Towards Automatically Linking Data Elements

by

Katharine Xiao

S.B. Electrical Engineering & Computer Science, Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© Katharine Xiao, MMXVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author .....
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by .....
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Towards Automatically Linking Data Elements

by

Katharine Xiao

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2017, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

When presented with a new dataset, human data scientists explore it in order to identify salient properties of the data elements, identify relationships between entities, and write processing software that makes use of those relationships accordingly. While there has been progress made on automatically processing the data to generate features or models, most automation systems rely on receiving a data model that has all the meta information about the data, including salient properties and relationships. In this thesis, we present a first version of our system, called ADEL-Automatic Data Elements Linking. Given a collection of files, this system generates a relational data schema and identifies other salient properties. It detects the type of each data field, which describes not only the programmatic data type but also the context in which the data originated, through a method called Type Detection. For each file, it identifies the field that uniquely describes each row in it, also known as a Primary Key. Then, it discovers relationships between different data entities with Relationship Discovery, and discovers any implicit constraints in the data through Hard Constraint Discovery. We posit two out of these four problems as *learning* problems.

To evaluate our algorithms, we compare the results of each to a set of manual annotations. For Type Detection, we saw a max error of 7%, with an average error of 2.2% across all datasets. For Primary Key Detection, we classified all existing primary keys correctly, and had one false positive across five datasets. For Relationship Discovery, we saw an average error of 5.6%. (Our results are limited by the small number of manual annotations we currently possess.)

We then feed the output of our system into existing semi-automated data science software systems – the Deep Feature Synthesis (DFS) algorithm, which generates features for predictive models, and the Synthetic Data Vault (SDV), which generates a hierarchical graphical model. When ADEL's data annotations are fed into DFS, it produces similar or higher predictive accuracy in 3/4 problems, and when they are provided to SDV, it is able to generate synthetic data with no constraint violations.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

I would like to first and foremost thank my advisor Kalyan for the many hours of feedback, guidance, and iteration put into this thesis project. I would not have been able to finish this thesis to this quality without his ideas and expertise.

I would also like to thank Carles for his manual annotations that provided the baseline for my testing results, Roy for his initial help in understanding the Synthetic Data Vault, and Arash for his wonderful graphics. A special thanks to Max and the other engineers behind Deep Feature Synthesis for helping me run my results.

We acknowledge generous funding support from Accenture under their program AI for Software Testing.

Finally, thanks to all my lab mates in the Data to AI lab who listened to my presentations and progress, and provided feedback.

Contents

1 Introduction **15**

1.1 Towards furthering data science automation 17

1.1.1 The Synthetic Data Vault 17

1.1.2 Deep Feature Synthesis 18

1.1.3 Different Views 18

1.2 Automatic Data Element Linking 18

1.3 Goals 19

1.4 Thesis Roadmap 19

2 Concepts and Terminology **21**

2.1 Databases 21

2.1.1 Primary Keys 21

2.1.2 Foreign Key Relationships 22

2.1.3 Hard Constraints 22

3 Overview - ADEL **25**

3.1 The meta.json File 27

4 Type detection and relationship discovery **33**

4.1 Type detection 33

4.1.1 Type Detection Algorithm 34

4.2 Primary Key Discovery 37

4.2.1 The Primary Key Discovery Algorithm 38

4.3 Relationship Discovery 40
4.3.1 The Relationship Discovery Algorithm 41

5 Constraint Discovery **45**
5.1 Motivation for discovering hard constraints 45
5.2 Discovering hard constraints 46
5.2.1 Hard Constraint Discovery Algorithm 46
5.2.2 Simple optimizations 47
5.3 Complexity Analysis 48
5.4 Transformations 49

6 Testing Automated Discovery **51**
6.1 Collecting Manual Annotations 51
6.2 Datasets 53
6.2.1 Schema Overview 54
6.3 Methods 57
6.3.1 Training 57
6.3.2 Testing 58
6.4 Results 60
6.4.1 Type Detection 60
6.4.2 Primary Key Detection 62
6.4.3 Relationship Discovery 62
6.5 Towards Improving Our Learning Algorithms 64

7 Towards Automating Data Science Endeavors **65**
7.1 Motivating Examples 65
7.2 Deep Feature Synthesis 67
7.2.1 Connecting the Pieces 67
7.2.2 Experiment 67
7.2.3 Results 68
7.3 Synthetic Data Vault 69

7.3.1 Connecting the Pieces 69
7.3.2 Hard Constraints 70
7.3.3 Experiment 71
7.3.4 Results 72

8 Conclusion and Future Work **73**
8.1 Future Work 73
8.2 Conclusion 74

A Synthetic Data Vault **75**
A.1 Overview 75
A.2 SDV API 76
A.3 API Feedback 77
A.4 API Enhancements 78

B ADEL API **79**
B.1 Training 79
B.2 Schema Generation 79
B.2.1 Hard Constraint Transformations 80
B.3 Testing and Validation 81
B.3.1 Isolating Modules 81
B.4 Generating Results 81

C Collecting manual annotations for data **83**
C.1 Type Classification 83
C.2 Primary Key Detection 85
C.3 Relationship Discovery 86
C.4 Hard Constraints Discovery 88

List of Figures

2-1 An illustration of a database foreign key relationship. 23

3-1 This is an overview of the tasks of the Automatic Data Element Discovery (ADEL) process, along with the different tasks' results. Given the path to a directory of *csv* files, ADEL will populate a schema file by learning the types of the tables' fields, find the primary keys of each table, discover relationships between the tables, and finally discover any hard constraints within the individual tables. This information will be summarized in an output schema file, and partially-populated schema files can be extracted from any point in the pipeline. 26

4-1 Type and subtype classifications for the SDV. The type that is not shown is 'text' which is not synthesized. 34

6-1 A summary of the Airbnb dataset types and relationships. 54

6-2 A summary of the Rossmann dataset types and relationships. 55

6-3 A summary of the Telstra dataset types and relationships. 56

6-4 A summary of the Biodegradability dataset types and relationships. . 56

6-5 A summary of the Mutagenesis dataset types and relationships. . . . 56

6-6 This figure depicts the automated testing suite for evaluating type detection, primary key detection, and relationship discovery. Each testing module takes in a partially-filled file of annotations to feed into each respective algorithm. The output is then compared with the original manual annotations file. 59

7-1 The Employees dataset, used for analyzing hard constraints. 71

A-1 The SDV workflow. 76

C-1 UI to collect annotations for a table's types. 84
C-2 UI to collect annotations for a table's primary key. 86
C-3 UI to collect annotations for foreign keys between two tables. 87
C-4 UI to collect annotations for hard constraints within a table. 89

List of Tables

4.1 A list of the features used in the type classification decision tree model. 36

4.2 A list of features calculated by the foreign key classification algorithm, as presented in Rostin et al. [19]. These features are extracted from an inclusion dependency pair to determine if the inclusion dependency is a foreign key reference or not. In this table, F is the potential foreign key field and P is the potential primary key field. 44

5.1 A summary of the discussed hard constraints and the corresponding transformations. 49

6.1 Parameters of the datasets. The # Fields per dataset corresponds to the number of type-classification examples that will be available for training and testing. Similarly, the # Inclusion Dependencies tells us the number of relationship-discovery examples that are available for training and testing for a certain dataset. The # FK references corresponds to the number of positive training examples available for relationship discovery. The # Tables corresponds to the number of primary key identifications that will be made for a given dataset. . . 53

6.2 Type misclassifications for each dataset, broken down by type. The values for each column show the ratio of the number of fields that should have been classified as the type indicated, but were instead classified as something else, over the total number of fields of that type. 61

6.3 Primary key results: False positives for each dataset - the number of fields that were not primary keys, but were classified as primary keys, over the total number of primary keys. 62
6.4 Foreign key relationship results: The number of missed foreign keys over the total number of foreign keys per dataset. 62

7.1 The prediction problem for each of these datasets is to predict the value of the specified field, which is part of the listed table. 68
7.2 The predictive accuracy results of the control and the Automatic Data Element Linking (ADEL) -generated schema for each dataset. 69
7.3 SDV Results: The percentage of violations in SDV-generated synthetic data, with the hard constraints enforcement and without. 72

C.1 The information that should be captured in a "type_classification" database when an annotation is submitted. 84
C.2 The information that should be captured in a "pk_classification" database when an annotation is submitted. 85
C.3 The information that should be captured in a "fk_classification" database when an annotation is submitted. 87
C.4 The information that should be captured in a "hc_classification" database when an annotation is submitted. 89

Chapter 1

Introduction

From virtual assistants to chess Grandmasters, more and more platforms are striving to replicate human intuition and problem solving. In data science, researchers are striving to solve problems by deriving predictive models using machine learning approaches. Platforms like Kaggle [13] host predictive modeling competitions to try and find the model that best captures what human intuition is so capable of identifying – semantic relationships and their context – along with the software to process it.

When given a dataset, a data scientist's first task is to understand the data. This involves finding and discovering properties of data elements – columns, rows, and relationships in the data. Such relationships include many-to-one relationships – for example, if many entries in a table representing Store Visits reference the same entry in the Customers table, this implies that a customer visited a store multiple times.

Once a data scientist identifies the properties and relationships within a dataset, he can then write software to process this data and generate features, or model it using a probability distribution. Here, let's consider how the properties of data and relationships play a role in generating features and models.

Feature generation: Consider a customer table. If one of the fields is the state where the customer lives, a data scientist may recognize that it is a categorical variable and choose to process it using *one-hot-encoding* to generate features for the machine learning algorithm. The data scientist uses the semantics of

the field name and the context to determine that the field type is categorical, and processes it accordingly.

Similarly, consider a *log* table that records all transactions across all customers over a period of time. A data scientist might recognize that a customer may have multiple transactions recorded in this table. To compute features for a customer, he/she will then write software to extract a single customer's transactions and aggregate the data to generate numeric features for that customer. These numeric features may include, for example, the total number of transactions the customer had, or the number of transactions where the `event-type` is `card-add`.

Modeling: Often, data scientists wish to learn a probability density model for individual columns in a table, a joint model for multiple columns, or a hierarchical graphical model for the entire dataset. To learn a model *pdf* for individual data columns, or a joint model, the data scientist would first recognize the types of variables involved and choose the appropriate distributions. If the data type is numeric, specifically float, s/he may choose a continuous density model. If the type is categorical, he/she may apply a transformation before modeling it using a continuous density model.

