Incremental Feature Encoding in Differential Dataflow

Elias Strauss
Technische Universität Berlin
Berlin, Germany
elias.strauss@campus.tu-berlin.de

Benjamin Greb
Technische Universität Berlin
Berlin, Germany
b.greb@tu-berlin.de

ABSTRACT

Machine learning pipelines often assume static training datasets, making them inefficient for dynamic data sources like user interactions or sensor streams. This work focuses on incremental feature encoding, where transformations like standard scaling and one-hot encoding require maintaining evolving dataset statistics. We propose a framework leveraging Differential Dataflow to efficiently process frequent updates while remaining compatible with scikit-learn.
Our evaluation shows that while traditional methods excel at initial encoding, our approach significantly improves update efficiency, maintaining near-constant update times. Microbenchmarks on key encoders confirm these gains, though challenges remain in constructing output vectors for full encoding pipelines.

1 INTRODUCTION

In today's machine learning system landscape it is a common assumption that training datasets are relatively static. Accordingly many functions and components are designed to process entire datasets at once. However, there are various potential use cases where training sets can be highly dynamic. This includes for example training data generated from user interactions with a website, to make predictions for a user's interests. Similarly, data from sensors in e.g. industrial machines or healthcare may continuously generate new datapoints that could be used to train a machine learning model. Moreover, it can also be necessary to remove samples from the dataset, for example to comply with privacy regulations. In those situations, many existing solutions become rather inefficient, as they are not designed for many, frequent updates, and therefore need to spend large efforts on always reprocessing the entire training dataset or introduce significant update delays to amortize the cost.

While there are multiple phases in a typical machine learning pipeline that suffer from this problem, we are particularly concerned about the feature encoding stage. While some encodings can be computed as a simple static mapping, problems start to arise when using feature encoding methods such as standard scaling or one hot encoding that first need to be fitted on the dataset by computing certain statistics or metadata, to then encode the samples accordingly [1]. Here, those statistics may change over time, e.g. through distribution shifts or the introduction of new distinct values, and applying those changes would also require updates of all previously encoded values. A naïve approach would be to never recompute the statistics at all, which could sometimes be a viable option if the changes are minor enough. However, the more likely outcome is that the feature encoding would degenerate over time, for example when the standard scaling doesn't apply a reasonably accurate standardization anymore.

We therefore see a desire to enable efficient and accurate incremental feature encoding for processing many smaller updates. Similar problems have also been studied in the area of database systems under the name "incremental view maintenance"[3], which is an umbrella term for methods that continuously update the result table of a materialized query. While existing technologies could likely be used by for example expressing feature encoding methods as SQL queries, our goal is to build a system that is specifically optimized for feature encoding, extensible, and integrable with other ML libraries, while building upon previous work for incremental computations.

Contributions. Our work consists of two main contributions:

- We developed a research prototype for incremental feature encoding, which makes use of existing incremental computation techniques. To ensure ease of use and consistency, we designed our API to align with the widely adopted machine learning library, `scikit-learn` [1], following its conventions and design principles. To make our system as efficient as possible, we also implemented various encoder-specific optimizations that reduce the computational overhead of incremental update processing.
- We evaluated our system by comparing it against scikit-learn. Here, we implemented multiple example pipelines using both our system and sklearn, and compared the runtime difference for incremental updates in our system versus the effort for complete recomputations using scikit-learn.

[1] https://scikit-learn.org/

2 PROBLEM & APPROACH

In this section, we delve deeper into the details of our system. We describe the system architecture, the software libraries utilized, and the algorithmic techniques used in its implementation.

Problem Statement. Our system is designed to efficiently perform incremental computation for feature encoding. This involves two key challenges: first, the incremental maintenance of dataset statistics during the fitting phase; and second, minimizing the continuous propagation of statistical changes during the tranform phase to all output tuples while ensuring that the output remains as close to the correct result as possible.

Approach. To lift much of the work for actually performing incremental computations, we use the Rust framework "differential dataflow" [5]. Differential dataflow itself is based on timely dataflow[6], which is a dataflow engine similar to e.g. Apache Flink [2] and enables building computation DAGs using operators like filter, map, reduce, etc. Differential dataflow extends timely dataflow by allowing efficient computations on multiple iterations of an input dataset, so that the individual operators only need to process the differences in their respective inputs, rather than the whole dataset all over again.

