Implementing Distributed Online Machine Learning Algorithms in Apache Flink

Zhanwang Chen, Oleksandr Chumak, Changbin Lu

Technical University of Berlin

Database Systems and Information Management Group, Big Data Analytics Project

Advisors: Bonaventura Del Monte, Dr. Alireza Rezaei Mahdiraji

Abstract

This report covers the main features of a Big Data Analytics project implemented with Scala. Some already known algorithms are implemented in an online and distributed fashion by the means of some APIs and mediator tools such as Flink's pipelining API and parameter server.

Keywords: online algorithm, distributed algorithm, Flink, machine learning

Task Description

The task, as specified in [1] implied implementation of the following five algorithms:

- Online Support Vector Machine (OSVM);
- Online Ridge Regression (ORR);
- Lasso with Shrinkage via Limit of Gibbs Sampling (SLOG);
- Asynchronous Stochastic Variational Inference (ASVI);
- Growing Gaussian Mixture Model (GGMM).

The implementation involved learning the Scala language, Flink streaming and pipeline API as well as parameter server which will be covered in further sections.

Project Plan

The following graph reflects the working process during the project:

The overall working process can be split up into four main parts. During the first phase Scala programming language, basics of APIs and parameter server were studied. In a second phase all of the algorithms were implemented one by one using the parameter server approach. During the implementation process we found out that not algorithms are of same complexity, thus some algorithms such as ORR were relatively easy to program, while ASYSVI turned out to be more involved and required a longer implementation period which overlapped with the testing phase and also required consultation with the authors of the method. Third phase consisted of evaluation of algorithms. This included successfully running algorithms both locally on a single machine as well as on cluster. After that the performance was compared with other existing implementations. Most of the algorithms were not available at StreamDM (a suggested framework for comparison as discussed in [1]). At the same time there are some other implementations that more or less resemble our experimental environment (i.e. online fashion), thus we decided to use them for a final evaluation. Final phase consisted of writing a report and preparing a project presentation.

Parameter Server

In order to train our algorithms in a distributed fashion, a parameter server abstraction was used [3]. The basic idea of how the parameter server works is depicted in a chart below.

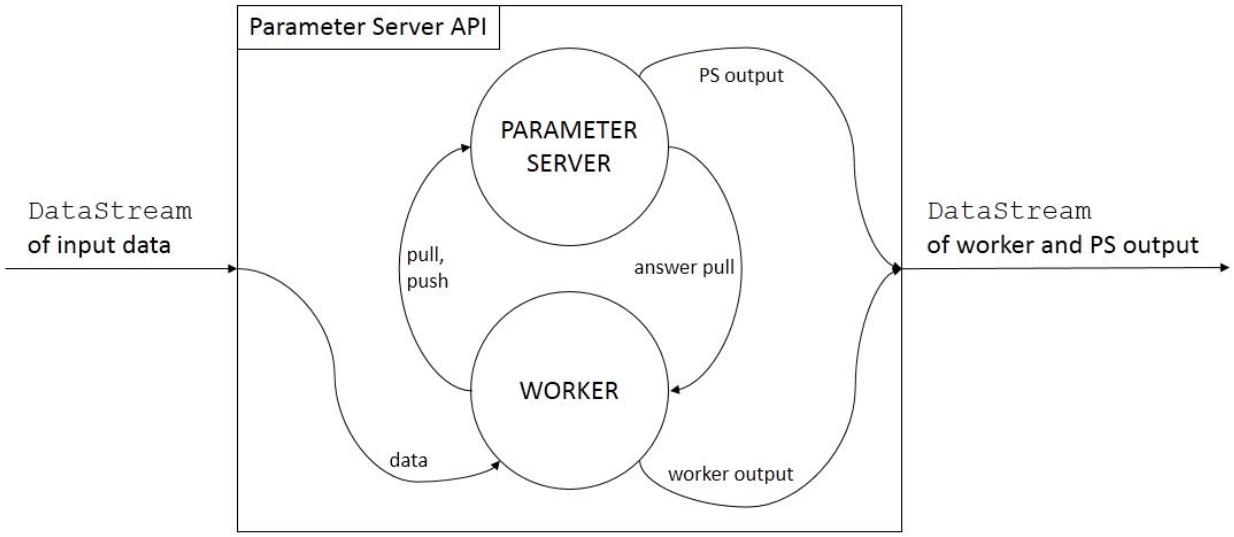


Figure 1. Parameter server workflow. Source [5]

A system has two main types of actors: workers and a parameter server. A worker collects input labelled and unlabelled data points until the specified window size is reached. After that a worker has two options: either start a computation or pull necessary parameters from a server. If latter, than the worker waits for the server's pull answer. After performing calculations a worker sends its model parameters to a server. Server's task is to obtain parameters from workers and merge and rebalance them into one model. This approach is of key importance in this project as it allows for efficient parallelizing of the workflow.

INSERT EXECUTION PLANS FROM FLINK UI

In order for algorithms to operate, certain requirements for the datasets have to be maintained. For example, an input stream element's have to contain a label and a vector of features. These are then processed by Parameter server as *LabeledVectors* with label part and a

DenseVector part. In general, parameter server has the following tuning parameters that influence the performance (how fast we process and train) of the algorithms:

- WorkerParallelism - number of parallel instances of workers. By default is 4;
- PSParallelism - number of parallel instances of server. By default is 4;
- WindowSize - specifies the batches of the points that have to be collected before performing optimization computations. By default is 100 observations;
- IterationWaitTime - time in milliseconds to wait for a new input before finishing the process;
- Label - defines observations that are unlabelled, e.g. if an observation comes with label -1 (by default), it is treated as unlabelled.
- LabeledVectorOutput - if set to false (by default), outputs the model, otherwise outputs the labels along with weights of unlabelled data.

During the experiments, the window size parameter tend to show the greatest impact on the performance (speed).

Online Support Vector Machine

Implementation

A Support Vector Machine (SVM) is defined by a separating hyperplane, an algorithm outputs an optimal hyperplane which categorizes new examples. SVM is a classification algorithm that tries to find a largest possible margin between groups of D-dimensional data points. One online version of SVM is an incremental learning which can be viewed as an online convex optimization problem. The related proof of convergence cam found in [14].

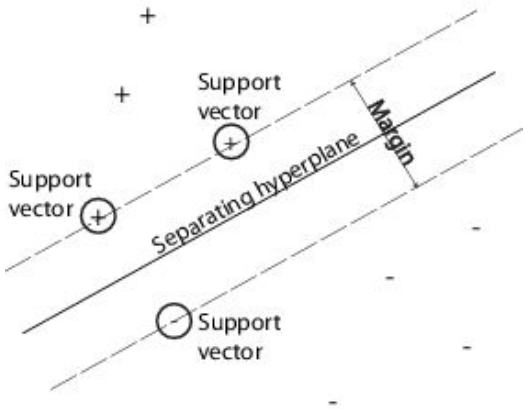


Figure 2. SVM logic

An implemented by us algorithm follows a PS-worker logic. At each iteration worker pulls current values of parameters **w** and **b** (weights). After that it predicts the labels of the unlabelled data. It is worth mentioning that we have queues of the training data and unlabelled data. Once the training queue is full (reached the window size) we perform updates of the local worker's parameters. In our case it's directions for **w** and **b**. Direction on **w** also contains a hyperparameter **c** that influences the strength of changing the direction. We then push these parameters to a server.