To model a hierarchical or nested graphical model, the data scientist may exploit the one-to-many or one-to-one relationship(s) that exist between different data entities.

Given that data scientists spend a non-trivial amount of time using their intuition to understand the data and its context in order to determine these salient properties, we ask whether we can automatically find these key properties for a given data set. We are motivated by a few key observations:

Larger datasets: Real world data sets now contain hundreds of data tables and several hundred fields (if not thousands).

Increased need for data science: With increasing need for data-based analytics,

companies are bringing in employees with expertise in data science. These employees are given the data, but rarely have enough documentation to help them understand it.

To facilitate this initial stage of the data science process, we propose an automatic system for linking data elements, which we call ADEL. In its first version, here is what our system does:

Starting with raw data values in *csv* files, ADEL automatically detects types of columns/fields in a file, discovers the primary key for each file, discovers relationships between different entities (files), and identifies implicit logical or arithmetic relationships between two or more columns, which are called hard constraints. It generates a *json* file with this information.

1.1 Towards furthering data science automation

In the past few years, multiple algorithms and software systems have been developed to automatically process data, essentially imitating what a data scientist would do when given a dataset and a goal. These systems assume that relational information about data is available. Below, we give two examples of such systems.

1.1.1 The Synthetic Data Vault

The Synthetic Data Vault (SDV) generates a hierarchical graphical model given a dataset with multiple tables. The model then can be used to sample data that is artificial. If noise is added to the model parameters, it allows users to sample from a perturbed distribution. The sampled artificial data can be used for testing and debugging software, and even published to allow a crowd of data scientists to work on it [17]. The SDV's algorithm relies on being provided all meta properties of the data, types for the fields, relationships between multiple tables and the primary key for each table. In this thesis, by generating this information automatically from a set of files, we are able to further usage of SDV in a variety of scenarios.

1.1.2 Deep Feature Synthesis

Another data science process that builds on top of relational datasets is feature generation. Competitions, such as the ones hosted on Kaggle [13], and projects, such as Deep Feature Synthesis (DFS) [14], expect input data that is structured and relational, and use those data to construct features. Both Kaggle and the automated feature discovery algorithm DFS utilize a high-level summary of the relational dataset's properties and constraints to create features for prediction. These features are expected to capture semantically important problems to solve a prediction problem.

1.1.3 Different Views

For data scientists or statisticians who prefer to view their data in a different structure, the schema of a relational dataset would provide enough information to allow them to restructure the raw data. These different views might provide a less efficient, but more intuitive way of representing data, such as in MongoDB, which makes use of embedded fields [12].

All three of the above data science endeavors involve building on structured data that has already been processed and organized by humans. If we could automatically detect the same intuitions as the ones captured in relational datasets, we could allow all of the applications that build on relational datasets to build directly on raw data with minimal processing and minimal delay.

1.2 Automatic Data Element Linking

The Automatic Data Element Linking (ADEL) platform represents our approach to capturing the same human intuitions that are present in relational databases. We aim to create an automated process that reads in raw data from *csv* files, and generates a schema that summarizes the dataset's organization, relationships, and constraints.

This schema can then be generalized, and passed into endpoints such as the Syn-

thetic Data Vault (for generative modeling and synthetic data) or Deep Feature Synthesis (for feature creation and predictive learning).

1.3 Goals

Ultimately, we wish to automate the detection of entity types, relationships and constraints in raw data to produce a relational schema for a dataset.

Success should include:

1. Accuracy: The detected structure, metadata, and constraints should be relatively accurate, and should result in a relatively small percentage of misclassifications and false positives.

2. Usability: The platform should be intuitive to use, and allow for a balance between automation and user control.

3. Generalizability: The resulting end-to-end platform should produce output that is useful for many different data science endeavors that build on relational datasets.

1.4 Thesis Roadmap

The remainder of the thesis is organized as follows:

Chapter 2 reviews key concepts and databases that are relevant in this thesis. Chapter 3 gives an overview of the Automatic Data Element Linking process, breaking down the end goal into 4 discrete steps.

Chapter 4 provides the algorithms and procedures used to extract general structure information. Chapter 5 delves into the technical details and the algorithm used to detect hard constraints, as well as the system created to propagate these constraints. Chapter 6 reviews the initial results of the automated detection algorithms, and summarizes the key findings.

In Chapter 7, we go through two endpoints to explore the applications of our Automatic Data Element Linking output. Chapter 8 proposes future work and wraps up the thesis with a conclusion.

This thesis presents a system for automatically detecting essential structural and semantic-based properties necessary for dataset analysis.

Chapter 2

Concepts and Terminology

In this thesis, we focus on generating a relational schema. In this chapter, we define the properties that we will be interested in identifying. These properties are present in all types of data, independent of how they are structured.

2.1 Databases

A database is a structured collection of data with some implicit meaning. The data in a database is typically derived or collected from some real world source, and has an intended application. A database is made up of different tables, each table with columns that each represent a entity with a different data type. A row, or a tuple, of a table represents an instance of this table object.

2.1.1 Primary Keys

In each table, there is a designated **primary key** that comprises of a subset of the table columns. The primary key satisfies a uniqueness constraint, requiring that the primary key value of any two rows in the table must be distinct. This allows for the primary key to uniquely identify every row in a given table. A primary key must also be minimal, such that removing any of the fields in the primary key subset would break the uniqueness constraint.

2.1.2 Foreign Key Relationships

In this thesis, we focus on relational data schemas, which consist of relations between entities of data. A **foreign key** is a many-to-one relationship that describes such a reference, in which a column of one table points to the primary key (the unique identifier) of another column. In this relationship, the values in the foreign key column, FK, of the first table must be a subset of the domain of the second table's primary key , PK. In other words, $t_1[FK] = t_2[PK]$, where t_1 represents a row in the first table, and t_2 represents a row in the second table. We call the first table the **child table** and the second table the **parent table**, as the child table has a reference into the parent table.

For example, we might have a parent table representing the `Customers` of a store, and a child table representing all of the `Visits` of every `Customer`. Many entries in the `Visits` table could reference the same `Customer`. Each entry in the Visits table would have a foreign key reference to a specific `Customer`. This relationship is illustrated in Figure 2-1.

For all valid foreign key references, the set of values in the foreign key field is a subset of the set of values in the referenced primary key field. We call this property an **inclusion dependency**.

2.1.3 Hard Constraints

In databases, there are often constraints that need to be met, which are obvious from the context the data was collected from. We call these constraints **hard constraints**. Types of hard constraints include arithmetic constraints, which can be a specification that Col A must always be greater than Col B within a given entry. There are also temporal constraints, which constrain new entries based on previous entries. In this thesis, we focus on arithmetic hard constraints within a row.

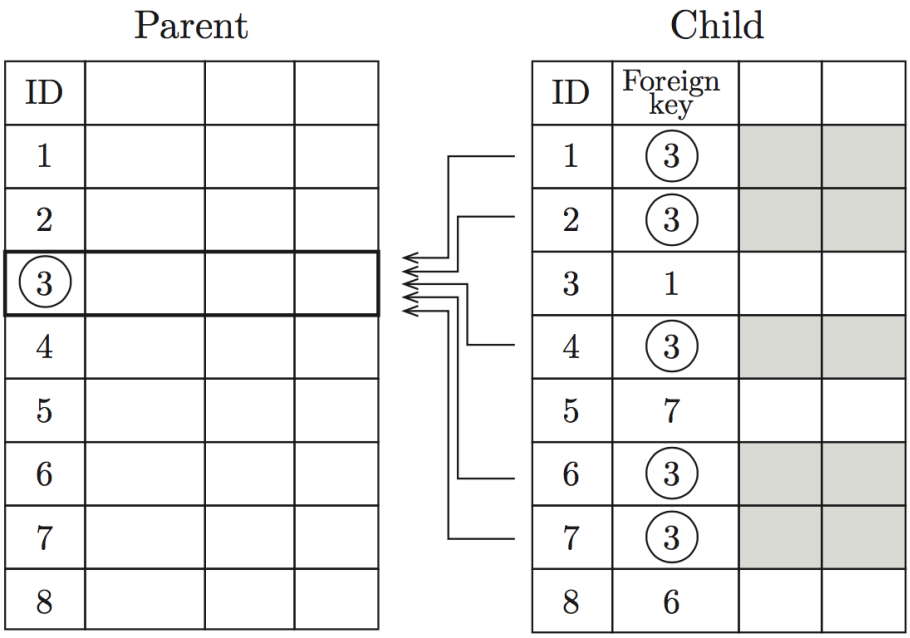

Figure 2-1: An illustration of a database foreign key relationship.

Chapter 3

Overview - ADEL

We present an overview of the Automatic Data Element Discovery process, and describe how the larger goal of creating a relational schema is broken down into stages. Each of the following sections describes one stage of the four-part process. A user can invoke each of these stages individually to selectively infer information, or the user can invoke all of these stages in order to generate a full schema.

A summary of this process is presented in Figure 3-1. From a user's perspective the flow is as follows:

1. Place all *csv* files into a single directory, where each *csv* file contains a single table.
2. Call `gen_empty_meta(directory_path)`, which creates a template json file, with only superficial information, such as the number of rows, filled in.
3. Detect the types of the fields in each file by calling `classify_field_types`.
4. Detect the primary key of each file by calling `find_PKs`.
5. Discover the relationships within the dataset with `discover_relationships`.
6. Discover any hard constraints within the tables with `discover_hard_constraints`.

A more detailed look at the Automatic Data Element Discovery API is presented in Appendix B. The final output of this process is a json file, called `meta.json`, which includes all of the information deduced during the four algorithms. After providing an overview of each of the four algorithms, we describe the `meta.json` output file and its format.