At the heart of differential dataflow sits the collection abstraction. A collection is a multiset of values, represented by a multiset of tuples of the form (v, t, d), where v is one of the distinct values of the dataset, t the logical timestamp of the iteration when the record was last inserted/updated, and d the count difference to the previous iteration (usually +1 for inserted and -1 for removed values). Eventually, multiple tuples for the same value may be consolidated to only store the total count of a value up to a certain timestamp.

The other core abstraction is operators. Operators transform one collection to another collection and can be composed to practically arbitrary computation DAGs. This is particularly useful, as the core operators in differential dataflow already implement the mechanisms for differential computations, and so any higher-level construct composed of them does so as well. A core composed operator in our work is the multi column encoder, which corresponds to an scikit-learn column transformer. For one, our multi column encoder receives the original input collection, where each row of the dataset is stored as a single composed value. Furthermore, it also receives a list of feature encoders that are each paired with a column id to specify on which column the encoder should be applied. It then breaks up the rows into individual row values, one for each column, and passes all values for a column to the respective encoders. The encoders will also return row values, which are then joined and concatenated using a row id to assemble a final dense vector as a result for each input row. As encoders receive and return the same data type, we can also enable to chain multiple encoders to a pipeline.

For the individual encoders we follow the typical scikit-learn-style API, where all encoders have a *fit()* method to e.g. collect statistics over an input collection, and a *transform()* method to perform the actual encoding. While the transformation often boils down to simple map operations, the fit stage tends to require complex aggregations. To approach this, we made strong use of monoids, which are custom foldable aggregation types that among others need to implement a *plus_equals()* function to aggregate partial results. The advantage of monoids is that differential dataflow can use them to cache and later re-use previous results to efficiently compute aggregations for further input iterations.

Implementation. In this subsection we want to give a short overview of the implementation details of some selected feature encoders from our system.

StandardScaler. The scikit-learn standard scaler performs standardization of a numerical column by shifting the mean to 0, and rescaling the standard deviation to 1, which can be important to achieve well behaved training of a model [1]. This transformation requires to continuously keep track of a good approximation of both the mean and standard deviation of the whole input dataset. While incremental updates to the mean are rather trivial, the standard deviation is tricky as the textbook formula requires to subtract the mean from every value, and therefore would need to trigger a re-computation on the whole dataset whenever the mean changes. To solve this we use Chan's method [4], which is a commonly used online single-pass formula for computing the variance incrementally. Another problem is that any change to the mean or standard deviation, and therefore an input to the transform method, would cause differential dataflow to trigger a complete re-transformation of the entire transformation phase. We approach this by applying a rounding to both returned values, so that minor fluctuations in the true values are not propagated to the transformation.

MinMaxScaler. The minmax scaler shifts and re-scales all values into the value range [0, 1] [1]. This requires continuous tracking of the minimum and maximum value of the column. In situations where values are only added and never removed, this is trivial, as the new min/max value can ever only be either the current one, or the newly added value. However, decremental computation also requires finding the next smallest/largest value, if the current min/max is removed. To perform this efficiently, we have a monoid which incrementally builds two indexed binary heaps (a min heap

and a max heap), holding all of the values in the column. This allows us to perform both deletions and insertions in O(log n) and retrieval of the min and max in O(1). A limitation here is that it requires the complete column to be small enough so that a single node can hold both heaps in memory, which may be infeasible for very large datasets.

OrdinalEncoder & OneHotEncoder. For encoding categorical values, we implemented both an ordinal encoder and a one-hot encoder. The ordinal encoder converts categorical values into an integer domain by assigning a unique integer for each distinct value, whereas the one-hot encoder transforms a value to a vector where each dimension represents one of the distinct values, with the appropriate dimension for each value set to 1 and the others to 0 [1]. The key challenge during the fit phase for both is to incrementally build a mapping from the distinct values in the dataset to consecutive integers, which can be either used directly by the ordinal encoder, or as vector indices in the one-hot encoder. For this we built a special position assignment monoid, which internally maintains a hashmap from values to integers, and the *plus_equals()* method then merges the right hand side map into the left hand side map. Extracting the distinct values from the dataset in the first place is a function that is already provided by differential dataflow. One performance concern here is the disappearance of distinct values in decremental scenarios. Providing strictly consecutive integer assignments would require us to compress the mapping and decrement all integers greater than the deleted integer, leading to potentially expensive recomputations during the transform phase. Instead, we leave the integers for deletes values as holes in the mapping, and mark those integers for re-assignment. Another problem, that is relevant for the one-hot encoder, is the appearance of new distinct values, as they would normally require to extend all previously encoded vectors by another dimension, which would also trigger a complete recomputation of the transform phase. To solve this, we do not extend them every single time, but only sporadically resize them by a factor of 1.5, and therefore overprovision vector dimensions for future values to decrease the number of extension operations.

