Kusto Query Language Runtime Prediction

Or Argaman
Tel-Aviv University
Microsoft R&D
orargaman@gmail.com

Arik Olsh
Tel-Aviv University
Microsoft R&D
arikolsh@gmail.com

1 Introduction

Azure Data Explorer (ADX) is a fully-managed big data analytics cloud platform and data-exploration service, developed by Microsoft and is mainly used by Azure's analytics and monitoring systems, Log Analytics and Application Insights. As part of this platform, Microsoft introduced a SQL-like language named Kusto Query Language (KQL), which enables querying and visualizing of the data. Unlike SQL, KQL can only be used for querying data. KQL queries vary in their complexity which can effect the time it takes for them to run, ranging from a few milliseconds to more than five minutes. Some queries are so complex that they are often canceled by the ADX server after ten minutes of wait time. Identifying query complexity without running the query can benefit ADX server side by enabling throttling over usage by users. Additionally, predicting complexity can help save users a vast amount of time by preventing them from running queries that will timeout and help them refine too complex queries.

In this work we present a transformer based classifier which aims to evaluate query runtime duration and complexity based on the query and the data container context. For that purpose we introduce, to our knowledge, the first dataset of $523{,}391$ KQL queries along with their runtime duration and container context collected from Log Analytics and Application Insights backend services. The result we show are insufficient to properly evaluate the duration of a KQL query, we assess that this is mainly due to lack of enough data.

2 KQL Query-Duration Dataset

Here we describe the collection and compilation of our Kusto query-duration dataset. For a sample of the dataset, see Table 1.

2.1 Collecting the data

In order to collect the data for our dataset, we rigorously scraped a month worth of internal Microsoft telemetry. During the process we masked out string values which might contain sensitive information. For every query we made sure to collect the duration in milliseconds it took to run it. Furthermore, since the query's duration might vary when executed upon different data containers, we also saved a unique identifier of the data container corresponding with that query. In addition, to avoid duplicate queries in the data, for every group of duplicate queries coming from same container, we saved a single entry instead, with the query and the average duration of all those queries. Using this method we accumulated $523{,}391$ distinct queries that span over $34{,}275$ unique data containers, with a length of a single query being no more than 8712 bytes.

2.2 Data preprocessing

we used the following methods of preprocessing the data before feeding it to our model:

- Applied Kusto's own tokenizer written in .NET to parse the query into distinct special tokens used by the language and remove unwanted characters such as comments. For an example see Figure 1.
- Discretized the duration column into a new column called "Bucket" given a predefined global parameter for the number of buckets. For every query, the "Bucket" column holds the interval or bin to which the query's duration belongs. We defined a closed list of intervals, i.e. buckets, based on the data at hand.
- Removed outliers queries with duration out of the 0.995 percentile.

Query	Duration[ms]	ContainerId	Bucket

`requests`
`| where timestamp > ago ( 90d )`
`| take 100...`
	8008.0	ab120...	(4042.0, 14209.0]

`union events, dependencies` `| where itemId == '123' or...`	157.5	cd620...	(36.999, 184.0]

`Perf`
`| where TimeGenerated > ago ( 4h )`
`| wher...`
	191.0	a5f20...	(184.0, 281.25]

Table 1: KQL dataset samples.

- Removed all containers that had less than predetermined number of queries in the dataset. For a number of 100, this step reduced our data set to 216,507 queries that span over 813 data containers.

The reason for the last preprocessing is that we want to gather some prior context on each container before being able to evaluate on it, as complex queries can still run very fast if the container doesn't have a lot of data in it. This does limit us from predicting on new containers before they execute a sufficient amount of queries, but we believe this is a necessity to be able to predict with better accuracy.

3 Model

We model our classifier as a query classification learning problem, where we map a query and a container Id to a predicted duration bucket. Our model consists of 4 parts, Discretization stage, a Tokenizer, Encoder, and a Feed Forward layer as described in Figure 1.

Discretization Since our data contains continuous duration values and our model essentially acts a discrete classifier, we prepend a stage where we perform discretization of all duration values in our dataset into well defined duration intervals, i.e. buckets. For this purpose we chose two main discretization approaches, Uniform and K-Means which we discuss the results of each method in the Experiments (4) section.

- **Uniform** - separated all possible duration values into B number of bins, each having the same width. Given the sample with maximum duration value D and the number of buckets B, we computed the fixed interval width W with the formula: $W = D/B$. See example at Figure 2.

