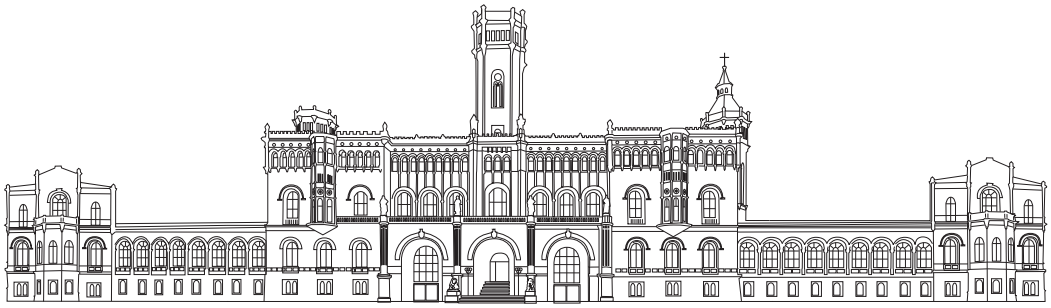


LEIBNIZ UNIVERSITÄT HANNOVER

FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK
INSTITUT FÜR PRAKTISCHE INFORMATIK

Improving Primary Key Detection with Machine Learning

Bachelor Thesis

submitted by

JANEK PRANGE

on 03.07.2022

First Examiner	:	Prof. Dr. Ziawasch Abedjan
Second Examiner	:	Prof. Dr. Sören Auer
Supervisor	:	Prof. Dr. Ziawasch Abedjan

DECLARATION

I hereby affirm that I have completed this work without the help of third parties and only with the sources and aids indicated. All passages that were taken from the sources, either verbatim or in terms of content, have been marked as such. This work has not yet been submitted to any examination authority in the same or a similar form.

Hannover, 03.07.2022

Janek Prange

ABSTRACT

The discovery of primary key candidates and unique columns in general is an important problem. The current algorithms to find unique columns are not fast enough for large tables as their runtime depends on the size of the table.

In this thesis, I propose a new method to detect unique columns. It uses a machine learning model to predict which columns are probably unique columns. Only positive guesses will be checked for duplicates. With this method, the time it takes to detect all unique columns is reduced by more than 60 % for large tables with more than 10 000 000 rows.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AI Artificial Intelligence

1

MOTIVATION

Tables are an increasingly important way to store and organize data. Be it SQL databases, No-SQL databases or simple Excel tables, the ability to organize data and present it concisely is at the core of many projects.

In the best case, the structure of the data set has already been considered before creating them in order to clarify which data type can occur in a column, whether values can be left empty and which columns act as primary keys.

However, there are cases where this did not happen for various reasons; either because the data had to be saved quickly and there was no capacity for reasonable planning, or because the data is no longer available in the original format.

In this case, it is an important task to recover the missing metadata as accurately as possible. While information such as the data type and the existence of empty values is comparatively easy to find within a linear runtime, identifying primary keys is more difficult.

One challenge is to distinguish between primary key candidates, which are characterized by the fact that they do not contain duplicates, and practically usable primary keys. For example, even if a column containing comments from users does not contain duplicates, text that is on average 100 characters long is not really suitable as a key.

Another problem is the runtime needed to determine if a column contains duplicates. For very large tables with several million rows, this step can take some time.

2

FUNDAMENTAL KNOWLEDGE

2.1 MACHINE LEARNING

Machine learning is a part of the field of Artificial Intelligence (AI). The focus is on extracting information from large amounts of data, with algorithms gradually improving themselves to mimic human learning [2].

The field of machine learning is itself divided into two main parts, "Classical learning" and "Deep learning"[7].

2.1.1 *Deep Learning and Neural Networks*

Deep Learning includes various techniques that use neural networks. In the simplest form, neural networks consist of different so-called nodes, which are arranged as layers[9].

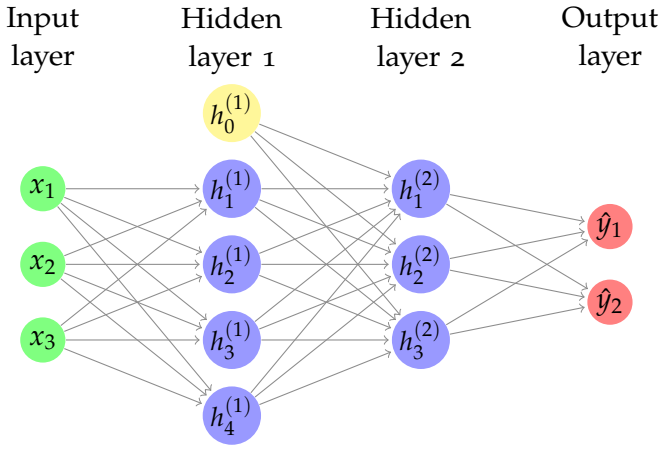
The first layer, also known as the "input layer", is the layer the input vector is applied to. If the input exceeds a certain value, the neuron is activated, i.e. it passes on an output.

