Database-Agnostic Query Featurization and Deep Learners for Cardinality Estimation

Oscar Bashaw
obashaw@usc.edu
oscar.bashaw@sigmacomputing.com

Advised by: Georg Menzl
Research Scientist, Sigma Computing
georg@sigmacomputing.com

Abstract

There has been an explosion of research in using machine and deep learning in database and query optimization in recent years. A large portion of this research seeks to estimate the cardinality of SQL queries, but some approaches generalize better than others. Most proposed solutions at some point encode the SQL query as a numerical vector. Nuances in databases, query structures and changes to schemas can render some of the more rigid approaches to query vectorization useless. In this paper, we propose a query featurization method that can encode a query on an arbitrary database in a format that can be used for predicting query cardinality, considering both the order of terms in the query and underlying metadata of the base tables. We then offer three different deep learning approaches to map these query vectors to cardinality. Our performance on synthetically generated data suggests that there is a method for efficiently encoding queries, but this approach needs to be further refined if it is to generalize well.

1. Introduction

1.1 Motivation

Querying information from relational databases is crucial in the modern age. As the amount of data and the prevalence of companies who manage and consume data has grown, the ability to store and query data efficiently has become critically important. Cloud Data Warehouses (CDW) like Snowflake offer a nice solution to this task: these CDW are purpose-built to scale with changing client demands and they provide plenty of metadata about records and queries. The data in the CDW is often not consumed by end-users via SQL; instead, a cloud-native analytics and business intelligence (ABI) application such as Sigma will query information from a CDW and display it in a digestible format (either visually or in tables).

Being able to accurately predict the performance of such queries is necessary to the success of a company like Sigma, for two main reasons. Performance is usually evaluated in terms of query runtime or cardinality. Cardinality is often defined as the number of rows of data produced, but can similarly be thought of as the uniqueness of the data. First, having a good proxy for the query's cardinality before execution can help optimize workload balancers, caching layers or other intermediate pieces of the application, so that the application performance does not degrade. Second, knowing an approximate runtime for the query can improve the user experience (UX) of the app by informing users which actions may result in particularly costly (long-running) queries.

1.2 Estimating Cardinality

There are two distinct approaches to query cardinality estimation: statistical and learned. Most CDW and other SQL Databases employ statistical approaches. For example, Microsoft SQL Server and PostgreSQL (two of the most common SQL databases) use histograms of the distribution of column values to help inform cost optimizers (algorithms that generate the optimal query plan for a given task) [1] [2]. However, these statistical approaches are likely not optimal as alluded to in [3]. This gives way to more advanced methods of cardinality estimation: learned approaches. Learned cardinality estimation seeks to understand the mapping from some query onto its cost: $CE: Q \rightarrow C$. Most of the difficulty in this task is in how to best featurize query and schema structure, but there are significant considerations in the relationship between query features and cost.

Query Vectorization

The most naive and generalizable featurization of a query would be to create an n-by-1 vector v that simply counts the occurrence of each of n SQL keywords in a query (SELECT, FROM, WHERE, etc). Though this approach would generate a database-agnostic representation of a query, it entirely disregards (1) the structure of the query and (2) any underlying information about the database the query is being run on. Query structure has a substantial impact on the both the cardinality and runtimes of the query, so it is vital to preserve this in some way. Additionally, including some knowledge about the cardinalities of the base tables (the tables the query draws from) can be very informative for learning the output cardinality of a query. For a trivial example, consider the two SQL statements below:

(a) SELECT * FROM TABLE_A; **(b) SELECT * FROM TABLE_B;**

A basic, histogram-like approach would encode these queries in the same manner, but the lengths of tables **A** and **B** could differ by orders of magnitude. Without knowing that, the output cardinality of either statement cannot be discerned.

To capture the some extrinsic information about the database (and the query structure), methods like those in [3] have been proposed. Some of these query vectorization techniques include featurizing the vector and including some statistics about the underlying table data, or creating one-hot encodings for categorical variables and ranges for numerical ones. However, these approaches are designed for static, single-table environments, and updates to the database require a retraining of the model. This approach does not meet the database-agnostic criterion, one of our goals in this project.

