ESQL: Efficient Sql queries through exploitation of database leftovers and workloads

Antonios Papadakis
tony.pap.962@gmail.com
University of Athens
Athens, Greece

Abstract

This work is inspired by the paper of [1]. We introduce a number of different machine and deep learning models, which may aid the user of a database, whoever he might be, develop, write and execute efficient and correct queries. To achieve this goal we explore traditional methods, which utilize Bag of Words (BOW) and TF-IDF [2], as well as more modern neural network based techniques.

Since we did not have access to an implementation of the specific implementation of the models and techniques in the paper we began our development from scratch. For this reason we tried to reverse-engineer the original implementation with our own personal touch, while aiming to achieve similar results. Our models where evaluated on the basis of training time in conjunction with a variety of different evaluation metrics such as Accuracy score and Mean Squared Error.

Keywords: SQL, queries, machine learning, workloads

1 Introduction

One of the more challenging tasks in the broader technology community is communicating efficiently with a system and in particular with a database. Database–person communication is very frequently achieved through the use of SQL queries. Even though SQL queries can be rather simple to formulate and easy to utilize for users familiar with computer science, inexperienced users might require several cycles of tuning and execution to reach the desired output. In this work we try to examine a subset of methods that can accelerate and improve this interaction, by providing insights about SQL queries prior to execution. We reach this goal by predicting a range of query properties, such as query answer size, without relying on database statistics. Preliminary results coming from experiments on well-known public query workloads such as the SDSS and the SQLshare workloads, are encouraging and may confirm that data driven methods can become an added tool towards easing database-human interaction through the facilitation of query composition and analysis.

1.1 Problem Definition

As in the original paper we are focused on the user and not on the database. As stated earlier we aim to improve his interaction with the database system. The original paper focused on 2 different groups of users i.e. *end users* as in the people that perform transactions with the database and *database administrators.* The first group was facilitated through the prediction of query properties, thus saving the users effort and time from executing inefficient, unlikely to work or slow queries .The second group was aided through the categorization of different clients into classes by identifying their session id. Unfortunately, we were not able to acquire access to the session id data of the workloads. The reason for this may possibly be changes to the legislation regarding the privacy of users' data shared through the internet.

It is worth mentioning that such property estimation is used in tasks such as admission control, access control, scheduling, and costing during query optimization as in [3].in our approach we utilize the property estimations to increase database usability and ameliorate user experience. The difference between the 2 approaches is that the first requires database access and manual construction of cost models in the query optimizer, oftentimes based on apriori simplifying assumptions.

To achieve our goal we use large-scale already existing query workloads i.e. the Sloan Digital Sky Survey [4] (SDSS) and SQLshare [5]. Synthetic and or small scale workloads are not an option as stated by [6]. Our methodology and techniques would be easier to test and implement if given access to private workloads generated and maintained by large corporations.

To summarize the above, we state that our goal is to train a Classifier C and/or a Regressor R that given a query q or a query workload W will be able to predict efficiently the answer size, the CPU time needed for execution in the DBMS and whether the query will produce an error or not.

$$C : Classifier, q : query, C(q) \rightarrow Y_e$$

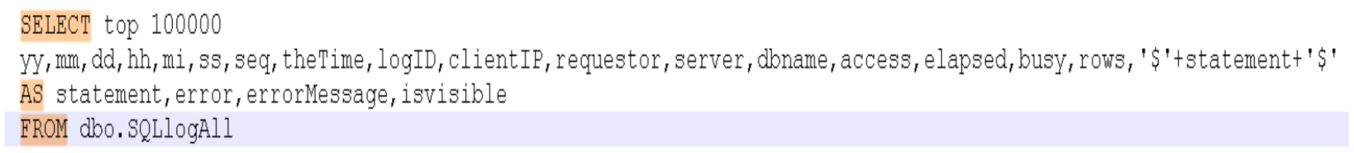
$$R : Regressor, q : query, R(q) \rightarrow Y_a || Y_c$$

where Y_e is the error label, Y_a the answer size and Y_c the CPU time needed for execution of the given query q.

