Distributed and Parallel Databases
https://doi.org/10.1007/s10619-023-07434-0

On combining system and machine learning performance tuning for distributed data stream applications

Lambros Odysseos[1] · **Herodotos Herodotou**[1]

Accepted: 2 May 2023

Abstract

The growing need to identify patterns in data and automate decisions based on them in near-real time, has stimulated the development of new machine learning (ML) applications processing continuous data streams. However, the deployment of ML applications over distributed stream processing engines (DSPEs) such as Apache Spark Streaming is a complex procedure that requires extensive tuning along two dimensions. First, DSPEs have a plethora of system configuration parameters, like degree of parallelism, memory buffer sizes, etc., that have a direct impact on application throughput and/or latency, and need to be optimized. Second, ML models have their own set of hyperparameters that require tuning as they can affect the overall prediction accuracy of the trained model significantly. These two forms of tuning have been studied extensively in the literature but only in isolation from each other. This manuscript presents a comprehensive experimental study that combines system configuration and hyperparameter tuning of ML applications over DSPEs. The experimental results reveal unexpected and complex interactions between the choices of system configurations and hyperparameters, and their impact on both application and model performance. These insights motivate the need for new combined system and ML model tuning approaches, and open up new research directions in the field of self-managing distributed stream processing systems.

Keywords Stream processing · Machine learning · System parameter tuning · Hyper-parameter tuning

✉ Herodotos Herodotou
herodotos.herodotou@cut.ac.cy

Lambros Odysseos
lambros.odysseos@cut.ac.cy

[1] Department of Electrical Engineering and Computer Engineering and Informatics, Cyprus University of Technology, 30 Archiepiskopou Kyprianou, 3036 Limassol, Cyprus

Published online: 17 May 2023

1 Introduction

Distributed stream processing engines (DSPEs) such as Apache Storm, Heron, and Spark Streaming [1] are now widely used for the extraction of insights in near-real time from large scale data streams. At the same time, new streaming machine learning (ML) algorithms for classification, regression, clustering, and concept drift detection have been developed to facilitate more advanced types of processing and data mining, required by modern data-driven applications [2, 3]. Examples of such applications are found in a variety of domains, such as recommending personalized content in social media, identifying credit fraud in banking, recognizing trading signals in financial markets, and more.

The deployment of ML applications over DSPEs is a complex procedure, characterized by two key challenges: (1) how to process the data efficiently and achieve good system performance, and (2) how to process the data effectively and achieve good ML model performance. A DSPE's system performance is affected by a large number of system configuration parameters such as the number of parallel tasks to execute, the amount of memory for each task, the use of compression, etc. Correspondingly, the ML model performance is affected by an array of model parameters, known as hyperparameters, that depend on the type of the ML model. Therefore, extensive tuning is typically required in order to achieve, on the one hand, good system performance in terms of latency and/or throughput, and on the other hand, good ML model performance in terms of prediction accuracy, precision, recall, etc.

The main challenges in effectively tuning the system parameters are the large and complex parameter space, the increased system scale and complexity, as well as the lack of input data statistics from previously executed experiments in a streaming setting [1, 4]. Numerous research studies address this problem by automating the process of finding near-optimal parameter values for executing applications in DSPEs. A wide range of methodologies have been employed in the past, including cost modeling, simulation-based, experiment-driven, and machine learning-based approaches [5–7]. More recently, some adaptive tuning approaches have also been proposed for changing system parameters at runtime based on real-time observations [8].

Automated hyperparameter tuning (or optimization) is another large area of research, primarily of the ML community, that strives to set the hyperparameters of the underlying ML model to values that optimize the model's performance. The grid search method is the most basic approach that can exhaustively explore the space of hyperparameter options, but it is unavoidably expensive [9]. A simple alternative is random search, which samples settings from the hyperparameter space randomly, until a certain budget for the search is exhausted [10]. More advanced methodologies have also been proposed, such as Bayesian optimization, surrogate-based collaborative tuning, and population-based methods (e.g., genetic algorithms, particle swarm optimization) [11, 12].

These two aforementioned forms of tuning have only been studied in isolation from each other, without considering potential interactions among them. To the

best of our knowledge, this work performs the first experimental study that blends system parameter and hyperparameter tuning to identify their combined impact on both system and ML model performance, with some surprising results. Specifically, our experimental results revealed that the choice of system configuration parameters can actually impact the prediction accuracy of the ML model, even when the same hyperparameters and data are used for training. Similarly, some hyperparameter settings can influence the application latency or throughput. Various tradeoffs were also identified, where system parameters that improve system performance can reduce model performance and vice versa. Therefore, the typical approach of first tuning the system parameters and then tuning the hyperparameters (or vice versa) can lead to sub-optimal (or even bad) system and/or model performance. Overall, these findings motivate the need for the development of new automated tuning approaches that combine system parameter and hyperparameter tuning, opening up new research directions.

The main contributions of this work include:

- An experimental analysis of the impact of system parameters to the prediction accuracy of ML models (in addition to DSPE system performance);
- An experimental analysis of the impact of hyperparameters to DSPE system performance (in addition to ML model performance);
- New insights into the combined effect of system parameter and hyperparameter tuning;
- A multi-objective optimization problem formulation for the combined tuning problem.

Some preliminary findings from this study were presented in [13]. Compared to the previous paper, this manuscript presents experimental results from a second ML model, more detailed results from two different applications and three deployment modes, an in-depth discussion of the combined impact of system and model parameters, as well as a new multi-objective optimization problem formulation for the combined tuning problem.

The remaining paper is organized as follows. Section 2 discusses the related work for both tuning problems. Section 3 describes the experimental methodology while Sect. 4 presents the key results. Section 5 further discusses our findings and proposes a new formulation for the combined tuning problem. Section 6 concludes this paper.

2 Related work

2.1 System parameter tuning

Effectively tuning system parameters is a complex procedure that highly impacts the performance of the underlying DSPE. There is a lot of work done on how to efficiently tune system parameters to their (near) optimal values, which can be organized into five categories [1, 14]. The first category is *cost modeling*, where

performance prediction models are utilized through analytical cost functions [5, 15]. This methodology, however, requires deep understanding of the system internals in order to be applied, since the cost functions need to be developed and evaluated. *Simulation-based* approaches perform either a modular or a complete system simulation with combinations of different system parameters in order to explore the search space [16, 17]. This method does not tune the system parameters directly, but is rather used to explore the system performance under various system parameter configurations.

Experiment-driven methods are expensive approaches that repeatedly execute workloads or jobs with varying system parameters in order to collect output statistics through logs, which are then assessed to determine the best parameter values [6, 18]. One main challenge of this approach is how to accurately predict the performance of the DSPE when different applications and/or data as used. A more recent trend is to employ a *machine learning* approach, where information from workload execution logs is gathered in order to train machine learning models to predict relevant system performance under various parameter configuration settings [7, 19]. This method does not require any knowledge of the system internals since the underlying DSPE is treated as a black box. The most ambitious category is *adaptive tuning*, which aims at dynamically changing the system parameter values while the system is still running, assessing the performance improvement (if any), and adapting the new system parameter values [8, 20]. This method is ideal for long-running applications, which may require multiple system parameter tweaks depending on the workload, rather than tuning them once prior to execution.

2.2 Hyperparameter tuning

Respectively, hyperparameter (HP) tuning plays an important role to applications of machine learning models, when prediction accuracy needs to be as high as possible. A decent amount of research has been done in this area and a variety of methods are widely available. The popular grid search method is unavoidably expensive, but it is able to completely and exhaustively explore the HP space as long as there are enough resources to explore the HP space and other computational restrictions or limitations are not applicable [9]. Random search is another common method, which randomly samples hyperparameter values between each applicable range, but it can suffer from unexplored parameters, causing it to miss the optimal values. Past studies have shown that the Random search method can be as effective as other more sophisticated methods and even faster than them [10]. This is typically applicable when the number of hyperparameters to be tuned is limited in the HP space, such as in the case of Support Vector Machines (SVMs).

Surrogate-based collaborative tuning (SCoT) seems to be outperforming the random search method in finding the best hyperparameters but requires an expert user in order to be applied [11]. Another expensive, yet effective framework is the Sequential model-based optimization (SMBO), which requires an increased number of trials for a given dataset to complete. SMBO also tries to optimize a rather expensive target evaluation function [21]. One of the most promising and recent

frameworks is AutoML, which enables effective model selection and hyperparameter tuning to inexperienced ML users by utilizing a variety of meta-learning and greedy algorithms to optimize performance towards a given optimization budget, performance metric, and configuration space [22–24]. One example application of the AutoML framework is Auto-sklearn [12, 22], which was developed on top of the scikit-learn [25] library, a Python module that enables the use of ML models and methods. Another AutoML framework is Auto-WEKA [23, 24, 26], which is based on WEKA, a collection of ML algorithms for data mining tasks implemented in Java [27, 28].