Probably the best approach is offered in [4] and was the motivation for the (simpler) methods used in this project. It seeks to build a zero-shot learner for cardinality, one that can make predictions on an unseen database with no additional training (purely database agnostic). Their key method is to use the SQL database's/CDW's query planner to create a graph representation of a query that has different node types and edge weights to capture the different operators, tables, predicates, etc in an arbitrary query. These nodes are "annotated with transferable features that generalize across databases" [4] and are insensitive to changes in the database schemas. Additionally, because queries have some tree structure that is defined by the query planner (not necessarily respective of the ordering of SQL query text), these graphs are directed and so message passing can be applied. What this amounts to is that the hidden state at each leaf node in the graph can be combined with its siblings (bottom-up) to reach some single final state [5] [6].

This hidden state can then be fed to an MLP and used for estimating performance. The Zero-Shot [4] approach is particularly nice because it captures the tree-structure of the query *and* the extrinsic database information in one fell swoop.

This is nice, but the query planner is not always easily accessible. We propose a simpler approach to vectorizing queries that is inspired by the methods in [4] [8] [9] in that it creates a zero-shot learner, but does not require access to anything more than raw SQL text and some database metadata. The details will be covered in more depth in section 3.1.

2. Theoretical Background

2.1 Long Short Term Memory (LSTM) Auto-Encoder Networks

Also known as LSTM Encoder-Decoders, these self-learning networks can be especially useful in creating a lower-dimensional representation of input data that can then be used in a predictive model or elsewhere. The main strategy of these networks is to use an LSTM to take some input sequence and map it to a hidden state of more manageable size, then use another LSTM to recreate the original input sequence using the hidden state as an input. The model's loss depends on how well it can learn to recreate the input sequence (self-learning).

Often, these networks are used in natural language processing (NLP) applications, where sequences of words are tokenized (represented as scalars), embedded (scalars are mapped to some vector of high dimension) and padded (sequence lengths made constant) before being sent to an LSTM Auto-Encoder [7]. The hope is that the model will be able to learn a relevant context window for a sample in the sequence. Once trained, the hidden state representation a sequence can be fed through the decoder (which may translate the sequence to another language) or extracted and used for a different predictive task.

This is a gross simplification of the algorithms and mathematics required for LSTMs and LSTM Encoder-Decoder networks, but this idea of tokenizing and embedding a query sequence is covered in depth in [8] and is a key source of inspiration for our query vectorization technique (described in Section 3.2). Particularly, the method of maintaining the order of a character-based input sequence while adding more relevant numerical information is quasi-applicable to query encoding.

2.2 Convolutional Neural Networks (CNNs)

CNNs are most popular in image and video processing, where data comes in as $n \times n$ matrices of values representing pixel activations (images, in 2D CNNs) or as $n \times n \times t$ matrices that represent sequences of images (video data, in 3D CNNs). Additionally, any given input image ($n \times n$ matrix) may have a number channels C associated with it. For example, color images are often represented by a matrix of $n \times n$ pixels, with an (r, g, b) tuple for each pixel (each example has $n \times n \times 3$ features). A sequence of t of these color images might represent a color video ($n \times n \times 3 \times t$ features). CNNs take these multidimensional inputs and map them to other dimensions through various kernel and pooling operations. To make a prediction, be it a basic classification or a regression, the dimensions are generally mapped from high-dimensional space down to a vector of shape $p \times 1$ ($p \geq 1$).

2.3 Artificial Neural Networks (ANNs)

ANNs are the most basic type of multilayer perceptrons. Linear matrix multiplications are interspersed by nonlinear activation functions, creating a nonlinear mapping from some $m \times 1$ input vector to a vector of shape $p \times 1$ ($p \geq 1$). They can be thought of as the lower-dimensional cousin of CNNs, but the two are often used in tandem. CNNs can be used to extract the meaningful features from some input and ANNs can then be used to make predictions based off of those features. We will explore both frameworks in this paper.

Note that the semantic meanings attached to the elements in the matrix or to the channels do not change the learning task from a mathematical standpoint. In our case, the $c \times 3$ featurization of each query can be thought of as a $c \times 1$ input with three channels ($c \times 1 \times 3$).

3. Methods

3.1 Data

A substantial amount of time was spent procuring workloads (query + cardinality pairs). Originally, the hope was to gain access to Sigma's query planner and use it to implement a graph-based query representation, but it quickly became clear that this would require substantial effort from our backend engineering team. The next potential source of data were the workloads used in the various papers [4] [8] [9] . However, most of these papers did not disclose the databases that their training workloads came from, or if they did, the actual data was nowhere to be found. We were able to find just the queries for one source, but no complete workloads or database information. That left us with just one source of workload and database information: [10].

