World Wide Web
https://doi.org/10.1007/s11280-022-01061-y

Zebra: A novel method for optimizing text classification query in overload scenario

Tianhuan Yu[1] · Zhenying He[2] · Zhihui Yang[3] · Fei Ye[2] · Yuankai Fan[2] · Yinan Jing[2] · Kai Zhang[2] · X. Sean Wang[2]

Received: 20 January 2022 / Revised: 6 April 2022 / Accepted: 26 April 2022

Abstract
Text classification is a crucial task in the text mining field, and it can be included in queries with user-defined functions(UDF). In many web applications, such as Twitter mining or Weibo real-time processing, when the amount of text data to be processed is enormous, there will be many overload phenomena. At the same time, when the system is overloaded, the delays in the query process can negatively affect the user experience in a streaming scenario. This paper focuses on the query with text classification on streaming data. We propose a novel method called Zebra with progressive pipelines to optimize the overload query situations. The core module of Zebra is the probabilistic filter which can reduce an incredible amount of text data based on semantic information of the query predicate. We train weak classifiers as filters using data with labels from brute-force pipelines. Next, we use a parameter search method to choose a suitable filter with the best settings and apply it to progressive pipelines. Experiments with several text workloads on real-world datasets show that Zebra can achieve higher accuracy stably while answering the query in time.

Keywords Query processing · Text classification · Overload · Probabilistic filter · Load shedding

1 Introduction

With the explosive development of the Internet, an incredible amount of text data is produced every day. For example, according to the statistics of 2020,[1] there were 187 million daily active users on Twitter, and the number of only movies and TV-themed tweets alone produced can reach 7000 per minute. Besides, 2 billion tweets about sports were produced all the year. Generally, text data contains more semantic information than traditional data, such as the information on whether the document is positive or the

[1] https://blog.hootsuite.com/twitter-statistics/

✉ Zhenying He
zhenying@fudan.edu.cn

Extended author information available on the last page of the article

keywords contained in the text. This indirect and relational information is always produced after executing machine learning user-defined functions(UDF). In the meantime, with the rapid rise of artificial intelligence, many relational data platforms or systems support the analysis of unstructured data such as text, images, or videos [2, 5, 28, 36] more popularly. The users also demand to post queries to dig out opaque information about the related data they focus on nowadays.

Many video query processing data platforms or engines including BlazeIt [19], MIRIS [4], NoScope [18], probabilistic predicates [29], SVQ [42], etc, have been proposed to accelerate the execution of video analytics in recent years. These engines work and optimize the pipeline, which mainly focuses on video analytics queries. These work all rely on totally offline training phases based on the UDF and predicates of the query. The other work, including opaque filters [11], save the cost time based on different SQL clauses like limit and join. None of these work try to optimize the query pipeline with machine learning tasks in an online scenario, especially overload situations in stream systems. However, processing the machine learning UDF query offline, which is based on the historical data results, needs a lot of related data to analyze the query.

Compared to offline query processing, streaming data pay much more attention to the query answer latency. In many data stream applications, processing delay is the most critical quality requirement since the value of query results decreases dramatically over time. It's very common in Data Stream Management Systems(DSMSs) that the ability to remain a desired level of delay is significantly damaged under overload situations. In general, DSMSs employ the load shedding technique in order to meet quality requirements and keep pace with the high rate of data arrivals [38]. For the system to continue to provide up-to-date query responses, load shedding always discards some fraction of the unprocessed data. In the streaming data scenario, the system typically works on unbounded data streams rather than static data sets. In fact, people always use sliding windows to limit the recent data over a stream, and the size of a window is often specified based on a time interval like 5 seconds or 10 seconds. The window ends at the current time, called a *suffix window*. In the background of the explosion of data and limited resources of computing platforms, the situations of data overloading frequently happen in a real-world scene.

We focus on a new scenario about a text stream system with overload situations and propose a new query pattern with text classification UDF and the limited time of the query execution. Generally, when the system is overloaded, DSMSs employ the load shedding technique without considering potential semantic information of the query with analysis UDF. The system usually sheds redundant data which can not be processed in a random way. Although this method can guarantee the query is answered in a limited time, the accuracy of the response is relatively low. Notice that when the overload percent is more serious, the accuracy of the query drops more rapidly, and the load shedding technique will shed the data which satisfies the query easily. For example, we suppose that the number of input documents is 1000, the system can process 100 documents per second based on real-time UDF, and the user hopes to receive the result within 5 seconds. In this situation, the load shedding technique will shed 50% of documents indiscriminately, whether the document is positive or negative for the real-time query. Some samples are absolutely not satisfied with the query predicate, but they have

the same shedding rate as the positive samples. In summary, the load shedding technique can help data stream reduce load, and this method is simple and brute. Besides, the overhead of applying this method is very low, and this technique is easy to implement. However, as described above, the query pipelines have low accuracy when only including load shedding, and this method is not suitable for machine learning inference queries.

In this paper, our optimization target is to improve the accuracy of the query and returning more useful data to users. Intuitively, we can shed data which tends to be negative based on the predicate and UDF of the query. Based on such motivation, We apply the probabilistic filter technique to the pipeline to help us select text data. However, there are two challenges in applying the probabilistic filter technique. The first is that how to train effective filters quickly. The second is how to adapt the filtering module to the real-time query and improve the query accuracy. Faced with these two challenges, we propose Zebra method to optimize the query when the data stream is overloaded. We train some classifiers as filters based on an online training phase, and the training data comes from previous online sliding data windows. We apply the filtering module to progressive pipelines, which helps improve the accuracy of the query pipelines and selects data biasedly. In contrast to brute-force pipelines with load shedding, this method improves the processing data efficiency of the system. In the next stage, we use an optimized parameter search method to find the best filter parameters adapted to the real-time query based on metrics. The parameter search helps solve the second challenge. Ultimately, this series of optimization methods will work and improve the query accuracy in overload scenarios. Moreover, we consider the overload scenario with dynamic data distribution and propose Zebra-Dynamic to deal with it. In summary, this paper claims to make the following contributions:

- We propose a query processing frame with text classification UDF, which supports users to propose the query response time under the stream scenario.
- We apply the probabilistic filter technique to progressive pipelines, which can improve the query accuracy when the data stream is overloaded. In progressive pipelines, we propose a progressive training style to train classifiers as filters and use an optimized search method to find the best configuration for the real-time query.
- We integrate these optimizations into the method called Zebra and conduct experiments on various query workloads and datasets. Experimental results show that Zebra can achieve higher query accuracy compared with the baseline. Besides, in the experiments, we use a function fitting method to control the latency of the query.
- We further consider the overload scenario with dynamic data distribution and propose Zebra-Dynamic to optimize the query. Experiments show that Zebra-Dynamic performs better than original Zebra method and baseline in this scenario.

The rest of this paper is organized as follows: Section 2 describes the problem and introduces the brute-force pipeline and the progressive pipeline. Section 3 describes the probabilistic filters we used. Section 4 presents the overview of Zebra method and explains how Zebra method works. Section 4 also introduces how to deal with dynamic distribution scenario. Section 5 presents the results of our experiments. In the end, we summarize the related work in Section 6 and draw the conclusion in Section 7.

Table 1 Frequently used notations

Notation	Description
W	number of documents that can be processed in the bounded time under **normal** conditions
N_d	number of documents that need to be processed in the bounded time under **overload** conditions
UDF	user defined function(text classification tasks)
x	the rate of random sampling before pipeline
r_f	**data reduction rate** of filters
c_f	cost time of using filters
u	cost time of UDF processing per document
t_{bound}	the **bounded answer time** of query given by users
t_{extra}	the extra time reserved for extra modules in progressive pipelines
t_{udf}	the time for text classification UDF
th	**threshold** of one filter
a_f	the **accuracy** of trained filters which can be tuned by **threshold**

2 Problem definition & processing pipelines

In this section, we focus on introducing the problem we solved and the overview of pipelines in our method. Table 1 describes the important notations and their descriptions in the paper.

2.1 Problem description

We support users to give a tuple that includes query(UDF) and the limited time that users hope to spend on waiting for the query. The tuple is as follows : $< query, t_{bound} >$. The *query* includes the specific UDF task given by users, such as sentiment analysis or the topic of documents with the concrete predicate value. For example, in sentiment analysis, some users want to find the highest level positive documents. The details about the query pattern are as follows:

SELECT $*$ **FROM** `TwitterSet`
WHERE`isPositiveUDF(document)` $=$ **True**

The SQL query above is the example of the query pattern, which always includes a text-related UDF and this UDF is a costly machine learning task including deep learning models with a deep neural network. In our paper, we focus on the text classification tasks, and our UDF consists of text classification only like presented in Table 1. In the example query, the user hopes to find the documents in TwitterSet, which mainly have positive emotions. The potential values of the sentiment predicate UDF can be three types: positive, neutral, and negative. Other typical classification tasks of text data are spam detection, keyword extraction, the topic of documents, etc. Obviously, users can not propose the query with unsupported UDFs.