In the streaming setting, MOA (Massive Online Analysis) is an ML software that enables inexperienced users to run (single-threaded) machine learning tasks in live data streams [2]. MOA does not support any automated hyperparameter tuning tasks at the moment. Despite its effectiveness, AutoML suffers from expensive function evaluations, the high dimensional hyperparameter space, and lack of generalization as it tunes the hyperparameters of the model according to specific training datasets. Surrogate models like Bayesian optimization and Tree Parzen Estimator show good performance in hyperparameter tuning tasks but require special treatment in integer and categorical hyperparameters. Other methods include multi-objective optimization and genetic algorithms, which can be costly [29, 30]. There are also hyperparameter tuning approaches that concern live data streaming environments where the tuning happens online, as the data flows are being processed [31, 32].

3 Methodology

This section starts by discussing the selection of the DSPE, along with its various deployment modes and system configuration options. Two distinct applications and datasets are used from two different application domains in order to assess the impact of application type. Two different ML algorithms and, consequently, different sets of hyperparameter settings, are also employed to study the impact of the ML model type. Finally, we present the experimental setup and the key performance metrics used to evaluate the results.

3.1 DSPE selection

Regarding stream processing, there are two types of DSPEs. Some DSPEs, such as Apache Storm and Heron, employ a record-based model that processes input data one record at a time. Other DSPEs, on the other hand, such as Apache Spark Streaming and Trident, employ a batch-based model that processes input data in micro-batches, whose size is configurable by the user [33]. In this study, we focus on micro-batching systems and more specifically on **Spark Streaming**, a well-matured product that enables high processing throughput of data streams in production environments. The processing happens in-memory, which enables small batches of live-streamed data to be processed efficiently while using the Resilient Distributed Datasets (RDDs) abstraction [34]. Spark Streaming has a large system

configuration space with over 200 parameter settings, many of which can have a drastic impact on system performance [1]. These parameters mainly affect some aspects of execution and allocation of computing resources, such as the number of cores and memory to use for task execution, the number of records processed per data partition, the use of compression, and more. Table 1 lists a small subset of the system parameters that were involved in the tuning process of this study and their respective space of values. The parameters and values were selected from a super set of experiments we conducted and match the list of parameters typically used in past parameter tuning studies [1]. In addition, most of the selected parameters are core system parameters that have not changed throughout the newest versions of Spark. Therefore, we expect the findings of our study to be applicable to multiple versions of Spark.

All other DSPEs have very similar configuration parameters that affect the same corresponding aspects of execution, including task parallelism and memory allocation. Past benchmarking studies [35, 36] offer interesting experimental comparisons between the different DSPEs. For example, even though Spark Streaming can scale up very well due to its partitioning feature of the RDDs, it can experience higher latency times than other DSPEs such as Apache Storm and Flink because of additional coordination and scheduling overheads introduced across different RDDs [35]. On the contrary, the latency variation seems to be more robust in Spark Streaming, meaning that it will not fluctuate as much as compared to the other DSPEs. Nonetheless, the impact of configuration parameters is fairly consistent across the DSPEs; for instance, increasing parallelism will increase throughput in a parallelizable application. Overall, *we anticipate our results with Spark Streaming to hold for the other DSPEs as well*, albeit with perhaps different levels of magnitude. We leave this verification for future work.

3.2 Spark streaming deployment modes

When an application is submitted to Spark for execution, a process called the *Driver* is responsible for handling the application lifecycle. Specifically, the Driver is responsible for scheduling jobs, which consist of tasks to be executed in parallel on the system. Spark Streaming is an extension of Spark, which receives live data streams and divides them into micro batches. Each batch is then processed by the Spark Engine as a set of jobs to generate a live stream of results in batches. Spark (and Spark Streaming) can be deployed in three different modes that have the same working principle as described above but with different resource allocation schemes. The three modes are:

1. **Local mode:** Runs the application locally in a single process with a configured number of worker threads to handle task execution;
2. **Pseudo-distributed mode:** Runs the application and the main Spark components (i.e., Master and Worker) on the same machine, while the tasks are executed in parallel in a separate Executor process;

Table 1 Selected set of system parameters for Spark Streaming and the parameter space used in this study

System parameter	Description	Space
spark.master	The cluster manager to connect to: "local" indicates to run Spark locally and "spark" to connect to the given Spark standalone cluster manager	[local, spark]
spark.submit.deployMode	The deploy mode of Spark driver program: "client" indicates to launch the driver locally and "cluster" on one of the nodes inside the cluster	[client, cluster]
spark.driver.cores	Number of cores to use for the driver process	4
spark.driver.memory	Amount of memory to use for the driver process	[4 g, 8 g, 12 g, 16 g]
spark.executor.instances	The number of executors to lunch per cluster node	1
spark.executor.cores	The number of cores to use on each executor for running tasks	[4, 8, 12]
spark.executor.memory	Total amount of memory to use per executor process	[8 g, 12 g, 16 g]
spark.shuffle.compress	Whether to compress map output files, offering a tradeoff between network transfer and CPU processing (for compression)	false
spark.streaming.receiver.maxRate	Maximum rate (number of records per second) at which each input receiver will receive data	[400, 800, 1200, 1600]
spark.streaming.blockInterval	Interval at which data received by each input receiver is partitioned into blocks of data before storing them in Spark	[125ms, 250ms, 500ms]
batchDuration	The time interval at which streaming data will be divided into batches (specified when creating the Spark Streaming Context)	1000

3. **Cluster mode:** The application is submitted for execution on the Cluster Manager, which is responsible for allocating cluster resources to the application, while the tasks are executed in parallel in Executor processes managed by the Workers;

In Cluster mode,[1] Spark supports several Cluster Managers, including the Spark Standalone Master, Apache Mesos, and Apache YARN. The main difference between them is the way cluster resources are allocated to each application, while the execution process is exactly the same. For our experiments, we used Spark Standalone for our cluster-based experiments, as it is widely used in practice [37]. The reason we introduced three different deployment modes in our experiments is that we were not only interested in studying the interactions between system parameters and hyperparameters, but we also wanted to study how different deployment modes might affect the system and ML model performance.

3.3 Applications and data

For the experiments, we used two applications along with their datasets from two very different domains. The first application comes from the **social media** domain and employs machine learning techniques to detect aggressive behavior in tweets [38]. In particular, this application processes a real Twitter dataset of size 330 MB provided by [39], containing 86k tweets classified as either normal (54k), abusive (27k), or hateful (5k). To avoid working with an imbalanced classification problem, we group abusive and hateful tweets together, and label them as aggressive, thereby creating a fairly balanced (63%/37%) binary classification problem. The application parses the incoming tweets and their metadata, and performs preprocessing, extraction of 16 features, normalization, and training and prediction with prequential evaluation [38].

To ensure that our results are application and data agnostic (i.e., generalizable), we employed a second application and dataset from the **banking** sector. The banking application determines whether a loan should be approved by the bank given various characteristics of the bank customer, such as age, salary, education level, etc. For this purpose, we generated synthetic data using the Agrawal stream generator, provided by the MOA framework [40]. The data generator takes a loan function as input (1–10) along with other parameters to generate banking data for bank customers. The loan function works as a heuristic function to determine how the target class is calculated. Specifically, we have used loan function 6 with perturbation 0 to generate 100k loan application records with size 40 MB, forming again a balanced (60%/40%) binary classification problem [40]. The banking application parses the incoming records directly into 9 features and then performs training and prediction with prequential evaluation.

[1] "Cluster mode" in this study refers to running Spark on a cluster of machines as opposed to only one machine. It does not refer to the driver's "cluster" deploy mode, which indicates that the driver is launched inside the cluster (recall Table 1). In our experiments, we run the driver using both "cluster" and "client" deploy modes.

Table 2 Hoeffding Tree (HT) hyperparameters and their space of values/options

Hyperparameter	Space
Split confidence (c)	[0.001, 0.01, 0.1]
Tie threshold (t)	[0.001, 0.05, 0.1]
Grace period (g)	[50, 800, 1500]
Split criterion (s)	[InfoGain, Gini]
Max tree depth (h)	[20, 30]

Table 3 Stochastic gradient descent (SGD) hyperparameters and their space of values/options

Hyperparameter	Space
Lambda (l)	[0.001, 0.01, 0.05, 0.1]
Loss function (o)	[Logistic, Squared, Hinge, Perceptron]
Regularizer (r)	[ZeroRegularizer, L2Regularizer]
Regularization parameter (p)	[0.001, 0.01]

3.4 Machine learning models

Several streaming ML algorithms are currently available for addressing classification problems. We have selected two well-performing and popular classifiers that are widely used in steaming environments, each having its own hyperparameter space [41]. The first one is the **Hoeffding Tree (HT)**, a decision tree specifically designed for large data streams [42]. During processing, a tree node in a HT is split into two nodes once there is sufficient statistical evidence that an optimal splitting feature exists, based on the distribution-independent Hoeffding bound. The second learner is the **Stochastic Gradient Descent (SGD)** optimizer used for learning various linear models, including logistic regression, linear regression, binary class SVM, and perceptron classifier [41]. To perform hyperparameter tuning for both models, we used the popular grid search method, thus capturing the full space of choices in each case. Tables 2 and 3 list the hyperparameters tuned for HT and SGD, respectively, along with the space of values/options considered for each hyperparameter during the grid search, which captures a wide spread of each parameter domain.