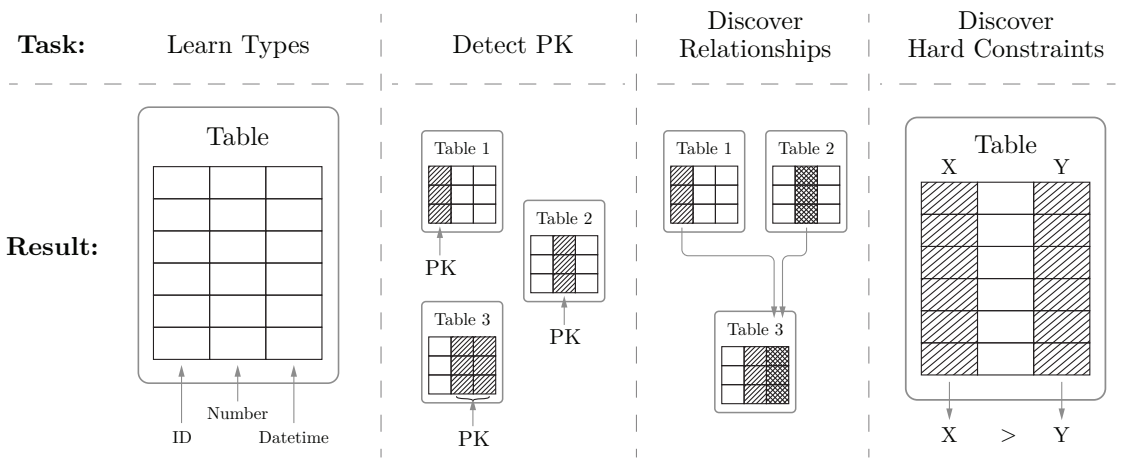

Figure 3-1: This is an overview of the tasks of the Automatic Data Element Discovery (ADEL) process, along with the different tasks' results. Given the path to a directory of *csv* files, ADEL will populate a schema file by learning the types of the tables' fields, find the primary keys of each table, discover relationships between the tables, and finally discover any hard constraints within the individual tables. This information will be summarized in an output schema file, and partially-populated schema files can be extracted from any point in the pipeline.

Detect Types In the first step, we find the type and subtype for each field. The type describes, at a high level, what kind of data is represented in a field. The different types are *categorical*, *number*, *ID*, *datetime*, and *text*. We are also interested in the subtype of each field, which breaks down each type into more specific categories. For example, a categorical type can be further classified as Boolean or ordered. The algorithm to detect field types is presented in Section 4.1.1.

Detect Primary Keys Each table has a set of fields that is used to uniquely identify each row, known as the *primary key*. In this next step, we identify the most likely primary key of each table, based on properties based in both the raw data and semantics. The algorithm to detect primary keys is presented in Section 4.2.

Discover Relationships After we know each table's primary keys, we determine if any of these primary keys are part of any foreign key references. These inter-table relationships are important for understanding how tables reference each

other, and as a result are extremely useful in feature generation (DFS) [14] and building hierarchical graphical models (SDV) [17]. The algorithm to discover relationships is presented in Section 4.3.

Discover Hard Constraints Finally, we detect a subset of predetermined hard constraints. These constraints are relevant to platforms that focus on model generation and synthetic data generation, as the synthetic data should adhere to these implicit constraints. The problem of hard constraints is explored in Chapter 5.

3.1 The meta.json File

The metadata and structural information about a relational database that is derived from the above four algorithms is now used to populate in a file called `meta.json`, which is all of the metadata structured in a json format. At the highest level, the file contains information about the path to the dataset, as well as a list of table objects. Each table object contains:

- The path to the table itself.
- The table's primary key.
- Whether or not the table *csv* file contains a header row.
- The number of rows.
- A list of objects describing the fields in a table.
- A list of hard constraints for the table, if any exist.

The example metadata object of a table called `users` is shown below.

```
{
    "path": "",
    "tables": [
```

```
        {
            "path": "users.csv",
            "name": "users",
            "headers": true,
            "fields": [
                ...
            ],
            "primary_key": "id",
            "number_of_rows": 213451,
            "hard_constraints:" {
                ...
            }
        },
        ...
    ]
}
```

The hard constraints entry is a mapping from the two types of potential constraints to a list of those respective constraints. The first type is of form $A > B$, and we will call this type of constraint 'GT' for 'Greater Than.' The second type has form $A+B = C$, and we will call this 'EQ' for 'Equal.' To specify the individual constraints, we need to specify the fields A, B and potentially C, which participate in the constraint. We also note the original field type, which is shared by all fields in the constraint. An example hard constraints entry might look like:

```
"hard_constraints:" {
    "GT": [
        {
            colA: "to_date",
            colB: "from_date",
            type: "datetime"
```

```
        }
    ],
    "EQ": [
        {
            colA: "count_one",
            colB: "count_two",
            colC: "total_count",
            type: "integer"
        }
    ]
}
```

Each field objects contains:

- The field *name*.
- The field *type*.
- The field *subtype*.
- The number of unique values.
- If the field is of type *datetime*, the datetime format.
- If the field is of type *ID*, the regex.
- The foreign key reference to another table's primary key, if one exists.

The type and subtype options are shown in Figure 4-1.

```
{
    "name": "language",
    "type": "categorical",
    "subtype": "categorical",
    "uniques": 25
}
```

For datetime fields, there is no subtype. Instead, the format of the datetime values needs to be specified, so applications know how to parse the datetime string when reading in the raw values.

```
{
    "name": "date_account_created",
    "type": "datetime",
    "format": "%Y-%m-%d",
    "uniques": 92
}
```

Similarly, if a field is an ID, the regex needs to be specified. There is optionally a subtype of "primary" if the field is the primary key of a table. If the field is an ID but not the primary key, the subtype field does not need to be included.

```
{
    "name": "id",
    "type": "id",
    "subtype": "primary",
    "regex": "^.{10}\$",
    "uniques": 213451
}
```

Foreign key references are also encoded in the `meta.json` files, as shown below. The `ref` keyword is used to represent that this field is a foreign key reference to another field in another table. In the example below, the `user_id` field is a foreign key into the `id` field of the `users` table.

```
{
    "name": "user_id",
    "type": "id",
    "regex": "^.{3,10}\$",
```

```
    "ref": {
        "table": "users",
        "field": "id"
    },
    "uniques": 135484
}
```

Chapter 4

Type detection and relationship discovery

In this chapter, we present algorithms for three problems - detecting types, discovering primary keys and discovering relationships.

4.1 Type detection

In our type detection problem, we are interested in categorizing fields into higher-level types that have semantic and contextual meaning. These higher-level types convey information about what a value represents. Such information can be used by data processing routines further down the data science pipeline.

Simply identifying that a value is of a certain programmatic type, such as an *integer*, does not convey information about what that value represents in its respective context. For example, an integer can represent a number, such as in the context of age. However, it can also represent a Boolean field, with 1 meaning 'yes' and 0 meaning 'no.' Furthermore, an integer could represent a year, which is a *datetime*, or it could represent one of the seven days of the week, making it *categorical*. Through Figure 4-1 we present the ontology of types we are interested in categorizing fields into, as well as their corresponding subtypes.

The high-level types that exist for Automatic Data Element Discovery are *categor-*

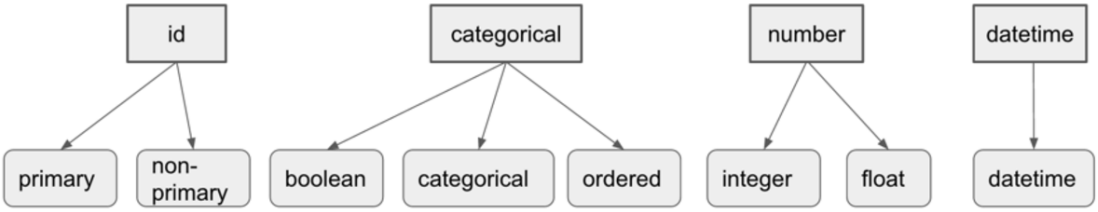

Figure 4-1: Type and subtype classifications for the SDV. The type that is not shown is 'text' which is not synthesized.

ical, *number*, *ID*, *datetime*, and *text*. Categorical types can have a subtype of *Boolean*, *categorical*, and *ordered*. Number types can have a subtype of *integer* or *float*. IDs have the option of being a *primary key*. Detection of primary keys is described in section 4.2.

4.1.1 Type Detection Algorithm

We first determine a field's type. Depending on the determined type, we then assign it to one of the corresponding subtypes. We posit type detection as a learning problem:

> Given a set of fields $f_1 \dots f_k$, data corresponding to those fields $d_1 \dots d_k$, and human annotations about their higher level type $type_1 \dots type_k$, can we learn a model td such that $type_i \leftarrow td(f_i, d_i)$.

The model $td(.)$ can then be used for predicting the higher-level types for data column whose higher-level type is not known. We use a machine learning approach to solve this problem, which traditionally involves the following steps:

Step 1: Collect a set of manually annotated data sets. That is, $f_1 \dots f_n$, $d_1 \dots d_n$ and their higher-level types annotated manually, $type_1 \dots type_n$.

Step 2: Extract *features* that can describe these data columns $d_1 \dots d_n$. The features extracted speak to both the data-based and semantic characteristics that may have played a role in humans classifying them into certain higher-level types. These features are not specific to any one dataset, and are meant to apply to generic raw data. In these features, we look for cases in which the ratio of

unique values to the number of total values in a field is one, which indicates a likely ID field. If field values are a programmatic type of integer or float, then it is likely that the type is *number*. However, if the built-in type is an integer but the number of unique values are small, the field is likely *categorical*.

We also look at semantic features such as if the column name contains certain phrases such as '_id' or 'date'.

The calculated features are listed in Table 4.1

Step 3: Type detection is a multi-class classification problem. Given the features that describe each data column and the field, we train a decision tree from Python's sklearn library, and store the classifier for later use.

Step 4: To validate our classifier, we follow the train-test methodology. We first partition our dataset into a training set and a testing set. We train the model on the training partition and test its performance on the test partition.

Learning this model in ADEL: A user can train a type classifier by calling `train_type_classifier` on a list of directory paths, which map to the set of training datasets. The `train_type_classifier` method will iterate through each field in the training datasets, and calculate the features listed in Table 4.1 for a given field, as well as pull out the corresponding manual annotation. Then, the function uses the extracted field features and labels to learn the classifier using sklearns decision tree.

Using the model in ADEL: A user can find type classifications for all the fields in a dataset by calling the `classify_field_types` method on a dataset. This method uses the previously-trained type classifier to determine types for all the fields in the given dataset, and writes it to the dataset's `meta.json` file.

