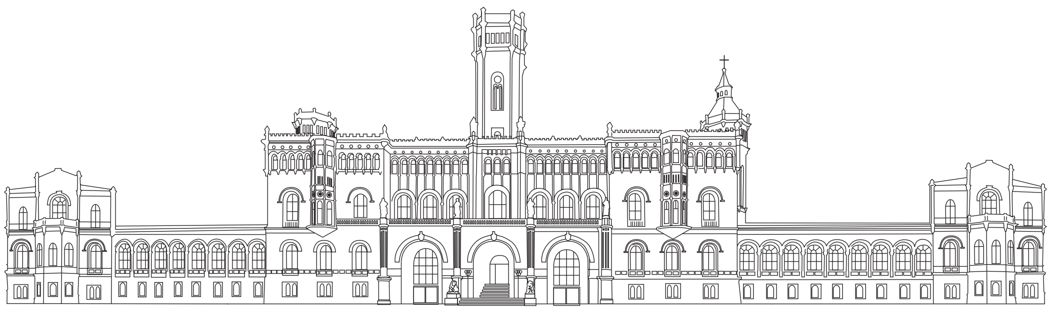


LEIBNIZ UNIVERSITÄT HANNOVER

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 (non-)unique columns in general is an important problem for many use-cases. While primary keys are required to efficiently interact with data stored in a table, various information can be obtained from the uniqueness of a column.

The conventional algorithms to find unique columns work in a linear runtime. This is fine for small to medium-sized tables. In tables with several million rows however the detection of unique columns can take a long time.

In this thesis, I propose a new method to detect unique columns. It uses a machine learning model to predict which columns are unique. Only positive guesses will be checked for duplicates using a conventional algorithm. With this method, the time it takes to detect all unique columns is reduced by more than 65 % for large tables with more than 10 000 000 rows.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 MOTIVATION

Tables are a way to store and organize data that continues to grow in importance. 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 dataset has already been considered before creating it in order to define which data type can occur in each column, whether values can be left empty and which columns act as primary keys.

However, there are cases where this information is not available 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, in particular the primary keys.

One challenge with this task is to distinguish between primary key candidates, which are characterized by the fact that they do not contain duplicates, and practically usable primary keys. An example for this is a column containing comments from users. Even though this column may not contain duplicates, text that is on average 100 characters long is not practically suitable as a key.

Another problem is the time it takes conventional algorithms to determine if a column contains duplicates. The at best linear runtime of these algorithms becomes a problem as the tables become very large and contain several million rows.

The focus of this thesis is to improve the efficiency to detect unique columns, especially for large tables.

2 FUNDAMENTAL KNOWLEDGE

2.1 NAIVE ALGORITHMS FOR FINDING PRIMARY KEY CANDIDATES

There are two basic ways of finding primary key candidates. Both of them function by traversing each column and comparing its values to detect duplicates.

The first method 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. This algorithm runs in a time of $O(n)$ as every value in the column has to be checked once to conclude that it is a unique column.

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. This algorithm requires a runtime of $O(n * \log(n))$ for the sorting operation and a runtime of $O(n)$ to check every value of the column.

2.2 MACHINE LEARNING

Machine learning is a part of the field of Artificial Intelligence. 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 branches, Classical learning and Deep learning [7].