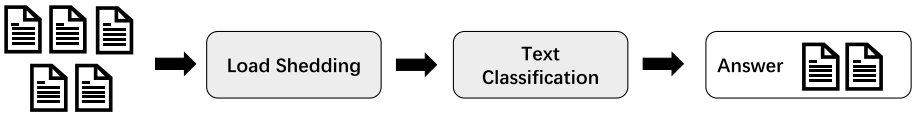

Fig. 1 Brute-force pipeline

Next, the t_{bound} can be considered as the desired answering time of the query that users can accept. In the stream scenario, the latency of the query is crucial, and it is critical for answer quality. Based on this, we hope that the accuracy of the query answer is higher than before, which means that the answer can bring more related data to users. Therefore, the main goal of our problem is as follows: users can get the answer to the query within their expected time, and on this basis, the system schedules all the resources and makes the accuracy of the answer as high as possible.

2.2 Pipeline overview

When the query arrives, the system will process the query on a recent data window in streaming data. In the stream scenario, the stream system usually processes data in a window unit, and the size of the window is usually fixed. For example, the TwitterSet comes as streaming data, and we control the window size by limiting the documents data in recent 5 minutes. We can obtain the data windows by documents' timestamps. In TwitterSet, each Tweet has a timestamp that records the time when users sent it. We utilize timestamps to limit the data window size. When the number of recent tweets has an explosive growth, the data text in the recent data window will become much more than usual. Under this condition, the system is overloaded and needs to take some different techniques to solve this problem.

Brute-force pipeline Without any optimization techniques or methods, when overloaded, the system must shed load in order to maintain the low-latency of the query [38]. The brute-force pipeline is as follows (Fig. 1):

We can see that the data window will firstly be input into the load shedding module, where the system will randomly drop some documents. The random drop stage can be seen as sampling the documents in a real-time data window. Next, the remaining documents are passed into the UDF module, which is related to the real-time query. After the rest of the documents are input to UDF, the system will return the answer to the user, and the UDF module costs most of the time in the pipeline.

In a brute-force pipeline, we can pre-compute some metrics of the answer by assigning the parameters in all pipeline modules in advance. We attempt to represent two metrics, the accuracy and the response time of the pipeline, by using the notations in Table 1. We assume that when the system is overloaded, the random shedding rate of the load shedding stage is x, the number of documents in the recent time window is N, and per document costs time u evenly. In the end, we can get two metrics as follows:

$$\begin{aligned} Accuracy &= x, \\ Cost &= N * x * u \leq t_{bound} \end{aligned} \tag{1}$$

The sample rate is x, and the true positives for the query predicate will be reduced into $N * x$. Eventually, the ratio of the last true positives and original true positives is x, so the

accuracy of the whole pipeline is x. The number of documents passed into the UDF module is $N * x$, which results in $N * x * u$ cost time. In the end, the total cost time needs to be limited within t_{bound}.

Progressive pipeline Compared with a brute-force processing pipeline, our method adds an intermediate module to improve the whole pipeline (Fig. 2). The input streaming data is assigned to the load shedding module also. After that, the remaining documents are passed into our trained probabilistic filters rather than the UDF module. The probabilistic filters will filter the documents semantically based on query predicate and tend to select the document which can pass the query condition in high probability. Eventually, the left unprocessed documents will be input into UDF tasks, and the answer is accomplished. When faced with the data stream overload problem, the filtering module will discard the data that is more inclined not to satisfy the query conditions and help reduce the load. Compared with the brute-force pipeline, more positive samples are preserved, which improves the query pipeline accuracy. Therefore, the system can process the query more efficiently, reducing the cost time of negative samples. However, injecting probabilistic filters into the pipeline brings the problem of accuracy that filters do not change the false-positive rate but can increase the false-negative rate. Notice that machine learning tasks are tolerant to this kind of error, and even the origin machine learning UDFs have the problem of true positives and false negatives [29]

As for our pipeline, we also assume that when the system is overloaded, we set the random rate of load shedding as x, and the number of documents is N. There are three variables of filters which are accuracy a_f, cost time c_f and reduction rate r_f. In the end, we can get two metrics as follows:

$$\begin{aligned} Accuracy &= x * a_f, \\ Cost &= N * x * c_f + N * x * (1 - r_f) * u \leq t_{bound} \end{aligned} \tag{2}$$

Notice that two modules process the original documents, firstly load shedding and secondly filtering. If the accuracy of the applied filter is a_f, there will be $a_f * x * N$ true positive documents input into the UDF. Thus, the result of the accuracy will be $x * a_f$. Next, we consider the cost time of the pipeline. Firstly, the unit time of the filter processing a single document is c_f. The number of documents after the load shedding module will reduce into $N * x$ for the random rate is x. Therefore, the cost time of the filter stage is $N * x * c_f$. Secondly, as the reduction rate of the filter is r_f, the number of documents input into UDF will be $N * x * (1 - r_f)$. Ultimately, the total cost time of the pipeline will be like (2), which should be restricted within t_{bound}.

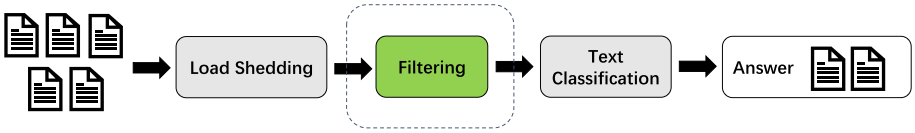

Fig. 2 Progressive pipeline

3 Probabilistic filters

In this section, we introduce the classifiers we used as filters and the dimension reduction technique we applied to data. Next, we describe the details of the training filter stage.

3.1 Classifier 1: Perceptron

First of all, we consider perceptron [10], which is a binary classifier for supervised learning. Perceptron is a linear classifier that makes its predictions based on a linear predictor function combining a set of weights with a feature vector.

In general, the binary classifier learned by perceptron can be seen as a threshold function that maps its input $\mathbf{x}$ (a real-valued vector) to an output value $f(\mathbf{x})$(a single binary value 0 or 1):

$$f(\mathbf{x}) = \begin{cases} 1 & if \quad \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & otherwise \end{cases} \tag{3}$$

in which $\mathbf{w}$ is a vector of real-valued weights, $\mathbf{w} \cdot \mathbf{x}$ is the dot product $\sum_{i=1}^{n} w_i x_i$, where n is the number of inputs of the perceptron, and b is the *bias*. The bias shifts the decision boundary away from the origin and does not depend on any input value. In the context of deep learning, a perceptron can be considered as an artificial neuron using the Heaviside step function as the activation function. Different from a multilayer perceptron, the common perceptron is also called a single-layer perceptron. In the training phase, the main target is to determine the real-valued weight vector $\mathbf{w}$ in (3). Moreover, the threshold function of linear SVM is the same as a perceptron, so we only pick one of them.

3.2 Classifier 2: Neural network classifier

Next, we consider the more complex version of perceptron or multilayer perceptron, which is well-known and solid in CV and NLP research fields called deep neural network(DNN) [24]. To a certain extent, the classifier can have multiple fully connected layers interpreted as varying input data with different matrices sequentially.

$$\begin{aligned} f^i(\mathbf{x}) &= g^i\left(W_i \cdots f^{i-1}(\mathbf{x}) + b_i\right), \\ g^i(x) &= ReLU(x) = max(0, x) \end{aligned} \tag{4}$$

The function g_i generally is a non-linear activation that receives the results from each fully connected layer, and commonly-used activation functions are ReLU, sigmoid, etc. Such a non-linear activation function guarantees the non-linearity of the whole model, which has an advantage over other linear classifiers when the system processes data with non-linearly distribution.

Obviously, DNN has performed excellently in various machine learning tasks [22, 25], concentrating on CV and NLP fields. However, on the other hand, the model is on a larger scale and needs to train more parameters than the other traditional machine learning classifiers.

3.3 Classifier 3: Bernoulli naive Bayes classifier

To filter the sample data that disagrees with the predicate's condition, we also consider naive Bayes classifiers [9], which are easy to build without complicated iterative parameter estimation. Even though naive Bayes is simple, the naive Bayesian classifier often performs surprisingly well and is widely used because it often outperforms more sophisticated classification methods. This classifier works mainly based on Bayes' Theorem, which can be stated mathematically as the following equation:

$$P(A\|B) = \frac{P(B\|A)P(A)}{P(B)} \tag{5}$$

where A and B are events and $P(B)$ can not be equal to zero. Meanwhile, the Bayes classifiers always observe a naive assumption which is also well-known. The assumption supposes that the features of the sample are independent of each other. In this situation, we consider the Bernoulli NB classifier, which can be seen as a binary classifier to separate the labeled data into two classes.