3.5 Streaming ML application processing

Figure 1 shows how a streaming ML application is executed on Apache Spark Streaming. The input stream data source can be a file, a database table, a sensor emitting data, a web API endpoint (e.g., Twitter Firehose API), or some other upstream system (e.g., Kafka, Flume). Spark Streaming receives live input data from the input stream through a set of input receivers, which generate input data partitions. Each input receiver streams some maximum number of records per second and creates partitions according to the settings of the `spark.streaming.receiver.maxRate` and `spark.streaming.blockInterval` parameters

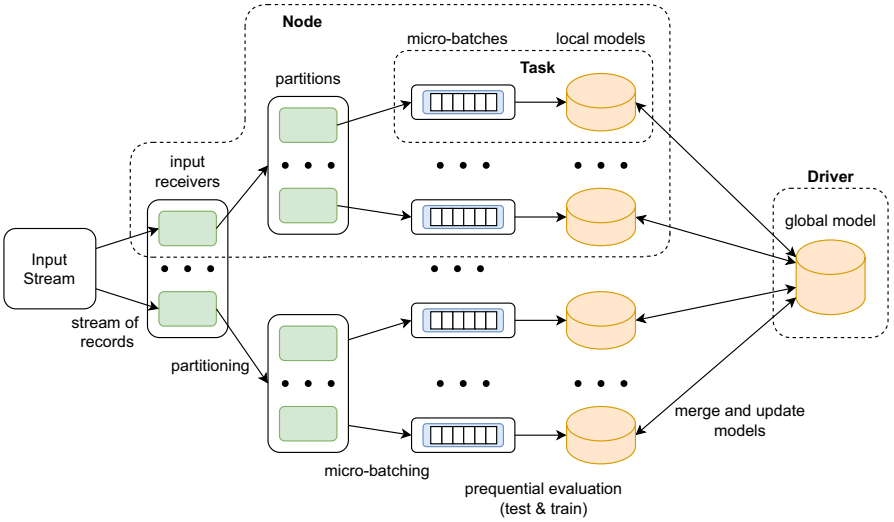


Fig. 1 Execution flow diagram for a streaming ML application on Apache Spark Streaming

(see Table 1). Spark then processes a fixed number of partitions per batch, where each partition is processed by one task. The total number of tasks executed during a batch is upper-bounded by the total number of cores available, which equals the number of Executor processes (`spark.executor.instances`) times the number of cores per executor (`spark.executor.cores`). Hence, the batch size equals the total number of records processed by all tasks during a batch.

In a distributed streaming environment, an ML model is trained and updated in real time in parallel. In particular, each task starts with its own local copy of the global ML model. Hence, the total number of local models equals the number of tasks, which in turn equals the number of partitions processed within a batch. The task's input records are then used to train the local ML model independently from the other tasks. At the end of each batch, the local models are collected by the Spark Driver, which is responsible for merging the local models together in order to derive one global ML model. In a production environment, input records that contain a label will be used to train and update the local models, while input records without a label will probe the model for a prediction. For our experimental evaluation, we used the popular prequential evaluation method, based on which the input records are first tested against the predictions of the model and then used to train the model [2].

The key difference between application processing in a batch-based DSPE like Spark Streaming and a record-based DSPE (e.g., Apache Storm, Flink) is the grouping of records into micro-batches. In record-based DSPEs, the input records flow directly from the input receivers into the long-running tasks for processing. However, when a task implements an ML application, the local models still need to periodically merge to create an updated global model, otherwise the individual task models can diverge. Thus, the frequency of merges becomes analogous to the batch

interval in a batch-based DSPE. This is another key reason for *expecting the findings of this study to be applicable in record-based DSPEs*.

3.6 Experimental setup

For carrying out the experiments, we have used three different environments, one for each deployment mode, all running Apache Spark Streaming v2.3.2. The applications were implemented using streamDM v0.2, an ML library for mining big data streams using Spark Streaming [41].

Local and Pseudo-distributed Mode: For these modes, we used a server with a 24-core 3.2GHz CPU, 128GB of RAM, and 1.7TB of disk storage space. The main system parameters we varied were (i) the number of cores available for the worker threads (in local mode) or the Executor process (in pseudo-distributed mode), ranging from 4-12 cores; (ii) the amount of allocated memory available for task execution, ranging from 8GB-16GB; (iii) the batch size, ranging from 2k-5k records. In the pseudo-distributed mode, the Spark Driver was allocated 4 cores and 4GB of memory. The input records were streamed through a file reader, which read file data in batches (based on the batch size). The files were physically located on the same machine as the Spark Driver process.

Cluster Mode: In this mode, we used a cluster consisting of 5 machines, 1 running the Spark Master and the Driver process (when in "client" deploy mode), and 4 running the Spark Workers. The master machine has an 8-core 3.2GHz CPU, 64GB of RAM and 1.7TB of disk storage space, while the workers have an 8-core 2.4GHz CPU, 24GB of RAM and 3 SAS hard disk drives with 500GB of storage each. The Driver is allocated 4 logical cores and 16GB of memory. Each Worker node is set up to use one Executor process, each using up to 12 logical cores for task execution. In contrast with the other modes, the input data was streamed with input receivers from files stored on a Hadoop Distributed File System (HDFS), running on the same 5 machines. To generate the distributed data stream, we ran one input receiver on each worker, generating input data partitions to the micro batches. Each input receiver streams a maximum number of records per partition (r) and Spark processes a fixed number of partitions per second (p) as explained in Sect. 3.5. This strategy enabled us to control the data ingestion rate in order to avoid large latency fluctuations between batches and achieve sustainable throughput [35]. In addition, by varying these two values (records per partition and partitions per second), we vary the degree of parallelism in the cluster and the batch size. We also varied the amount of allocated memory available for task execution and to the Driver, ranging from 8GB-16GB, but memory size did not seem to have a significant impact on performance. For ease of presentation, Table 4 lists 9 distinct configurations we used in our experiment, which can be divided into three logical groups:

- In group $g1$, we fix partitions per second and vary records per partition ($s1$, $s2$, $s3$, $s4$)
- In group $g2$, we fix records per partition and vary partitions per second ($s5$, $s2$, $s6$, $s7$)

Table 4 Cluster mode system configurations with selected parameters

Group ID	Setup ID	Records per Partition (r)	Partitions per Second (p)	Batch Size
$g1$	$s1$	100	16	1600
	$s2$	200	16	3200
	$s3$	300	16	4800
	$s4$	400	16	6400
$g2$	$s5$	200	8	1600
	$s2$	200	16	3200
	$s6$	200	24	4800
	$s7$	200	32	6400
$g3$	$s8$	100	48	4800
	$s6$	200	24	4800
	$s3$	300	16	4800
	$s9$	400	12	4800

- In group $g3$, we vary both records per partition and partitions per second but fix the total records per second, i.e., batch size to 4800 ($s8$, $s6$, $s3$, $s9$)

Overall, for each deployment mode, for each application and dataset, and for each ML model, we executed all possible combinations of system configuration settings and hyperparameter settings.

3.7 Performance metrics

For evaluating the results, we computed a variety of ML and system metrics. Regarding the ML model evaluation, we recorded all typical ML classification metrics, including prediction accuracy, precision, recall, and F1-score. As the overall trends of accuracy were very similar with the F1-score, we present overall *prediction accuracy*, which shows the percentage of correctly predicted and classified data samples into the appropriate target classes. For the purpose of evaluating the system performance, we used the two most important metrics in distributed stream processing, namely throughput and latency. *Throughput* denotes the number of ingested and processed records per second of each executed experiment (with a fixed set of hyperparameter settings). *Latency*, on the other hand, denotes the time difference between the moment a record is ingested in the system and the moment it produces an output. In stream processing, the goal is to maximize throughput and/or minimize latency.