2.2.1 *Deep Learning and Neural Networks*

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

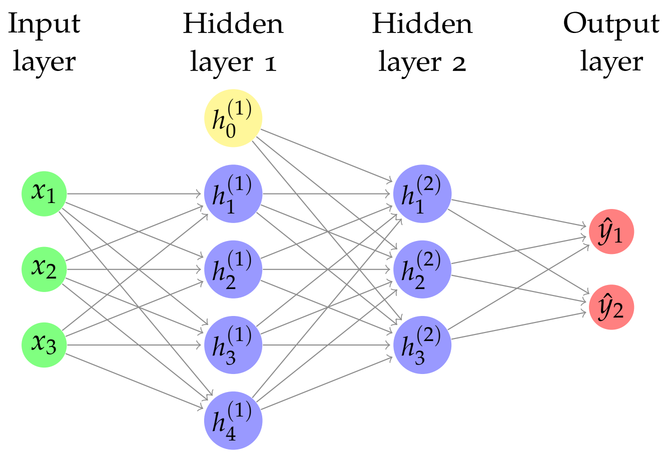
The first layer, also known as the input layer, is the layer the input vector is applied to. If the input exceeds a certain value, the neuron is activated, meaning that it passes on an output.

In the next layers, which are called hidden layers, the outputs from all nodes from the previous layer and the bias are computed in each node after which the output is passed on to the next layer. The bias node 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.



2.2.2 *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 and does not need as much training data compared to deep learning algorithms [6].

Another benefit in using classical learning is that it is easier to understand how the machine learning algorithm made its decision, since it is calculated from statistical dependencies between the data points. This comes with the drawback that a classical learning model is not capable of learning general concepts and applying them to completely new data [2].

2.2.3 *Categories of Machine Learning*

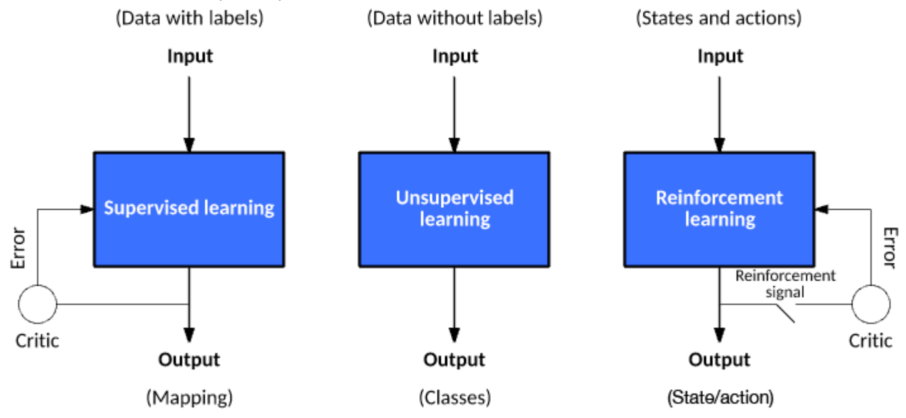
There are different types of machine learning depending on the task and training data the machine learning model has to work with.

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 association, which tries to find relationships between variables in a dataset [1].

Supervised learning on the other hand makes use of labeled data which consists of several features that were selected in advance. 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 training data, it also raises the accuracy of the model since it is trained to produce exactly the desired result.

With reinforcement learning, the machine learning model is trained by interacting with its environment through the context of states. The model learns which action it has to perform to reach its goal for each given state [4].

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



2.3 METRICS

During the training and testing of a machine learning model, its performance is measured with different metrics which indicate how closely the prediction of the machine learning model matches reality. Table 2.1 shows the different possible relations between prediction and reality.

Table 2.1: Definition of true and false negatives and positives

		Prediction	
		Negative	Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

Four metrics exist based on those measurements to calculate the correctness of the machine learning model.

The first of these metrics is accuracy, which is calculated by dividing the number of correct guesses by the total number of guesses. It indicates how close the prediction is to the actual result and is therefore a good general reference point for the correctness of the model.

The second metric, precision, is defined as the number of true positive guesses divided by the total number of positive guesses. It is very well suited for determining the correctness when false positives are associated with a high cost.

Recall is the third metric and defined as the number of positive guesses divided by the number of actual positives. It is a good metric to measure the performance when false negatives are associated with a high cost.

Finally, F1 is calculated by dividing the product of precision and recall by their sum. Just like accuracy, F1 is well suited when both precision and recall are equally important. The difference is that F1 is especially good in cases where the number of positives and negatives is very unbalanced. An example for this is a dataset where 99 % of the actual results are negatives.