Detecting Subtypes

Once we determine the field type using our pre-trained classifier, we rely on a hard-coded set of rules because not every type has a subtype, and the subtype choices for a given field is at most three. This classification is more straightforward. Here are how subtypes are identified:

Type Classification Features
Ratio of the number of unique values to the total number of values.
If the number of unique values is less than a hard-coded categorical threshold.
If the values are ints or floats (built-in type).
If the values are strings (not a numerical built-in type).
If the field name contains '_id' or field name is 'id.'
If the field name contains 'date' or 'time.'

Table 4.1: A list of the features used in the type classification decision tree model.

1. If the field type is *number*, the corresponding subtype is determined from the Python pandas library "dtype" parameter, which can detect what built-in Python primitive type the values are.

2. If the field type is *categorical*, it is further broken down into subtypes using the number of unique values in the data corresponding to that field. If number of unique values (excluding null entries) is ≤ 2, it is type boolean. Otherwise, it is labeled as categorical. We do not yet have a way of differentiating between categorical and ordered.

3. If the field is *id* we resort to primary key detection algorithm described in Section 4.2.

Format

If a field is classified as a *datetime*, we also need to find the datetime format, so that applications can parse the value strings into dates. When determining the format of datetime, we encode a list of popular datetime formats, such as `'%Y-%m-%d'`. Then we iterate through the list of common datetime formats and try to parse the entire column with a specific format. If there are no exceptions raised throughout the parsing, then we assign that datetime format to the field.

Regex

If a field is classified as an ID, we must deduce the regex to formalize how the IDs

are structured. To determine the regex, we convert the values to strings, and check for the most common cases:

1. If the values are all numerical [0-9]
2. If the values are all strings [a-z] or [A-Z]
3. If the values are a combination of numeric and string (or if they are other)
4. The min length of the values
5. The max length of the values

Finally, we form a regex string out of the above information.

Related Work

When using a programming language like Python, there are language-specific, built-in datatypes. For Python, this includes *int*, *float*, and *string*. Python libraries, such as the data analysis toolkit `pandas`, have functions to return the value's associated built-in type [2]. Apache Spark also has a functionality called `inferSchema` to generate a schema from csv data. The `inferSchema` function also returns built-in types, and can parse values as a *date*, but only if the date format is explicitly specified by the user [1]. Ultimately, these methods of type detection differ from ours, as ours is striving to automatically determine a higher-level type information.

4.2 Primary Key Discovery

Next, we detect the primary key of each table in the dataset. Our goal is to identify the primary key given multiple files, which we consider as multiple tables. We assume that a table does not contain data redundancies (we describe what we mean by this later), and has a primary key. Taking our simplified problem, we can make a few optimizations by making note of two properties:

1. The primary key of a table necessarily uniquely identifies each row. This means that each value of the primary key is unique for all rows.
2. The primary key of a table is a set of one or more fields. This means that

the primary key can be one column, which is very common, but can also be a combination of fields.

4.2.1 The Primary Key Discovery Algorithm

The primary key detection problem can be posited in the following manner:

> Given a Table with fields $f_1 \dots f_k$, find a field f_i or a set of fields $f_{i \dots r}$, where $i \dots r$ is a subset of $1 \dots k$, that uniquely determine each row in the table.

To discover primary key for each table, we developed a deterministic algorithm. The algorithm performs the following steps:

Step 1: Identify the candidate set. In the first step, we find all candidates for the primary key by making use of property 1. We calculate the ratio of the number of unique values in the potential primary key set to the total number of rows. If this ratio is one, then this potential primary key set becomes a candidate.

Additionally, when looking at primary candidate sets that contain more than one field, we aim to eliminate redundancy in our analysis. Specifically, if $fieldA$ has already been identified as a primary key candidate, we do not consider any multi-field candidate sets that include $fieldA$. This is because any multi-field candidate set that includes $fieldA$ is guaranteed to be unique across all rows, and does not convey any new information.

Optimization When picking the primary key candidates, it would be ideal to consider all possible sets of fields of different cardinalities. However, because analyzing all combinations of fields is computationally expensive, and it is most common for primary keys to be from a set that contains one or two field names, we limit our analysis to sets of size 1 and 2.

Step 2: Filtering the candidates. Once we have identified the possible primary key candidate set, our goal is to identify the most likely primary key for the table. We note that the uniqueness property is necessary but not sufficient.

For example, let us consider a field *startTime*. It is likely that a field like *startTime* will be unique for all rows, especially if we are being precise to the second. However, a field like *startTime* should not be used to uniquely identify rows, because it is possible for two rows to have the same start time, even if the current set of data does not present itself in this way. Human data scientists can identify such situations by analyzing this field semantically.

As a result, in the second step of primary key detection, we identify unlikely candidates and choose one candidate to be the table's primary key. If there are multiple primary key candidates, other candidates are favored over the unlikely ones. To annotate the candidates that are unlikely, we use the following rules:

- Unlikely candidates include fields that are of type *datetime* or of subtype *float*.
- Additionally, we favor length-one primary keys over primary keys of longer length. For example, if we have two candidate primary keys, (`user_id`) and (`user_name`, `date_joined`), we favor the most minimal candidate, which is `user_id`.

If there are still multiple candidates after applying the above rules, we select the first candidate in the list.

When a primary key for a table is determined, we also update the type of the primary key field to *id*, if it was not already that type.

Algorithm 1 Primary Key Discovery

```
1: procedure GetCandidates
2:     possible_cands ← all length-1 and length-2 sets of field combinations
3:     candidate_sets ← []
4:     for candidate_set ∈ possible_cands do
5:         if candidate_set is unique across all rows then
6:             candidate_sets ← candidate_sets ∪ {candidate_set}
       return FilterCandidates()
```

Related work: In our literature survey, we found that primary key detection has often been done in the context of database normalization [4][9], which is a process of

removing redundancies by splitting one table into many. For example, if a table representing class assignments includes fields `studentID` and `studentName`, and every time the `studentID` is different, the `studentName` is also different, that data is redundant; only the `studentID` needs to be included in the assignments entry to identify the student. The `studentID` $\rightarrow$ `studentName` mapping can be stored in another table. To perform database normalization, the dependencies between fields (such as the one between `studentName` and `studentID`) need to be determined first to identify redundancies [10]. Hence when normalizing, the primary key is automatically detected, because a primary key is a set of fields that all other fields depend on. Database normalization is an expensive process, as it involves looking for dependencies between all possible field combinations of all possible lengths.

In this thesis, we are focused on a different problem. We are given a set of files, and assume each has a primary key of its own. Our goal is to identify those.

4.3 Relationship Discovery

Once the types of each column have been identified and the tables' primary keys have been identified, we can perform relationship discovery. We are identifying foreign key-primary key (FK-PK) relationships, which are many-to-one relationships where many entries of one table reference the primary key of another table. An example of this is when many entries in a table called `StoreVisits` reference the same entry in the `Customers` table, representing the many store visits of one specific customer. To do so, we follow the general methodology of Rostin et al. [19], which we will detail below.

As mentioned in Section 2.1.2, all valid foreign keys contain an inclusion dependency. An inclusion dependency exists between two fields f_1 of table t_1 and f_2 of table t_2, when $t_1 \neq t_2$ and all of the values present in f_1 are also present in f_2. In other words, the set of values in f_1 is a subset of the set of values in f_2. Furthermore, in a foreign key reference, a field f_1 must reference the *primary key* of another table. However, not all inclusion dependencies that reference a primary key are valid foreign

keys; if a primary key field has a common set of values, for example [1,100], it is likely that another field will contain a subset of those values, despite not having any relation to that primary key field.

4.3.1 The Relationship Discovery Algorithm

The problem of identifying (FK-PK) relationships can be posited in the following manner:

> Given a set of tables, $Table_{1...p}$ and fields associated with each given by $f^1_{1...k} \dots f^p_{1...k}$, identify all pairs of (f^i_j, f^l_m) where

Like [19] we follow a two step methodology.

1. We first find all the candidate field pairs for (FK-PK) references based on inclusion dependencies.
2. We posit the selection of the valid foreign key reference as a learning problem and use a machine learning model.

Our algorithm works as follows:

Step 1: Find all candidate pairs. To determine all the field pairs that are candidates for being a foreign key reference, we make use of the inclusion dependency property of foreign key references - while an inclusion dependency is not necessarily a foreign key reference, all foreign key references necessarily contain an inclusion dependency. Therefore, by finding all pairs of fields that contain an inclusion dependency, we can narrow down the options for which field pairs can contain foreign key references.

To find inclusion dependencies, we iterate through all the fields in a table and compare those fields to the primary key fields of every other table. In a comparison, we find the set of values contained in a field, and see if it is a subset of the values contained in the primary key field. If so, we mark that pair of fields

as an inclusion dependency. Finally, we pass that list of inclusion dependencies to the next step.

Tolerating error: In an inclusion dependency, it is required that *all* of the values in a foreign key field are contained in the corresponding primary key field. However, this may not always be the case, as in cases where a value is incorrectly entered into the foreign key field and does not correctly reference a primary key field, or if the corresponding primary key field is deleted or perturbed. In our algorithm, we chose to allow for a small threshold to account for small errors, in which a pair of fields is actually a FK-PK reference, but one or two of the FK values do not correctly reference a PK value.

To add a small tolerance for error when finding inclusion dependencies, we instead check to see if the number of values in the FK field that are not present in the PK field is less than $\frac{1}{10}$ of the number of values in the FK field.

Our algorithm for finding candidate foreign key refernces differs from Rostin et al. [19] in how we find inclusion dependencies and the addition of error tolerance. The Data Civilizer system [8] also uses the method presented in Rostin et a. [19], and presents a different method for tolerating error based on string similarities of the field values.

Step 2: Filtering the Candidates. To determine which of the candidates are valid foreign key-primary key pairs, we take a machine learning approach. For this approach, we follow these steps:

1. **Collect manually annotated training examples**: Each dataset in the training set contains manual annotations of all foreign key references within that dataset.

2. **Extract features**: For each pair in the candidate set, we extract the features presented in Rostin et al. [19]. These features are listed in Table 4.2. These features draw on observations of trends that occur in most FK references. Specifically, we observe that foreign keys are not often

referenced by other foreign keys. Also, if the values in a field are contained in many other fields, then it is more likely that this is a common set of values, rather than a foreign key reference. On the other hand, it is common for primary keys to be referenced by many foreign keys. Also, it is common for the entries in a foreign key field to reference a large fraction of the values in the corresponding primary key (not just a few).