In the multivariate Bernoulli event model, features are independent Booleans (binary variables) describing inputs. If we use x_i to describe the occurrence or absence of the i th term from the vocabulary, then the likelihood of a class C_k of a document is given by [33]

$$p(\mathbf{x}\|C_k) = \prod_{i=1}^{n} p_{k_i}^{x_i}\left(1 - p_{k_i}\right)^{(1-x_i)} \tag{6}$$

where p_{k_i} is the probability of class C_k generating the term x_i. This model is prevalent for classifying short texts, and in our scenario, it's suitable as a weak classifier.

3.4 Dimension reduction

Real-world input data always have high dimensions. When the corpus of the dataset is vast, the dimension of input text will be very high. If we feed the input text into the filters or UDF as a high-dimensional sparse vector, the complexity of the inference stage will be too high for computing resources to support. Dimension reduction techniques can reduce the input text dimension and the inference complexity of models, which can be seen as helping lighten the overload situation. Therefore, before inputting to filters, we need to apply dimension reduction techniques.

The current mainstream dimension reduction techniques in machine learning include **Principal Component Analysis (PCA)** [17] and **Feature Hashing (FH)** [40]. In practice, we mainly use feature hashing to reduce dimension. Compared with PCA, feature hashing can be thought of as a simplified form and needs no training, which performs well when the feature vector $\mathbf{x}$ is very sparse. Normally, the equation of feature hashing is as follows, and it has two hash functions h_1 and h_2:

$$\psi_i^{(h_1,h_2)}(\mathbf{x}) = \sum_{j=1}^{n} \mathbf{1}_{h_1(j)=i} \cdot h_2(j) \cdot \mathbf{x}_j, \forall i = 1 \ldots m \tag{7}$$

where the original dimension of the feature vector $\mathbf{x}$ is n and the output after feature hashing is m. It has been shown that feature hashing is inexpensive and unbiased to data.

3.5 Training filter stage

Assuming we have the training data, we can train the filters online for a real-time query. The filter can be characterized as two partition: **Training Set D** and **Accuracy a**. The specific kinds of filters have been introduced in the previous section. The set **D** has two kinds of data blobs, and each blob has a label $+1$ or -1, which agrees or disagrees with the query. The accuracy **a** represents the accuracy of the filter, which can be tuned dynamically during the pipeline, and **a** has a relationship with $\mathbf{r_a}$ which means the portion of data thrown away by the filter.

About training set If we train the filter on the whole training data set, it can easily lead to overfitting. To avoid this situation happening, we randomly divide the set into training and validation parts. The filter is trained on the D_{train} part while validated on the D_{val} part, and the relation between filter accuracy and data reduction is calculated on validation part D_{val}.

About accuracy Once we have trained the filter, the parameters of the filter are determined, and we can use the following decision function to predict the labels of new input data:

$$Filter(\mathbf{x}) = \begin{cases} +1 & if \quad f(\mathbf{x}) > th_a, \\ -1 & otherwise \end{cases} \tag{8}$$

$f(\mathbf{x})$ represents the output of the filter classifier, and $\mathbf{x}$ is the input feature to filter. The input feature is a simple representation of the documents. In the text tasks, $\mathbf{x}$ is usually tokenized word vectors for documents. th_a represents the threshold in the filter, and it is crucial for calculating the relation between accuracy and data reduction. If we fix the value of accuracy a, we choose the th_a as follows:

$$th_a = max\ th\ , \frac{|\{x \in D : f(\mathbf{x}) > th\ \&\ label(x) = +1\}|}{|\{x \in D : label(x) = +1\}|} \geq a \tag{9}$$

Different values of th_a can lead to different accuracy and reduction rates, and once the threshold is fixed, the decision function of a filter is deterministic. If we let $th = 0$, all data samples will pass the filter, resulting with accuracy $a = 1$ and reduction rate $r_a = 0$. At the same time, we can get the r_a as follows:

$$r_a = 1 - \frac{\left|\left\{x \in D : f(\mathbf{x}) > th_a\right\}\right|}{|D|} \tag{10}$$

From Equation (9) and Equation (10), we can see that the data reduction rate varies with the filter's accuracy. We can use the trained filter to predict data samples on D_{val}, and we will get a series of $f(\mathbf{x})$ of the data in D_{val}. Then the samples on validation are ranked in ascending order according to their probability values $f(\mathbf{x})$, and we use these data with labels to characterize the accuracy-data reduction curve r_a. In Figure 4, we can find that when the filter's threshold is $th_{1.0}$, the documents passed by the filter are 12, and 10 of them are positive, which leads to accuracy $a = 1.0$. At the same time, 4 documents are discarded by the filter. When the threshold is set to the second one, we can get the accuracy $a = 0.8$ because two true positive data are abandoned, and more documents are thrown away, causing a high data reduction rate.

In the end, our method can compute the accuracy and reduction rate curve like the example in Figure 3.

4 Zebra method overview

This section shows the whole overview of Zebra method. Specifically, we show how we prepare training data and train the filters in overload scenarios. Then we introduce how to calculate the relations among accuracy, threshold, and data reduction rate. Finally, this section states how we apply these filters to pipelines and how we determine the parameters.

4.1 Zebra overall architecture

Figure 4 shows all modules in Zebra. Firstly the query arrives, and in the previous few pipelines, the method will process the query and streaming text data in the brute-force way that it will abandon text data randomly; when the trigger condition is satisfied, the method will start to train the filter using the labeled data collected in brute-force pipelines. The trigger condition is described in Section 4.2. Once the filter is available, the future pipelines will become progressive pipelines with applying filters in the inference stage. Before entering the load shedding module in pipelines, the method selects the values of parameters using the Parameter Search module. As shown in Figure 3, the query is a long-running query for all pipelines. Notice that the arrows with different colors in the figure mean whether the filtering module is available, and each pipeline has its own input window data. Besides, the arrows represent the timing relationship, and the arrow endpoint pipeline will run after the arrow starting point.

4.2 Data preparation for training filter

At the beginning stage of our method, we can not utilize our filters and apply them to the pipeline because we have not trained filters yet, and in the meanwhile, we do not have the training data suitable for our real-time query task. So, firstly, we need to get some labeled data for training filters.

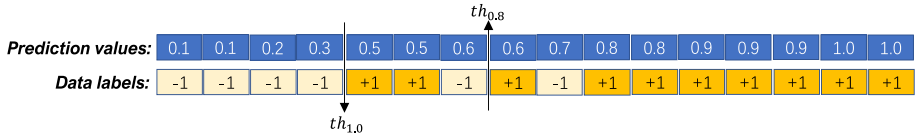

Fig. 3 Documents in D_{val} are ranked in ascending order according to prediction values. The second array is the label values of the corresponding documents

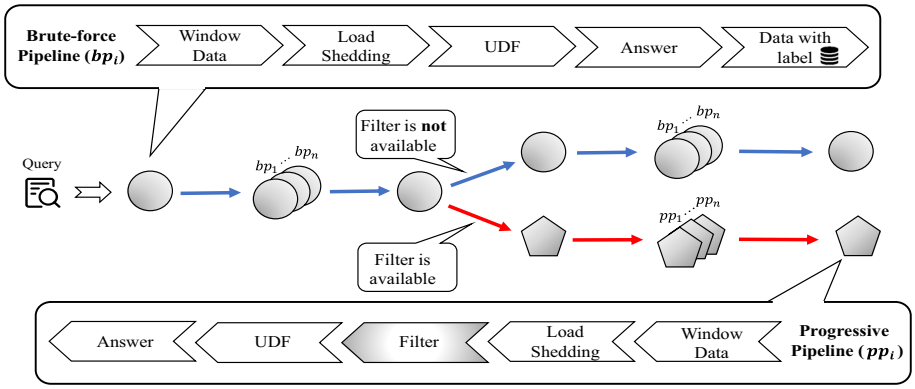

Fig. 4 Zebra architecture overview

Cold start In Zebra method, at first, due to that we do not have trained filters, we have to process the query and the streaming data like brute-force pipelines, which abandon data randomly for overload scenarios. In the first few pipelines, the method will process the query based on a brute-force pipeline pattern with a sample rate that represents the Load Shedding module, and the system is saving the query answer to local at the same time. We can see that the query answer are the documents with labels ready to use. Obviously, we may not train a filter based on only one pipeline data owing to that the data is not enough to train an efficient classifier. When a new window of streaming data comes, the system will check the data size threshold for training. If the training data stored locally is larger than the threshold we set in the new pipeline, the system will use the local data with labels to train filters online. Generally, we set the threshold as 10 thousand for text classifiers. To guarantee the filter can classify the two categories well, we need to ensure that the training data has label $+1$ and label -1 enough both. The $+1$ label data which satisfies the query can also be stated as *Positive* data, and on the contrary, -1 label data is *Negative* data. The data size needs to satisfy the following equation.

$$\begin{cases} TrainingDataSize \geq threshold_{all} \\ PositiveDataSize \geq threshold_{pos} \\ NegativeDataSize \geq threshold_{neg} \end{cases} \tag{11}$$

Training filters Since we have the above training data, we can train the classifiers as filters, and the specific classifier types have been introduced in Section 3. The training stage details can be found in Section 3.5. Once we have trained the filters, we can use them to optimize the query pipelines.