3 PROBLEM STATEMENT

Primary Keys are columns which are used to identify rows in a table. They are characterized by the fact that they are unique, meaning they do not have any duplicate values. Columns that do contain duplicate values are non-unique.

For the purposes of this thesis, a column which includes an empty value is considered non-unique even if technically all values are distinct. This constraint ensures every unique column that is detected to be suitable as a primary key, as it is not possible to identify a row based on an empty value.

The conventional naive algorithm to find unique columns has a runtime in $O(n)$ if a hashing based algorithm or $O(\log(n))$ if a sorting based algorithm is used. The problem with these algorithms is that the runtime depends on the number of rows. This leads to a long runtime for very large tables with several million rows.

Additionally, for such large tables memory management becomes a problem as the whole table has to be read at least once and held in memory at least partially.

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 each column to guess if it 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.

The source code of the proposed method as well as the experiments in Chapter 5 can be found in the GitHub repository `https://github.com/LUH-DBS/Prange`.

4.2 FEATURE EXTRACTION

The machine learning model which is used by the proposed method cannot work on the tables directly as it is trained with supervised learning. The model therefore uses a feature table as input, an example of which can be seen in Table 4.1.

The code in Listing 4.1 extracts the features from each column to produce such a feature table. The feature extraction of the proposed method works in different steps which are executed one after the other.

The variable column contains the first n rows of the column in the table where n is the input size of the method.

Table 4.1: An example for a feature table which is used by the model to predict unique columns. 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	0	1.0	62.0	28.941	21.496	0.0	0.0	0.0
0	1	0	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	1	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	1	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

First all columns which contain duplicate values in the first rows are sorted out in line 1 and 2 by setting the feature "Duplicates" to 1, all other features to 0 and returning the row.

In line 5 to 12 the remaining columns 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 cannot be compared. This is mostly the case if there is an empty value in the column. In this case the column is considered not sorted.

In the following lines, a distinction is made between the different types of values.

If the column contains only boolean values, the feature "Data Type" is set to 3 while all other features stay on 0 in line 13 to 16. Although it is very unlikely that a column contains exclusively boolean values without any duplicate values, there are tables in the GitTables dataset that is used during the experiments in Chapter 5 to which this applies.

Line 17 to 23 handle columns that contain exclusively numeric values. The "Data Type" feature is set to 1 in line 18 and the four features specific to numeric values are extracted in line 19 and 20. The three remaining features are set to 0 in line 22.

Finally, the columns that contain exclusively string or a mix of different types are handled in the remaining lines.

First, the features for numeric values are set to 0 in line 25. After that, the average of the length of every item in the column is formed together with the minimum and maximum length in line 35 to 40. At the end the "Data Type" feature is set to 2 in line 41.

If there is any value in the column which is not a string, the "Data Type" feature is set to 4 in line 32 and the string specific features set to 0.

It should be emphasized that the column header is not being extracted as a feature. While this could lead to a better performance in some cases, it would be challenging to encode the various headings in a way that is understandable for the machine learning model.

4.3 USED PACKAGES AND LIBRARIES

For this thesis, the algorithms and experiments where implemented using Python 3.10.2. The model was trained with Auto-Sklearn, a python library that implements automated machine learning on top of the popular library Scikit-learn. To interact with the data, the python library pandas was used as its dataframe structure has useful additional functions as well as a good compatibility to Scikit-learn.

4.4 TRAINING THE MODEL

The machine learning model was trained on the training dataset, which is a subset with 10 000 tables of the GitTables dataset [3] that is used for the correctness test in Section 5.2. Each of the tables has a minimum size of 100 rows and 3 columns.

The features where extracted from each table in the training dataset and saved as the "training_data" table. Additionally, the data was correctly classified using the naive algorithm to form the "training_result" table.