Once a server has obtained all the local parameters (directions), it performs their aggregation into global **w** and **b**. All the formulas as well as the algorithm prototype can be found in [3].

Evaluation

We evaluated our implementation both on a local machine and on cluster (distributed mode). In order to simplify the process of our experiments, we decided to use manually generated dataset. This decision allowed us to experiment with a clean big-sized data. The data

(used on cluster) consists of 6 million data points. Each data point is a label ("-1" or "1") and a corresponding feature vector of 7 elements. The scikit-learn's *make_classification* was used for generation. The total size of a dataset is 5 GB in HDFS. A dataset for local testing possesses the same structural properties, but has a size of 1,603,590 data points, among which 481,077 for testing (have label "-9").

For a local testing we compared with scikit-learn's implementation of SVM as we couldn't find an existing online solution for our algorithm. Moreover, described in a project requirements StreamDM [4], does not have this algorithm implemented either. As a metric for classification, F1-score was used. Our implementation showed result 0.91 compared to 0.94 obtained from scikit-learn.

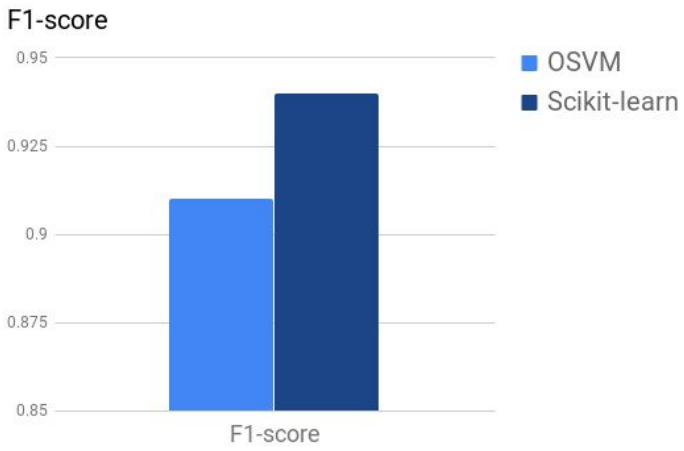


Figure 3. F1-score of a local implementation. Source: own work.

Our implementation though a little underperforms scikit-learn's offline implementation (fig. 3), shows a substantially smaller amount of time required for training and prediction (fig. 4). We also can see that a distributed implementation scales well (unlike Python's solution), fig. 5.

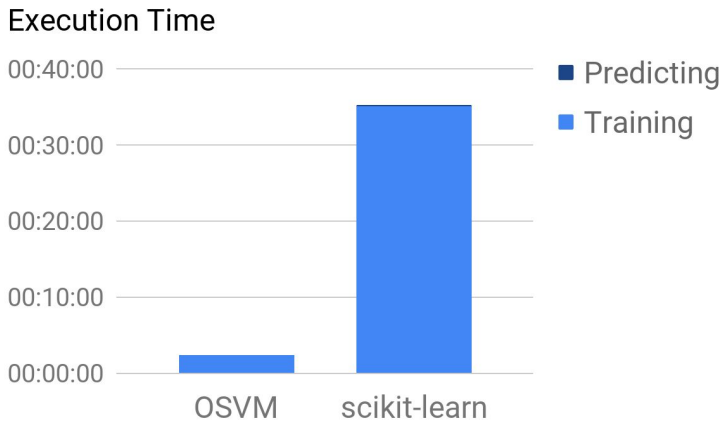


Figure 4. Execution time comparison between our and scikit-learn's implementation. Source: own work

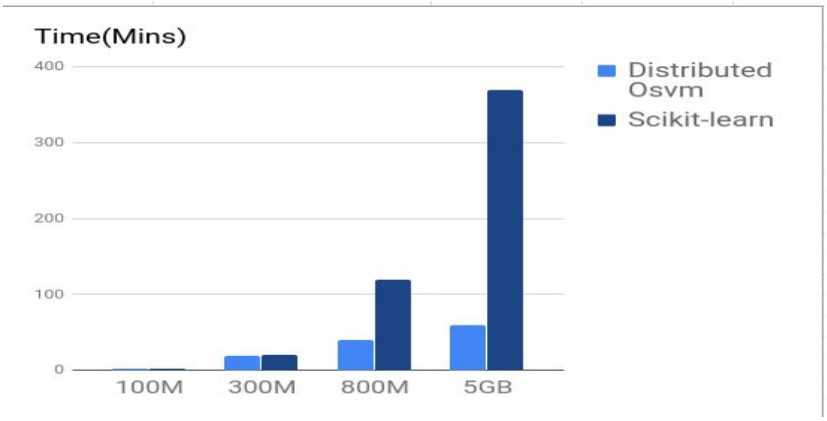


Figure 5. Execution time comparison between distributed implementation of OSVM and scikit-learn for different data sizes. Source: own work

Fig. 6 shows the experiments in the cluster. When the size of window is 500, the larger degree of parallelism of parameter server makes execution time longer. Once the window size increase to 1000, the execution time will not be affected by the degree of parallelism of parameter server.

The below figure shows the degree of parallelism of parameter server increase the execution time significantly when the degree of parallelism of worker is 100.