These features also draw on semantic observations: for example, the names of the FK and PK fields in a FK reference are often similar, if not the same. Also, the foreign key field name is usually representing the ID of another table, and as a result, often contains a substring denoting that.

3. **Learn a classifier**: As in type detection (Section 4.1.1), we use a decision tree classifier that takes in features and learns a classifying model that takes in inclusion dependencies and outputs "yes" or "no."
4. **Validate our classifier**: We partition our set of manually annotated datasets into a test set and a training set. We train the model on the training set and validate on the test set.

Learning a model in ADEL: To train a classifier, a user can call the a method called `train_fk_classifier`, which extracts all candidate sets as described above, as well as the label of each candidate pair from the manual annotations. We then use these features and labels to train a sklearn decision tree.

Using the model in ADEL To discover the foreign key relationships that exist in a dataset, a user can call the `discover_relationships` method. This method first finds all candidate pairs, extracts the relevant features from these candidate pairs, and then applies the previously-learned decision tree classifier on the extracted features. The classifier outputs a Boolean value, which corresponds to whether or not a given candidate set is a valid foreign key reference.

Relationship Detection Features
The number of unique values in F
The ratio of $\frac{\lvert s(F) \cap s(P) \rvert}{\lvert s(F) \rvert}$, where $s(X)$ represents the set of values in field X.
How many times F is referenced in all other inclusion dependencies.
How many times F is referencing other fields in other inclusion dependencies.
How many times P is referenced in all other inclusion dependencies.
The name similarity between fields F and P.
The difference between average string length of values in F and values in P.
The percentage of values in P that are not within the range of values in F.
If the name of F ends in a substring like 'id' or 'key'.
The ratio of the number of rows in their respective tables.

Table 4.2: A list of features calculated by the foreign key classification algorithm, as presented in Rostin et al. [19]. These features are extracted from an inclusion dependency pair to determine if the inclusion dependency is a foreign key reference or not. In this table, F is the potential foreign key field and P is the potential primary key field.

Chapter 5

Constraint Discovery

In a dataset, a table may contain implicit constraints that the data must also follow. These are arithmetic or logical relationships between the fields in the table. Although they may not be explicitly stated anywhere in the dataset, they are usually interpreted by human data scientists when processing or modeling the data. This is the case when learning generative/graphical models for the data, as we mentioned in Chapter **??**. Graphical models are capable of capturing statistical correlations and dependencies, but not implicit arithmetic/logical relationships. In most cases, a human data scientist will look at the data and understand the fields based on the semantics, the context, and the data values, and make some transformations to define random variables for learning the graphical models. Our goal through constraint discovery is to automatically identify these constraints to enable transformation of the original field into a new random variable that can be fed into learning a generative model.

5.1 Motivation for discovering hard constraints

Statistical models are good for capturing general properties and distributions of data. However, they are incapable of capturing certain relationships that necessarily take place between and within rows. We define these relationships as 'hard constraints.' One example of a hard constraint that is not captured by a statistical model is a greater-than / less-than relationship. If Dataset A has a `startTime` column and an

`endTime` column, then for all rows, each `startTime` value is necessarily less than the `endTime` value.

Often, we expect data samples from generative models to behave like the original data. Any applications that use Dataset A (described above) would likely operate under the valid assumption that `startTime` is always less than `endTime`. Having a statistically generated data that sometimes violates this property would not be desirable.

5.2 Discovering hard constraints

Hard constraints are largely only detected via examining semantics, and require some knowledge of the dataset's context. In the above example, we can infer from the meaning of the column names `startTime` and `endTime` that one column represents a start and the other represents an end, and that the end column will come after the start. There are various types of hard constraints that are common in relational datasets. The data generation system created by Houkjaer et al. allows users to specify intra-row dependencies, like the example presented above, as well as intra-column dependencies, which are dependencies within columns [11]. In this thesis, we consider two types of intra-row hard constraints: *logical* and *arithmetic*.

In the following section, we propose an algorithm that discovers possible hard constraints. In the resulting set of candidates, it is possible that there will be false positives – identified hard constraints that do not make sense semantically, and do not always hold.

5.2.1 Hard Constraint Discovery Algorithm

The hard constraint discovery problem can be specified as follows:

> Given a table with fields $f_1 \dots f_k$, identify all possible pairs or triplets of fields f_i, f_j or f_i, f_j, f_k that conform to one of the relationships from a predefined relationship set R

In the first iteration of the hard constraints algorithm, our goal is to discover hard constraints:

1. Involving two or three columns (a.k.a fields),
2. and pertaining to a relationship set $R \leftarrow \{>, < .+, -\}$. That is, for columns A, B and C, the possible cases are:

   I. $A > B$

   II. $A < B$

   III. $A = B + C$

   IV. $A = B - C$

Out of the possible column types, only datetime- and numerical-typed columns are candidates for the specified hard constraints. As a pre-processing step, we first identify the names of the columns for these two types, and store them in $candidates_D$ and $candidates_N$ respectively.

Algorithm 2 Hard Constraint Discovery

```
1: procedure DiscoverHardConstraints
2:     for col_A ∈ candidates, col_B ∈ {candidates − col_A} do
3:         if col_A > col_B then
4:             GT ← True
5:         else
6:             if col_A < col_B then
7:                 LT ← True
8:         for col_C ∈ {candidates − col_A − col_B} do
9:             if col_A = col_B + col_C then
10:                EQ_A ← True
11:            else
12:                if col_A = col_B − col_C then
13:                    EQ_B ← True
```

5.2.2 Simple optimizations

Redundancies: First, we notice that constraints II and IV are inverses of constraints I and III, respectively. More explicitly, if we check constraint I ($col_A > col_B$) with

$col_A = endTime$ and $col_B = startTime$, it is redundant to also check constraint II ($col_A < col_B$) with $col_A = startTime$ and $col_B = endTime$, which would happen at a later iteration. It follows that we can reduce the number of for-loop iterations to only check the unique sets of column pairs.

Likewise, it is clear that we do not need to record the inverse constraints for each pair of columns. Therefore, we can simplify the constraint checks to constraints I and III. To eliminate unnecessary redundancies, we propose the following revised algorithm. This optimization does not change the runtime analysis, but in practice would reduce the runtime by a factor of 2, which is significant in the more common, smaller cases. The updated list of possible hard constraints:

I. $A > B$

II. $A = B + C$

Algorithm 3 Revised Hard Constraint Discovery

```
1: procedure DiscoverHardConstraints
2:     visited_candidates ← {}
3:     for col_A ∈ candidates, col_B ∈ {candidates − col_A − visited_candidates} do
4:         visited_candidates ← visited_candidates ∪ col_A
5:         if col_A > col_B then Record as hard constraint I, with colA = col_A and colB = col_B
6:         else
7:             if col_A < col_B then Record as hard constraint I, with colA = col_B and colB = col_A
8:         for col_C ∈ {candidates − col_A − col_B} do
9:             if col_A = col_B + col_C then Record as hard constraint II, with colA = col_A, colB = col_B, colC = col_C
10:            else
11:                if col_A = col_B − col_C then Record as hard constraint II, with colA = col_B, colB = col_A, colC = col_C
```

5.3 Complexity Analysis

Let us denote the number of datetime columns as d and the number of numerical columns as n. In the search for hard constraints among the d datetime columns, we iterate through all $\frac{d*(d-1)}{2}$ pairs of column candidates. For each pair, we do two $O(1)$-

Constraint	Transformation
$A > B$	$A' = A - B$
$A = B + C$	remove A

Table 5.1: A summary of the discussed hard constraints and the corresponding transformations.

complexity comparisons to check for constraints I and II. Then we iterate through the $d-2$ remaining col_C candidates to check for constraints III and IV. The complexity for discovering hard constraints in the datetime columns is $O(\frac{d*(d-1)*(d-2)}{2}) = O(d^3)$. Checking for hard constraints among the n numerical columns follows the same analysis, leaving us with an overall complexity of $O(d^3 + n^3)$ for the hard constraint detection algorithm.

5.4 Transformations

For each of the hard constraints we have discovered, we now need to transform the data such that we can create an equivalent random variable that a generative model can use. Below, each hard constraint and its corresponding transformation is explained.

I. A > B

In this constraint, column A is always greater than column B within the same row. With a statistical model, we might generate A-values that are greater than B-values with some small probability. To disallow this possibility, we can map the original column B to a $A' = A - B$. This way, we use a statistical model to generate the positive differences between columns A and B. At the end we recover column B with $B = A + B'$.

II. A = B + C

In this constraint, column A can be derived from columns B and C. To preserve this hard constraint, we only model columns B and C and repopulate column $A = B + C$ as a post-processing step.

Chapter 6

Testing Automated Discovery

In this chapter, we test each individual algorithm, as presented in Chapters 4 and 5, for its ability to make the correct classifications. This involves two main steps:

1. Training classification models for type classification and relationship discovery.
2. Applying automatic detection to new datasets.

To train, test, and validate our approaches, we need datasets that are fully annotated. In other words, we require datasets for which we have fully populated and verified `meta.json` files.

First, we describe the methodology for collecting manual annotations for validation, along with its challenges. While our current results are limited by the small number of manual baselines available, we describe our plan to accumulate more manual annotations, and introduce an systematic testing framework.

Within this chapter, we also describe our chosen datasets and the context of their data. Finally, we describe the training and validation steps, along with results and insights.

6.1 Collecting Manual Annotations

Every automated platform faces the challenge of finding or creating a baseline derived from humans with which to compare the automated results. In their paper on ap-

plying a machine learning approach to foreign key discovery, Rostin et al. evaluated their results on a total of six datasets, which they manually annotated themselves [19]. By using one very large dataset, they were able to get a significant number of training examples. However, the challenge of collecting a large number of manual annotations in a scalable manner remains.

For this reason, we sought a small number of baseline `meta.json` files for comparison. To complete the manual annotations that we do have, we hired a data scientist named Carles Sala Cladellas. Cladellas's job was to manually annotate the given set of datasets that we present in Section 6.2.

For type detection, Cladellas created a script to calculate summary statistics over each field, such as whether the field values were programmatic type ints, how many unique values there were in a field, and the min and max lengths of the value strings. His script then used those features to give a preliminary type assignment. After the script completed, he went through and manually fixed type classifications that were incorrect.