In the next layers, which are called "hidden layers", the outputs from all the nodes of the previous layer and a "bias layer" are computed in each node and the output is passed on. The bias layer consists of a single value that influences all nodes on a layer.

Finally, the last layer outputs the result of the computation. How strong each previous node influences the value of the current node is what the neural network is trained on.

Neural networks are particularly good at recognizing patterns in large unordered data, such as images, videos, and audio tracks. Examples of this include facial recognition or verifying hand signatures[5].

Figure 2.1: A simple neural network with three input nodes and two output nodes.



2.1.2 *Classical Learning*

Algorithms in the field of classical machine learning are largely based on statistical or probabilistic methods, which originated as pattern recognition in the 1960s. It has the advantage of being a fast method that does not require much computing power compared to deep learning algorithms[6].

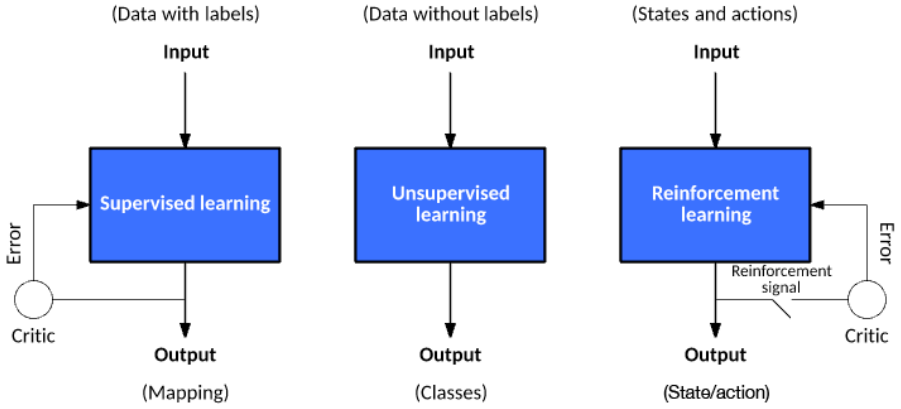
2.1.3 *Categories of Machine Learning*

Both deep learning as well as classical learning are dived into different categories based on the kind of task and trainingsdata.

Unsupervised learning is used on unlabeled data. Its main use is to organize the data or to make it more manageable. Tasks in this domain include clustering, which categorizes data based on similar data points, or assoziation, which tries to find relationships between variables in a dataset[1].

Supervised learning on the other hand makes use of labeled data. This comes with the disadvantage that the structure of the data as well as the desired results have to be known[6]. Even though it increases the effort of the preparation of the trainingsdata, it also raises the accuracy of the model since it is trained to produce only the desired result.

Figure 2.2: This picture shows the different types of machine learning and the way they are trained[4].



2.2 NAIVE ALGORITHMS FOR FINDING PRIMARY KEY CANDIDATES

There are fundamentally two ways of finding primary key candidates. The first one works by traversing a column, forming the hash value of each row and saving this hash value in a suitable data structure. If the hash value of a row is already present, the algorithm aborts because it is clear that the column is not unique.

The other method operates by sorting the column in a first step and then comparing each row with its neighbors. If two rows are the same, the column is not unique.

The naive algorithm that is used for the experiments in Chapter 5 is a sorting based method. However, it has the specialty of aborting if one of the values is None/Null since it is not unique for the purposes of this thesis as explained in Section 3.

The reason the hashing based method was not used for the experiments was to be able to reliably carry out the efficiency experiments. Although the sorting based method is less efficient, it enables the experiments to be set up in a way that enables the machine learning model to make the correct prediction without increasing the speed of the naive algorithm.

2.3 USED PACKAGES AND LIBRARIES

2.3.1 *Pandas*

2.3.2 *Scikit-Learn and Auto-Sklearn*

3

PROBLEM STATEMENT

Primary key candidates are columns in a table which do not have any duplicate values and are therefore suited to identify a row in the table. In this thesis, a column that contains only distinct values, one of which is a None/Null value, is considered non-unique as it can not be used as a primary key.