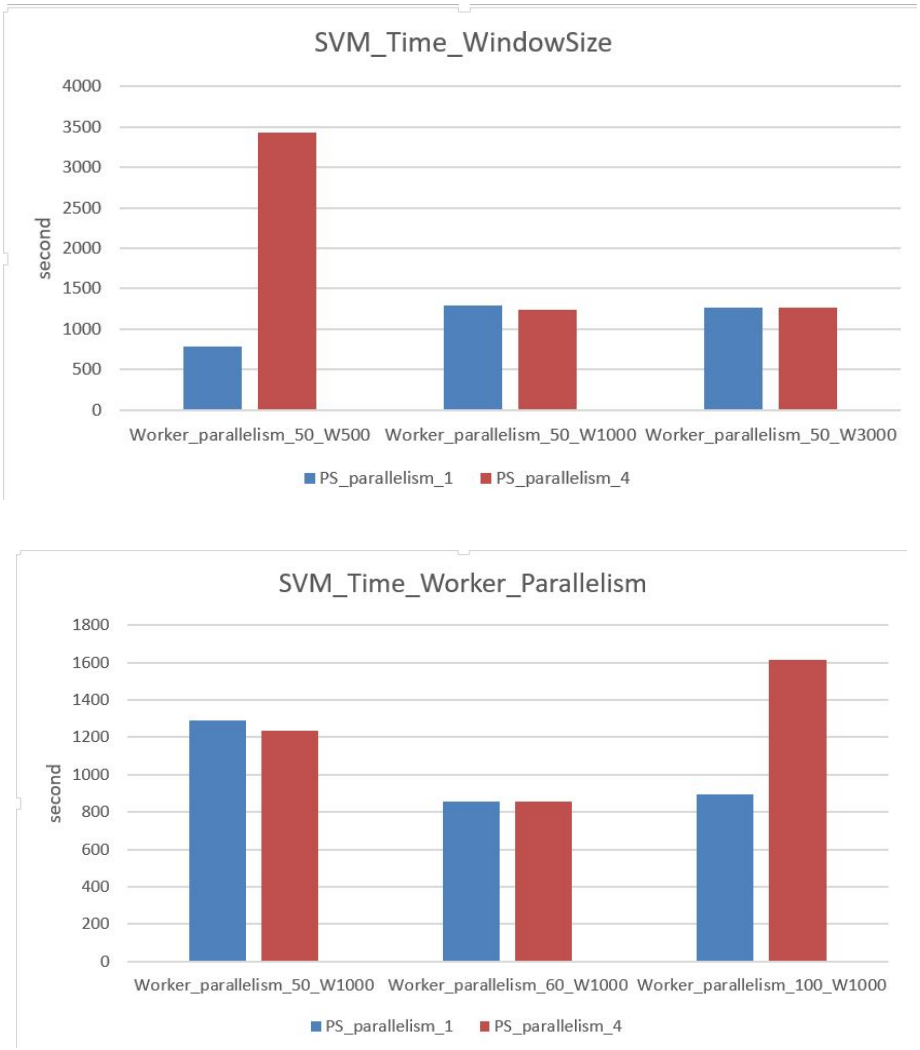


Figure 6. Execution time(second) comparison between different window size(500,1000,3000) and different degree of parallelism of worker(50,60,100).

Fig. 7 points out the relationship between execution time and the number of window size is negative. The higher window size, the lower execution time.

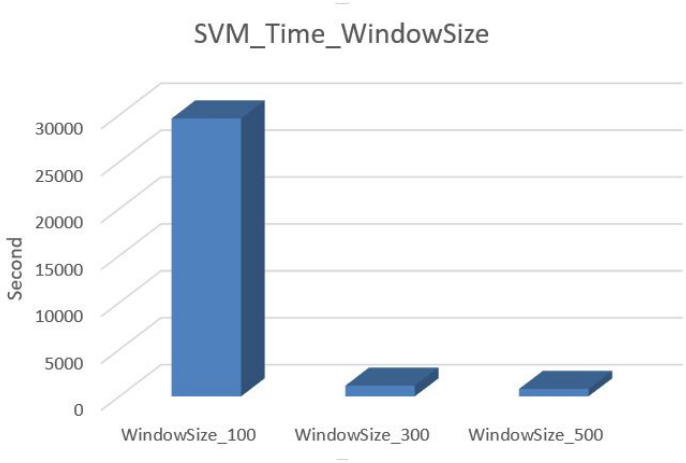
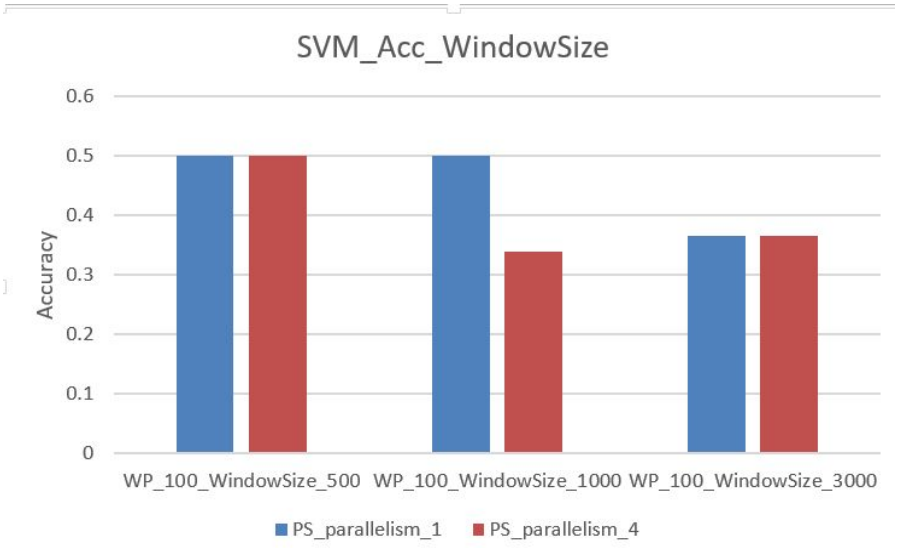


Figure 7. Execution time comparison between different window sizes(100,300,500).



Online Ridge Regression

Implementation

Ridge regression is a modified version of a standard regression. If we build a $\boldsymbol{X`X}$ matrix in form of a correlation matrix, then the usual estimation of $\boldsymbol{\beta}$ will be adequate if the matrix is close

to unit matrix. Instead, ridge regression uses $X`X+kI_p$ for estimation. This allows to avoid difficulties that occur in the above-mentioned situation. [6]

Ridge regression is a standard regression of form $y = X\beta + \lambda \| \beta \|_2$. Here we impose extra regularistation on the coefficients of our regression.

The online and distributed fashion of an algorithm again exploits Flink's data streams and parameter server architecture.

Worker part. When we receive a new labelled data point and if the window size is full, we start our calculations. A worker computes $x_t x_t`$ and $y_t x_t$, sums them up for all the data points in a window and pushes these statistics to a server. The unlabeled data is just put into a queue for later processing. Unlabeled data is the one that has a special value for a label in a data set.

A method that is responsible for labelling the data is *onPullRecv*. Here we calculate our prediction r.

Server part. When a server receives a pull request from worker, it sends the current values of A and b. A is updated as following:

- scale down current A by the amount of the observations that took part in calculating A coming from worker divided by the sum of historical number of observations and current observations.
- add A obtained from worker to the global A by weighted average;

The same logic is applied to b.