Kipf et. al were kind enough to publicize their repository [11] which included workloads with raw SQL text *and* semi-parsed SQL queries coupled with their cardinalities. These queries are auto-generated from various query templates limited in complexity to include varying numbers of only joins and where predicates (no subqueries or other clauses). This upper bound on query complexity means that the models described in Section 3.2 will not be able to generalize to production-level SQL queries, but this dataset still ended up being a hard to learn. Additionally, the workloads provided are based on a single database: the IMDb Movies Database. At first glance, this seems like the complete antithesis of our project statement (to create a database agnostic learner). Even though these workloads are of a single origin, the featurization method used in this paper works for a single-layer query with just **FROM**, **JOIN** and **WHERE** operators on *any* database. Also important to note is that the IMDb dataset seems to be one of the only benchmarks for cardinality estimation. With these disclaimers out of the way, we can now discuss the featurization.

3.2 Preprocessing

Before any featurization happened, workloads had to be parsed out into query clauses that could be linked to database metadata. In addition to raw text, queries are provided in the following semi-parsed format:

{Tables T_i} # { (T_{left}.key, <operator>, T_{right}.key) } # { (Column C_i , <operator> , value)}] # cardinality

All tables and their aliases are comma separated in clause 1, any joins in clause 2 and any where predicates in clause 3 (clauses separated by #). Note that some queries do not contain joins or where predicates, which is reflected in the featurizations. That is, featurizations only include clause tokens and clause tuples for clauses appearing in the query (a query with no joins does not receive the join token). The cardinality for each query is included at the end of each line as a scalar.

Also from [11], the metadata of the schema is provided for each column in each table. The metadata includes column cardinality (equal for all columns in a table), column minimum and maximum values and number of distinct values. We leverage this data in our vectorization.

First, queries are split into the three clauses: **FROM**, **JOIN** and **WHERE**. Each clause is given a 1×3 one-hot encoding. Then a 1×3 tuple is created for each sub-piece of each clause (each table in **FROM**, predicate in **WHERE**, etc) that captures some information about the logic of the operator and/or the cardinality of the columns involved. For the **FROM** clause, each tuple contains the table cardinality divided by the maximum table cardinality (scaled cardinality) seen on training data followed by zeros in positions 2 and 3. **JOIN** clauses contain the scaled cardinality of the left-side table, the operator type represented as a scalar, and the scaled cardinality of the right-side table. **WHERE** clause tuples consist of the scaled cardinality of the column referenced (equal to the cardinality of the table it belongs to), a scalar representation of the operator and the number of unique values in the column divided by its absolute cardinality.

Each of these tuples is then concatenated in the order of the query. For queries q_i with some length $l_i = n_{tables} + n_{joins} + n_{predicates} + n_{clauses}$, $n_{clauses} \in \{1,2,3\}$, we then have query vectorizations of shape $l_i \times 3$. These vectors are then padded with zeros on the tail to a constant length c, yielding a $c \times 3$ query vector that can be interpreted mathematically. This algorithm is defined more rigorously in Algorithm 3.2 below, and some example query featurizations are given in Tables 3.1 and 3.2.

Algorithm 3.2: Query Parsing

1. Define some operator tokens (scalars) as '=' : 1, '>' : 2, '<' : 3, etc.
2. Split each query into clauses on the # character
3. For each clause (max 3)
    a. If the clause has content
        i. Initialize some list with 1 element: the clause token
        ii. Find each item (table, join or predicate) in the clause and append the appropriate tuple to the list
            1. **FROM**: ($\frac{Table\ Cardinality}{maxCardinality}$, 0 , 0)
            2. **JOIN:** ($\frac{Left\ Table\ Cardinality}{maxCardinality}$, $<_{operator\ token}>$, $\frac{Right\ Table\ Cardinality}{maxCardinality}$)
            3. **WHERE:** ($\frac{Column\ Cardinality}{maxCardinality}$, $<_{operator\ token}>$, $\frac{\#\ Unique\ Values\ in\ column}{Column\ Cardinality}$)

There are some important mentions about this query encoding. First, it is database and table agnostic because it drops literals and includes schema metadata (column and table measures), which is the first step in creating a zero-shot model. Second, it preserves the written order of the query, which is not necessarily the execution order of the query. This is where the graph-based featurization [4] leverages its access to the query planner and query tree, but we offer an implementation that assumes no access to such tools (which may often be the case).