Runtime of naive algorithm (sorting and hashing based)

No problem with small tables, but problem with tables with more than a few million rows

4

PROPOSED METHOD

4.1 OVERVIEW

In this thesis I present a method to increase the efficiency of finding unique columns in a table. The method is based on a machine learning model which uses the first few rows of a row to guess if it will have any duplicate values anywhere in the column. Each positive guess will subsequently be validated using a conventional naive method.

The proposed method works in three steps. First, the features are extracted from the first rows of the table. After that, the model tries to predict the existence of duplicate values from the features. Finally, the columns which are unique according to the model are checked with a naive method.

4.2 EXTRACTING FEATURES

The feature extraction is necessary as explained in Section 2.1 to get a table like Table 4.1...

The code in Listing 4.1 prepares each column to produce such a feature table.

The feature extraction of the proposed method works in different steps which are executed on after the other. First all columns which contain duplicate values in the first rows are sorted out by setting the feature "Duplicates" to 1 and all other features to 0.

The remaining rows are checked in order to determine whether they are sorted or not. During this step, it is possible that an error occurs because two values in the column can not be compared. This is mostly the case if there is an empty/None value in the column.

Next, a distinction is made between the different types of values. If the column contains only boolean values, the feature "Data Type" is

Table 4.1: An example for a feature table which is used by the model proposed in Chapter 4. The following values are possible in the column "Data Type": 0: The inspected rows contain a duplicate value. 1: The column contains only integer. 2: The column contains only text. 3: The column contains only boolean values (which does rarely occur without containing duplicates). 4: The column contains a mix of different types.

Duplicates	Data Type	Sorted	Min. value	Max. value	Mean	Std. Deviation	Avg. string length	Min. string length	Max. string length
1	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0	1	1	7.0	562.0	290.706	187.489	0.0	0.0	0.0
0	1	1	1.0	62.0	28.941	21.496	0.0	0.0	0.0
0	1	1	611.0	946.0	789.118	107.904	0.0	0.0	0.0
0	1	1	1.0	17.0	9.0	5.050	0.0	0.0	0.0
0	2	0	0.0	0.0	0.0	0.0	86.25	78.0	92.0
0	2	0	0.0	0.0	0.0	0.0	78.75	60.0	96.0
0	2	0	0.0	0.0	0.0	0.0	123.032	51.0	296.0
0	2	0	0.0	0.0	0.0	0.0	94.130	46.0	204.0
0	3	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0	4	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

set to 3, all other features stay on 0. Although it is very unlikely that a column contains exclusively boolean values without any duplicate values, there are tables in the gittables dataset to which this applies.

Numeric values

String values, mixed types

4.3 TRAINING THE MODEL

How was the model trained? What settings where used for the training and why?

Listing 4.1: This code shows how a column is prepared for the model. This process is repeated for each row; the result forms the feature table. The variable `column` contains the first n rows of the column where n is the input size of the model.

```
if has_duplicates(column):
  return [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result = [0, 0, 0]
# check if entries are sorted
try:
  if all(column[i+1] >= column[i] for i in range(len(column)-1)):
      result[2] = 1
  if all(column[i+1] <= column[i] for i in range(len(column)-1)):
      result[2] = 1
except TypeError:
  # mostly this means the column contains None/NaN values
  pass
if only_bool(column):
  result[1] = 3
  result += [0, 0, 0, 0, 0, 0, 0]
  return result
if only_numeric(column):
  result[1] = 1
  result += [min_value(column), max_value(column),
             mean_value(column), std_deviation(column)]
  # values for strings
  result += [0, 0, 0]
  return result
result[1] = 2
# values for numbers
result += [0, 0, 0, 0]
try:
    length_list = []
    for value in column:
        if isinstance(value, str):
            length_list.append(len(value))
        else:
            raise ValueError("Not a String")
    if len(length_list) == 0:
        average = 0
    else:
        average = sum(length_list)/len(length_list)
    minimum = min(length_list)
    maximum = max(length_list)
    result += [average, minimum, maximum]
except ValueError:
    result[1] = 4  # mixed column, mostly None/NaN value
    result += [0, 0, 0]
return result
```

5

EXPERIMENTS

5.1 EXPERIMENT SETUP

5.1.1 *Hardware*

The following experiments where conducted on a server of the university. The machine uses 514 GiB working memory and an `AMD EPYC 7702P 64-Core` processor. A graphic card was not used in the experiments.