4.3 Parameter search

Algorithm 1 Exhaustive search.

Input: The answer time, t_{bound}; The input streaming documents data, D; Current data window size, $Batch$; Accuracy-Data reduction rate curve of the filter m, $AccReduction_m$; the extra time reserved for extra modules in progressive mode, t_{extra};

Output: Sample rate of the pipeline, x_p; Filter accuracy of the pipeline, a_p;

Scan the input data D and get the unit udf cost time as t_{udf}
Initialize t_{extra}, Acc_{max}
if Mode == 'Brute-force' **then**
    $load_{doc} \leftarrow T_L / t_{udf}$
    $x_p \leftarrow Min(1, load_{doc}/Batch)$
    $a_p \leftarrow Null$
else if Mode == 'Progressive' **then**
    for $Acc \in [0, 1]$ **do**
        for $s \in [0, 1]$ **do**
            for $m \in \{Perceptron, NaiveBayes, DNN\}$ **do**
                $Acc_t \leftarrow Acc \times s$
                $num_{filter} \leftarrow s \times Batch$
                $num_{udf} \leftarrow (1.0 - AccReduction_m[Acc]) \times s \times Batch$
                $t_{cost} \leftarrow num_{udf} \times t_{udf} + num_{filter} \times t_{filter} + t_{extra}$
                if $t_{cost} \leq t_{bound}$ **and** $Acc_{max} \leq Acc_t$ **then**
                    $a_f \leftarrow Acc$
                    $x_p \leftarrow s$
                    $Acc_{max} \leftarrow Acc_t$
                end if
            end for
        end for
    end for
end if
return x_p, a_p;

Thus far, we have trained the filters applied to progressive pipelines, but we don't know how to determine the parameters in progressive pipelines to make the response result of the query best. The main parameters in the progressive pipelines are sample rate x, accuracy a_f of the filter, and the specific filter to apply. The premise of the answer is that we need to satisfy the cost time condition in Section 2, and based on this, we select the best parameter configuration by maximizing the whole pipeline accuracy:

$$[x,\ a_f,\ m_{filter}] = arg\ max\ Accuracy \qquad (Cost\ <= t_{bound}) \tag{12}$$

Notice that the domains of sample rate x and filter accuracy a_f are both [0,1], and there are only three possible filter types. We propose Exhausted Search method to find the best configuration of progressive pipelines.

The domain [0,1] is a continuous space, and if we try to enumerate the sample rate and accuracy of the filter, we can divide the domain into discrete values. For example, [0,1] can be divided into 100 pieces with a step size equal to 0.01. In the end, if we divide the domain of

sample rate, filter accuracy, and filter kinds into N parts, M parts, and K parts, respectively, the complexity of Exhaustive Search is $O(N * M * K)$ as cubic level. Before Exhaustive Search, we first need to obtain the unit cost time of UDF, and it is an average time for the current streaming text data. More details about how we get t_{udf} will be introduced in Section 5. We use a linear fitting method to get this value. In Exhaustive Search, we search the parameters in two modes separately. In brute-force pipelines, the method calculates how much load the system can afford based on t_{udf}, then gets the sample rate x_p; in progressive pipelines, the method enumerates the two parameters and finds a max value of the whole pipeline while satisfying the time cost constraint. In progressive pipelines, there are lots of different operations or functions, so we add an additional cost time t_{extra} to the pipeline cost.

The pseudocode in Algorithm 1 covers the details for further reference.

4.4 Overload scenario with dynamic distribution

The basic data steam overload scenario is based on the assumption that in the close time interval, the workload of input data is similar and the distribution of streaming data is basically constant. In this scenario, we can use the above Zebra method to optimize the query pipeline. However, in real application scenarios, the phenomenon of dynamic data distribution can still occur, and the previous data window distribution is quite different from the latter one.

When the distribution of input data changes, the calculated accuracy and reduction rate curves of filters are no longer applicable, and the searched parameters before are not suitable for the real-time query. Therefore, we have to recalculate the curve of filters, and we find that the curves of one filter on the same distribution almost match each other. Based on this, we propose Zebra-Dynamic to optimize the dynamic overload scenario.

Zebra-dynamic Zebra-Dynamic is totally based on original Zebra method and adapts the parameter curves of filters to the real-time distribution. Because we can not directly obtain the real-time data window distribution, Zebra-Dynamic uses a sampling-based method to approximate the distribution. Specifically, Zebra-Dynamic samples a small set of streaming data and inputs the data set into the UDF module directly. After that, the small set of data has labels, and we can use this distribution as a microcosm of the entire data window. Next, Zebra-Dynamic adjusts the distribution of the validation set to be consistent with the small set. In the end, we recalculate the curves of filters on the new validation set as introduced in Section 3.5. The next steps of Zebra-Dynamic are the same as original Zebra. Figure 5 shows the overview of Zebra-Dynamic.

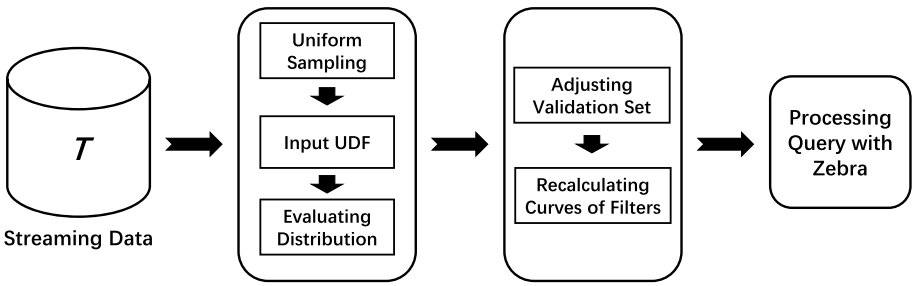

Fig. 5 Zebra-dynamic method overview

5 Experiments

5.1 Experiments setup

Environment All the experiments and our method are implemented with C++ and Python3.7. Our programs are all running on a Dell PowerEdge R540, and the operating system is Ubuntu 18.04 LTS with a 2.1GHz Intel Xeon Silver CPU, 160GB RAM and 7.3 TB HDD.

Datasets We perform our experiments on three real-world datasets, Amazon Review Data, IMDB data, and Twitter dataset(Tweets). We input the documents into the system as a data stream. If the dataset doesn't have a timestamps attribute, we divide the documents into data windows with different sizes; if the dataset has a timestamps attribute, we divide the documents into data windows according to timestamps, such as one data window containing the documents in one minute. In the experiments in Section 5.3, we control the selectivity of the input data to be almost stable. In the dynamic overload experiment, input data distribution changes over time.

Performance metrics & baseline In our experiments, we focus on two metrics: the accuracy and the response latency of the query in one pipeline. Accuracy of the query can also be seen as the recall for the query owing that the accuracy is the ratio of the positive data number in the query answer to the total positive data number in the current data window. Finally, the accuracy is calculated in the following equation, and S_{ans} means the set of query answers in one pipeline. The latency of the query is the response time of processing the real-time data window, and we hope the latency can be limited into t_{bound} given by users. In the experiment stage, we principally focus on the accuracy metric, which is the primary optimization target that our method improves. The baseline we mainly compare is applying the load shedding technique only, which means that the baseline only contains brute-force pipelines in experiments. To calculate the accuracy of pipelines, in the end, we run related text classification UDF in advance to label the data.

$$Accuracy = \frac{\left|\left\{x \in S_{ans} \,:\, label(x) = true\right\}\right|}{|label(x) = true|} \tag{13}$$

5.2 Query case & determine UDF time

There are lots of text classification tasks, like sentiment analysis, named entity recognition, and information extraction, and we focus on the UDF related to text classification.

For every streaming data window, we need to first evaluate the unit cost time of the real-time UDF in a query because, in Parameter Search module, we compute the values based on this. It's easy to see that the UDF cost time of a single document is related to the document's token number because the size of feature input to the model is proportional to the token number. Therefore, we propose four examples of queries as SQL patterns, and the t_{bound} of these queries are 60 seconds in default. The first SQL, which includes sentiment analysis UDF and classifies the text into positive and negative, is as follows:

```
Query1 :
SELECT * FROM TwitterSet/AmazonReviewText/IMDB
WHERE is PositiveUDF(document) = True
```

This SQL is about finding the positive documents in TwitterSet streaming data, and we implement the UDF through AllenNLP[2] library. The actual model of the sentiment analysis in Allen NLP is GLoVe-LSTM which is a typical RNN model.

```
Query2 :
SELECT * FROM TwitterSet/AmazonReviewText/IMDB
WHERE has PersonUDF(sentence) = True
Query3 :
SELECT * FROM TwitterSet/AmazonReviewText/IMDB
WHERE has LocationUDF(sentence) = True
Query4 :
SELECT * FROM TwitterSet/AmazonReviewText/IMDB
WHERE has OrganizationUDF(sentence) = True
```

The second SQL is about finding the text which has a personal name, and the text classification UDF is a named entity recognition task. We implement it also through Allen NLP library. According to the official Allen NLP guide, it uses a Gated Recurrent Unit (GRU) [7] character encoder as well as a GRU phrase encoder, and it starts with pretrained GloVe vectors for its token embeddings. The rest queries are similar to the previous two but different text classification tasks. One is to find the text, including location name, and the other is aimed to find text containing organization. The queries we present above all have high selectivity, and these text classification predicates are popular in common text data.