1.2 Workloads

SDSS

The full SDSS dataset contains logs of queries and requests submitted to SDSS servers. It is described in [4], which we briefly summarize here. The Sloan Digital Sky Survey was



```
SELECT top 100000
yy,mm,dd,hh,mi,ss,seq,theTime,logID,clientIP,requestor,server,dbname,access,elapsed,busy,rows,'$'+statement+'$'
AS statement,error,errorMessage,isvisible
FROM dbo.SQLlogAll
```

Figure 1. The query posed against the SDSS database.

and remains the most ambitious astronomical survey ever undertaken. The surveys' goal is to map one-quarter of the entire sky in detail, determining the positions and absolute brightnesses of hundreds of millions of celestial objects. It will also measure the distances to more than a million galaxies and quasars.

The SDSS project also provides an online platform where you can query their database and get any information you may want on the universe and the stars. In addition to that, the SDSS database schema provides SQLlog, a table that contains each query ever posed to the database as well as information about the date and time it was executed, the logID and clientIP, rows returned after execution, CPU busy time while the query was executed, error types and many more. You can pose queries to their database by using the online casjobs platform after registering. The SDSS project provides each user with storage space where they can store the extracted tables from their database.

More specifically, the valuable columns extracted from the SDDS.SQLlog are:

• The query statement, extracted from the "Sqllog.statement" column, which may range from a correct SQL statement to a random text sequence.

• The query CPU time label, extracted from the "SqlLog.busy" column. This value is a real number and represents the query CPU time in seconds.

• The query answer size label, extracted from the "SqlLog.rows" column. This value is an integer and represents the number of rows retrieved for the query.

• The query error class label, extracted from the "SqlLog.error" column The error class indicates whether the query successfully executed, had a severe error, or a non-severe error. In the workload, there are 3 values allocated with respect to each kind of error.The 3 error values include success (the numeric value 0 for successful execution), non severe error(the numeric value 1) and severe error (the numeric value -1, indicates an invalid query that was rejected by the web portal). Those columns can be seen in *table0.*

SQLShare

The SQLShare query workload [5] is the result of a multiyear deployment of a database-as-a-service platform, where users upload their data, write queries, and share their results. This workload represents short-term, ad-hoc analytics over user-uploaded datasets. We extracted the following information fro this data source.

Name	Type	Description
TheTime	datetime	the timestamp
webserver	varchar(64)	the url
winname	varchar(64)	the windows name of the server
clientIP	varchar(16)	client IP address
seq	int	seq num for unique PK
server	varchar(32)	the name of the database server
dbname	varchar(32)	the name of the database
access	varchar(32)	the website DR1, collab,..
statement	varchar(7800)	The SQL statement
elapsed	real	lapse time for query
busy	real	total CPU time for query
rows	bigint	the number of rows generated
error	int	0-ok elseSQL error #; or <0
errorMessage	varchar(2000)	the error message

Table 1. SDSSs' SQLlog table

• The raw query statement, extracted from the "Query" as statement column to be the same as the one from SDSS. This may be a syntactically incorrect SQL query.

• The query CPU time label, extracted from the "QExec Time" as busy column to be the same as the one from SDSS. This value is an integer and represents the query CPU time in seconds.

1.3 Preprocessing

Before, we analyze or train our models we must first preprocess our raw data. We follow 2 different strategies with respects to preprocessing. The first one includes dealing with the SQL statements as sequences of words, which makes sense, since unlike natural language statements, SQL ones include words with certain syntactic meaning e.g. SELECT word is used to pose a query to the database and extract from it a number of rows (or columns). So, in order for this strategy to be successful we decide to replace all numeric values in our statement (not alphanumeric variables or parameters) with the <d> token (d stands for digit). This helps us avoid the unbounded or open vocabulary problem [7]. However, because SQL statements include valuable information in the form of table, predicate names or words with certain application meaning e.g. FROM, WHERE, ORDER BY, it is not possible to perform any other simplification on our statements (in a natural language problem we would remove punctuation and or stop words). The second one is managing our workloads' statements as sequences of characters whereby we try to obtain a meaningful connection between statements. In this case, we do not remove our digits, since we are not capable of knowing their exact meaning in a statement. This of course comes in contrast with our earlier approach where we could know if a digit is part of name/word or if it was just a numeric indicator i.e. *SELECT TOP 10 jobs FROM uni.Joblog*