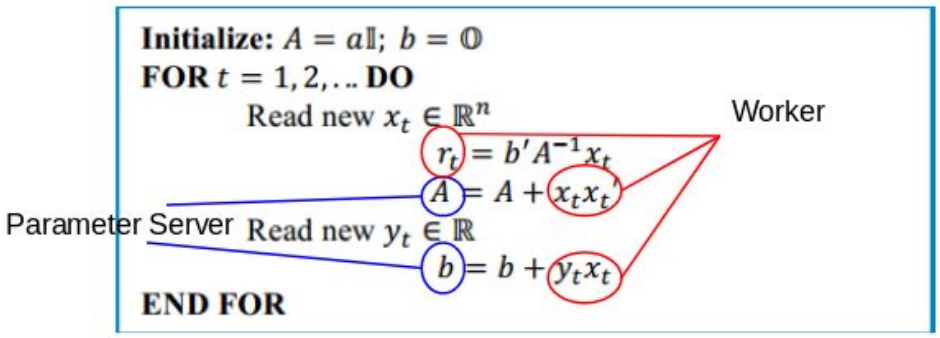


Figure 8. ORR algorithm summary. Source [3]

Evaluation

For a local evaluation a *Turbofan engine degradation simulation data set* was used. [7] It consists of 160,359 data points and 7 features. Features reflect sensor channels, while observations themselves show different scenarios of engine failing. A setting of the engine is set as a dependent variable.

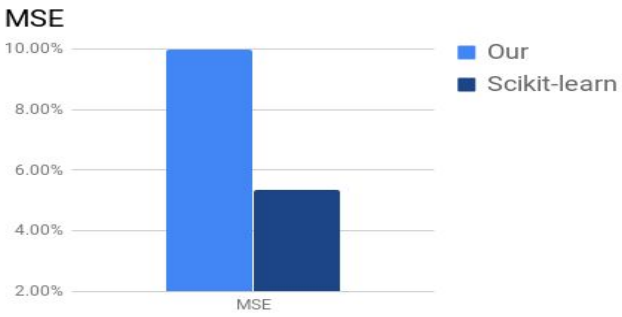


Figure 9. MSE comparison between ORR and scikit-learn's ridge regression. Source: own work

Fig. 9 shows that MSE of our implementation is higher than the corresponding measure showed by Python's solution. This is due different settings of the algorithms. Our algorithm has online nature thus it is normal to have an underperformance in comparison to an offline one.

For cluster setting another data set was used. As with OSVM, we used a scikit-learn's built-in *make_regression* function. The structure of the data points stays the same (i.e. dependent variable and a corresponding feature vector of size 7). The data set contains 1 million observations and has a size of 800Mb.

Fig. 10 shows the increasing number of window size will decrease the mean square error slightly. And the number of parameter servers has no significant effects on mean square error. Fig. 11 shows the number of worker has small positive effects when the number of parameters server is larger than one. Fig. 12 shows window size has no significant effects on processing time but the number of workers and parameter servers both have positive effects.

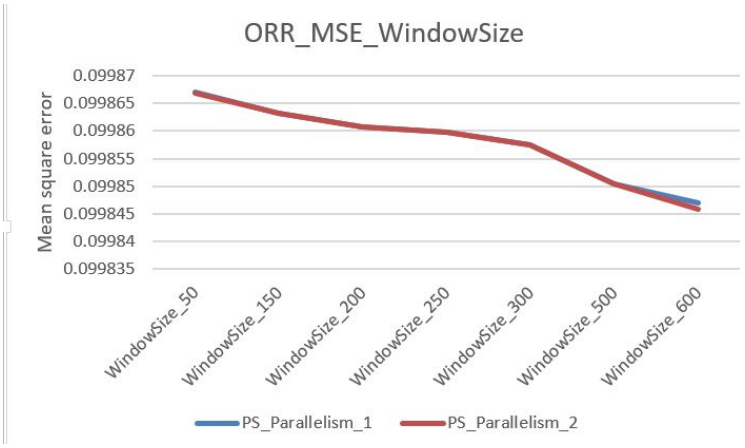


Figure 10. Mean square error comparison between different window sizes(50,150,200,250,300,500,600). Blue line is indicates the degree of parallelism of parameter sever is 1.

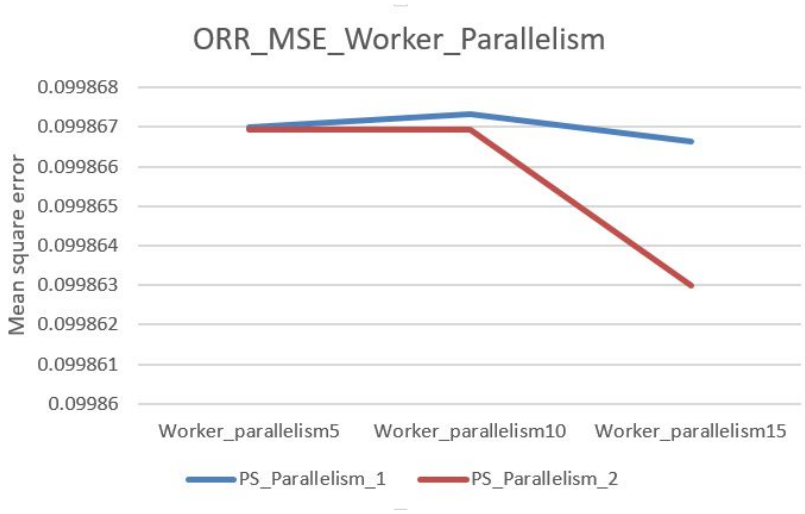


Figure 11. Mean square error comparison between different degree of parallelism of worker(5,10,15). Blue line is indicates the degree of parallelism of parameter sever is 1.

	PS_Parallelism_1	PS_Parallelism_2
Ridge_workerp5_W50	42	41
Ridge_workerp10_W50	36	41
Ridge_workerp15_W50	40	34
	PSpara1	PSpara2
Ridge_workerp5_W150	40	33
Ridge_workerp10_W150	35	41
Ridge_workerp15_W150	40	42
	PSpara1	PSpara2
Ridge_workerp5_W200	43	32
Ridge_workerp10_W200	44	43
Ridge_workerp15_W200	43	36
	PSpara1	PSpara2
Ridge_workerp5_W250	49	34
Ridge_workerp10_W250	40	32
Ridge_workerp15_W250	42	33
	PSpara1	PSpara2
Ridge_workerp5_W300	43	44
Ridge_workerp10_W300	34	36
Ridge_workerp15_W300	45	35
	PSpara1	PSpara2
Ridge_workerp5_W500	42	37
Ridge_workerp10_W500	43	36
Ridge_workerp15_W500	43	34
	PSpara1	PSpara2
Ridge_workerp5_W600	41	41
Ridge_workerp10_W600	36	35
Ridge_workerp15_W600	44	36

Figure 12. Time comparison between different number of window sizes(50,150,200,250,300,500,600), different degree of parallelism of worker(5,10,15) and parameter servers(1,2).

Lasso with Shrinkage via Limit of Gibbs Sampling

Implementation

SLOG [8] is a modified version of LASSO that does shrinkage via limit of Gibbs sampling. [3]. The standard LASSO does least absolute shrinkage. SLOG is a regression problem.