How to determine UDF time In Table 2, we list some mainstream model structures of text UDF tasks and their time complexity related to text token number and depth of the neural network. In [39], there are theoretical analyses about it, and finally, the results are given. Based on this, for example, as for the first SQL statement above, we know that the sentiment analysis UDF was implemented with the LSTM [13] model, and we can inquire about the time complexity of UDF from Table 2 for the LSTM model is a typical RNN [31] structure. For the sentiment analysis task, the UDF cost time of a single document has a linear relationship with the text token number. We can assume UDF cost time $u = k * n + b$, and n represents the text token number. We try to use the linear fitting method to solve the parameters k and b in the equation. We select the inference time of thousands of documents on the current UDF task, like sentiment analysis, and use these data to fit linear relationships in advance. As shown in Figure 6, we can find that the relationship between two variables basically satisfies the linearity, and although obviously, there will be some errors, we still can use this equation to calculate UDF time approximately. The experiment about sentiment analysis UDF cost time was realized on about 30 thousand documents. Finally, other UDF cost times can be deduced by analogy. In experiments, the cost of UDF for single documents ranges from 0.02s to 0.5s, and it is consistent with the data about UDF cost mentioned in [11].

[2] **AllenNLP** library: https://allennlp.org/

Table 2 Complexity on text models

Model Structure	Inference Complexity
Neural Network With Attention	$O(n^2 \cdot d)$
Recurrent Neural Network	$O(n * d^2)$
Convolutional Neural	$O(n * d^2)$

n represents the document token length and d represents the depth of the neural network

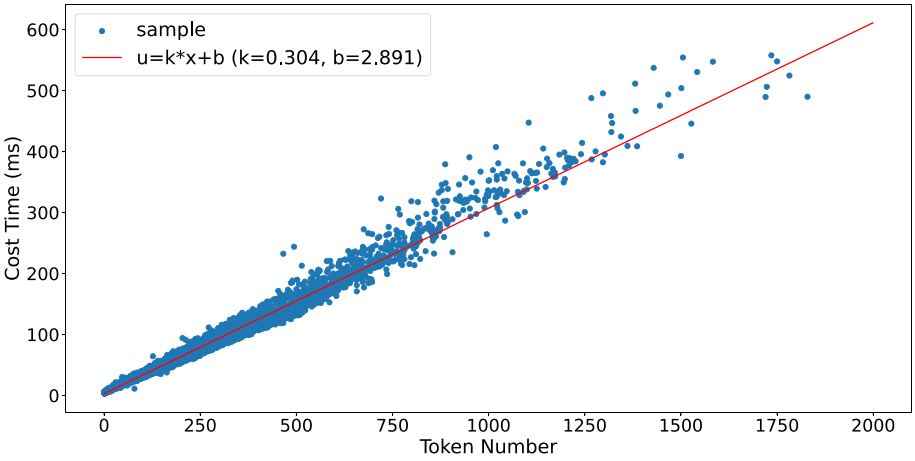

Fig. 6 The relationship between UDF cost time and document token number of Sentiment Analysis

5.3 Accuracy evaluation of overload experiment

In these experiments, we set the training data size to ten thousand, positive data size to four thousand, and negative data size to four thousand. These hyperparameters are mentioned in Section 4.3, and the target of setting in this way is to ensure that the classification effect of the filters is good enough.

Incremental Overload Experiment In this experiment, we compare the performance between Zebra method and baseline on the Query in Section 5.2 with the $t_{bound} = 60s$. Next, we divide the three datasets into lots of batches to stimulate the stream scenario, and one data batch can be seen as a data window with window size, such as recent 120 seconds text data input to Twitter, and we respond to the query in the limit time t_{bound}. In this scenario, the given queries are all long-running, and the rate of the positive samples in one data window is about 40% to 60%. We mainly focus on the overload situations that our method performs better than the baseline. We propose a measure of the overload ratio of the input data window, as shown in the equation below:

$$Overload_{rate} = \frac{N_d - W}{W} \tag{14}$$

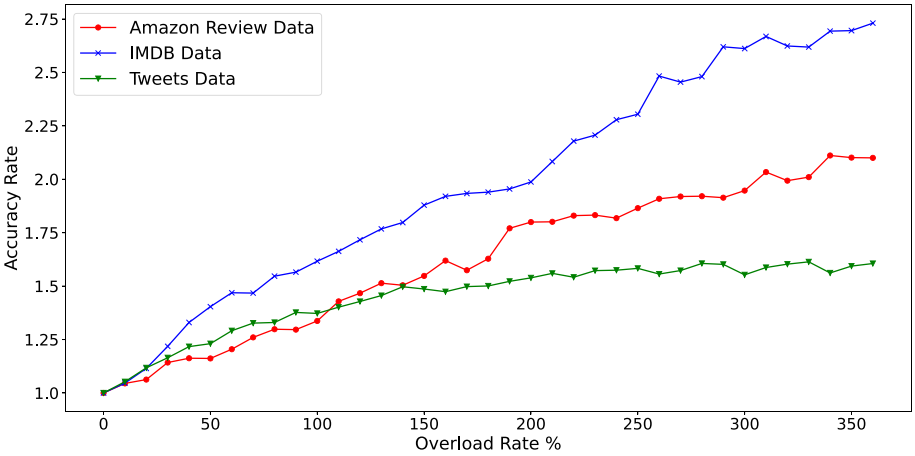

Fig. 7 Accuracy rate of the Progressive pipeline to the Brute-force pipeline with incremental overload percent for Query1

N_d means the documents number needs to be processed for the real-time data window, and W means the number of documents that can be processed in t_{bound}. So the overload rate is the rate of extra documents which can not be processed normally. We fix the t_{bound} in our experiments because for a fixed-size data window, changing the response time can be seen as adjusting the overload in disguise. Different text classification tasks have a different load for data, and diverse datasets also impact the unit processing capacity that the average documents length change with datasets. For example, in our experiments, the processing capability of the sentiment analysis classification task is about 4k to 6k documents per minute.

We firstly design an experiment with the effect of incremental overload rate, and we make the incremental step equal to 10%, which means each data window has 10% more data than the previous window. Then, we do experiments on three different datasets and use the accuracy rate to represent the improvement rate of Zebra to baseline. Accuracy rate equals Zebra method accuracy divided by baseline accuracy. In Figure 7, we can see that on all three datasets, as the overload percent increases, the improvement of Zebra becomes larger, and on IMDB Data, Zebra performs best and can achieve more than 2.5x of the baseline. In the other two datasets, the improvement can be up to 1.5x and 2.0x, respectively. Also, the results show that as the overload percent increases, the improvement tends to stabilize, and the accuracy of Zebra method drops slower than the baseline. For that, if the overload is too high, the accuracy of the answer will be very low, and it doesn't make sense in actual sense. In the end, we set the upper bound of our experiments as about 400% overload.

Performance of fixed overload rate In this experiment, we control the overload rate of input data consistently. Figure 8 shows the results of running Query 1 on 10%, 20%, and 40% overload Tweets data windows. It shows that progressive pipelines perform all the better than brute-force pipelines, and with the incremental overload percent, the accuracy of the progressive pipelines drops slower than the brute-force pipelines. In 20% and 40% overload experiments, the accuracy can be improved by about 10 percent.

Figure 9 shows the results of Query 2 on Tweets. Comparing Figures 7 and 8, we can see that Zebra performs better on Query 1, and the improvement on Query 1 is more stable

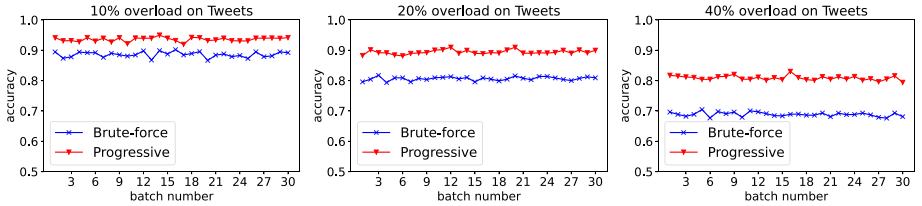

Fig. 8 Accuracy of the Progressive pipeline and the Brute-force pipeline on 10%, 20% and 40% Tweets overload data windows for Query 1

and more significant than Query 2. The difference in the improvement effect is mainly related to the complexity of the text classification task itself.