5.1.2 *Software*

5.1.3 *Metrics*

5.2 CORRECTNESS

The correctness of the model is probably the most important metric to determine its usability. While a false positive is not a major problem because each positive guess is verified (see Chapter 4), a false negative will mean that a primary key candidate gets ignored.

In this section, different experiments will be conducted to determine which parameters are the best to train the model. Additionally, in Section 5.2.5 the column which led to false guesses by the model will be examined.

5.2.1 *Experiment Data*

The experiments where performed on the GitTables dataset, which is a large corpus of relational tables extracted from CSV files in GitHub[3].

The training dataset is used to train every model which was tested in the following experiments. It is a subset of the GitTables dataset with 10 000 tables. Every table has to have at least 100 rows and 3 columns to ensure the table preparation described in Section 4.1 does not become the naive algorithm with extra steps. The tables where selected by traversing the GitTables dataset and skipping tables which are to small.

The test dataset was generated the same way as the training dataset. By traversing the GitTables and skipping every table which is too small or part of the training dataset a collection of 5000 tables with 57 211 columns and an average of 184 rows was generated. It was used for every experiment except the one in Section 5.2.2 where a dataset with 30 000 tables with 307 030 columns and an average of 277 rows was used.

5.2.2 *Comparing models with different input sizes*

As described in Section 4.2, the proposed method extracts features from the first rows of a table and uses a machine learning model to predict if a column has duplicate values based on those features.

In this experiment, I compare different models which use 5, 10, 20 and 50 rows to extract features. A model with an input size larger than 50 rows was not feasible as the tables used for training and testing had a minimum size of 100 rows. With a bigger input, the preparation would end up working like the naive algorithm with an extra step.

Each model was trained for 5 hours. During the experiment, the test dataset with 5000 tables that was described in Section 5.2.1 was used to test each model.

Table 5.1 shows the results of this experiment, which demonstrate that the input size has a large influence on the quality of the prediction. While none of the models has any false negative guesses, the ratio of false positive guesses decreases with an increasing input size.

This experiment shows that an increase in the input size of the model does have a great impact on the number of false positive guesses. However, since each positive guess by the model is verified using the naive algorithm, the number of false guesses has no effect on the quality of the final prediction of the proposed method. What is effected by it is the efficiency of the method; this is explored further in Section 5.3.4.

Another important finding of this experiment is that even with a small input size of only 5 rows, the model has not made any false negative guesses. As the negative guesses of the model are not checked, it is important for them to be correct to ensure the overall correctness of the proposed method.

Table 5.1: The result of the correctness experiment comparing models with different input sizes. Each of the models was trained for 5 hours the training dataset described in Section 5.2.1. The experiment was conducted on the test dataset with 30 000 tables.

Model Input Size	Accuracy	Precision	Recall	F1
5	0.704	0.269	1.0	0.424
10	0.833	0.394	1.0	0.566
20	0.910	0.547	1.0	0.707
50	0.970	0.783	1.0	0.878

5.2.3 *Altering the training time*

The Auto-Sklearn library, which was used to train the machine learning models, automatically searches for the best learning algorithm and optimizes it, as was described in Section 2.3. Since this process takes time to run, the theory is that the performance of the model increases with higher training time.

In this experiment, different models with an input size of 10 rows are being trained for different amounts of time. Each model is trained on 10 000 tables with at least 100 rows and 3 columns. Subsequently, each of the models is tested on 5000 different tables.

The Table 5.2 presents the results of this experiment. With a training time of 1 or 2 min the performance of the machine learning model is indeed slightly worse as there are some false positive guesses.

However, apart from producing false negative guesses, the models with the two shortest training times seem to performed slightly better

than most of the other models as their accuracy and precision is higher. Additionally, the model with the best performance is not the one with the highest training time but the one a training time of 120 min.

This experiment very clearly demonstrates that the machine learning model for this task does not need a long training time. Furthermore, it shows that there might be advantageous to train a model multiple times and compare them to find the best performing, since the performance is not strongly correlated to the training time.

Table 5.2: The result of the correctness experiment comparing models which were trained for different amounts of time on the training dataset described in Section 5.2.1. The experiment was conducted on the test dataset with 5000 tables.