TfidfTransformer. The tf-idf transformer operates on the result of another text vectorizer, like e.g. the hash vectorizer, and then performs a re-weighting of the tokens based on how frequent they are in the whole column, giving more relevance to rare tokens [1]. For this, we have defined a monoid that incrementally maintains a vector containing the document frequency of each token.

3 EVALUATION

Experimental Setup. To evaluate the performance and correctness of our incremental feature encoding framework, we employed the following methodology and infrastructure:

- **Correctness Testing:** We conducted unit tests to verify the accuracy of the output for each feature encoder.
- **Performance Benchmarking:**
  - Micro-benchmarks on individual feature encoders
  - Comparative benchmark of an entire feature encoding pipeline with incremental updates
- **Hardware & Software Configuration:**
  - **Machine:** Linux, AMD Ryzen 5 PRO 5675U (6 cores, 4.3 GHz turbo, 16MB L3 Cache), 64GB RAM.
  - **Implementation Stack:**
    - Rust 1.82.0 w/ `differential-dataflow` (0.13.1)
    - Python 3.9.6 w/ `scikit-learn` 1.5.2

In the following section, we present a detailed analysis of the performance benchmarking. We first describe the datasets used and the specific experiments conducted, followed by a discussion of the experimental results and their implications.

Datasets. For micro-benchmarking, we used synthetic data to control characteristics such as size, number of unique values, mean, and variance, allowing us to evaluate the framework's behavior under specific conditions.

For benchmarking the entire ML pipeline, we utilized real datasets, including the Diabetes Dataset, which contains 22 numeric features and approximately 70,000 rows.

Experiments and results. We limit our micro-benchmarking to the *OrdinalEncoder* and the *StandardScaler*, as they are representative of a broad range of other encoders in terms of behavior.

In the micro-benchmarks, we begin by fitting and transforming a single column of initial size n using a single-column encoder. Next, we perform a series of updates to this initial column, triggering the computation of the encoded output column within Differential Dataflow (DD). Each update consists of a single row and is processed individually, allowing us to measure the worst-case runtime for updating the encoded column in real-time.

In these experiments, we measure both the runtime of the initial fitting and transformation as well as the runtime of the updates. Finally, we compare our framework's performance in against the reference implementations in `scikit-learn` (SK), which appends each update to the initial column and iterates over the entire column to compute the updated encoding.

Micro-Benchmark: OrdinalEncoder. In this experiment, we initialized all column sizes n with 100 unique values. Every 20th update introduced a new unique value, triggering an internal update of the fit state.

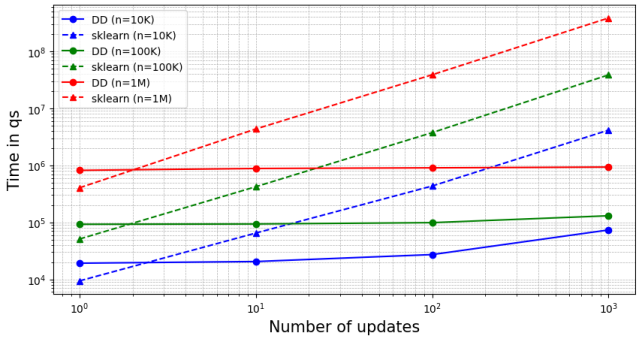

Figure 1: Micro-Benchmark: Ordinal Encoder

We observe that SK's initial computation, without updates, is consistently faster than DD, regardless of the initial size. This is because DD constructs indexing structures in the background to facilitate more efficient update propagation later on. However, after just 10 updates, DD outperforms SK, as it processes updates in constant time, independent of the initial column size.

Micro-Benchmark: StandardScaler. Similar to the *OrdinalEncoder*, the initial computation in DD is slower than in SK. However, the performance gap is even larger in this case, as SK internally leverages SIMD instructions for optimization. Additionally, it is more challenging for DD to amortize the initial runtime during updates. This is because certain updates push the mean and variance beyond rounding thresholds, requiring all output tuples to be updated. Nonetheless, after approximately 100 updates, regardless of the initial column size, DD outperforms SK. For small column sizes, the overhead of iterating through the entire column is not particularly costly for SK. This explains why the performance difference between DD and SK remains relatively small for n = 10K and 100K.