Query 1: FROM table $\mathbf{T_1}$, table $\mathbf{T_2}$ WHERE $\mathbf{T_1}$.id=$\mathbf{T_2}$.id and $\mathbf{T_1}.\text{col}_1 > a$ and $\mathbf{T_2}.\text{col}_4 < b$

$\mathbf{v_1}$	$\mathbf{v_2}$	$\mathbf{v_3}$	**Description**
1	0	0	FROM Token
$\frac{Card(T_1)}{maxCard}$	0	0	First table in query
$\frac{Card(T_2)}{maxCard}$	0	0	Second table in query
0	1	0	JOIN Token
$\frac{Card(T_{left})}{maxCard}$	= Operator Scalar	$\frac{Card(T_{right})}{maxCard}$	First join in query
0	0	1	WHERE Token
$\frac{Card(column1)}{maxCard}$	> Operator Scalar	$\frac{Unique(column1)}{Card(column1)}$	First predicate
$\frac{Card(column2)}{maxCard}$	< Operator Scalar	$\frac{Unique(column2)}{Card(column2)}$	Second predicate

Table 3.1: This table depicts a broken down version of Algorithm 3.1 applied to the Query 1 (listed at the top of the table). Columns $\mathbf{v_i}$ represent the i-th element in the tuple for each token or clause. Descriptions of each clause tuple are given in the right column. The final vectorization (of shape 8×3 before padding) is highlighted in green.

Query 2: FROM table $\mathbf{T_1}$ WHERE $\mathbf{T_1}.\text{col}_1 > a$ and $\mathbf{T_1}.\text{col}_3 < b$ and $\mathbf{T_1}.\text{col}_4 = c$

$\mathbf{v_1}$	$\mathbf{v_2}$	$\mathbf{v_3}$	**Description**
1	0	0	FROM Token
$\frac{Card(T_1)}{maxCard}$	0	0	First table in query
0	0	1	WHERE Token
$\frac{Card(column1)}{maxCard}$	> Operator Scalar	$\frac{Unique(column1)}{Card(column1)}$	First predicate
$\frac{Card(column3)}{maxCard}$	< Operator Scalar	$\frac{Unique(column3)}{Card(column3)}$	Second predicate
$\frac{Card(column4)}{maxCard}$	= Operator Scalar	$\frac{Unique(column4)}{Card(column4)}$	Third predicate

Table 3.1: This table depicts a broken down version of Algorithm 3.1 applied to the Query 2 (listed at the top of the table). The final vectorization (of shape 6×3 before padding) is highlighted in green.

3.3 Models

Two main model structures were implemented for this task: ANNs and CNNs. The following subsections will discuss the implementation details and rationale for each of the two.

3.3.1 ANNs for Cardinality Estimation

ANNs were selected as a model class for two reasons. First, they are the most fundamental type of deep learners and serve as a basis of comparison for more advanced deep learning models. This is attractive because ANNs are known to work well on many different types of learning problems and because they might provide a proxy for the question of "is this problem learnable by deep learning?". ANNs are also relatively easy to configure and inexpensive to train when compared to other network structures. Second, ANNs assume no sequential meaning to the input features. That is, the ordering of the input features is meaningless and should have no impact on the performance of the model.

Though this fact works against the query representation described in Section 3.2, there is good reason to implement ANNs for learning query cardinality: SQL text does not necessarily represent the order of execution of a query. SQL is great for human interpretation of a query's semantic structure (what data is being queried from which tables, what relations are there between pieces of data, what filters are being applied, etc.). However, a database's query planner/optimizer will likely not execute the query in such a linear order, especially as the query complexity increases to include more types of clauses, more tables and more subqueries. This effect is amplified when CDW are considered, as data from the same table might be partitioned across hundreds or even thousands of nodes.

Without access to the query planner, sequencing the query in the correct order becomes exceedingly difficult. Instead of exploring that, we choose to use the information already in our vectorization, because it respects the amount of data being scanned and the various operations performed at a more aggregated level. To feed these $c \times 3$ query vectors into an ANN, they must be flattened to $3*c$ dimensions. Using this flattened vector to train a cardinality-predicting model assumes that the aggregated query statistics are sufficient inputs for the learning problem, which may well be a fair assumption.

