.. \_batch:

Batch aggregate feature jobs

============================

In the previous section, we went over the core concepts of the aggregation framework and discussed how you can set up you own `AggregateGroups` to compute aggregate features.

Given these groups, this section will discuss how you can setup offline batch jobs to produce the corresponding aggregate features, updated daily. To accomplish this, we need to setup a summingbird-scalding job that is pointed to the input data records containing features and labels to be aggregated.

Input Data

----------

In order to generate aggregate features, the relevant input features need to be available offline as a daily scalding source in `DataRecord` format (typically `DailySuffixFeatureSource <https://cgit.twitter.biz/source/tree/src/scala/com/twitter/ml/api/FeatureSource.scala>`\_, though `HourlySuffixFeatureSource` could also be usable but we have not tested this).

.. admonition:: Note

The input data source should contain the keys, features and labels you want to use in your `AggregateGroups`.

Aggregation Config

------------------

Now that we have a daily data source with input features and labels, we need to setup the `AggregateGroup` config itself. This contains all aggregation groups that you would like to compute and we will go through the implementation step-by-step.

.. admonition:: Example: Timelines Quality config

`TimelinesAggregationConfig <https://cgit.twitter.biz/source/tree/src/scala/com/twitter/timelines/prediction/common/aggregates/TimelinesAggregationConfig.scala>`\_ imports the configured `AggregationGroups` from `TimelinesAggregationConfigDetails <https://cgit.twitter.biz/source/tree/src/scala/com/twitter/timelines/prediction/common/aggregates/TimelinesAggregationConfigDetails.scala>`\_. The config is then referenced by the implementing summingbird-scalding job which we will setup below.

OfflineAggregateSource

----------------------

Each `AggregateGroup` will need to define a (daily) source of input features. We use `OfflineAggregateSource` for this to tell the aggregation framework where the input data set is and the required timestamp feature that the framework uses to decay aggregate feature values:

.. code-block:: scala

val timelinesDailyRecapSource = OfflineAggregateSource(

name = "timelines\_daily\_recap",

timestampFeature = TIMESTAMP,

scaldingHdfsPath = Some("/user/timelines/processed/suggests/recap/data\_records"),

scaldingSuffixType = Some("daily"),

withValidation = true

)

.. admonition:: Note

.. cssclass:: shortlist

#. The name is not important as long as it is unique.

#. `timestampFeature` must be a discrete feature of type `com.twitter.ml.api.Feature[Long]` and represents the “time” of a given training record in milliseconds - for example, the time at which an engagement, push open event, or abuse event took place that you are trying to train on. If you do not already have such a feature in your daily training data, you need to add one.

#. `scaldingSuffixType` can be “hourly” or “daily” depending on the type of source (`HourlySuffixFeatureSource` vs `DailySuffixFeatureSource`).

#. Set `withValidation` to true to validate the presence of \_SUCCESS file. Context: https://jira.twitter.biz/browse/TQ-10618

Output HDFS store

-----------------

The output HDFS store is where the computed aggregate features are stored. This store contains all computed aggregate feature values and is incrementally updated by the aggregates job every day.

.. code-block:: scala

val outputHdfsPath = "/user/timelines/processed/aggregates\_v2"

val timelinesOfflineAggregateSink = new OfflineStoreCommonConfig {

override def apply(startDate: String) = new OfflineAggregateStoreCommonConfig(

outputHdfsPathPrefix = outputHdfsPath,

dummyAppId = "timelines\_aggregates\_v2\_ro", // unused - can be arbitrary

dummyDatasetPrefix = "timelines\_aggregates\_v2\_ro", // unused - can be arbitrary

startDate = startDate

)

}

Note: `dummyAppId` and `dummyDatasetPrefix` are unused so can be set to any arbitrary value. They should be removed on the framework side.

The `outputHdfsPathPrefix` is the only field that matters, and should be set to the HDFS path where you want to store the aggregate features. Make sure you have a lot of quota available at that path.

Setting Up Aggregates Job

-------------------------

Once you have defined a config file with the aggregates you would like to compute, the next step is to create the aggregates scalding job using the config (`example <https://cgit.twitter.biz/source/tree/timelines/data\_processing/ad\_hoc/aggregate\_interactions/v2/offline\_aggregation/TimelinesAggregationScaldingJob.scala>`\_). This is very concise and requires only a few lines of code:

.. code-block:: scala

object TimelinesAggregationScaldingJob extends AggregatesV2ScaldingJob {

override val aggregatesToCompute = TimelinesAggregationConfig.aggregatesToCompute

}

Now that the scalding job is implemented with the aggregation config, we need to setup a capesos config similar to https://cgit.twitter.biz/source/tree/science/scalding/mesos/timelines/prod.yml:

.. code-block:: scala

# Common configuration shared by all aggregates v2 jobs

\_\_aggregates\_v2\_common\_\_: &\_\_aggregates\_v2\_common\_\_

class: HadoopSummingbirdProducer

bundle: offline\_aggregation-deploy.tar.gz

mainjar: offline\_aggregation-deploy.jar

pants\_target: "bundle timelines/data\_processing/ad\_hoc/aggregate\_interactions/v2/offline\_aggregation:bin"

cron\_collision\_policy: CANCEL\_NEW

use\_libjar\_wild\_card: true

.. code-block:: scala

# Specific job computing user aggregates

user\_aggregates\_v2:

<<: \*\_\_aggregates\_v2\_common\_\_

cron\_schedule: "25 \* \* \* \*"

arguments: --batches 1 --output\_stores user\_aggregates --job\_name timelines\_user\_aggregates\_v2

.. admonition:: Important

Each AggregateGroup in your config should have its own associated offline job which specifies `output\_stores` pointing to the output store name you defined in your config.