- **K-Means** - applied K-Means clustering to the continuous duration column, K corresponding to the predefined number of buckets B.

The Tokenizer is a byte-pair encoding (BPE) (Gage, 1994) tokenizer which we train on the data to produce the KQL vocabulary. We use the tokenizer to encode our queries along with a [CLS] token as a prefix for the query as introduced in Devlin et al. (2018).

The Encoder is a transformer encoder (Vaswani et al., 2017) constructed of 6 encoder layers with 12 attention heads in each Multi-Head Attention.

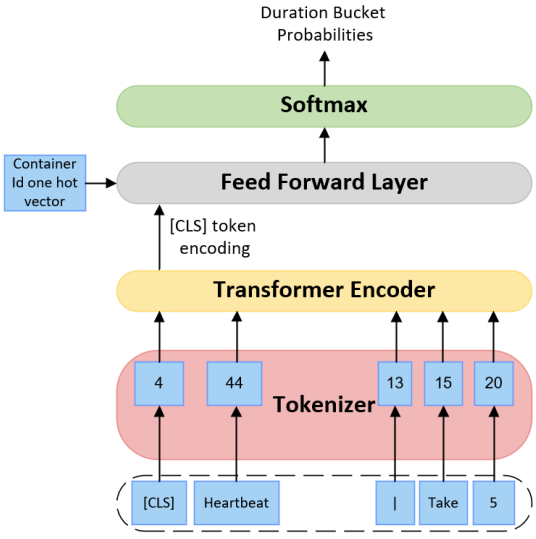

Figure 1: Model Architecture

	LR	Bucketing	Weights	Accuracy	RMSE [ms]	Accuracy 3 Buckets
1	2e^-3	Uniform	TRUE	0.169	15090.130	0.259
2	2e^-3	Uniform	FALSE	0.417	7234.350	0.655
3	2e^-3	KMeans	TRUE	0.157	14598.833	0.254
4	2e^-3	KMeans	FALSE	0.390	7236.073	0.656
5	1e^-2	Uniform	TRUE	0.183	16368.893	0.260
6	1e^-2	Uniform	FALSE	0.419	7244.624	0.655
7	1e^-2	KMeans	TRUE	0.167	17394.322	0.259
8	1e^-2	KMeans	FALSE	0.394	7246.636	0.657

Table 2: Grid Search results over Learning rate, bucketing method and weights. All set with minimum Queries Per Container of 100

AST	Accuracy	RMSE [ms]	Accuracy 3 Buckets
FALSE	0.417	7234.350	0.655
TRUE	0.417	7235.350	0.647

Table 3: Comparing AST experiment results

Minimum Queries Per Container	Accuracy	RMSE [ms]	Accuracy 3 Buckets
100	0.417	7234.350	0.655
50	0.457	5947.903	0.695
25	0.478	5828.964	0.712
0	0.582	6466.179	0.687

Table 4: Comparing different results for values of Minimum Queries Per Container

K	Accuracy	RMSE [ms]	Accuracy 3 Buckets
1	0.419	7406.361	0.640
2	0.419	7379.293	0.631
3	0.418	7346.468	0.644
4	0.419	7317.333	0.646
5	0.419	7289.098	0.663
6	0.418	7263.632	0.661
7	0.417	7234.350	0.655

Table 5: Comparing results of various K values for Top K prediction

From the encoder we extract the the encoded context of the CLS token and concatenate it with a one-hot vector which represents the container Id which ran the query.

Feed Forward Layer we insert the concatenated vector into a fully connected layer with one hidden layer and a Softmax layer. During training we used Argmax to predict the duration bucket label. At inference time we took the top K predictions values, normalized them to sum up to 1, and then multiplied each distribution by the average duration of the corresponding bucket. We are then able to give the exact expected duration for the the query, or use this prediction to predict the corresponding duration interval. [1]

4 Experiments

We train the model using a Cross Entropy loss, and used Adam (Kingma and Ba, 2014) optimization along with linear scheduler starting with 2500 warmup steps. Our model converged after 3 epochs, each epoch took 1.5 hours to train.

To best optimize our model, we used a grid search to validate 3 hyper-parameters: bucket discretization algorithm, loss weights, and learning rate (Table 2).

Loss Weights - as our distribution of data is highly uneven between short duration queries to long duration queries (Figure 2) we experimented with both weighted and unweighted loss functions.