1.4 Analysis

In this part of our facilitation pipeline we have to analyse our query workloads. In [1] the authors used an ANTLR(ANother Tool for Language Recognition) parser to generate the abstract syntax trees of their query statements and subsequently extract from those syntax trees valuable information to be used for the correct model selection. In our case though, an ANTLR parser was not an option since we did not use C++ or Java but Python [8] for our implementation. That lead us to a search for possible solutions through Python. SQLparse [9] or Pyparsing [10] python packages where not viable options since they required a lot of hard coding of parsing cases, thus making them too complex to modify in a short period of time. Fortunately, we were able to locate and use the sqlparser [11] python package. The Sqlparser is a currently inactive python 2 package which we were able to modify and utilize to create the new tool analysis.py which is able to run through a single query statement or even a large query workload and provide information about its syntactic and function properties. We used our analysis tool to extract the following 10 properties from our workloads:
• number of characters: the number of characters in a query. • number of words: the number of words in a query • number of functions: the number of function calls. • number of joins: the number of join operators. • number of unique table names: the number of unique • table names in the query. • number of selected columns: the number of selected columns in the query. • number of predicates: the number of predicates (logical conditions, e.g., $s.flags_s = 0$) used in a query. • number of predicate table names: the number of table names in the predicates. • nestedness level: the level of nestedness. • nested aggregation: it is true if nested queries involve aggregation and false otherwise.

We can observe below the bar charts *Figure2*, *Figure3* showcasing the different syntactic properties of the 2 workloads in logarithmic scale. As you can see there is a difference in the size of the 2 workloads with SDSS containing our extracted 100.000 queries and SQLShare approximately 26.700. However, one can observe that even though the SDSS workload is significantly larger it does not contain as much aggregate and nested queries as the SQLshare. Here we can find a difference between this work and the original, since we were not able to download the whole SDSS dataset and subsequently extract all the valuable information from it due to hardware limitations, even though according to their work SQLShare contains much more aggregate information. The best practice would be to download all of the available data and then extract all the available error queries (which are very rare) and to create a balanced summation made out of ordinary queries, nested queries, aggregate queries, queries with UNION, error queries etc. In any case it becomes prevalent that SQLShare has more complex queries syntactically.

We can also observe an imbalance to the corresponding labels. More than 96% of the queries in SDSS run without an error, approximately 4.3% t had non severe errors and 2.3% had severe errors. There is also a great gap between the answer size and CPU time labels where values may range from 0 to several thousands. The fact that our labels are highly skewed prompts us to normalize them.

2 Methodology-Implementation

Now its time to showcase the models we chose to train and subsequently use to solve our problem. Our approach is based on 2 axes. The first is a traditional machine learning model with a twist and the second one is a more modern deep learning approach. Each model obviously requires 2 steps for training, the first one being the vectorization step, since we are tackling an NLP adjacent problem and the second one being the data fitting or training step. Keep in mind that all representations where calculated twice for both word and character level models i.e. one time for each representation level.

2.1 Traditional model

For the Traditional model representation we chose a Bag Of Words inspired approach. So, we begun by calculating the set of the most frequent ngrams of our workload and then created a vocabulary out of the most frequent ngrams (floor values of most frequent ngrams were 300 for word level and 1700 for character level so as to include all the useful ngrams). Then having already represented our queries as sequences of ngrams, based on this vocabulary we map all their respective ngrams' to their one hot encodings and sum them to a one hot summation vector. Then for each token in the summation vector we calculate its TF-IDF weight and we obtain a normalized representation vector for each of our queries, whilst preventing bias towards longer queries. We then apply a prediction model given this fixed vdimensional feature vector. For classification problems, we apply the multinomial logistic regression model. For regression problems, we use Huber loss [13]. We optimize the parameters of the prediction model using scikit-learn [14]. We choose to use Huber Loss for regression due to the fact that our data have many outliers and Huber loss is not very sensitive to them as it treats error as square only inside an interval. Huber loss is shown below:

$$L_\delta = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & if\ |(y-\hat{y})| < \delta \\ \delta((y-\hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases}$$