For primary key detection, he wrote a script that classified a field as being a primary key if all of its entries were unique. At the end, he went through and manually corrected any primary key errors, and manually added in any that were missed.

For relationship and hard constraint discovery, Cladellas manually identified foreign keys and hard constraints, and manually populated the `meta.json` file.

Cladellas's annotation process was not completely manual for type and primary key detection, as he made use of aggregate properties of the data that were calculated via a Python script. However, his method still captures the human intuition involved in creating relationship schemas, as he manually checked all the information, and made corrections when necessary. This manual checking applied his understanding of the dataset and its contextual information to the resulting schema.

Cladellas's method is also not completely automated, and differs from our approach in that the rules in his script were often curated for a specific dataset. His script had hard-coded rules that were specific to the individual datasets – for example,

Dataset	# Fields	# Inclusion Deps	# FK references	# Tables
Biodegradability	17	3	5	5
Mutagenesis	14	3	3	3
Airbnb	49	3	3	4
Rossmann	27	7	2	2
Telstra	14	6	5	5

Table 6.1: Parameters of the datasets. The # Fields per dataset corresponds to the number of type-classification examples that will be available for training and testing. Similarly, the # Inclusion Dependencies tells us the number of relationship-discovery examples that are available for training and testing for a certain dataset. The # FK references corresponds to the number of positive training examples available for relationship discovery. The # Tables corresponds to the number of primary key identifications that will be made for a given dataset.

if the field name is `startTime`, then the datetime format is `%Y-%m-%d`. Additionally, his script did not detect foreign key relationships or hard constraints.

We make use of Cladellas's semi-manual annotations in our comparisons, and we treat them as the baseline 'ground truth.' Next, we will introduce the datasets for which we have these manual baseline `meta.json` files.

6.2 Datasets

We collected a total of five relational datasets, three from Kaggle [13] and two from an online dataset repository [15]. In Table 6.1, we provide a summary of the relevant parameters for each dataset.

The number of fields tells us the number of training points per dataset that are available for training the type-classification problem. Similarly, the number of inclusion dependencies tells us how many training points per dataset are available for the foreign key classification problem, and the number of foreign key references tells us the number of positive examples in a dataset. The number of tables tells us how many primary keys will be determined for each dataset.

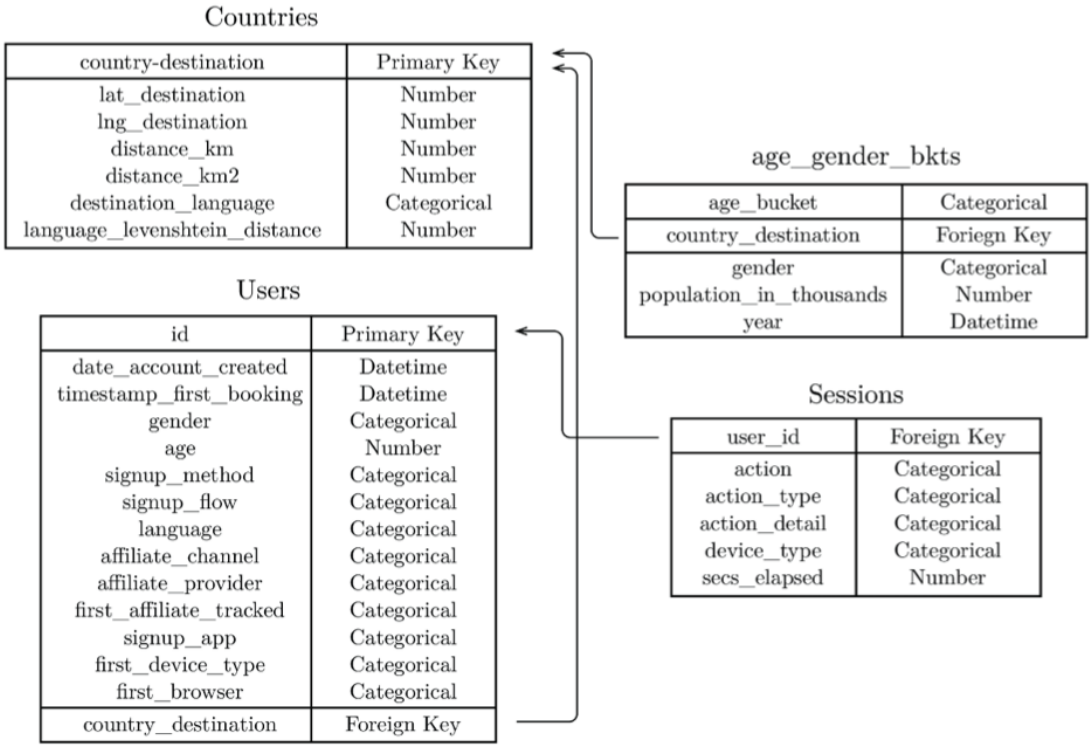

Figure 6-1: A summary of the Airbnb dataset types and relationships.

6.2.1 Schema Overview

Next, we will give a brief overview of each dataset.

Airbnb

The Airbnb dataset comes from a data prediction competition on Kaggle [3]. The schema is summarized in Figure 6-1. Each entry in the `Users` table represents an account on the Airbnb platform, and each entry in `Sessions` represents a log of web access by a certain user. The `Countries` table holds information about different countries, and is referenced by the `country_destination` field in the `Users` table, detailing which countries users booked stays in. The `age_gender_bkts` table provides demographic information about users traveling to the referenced countries.

Rossmann

The Rossmann dataset is another dataset from Kaggle, presented in a competition for Rossmann Store Sales [18]. The schema is summarized in Figure 6-2, and repre-

Store

Store	Primary Key
SoreType	Categorical
Assortment	Categorical
CompetitionDistance	Number
CompetitionOpenSinceMonth	Number
CompetitionOpenSinceYear	Number
Promo2	Categorical
Promo2SinceWeek	Number
Promo2SinceYear	Number
PromoInternal	Categorical

←

Train

Store	Foreign Key
DayofWeek	Categorical
Date	Datetime
Sales	Number
Customers	Number
Open	Categorical
Promo	Categorical
StateHoliday	Categorical
SchoolHoliday	Categorical

Figure 6-2: A summary of the Rossmann dataset types and relationships.

sents a log of sale information for the Rossmann franchise. The `Store` table represents a store in the Rossmann franchise, and the entries in the `Train` table represent a day in a specific store.

Telstra

The Telstra dataset is the third Kaggle dataset [16]. It is summarized in Figure 6-3 and represents the severity of different types of mobile phone network outages in areas serviced by the Telstra company.

Biodegradability

The Biodegradability dataset is from an online relational database repository [5]. Its schema is summarized in Figure 6-4. The dataset conveys information about different molecules and their atomic makeup. Atoms can also be a part of different bonds and atom groups.

Mutagenesis

The Mutagenesis dataset is also from an online relational database repository [7], and its schema is summarized in Figure 6-5. This dataset conveys the same type of information as in the Biodegradability dataset; molecules are made up of atoms, which can also be involved in different bonds.

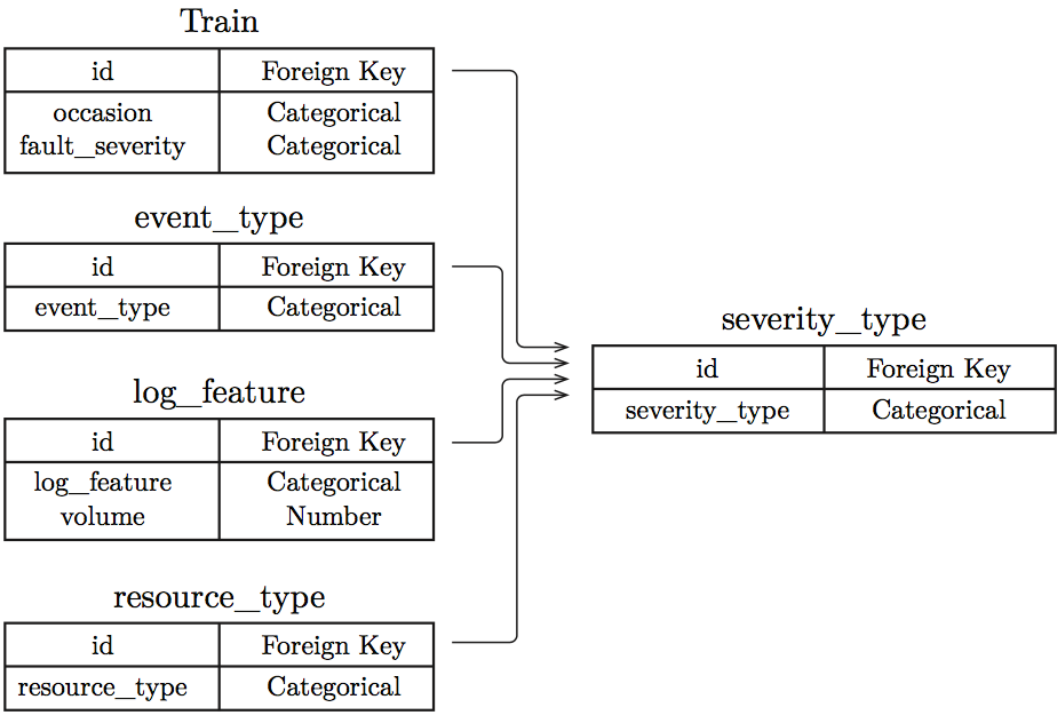

Figure 6-3: A summary of the Telstra dataset types and relationships.

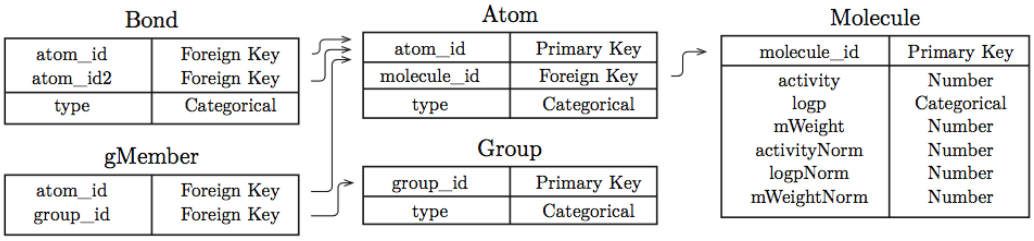

Figure 6-4: A summary of the Biodegradability dataset types and relationships.