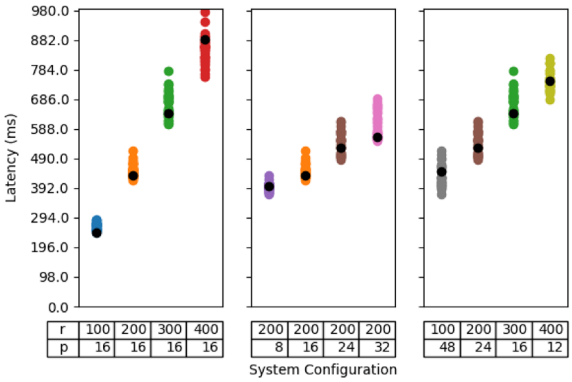


Fig. 2 Average latency per system configuration when executing the social media application with HT model. The black dot corresponds to the hyperparameter settings with the lowest latency of $s1$

4 Experimental results

This section presents the key findings of our experimental study for several selected Spark cluster mode experiments (with the Driver running in "client" deploy mode) due to space constrains. However, the results from the local, pseudo-distributed, and other Cluster mode experiments are similar and lead to the same conclusions.

4.1 Impact of system configurations

For the experiments in this section, we systematically vary two system configurations, namely the records/partition (r) and the partitions/second (p) as shown in Table 4. As frequently reported in the literature, system configurations can significantly impact the system performance in terms of latency and/or throughput. Figure 2 shows the average latency per system configuration, grouped into three logical groups for ease of presentation, when executing the social media application with HT model on Spark cluster mode. Each dot corresponds to the average latency when executing with a distinct set of system configuration and hyperparameter settings. Our first observation is that as we increase r and fix p, the latency increases almost linearly. This is to be expected since a large r means that more records are now batched together and processed within each task, forcing latency, i.e., the time needed from the moment a record is fed into the system until the moment it gets processed, to increase. Fixing r and increasing p also increases latency but to a much lower extend. As the degree of parallelism increases, so are interactions among the tasks as well as shuffling times for exchanging model information between the tasks. Finally, latency increases as r increases but p decreases (while maintaining a fixed batch size). The effect of batching more records into each task is much stronger than the impact of parallelism (as evident from the results from the first two groups), leading to higher overall latencies.

Regarding throughput, increasing either records/partition (r) or partitions/second (p) leads to a general increase of throughput, as shown in Fig. 3 for the same experiments as the ones presented in Fig. 2. This is also expected as in either

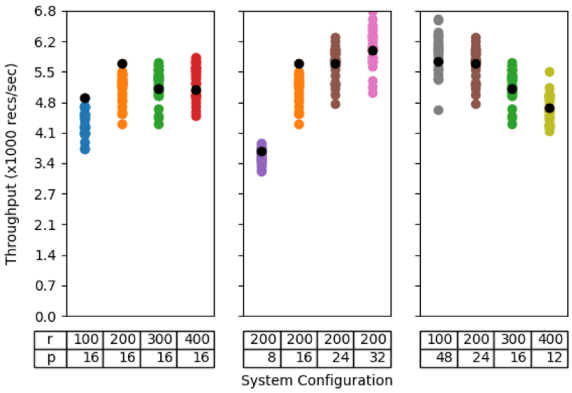


Fig. 3 Average throughput per system configuration when executing the social media application with HT model. The black dot corresponds to the hyperparameter settings with the highest throughput of $s1$

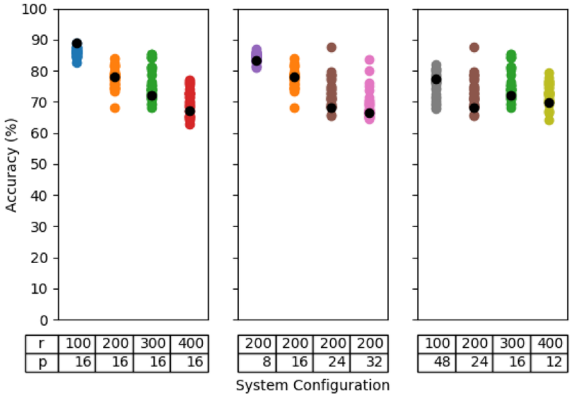


Fig. 4 Prediction accuracy per system configuration when executing the social media application with HT model. The black dot corresponds to the hyperparameter settings with the highest accuracy of $s1$

case, more records are processed overall within a unit of time. However, as indicated by the results of the first group of system configurations, increasing r past 200 records/partition does not yield benefits in terms of throughput as the system has reached its limit for this degree of parallelism ($p = 16$). Opposite to latency, the impact of p is much greater to throughput compared to the impact of r. Hence, in the third group, as p decreases so does the overall throughput, even though the batch size remains constant.

Surprisingly, *the prediction accuracy of the machine learning model being trained is also greatly affected by the various system configurations* as seen in Fig. 4 for the social media application using the HT model. Specifically, we have noticed significant drops in accuracy (ranging between 11% and 23%) while increasing either the records/partition r or the partitions/second p (and keeping the same hyperparameter settings). As explained in Sect. 3.5, in batch-based streaming ML, each task has its own local ML model that gets updated during each batch in isolation from the other tasks. At the end of the batch, all task models are merged together and the new global model is distributed to the tasks of the next batch iteration. Hence, by increasing r, more records are used for training

the individual local task models during each batch. Similarly, by increasing the degree of parallelism p, the global model is updated less frequently in terms of the total number of processed records. This means that instead of training the global model with small and frequent updates, we have less updates that are larger and rarer, causing the model to miss its optimal weights. This is a very important finding that has not been recorded in the literature, with various practical implications. In particular, *a trade-off has been observed between the degree of parallelism and prediction accuracy, where by increasing the former we sacrifice the latter and vice versa*.

Another interesting observation in Fig. 4 is that the hyperparameter settings used in the best execution in terms of accuracy is not necessarily leading to the best (or even a good) execution with other system configurations, as indicated by the black dots in the figure. For instance, the highest accuracy for system configuration $s1$ is 89%. When the same hyperparameters are used with system configuration $s2$ (where the only change comes from increasing records/partition), the achieved accuracy is only 78%, while different hyperparameters yield 84% accuracy for $s2$. Therefore, *optimal hyperparameters can vary according to each system configuration*.

Effect of ML model: We have already demonstrated that different system configurations have an impact on latency, throughput, and accuracy for a particular application and ML model as shown in Figs. 2, 3, 4. Next, we repeated the experiments using a different ML model, namely SGD. Figure 5 shows the same three metrics when using the SGD model for the first two groups of system configurations (due to space constraints). For latency and throughput, the trends are identical with the case of the HT model, while for accuracy we observe a very peculiar behavior. In particular, with the SGD model, prediction accuracy values tend to be grouped into two discrete categories, generated from two groups of hyperparameter settings. The first group of settings tends to achieve high accuracy values, ranging from 85% to 91%, which is slightly impacted by the choice of system configurations. For example, fixing records/partition and increasing partitions/second, increases the spread of the accuracy from 0.2% to 4.1%. The second group of hyperparameter settings leads to lower accuracy values, which are strongly impacted by system configurations. In fact, the range of accuracy values varies widely between 62% and 85% (similar to the HT model). Therefore, some hyperparameters of specific models seem to be more sensitive to system configurations than others. In the case of the SGD model, the culprit is the learning parameter l, which controls the rate of update for the gradient descent. When the value of l ranges below 0.01 (note that the default value of l is 0.001 for SGD), then the overall model accuracy is greatly affected by the degree of parallelism, with a clear inverse relationship. In conlcusion, *the level of impact of system parameters to ML model performance depends on the ML model, and in some cases to specific hyperparameter settings, revealing the complex interactions between system and hyperparameter settings*.

Effect of application: In order to generalize our findings, we repeated the experiments using a very different application from the banking domain. Figure 6 shows the average latency, average throughput, and prediction accuracy for the banking application with HT model when using the first four system configurations listed in Table 4. The banking application requires much simpler processing for extracting