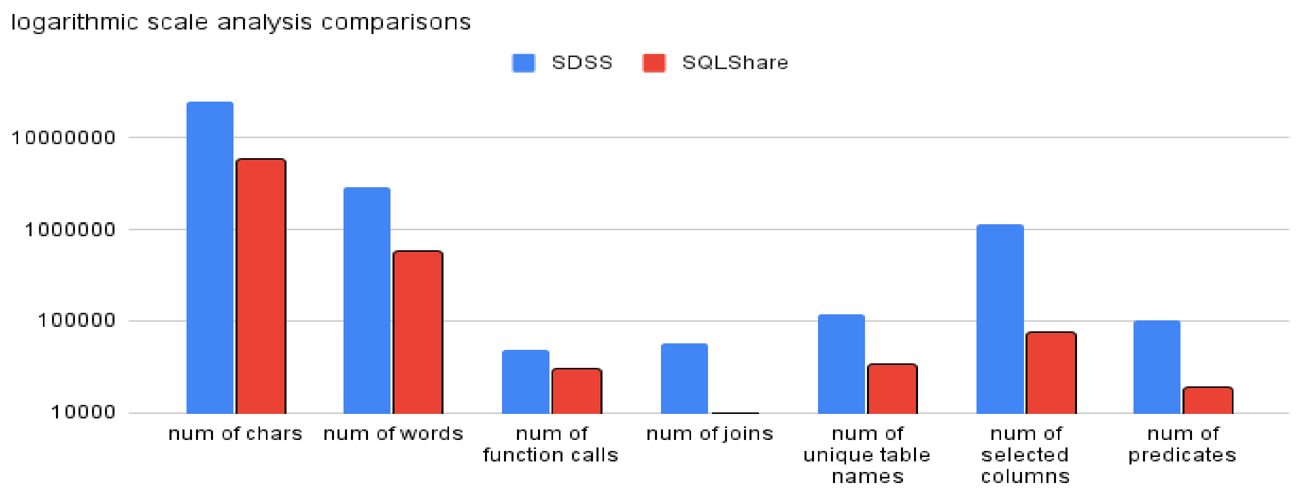


Figure 2. The comparison between number of chars, number of words, number of function calls, number of joins, number of unique table names, number of selected columns, number of predicates in logarithmic scale.

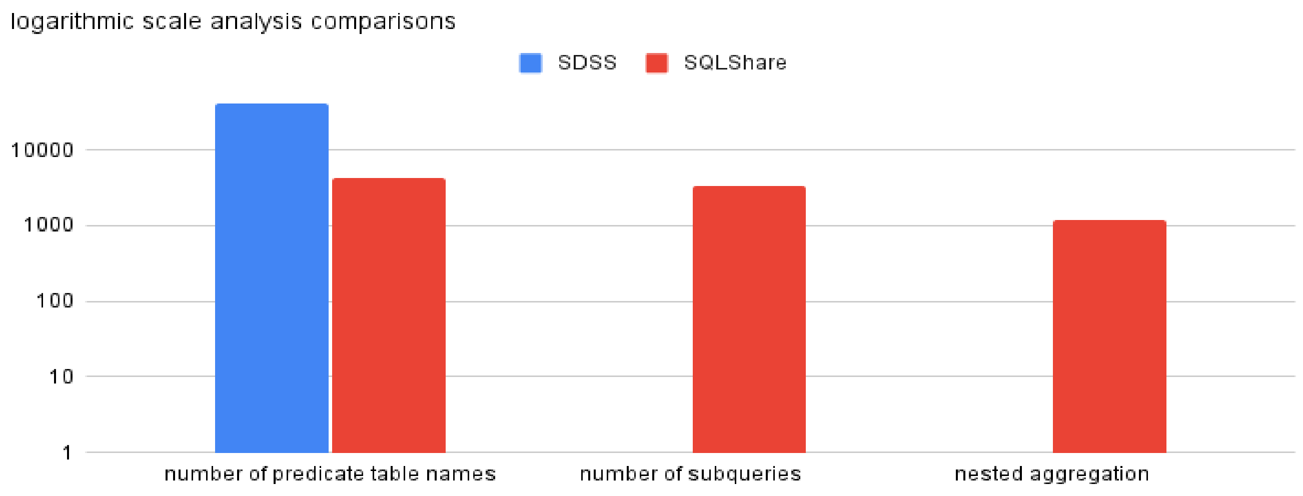


Figure 3. The comparison between number of predicate table names, number of subqueries, nested aggregation in logarithmic scale.