For that, the results of other query experiments are similar to Query 1 and Query 2, and we present the statistical results. Figure 10 shows the average accuracy of two methods on fixed overload data windows. We run all the queries on 30 data windows and calculate the average accuracy. It shows that among different queries, Zebra method performs better for Query 1 and Query 3, and accuracy for Query 1 and Query 3 is improved more than for Query 2 and Query 4.

Eventually, in all the experiments above, Zebra method can satisfy the latency condition t_{bound} based on the method of evaluating UDF cost in 5.2, and to prevent the pipeline from exceeding the time limit, we reserve a bit more time in t_{extra}. The actual latency of progressive ranges from 57 seconds to 59 seconds when the $t_{bound} = 60s$.

5.4 Extra cost time discussion

In this section, we evaluate the extra cost of our method, including the training stage and inference stage of the filters, the parameter search module, and some preprocessing stages.

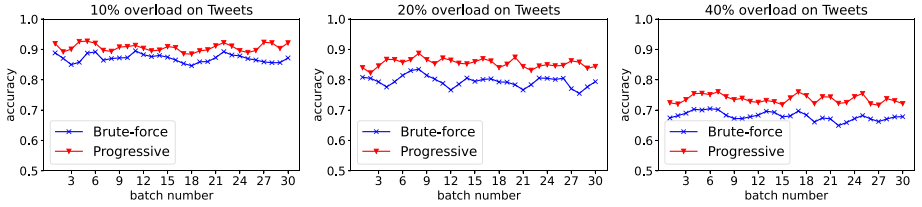

Fig. 9 Accuracy of the Progressive pipeline and the Brute-force pipeline on 10%, 20% and 40% Tweets overload data windows for Query 2

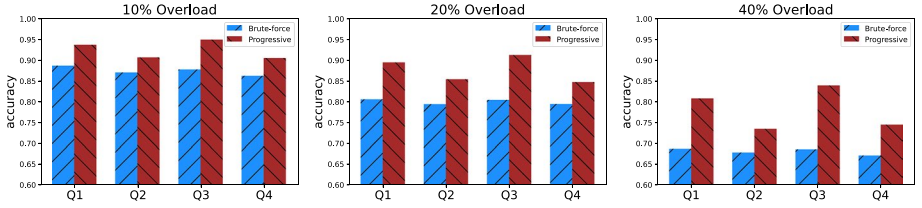

Fig. 10 Average accuracy of the Progressive pipeline and the Brute-force pipeline on 10%, 20% and 40% Tweets overload data windows for Query 1-4

Table 3 Training time and inference time of the classifiers in experiments

Classifier	Training Time(s)	Inference Time(ms)
Perceptron	0.31488	0.372
Naive Bayes Classifier	0.03712	0.840
DNN	4.17079	10.367

Table 4 Other extra cost time in our experiments

Other Modules	Average Cost Time(s)
Dimension Reduction(Feature Hashing)	0.09270
Scanning Documents	1.19257
Parameter Search	0.72236

In our experiments above, we set the inference time and training time into the t_{extra} with t_{udf} for text classification UDF processing. We list the cost time of filters we trained in Table 3, and we train the filters on 8000 documents and test on 2000 documents. We can see that the overhead of classifiers is very low compared to UDF cost time and even the training time of DNN classifier is less than 10% of 60 seconds. In experiments, we use the t_{extra} to contain this overhead, and only the first progressive pipeline contains the training stage, which affects the accuracy of the query a bit.

Besides the cost of filters in our pipeline, we still have the cost of other modules. We list the cost in Table 4, and these sections have a higher magnitude compared with the inference time of filters. So the total extra cost time is less than 2 seconds generally. In experiments, we set the t_{extra} related to the data window size for that the scanning stage occupies a large part of the time.

5.5 Sample rate discussion & parameter search optimization

Why do the progressive pipelines need the Load Shedding module? In the progressive pipeline, the Load Shedding module is the previous module of the filtering module, and the sample rate is the core part of Load Shedding. At present, we have filters, so why do we still need load shedding to sample data before the filter stage? Intuitively, we can apply filters to the pipeline directly and search the parameters to make the filtered data can respond in answer time. Although we can optimize the pipeline based on such a direct method, there will be a better solution theoretically based on the parameter search method. We can see the example in Table 5. It's a real example coming from the filters we trained, and the progressive pipeline with the sample rate parameter can achieve higher accuracy than the pipeline directly applied filters. The premise of comparison between the two methods is

Table 5 Comparison the progressive method with only applying filters method

Method	Sample Rate	Filter Accuracy	Pipeline Accuracy	Reduction Rate
Progressive	0.96	0.74	0.96*0.74=0.7104	0.61
Only Filters	1.0	0.7	1.0*0.7=0.7	0.63

that the whole latency of pipelines is almost the same as the other, which means they all meet the constraint condition of (2) and have the same data reduction rate. In this example, a method with only filters means there is no sample stage before the filtering module, and the sample rate can be seen as 1.0. In the meanwhile, the accuracy of the whole pipeline equals to sample rate times the accuracy of filters, and in this case, the progressive pipeline can achieve 0.7104 accuracy, which is a bit better. In the actual specific experiment stage, if only the accuracy and reduction rate curve has a small error on the real-time data, the progressive method will perform better.

Optimization for parameter search In the experiments above, we also found that the sample rate value from Parameter Search is always very high, and the method of applying filters directly is a proper solution for our target. Based on this condition, we can optimize Exhaustive Search, and we propose a new version called Advanced Search. We mainly accelerate the search method for progressive mode. The details of the optimization algorithm can be found in Algorithm 2. We fix the latency firstly, and we utilize the latency to calculate the data reduction rate directly. Then we can obtain the accuracy of filters by Binary-Search method as the order in $AccReduction_m$ map is sorted. The equation of pseudocode line 11 is derived from (15). Compared with Exhausted Search, the complexity of Advanced Search is $O(N * K * log(M))$, where N represents the size of sample rate values, M represents the domain size of filters accuracy, and K represents the number of filter category. Besides, based on experiment experience, we limit the lower bound of sample rate value. In the end, the time complexity is much lower than Exhausted Search algorithm.

Algorithm 2 Advanced search.

Input: Same as Exhaustive Search
Output: Same as Exhaustive Search

1: Preprocessing stage and method for 'Brute-force' mode are same as Exhausted Search.
2: Initialize L_B (Lower bound of sample rate value)
3: **if** Mode == 'Progressive' **then**
4: $r_f = 1 - ((t_{bound} - t_{extra})/t_{udf})/Batch$
5: **for** $m \in \{Perceptron, NaiveBayes, DNN\}$ **do**
6: $a_{tmp} \leftarrow$ Binary-Search$(AccReduction_m, r_f)$
7: **if** $Acc_{max} \leq a_{tmp}$ **then**
8: $a_f \leftarrow a_{tmp}, x_p \leftarrow 1.0, Acc_{max} \leftarrow a_{tmp}$
9: **end if**
10: **for** $s \in [L_B, 1]$ **do**
11: $r \leftarrow (s + r_f - 1)/s$
12: $a \leftarrow$Binary-Search$(AccReduction_m, r)$
13: **if** $Acc_{max} \leq s \times a$ **then**
14: $a_f \leftarrow a, x_p \leftarrow s, Acc_{max} \leftarrow s \times a$
15: **end if**
16: **end for**
17: **end for**
18: **end if**
19: **return** x_p, a_p;

Table 6 Average cost time of two search methods

Search Method	Average Cost Time(ms)
Exhausted Search	741.40
Advanced Search	58.880

$$s \times (1 - r) = 1 - r_f \tag{15}$$

The actual results of different Parameter Search algorithms are shown in Table 6, and we test the two algorithms on different datasets. Obviously, this module makes no reference to queries or datasets. In experiments, Advanced Search accelerates the Parameter Search module about 100x to 150x compared with naive Exhausted Search, which saves more time for the next module to process more positive samples, and it also improves the throughput of the answer a little.

5.6 Evaluation of dynamic overload experiment

We evaluate Zebra-Dynamic in this experiment. The difference from the experiments in Section 5.3 is that the distribution of input data is dynamic. In the actual implementation, we set a threshold for the difference in distribution, and if the distribution difference between adjacent data stream windows exceeds the threshold, we apply the Zebra-Dynamic method to optimize the query pipeline.