When a worker receives a pull request from parameter server it calculates its own version of parameters for labeled data and predicts the unlabeled data. Worker calculates the vector ***b*** and pushes it to the server. For evaluation, a worker takes the unlabeled vector (datapoint) and multiplies it by the current version of weight vector ***b***, obtained from the parameter server.

An overview of an algorithm is presented in Fig. 8. Here, an $\boldsymbol{x_t x_t'}$ is calculated on a server side. We used the simplified formula to calculate $\boldsymbol{b}$ since the during the processing the matrix sometimes is not invertible.

Initialise B uniformly
FOR t=1,2,…T

$$B^{(t)} = diag\left(\left|b_1^{(t)}\right|, \ldots, \left|b_p^{(t)}\right|\right),$$

$$b^{(t+1)} = [\sum_{t=1}^{T} x_t x_t{}' + \lambda (B^{(t)})^{-1}]^{-1} x_t y_t$$

$$s_t = x_t \hat{b}^{(t+1)}$$

END FOR

Figure 13. SLOG algorithm. Source [3]

Evaluation

The same data sets as for ORR were used for evaluation. Results of testing on a small data locally underperform a little (fig.14) scikit-learn's *Lasso* implementation due to the same learning type reason that we saw in ORR results: MSE is 12.45% against 6.18% by scikit-learn.

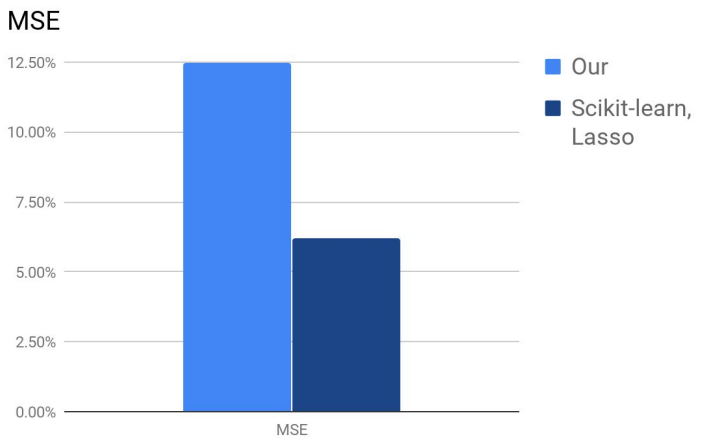


Figure 14. MSE comparison between SLOG and scikit-learn's Lasso regression. Source: own work

Figure. 15 shows the MSE beyond 1 when the number of window size increase. It is caused by the failed inverse matrix which makes weight ***b*** exploding. When we implement algorithm, sometimes matrix was not support inverse or eigen decomposition. And we still maintain doubts about the accumulation of previous data outer production.

Figure. 16, 17 show similar impact of the number of worker and parameter server. With the increasing number of parameter server, the processing time will decrease. Also the number of worker has positive effects on processing time.

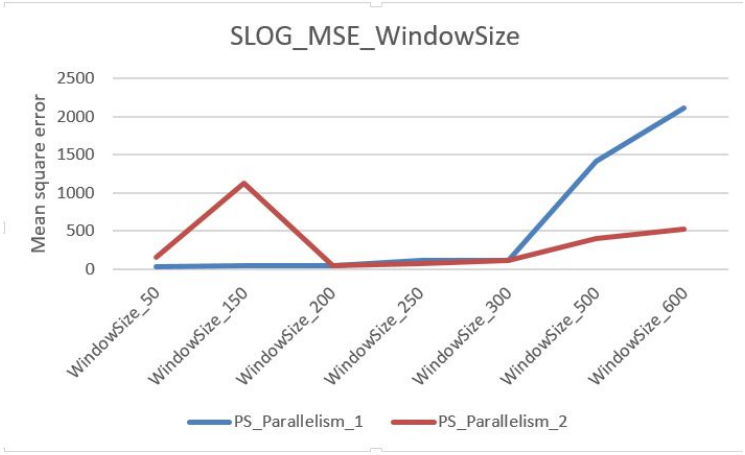


Figure 15. Mean square error comparison between different number of window sizes(50,150,200,250,300,500,600).

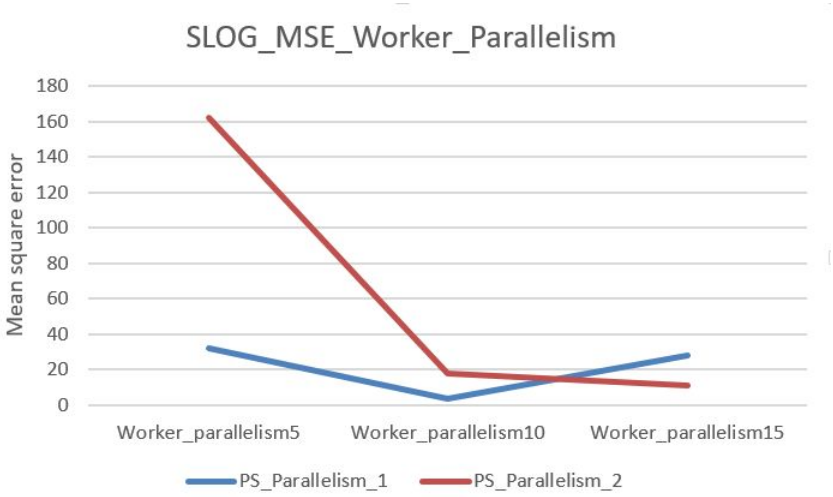


Figure 16. Mean square error comparison between different degree of parallelism of worker(5,10,15). Blue line indicates the degree of parallelism of parameter server (1)

	PS_Parallelism_1	PS_Parallelism_2
slog_workerp5_W50	45	39
slog_workerp10_W50	37	39
slog_workerp15_W50	44	44
	PSpara1	PSpara2
slog_workerp5_W150	44	37
slog_workerp10_W150	45	45
slog_workerp15_W150	40	37
	PSpara1	PSpara2
slog_workerp5_W200	43	43
slog_workerp10_W200	37	33
slog_workerp15_W200	44	36
	PSpara1	PSpara2
slog_workerp5_W250	46	37
slog_workerp10_W250	45	35
slog_workerp15_W250	41	38
	PSpara1	PSpara2
slog_workerp5_W300	43	38
slog_workerp10_W300	43	35
slog_workerp15_W300	42	37
	PSpara1	PSpara2
slog_workerp5_W500	43	34
slog_workerp10_W500	36	35
slog_workerp15_W500	41	41
	PSpara1	PSpara2
slog_workerp5_W600	35	38
slog_workerp10_W600	36	41
slog_workerp15_W600	34	43

Figure 17.Time comparison between different number of window sizes(50,150,200,250,300,500,600), different degree of parallelism of worker(5,10,15) and parameter servers(1,2).