3.3.2 CNNs for Cardinality Estimation

CNNs represent an interesting solution for the cardinality estimation task. Most generally, CNNs can handle multidimensional input vectors of the form $m \times n \times p \times q$ (tensors) for each data sample. This input shape is a nice fit for the query tensors that have $c \times 3$ features each, so the data structure and ordering can be preserved by a 1D convolution over 3 input channels.

Also important is the capability of CNNs to perform feature extraction from inputs that can respect and identify (but not necessarily preserve) the structures present in an input. Such is the reason that CNNs are used for image processing (recognition and segmentation). The movement of a kernel over an input tensor can be thought of as a mapping to lower dimension that respects the neighborhoods of the input.

This concept ties nicely with the order-respecting query featurization we detail previously. Moving a kernel over the features of each channel in the query tensor might well extract some local features of the query, like information about the **WHERE** clause, for example. These latent states are gradually transformed (via pooling and kernels) towards a flattened feature vector that can be used to predict

cardinality. This approach assumes that there is value in the order of the query text, which goes against the logic behind using ANNs for this task, but does not go as far as an LSTM Encoder-Decoder that seeks to represent exactly the text structure of a query.

Loss Function

While mean-squared error (or some cousin of it) is often used for regression-style learning, a more applicable metric is proposed in [3][10]: Q-error (defined below). Q-error seeks to measure the degree of magnitude by which a prediction differs from a target. It is defined as

$$\varepsilon_Q = \frac{max(pred, label)}{min(pred, label)}$$

In our case, this is particularly appealing because of the range of magnitudes of query cardinalities. Some queries may have cardinalities on the order of 1e2, where some cardinalities may lay in the range of 1e7 (or higher, in some cloud-based applications). Understanding whether the cardinality of a workload is 10 vs. 20 is not very valuable in practice because it will not have a noticeable impact on system performance. It is crucially important, however, to be able to estimate if a workload will have a cardinality of 1 million or 100 (consider the different compute and post-processing resources required for each).

4. Application

Three different model structures were tuned and evaluated over the train dataset: a simple 2-layer ANN, a simple CNN and an advanced CNN. For all models, the same 70,000 rows were set for training and 30,000 for validation, with no cross validation performed due to the compute resources that would be needed. An Adam() optimizer was applied to the Q-error loss functions implemented by Kipf [11]. All of the models were implemented and trained in pyTorch. The hyperparameter search space was similar but not identical for the ANN and CNNs. Table 4.1 lists the explicit hyperparameters and their values.

Model	Learning Rates	Weight Decay	Batch Size
ANN	3e-4, 3e-3, 3e-2	2e-3, 2e-2	16, 32
BaseCNN	3e-4, 3e-3	2e-3, 2e-2	16, 32
AdvCNN	3e-4, 3e-3	2e-3, 2e-2	16, 32

Table 4.1: Hyperparameter search space for three model structures. Note that the CNNs do not evaluate with learning rate 3e-2 in order to reduce compute cost.

Model Structures

The ANN had 2 hidden layers, the first with 32 nodes and the second with 16. Each layer had ReLU activation with a dropout probability of .25. Inputs were 42-feature vectors and outputs were scalars. The BaseCNN took in a 14×3 vector to a Conv1D layer that output a single channel. Batch normalization and dropout regularization were then applied and the resulting vector was sent through 2 Linear layers with ReLU activation before outputting a scalar estimate of cardinality. The final model was AdvCNN, a deeper CNN with a feature extraction block that increased the number of channels while decreasing the

feature size with kernels and MaxPooling. A flattened vector was then fed to a 2-layer MLP. A diagram is included in Figure 4.1 below.

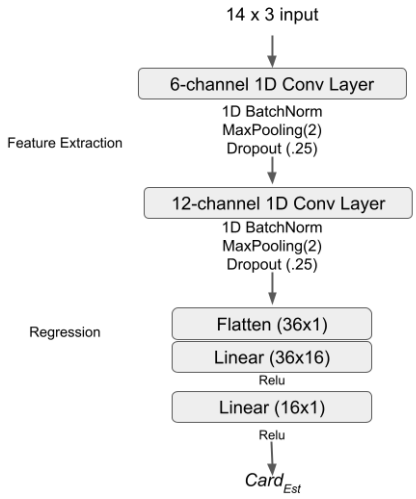

Figure 4.1: AdvCNN accepts a 14 feature input, where each feature has 3 channels. There are two blocks of 1D-Convolution, BatchNorm, MaxPooling and Dropout followed by a flattening to a 36 feature vector. That flattened vector is passed through two final Linear layers with ReLU activation to estimate cardinality.