The Listing 4.2 presents the training process. First in line 5 and 6 the feature table and the result table where loaded as pandas dataframes.

Listing 4.1: This code shows how the features are extracted from a column. This process is repeated for each column; the result forms the feature table which is the input for the machine learning model. 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
# values for numbers
result += [0, 0, 0, 0]
length_list = []
for value in column:
    if isinstance(value, str):
        length_list.append(len(value))
    else:
        # mixed column, mostly None/NaN value
        result[1] = 4
        result += [0, 0, 0]
        return result
if len(length_list) == 0:
    average = 0
else:
    average = sum(length_list)/len(length_list)
minimum = min(length_list)
maximum = max(length_list)
result[1] = 2
result += [average, minimum, maximum]
return result
```

In line 8 to 11, the training parameters where selected. The train time was changed for the different experiments in Chapter 5. The model was always trained with the metric recall used to measure its performance, as false negatives have the highest cost for the performance of the model. While there is a cost associated with false positives too, they do not decrease the correctness of the proposed method as each positive guess by the model is verified using a naive algorithm.

Finally, in line 12 the model is trained on the training data. Auto-Sklearn automatically chooses the best of a range of classical learning algorithms.

Listing 4.2: This code shows how the machine learning model is trained using the python library Auto-Sklearn.