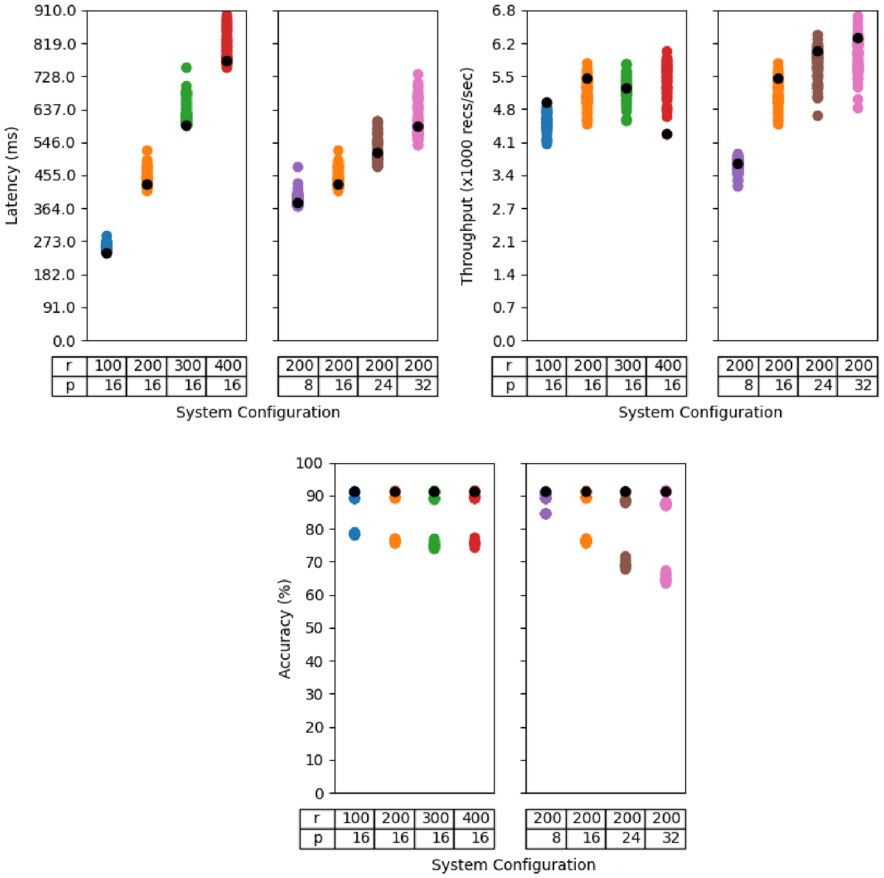
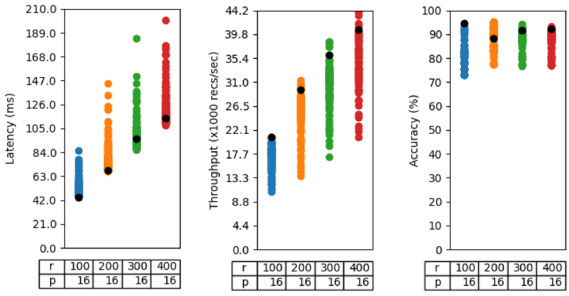


Fig. 5 Average latency, average throughput, and prediction accuracy per system configuration for the first two configuration groups when executing the social media application with SGD model. The black dot corresponds to the hyperparameter settings with the lowest latency, highest throughput, and highest accuracy of $s1$, respectively

Fig. 6 Average latency, average throughput, and prediction accuracy per the first four system configurations when executing the banking application with HT model. The black dot corresponds to the hyperparameter settings with the lowest latency, highest throughput, and highest accuracy of $s1$, respectively



the feature vector compared to the social media one, and as such, is able to achieve lower latency and higher throughput. The trends, however, are the same as the ones observed in Figs. 2, 3; as records/partition increase, so does the latency and the throughput. With regards to prediction accuracy, varying the system configuration while maintaining the same hyperparameter settings lead to big variations in accuracy of up to 21%. This is evident from the black dots shown in the rightmost plot of Fig. 6, which represent runs with the same hyperparameter settings but different records/partition. This observation will become more clear from the results presented in Sect. 4.2 and Fig. 7. Thus, *we conclude that our insights do not apply only to a specific application but rather are applicable to other applications as well.*

4.2 Impact of hyperparameters

While it is widely known that hyperparameters can have a strong impact on a model's prediction accuracy, our results clearly indicate that they can also have a large effect on system performance. This is clearly visible in Figs. 7a, b, where the average latency is shown as a function of the hyperparameter settings, across all three groups of system configurations used for the banking application using the HT model. Consider the latency observed for each experiment when using the second system configuration $s2$ ($r = 200, p = 16$). As we vary the hyperparameter settings, average latency varies between 67ms and 145ms, indicating a difference of up to 2.1x between the highest and lowest achievable latency. The trends are similar for all other system configurations, with the difference ranging from 1.6x to 2.5x between the highest and lowest latency. It is interesting to note that some system configurations seem to be more "robust" to latency variations caused by hyperparameter settings, such as $s5$ ($r = 200, p = 8$), with latency ranging between 59ms–95ms. These are typically associated with lower degrees of parallelism, where fewer individual task models are merged during each batch. The above observation makes identifying which hyperparameters can impact system performance the most a challenging tasks. In the case of the HT, the main case that stands out is when the maximum tree depth (h) is set high (e.g., $h > 20$) and the grace period (g), i.e., the number of records a leaf should observe before a split attempt, is set low (e.g., $g < 200$). In this scenario, the HT can grow faster and bigger, negatively impacting system performance. The observed trends are the same for throughput (see Figs. 7d–f), showing that *hyperparameter tuning has a complex impact on system performance that has not been studied in the past*.

Likewise, Fig. 8 shows the latency as a function of the hyperparameter set across the three groups of system configurations used for the social media application when also using the HT model. For each system configuration, we still notice variations in latency as we vary the hyperparameter settings across our experiments, but to a much lower extend compared to the variations we observed above for the banking application in Figs. 7a, b. The reason for this effect is that the social media application is a much more CPU-intensive application compared to the banking application, thereby dampening the impact of hyperparameter setting changes to system performance. This dampening is more noticeable for system configurations with a relatively low

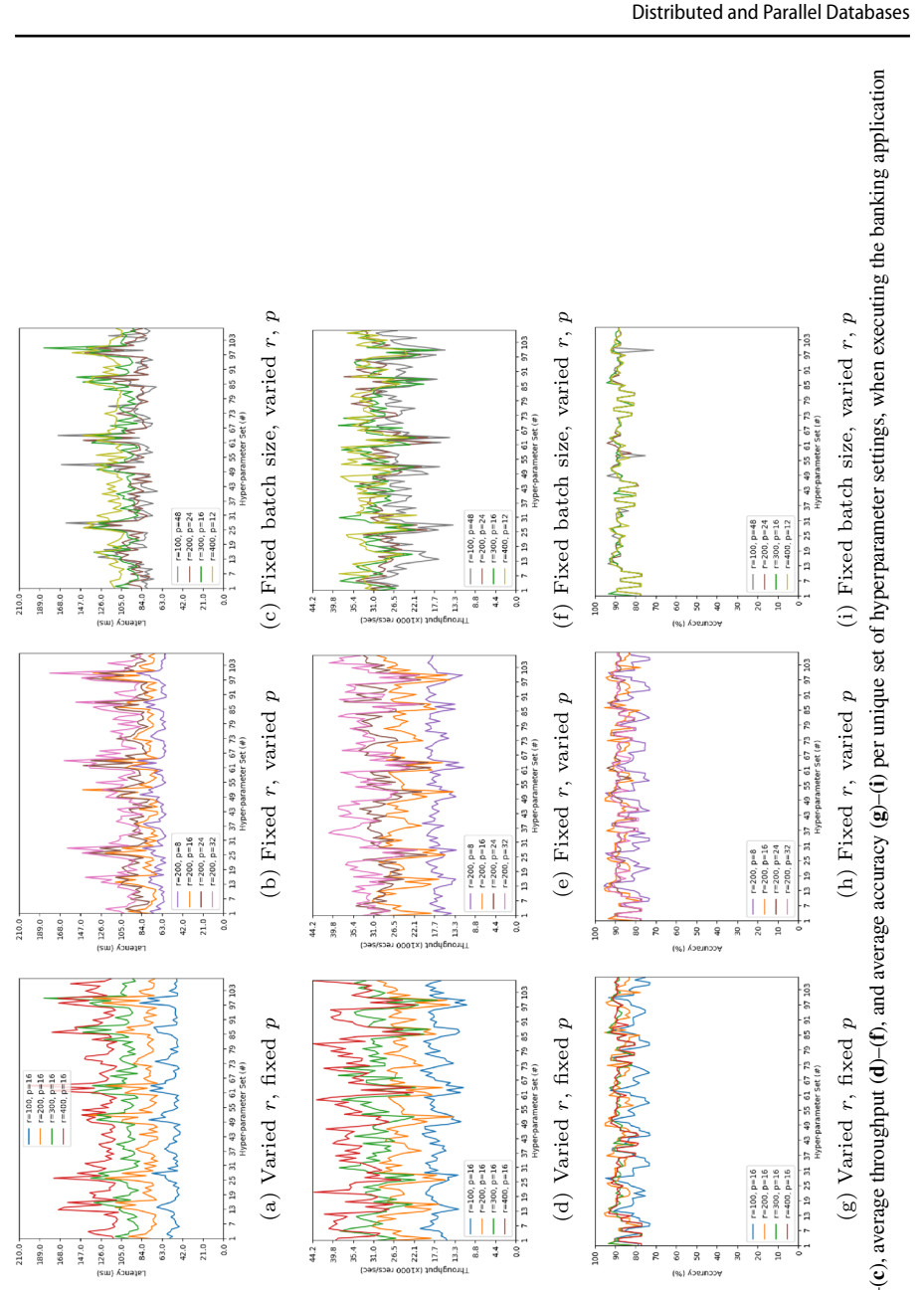
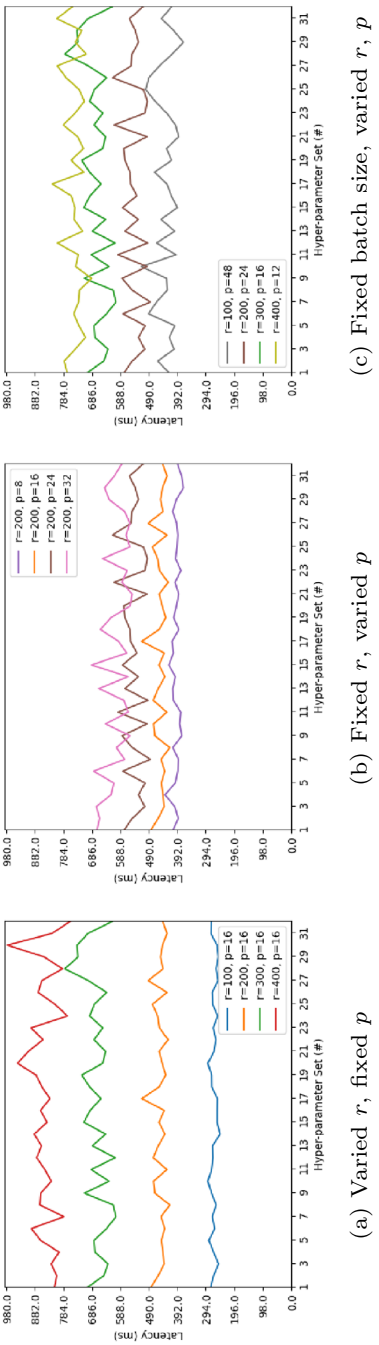
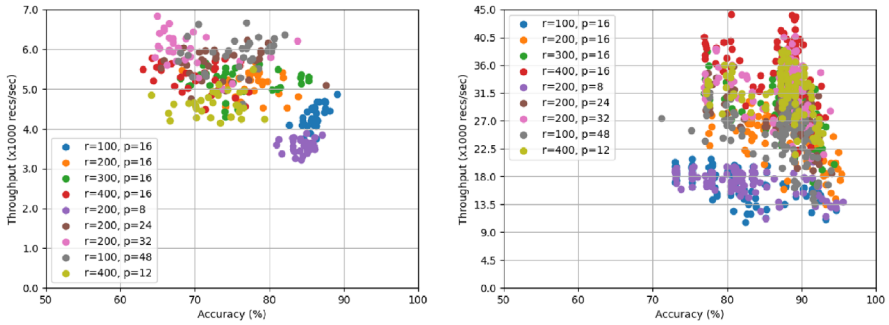