Running The Job

---------------

When you run the batch job for the first time, you need to add a temporary entry to your capesos yml file that looks like this:

.. code-block:: scala

user\_aggregates\_v2\_initial\_run:

<<: \*\_\_aggregates\_v2\_common\_\_

cron\_schedule: "25 \* \* \* \*"

arguments: --batches 1 --start-time “2017-03-03 00:00:00” --output\_stores user\_aggregates --job\_name timelines\_user\_aggregates\_v2

.. admonition:: Start Time

The additional `--start-time` argument should match the `startDate` in your config for that AggregateGroup, but in the format `yyyy-mm-dd hh:mm:ss`.

To invoke the initial run via capesos, we would do the following (in Timelines case):

.. code-block:: scala

CAPESOSPY\_ENV=prod capesospy-v2 update --build\_locally --start\_cron user\_aggregates\_v2\_initial\_run science/scalding/mesos/timelines/prod.yml

Once it is running smoothly, you can deschedule the initial run job and delete the temporary entry from your production yml config.

.. code-block:: scala

aurora cron deschedule atla/timelines/prod/user\_aggregates\_v2\_initial\_run

Note: deschedule it preemptively to avoid repeatedly overwriting the same initial results

Then schedule the production job from jenkins using something like this:

.. code-block:: scala

CAPESOSPY\_ENV=prod capesospy-v2 update user\_aggregates\_v2 science/scalding/mesos/timelines/prod.yml

All future runs (2nd onwards) will use the permanent entry in the capesos yml config that does not have the `start-time` specified.

.. admonition:: Job name has to match

It's important that the production run should share the same `--job\_name` with the initial\_run so that eagleeye/statebird knows how to keep track of it correctly.

Output Aggregate Features

-------------------------

This scalding job using the example config from the earlier section would output a VersionedKeyValSource to `/user/timelines/processed/aggregates\_v2/user\_aggregates` on HDFS.

Note that `/user/timelines/processed/aggregates\_v2` is the explicitly defined root path while `user\_aggregates` is the output directory of the example `AggregateGroup` defined earlier. The latter can be different for different `AggregateGroups` defined in your config.

The VersionedKeyValSource is difficult to use directly in your jobs/offline trainings, but we provide an adapted source `AggregatesV2FeatureSource` that makes it easy to join and use in your jobs:

.. code-block:: scala

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.conversion.\_

val pipe: DataSetPipe = AggregatesV2FeatureSource(

rootPath = "/user/timelines/processed/aggregates\_v2",

storeName = "user\_aggregates",

aggregates = TimelinesAggregationConfig.aggregatesToCompute,

trimThreshold = 0

)(dateRange).read

Simply replace the `rootPath`, `storeName` and `aggregates` object to whatever you defined. The `trimThreshold` tells the framework to trim all features below a certain cutoff: 0 is a safe default to use to begin with.

.. admonition:: Usage

This can now be used like any other `DataSetPipe` in offline ML jobs. You can write out the features to a `DailySuffixFeatureSource`, you can join them with your data offline for trainings, or you can write them to a Manhattan store for serving online.

Aggregate Features Example

--------------------------

Here is an example of sample of the aggregate features we just computed:

.. code-block:: scala

user\_aggregate\_v2.pair.any\_label.any\_feature.50.days.count: 100.0

user\_aggregate\_v2.pair.any\_label.tweetsource.is\_quote.50.days.count: 30.0

user\_aggregate\_v2.pair.is\_favorited.any\_feature.50.days.count: 10.0

user\_aggregate\_v2.pair.is\_favorited.tweetsource.is\_quote.50.days.count: 6.0

meta.user\_id: 123456789

Aggregate feature names match a `prefix.pair.label.feature.half\_life.metric` schema and correspond to what was defined in the aggregation config for each of these fields.

.. admonition:: Example

In this example, the above features are capturing that userId 123456789L has:

..

A 50-day decayed count of 100 training records with any label or feature (“tweet impressions”)

A 50-day decayed count of 30 records that are “quote tweets” (tweetsource.is\_quote = true)

A 50-day decayed count of 10 records that are favorites on any type of tweet (is\_favorited = true)

A 50-day decayed count of 6 records that are “favorites” on “quote tweets” (both of the above are true)

By combining the above, a model might infer that for this specific user, quote tweets comprise 30% of all impressions, have a favorite rate of 6/30 = 20%, compared to a favorite rate of 10/100 = 10% on the total population of tweets.

Therefore, being a quote tweet makes this specific user `123456789L` approximately twice as likely to favorite the tweet, which is useful for prediction and could result in the ML model giving higher scores to & ranking quote tweets higher in a personalized fashion for this user.

Tests for Feature Names

--------------------------

When you change or add AggregateGroup, feature names might change. And the Feature Store provides a testing mechanism to assert that the feature names change as you expect. See `tests for feature names <https://docbird.twitter.biz/ml\_feature\_store/catalog.html#tests-for-feature-names>`\_.