```
1  import pandas as pd
2  from autosklearn.classification import AutoSklearnClassifier
3  from autosklearn.metrics import recall
4
5  X = pd.read_csv('training_data.csv')
6  y = pd.read_csv('training_result.csv')
7
8  automl = AutoSklearnClassifier(
9      time_left_for_this_task=train_time,
10     metric=recall,
11 )
12 automl.fit(X, y)
13
14 return automl
```

5 EXPERIMENTS

5.1 EXPERIMENT SETUP

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 necessary for the experiments.

The naive algorithm that is used during the experiments is a sorting based method with the specialty of aborting if one of the values is empty since then the column cannot be used as a primary key.

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

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 for the correctness because each positive guess is verified (see Chapter 4), a false negative leads to a primary key candidate being ignored.

In this section, different experiments will be conducted to determine which parameters have an influence on the correctness of the prediction. Additionally, in Section 5.2.5 the columns which led to false predictions 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]. However, not the whole dataset was used, only a subset of tables which were split into a training and a test dataset.

To train each of the models tested in the following experiments, including the efficiency experiments, the training dataset was used. It is a subset of the GitTables dataset with 10 000 tables that where selected by traversing the GitTables dataset and skipping tables which are too small.

Every table in the training dataset has to have at least 100 rows and 3 columns to ensure the feature extraction described in Section 4.1 does not become the naive algorithm with extra steps. This could be a problem as searching for duplicates with the naive algorithm in the first n rows of each column is part of the feature extraction.

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 apart from the one in Section 5.2.2 where a dataset of 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 any duplicate values based on those features.

In this experiment, I compare different models which use the first 5, 10, 20 and 50 rows of each column 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 only 100 rows.

Each model was trained for 5 hours on the training dataset. During the experiment, the test dataset with 5000 tables 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 have any false negative guesses, the number of false positive guesses decreases with an increasing input size.

An important finding is that even the worst performing model with an input size of 5 rows predicts that only 40 % of all columns are unique. This rate drops to 14 % for the model with an input size of 50 rows, which is very close to the actual ratio of unique columns of about 11 %.

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. Since each positive prediction from the model is verified using the naive algorithm, the number of false positive guesses has no effect on the quality of the final prediction from the proposed method. However, this verification has an influence on 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 prediction. 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 on 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	Predicted Positives
5	0.704	0.269	1.0	0.424	124 332
10	0.833	0.394	1.0	0.566	84 714
20	0.910	0.547	1.0	0.707	61 143
50	0.970	0.783	1.0	0.878	42 705

5.2.3 *Altering the training time*

The library Auto-Sklearn, which was used to train the machine learning models, automatically searches for the best learning algorithm and optimizes it as described in Section 4.3. Since this process takes time to run, theoretically the performance of the model should increase with higher training time.

In this experiment, different models with an input size of 10 rows are being trained for different amounts of time on the training dataset. Subsequently, the experiment is conducted for each model on the test dataset with 5000 tables.

The Table 5.2 presents the results of this experiment. With a training time of one minute the performance of the machine learning model is indeed slightly worse as there are 22 false negative guesses.

However, apart from the false negative guesses, the models with a training time of one and two minutes have a slightly better performance than the other models.

This experiment very clearly demonstrates that the machine learning model does not need a long training time to find unique columns. Furthermore, it shows that it might be advantageous to train a model multiple times and compare them to find the best performing, since a longer training time does not necessarily lead to higher performance.