Fig. 7 Average latency (**a**)–(**c**), average throughput (**d**)–(**f**), and average accuracy (**g**)–(**i**) per unique set of hyperparameter settings, when executing the banking application with HT model



(a) Varied r, fixed p (b) Fixed r, varied p (c) Fixed batch size, varied r, p

Fig. 8 Average latency per hyperparameter settings when executing the social media application with HT model



(a) Social media application (b) Banking application

Fig. 9 Average throughput as a function of prediction accuracy for all system configurations and hyperparameter settings when executing (**a**) the social media and **b** the banking application with HT model

degree of parallelism, for which latency fluctuations seem to be restricted. For example, notice configuration $s1$ with the blue line in group $g1$ ($r = 100, p = 16$) or configuration $s5$ with the purple line in group $g2$ ($r = 200, p = 8$), which have the lowest degree of parallelism in our experiments. These two configurations seem to trivially affect latency across their hyperparameter space, in contrary with the rest configurations that tend to fluctuate latency more. Specifically, the standard deviation of latency for $s1$ and $s5$ is $11ms$ and $13ms$ for HT, and $10ms$ and $17ms$ for SGD, respectively. In contrary, configurations with a higher degree of parallelism, such as $s6$ ($r = 200, p = 24$) and $s7$ ($r = 200, p = 32$), have a latency standard deviation of $37ms$ and $44ms$ for HT, and $36ms$ and $48ms$ for SGD, respectively. Overall, *an increased degree of parallelism tends to cause larger latency fluctuations across the hyperparameter space, irrespective of the ML model.*

4.3 Combined impact

In the previous sections we have seen that both system parameters and hyperparameters can impact both system and ML model performance. To further study their interactions, we needed a way of associating prediction accuracy with system performance and at the same time include all system configurations across all the hyperparameters in their space. This is depicted in Fig. 9, where we show the throughput as a function of accuracy for all experiments ran with (a) the social media application and (b) the banking application using the HT model. First, we note that there is no point on either figure that is able to achieve both a very high throughput and a high accuracy. For example, for the social media application, no combination of system configuration and hyperparameters can simultaneously yield throughput greater than $6k$ records per second and accuracy greater that 85%. In addition, there seems to be a trade-off between the two metrics: settings that lead to high throughput typically achieve lower accuracy and vice versa.

Another interesting behavior can also be observed in Fig. 9a. For some system configurations in the social media application, there is a clear clustering of data

points achieving similar throughput and accuracy. This is particularly true for settings with low degree of parallelism such as $s1$ ($r = 100, p = 16$) with the blue color or $s5$ ($r = 200, p = 8$) with the purple color. As the degree of parallelism increases, so does the range of expected values in both accuracy and throughput. However, this observation does not hold for the banking application, where the data points for $s1$ and $s5$ are widely spread across the accuracy dimension, ranging between 73% and 96%. Specifically, it seems that the combination of a high grace period (e.g., $g > 800$) and the use of the "InfoGain" as the split criterion (as opposed to "Gini") when the micro-batch size is small (as in the case with $s1$ and $s5$), leads to a significant decrease in accuracy for the banking application. Therefore, *the type of application can play a crucial role into how the different complex interactions between system parameters and hyperparameters are manifested and thereby impact both throughput and accuracy*.

5 Discussion and future work

Our work includes experiments conducted with different Spark Streaming deploy modes, applications and datasets, machine learning models, hyperparameters, and system parameters; all revealing complex performance interactions between system configuration and hyperparameter settings. Specifically, we have shown that as the total batch size increases, either due to increases of the degree of parallelism (p) or the number of records processed per task (r), so is latency and throughput but typically in the expense of declining prediction accuracy. Even when the total batch size ($p \times r$) is fixed, there are still significant accuracy deviations observed (up to 23%). From our results, it is also clear that the best hyperparameters are not actually the best across all system configurations. In other words, each system configuration has its own locally optimized set of hyperparameters. Beyond that, the magnitude of the degree of parallelism seems to increase the variation range of all the metrics, causing values to be grouped together more sparsely.

A specific impact of micro-batch size on the ML model performance in a streaming environment has been shown in the past, revealing that a relatively decreased batch size allows ML models to train better and thus, achieve higher prediction accuracy values [43, 44]. Conversely, larger micro-batch sizes suffer from a worse model generalization ability, leading to restricted prediction performance [45, 46]. According to a past experimental study, it was empirically proven that, on average, during each training iteration the model's loss function, and thus its generalization ability, is a decreasing function of the batch size [47]. Smaller batch sizes bring the advantage of offering better generalization but at the expense of higher training times. This introduces a trade-off that we also observed in many of our experiments. However, our experimental study has revealed much more complex interactions between batch size and ML model performance. For example, in the case of the banking application, system configurations that induced small batch sizes also experienced some of the lower accuracy values across all experiments, while other configurations with larger batch sizes yielded very high accuracy values (recall Sect. 4.3). In addition, several experiments with the same batch size (namely group $g3$) led to large variations of accuracy values for both applications,

depending on the settings of both system parameters and hyperparameters. In conclusion, while batch size can significantly impact both system and ML model performance, other variables (e.g., type of application, other parameter settings, hyperparameters) can have equal, if not greater, impact.

Furthermore, our experimental analysis shows that hyperparameters do not only affect the model's prediction accuracy, but they also cause an impact to the system's overall performance as well, causing significant variations in both throughput (up to 2.4x) and latency (up to 2.5x). The level of impact depends on the system configurations and the type of application. These are all important lessons that need to be taken into consideration when performing either system parameter tuning or hyperparameter tuning. Our findings show that it is vital to orchestrate the optimization process of both tuning types simultaneously in order to address the variations and trade-offs that arise. The combination of system parameter and hyperparameter tuning introduces new challenges, including a much larger search space as well as complex, non-convex interactions between parameters that affect different aspects of application performance and different evaluation metrics.