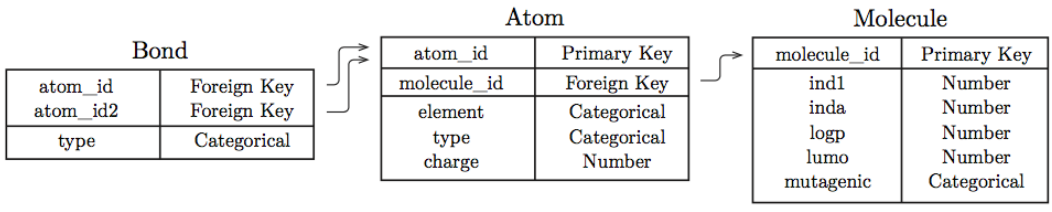

Figure 6-5: A summary of the Mutagenesis dataset types and relationships.

6.3 Methods

Ultimately, the Automatic Data Element Linking system is meant to be run end-to-end, filling in all the information necessary to build a relational schema. However, in order to evaluate the performance of individual algorithms, we looked at each in isolation, and derived separate metrics for the different algorithms' success. In this section, we describe the methods used to train and validate the different algorithms.

6.3.1 Training

Training applies to two out of the four algorithms - type detection and relationship discovery. For these two algorithms, part of the algorithm involves applying a decision tree classifier that has been trained on a subset of the dataset. Ideally, the training set contains approximately half (or fewer) of the available data points, and includes a variety of example types.

Type Detection

In type detection, we want to choose a set of datasets that:

1. Has enough datapoints.
2. Contains examples of classifications for all 4 different types (id, number, date-time, categorical).

From the summary in Table 6.1, we see that the Airbnb dataset has 49 fields, which is around half of the total number of fields across all five datasets. Additionally, from Airbnb's schema summary in Figure 6-1, we can see that the Airbnb dataset encompasses all the different types. Furthermore, certain types, such as Number, contain examples of different subtypes within them. In the `Users` table, we see that `age` is a Number with subtype Integer. On the other hand, a field like `lng\_destination` in the `Countries` table, which represents longitude, would be a Number with subtype Float. Similarly, the Airbnb dataset also contains many Categorical fields, some of which are Boolean (`age`) and others which are not (`language`).

The number of fields in the Airbnb dataset, as well as the variety of types and

subtypes included in these fields, make it a good candidate for training a type classifier.

Relationship Discovery

While discovering relationships, we have a more limited training set, as each dataset includes relatively few inclusion dependencies and foreign key relationships. We train this classifier on the inclusion dependencies: an inclusion dependency tells us if a certain column's values are a subset of (included in) the set of values of another table's primary key.

The best dataset to include in this training set is Rossman, as it has seven inclusion dependencies, and therefore seven training points, only two of which are actual foreign key references. In most of the other datasets, all of the detected inclusion dependencies are also foreign key references, so they only provide positive examples. Rossman gives us a good baseline for which kinds of inclusion dependencies are *not* foreign key references.

For example, there is an inclusion dependency that is not a foreign key between the `DayOfWeek` field in the `Train` table and the `Store` field in the Store table (Figure 6-2). This is because integers are used to represent both of these fields; the `DayOfWeek` column uses integers [1,7], which is a subset of [1,1115], the integers used to identify stores. However, this inclusion dependency is merely coincidental and not meaningful, because the set of values [1,7] is very common.

To complete the training set for relationship detection, we also include the Airbnb dataset. This provides three more positive examples for a more balanced training set.

6.3.2 Testing

To evaluate the performance of individual algorithms, we created a testing suite that takes as an input the path to the directory containing a the files corresponding to a dataset. We expect this dataset to contain a manually annotated meta.json file, which we treat as the ground truth for our generated meta.json files. The purpose of the test suite is to take the path to the dataset directory as the only input, and

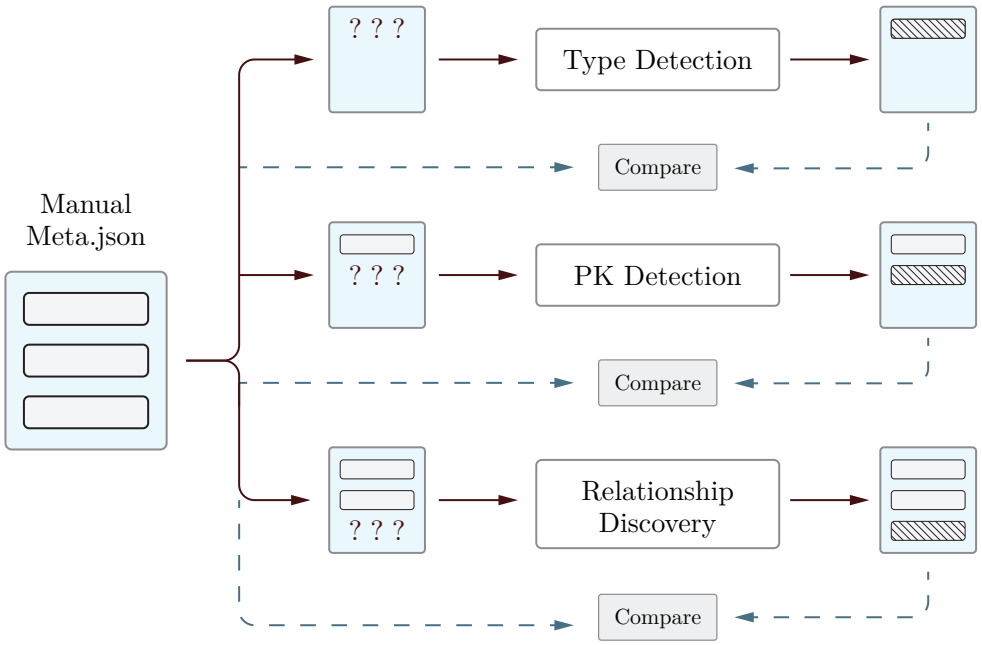

Figure 6-6: This figure depicts the automated testing suite for evaluating type detection, primary key detection, and relationship discovery. Each testing module takes in a partially-filled file of annotations to feed into each respective algorithm. The output is then compared with the original manual annotations file.

automatically validate all four algorithms, writing important metrics to *csv* files as output. Once a repository of manually-annotated datasets has been collected, we can pass this list into the test suite and collect more general aggregate results.

We also note that there exist some dependencies between algorithms. Misclassifications in one algorithm could have an effect on the accuracy of the classifications of a later algorithm. For each algorithm, we assume that all the information that precedes it in the Automatic Data Element Linking flow is correctly identified. For example, this means that when testing Relationship Detection, we assume that the types, subtypes, and primary keys have all been correctly identified. As a result, we only evaluate the work done by the algorithm at hand.

To provide this modularity in testing, the test suite automatically creates partially-filled meta.json files, which contain only a subset of the 'ground truth' information. We generate four different partially-filled 'ground truth' meta.json files, one each for input into the four different modules. The input meta.json file for type detection assumes that the overall dataset structure is filled in, including the table names, the field names, and the number of rows; the input meta.json file for primary key detection has all of its field types and subtypes populated, and so on. The API for accessing the test suite is explained in further detail in Appendix B.

6.4 Results

6.4.1 Type Detection

From the results in Table 6.2, the most mistakes in type classification involve fields labeled *id* and *number* being classified as something else. The *id* fields were mistakenly classified as either number or categorical. However, *id* misclassifications aren't very concerning, as most of these *id*s are primary keys or foreign keys, and will be identified in the following algorithms; when a field is identified as a primary key or a foreign key, the type is updated to *id*. The *number* fields that were misclassified were all incorrectly labeled as *categorical*. This misclassification occurred even in the

Dataset	id	number	datetime	categorical
Airbnb	1/6	1/8	1/7	0/28
Rossman	3/4	2/5	0/2	0/16
Telstra	0/6	0/1	0/0	0/7
Biodegradability	0/8	0/6	0/0	0/3
Mutagenesis	0/5	0/3	0/0	0/6

Table 6.2: Type misclassifications for each dataset, broken down by type. The values for each column show the ratio of the number of fields that should have been classified as the type indicated, but were instead classified as something else, over the total number of fields of that type.

`Airbnb` dataset, which was part of the training set. The reason this is such an easy misclassification is because integers, such as [1-10], are commonly used to represent different categories. Also, sometimes it is objectively unclear, even through human intuition, whether to classify a field as an integer or as categorical.

We see the case of numerical fields misclassified as *categorical* in the `Rossman` dataset. An example of this is with `CompetitionOpenSinceYear`, which represents the year a specific competitor opened up. The year is an integer, but there is a relatively small set of year options included in the dataset, as most of the competitor stores opened in the years 2000 - 2016. Furthermore, there is another field, `CompetitorOpenSinceMonth`, that represents the month in which the competitor store opened. It would be intuitive to classify the set of integers representing months as *categorical*, since there are only 12 months and the value of this field is one of 12 options. Because there is a fine line between *number* and *categorical* when it comes to integers that represent categories, the type-classification decision tree seems to misclassify this case the most often.

Semantic features, such as a column name containing the substring 'date' or 'time,' seem to be very powerful, as most *datetime* fields had some variation of those substrings and were classified correctly. Additionally, most *id* fields containing the substring '_id' were also classified correctly, despite confusion occurring for other, not obviously-named, *id* fields.

Dataset	# False positives
Airbnb	$0/4$
Rossman	$0/2$
Telstra	$1/5$
Biodegradability	$0/5$
Mutagenesis	$0/3$

Table 6.3: Primary key results: False positives for each dataset - the number of fields that were not primary keys, but were classified as primary keys, over the total number of primary keys.

Dataset	Missed FKs
Airbnb	$0/3$
Rossman	$0/2$
Telstra	$0/5$
Biodegradability	$3/5$
Mutagenesis	$0/3$

Table 6.4: Foreign key relationship results: The number of missed foreign keys over the total number of foreign keys per dataset.