Training Time	Accuracy	Precision	Recall	F1
1 min	0.847	0.369	0.999	0.540
2 min	0.846	0.367	0.989	0.536
5 min	0.823	0.337	1.0	0.504
10 min	0.823	0.337	1.0	0.504
20 min	0.823	0.337	1.0	0.504
40 min	0.823	0.337	1.0	0.504
60 min	0.823	0.337	1.0	0.504
120 min	0.847	0.369	1.0	0.540
180 min	0.873	0.414	0.999	0.585
300 min	0.823	0.337	1.0	0.504

5.2.4 *Summary*

5.2.5 *Examine columns which led to false guesses*

The greatest possible weakness of the of the proposed method are false guesses, as false positive guesses lead to a reduced efficiency and false negative guesses result in primary key candidates being ignored. It is therefore important to examine the columns which lead to false guesses to improve the model if possible.

False positive guesses occur very often as the model is trained to avoid any false negative guesses. The experiment in Section 5.2.2 has shown that depending on the input size between 18 % and 77 % of the positive guesses are false positives.

The false guesses are unfortunately mostly unavoidable as they are caused by empty cells which are located after the input rows of the model. As the column would be a primary key candidate without these missing values, there is nothing that can be changed to improve the correctness of the model in this case.

Another example for a column leading to a false positive guess is one containing the name of authors. For the model, this column contains short strings which do not have any duplicates in the first rows.

False negatives

5.3 EFFICIENCY

Next to the correctness, the efficiency of the proposed method is the most important metric to determine its feasibility. The main question is if or from what table size the proposed method is faster than the naive method. This becomes even more interesting as each positive guess of the model has to be verified using the naive algorithm to increase the accuracy.

The following experiments explore the efficiency of the proposed method in comparison to the naive algorithm and which parameters have the greatest influence on it.

5.3.1 *Experiment Data*

The experiments in this section were conducted on a set of generated tables to control the size of the table as well as the number of unique and non-unique columns. A small example of such a table can be seen in Table 5.3.

The generated tables each have 10 columns and between 100 and 100 000 000 columns. To ensure the correct prediction by the model, the columns where generated in a specific way. The unique columns are evenly incrementing for the first 50 rows, while the first two rows of the non-unique columns contain the same value. The rest of each column contains distinct incrementing values which are mixed up to increase the time the sorting based naive algorithm takes to find unique columns.

Table 5.3: A table generated for the efficiency experiment. The columns 0 and 1 do not contain any duplicates, the columns 2, 3 and 4 do. To guarantee that the model guesses the unique and non-unique columns correctly, the unique columns are evenly incrementing for the first 50 rows, while the duplicate value of the non-unique columns is in the first two rows.

Index	Column 0	Column 1	Column 2	Column 3	Column 4
0	0	0	100	100	100
1	1	1	100	100	100
2	2	2	93	93	93
3	3	3	45	45	45
⋮	⋮	⋮	⋮	⋮	⋮
48	48	48	89	89	89
49	49	49	39	39	39
50	91	91	60	60	60
51	77	77	49	49	49

5.3.2 *Base experiment*

The first experiment explores the efficiency of the proposed method compared to the naive algorithm. The generated tables that were used contained 3 unique and 7 non-unique columns.

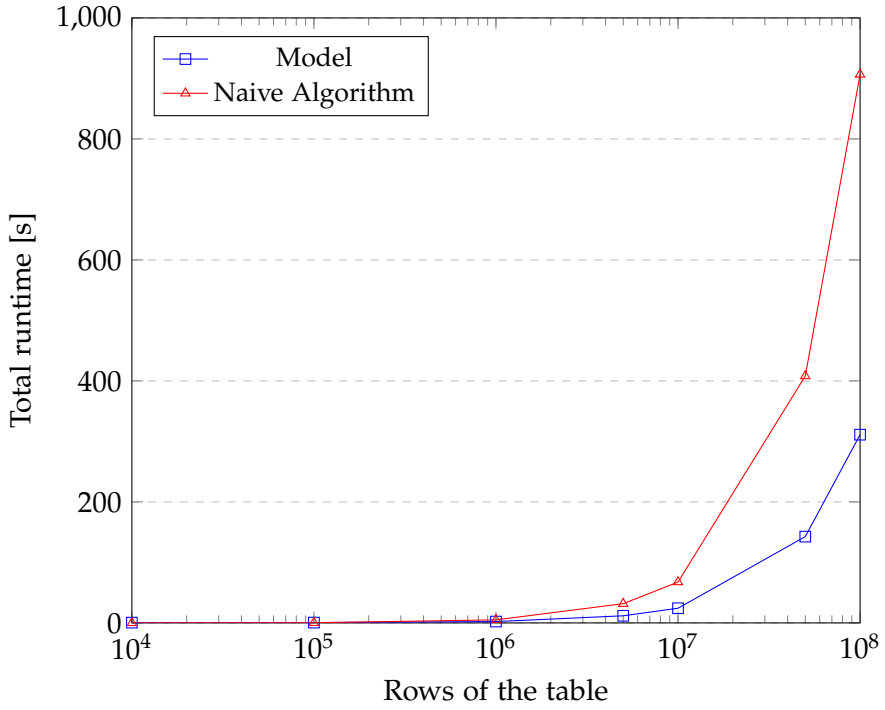
Figure 5.1 and Table 5.4 show that for tables with up to 100 000 rows, the naive algorithm takes only a fraction of a second and is therefore faster than the proposed machine learning model. However, since the model takes a roughly constant time of half a second to compute its prediction, it becomes faster as the table size surpasses one million rows.