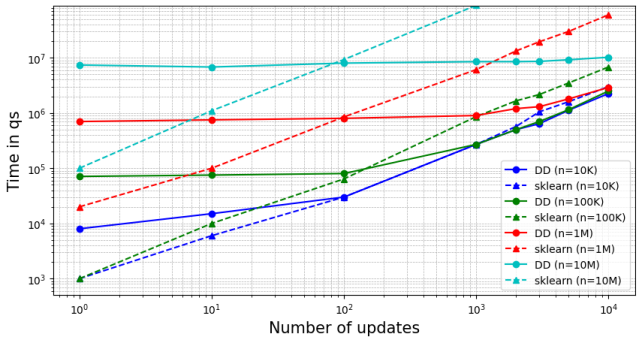

Figure 2: Microbenchmark: Standard Scaler

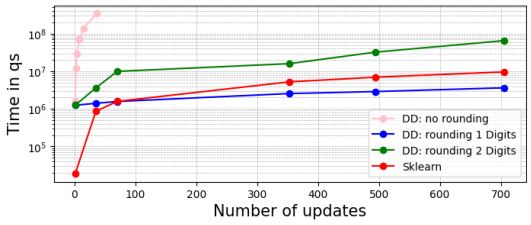

Figure 3: Feature Encoding Pipeline Diabetes Dataset

Diabetes Dataset Feature Encoding Pipeline consists of 21 *StandardScalers*. In this experiment, we simulate updates on the real dataset by first transforming 100 - X% of the data and then using the remaining X% to simulate updates.
The results indicate that it is currently challenging for the DD implementation to outperform SK. While the previous micro-benchmark on the *StandardScaler* demonstrated that rounding can help mitigate continuous updates for a single column—leading to fast updates—the primary bottleneck in this scenario lies in constructing the final output tuples.
Since each encoder processes a single column independently, we must join the outputs of all 21 encoders based on their row IDs and construct a unified feature vector from the join result. This step is the main bottleneck. Even though it is unlikely that all column encoders update their outputs simultaneously, the presence of just one encoder that modifies all its outputs is sufficient to trigger the reconstruction of all output vectors.

4 DISCUSSION AND CONCLUSION

In this work, we demonstrated the optimization potential of incremental computation for a wide range of feature encoding techniques. Furthermore, the current framework is designed to be easily extendable with new feature encoders, which can be seamlessly integrated with existing encoders, either within a multi-column encoder or as part of an encoder pipeline.
The experimental evaluation yielded remarkable results for many feature encoders. However, the full feature encoding pipeline—comprising multiple encoders—revealed opportunities for future work. In particular, improving the construction of output vectors remains an open challenge. Exploring more efficient approaches, such as in-place updates within a numeric representation outside of Differential Dataflow, could significantly enhance performance. A promising direction could be to efficiently determines update positions and values in Differential Dataflow, while materializing the actual numeric representation outside of the updates computing Differential Dataflow.

Detailed Contributions.

- **Elias:**
  - Code: General encoder framework inc. multi-column encoder, standard scaler, min-max-scaler, hash-ordinal-encoder, kbins-discretizer, pass-through-encoder, micro-benchmarks standard scaler + ordinal encoder
  - Paper: Abstract + Eval + Conclusion
  - Presentation: 50/50
- **Benjamin:**
  - Code: General encoder framework inc. pipeline, polynomial-encoder, one-hot-encoder, function-encoder, hash-vectorizer, count-vectorizer, tfidf-transformer, parts of min-max-scaler and ordinal encoder
  - Paper: Introduction, Problem & Approach
  - Presentation: 50/50

Github repository.
https://github.com/e-strauss/DiffDataflowMLPipelines

REFERENCES

[1] Andriy Burkov. 2020. *Machine Learning Engineering*. True Positive Incorporated.

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[3] Stefano Ceri and Jennifer Widom. 1991. Deriving Production Rules for Incremental View Maintenance.. In *VLDB*, Vol. 91. 577–589.

[4] Tony F Chan, Gene H Golub, and Randall J LeVeque. 1982. Updating formulae and a pairwise algorithm for computing sample variances. In *COMPSTAT 1982 5th Symposium held at Toulouse 1982: Part I: Proceedings in Computational Statistics*. Springer, 30–41.

[5] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow.. In *CIDR*.

[6] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.