on timestamps, and with inexact matching of timestamps permitted, it is typically useful to join `left` and `right` along one of the following temporal directions:

1. Looking behind: if a record from `left` has timestamp `t`, then it gets matched with ones from `right` having the most recent timestamp less than or equal to `t`.
2. Looking ahead: if a record from `left` has timestamp `t`, then it gets matched with ones from `right` having the smallest timestamp greater than or equal to (or alternatively, strictly greater than) `t`.

However, oftentimes it is not useful to consider two timestamps as "matching" if they are too far apart. Therefore, an additional constraint on the maximum amount of time to look behind or look ahead is usually also part of an ASOF join operation.

In `sparklyr.flint` 0.2, all ASOF join functionalities of Flint are accessible via the `asof_join()` method. For example, given 2 timeseries RDDs `left` and `right`:

```
library(sparklyr)
library(sparklyr.flint)

sc <- spark_connect(master = "local")
left <- copy_to(sc, tibble::tibble(t = seq(10), u = seq(10))) %>%
  from_sdf(is_sorted = TRUE, time_unit = "SECONDS", time_column = "t")
right <- copy_to(sc, tibble::tibble(t = seq(10) + 1, v = seq(10) + 1L))
%>%
  from_sdf(is_sorted = TRUE, time_unit = "SECONDS", time_column = "t")
```

The following prints the result of matching each record from `left` with the most recent record(s) from `right` that are at most 1 second behind.

```
print(asof_join(left, right, tol = "1s", direction = ">=") %>%
to_sdf())
```

```
## # Source: spark [?? x 3]
##    time                    u     v
##
##  1 1970-01-01 00:00:01     1    NA
##  2 1970-01-01 00:00:02     2     2
##  3 1970-01-01 00:00:03     3     3
##  4 1970-01-01 00:00:04     4     4
##  5 1970-01-01 00:00:05     5     5
##  6 1970-01-01 00:00:06     6     6
##  7 1970-01-01 00:00:07     7     7
##  8 1970-01-01 00:00:08     8     8
##  9 1970-01-01 00:00:09     9     9
## 10 1970-01-01 00:00:10    10    10
```

Whereas if we change the temporal direction to "<", then each record from `left` will be matched with any record(s) from `right` that is strictly in the future and is at most 1 second ahead of the current record from `left`:

```
print(asof_join(left, right, tol = "1s", direction = "<") %>% to_sdf())
```

```
## # Source: spark [?? x 3]
##    time                    u     v
##
##  1 1970-01-01 00:00:01     1     2
##  2 1970-01-01 00:00:02     2     3
##  3 1970-01-01 00:00:03     3     4
##  4 1970-01-01 00:00:04     4     5
##  5 1970-01-01 00:00:05     5     6
##  6 1970-01-01 00:00:06     6     7
##  7 1970-01-01 00:00:07     7     8
##  8 1970-01-01 00:00:08     8     9
##  9 1970-01-01 00:00:09     9    10
## 10 1970-01-01 00:00:10    10    11
```

Notice regardless of which temporal direction is selected, an outer-left join is always performed (i.e., all timestamp values and `u` values of `left` from above will always be present in the output, and the `v` column in the output will contain `NA` whenever there is no record from `right` that meets the matching criteria).

OLS Regression

You might be wondering whether the version of this functionality in Flint is more or less identical to `lm()` in R. Turns out it has much more to offer than `lm()` does. An OLS regression in Flint will compute useful metrics such as Akaike information criterion and Bayesian information criterion, both of which are useful for model selection purposes, and the calculations of both are parallelized by Flint to fully utilize computational power available in a Spark cluster. In addition, Flint supports ignoring regressors that are constant or nearly constant, which becomes useful when an intercept term is included. To see why this is the case, we need to briefly examine the goal of the OLS regression, which is to find some column vector of coefficients $\mathbf{\beta}$ that minimizes $\|\mathbf{y} – \mathbf{X} \mathbf{\beta}\|^2$, where $\mathbf{y}$ is the column vector of response variables, and $\mathbf{X}$ is a matrix consisting of columns of regressors plus an entire column of 1s representing the intercept terms. The solution to this problem is $\mathbf{\beta} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}$, assuming the Gram matrix $\mathbf{X}^\intercal\mathbf{X}$ is non-singular. However, if $\mathbf{X}$ contains a column of all 1s of intercept terms, and another column formed by a regressor that is constant (or nearly so), then columns of $\mathbf{X}$ will be linearly dependent (or nearly so) and $\mathbf{X}^\intercal\mathbf{X}$ will be singular (or nearly so), which presents an issue computation-wise. However, if a regressor is constant, then it essentially plays the same role as the intercept terms do. So simply excluding such a constant regressor in $\mathbf{X}$ solves the problem. Also, speaking of inverting the Gram matrix, readers remembering the concept of "condition number" from numerical analysis must be thinking to themselves how computing $\mathbf{\beta} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}$ could be numerically unstable if $\mathbf{X}^\intercal\mathbf{X}$ has a large condition number. This is why Flint also outputs the condition number of the Gram matrix in the OLS regression result, so that one can sanity-check the underlying quadratic minimization problem being solved is well-conditioned.

So, to summarize, the OLS regression functionality implemented in Flint not only outputs the solution to the problem, but also calculates useful metrics that help data scientists assess the sanity and predictive quality of the resulting model.

To see OLS regression in action with `sparklyr.flint`, one can run the following example:

```
mtcars_sdf <- copy_to(sc, mtcars, overwrite = TRUE) %>%
  dplyr::mutate(time = 0L)
mtcars_ts <- from_sdf(mtcars_sdf, is_sorted = TRUE, time_unit =
"SECONDS")
model <- ols_regression(mtcars_ts, mpg ~ hp + wt) %>% to_sdf()

print(model %>% dplyr::select(akaikeIC, bayesIC, cond))

## # Source: spark [?? x 3]
##   akaikeIC bayesIC    cond
##
## 1     155.    159. 345403.

# ^ output says condition number of the Gram matrix was within reason
```

and obtain $\mathbf{\beta}$, the vector of optimal coefficients, with the following:

```
print(model %>% dplyr::pull(beta))

## [[1]]
## [1] -0.03177295 -3.87783074
```

Additional Summarizers

The EWMA (Exponential Weighted Moving Average), EMA half-life, and the standardized moment summarizers (namely, skewness and kurtosis) along with a few others which were missing in `sparklyr.flint` 0.1 are now fully supported in `sparklyr.flint` 0.2.

Better Integration With `sparklyr`

While `sparklyr.flint` 0.1 included a `collect()` method for exporting data from a Flint time-series RDD to an R data frame, it did not have a similar method for extracting the underlying Spark data frame from a Flint time-series RDD. This was clearly an oversight. In `sparklyr.flint` 0.2, one can call `to_sdf()` on a timeseries RDD to get back a Spark data frame that is usable in `sparklyr` (e.g., as shown by `model %>% to_sdf() %>% dplyr::select(...)` examples from above). One can also get to the underlying Spark data frame JVM object reference by calling `spark_dataframe()` on a Flint time-series RDD (this is usually unnecessary in vast majority of `sparklyr` use cases though).

Conclusion

We have presented a number of new features and improvements introduced in `sparklyr.flint` 0.2 and deep-dived into some of them in this blog post. We hope you are as excited about them as we are.

Thanks for reading!