2.2 Neural Network models

In an attempt to provide better results we also try 2 more modern neural network architectures. For the SDSS problem due to the less complexity of the given query workload (can be seen in *Figure1* and *Figure2* we utilised the following models: The first one is a Long Short Term Memory (LSTM) based neural network model. Specifically, our model is made out of 3 different LSTM layers consisting of 32, 16 and 8 nodes in each layer respectively and a Dropout layer to avoid overfitting. The chosen activation function is RELU (Rectified Linear Unit) and the chosen optimizer is Adam. For the classification problem a categorical crossentropy loss function is chosen (since we have more than 2 classes) and a dense layer with 3 nodes and a softmax activation function is added after the LSTM layers. For the regression problems Huber Loss is chosen and a dense layer with 1 node and a linear activation is added after the LSTM layers. The second one is a Shallow Convolutional Neural Network (CNN) based model. Specifically, our model is made out of 1 one-dimensional CNN layer made out of 32 nodes and with a RELU activation function. Additionally a one dimensional Max Pooling layer is added so as to preserve sequential information, as well as a Dropout layer to avoid overfitting. We then pass our data through a Flatten layer and then through a Fully connected layer with 16 nodes. We empirically found this setup to be be more accurate than the plain Convolutional model,which we later proved experimentally.The chosen optimizer is Adam, although we also experimented with AdaMax for regression. For the classification problem as stated before, a categorical crossentropy loss function is chosen (since we have more than 2 classes) and a dense layer with 3 nodes and softmax activation function is added after the fully connected layer. For the regression problem Huber Loss is chosen and a dense layer with 1 node and a linear activation is added after the fully connected layer. For the SQLShare dataset where more nested queries, and nested aggregations can be found we decided to use the same lstm architecture but try a deeper cnn model. Our model was a 3 layered CNN architecture with 128, 64 and 32 nodes at each level, followed each time by a Max Pooling layer (with kernel size equal to 4) and a Dropout layer. Then. same as before a flatten and a fully connected layer leading to a linear unit with huber loss. Same optimizer as well as activation and loss functions as before were used. The reason for us using this deeper cnn architecture is empirical, since we believe the ability of the cnn network to identify spatial patterns, may provide better results in a workload with more subqueries.

The implementation of our neural network models was made in python using tensorflow [12] and keras [15]. We should mention hereby that during training of our neural network models we utilize keras' callback capabilities to monitor our models' training process, keep checkpoints and stop the process when the models performance is maximized. Keras.tokenizer fitted to the query statements is another utility we made good use of, to provide us with the appropriate vector representation for our query statements, so that we are able to feed them to the neural networks for training. Appart from the aforementioned neural network models we made an effort to add an extra embedding layer to our model so as to facilitate quality feature selection from the model, but it did not provide as any performance improvement. The same is true for deeper cnn or lstm models.

2.3 Normalisation

It is not possible to normalise our classification labels even though the error classes are imbalanced, due to the fact that we want to retain all the available information coming from real queries posed to the SDSS database. In any other case we would have to deal with synthetic workloads, results from which may not be applicable to real time query statements. However we can do a lot for our regression labels, which as we already mentioned are skewed. For this reason we experimented with the normalisation procedure provided by the original paper, but also tried a variation of min-max scaling. For our min-max variation we first do $y_i' = y_i + min(y)$ where = 1 and then we divide each value by the difference $|max - min|$. For the original paper normalisation we skip the division step and calculate $y_i' = ln(y_i')$.

2.4 run details

In this projects GitHub https://github.com/AntonyPapadakis/ and at the DBsystems repository you will find all the scripts you need to run it on your own. If you want to analyse a query workload and run all the experiments mentioned in this work you may run the main.py script. The main.py supports a number of flags: -r [arg] which is the argument flag for representation level char or word you want to use in your experiments, -d [arg] which is the argument flag for the dataset to be used, -f [arg] which is the argument flag for the filepath of the .npy file containing the representation vectors to be loaded. All these flags are optional and if you do not have any/some of the args available you may run main.py without arguments and you will be guided through training by the program. If you choose to do so please check the message prompts from the program, as you will be requested to input some answers. The evaluation.py script is used for evaluating each of the different models you have trained in accordance with the experiment protocols used in this paper. The file "commands for evaluation.txt" provides sample commands for calling evaluation.py. Any more details on the code implementation may be found by reading the code and its comments.