After completing the grid search, we took the hyper-parameters of the model which presented the best result (table 2 row 2) and used it as a baseline to experiment over 3 more hyper-parameters, AST query form, Min queries per Container and Top-K Results.

AST Query Form - Similar to the approach presented in (Aharoni and Goldberg, 2017) we used the KQL parser to transform all queries into their AST tree representation. We then used that data as the input for our model. See Table 3 for results and Table 6 for format comparison.

[1] Our code, model and links to the dataset blob: https://github.com/arolshan/kql-nlp

Raw	AST
Heartbeat | take 10	QueryBlock List SeparatedElement ExpressionStatement PipeExpression NameReference TokenName IdentifierToken Heartbeat BarToken | TakeOperator TakeKeyword take List LongLiteralExpression LongLiteralToken 10 EndOfTextToken

Table 6: Query in raw and AST formats.

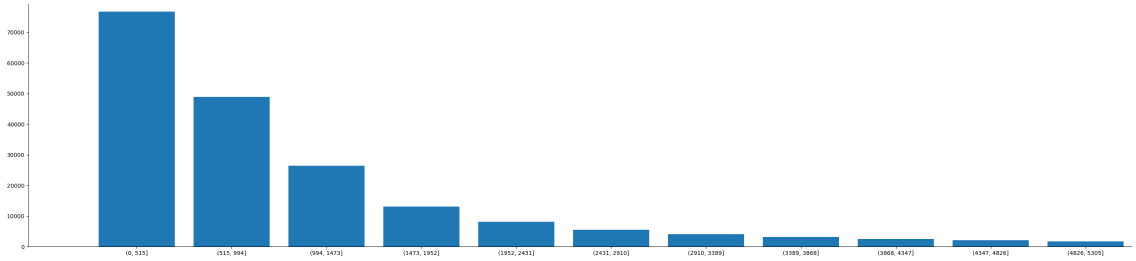

Figure 2: Uniform Discretization B=150.

Min Queries Per Container - As part of our model we decided the we require a certain amount of queries on a container in order for the model to predict the duration of a query. The reasoning for this is that while a query can be complex it can still have a short running time if the container has a small amount of data. We experimented with 4 values of this parameter, 100, 50, 25 and 0. Result are shown in Table 4.

Top-K Results - Our model calculates it final prediction based on a weighted average of the top K results we get from the Softmax function, we experimented with various values of K ranging from 1 to 7. Results are shown in Table 5.

For evaluation of our model we decided on 3 metrics:

- **Standard Bucket Accuracy** The accuracy over the B amount of buckets we had during training time

- **Accuracy Over 3 Buckets** We have concluded being able to give a rough estimation of the duration based on 3 bucket intervals, $[0_{ms}, 1000_{ms}]$ $(1000_{ms}, 10000_{ms}]$ $(10000_{ms}, \infty)$, will create enough value for the goals of this model.

- **Root Mean Squared Error** We calculated the RMSE between our inference time duration prediction to the true duration.

5 Results

The final validation accuracy for each hyperparameters setup is reported in Table 2. One can observe that using weighted cross entropy loss did have a significant negative impact on the results. In addition, using Uniform discretization on the data led to relatively better results then the KMeans discretization method.

We can see that reducing the Minimum Queries Per Container value improves the accuracy, but this is mostly attributed to an increase in short duration queries that only increased the imbalance in the data. From the same experiment we can also learn that having the query minimum threshold may improve the model results as we can see that both in the RMSE and the Accuracy 3 Buckets our model had better results with the threshold on 25 or 50 rather then with no minimum threshold.

Overall, the results show that the method took in the paper has not been proved to be an effective method of classifying KQL queries to their respective duration buckets. We attribute the low performance of our model to a deficiency in diverse data. The data we collected had a significant imbalance towards very short duration buckets. In other words, most of the queries lied in one side of the duration spectrum which resulted in a very biased model, this is illustrated in Figure 2. We believe that with a sufficient amount of data, which contains a higher rate of long duration samples, we would have been able to produce better results.

References

Roee Aharoni and Yoav Goldberg. 2017. Towards string-to-tree neural machine translation. *CoRR*, abs/1704.04743.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Philip Gage. 1994. A new algorithm for data compression. *C Users J.*, 12(2):23–38.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. Cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR*, abs/1706.03762.