The column "Model: Validation" in Table 5.4 additionally illustrates that the validation time of the proposed method is proportional to the number of positive guesses by the model. This highlights the importance of a high accuracy as explained in Section 5.2.2, because each false positive guess unnecessarily increases the time it takes to validate the guesses and therefore decreases the efficiency.

In conclusion, this experiment illustrates that for large tables loading the dataset and checking the columns for duplicates with the naive algorithm takes the most time. Ways to reduce the loading time will be explored in Section 5.3.3. While a more efficient naive algorithm is

not part of this thesis, Section 5.2.2 and 5.2.5 deal with the question of how to decrease the number of false positive guesses.

Figure 5.1: This plot shows the total runtime of the naive algorithm and the proposed method to compute the unique columns as can be seen in Table 5.4. The tables were saved in a CSV format.



5.3.3 *Reduce loading times*

While CSV files are very easy to use, they are not meant to efficiently store large quantities of data. A file format which is substantially more suitable to handle large datasets is the parquet format[8].

It achieves this through the use of various features such as column wise compression, which tends to be more efficient since the values in the same column are usually very similar. This has the additional benefit of enabling the algorithm to only read the required columns which may decrease I/O as only positive guesses need to be loaded for the validation.

Another advantageous property of this format is the concept of row groups, which ensure that a batch of rows is being saved together and can therefore be read together too. This makes it possible to read just the first row group and use these rows as an input for the model.

Table 5.4: The result of the efficiency experiment with a generated table with 30 % unique columns in a CSV file format. The experiment was conducted on a model with an input size of 10 rows on tables generated with 10 columns.

Rows	Model: Loading	Model: Computing	Model: Validation	Model: Total	Naive: Loading	Naive: Computing	Naive: Total
100	0.001	1.082	0.000	1.084	0.004	0.000	0.004
1000	0.002	0.449	0.001	0.451	0.002	0.002	0.004
10 000	0.006	0.446	0.007	0.459	0.005	0.022	0.026
100 000	0.045	0.452	0.076	0.574	0.045	0.257	0.302
1 000 000	0.434	0.454	1.369	2.260	0.427	4.623	5.050
5 000 000	2.476	0.451	8.824	11.771	2.456	29.341	31.797
10 000 000	4.655	0.446	19.116	24.258	4.606	62.935	67.541
50 000 000	27.298	0.458	114.694	142.650	27.075	381.059	408.133
100 000 000	52.429	0.444	257.904	311.173	52.230	854.417	906.646

The Table 5.5 shows the result of the base experiment from Section 5.3.2 repeated with tables generated as parquet files. While the computing time for the model and the naive algorithm remain roughly equal compared to Table 5.4, the loading time is decreased significantly for large tables.