3 Experiments

3.1 Setup

Due to the great size and complexity of our datasets mainly SDSS we created the following protocol. Since no GPU was available on site for the neural network experiments, they were conducted either on google collabs' or kaggles' cloud shared GPU resources. We have to note here that without the use of a GPU the aforementioned neural network architectures are very difficult to train. The computational cost becomes restricting for the average database user. However, the traditional model approach may be deployed on a simple personal computer. Our traditional model was trained and tested on my personal computer which possesses 12 GB of RAM and an old generation Intel i5 CPU. We should also note here, though, that at least 8 GB of RAM or more are necessary for the traditional model to train due to the very high spatial complexity for the creation of the representation vectors and the BOW vocabulary. This problem may be surpassed by increasing the minimum bounding frequency for an ngram to be included in the vocabulary, at the cost of the models performance, since the smaller the dimensions of the representation vector, the more different representation vectors will overlap. We should note here that for the evaluation purposes of the regression problems we split the data to training and testing. For the regression problems a random ratio of 9 to 1 out of the 100000 SDSS and the 26700 SQLShare queries was used for train and test data. However, for the classification problem there was a lack of certain class labels in the original data. So instead of splitting our data we downloaded a further 90000 statements from the SDSS casjobs platform to be used only for testing in random order.

3.2 Evaluation Metrics

For the evaluation of our models we decided to showcase the following metrics: For our classification problems the Accuracy score and the F1 score for each error class metric:

$$\text{Accuracy}=\frac{TP+TN}{TP+TN+FP+FN}$$

$$\text{F1}=\frac{2*Precision*Recall}{Precision+Recall}$$

where TP symbolises the True Positive, TN the True Negative, FP the False Positive and FN the False Negative values coming from our models' predictions. These can be easily calculated by making the confusion matrix out of a models' predictions. Basically, The F1 score can be interpreted as a weighted average of precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. We also provide the number of missed values during our predictions so as to provide an overview of each models performance. For our regression problems we used the Mean Squared Error (MSE) and the Max Error score metrics:

$$\text{MSE}=\frac{1}{n}\sum_{i=1}^{n}(y_i-\widehat{y})^2$$

$$\text{MaxError}=Max(y-\widehat{y})$$

where we want MSE to be as close to 0 as possible and MaxError as close to 0 as possible.

3.3 Classification

We first present the results of our classification models' implementation of both the traditional and the neural network models. Keep in mind that the SDSS data are not the same as in the [1] since the database was accessed at a different point in time and as mentioned earlier we had a number of hardware limitations leading to the availability of less SDSS data. In *table2* one can observe the results of our models. All models where run at least 3 times and the mean was allocated to the table. We also provide a simple baseline which can be seen at the start of our table. This baseline is the most frequent class for classification and the median value for regression problems.

In *Figure 5* and *table2*, we can see that our implementation achieves very similar results with regards to classification as the best original model. However, one can observe that our models are not very successful in accurately predicting queries with errors and more specifically queries with severe errors. We can observe though, that our best models are better than the original at predicting non severe error queries. That urges us to assume that our models might perform badly with respect to the severe error class due to the lack of available data. In other words if we trained our models

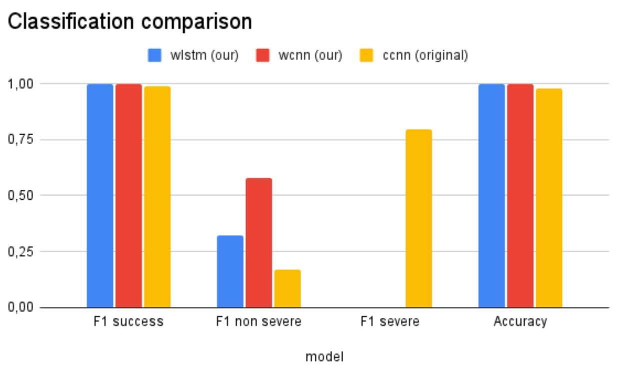


Figure 4. comparison between best original[1] error classification model and ours

with a dataset of the same size as the one used in the original work we may achieve better results.