We process the input text data into dynamic distribution, and the proportion of positive data in adjacent data stream windows has a certain difference. Firstly, we fix the overload rate of each data stream window and evaluate methods on 100 windows. This experiment fixes the overload rate at 40% and evaluates methods on the Tweets dataset with Query 1-4. Figure 11 shows the results, and we use the average accuracy of 100 data stream windows as the metric. We can find that Zebra-Dynamic method performs better than original

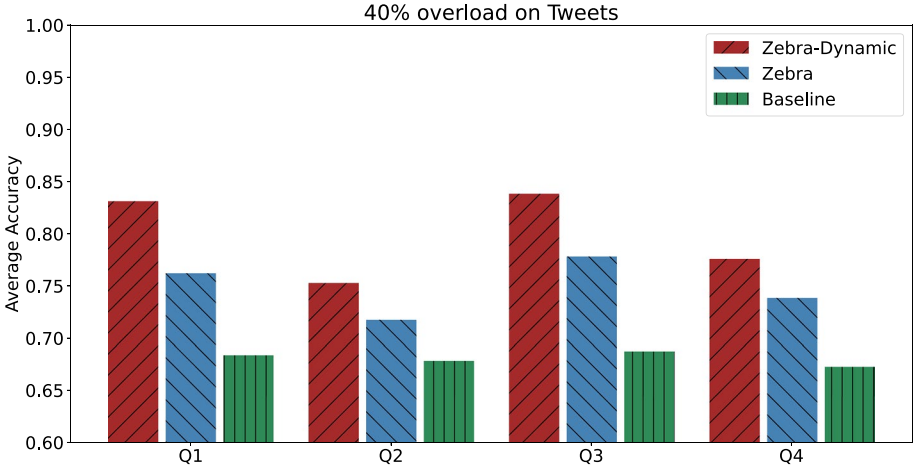

Fig. 11 Average accuracy of three methods on 100 dynamic data stream windows

Table 7 Average Overhead Time for Query 1-4

Extra Module Overhead (ms)	Query 1	Query 2	Query 3	Query 4
Sampling Stage Overhead	1287.2	3216.4	1635.7	2183.5
Recalculating Curves Overhead	145.79	133.62	144.68	152.72

Zebra method in the dynamic distribution scenario. The reason is that the results from Zebra-Dynamic parameter search are more accurate than Zebra method. In the meantime, although the accuracy of Zebra method drops, Zebra still outperforms the baseline due to the filtering module. In Figure 11, we can find that Zebra-Dynamic can improve the accuracy by 4 to 7 percentage points compared with Zebra method. Consistent with the experiments in Section 5.3, the performance improvement is better on Query 1 and Query 3.

In Tables 7 and 8, we list some extra details of the experiment in Figure 11. Table 7 shows the average overhead time of the additional modules in Zebra-Dynamic on Query 1-4. For different queries, the cost time in the sampling stage varies, and if the inference time of query UDF is high, the overhead of this stage will also become high.

Due to that, the accuracy and reduction rate curves of Zebra method are inaccurate, many query responses in Zebra method will fail, and only 64.3% of queries are completed in time. When calculating the average accuracy of methods, we only consider the completed pipelines for Zebra method. Meanwhile, Table 8 shows that the average response time of Zebra method is lower than the other two methods. This is caused by the inaccuracy curves of Zebra method's filters. The reason why the response rate of the other two methods does not reach 100% is due to the experimental error.

Finally, we fix the overload rate of data stream windows and run all methods on Query 1 with three different datasets. Figure 12 shows the results. We find that Zebra-Dynamic method achieves the best improvement on the IMDB dataset. Compared with Zebra method and baseline, it can improve the accuracy by 20% and 33% respectively.

6 Related work

There are several different research areas related to our work, and we discuss them in depth below:

Load shedding in streaming scenario Load Shedding is a traditional technique in streaming data scenarios, and owing to that, latency is the most significant requirement [38]. When input data exceed system capacity, DSMSs will become overloaded and subsequently employ the load shedding technique to satisfy quality requirements [3, 35]. In our query pattern, the query is a long-running query. Compared with merely

Table 8 Response completion of three methods in overload scenario with dynamic data distribution

Metrics	Zebra-Dynamic	Zebra	Baseline
Percentage of Completed Queries in t_{bound}	97.5%	64.3%	98.6%
Average Response Time (seconds)	58.6	54.4	59.6

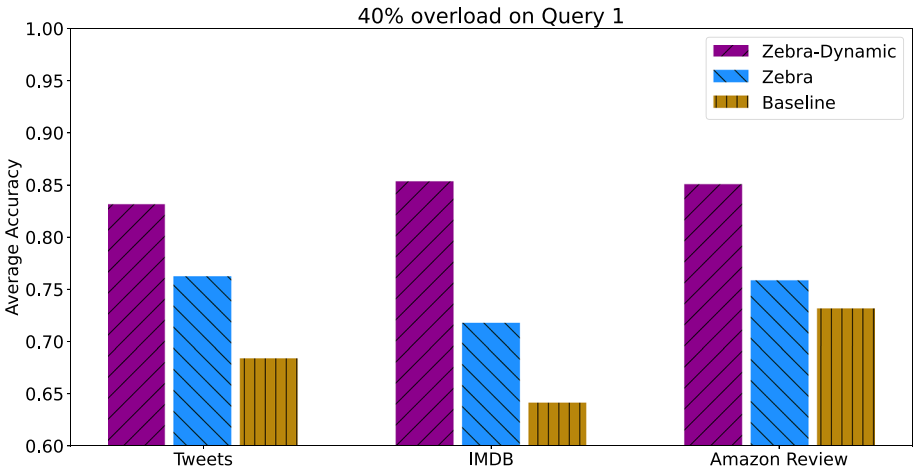

Fig. 12 Average accuracy of three methods on 100 dynamic data stream windows of three datasets

applying the load shedding technique, the Zebra method also contains the load shedding module, and for that, load shedding only abandons data randomly; our method filters data semantically, which results in high accuracy for the query while meeting the latency demand.

UDF optimization In the database scenario, lots of researchers have been working on optimizing queries with UDFs for decades [12]. Meanwhile, with the rapid development of artificial intelligence, there are many UDFs about AI tasks, like CV or NLP, and users have more demands on such queries. The traditional optimization techniques, such as operator reordering [14, 15], sub-query optimization [32], and index selection [6, 16], can not be applied or perform well in this scenario, considering that these UDFs have opaque semantic information. Other recent researches about UDF optimization are to create new predicates for pre-filtering using machine learning methods [1, 6, 29] and do not optimize the query in a streaming scenario. The Zebra method trains the filter online and searches the parameters to make the filter available for the real-time data batch.

Query optimization with machine learning AI for DB has been a hot research topic in recent years. Numerous artificial intelligence technologies have emerged, such as deep learning [26] and reinforcement learning [34]. These popular methods have been widely used in query optimization. Li et al. [27] summarize the techniques of AI for DB, and in query optimization, learning-based techniques have been applied in many areas, including end-to-end optimizer [30, 41] for SQL queries using neural networks, cardinality estimation [20], selectivity estimation [23] and join order selection [21, 37] with deep reinforcement learning and Monte-Carlo tree search [8]. These research works perform better than traditional optimization methods using state-of-art machine learning techniques and address the hard problems in database optimization. Our work is designed for streaming scenario queries with specific UDF for text tasks.

7 Conclusion

This paper studies a new scenario of query optimization for nowadays hot tasks. In this paper, we propose a new query pattern with UDF for text classification and the bounded time, satisfying the demand for the stream scenario. We apply the probabilistic filters techniques to select more potential data that users want and make the accuracy of answers higher within the limited time to deal with overload scenarios. Next, we present a progressive training style to train several classifiers as filters during the method running phases and use an advanced search method to find the best configuration for the real-time query and data. For better results of parameter search and stable query latency, we also propose a method based on linear fitting to predict the cost time of input text data. We implement these techniques into the method called Zebra. Experiment results show that Zebra can achieve up to 1.5x-2.5x accuracy higher than the brute-force fashion method when the data overloads the system. Besides, we propose Zebra-Dynamic method to deal with the dynamic distribution scenario, and experimental results show that Zebra-Dynamic method performs better than Zebra method and the baseline in this scenario.

In future work, we aim to study how to optimize Zebra-Dynamic method in the dynamic distribution scenario. Besides, we consider integrating Zebra method into existing streaming data systems.

Acknowledgements This work was mainly supported by National Natural Science Foundation of China under Grant Nos. 61732004, 62072113. This work was also supported by the Research Projects of Zhejiang Lab (No. 2021PE0AC01).

We are also grateful for the insightful comments offered by the anonymous reviewers. The generosity and expertise of one and all have improved this study in innumerable ways and saved us from many errors.

Declarations

Competing interests The authors declare that they have no competing interests.