5.1 Proposed problem formulation

In this experimental study, we focused on three important and commonly-used objectives that need to be optimized simultaneously, namely latency, throughput, and prediction accuracy. The tradeoffs and intricacies of these objectives have led us to formulate the streaming ML tuning problem as a **multi-objective optimization problem** (**MOOP**). Under MOOP, each objective is expressed with an objective function as well as an upper-bound function associated with the optimal solution. The three objectives are then combined to define the vector-valued objective function $f : \overrightarrow{\mathbb{C}} \times \overrightarrow{\mathbb{P}} \rightarrow \mathbb{R}^3$ as:

$$f(\vec{c},\vec{p}) = \left(f_{lat}(\vec{c},\vec{p}), f_{thru}(\vec{c},\vec{p}), f_{acc}(\vec{c},\vec{p})\right)^T \tag{1}$$

where $f_{lat}(\vec{c},\vec{p})$, $f_{thru}(\vec{c},\vec{p})$, and $f_{acc}(\vec{c},\vec{p})$ represent the objective functions for latency, throughput, and accuracy, respectively, that take as input a specific set of system configuration settings $\vec{c}$ and hyperparameter settings $\vec{p}$. The set $\overrightarrow{\mathbb{C}} \times \overrightarrow{\mathbb{P}}$ represents the *feasible decision space* and corresponds to the Cartesian product of the system configuration parameter space $\mathbb{C}$ and the hyperparameter space $\mathbb{P}$.

Given the conflicting nature of the objectives, there does not exist a feasible solution that maximizes all objective functions simultaneously. Hence, we focus on the Pareto optimal solutions, which are bounded by the *ideal objective vector* z^* defined as:

$$z^*(\vec{c},\vec{p}) = \left(f^*_{lat}(\vec{c},\vec{p}), f^*_{thru}(\vec{c},\vec{p}), f^*_{acc}(\vec{c},\vec{p})\right)^T \tag{2}$$

where $f^*_{lat}(\vec{c},\vec{p})$, $f^*_{thru}(\vec{c},\vec{p})$, and $f^*_{acc}(\vec{c},\vec{p})$ represent the upper-bound functions for latency, throughput, and accuracy, respectively Even though solution z^* does not exist, it denotes the theoretical upper bounds of all objective functions. Hence, we can define the **Pareto Distance** metric D for computing the percent difference of a data point in the feasible decision space from the ideal objective (i.e., the

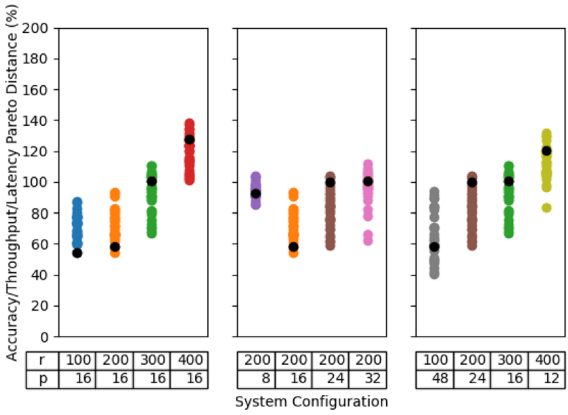


Fig. 10 Pareto distance that combines latency, throughput, and prediction accuracy for all system configurations and hyperparameter settings when executing the social media application with HT model (the lower the better)

theoretical data point that achieves the best latency and the best throughput and the best accuracy):

$$D = \left\| \frac{z^*(\vec{c},\vec{p}) - f(\vec{c},\vec{p})}{z^*(\vec{c},\vec{p})} \right\|, \quad s.t. \, \vec{c}, \, \vec{p} \in \vec{\mathbb{C}} \times \vec{\mathbb{P}} \tag{3}$$

To solve the MOOP, our new objective is to find the solution $(\vec{c},\vec{p})$ that *minimizes* the Pareto Distance D.

Note that other distance metrics could be used in the above formulation such as the weighted sum method, which combines all the multi-objective functions into one scalar and then computes the difference from the ideal solution. The main benefit from our formulation is avoiding the use of weight parameters, whose settings typically fall to the hands of system administrators with little knowledge on how to set them effectively.

Figure 10 shows the computed Pareto distance that combines latency, throughput, and prediction accuracy for all system configurations and hyperparameter settings for the HT model on the social media application. According to the results, the solution that minimizes the Pareto distance uses the system configuration $s8$ ($r = 100$, $p = 48$) and is able to achieve the second best latency (see Fig. 2), the second best throughput (see Fig. 3), and a relatively high prediction accuracy at 83% (see Fig. 4). Similarly, it is apparent from Fig. 10 that system configuration $s4$ ($r = 400$, $p = 16$) yields the worse overall performance, which can be easily verified from Figs. 2, 3, 4.

Note that in the above calculations, we are using two objectives related to system performance and one objective related to ML model performance; therefore, the outcome is biased towards system performance. However, our formulation is easily adjustable as objective functions can be simply added or removed from the objective vector in Eq. 1. We repeated the calculations for the Pareto distance using only throughput and prediction accuracy. In this case, the best performance data point comes from the system configuration $s7$ ($r = 200$, $p = 32$) and corresponds to the top right pink dot in Fig. 9a. Hence, an additional benefit of

our formulation is the added flexibility to the users, who can adjust the objective functions based on which performance metrics they want to optimize.

The use of multi-objective optimization has recently been proposed within the context of hyperparameter tuning for traditional batch-based ML models. One work focused on simultaneously tuning hyperparameters and selecting appropriate features in order to (a) minimize the estimated generalization error (analogous to maximizing prediction accuracy), and (b) maximize feature sparsity to yield better model interpretability [29]. Horn et al. [30] performed a comparative study on Kernelized support vector machines (SVMs), investigating the use of subsampling during training in order to reduce training time, while at the same time minimizing prediction error. Finally, EMORL is a multi-objective reinforcement learning-based hyperparameter tuning method for simultaneously optimizing accuracy and inference latency for eXtreme Gradient Boosting (XGBoost) [48]. The aforementioned approaches only apply to specific batch-based ML models and have a fairly narrow scope. For instance, they attempt to minimize either training time or inference time. However, distributed streaming ML models have vastly different requirements and processing styles since both training and inference can happen at the same time and across parallel tasks. Thus, more general multi-objective approaches must be developed to comprehensively address the combined system parameter and hyperparameter tuning problem in DSPEs.

5.2 Future work

The proposed problem formulation is amenable to various tuning methodologies for finding (near) optimal system parameters and hyperparameter settings for streaming ML applications. For example, the individual objective functions can be treated as black-box, whose values are obtained through experimentation. Hence, an *experiment-driven* approach can be devised that executes the (or part of the) application multiple times with different configuration and hyperparameter settings each time. The key challenge here is how to develop a search algorithm that will utilize the feedback provided after each execution in order to guide the sequence of executions and converge to optimal settings as quickly as possible. While the experiment-driven methodology has been shown to provide near-optimal settings, it is characterized by long tuning times [6, 18].

An alternative methodology would be to develop white-box models for the individual objective functions and follow a *cost modeling* approach to tuning. These approaches are computationally very efficient and can yield highly accurate models. With regards to modeling system performance (such as latency and throughput), there is some past work in the area of system parameter tuning that can be taken advantage of [5, 15]. Equivalently, Bayesian optimization can be used to fit a cheap surrogate model to capture the relationship between hyperparameters and prediction accuracy [29]. However, capturing cross interactions (i.e., how hyperparameters impact system performance and how system parameters impact prediction accuracy) have not been explored yet in cost modeling approaches.

Finally, one of the most promising yet challenging methodologies to employ is an *adaptive* approach that can change system parameters and hyperparameters adaptively while an application is running, based on online performance metrics computed on-the-fly. These approaches enable the performance tuning of ad-hoc applications without assuming any prior knowledge. However, multiple open challenges arise, including how to make the decision for changing any settings, what mechanisms to use for the changes to guarantee application correctness, and how to ensure that changes will not cause any performance or stability issues.

6 Conclusion

This manuscript has performed an extensive experimental analysis of how system parameters and hyperparameter settings can impact both system performance and prediction accuracy of streaming ML applications executed on DSPEs. The results revealed several unexpected and complex interactions, including the impact of system parameters on prediction accuracy, the impact of hyperparameters on system performance, as well as various tradeoffs that are affected by the type of application and ML model used. These results motivate the need for new combined system parameter and hyperparameter automated tuning approaches. Further, given the identified tradeoffs, we formulate the combined tuning problem as a multi-objective optimization problem for finding pareto optimal solutions. Overall, we expect these findings and our generalized problem formulation to open up new research directions in the field of self-managing stream processing systems in the context of streaming machine learning applications.

Author Contributions Conceptualization: HH; Methodology: LO, HH; Formal analysis and investigation: LO, HH; Writing - original draft preparation: LO, HH; Writing - review and editing: LO, HH; Supervision: HH.

Competing interests

The authors declare no competing interests.