Asynchronous Stochastic Variational Inference

Variational Inference

Variational inference is important when it is difficult to compute a posterior distribution ***p(x|D)***, ***x*** is a variable and ***D*** is data. The main idea of this approach is to approximate the posterior distribution ***p(x)*** by a more tractable distribution ***q(x)*** chosen from a family of simpler distributions.

Stochastic variational inference (SVI) employs stochastic optimization to scale up Bayesian computation to massive data. A lock-free parallel implementation for SVI allows distributed computations over multiple slaves in an asynchronous style.

Latent Dirichlet Allocation (LDA)

LDA represents documents as mixtures of topics that split out words with certain probabilities. The goal is to find topics (distribution of words in topics) and document topics (distribution of topics in documents). The logic is depicted in fig. 18.

Implementation

Server part: A master machine maintains the global variational parameter λ. Once receive send request from worker, it will aggregate predefined amounts of stochastic gradients from workers nonchalantly about the sources of the collected stochastic gradients and updates its global variational parameter.

Worker part: compute the local variational parameter φ and stochastic gradients of ELBO(evidence lower bound) L(λ)

Evaluation

Two data sets were used for evaluation. First is New York Times dataset [15]. It consists of 1.8 million articles and has a vocabulary size of 100,000. This data set was used for testing on cluster. Another, smaller data set for local testing is called 20newsgroup [16] and has around 18,000 news groups' posts on 20 topics. The vocabulary size is 2,000.

We compared our implementation with scikit-learn's online LDA.

```
topic 7:
              image    ---   0.0002
            tethered   ---   0.0000
           zzzmorrow   ---   0.0000
         restraining   ---   0.0000
              absurd   ---   0.0000

topic 8:
              treat    ---   0.0007
              party    ---   0.0005
        zzznotredame   ---   0.0001
                ill    ---   0.0001
           collapse    ---   0.0001

topic 9:
                air    ---   0.0007
             merger    ---   0.0005
         zzzmusgrove   ---   0.0003
           original    ---   0.0001
              chose    ---   0.0001
```

ASVI results on New York Times dataSet

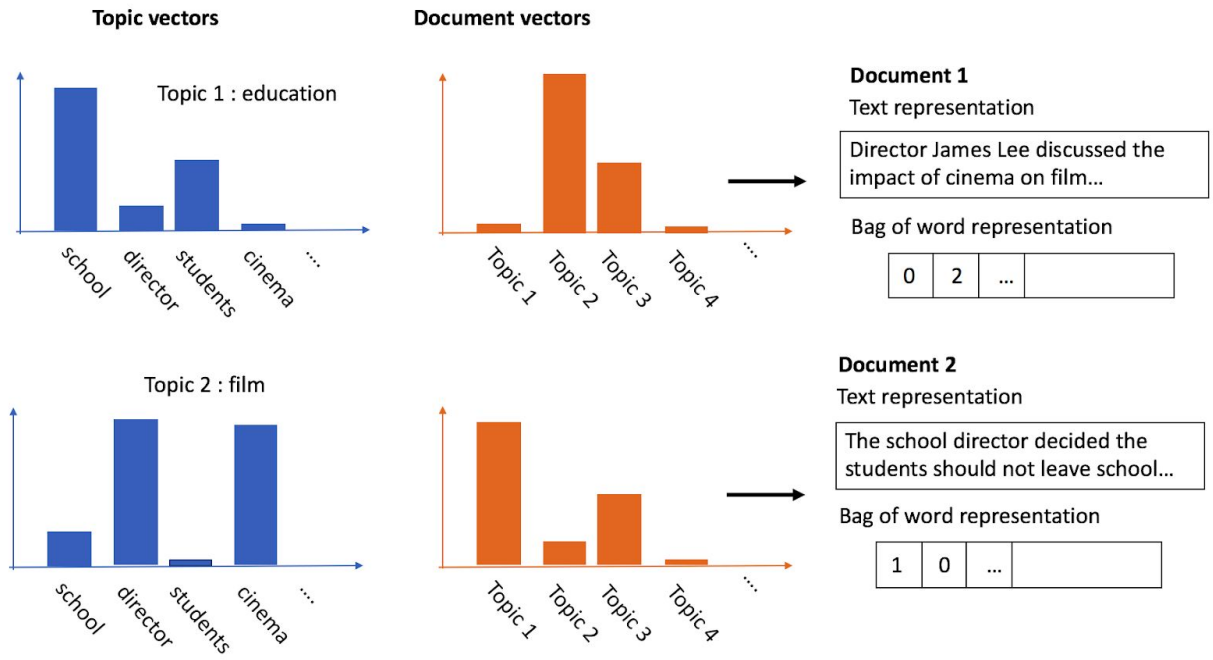


Figure 18. LDA logic. Source:[15,16]

Growing Gaussian Mixture Model

The last algorithm we have implemented is a Growing Gaussian Mixture Model (GGMM) which is an online version of a known GMM algorithm. It is a clustering technique which is backed by the idea of defining clouds of data points as mixtures of Gaussian distributions.

Implementation

As we develop a distributed version of GGMM we follow the idea of parameter server and thus, again, can describe separately worker and server parts and how the communicate in terms

of parameters. A sequence of Gaussians is our parameter. Each Gaussian is characterized by model number, its weight in the whole mixture model, sum of the expected posteriors, center, covariance matrix and a label.

Worker. When a new point arrives we look at it label. If it is unlabeled (has label -1) we put it in a prediction queue. If the queue is large enough, we pull current state of Gaussians from the server. If the point is labeled, basically the same logic as previously applies, only this time the size of the queue is determined by the window size.

On pull receive from server, each worker calculates how the Gaussians need to be updated (in terms of their labels). For each data point from a training queue with determine the most matching Gaussian based on a closeness threshold (hyperparameter). After that we update the labels of Gaussians in the following way: if the highest matching Gaussian is unlabeled, we assign it a label of data point. If the matching Gaussian and data point have different labels, we create a new Gaussian. If the labels match we add the point to the contribution and update Gaussians. [10]

Server. The server is responsible for gathering information about Gaussians and appropriately merge or split them. These operations depend on hyperparameters such as split and merge thresholds.

Evaluation

For local testing a generated dataset was used, which is known as Cassini: a 2 Dimensional Problem [11]. F1-score and confusion matrix were chosen for comparison. Unfortunately, we were not able to find a decent online algorithm for comparison, though there is one solution in

Python [12], it does not actually assign points to clusters but rather draws Gaussians on top of point clouds. We still used that solution to determine proper merge (0.01) and split (1) thresholds.

GGMM results depend on hyperparameters settings as well as window size and the amount of training data. As we can see from figure 19, with the increase of training data our algorithm performs better. *Small dataset* consists of 10,000 points, while the *middle dataset* has 100,000. Both have 3 clusters on a 2D surface. Another fact one can notice is that GGMM performed well with small window sizes.