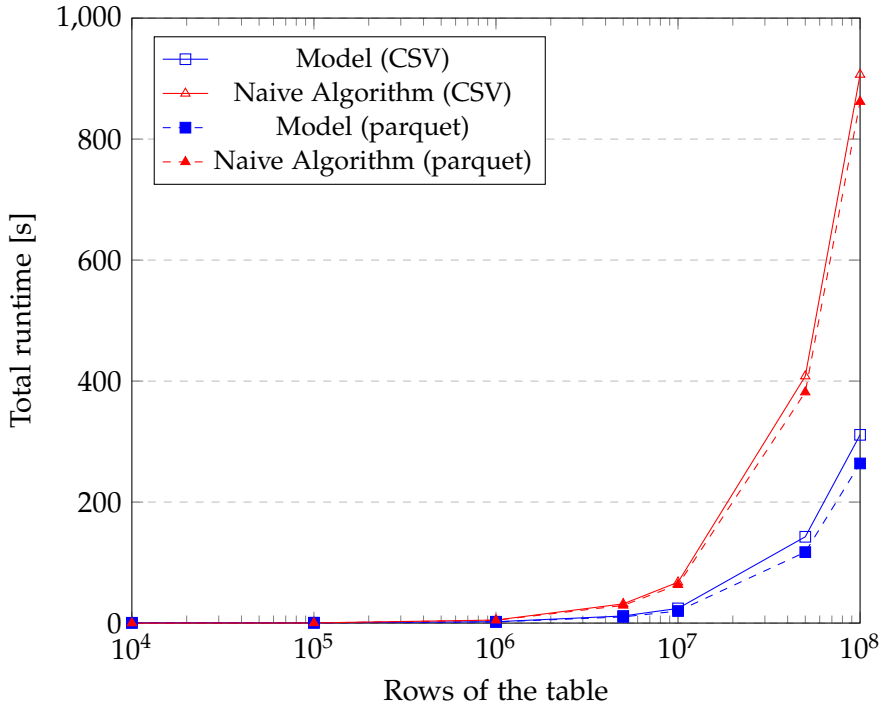
Table 5.6 presents the result for the experiment using the advantages of the file format by loading only the necessary rows and columns. This leads to two loading times for the model. The first time only the first row group is being loaded while the second time only the columns which are unique according to the model are loaded. However, this does not make any difference except for the largest table and even then the total time is hardly changing.

In summary, while the reduced loading time does make a notable difference, it is not very large compared to the efficiency gain through the use of the proposed method, as can be seen in Figure 5.2. This could change, however, if the file reading speed would be slower, for example because the data had to be read over the internet. In this case, reading only the necessary rows and columns and thus decreasing I/O further could make a larger difference too.

5.3.4 *Changing the ratio of unique columns*

The last variable that has an impact on the runtime of the model is the percentage of unique columns in the table. Since every positive

Figure 5.2: This plot shows runtime of the proposed method and the naive algorithm as they search for unique columns in tables of different sizes. I compare the runtime reading the data from CSV files and reading it from parquet files.



prediction by the model has to be verified using the naive algorithm, the total runtime increases the more unique columns the model detects. The number of unique columns predicted by the model correlates roughly with the number of actual unique columns in the table.

In this experiment, a model with an input size of 10 rows is used on 4 tables, which are saved as parquet files. The difference between these tables is the percentage of unique columns, which range from 60 % to 90 %. Each table has 100 000 000 rows and 10 columns.

Table 5.7 shows that nearly every step of the process takes the same amount of time, just the validation step is proportional to the number of unique columns.

In the GitTables dataset, which is used in the correctness experiment, the ratio of unique columns is 10 %. The positive guesses of the model are quite a bit higher since its priority is to avoid false negatives, not false positives. Still, the share of positive guesses during tests on the GitTables dataset is around 30 %, which is low enough to be a clear improvement over the naive algorithm given large enough tables.

Table 5.5: The result of the efficiency experiment on a generated table with 30 % unique columns in a parquet file format. The experiments was conducted on a model with an input size of 10 rows on tables generated with 10 columns.

Rows	Model: Loading	Model: Computing	Model: Validation	Model: Total	Naive: Loading	Naive: Computing	Naive: Total
100	0.004	0.454	0.000	0.458	0.006	0.001	0.006
1000	0.003	0.445	0.001	0.448	0.003	0.002	0.005
10 000	0.004	0.446	0.007	0.457	0.004	0.021	0.025
100 000	0.010	0.447	0.077	0.534	0.010	0.246	0.256
1 000 000	0.040	0.462	1.396	1.902	0.047	4.618	4.665
5 000 000	0.197	1.264	8.751	10.233	0.184	29.033	29.216
10 000 000	0.482	0.480	18.934	19.938	0.529	62.746	63.275
50 000 000	2.216	0.987	113.886	117.294	2.205	379.468	381.673
100 000 000	4.473	1.549	257.471	263.898	4.395	857.437	861.833

5.3.5 *Summary*

The experiments in this section show that the proposed method of finding primary key candidates is suitable for some cases. If the tables which will be examined contain mostly viewer than 1 000 000 rows or the ratio of unique columns is high, the model is probably slower than the naive algorithm. On very large tables with 100 000 000 or more rows however the model can significantly improve the overall runtime.

Section 5.3.4 additionally demonstrates that it is important for a high efficiency to decrease the number of false positive guesses by the model as much as possible.