References

1. Herodotou, H., Chen, Y., Lu, J.: A survey on automatic parameter tuning for big data processing systems. ACM Comput. Surv. **53**(2), 1–37 (2020)
2. Bifet, A., Holmes, G., Pfahringer, B., Kranen, P., Kremer, H., Jansen, T., Seidl, T.: MOA: Massive Online Analysis, a Framework for Stream Classification and Clustering. In: Proceedings of the First Workshop on Applications of Pattern Analysis, pp. 44–50 (2010). PMLR
3. Hoi, S.C., Wang, J., Zhao, P.: Libol: a library for online learning algorithms. J. Mach. Learn. Res. **15**(1), 495 (2014)
4. Lu, J., Chen, Y., Herodotou, H., Babu, S.: Speedup your analytics: automatic parameter tuning for databases and big data systems. PVLDB **12**(12), 1970–1973 (2019)

5. Kalim, F., Cooper, T., Wu, H., Li, Y., Wang, N., Lu, N., Fu, M., Qian, X., Luo, H., Cheng, D.: Caladrius: a performance modelling service for distributed stream processing systems. In: 35th International Conference on Data Engineering (ICDE), pp. 1886–1897 (2019). IEEE
6. Bilal, M., Canini, M.: Towards automatic parameter tuning of stream processing systems. In: Proceedings of the 2017 Symposium on Cloud Computing (SoCC), pp. 189–200 (2017). ACM
7. Wang, C., Meng, X., Guo, Q., Weng, Z., Yang, C.: Automating characterization deployment in distributed data stream management systems. IEEE Trans. Knowl. Data Eng. **29**(12), 2669–2681 (2017)
8. Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M.J., Recht, B., Stoica, I.: Drizzle: fast and adaptable stream processing at scale. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), pp. 374–389 (2017). ACM
9. Feurer, M., Hutter, F.: Hyperparameter Optimization. In: Automated Machine Learning, pp. 3–33 (2019). Springer, Cham
10. Padierna, L.C., Carpio, M., Rojas, A., Puga, H., Baltazar, R., Fraire, H.: Hyper-parameter tuning for support vector machines by estimation of distribution algorithms. In: Nature-inspired Design of Hybrid Intelligent Systems, pp. 787–800 (2017). Springer
11. Bardenet, R., Brendel, M., Kégl, B., Sebag, M.: Collaborative hyperparameter tuning. In: International Conference on Machine Learning, pp. 199–207 (2013). PMLR
12. Feurer, M., Klein, A., Eggensperger, Katharina Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. Adv. Neural Inf. Process. Syst. **28**, 2962–2970 (2015)
13. Odysseos, L., Herodotou, H.: Exploring system and machine learning performance interactions when tuning distributed data stream applications. In: IEEE 38th International Conference on Data Engineering Workshops (ICDEW), pp. 24–29 (2022). IEEE
14. Herodotou, H., Odysseos, L., Chen, Y., Lu, J.: Automatic performance tuning for distributed data stream processing systems. In: IEEE 38th International Conference on Data Engineering (ICDE), pp. 3194–3197 (2022). IEEE
15. Bansal, M., Cidon, E., Balasingam, A., Gudipati, A., Kozyrakis, C., Katti, S.: Trevor: Automatic configuration and scaling of stream processing pipelines. CoRR **abs/1812.09442** (2018)
16. Kroß, J., Krcmar, H.: Model-based performance evaluation of batch and stream applications for big data. In: IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 80–86 (2017). IEEE
17. Lin, J., Lee, M., Yu, I.C., Johnsen, E.B.: Modeling and simulation of spark streaming. In: IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA), pp. 407–413 (2018). IEEE
18. Liu, X., Dastjerdi, A.V., Calheiros, R.N., Qu, C., Buyya, R.: A stepwise auto-profiling method for performance optimization of streaming applications. ACM Trans. Auton. Adapt. Syst. **12**(4), 1–33 (2017)
19. Li, T., Tang, J., Xu, J.: Performance modeling and predictive scheduling for distributed stream data processing. IEEE Trans. Big Data **2**(4), 353–364 (2016)
20. Petrov, M., Butakov, N., Nasonov, D., Melnik, M.: Adaptive performance model for dynamic scaling apache spark streaming. Procedia Comput. Sci. **136**, 109–117 (2018)
21. Yogatama, D., Mann, G.: Efficient transfer learning method for automatic hyperparameter tuning. In: Artificial Intelligence and Statistics, pp. 1077–1085 (2014). PMLR
22. Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., Hutter, F.: Auto-Sklearn 2.0: hands-free AutoML via meta-learning. J. Mach. Learn. Res. **23**(261), 1–61 (2020)
23. Vogel, A., Griebler, D., Danelutto, M., Fernandes, L.G.: Self-adaptation on parallel stream processing: a systematic review. Concurrency and Computation: Practice and Experience, 6759 (2021)
24. Kotthoff, L., Thornton, C., Hoos, H.H., Hutter, F., Leyton-Brown, K.: Auto-WEKA: Automatic model selection and hyperparameter optimization in WEKA. In: Automated Machine Learning, pp. 81–95 (2019). Springer, Cham
25. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
26. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 847–855 (2013)

27. Eibe, F., Hall, M.A., Witten, I.H.: The WEKA Workbench. Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques. In: Morgan Kaufmann, (2016)
28. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. ACM SIGKDD Explor. Newsl **11**(1), 10–18 (2009)
29. Binder, M., Moosbauer, J., Thomas, J., Bischl, B.: Multi-objective Hyperparameter Tuning and Feature Selection using Filter Ensembles. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, pp. 471–479 (2020). ACM
30. Horn, D., Demircioğlu, A., Bischl, B., Glasmachers, T., Weihs, C.: A comparative study on large scale kernelized support vector machines. Adv. Data Anal. Classif. **12**(4), 867–883 (2018)
31. Veloso, B., Gama, J., Malheiro, B.: Self hyper-parameter tuning for data streams. In: International Conference on Discovery Science, pp. 241–255 (2018). Springer
32. Carnein, M., Trautmann, H., Bifet, A., Pfahringer, B.: confStream: automated algorithm selection and configuration of stream clustering algorithms. In: International Conference on Learning and Intelligent Optimization, pp. 80–95 (2020). Springer
33. Lal, D.K., Suman, U.: Towards comparison of real time stream processing engines. In: 2019 IEEE Conference on Information and Communication Technology, pp. 1–5 (2019). IEEE
34. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI), pp. 2–14 (2012). USENIX Association
35. Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., Markl, V.: Benchmarking distributed stream data processing systems. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1507–1518 (2018). IEEE
36. Hesse, G., Matthies, C., Perscheid, M., Uflacker, M., Plattner, H.: ESPBench: The enterprise stream processing benchmark. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering, pp. 201–212 (2021)
37. Herodotou, H., Chatzakou, D., Kourtellis, N.: Catching them red-handed: real-time aggression detection on social media. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 2123–2128 (2021). IEEE
38. Herodotou, H., Chatzakou, D., Kourtellis, N.: A streaming machine learning framework for online aggression detection on Twitter. In: International Conference on Big Data, pp. 5056–5067 (2020). IEEE
39. Founta, A.M., Djouvas, C., Chatzakou, D., Leontiadis, I., Blackburn, J., Stringhini, G., Vakali, A., Sirivianos, M., Kourtellis, N.: Large scale crowdsourcing and characterization of Twitter abusive behavior. In: Twelfth International AAAI Conference on Web and Social Media (2018)
40. Agrawal, R., Imielinski, T., Swami, A.: Database mining: a performance perspective. IEEE Trans. Knowl. Data Eng. **5**(6), 914–925 (1993)
41. Bifet, A., Maniu, S., Qian, J., Tian, G., He, C., Fan, W.: StreamDM: Advanced data mining in spark streaming. In: 2015 IEEE International Conference on Data Mining Workshop (ICDMW), pp. 1608–1611 (2015). IEEE
42. Domingos, P., Hulten, G.: Mining High-speed data streams. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 71–80 (2000). ACM
43. Kandel, I., Castelli, M.: The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset. ICT Express **6**(4), 312–315 (2020)
44. Smith, S.L., Le, Q.V.: A Bayesian Perspective on Generalization and Stochastic Gradient Descent. arXiv preprint arXiv:1710.06451 (2017)
45. LeCun, Y.A., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient backprop. In: Neural Networks: Tricks of the Trade, pp. 9–48 (2012). Springer
46. Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P.: On Large-batch Training for Deep Learning: Generalization Gap and Sharp Minima. arXiv preprint arXiv:1609.04836 (2016)
47. Qian, X., Klabjan, D.: The impact of the mini-batch size on the variance of gradients in stochastic gradient descent. arXiv preprint arXiv:2004.13146 (2020)
48. Chen, S., Wu, J., Liu, X.: EMORL: effective multi-objective reinforcement learning method for hyperparameter optimization. Eng. Appl. Artif. Intell. **104**, 104315 (2021)