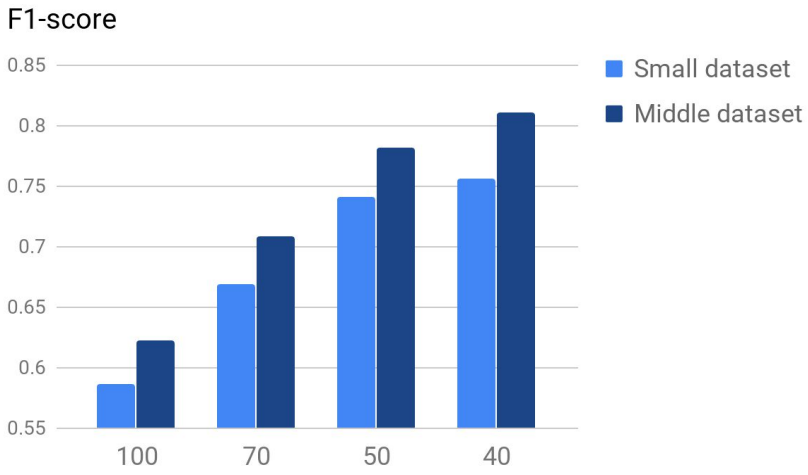


Figure 19. F1-score for different window sizes. Source: own work

We did not experiment much with other hyperparameters, though in general we can say that they do also tend influence the result, but a rather depend on the data distribution. For example, setting merge threshold from 1 to 0.01 improved our results. Also, setting the number of clusters that need to be initialised at the beginning plays not the last role, as it may involve some extra unnecessary merge/split

phases. The best result (window size 40 on middle dataset) we could achieve are depicted in fig. 12 and 13.

We also planned to test an algorithm with big data set (75 million point, 2.5 Gb) but experienced some time shortage.

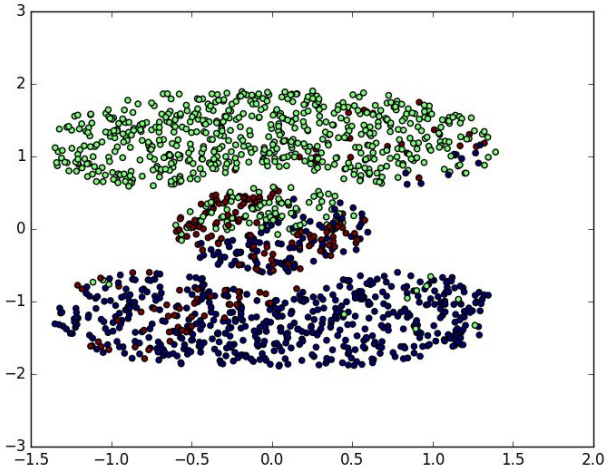


Figure 20. Cluster assignment by GGMM. Source: own work

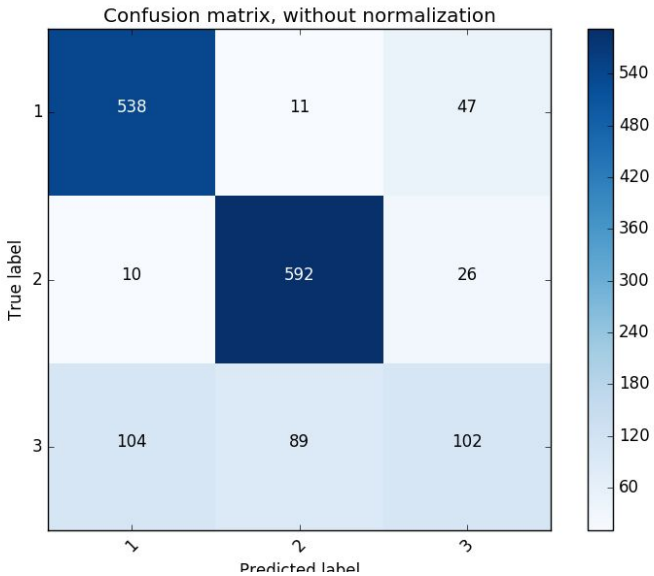


Figure 21. Confusion matrix of GGMM classification. Source: own work

Future work

So far, most of our experiments were compared with offline solutions such as scikit-learn. This can be partly explained by the lack of corresponding online solutions, as the algorithms implemented by us are not trivial. Future investigation in this direction is required. One can also dedicate time for the search of the real-world data for testing. So far, we used real-world data for local testing, but in order to evaluate the performance of a distributed strategy, big data sets are required which usually are not common. In our project for testing on cluster we used data generation functions to obtain files of several gigabytes.

Also, further investigation of the precise impact of the hyperparameters on the accuracy is required. For example, in GGMM literature this influence is described very vaguely and it is not clear which range can certain parameters be defined in.

Summary

We have developed five machine learning algorithms that make use of Flink's *DataStreams* (online learning) and a parameter server architecture (distributed learning). We showed that an algorithm implemented in such manner scales well (in comparison with scikit-learn's algorithms) and can produce competitive results. We have also imposed a pipeline API [13] on top of our algorithms in order to ease their integration into Machine Learning chains of algorithms.

References

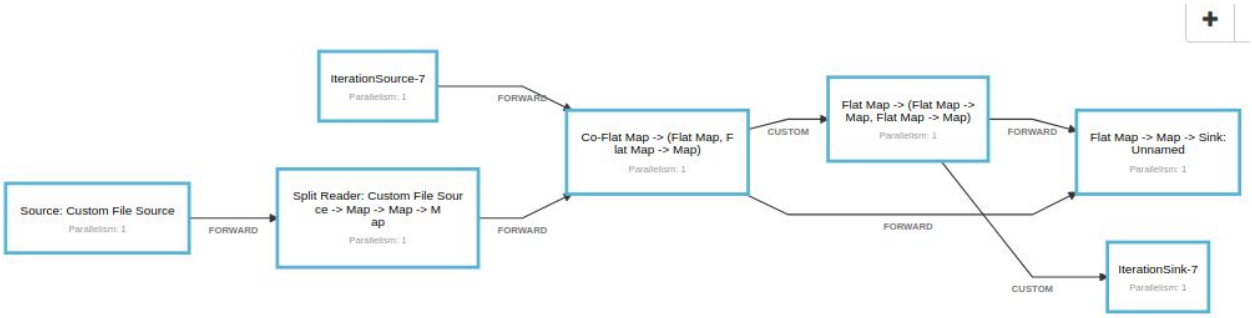
1. Bonaventura Del Monte, “Implementing Distributed Online Machine Learning Algorithms in Apache Flink”, 2017, https://github.com/TU-Berlin-DIMA/BDAPRO.WS1718/issues/18
2. Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., ... & Su, B. Y. (2014, October). Scaling Distributed Machine Learning with the Parameter Server. In *OSDI* (Vol. 14, pp. 583-598).
3. Bouchachia H., et. al. (2017), Novel Scalable Online Machine Learning algorithms for data streams.
4. StreamDM framework, https://github.com/huawei-noah/streamDM
5. Parameter Server implementation, https://github.com/gaborhermann/flink-parameter-server
6. A. Hoerl, and R. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems." Technometrics 55-67, 1970.
7. A. Saxena and K. Goebel (2008). "Turbofan Engine Degradation Simulation Data Set", NASA Ames Prognostics Data Repository, NASA Ames, Moffett Field, CA.
8. B. Rajaratnam, S. Roberts, and D. Sparks, and O. Dalal, "Lasso regression: estimation and shrinkage via the limit of Gibbs sampling." Journal of the Royal Statistical Society: Series B (Statistical Methodology) 153-174, 2016.
9. Zhang, Jiong, et al. "Extreme Stochastic Variational Inference: Distributed and Asynchronous." *arXiv preprint arXiv:1605.09499* (2016).

