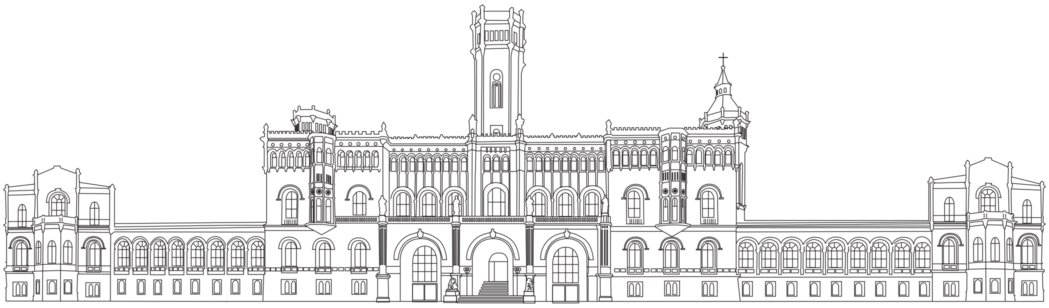


LEIBNIZ UNIVERSITÄT HANNOVER

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

Short summary of the contents in English... a great guide by Kent Beck how to write good abstracts can be found here:

https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AI Artificial Intelligence

1

MOTIVATION

2 PROBLEM STATEMENT

3 FUNDAMENTAL KNOWLEDGE

- Introduce the subjects used in the thesis
- The explanations have to be sufficient for a student after the lecture DBS I

3.1 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 [1].

Machine learning algorithms generally require labeled data to extract information. This labelled data consists mostly of a table where each column corresponds to a feature, which is a specific aspect the algorithm can use to infer some information.

TODO: Deep Learning, unlabeled data (images etc)

3.1.1 *Categories of Machine Learning*

- Supervised
  - Classification
  - Regression
- Unsupervised
- explain the idea behind the feature extraction (it has to be understandable by a student)

3.1.2 *Scikit-Learn and Auto-Sklearn*

3.2 NAIVE ALGORITHMS

Here or in a subsection of Proposed Method?

3.3 USED PACKAGES AND LIBRARIES

3.3.1 *Pandas*

3.3.2 *Scikit-Learn and Auto-Sklearn*

4 PROPOSED METHOD

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 to guess if each column will have any duplicate values. 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.

4.1 EXTRACTED FEATURES

The feature extraction is necessary as explained in Section 3.1

Explain the feature extraction with the function in Listing 4.1.

An example of a table with the extracted features is the Table A.1.

4.2 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 with the hostname `herkules.dbs.uni-hannover.de`. 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 5.2.6 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[2].

For these experiments, only the tables with at least 100 rows and 3 columns where used.

5.2.2 *Comparing models with different input sizes*

Experiment Setup

Result

Conclusion

5.2.3 *Altering the training time*

Experiment Setup

Result

Conclusion

5.2.4 *Testing only non-trivial columns*

Experiment Setup

Result

Conclusion

5.2.5 *Summarized Results*

5.2.6 *Examine columns which led to false guesses*

5.3 EFFICIENCY

Another important metric to determine the feasibility of the machine learning model is the efficiency. The main question is if or from what table size the model is faster than a naive method. This becomes even more interesting as each positive guess of the model has to be verified using the naive algorithm because the model is trained for the highest recall and precision, not accuracy.

5.3.1 *Experiment Data*

The experiment was conducted on a set of generated tables to control for the size of the table as well as the number of unique and non-unique columns, as can be seen in Table 5.1.

For the experiments, tables with different sizes were generated. They each had 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 of the naive algorithm which works by sorting each column.

5.3.2 *Base experiment*

For the first experiment, a model which uses 10 rows as its input was used. The experiment data consisted of 9 tables with 10 columns each and between 100 and 100 000 000 rows. Three of the columns where unique, seven contained duplicates.

Figure 5.1 and Table A.2 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

Table 5.1: A table generated for the efficiency test. 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

model takes a roughly constant time of half a second, it becomes faster as the table size surpasses one million rows.

Particularly noteworthy is the fact that the validation time for the model takes very close to 30 % of the time of the naive algorithm. That is because only columns which are unique according to the model are validated, so in this case three.

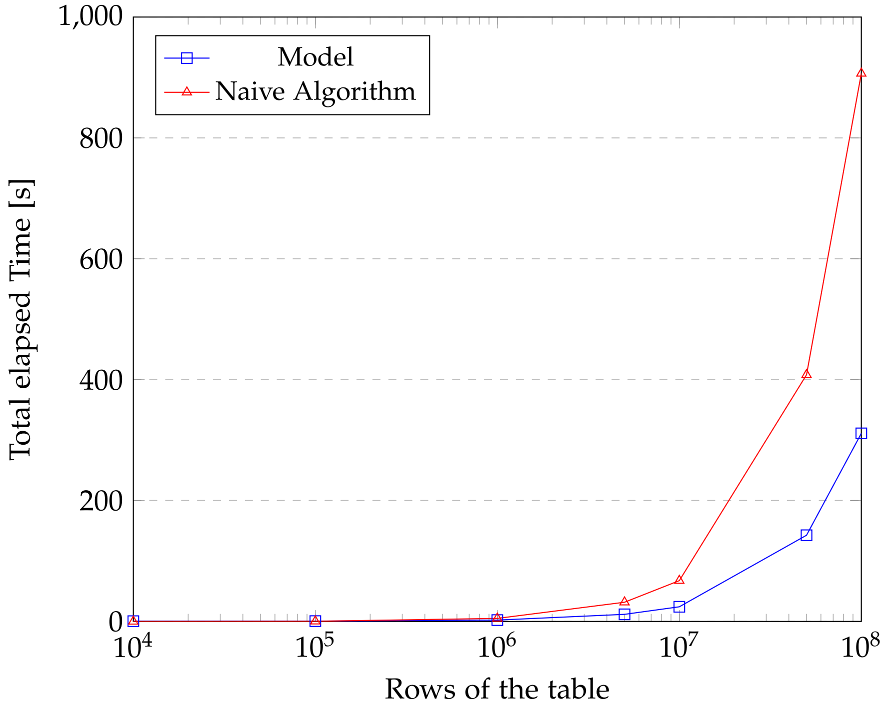
In conclusion, it is clear that for large tables the most time is used loading the dataset and checking the columns for duplicates with the naive algorithm. While a possible improvement of the naive algorithm is not part of this thesis, in Section 5.3.3 two different ways to shorten the loading times will be explored.