Train and Validation Performance

Generally, the models were all able to reduce the loss by multiple orders of magnitude over their 25 training epochs. Generally, the ANNs (the simplest of the models) performed the best over the validation set, with a minimal Q-error of 248 coming from the MLP with learning rate set to 3e-3, weight decay set to 2e-3 and batch size set to 16. The best Basic CNN and Advanced CNN had the same set of parameters and yielded Q-errors of 450 and 317, respectively. Their learning curves are depicted below in Figures 4.2. The fact that the ANN performed the best on the training set suggests, but not confirms, that the sequence of clauses in query text contains little value in predicting cardinality.

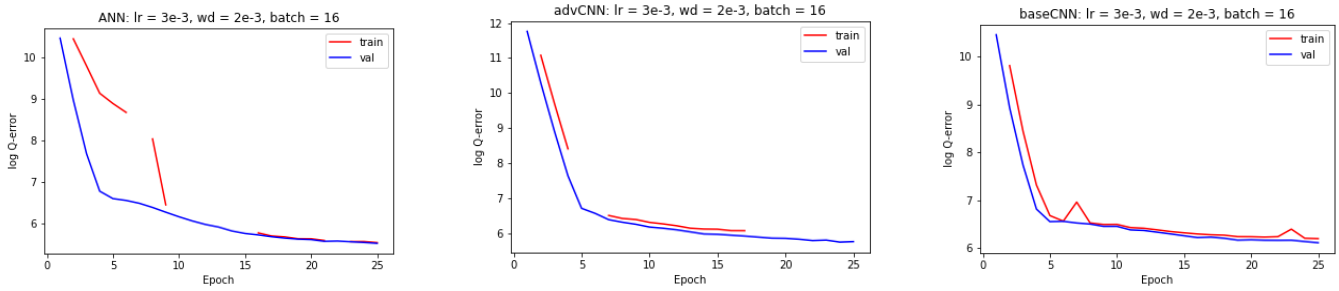

Figure 4.2: a) Left: The ANN configuration achieved a minimum Q-error of 248. The gaps in the training curve represent epochs with 'inf' loss, likely because the cardinalities (labels) were not normalized. **b) Middle:** The advanced CNN performed second-best, achieving a minimum Q-error of 317. The missing training epochs exist for the same reason as in the ANN. **c) Right:** The baseCNN achieves a minimum Q-error of 450: not as good as the other two but not terrible. The loss did not reach infinity at any point here.

All three best-performers are then evaluated on each of the three test sets: Synthetic, Scale and JOB-Light. We do not select one model because it is interesting to see how each model structure

generalizes to new data. These three test sets are also provided by Kipf et. al and are of the same format as the original training data. Each of the three has a nuance: Synthetic is an additional 5,000 workloads that come from the same generator used to build the training data, Scale contains an additional 500 workloads from the same generator but with more joins, and JOB-Light workload contains queries not generated from the templates.

Not knowing what the testing data would look like, all models were trained to accept feature input of 14 tuples (or 42 flattened features). However, the JOB-Light queries have a maximum of 17 tuples once parsed using our methods, and Scale queries have a maximum of 19 tuples. Instead of retraining the models to accommodate these larger workloads, we opted to take only the first 14 tuples (or 42 elements in the flattened case). This is a fairer evaluation of our entire algorithm and exposes one of its limitations.

Q-Error loss on the three test sets is as listed in Table 4.1 below. Quite surprising to see is that the BaseCNN generalized much better to the Scale and JOB-Light sets than either of the other models, despite lagging behind during training and validation. Also notable is that both CNNs performed better by an order of magnitude when compared to the ANN on those foreign sets. As expected, the performance rankings on the Synthetic dataset mirrored that of training.