References

1. Anderson, M.R., Cafarella, M.J., Ros, G., et al: Physical representation-based predicate optimization for a visual analytics database. In: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, pp 1466–1477. IEEE (2019)
2. Armbrust, M., Xin, R.S., Lian, C., et al.: Spark SQL: relational data processing in spark. In: Sellis, T.K., Davidson, S.B., Ives, Z.G. (eds.) Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, pp 1383–1394. ACM (2015)
3. Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams. In: Özsoyoglu, Z.M., Zdonik, S.B. (eds.) Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, pp 350–361. IEEE Computer Society, Boston, MA, USA (2004)
4. Bastani, F., He, S., Balasingam, A., et al: MIRIS: fast object track queries in video. In: Maier, D., Pottinger, R., Doan, A., et al. (eds.) Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, pp 1907–1921. ACM (2020)
5. Chaiken, R., Jenkins, B., Larson, P., et al.: SCOPE: easy and efficient parallel processing of massive data sets. Proc. VLDB Endow. **1**(2), 1265–1276 (2008)
6. Chaudhuri, S., Narasayya, V.R., Sarawagi, S.: Efficient evaluation of queries with mining predicates. In: Agrawal, R., Dittrich, K.R. (eds.) Proceedings of the 18th International Conference on Data

Engineering, San Jose, CA, USA, February 26 - March 1, 2002, pp 529–540. IEEE Computer Society (2002)
7. Chung, J., Gülçehre, Ç, Cho, K., et al.: Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv:1412.3555(2014)
8. Coulom, R.: Efficient selectivity and backup operators in monte-carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers, Lecture Notes in Computer Science, vol. 4630, pp 72–83. Springer (2006)
9. Frank, E., Bouckaert, R.R.: Naive bayes for text classification with unbalanced classes. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) Knowledge Discovery in Databases: PKDD 2006, 10th European Conference on Principles and Practice of Knowledge Discovery in Databases, Berlin, Germany, September 18-22, 2006, Proceedings, Lecture Notes in Computer Science, vol. 4213, pp 503–510. Springer (2006)
10. Gallant, S.I.: Perceptron-based learning algorithms. IEEE Trans. Neural Netw. **1**(2), 179–191 (1990)
11. He, W., Anderson, M.R., Strome, M., et al.: A method for optimizing opaque filter queries. In: Maier, D., Pottinger, R., Doan, A., et al. (eds.) Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, pp 1257–1272. ACM (2020)
12. Hellerstein, J.M., Stonebraker, M.: Predicate migration: Optimizing queries with expensive predicates. In: Buneman, P., Jajodia, S. (eds.) Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993, pp 267–276. ACM Press (1993)
13. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
14. Hueske, F., Peters, M., Sax, M., et al.: Opening the black boxes in data flow optimization. Proc. VLDB Endow. **5**(11), 1256–1267 (2012)
15. Hueske, F., Peters, M., Krettek, A., et al.: Peeking into the optimization of data flow programs with mapreduce-style udfs. In: Jensen, C.S., Jermaine, C.M., Zhou, X. (eds.) 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013, pp 1292–1295. IEEE Computer Society (2013)
16. Jahani, E., Cafarella, M.J., Ré, C.: Automatic optimization for mapreduce programs. Proc. VLDB Endow. **4**(6), 385–396 (2011)
17. Jolliffe, I.T.: Principal Component Analysis. Springer Series in Statistics, Springer (1986)
18. Kang, D., Emmons, J., Abuzaid, F., et al.: Noscope: Optimizing deep cnn-based queries over video streams at scale. Proc. VLDB Endow. **10** (11), 1586–1597 (2017)
19. Kang, D., Bailis, P., Zaharia, M.: Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. Proc. VLDB Endow. **13**(4), 533–546 (2019)
20. Kipf, A, Kipf, T., Radke, B, et al: Learned cardinalities: Estimating correlated joins with deep learning. In: 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019. Online Proceedings. www.cidrdb.org (2019)
21. Krishnan, S., Yang, Z., Goldberg, K., et al.: Learning to optimize join queries with deep reinforcement learning. arXiv:1808.03196 (2018)
22. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Bartlett, P.L., Pereira, F.C.N., Burges, C.J.C. (eds.) Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States, pp 1106–1114 (2012)
23. Lakshmi, M.S., Zhou, S.: Selectivity estimation in extensible databases - A neural network approach. In: Gupta, A., Shmueli, O., Widom, J. (eds.) VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, pp 623–627. Morgan Kaufmann, New York City, New York, USA (1998)
24. LeCun, Y., Boser, B.E., Denker, J.S., et al.: Handwritten digit recognition with a back-propagation network. In: Touretzky, D.S. (ed.) Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989], pp 396–404. Morgan Kaufmann (1989)
25. LeCun, Y., Haffner, P., Bottou, L., et al.: Object recognition with gradient-based learning. In: Forsyth, D.A., Mundy, J.L., Gesù, V.D., et al. (eds.) Shape, Contour and Grouping in Computer Vision, Lecture Notes in Computer Science, vol. 1681, p 319. Springer (1999)
26. LeCun, Y., Bengio, Y., Hinton, G.E.: Deep learning. Nature **521**(7553), 436–444 (2015)
27. Li, G., Zhou, X., Cao, L.: AI meets database: AI4DB and DB4AI. In: Li, G., Li, Z., Idreos, S., et al. (eds.) SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pp 2859–2866. ACM (2021)

28. Lu, Y., Chowdhery, A., Kandula, S.: Optasia: A relational platform for efficient large-scale video analytics. In: Aguilera, M.K., Cooper, B., Diao, Y. (eds.) Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016, pp 57–70. ACM (2016)
29. Lu, Y., Chowdhery, A., Kandula, S., et al.: Accelerating machine learning inference with probabilistic predicates. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, pp 1493–1508. ACM (2018)
30. Marcus, R.C., Negi, P., Mao, H., et al.: Neo: A learned query optimizer. Proc. VLDB Endow. **12**(11), 1705–1718 (2019)
31. Mikolov, T., Karafiát, M., Burget, L., et al.: Recurrent neural network based language model. In: Kobayashi, T., Hirose, K., Nakamura, S. (eds.) INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010, pp 1045–1048. ISCA (2010)
32. Ramachandra, K., Park, K., Emani, K.V., et al.: Froid: Optimization of imperative programs in a relational database. Proc. VLDB Endow. **11**(4), 432–444 (2017)
33. Singh, G., Kumar, B., Gaur, L., et al.: Comparison between multinomial and bernoulli naïve bayes for text classification. In: 2019 International Conference on Automation, Computational and Technology Management (ICACTM), pp 593–596. IEEE (2019)
34. Sutton, R.S., Barto, A.G. Reinforcement learning - an introduction. Adaptive computation and machine learning. MIT Press (1998)
35. Tatbul, N., Çetintemel, U., Zdonik, S.B., et al.: Load shedding in a data stream manager. In: Freytag, J.C., Lockemann, P.C., Abiteboul, S., et al. (eds.) Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003, pp 309–320. Morgan Kaufmann (2003)
36. Thusoo, A., Sarma, J.S., Jain, N., et al.: Hive - A warehousing solution over a map-reduce framework. Proc. VLDB Endow. **2**(2), 1626–1629 (2009)
37. Trummer, I., Wang, J., Maram, D., et al.: Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In: Boncz, P.A., Manegold, S., Ailamaki, A., et al. (eds.) Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, pp 1153–1170. ACM (2019)
38. Tu, Y., Liu, S., Prabhakar, S., et al.: Load shedding in stream databases: A control-based approach. In: Dayal, U., Whang, K., Lomet, D.B., et al. (eds.) Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006, pp 787–798. ACM (2006)
39. Vaswani, A., Bengio, S., Brevdo, E., et al.: Tensor2tensor for neural machine translation. arXiv:1803.07416 (2018)
40. Weinberger, K.Q., Dasgupta, A., Langford, J., et al.: Feature hashing for large scale multitask learning. In: Danyluk, A.P., Bottou, L., Littman, M.L. (eds.) Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009, ACM International Conference Proceeding Series, vol. 382, pp 1113–1120. ACM (2009)
41. Wu, C., Jindal, A., Amizadeh, S., et al.: Towards a learning optimizer for shared clouds. Proc. VLDB Endow. **12**(3), 210–222 (2018)
42. Xarchakos, I., Koudas, N.: SVQ: streaming video queries. In: Boncz, P.A., Manegold, S., Ailamaki, A., et al. (eds.) Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, pp 2013–2016. ACM (2019)

Authors and Affiliations

Tianhuan Yu[1] · Zhenying He[2] · Zhihui Yang[3] · Fei Ye[2] · Yuankai Fan[2] · Yinan Jing[2] · Kai Zhang[2] · X. Sean Wang[2]

Tianhuan Yu
thyu19@fudan.edu.cn

Zhihui Yang
zhyang14@zhejianglab.com

Fei Ye
fye21@m.fudan.edu.cn

Yuankai Fan
ykfan19@fuan.edu.cn

Yinan Jing
jingyn@fudan.edu.cn

Kai Zhang
zhangk@fudan.edu.cn

X. Sean Wang
xywangcs@fudan.edu.cn

[1] School of Software, Fudan University, 2005, Songhu Road, Shanghai 200438, China

[2] School of Computer Science, Fudan University, 2005, Songhu Road, Shanghai 200438, China

[3] Research Institute of Intelligent Computing, Zhejiang Lab, Yuhang District, Hangzhou 311100, China