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

Training Time [min]	Accuracy	Precision	Recall	F1	False Negative Predictions
1	0.847	0.369	0.996	0.538	22
2	0.847	0.369	1.0	0.540	0
5	0.823	0.337	1.0	0.504	0
10	0.823	0.337	1.0	0.504	0
20	0.823	0.337	1.0	0.504	0
40	0.823	0.337	1.0	0.504	0
60	0.823	0.337	1.0	0.504	0
120	0.823	0.337	1.0	0.504	0
180	0.823	0.337	1.0	0.504	0
300	0.823	0.337	1.0	0.504	0

5.2.4 *Summary*

The experiments in this section demonstrate that the predictions made by the model have a sufficiently high accuracy. Models which have been trained for more than one minute produced no false negative predictions. This is a very good result as false negative guesses would lead to unique columns being ignored.

Furthermore, the positive guesses of the models in the experiments are correct at least one third of the time and even more for models with a larger input size. This is important as false positive guesses reduce the efficiency, which is explored further in Section 5.3.

5.2.5 *Examining columns which led to false guesses*

The greatest weakness of the proposed method are false guesses. False positive guesses lead to a reduced efficiency because more columns need to be checked with the naive algorithm. False negative guesses on the other hand result in primary key candidates being ignored which reduces the correctness. 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 primarily trained to avoid any false negative guesses. The experiment in Section 5.2.2 has shown that depending on the input size between 27 % and 78 % of the positive predictions are true positives.

The false positive guesses are unfortunately mostly unavoidable as they are caused mainly by empty cells which are located after the input rows of the model in otherwise unique columns. As the column would be a primary key candidate without these missing values, there is no possible change which would 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. Since the model only sees short strings without duplicates in these columns, there is no good way for the current implementation to recognize the column as non-unique. However, it could be possible to additionally include the column heading as a feature to enable the model to learn that the column with the names of authors is more likely to contain duplicate values.

False negative guesses occurred only in one of all the correctness experiments, namely with the model which was trained for just one minute in Section 5.2.3. And even then the false negatives were rare, and the corresponding columns contained only negative numbers or numbers smaller than 1. While these columns are unique, they are not suited very well as primary keys.

5.3 EFFICIENCY

Next to the correctness, the efficiency of the proposed method is the most important property 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.

Each generated table has 10 rows 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 which the sorting based naive algorithm takes to find unique columns.

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.

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

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 finding as few false positive guesses as possible because each false positive guess unnecessarily increases the runtime through the required validation and therefore decreases the efficiency.

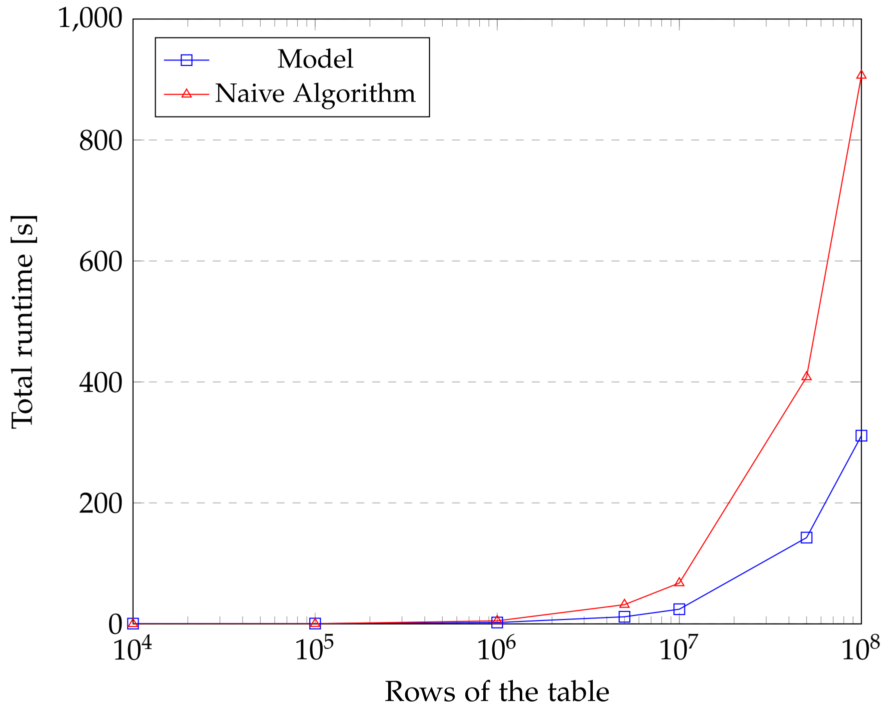
In conclusion, this experiment illustrates that for large tables loading the dataset and checking the columns for duplicates with the naive algorithm takes the most time. Possibilities to reduce the loading time will be explored in Section 5.3.3. While a more efficient naive algorithm is not part of this thesis, Section 5.2.2 and 5.2.5 deal with the question of how to decrease the number of false positive guesses.

5.3.3 *Reducing 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

Figure 5.1: This plot shows the total runtime of the proposed method and the naive algorithm as they search for unique columns in tables of different sizes. The tables were saved in a CSV format.



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 5.5 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.

Table 5.4: The result of the efficiency experiment conducted on a table generated with 3 unique and 7 non-unique columns saved as a CSV file. The experiment was conducted on a model with an input size of 10 rows which was trained on the training dataset. The time was measured in seconds.

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