Model	**Scale**	**Synthetic**	**JOB-Light**
ANN	6,116,056	241	373,359
BaseCNN	49,058	457	10,138
AdvCNN	208,432	323	28,744

The performance by all models on the Scale and JOB-Light test sets are much worse than on Synthetic, almost surely due to the truncation of the query vectorizations in those sets to match the trained data. Meaningful information was lost to this operation, but it is a necessary and fair evil. That said, the assumption is that even with the entire query featurizations, the models still would have performed worse on Scale and JOB-Light because they come from an unseen feature space. Compared to the models covered in [10] on the same data, our models do not perform as well (Q-error for the MSCN on Synthetic was 1.18).

5. Extensions

There are many ways to extend and improve the work covered in this paper. Foremost, we need a better way to embed queries as vectors. The test set results showed that setting a maximum processable length yields drastic results when new data is introduced to the system. We could explore a tokenization strategy as mentioned in section 2.1 that would embed any SQL text sequence into k elements before annotating those elements with more data, but that may not be optimal. The ideal case would be to implement a version of the Graph Embedding from [4], as it is the most robust one we have seen.

To do that, there is a lot of exploration needed on the various query optimizers and if these optimizers are generally available in production settings. It would also be important to consider more varied SQL queries in terms of query structure and underlying databases (we use just the IMDb data). But, a good first step is to find a featurization algorithm that can truly produced a zero-shot learner.

Something that was overlooked in this project was to normalize the query cardinalities before training the algorithm, just so that all the features and labels were in a similar ballpark. Another route of prediction would be to classify queries as costly or not-costly (or some multiclass order-of-cost-magnitude estimation). Thinking back to the original motivation, these outputs might be sufficient for initial improvements to the Sigma platform.

6. Conclusion

We proposed a method for vectorizing SQL queries for zero-shot learning and some models to learn the problem, with the goal of being able to read some arbitrary query and give a reasonable estimate of the cardinality. Though the results were not comparable to other state of the art methods, there were some valuable learnings. For one, the task of predicting cardinality can likely be learned with a more optimal embedding, and more importantly, there are many prospects for database-agnostic query representations and learning models. There is much work to be done before a cardinality estimator is deployed to a cloud application like Sigma, and the intricacies of CDW pose yet another challenge when concurrency, node clustering and warehouse sizes are taken into account.

Special thanks to Georg Menzl and Catagay Demiralp for advising me on this project and letting me make an attempt at a hard problem. Thanks also to Dr. Franzke for allowing me to pursue this as a term project for EE541 and for all of the knowledge shared this semester.

References

[1] WilliamDAssafMSFT, “Cardinality Estimation (SQL Server) - SQL server,” *SQL Server | Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver15.

[2] “72.1. row estimation examples,” *PostgreSQL Documentation*, 10-Feb-2022. [Online]. Available: https://www.postgresql.org/docs/current/row-estimation-examples.html.

[3] X. Wang, C. Qu, W. Wu, J. Wang, and Q. Zhou, “Are we ready for learned cardinality estimation?,” *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1640–1654, 2021.

[4] B. Hilprecht and C. Binnig, “Zero-shot cost models for out-of-the-box learned cost prediction,” *arXiv.org*, 03-Jan-2022. [Online]. Available: https://arxiv.org/abs/2201.00561v1.

[5] K. Kubara, “Introduction to message passing neural networks,” *Medium*, 05-Oct-2020. [Online]. Available: https://towardsdatascience.com/introduction-to-message-passing-neural-networks-e670dc103a87.

[6] K. Kubara, “Feature extraction for graphs,” *Medium*, 28-Sep-2020. [Online]. Available: https://towardsdatascience.com/feature-extraction-for-graphs-625f4c5fb8cd.

[7] P. Dhote, “Seq2Seq-encoder-Decoder-LSTM-model,” *Medium*, 20-Aug-2020. [Online]. Available: https://pradeep-dhote9.medium.com/seq2seq-encoder-decoder-lstm-model-1a1c9a43bbac.

[8] Jain, S., Howe, B., Yan, J., & Cruanes, T. (2018). Query2vec: An evaluation of NLP techniques for generalized workload analytics. *arXiv preprint arXiv:1801.05613*.

[9] Anonymous. 2018. PreQR: Pre-training Representation for SQL Under- standing. In Woodstock ’18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

[10] Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., & Kemper, A. (2018). Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*.

[11] andreaskipf. learnedcardinalities, GitHub. https://github.com/andreaskipf/learnedcardinalities