10. Bouchachia, Abdelhamid, and Charlie Vanaret. "GT2FC: An online growing interval type-2 self-learning fuzzy classifier." *IEEE Transactions on Fuzzy Systems* 22.4 (2014): 999-1018.
11. E. Dimitriadou, A. Weingessel, “Cassini: A 2 Dimensional Problem”, http://ugrad.stat.ubc.ca/R/library/mlbench/html/mlbench.cassini.html
12. Infinite Gaussian Mixture Model, https://github.com/mim/igmm
13. Proteus SOLMA streaming pipeline, https://github.com/proteus-h2020/proteus-solma/tree/master/src/main/scala/eu/proteus/solma/pipeline
14. Zinkevich 2003, Online Convex Programming and Generalized Infinitesimal Gradient Ascent
15. New York Times dataset source: https://catalog.ldc.upenn.edu/ldc2008t19
16. 20Newsgroup: http://qwone.com/~jason/20Newsgroups/

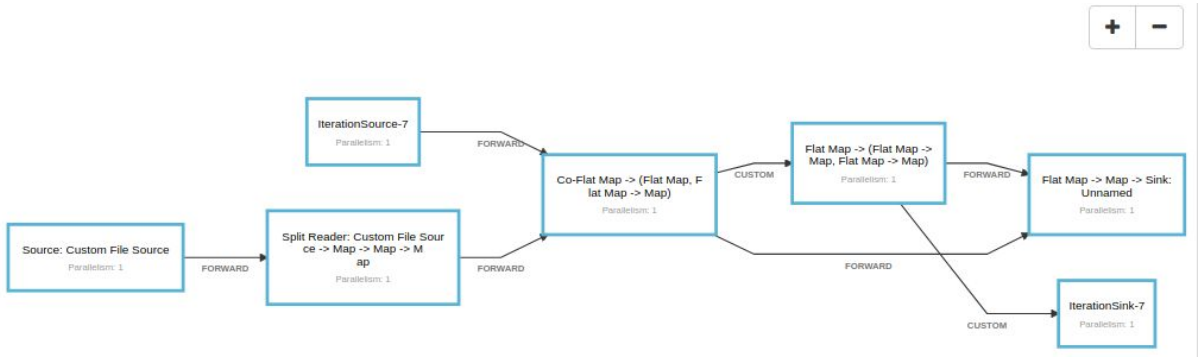
Appendices

Execution Graph

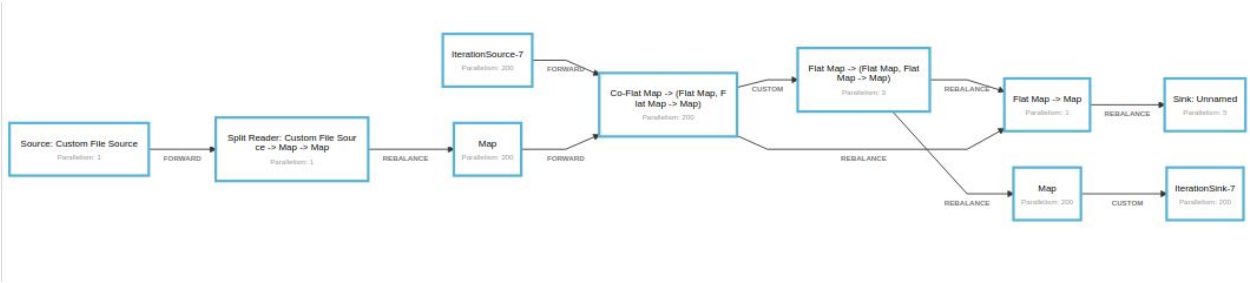
SVM:



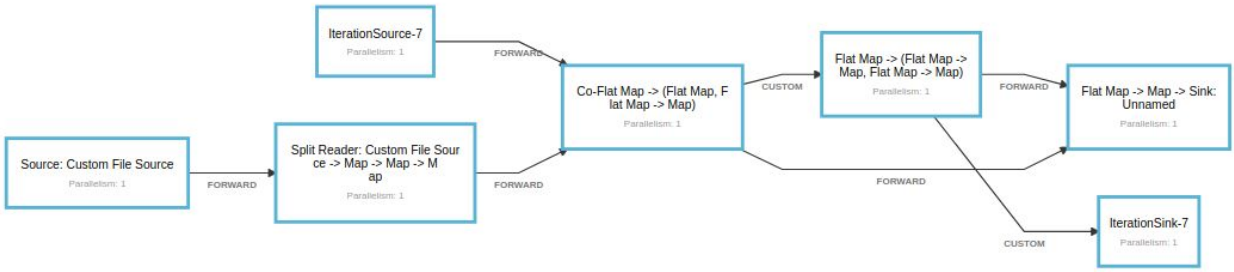
SLOG:



Ridge Regression:



GGMM:



How the run the 5 algorithms:

1. Install Flink 1.3.2(scala 2.11) and

Parameter server

(https://github.com/proteus-h2020/proteus-solma/blob/master/scripts/build-flink-ps.sh)

Important: make sure all the component use the same version, have tested on Flink 1.3.2. (version conflict might show up on flink 1.4.0).

2. • Start Flink on cluster (/share/flink/flink-1.3.2/bin/start-cluster.sh)

• Additional on the cluster: Become hadoop user (sudo -iu hadoop)

3. • /share/flink/flink-1.3.2/bin/flink run -c AlgorithmClass jar-file-with-dependencies [additional arguments]

For example:

./flink-1.3.2/bin/flink run -c eu.proteus.solma.ggmm.GGMM_Local proteus-solma_2.11-0.1.3-jar-with-dependencies.jar

4. • Stop Flink

(Scripts for run all or run multiple experiments and evaluation scripts see the **scripts** folder.)