Table 5.6: The result of the efficiency experiment on a generated table with 30 % unique columns in a parquet file format. The experiment was conducted on a model with an input size of 10 rows on tables generated with 10 columns. During the experiment, only the necessary rows and columns were loaded.

Rows	Model: Loading I	Model: Computing	Model: Loading II	Model: Validation	Model: Total	Naive: Loading	Naive: Computing	Naive: Total
100	0.002	0.458	0.003	0.000	0.463	0.004	0.000	0.004
1000	0.002	0.456	0.003	0.001	0.461	0.003	0.002	0.004
10 000	0.002	0.451	0.003	0.007	0.463	0.004	0.020	0.024
100 000	0.009	1.468	0.006	0.081	1.564	0.009	0.247	0.256
1 000 000	0.032	0.455	0.026	1.356	1.869	0.050	4.666	4.716
5 000 000	0.115	0.463	0.106	8.688	9.371	0.195	29.001	29.196
10 000 000	0.243	0.447	0.258	18.961	19.909	0.544	63.122	63.666
50 000 000	1.183	0.447	1.309	114.895	117.834	2.225	379.731	381.956
100 000 000	1.602	0.446	2.425	256.993	261.467	4.437	856.737	861.174

Table 5.7: The result of the efficiency experiment where each table has a size of 100 000 000 rows and 10 columns and is read from a parquet file. The only thing that is changing is the number of unique columns.

Unique Columns	Model: Loading	Model: Computing	Model: Validation	Model: Total	Naive: Loading	Naive: Computing	Naive: Total
4	4.489	1.308	344.742	351.127	4.493	861.111	865.603
3	4.473	1.549	257.471	263.898	4.395	857.437	861.833
2	4.445	1.180	171.984	177.881	4.388	862.440	866.828
1	4.568	0.469	87.070	92.244	4.497	856.509	861.006

6

CONCLUSION

6.1 POSSIBLE APPLICATIONS

Usage as part of a larger algorithm or programm to speed up the process of finding unique columns. This way it is possible to ensure to only use the table if it is large enough for the proposed method to be advantageous.

Maybe an advantage that the method produce possible uniques very fast and verifies them afterwards.

Perhaps the principle of narrowing the columns of the table to inspect closely with a machine learning model can be transferred to other aspects than finding unique columns.

6.2 LIMITATIONS OF THE METHOD

The method only searches for single column uniques. Maybe it is possible to combine it with an existing algorithm for finding multi column uniques and speeding it up.

There is a possibilty that a unique column is classified as a non-unique by the proposed method. To rule out this possibilty, all the columns would have to be inspected with the naive algorithm.

BIBLIOGRAPHY

[1] Julianna Delua. *Supervised vs. Unsupervised Learning: What's the Difference?* Mar. 12, 2021.
URL: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning (visited on 06/23/2022).

[2] IBM Cloud Education. *Machine Learning*. July 15, 2020.
URL: https://www.ibm.com/cloud/learn/machine-learning (visited on 05/18/2022).

[3] Madelon Hulsebos, Çağatay Demiralp, and Paul Groth. "GitTables: A Large-Scale Corpus of Relational Tables." In: *arXiv preprint arXiv:2106.07258* (2021).
URL: https://arxiv.org/abs/2106.07258.

[4] Tim Jones. *Models for machine learning*. Dec. 4, 2017.
URL: https://developer.ibm.com/articles/cc-models-machine-learning/ (visited on 06/23/2022).

[5] Vanshika Kaushik. *8 Applications of Neural Networks*. Aug. 27, 2021.
URL: https://www.analyticssteps.com/blogs/8-applications-neural-networks (visited on 06/21/2022).

[6] Bardh Rushiti. *Classical Machine Learning — Supervised Learning Edition*. Aug. 3, 2020.
URL: https://medium.com/swlh/classical-machine-learning-7efc6674fca1 (visited on 06/21/2022).

[7] Markus Schmitt. *What Is Machine Learning? – A Visual Explanation*.
URL: https://www.datarevenue.com/en-blog/what-is-machine-learning-a-visual-explanation (visited on 06/21/2022).

[8] Deepak Vohra. "Apache Parquet." In: *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Berkeley, CA: Apress, 2016, pp. 325–335. ISBN: 978-1-4842-2199-0. DOI: 10.1007/978-1-4842-2199-0_8.
URL: https://doi.org/10.1007/978-1-4842-2199-0_8.

[9] Tony Yiu. *Understanding Neural Networks*. June 2, 2019.
URL: https://towardsdatascience.com/understanding-neural-networks-19020b758230 (visited on 06/21/2022).