3.4 Regression

Things are easier concerning regression, since we are able to normalize our values and diminish the influence of outliers. This however, does not mean that with more data from the SDSS we would achieve better results. In *table3* and *table4* we provide regression results per model for the SDSS.

From *table3* we can observe that the word lstm model achieves the best performance for answer size prediction with the word cnn and word tfidf following. We notice that word models perform better in this problem than character ones, something that didn't happen in the original paper. It is possible that there was a difference in the implementation of the different word and/or character vocabulary and in the model training procedure that impacts the character model performance for the SDSS dataset. An easy explanation would be the difference in dataset size.

Now from *table4* it seems that the ccnn achieves the best cpu prediction performance. We notice that both the MSE and the Max Error are very low. It is possible that given more data i.e. the dataset used in the original paper, that the error values may change. This is supported by the fact that in the original paper their best error score was 0.0441 which is worse than ours by a factor of 10^3. Thus, it is not safe to directly compare the cpu time problem of our approach to the original due to our subset of query statements out of the whole SDSS being smaller and more simple. Also, we should note again that since we are used the original paper as a starting point we might have differences in the value handling of the regression labels.

We do the same in *table5* and *table6* for the SQLShare dataset.

As seen in the *table5* results for the SQLShare data are similar with those for the SDSS. The wlstm is the best for predicting cpu time and outperforms all other models by

Model	Train Time (sec)	$F_{success}$	$F_{nonsevere}$	F_{severe}	Accuracy	missed values
mfreq	5	0.9996	0	0	0.9992	69
ctfidf	211.57	0.9996	0	0	0.9992	69
cccn	39.11	0.9996	0.56	0	0.9992	68
clstm	3330.66	0.9996	0	0	0.9992	69
wtfidf	134.61	0.9996	0.08	0	0.9992	68
wcnn	47.17	0.9997	**0.58**	0	0.9994	53
wlstm	4311.29	**0.9999**	0.32	0	**0.9997**	**23**

Table 2. Classification results displaying the training time for each model, the F1 score for each one of the different error classes, the accuracy score of our models and the exact number of missed values for our models to put everything into perspective. Please note that c stands for character and w for word representation model respectively. mfreq is a baseline model where all predicted values are equaled with the most frequent value from our data.

Model	Train Time (sec)	MSE	Max Error
median	**5**	0.44	17.32
ctfidf	701	0.16	15.29
ccnn	44	1.03	16.34
clstm	3000	0.42	17.17
wtfidf	341	0.18	16.53
wcnn	94	0.29	16.54
wlstm	2240	**0.12**	**14.04**

Table 3. SDSS Regression answer size results displaying the training time for each model, the MSE and the Max error.

Model	Train Time (sec)	MSE	Max Error
median	**6**	0.000063	0.39
ctfidf	660	0.000062	0.39
ccnn	133	**0.000018**	**0.22**
clstm	2800	0.00028	1.39
wtfidf	414	0,00059	0.38
wcnn	94	0,00024	1.38
wlstm	2328	0.00065	2.40

Table 4. SDSS Regression cpu time results displaying the training time for each model, the MSE and the Max error.

Model	Train Time (sec)	MSE	Max Error
median	**2**	0.00017	0.32
ctfidf	81	0.00010	0.33
ccnn	133	**0.00015**	**0.37**
clstm	1709	0.00012	0.37
wtfidf	67	0.00010	0.33
wcnn	39	0.00023	0.42
wlstm	1265	0.000047	0.29

Table 5. SQLShare Regression cpu time results displaying the training time for each model, the MSE and the Max error.

approximately a factor of 10. For this implementation we will not directly compare our results to the original since these results came out of data labels (CPU time) scaled explicitly between 0 and 1 (we used Min-Max normalisation for this problem).

We notice in both datasets, that our simple baselines do provide decent results compered to the more computational heavy methods. So for users with less complex databases or with little computational ability to spare, a simple solution would be to provide the user with some feedback about the query they posed using a baseline model as a starting point. This may prove tricky though and may not scale well when the database or the workload scale up both in size and complexity.