In summary, while the reduced loading time does make a notable difference, it is not very large compared to the efficiency gain achieved through the use of the proposed method, which is additionally demonstrated 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 to non-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 prediction by the model has to be verified using the naive algorithm, the total runtime increases the more unique columns the model predicts.

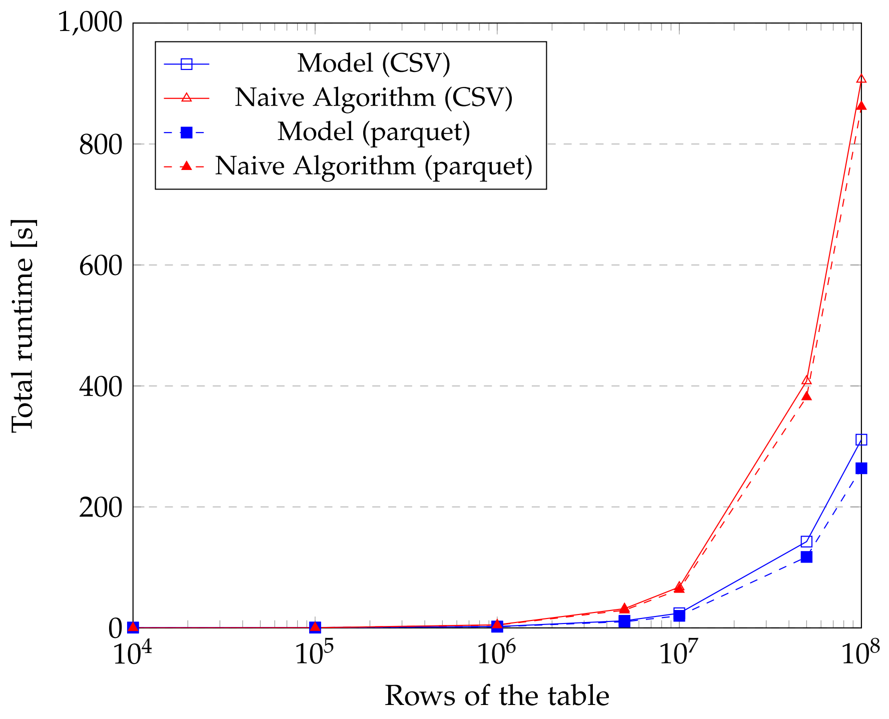
In this experiment, a model with an input size of 10 rows is used on 4 tables, which are saved as parquet files and each have 100 000 000 rows and 10 columns. The difference between these tables is the percentage of unique columns, which range from 60 % to 90 %.

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

Table 5.5: The result of the efficiency experiment conducted on a table generated with 3 unique and 7 non-unique columns saved as a parquet file. The experiment was conducted on a model with an input size of 10 rows which was trained on the training dataset. The time was measured in seconds.

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

Figure 5.2: This plot shows the total runtime of the proposed method and the naive algorithm as they search for unique columns in tables of different sizes. Additionally, the runtime when reading the data from CSV files and from parquet files are being compared.



In the GitTables dataset, which is used in the correctness experiment, the ratio of unique columns is approximately 11 %. The positive guesses of the model are quite a bit higher since its priority is to avoid false negatives, not false positives. Still, the experiment in Section 5.1

Table 5.6: The result of the efficiency experiment conducted on a table generated with 3 unique and 7 non-unique columns saved as a parquet file. The experiment was conducted on a model with an input size of 10 rows which was trained on the training dataset. During the experiment, only the necessary rows and columns were loaded. The time was measured in seconds.

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

has shown that the share of positive guesses during tests on the GitTables dataset is smaller than 30 % for a model with an input size of at least 10 rows. This is low enough to be a clear improvement over the naive algorithm given large enough tables.

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. The time was measured in seconds.

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

5.3.5 *Summary*

The experiments in this section show that the proposed method to find primary key candidates is suitable for some cases. If the tables that will be examined contain mostly viewer than 1 000 000 rows or the ratio of unique to non-unique columns is too high, the model is probably slower than the naive algorithm. However, on very large tables with

100 000 000 or more rows the model can significantly improve the overall runtime.

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

6 CONCLUSION

6.1 POSSIBLE APPLICATIONS

The proposed method is not suitable as a standalone application the way it is implemented as there are too many cases where its efficiency is far behind the naive algorithm. A more promising possibility would be to use this method as a part of a larger algorithm or program where it would be used only if it has an advantage over the naive algorithm.

Another useful characteristic of the proposed method is the fact that it produces a set of probable unique columns in a very short amount of time even for large tables. This would make it possible to display columns which may be primary key candidates very fast and validate the predictions in the background.

It may be possible to transfer the idea behind the proposed method, to use a machine learning algorithm which predicts if a column has certain properties (such as uniqueness), to other properties of tables. This way it could be possible to improve the speed in several areas of data analysis, not just the detection of primary key candidates.

6.2 LIMITATIONS OF THE PROPOSED METHOD

A big limitation of the proposed method is the fact that it only predicts uniqueness for single columns. This is a problem for two reasons. On the one hand, not every table has a primary key that includes only one column. On the other hand, information can be drawn from whether a column combination is unique. Here, the method needs to be further developed.

Additionally, there is the possibility that a unique column is classified as a non-unique by the proposed method. However, to rule out this possibility, all the columns would have to be inspected with the naive algorithm which would defeat the purpose of the method.

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).