5.3.3 *Shorten 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[3].

It achieves this through the use of various features such as column wise compression, which tends to be more efficient since the values

Figure 5.1: A diagram comparing the total time for the naive algorithm and the machine learning model for the base experiment as can be seen in Table A.2.



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.

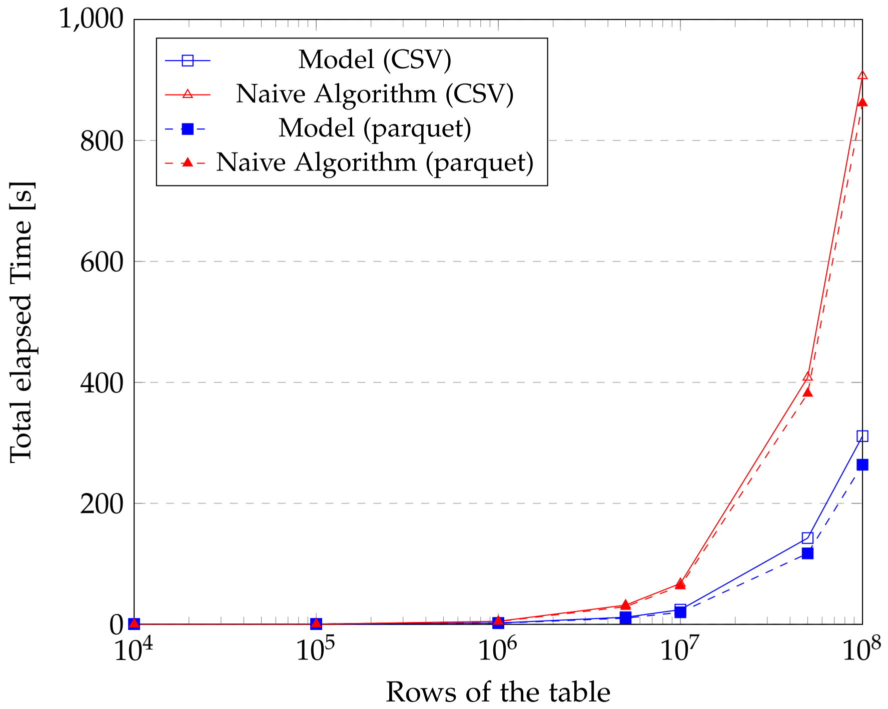
Another advantageous property of this format is the concept of row groups, which ensure that a batch of rows is being saved together and can therefore be read together too. This makes it possible to read just the first row group and use these rows as an input for the model.

The Table A.3 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 A.2, the loading time is decreased significantly for large tables.

Table A.4 presents the result for the experiment using the advantages of the file format by loading only the necessary rows and columns. That is why in this case, there are 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 for validation. However, this does not make any difference except for the largest table and even then the total time is hardly changing.

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

Figure 5.2



5.3.4 *Comparing models with different input sizes*

Short description of what “different input sizes” means (long explanation in earlier section).

Compare times for 70 %.

Conclusion: No large difference, the difference may correlate with the larger file size of the model itself.

5.3.5 *Changing the ratio of unique columns*

The last variable that has an impact on the time it takes the model to predict is the percentage of unique columns in the table. Since every positive guess of the model has to be verified using the naive algorithm, the prediction of the model takes longer the more unique columns the model detects which correlates roughly with the amount 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 A.5 shows that nearly every step of the process takes the same amount of time, just the validation step is proportional to the amount of unique columns.

In the gittables dataset which is used in the correctness test, 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 feasible with large enough tables.

5.3.6 *Summarized results*

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 less 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 improve the time it takes to find all unique columns significantly.

6 CONCLUSION

6.1 POSSIBLE APPLICATIONS

6.2 LIMITATIONS OF THE METHOD

7

RELATED WORK

A

APPENDIX

Table A.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 columns contains integer. 2: The column contains text. 3: The column contains 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

Table A.2: The result of the efficiency test with a generated table with 30 % unique columns in a csv file format. The test was conducted on a model with an input size of 10 rows on tables 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

Table A.3: The result of the efficiency test with a generated table with 30 % unique columns in a parquet file format. The test was conducted on a model with an input size of 10 rows on tables 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

Table A.4: The result of the efficiency test with a generated table with 30 % unique columns in a parquet file format. The test was conducted on a model with an input size of 10 rows on tables with 10 columns.

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

Table A.5: The result of the efficiency test 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	Naiv Tota
4	4.489	1.308	344.742	351.127	4.493	861.111	865.60
3	4.473	1.549	257.471	263.898	4.395	857.437	861.83
2	4.445	1.180	171.984	177.881	4.388	862.440	866.82
1	4.568	0.469	87.070	92.244	4.497	856.509	861.00

BIBLIOGRAPHY

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

[2] 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.

[3] 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.