3.5 Discussion

While Accuracy, F1 score, MSE and max error are the main evaluation metrics we used, one should also take into account the training requirements of each method. For example we observed that objectively the most efficient methods include a neural network model implementation, even though this implementation requires exponentially more time to train than the traditional methods in a regular computer. Even with GPUs we can see in the tables that the time required for a 3-layered lstm to train is more than an hour. So having to work with a bigger workload would forbid the use of a simple computer for training purposes. The use of cloud seems to be a viable option but only with a fast and stable internet connection as well as an abundance of GPU and CPU resources. All in all, it seems that we achieved the desirable performance given the various implementation details.

In more detail, one could empirically expect that the lstm and cnn would perform better than their traditional counterparts due to their ability to hold and process spatiotemporal information. This is also the reason for the cnn's use in computer vision related problems. The cnn is able to exploit the parallelization abilities of the GPU more so than the lstm and train faster. It is mentioned in [1] paper that empirically as the available queries increase, more rare words are introduced, thus the word level models will have decreased performance in comparison to character level ones. This was not the case in my implementation and one could state that although this is certainly true for any other NLP model this may not be the case for an SQL-NLP model, since there is an upper bound to the number of words that may be introduced, which depends on a number of metrics such as the number of table names in the database schema.

All in all our results where comparable and in certain cases (overall classification accuracy) better than those of the original paper. Notable performances where also achieved in the word models for answer size and cpu time prediction problems, with the latter achieving very low error scores.

4 Conclusions

We have showcased hereby, our attempt at reverse engineering and extending the paper on facilitating SQL composition and analysis [1]. Credits for the original work must be given to the authors, who presented a novel approach on exploiting database workloads for the creation of facilitating SQL query composers. During the implementation of this paper we had to work based on our own axioms and theorems derived from the explanation of the various methodologies. This means that we our effort may have resulted to another version of the facilitation pipeline, imbued with our own personal touch and not the original.

Notable mentions from our implementation include the SQL parsing - syntactic analysis tool we created for python utilizing the sqlparser package.

Of course the problem of query composition facilitation is not an easy one and there are still many possible extensions to the existing methodologies. An example would be the creation of a SQL query workload dataset comprised of many different workloads from different databases working on different DBMSs. This great query workload may then be used for more challenging query property prediction problems. It would also be great to have greater availability of session data and/or more sensitive database usage data concerning the queries posed on a database (of course with respect to each persons' privacy) so as to be able to train more complex models.

All in all, it is worth mentioning that the unavailability of real world query workloads is a great problem for researchers wanting to experiment in the field of query facilitation. It seems that this field is more likely to develop privately, by corporations or people who are able to maintain their own massive database with a respective massive query workload.

References

[1] Zainab Zolaktaf, Mostafa Milani, and Rachel Pottinger. 2020. Facilitating SQL Query Composition and Analysis. In <i>Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data</i> (<i>SIGMOD'20</i>). Association for Computing Machinery, New York, NY, USA, 209–224. DOI:https://doi.org/10.1145/3318464.3380602

[2] Ramos, Juan. (2003). Using TF-IDF to determine word relevance in document queries.

[3] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-Based Query Performance Modeling and Prediction. In ICDE. 390–401.

[4] M Jordan Raddick, Ani R Thakar, Alexander S Szalay, and Rafael DC Santos. 2014. Ten Years of SkyServer I: Tracking Web and SQL eScience Usage. Computing in Science Engineering 16, 4 (2014), 22–31.

[5] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In SIGMOD. 281–293.

[6] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? PVLDB 9, 3 (2015), 204–215

[7] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-Aware Neural Language Models. In AAAI. 2741–2749.

[8] Van Rossum, G., Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.

[9] https://github.com/andialbrecht/sqlparse

[10] https://github.com/pyparsing/pyparsing

[11] https://github.com/timogasda

[12] Abadi M. et al. (2016) TensorFlow: A system for Large-Scale Maching Learning. In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[13] Peter J Huber et al. 1964. Robust estimation of a location parameter. The Annals of Mathematical Statistics 35, 1 (1964), 73–101.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. JMLR 12 (2011), 2825–2830.

[15] F. Chollet, *keras-team/keras. [online] GitHub*. Available at: https://github.com/fchollet/keras [Accessed 17 June 2019].