6.4.2 Primary Key Detection

The results for primary key detection are shown in Table 6.3. All of the manually-identified primary keys were correctly classified as primary keys. We saw one false positive in the `Telstra` dataset, where a field that was not labeled as a primary key was classified as a primary key. Specifically, this was the `id` field of the `train` table. In this case, the `train` table did not have a labeled primary key, as it is not a mandatory label. The primary key `id` identified by the algorithm is still a valid primary key, despite not being labeled as one. Overall, the primary key algorithm performed very well.

6.4.3 Relationship Discovery

Looking at the relationship discovery results in Table 6.4, the only dataset that contained missed foreign keys was `Biodegradability`. This dataset has a total of five foreign key relationships. However, only three inclusion dependencies, and therefore

three candidates for relationships, were found; for the two foreign keys contained in the gMember table, valid inclusion dependencies were not found. In the atom_id foreign key of the gMember table, there were 72 values present in the foreign key column of the gMember that were not present in the primary key column of the Atom group. For the group_id foreign key, there were 863 values present in the foreign key column of the gMember group that were not present in the primary key column of the Atom group. Such a large number of discrepancies means that a significant number of gMember entries were not referencing valid Atoms or valid Groups, and that perhaps those data entries are invalid. Our foreign key detection algorithm builds in a small tolerance for error, so it would accept if a small fraction ($< 1/10$) of the values were mismatched, but such a large error is not tolerated.

The three remaining foreign keys in Biodegradability were classified as inclusion dependencies, and therefore became candidates for a foreign key relationship. However, the decision tree classifier incorrectly classified one of the inclusion dependencies (the atom_id in the Bond table) as not a foreign key. Interestingly enough, there is another field in the Bond table that is very similar to atom_id – atom_id2, representing the second atom in the bond – and that column was correctly classified as a foreign key. One possible explanation is that one of the features in the decision tree classifier expects that the values in a foreign key field cover the majority of the values in the corresponding primary key field. While that assumption is true for atom_id2, which covers 95.2% of the primary key values, the atom_id field only covers 45.7%, making it a less likely target for a foreign key. There are other features – such as one that assumes the more times a field is referenced, the more likely it is to be a primary key in a foreign key relationship – that would categorize the atom_id as a likely foreign key relationship. However, we did not train on examples that activated those specific features, so the decision tree likely does not rely on that feature very much.

Given that the misclassification of atom_id is the only true error in the relationship discovery phase, the overall accuracy is relatively high. Moving forward, we will train on more datasets, which will allow the decision tree to consider other features more

heavily, creating a more balanced classifier.

6.5 Towards Improving Our Learning Algorithms

Looking forward, our goal is to accumulate a larger number of manually-annotated `meta.json` files to improve our classifiers and obtain more general results.

To this end, we are in the process of creating a platform that will collect manual annotations on different levels. Some manual annotations will be based on table and field names alone (and perhaps a written description of the dataset), to see how annotators perform based on purely semantic and contextual information. Other manual annotations will be collected by showing annotators both contextual/semantic information and raw data values. The proposed interface and design for this platform is outlined in Appendix C.

We will make this platform available on Mechanical Turk, and crowdsource the annotations to create a repository of manual baseline annotations. The annotation collection will be a continuous process, and once a dataset has a complete set of annotations, we will feed the manual `meta.json` file directly into our testing framework to generate feedback for our algorithms.

Chapter 7

Towards Automating Data Science Endeavors

In this chapter, we demonstrate the Automatic Data Element Linking's contribution to a fully automated data science process. Specifically, we take the schema generated from Automatic Data Element Linking and feed it into two endpoints: Deep Feature Synthesis and the Synthetic Data Vault. This section will accomplish two things:

1. Show that the information identified in Automatic Data Element Linking is sufficient for existing platforms that take in human-structured datasets and metadata.
2. Evaluate the impact of the information extracted by Automatic Data Element Linking on the performance of these platforms.

7.1 Motivating Examples

Imagine a data scientist who is looking to develop features for the Airbnb dataset laid out in Figure 6-1. She would likely notice that many different entries in the `Sessions` table correspond to the same `User`. As a result, she would use information about a session entry's corresponding user entry when creating a feature for fields in the `Sessions` table. Additionally, if she were creating a feature for the `action_type` in `Sessions`, she would note that different values represent different types of actions that

the user took during a particular session. She would then make use of the knowledge that `action_type` is categorical in order to create features for `action_type`, perhaps using a conditional distribution that depends on which action type category an entry falls into. In this data science problem of creating informed features, a data scientist makes use of the two important pieces of information captured by the Automatic Data Element Linking process - types and foreign keys.

Now, imagine the same data scientist wants to create a generative model of a table of class times for her university. She hasn't looked at all the entries in the table yet, but she knows that the `startTime` field must necessarily come before the `endTime`, as classes must start before they can end. To make her generative model accurate, she creates a new random variable, calculated by `endTime - startTime`. Instead of sampling from a generative model of `endTime` and potentially getting `endTime` values that are before the corresponding `startTime` value, she models this new random variable that represents the duration of the class. In the end, she transforms the duration back to `endTime` by adding it to `startTime`. In this generative modeling problem, a data scientist identifies the same hard constraint information that is identified in Automatic Data Element Linking. Although she does not make it explicit, she uses her contextual understanding of the problem to decide what variables to model.

As we have shown in the above examples, the information extracted by Automatic Data Element Linking is analogous to the information understood by data scientists when they approach data with implicit and explicit relations. To illustrate this point in the setting of automated processes, we will feed our results into two endpoints. The two platforms we will use as endpoints are Deep Feature Synthesis [14], an algorithm that automatically generates features for a given relational dataset, and Synthetic Data Vault [17], a system that creates a generative model of a relational dataset for the purposes of creating synthetic data. Both Deep Feature Synthesis and Synthetic Data Vault already aim to automate their respective parts of the data science process. By allowing for the automation of relational schema generation, Automatic Data Element Linking would allow these platforms to begin from raw data, extending this automation even further than before.

7.2 Deep Feature Synthesis

Deep Feature Synthesis (DFS) is an algorithm that utilizes the relationships between fields to automatically generate features for data prediction problems [14].

7.2.1 Connecting the Pieces

The DFS algorithm calculates two types of features: *entity* features and *relational* features. *Entity*-level features are calculated based on some transformation of the original data. These transformations depend on the field type, which can be *numeric, categorical, timestamp,* or *freetext* in DFS. The field type determines which transformations and functions can be applied to the values of that field, and, as a result, what *entity* features are discovered.

Relational features are derived by analyzing relationships between entities. DFS utilizes knowledge of data dependency and makes use of information about the parent entity to create features about the children [14].

As a result, DFS relies heavily on knowing the higher-level field types, as well as the foreign key relationships between fields. The Automatic Data Element Linking algorithms create a `meta.json` file that contains those same pieces of information. By feeding the pieces of information extracted by Automatic Data Element Linking into DFS, we can automate the feature discovery process beginning with the raw data stage. This removes the reliance on human intuition to understand and annotate a dataset and its relationships.

7.2.2 Experiment

In order to feed the `meta.json` file into DFS, Max Kanter created a script to translate the ontology of our `meta.json` into that of DFS. This script only needed to convert certain vocabulary and structure, as the information required by DFS is the same.

Because DFS needs a prediction problem in order to evaluate its features, we used the Kaggle predictive problems from the three Kaggle datasets in Section 6.2, which are summarized in Table 7.1. In these predictive problems, one specific field of one of

Dataset	Entity to predict	Table of entity
Airbnb	`country_destination`	`Users`
Rossman	`Sales`	`Train`
Telstra	`fault_severity`	`Train`
Biodegradability	`logp`	`Molecule`
Mutagenesis	`mutagenic`	`Molecule`

Table 7.1: The prediction problem for each of these datasets is to predict the value of the specified field, which is part of the listed table.

the dataset's tables is chosen to be to predicted. To create a training dataset and a testing dataset, we split the given tables, and used k-folding cross-validation to create a model. For each of the datasets, the experiment went as follows:

1. Convert the respective `meta.json`, and feed it into DFS, along with the corresponding prediction problem.
2. Use DFS to generate features.
3. Evaluate the predictive accuracies of those features on the test set.

We repeat this above process twice, once with the `meta.json` generated from Automatic Data Element Linking, and once with the manually-annotated "ground truth" `meta.json`.

7.2.3 Results

In Section 6.4, we saw that the different datasets' `meta.json` files contained different results. Some had types that were misclassified, while others had relationships that were missed. Because of this variation, we can analyze how these misclassifications and false positives affected the feature generation process, and gain insight into which algorithm (type classification, primary keys, relationship detection) had the most impact on the generated features.

The predictive accuracy results are presented in Table 7.2, and we can use these to evaluate the effectiveness of the features that resulted from a given `meta.json` file. In our results, we see that the generated `meta.json` files resulted in comparable or even better performance, except for in the case of Biodegradability. In Section 6.4.3,

Dataset	Control Predictive Accuracy	ADEL Predictive Accuracy
Airbnb	0.8124	0.8124
Rossman	0.6115	0.6117
Telstra	0.5695	0.6007
Biodegradability	0.3015	0.2470
Mutagenesis	0.7587	0.7878

Table 7.2: The predictive accuracy results of the control and the Automatic Data Element Linking (ADEL) -generated schema for each dataset.

we saw that Biodegradability was the only dataset to have foreign key references that were not found. Specifically, three out of the five existing foreign keys were not identified. This correlation of missed foreign key references and a lower prediction accuracy suggests that the identification of foreign key references is important for feature discovery. The misclassification of a few types, specifically when *number* fields are misclassified as *categorical* has less of a noticeable impact.

7.3 Synthetic Data Vault

The Synthetic Data Vault is a system that creates a generative model of a relational database using a multivariate modeling algorithm. This model can then be used to generate synthetic data, which is data that upholds the underlying mathematical properties present in the original data [17]. The generated model can also be made available to people who do not have access to the original data, essentially allowing outside parties work on the dataset without compromising any privacy issues that may arise from sharing the original dataset.

7.3.1 Connecting the Pieces

From an automation standpoint, the Synthetic Data Vault (SDV) is completely autonomous, and involves no user intervention when learning the generative model from an existing schema. The user can then interact with SDV through its API endpoints. For example, a user can request to synthesize data for a specific table.

However, SDV takes in a schema file in order to properly model each field [17]. More specifically, each field's type determines which kinds of models can be applied to that field. The foreign key relationships between fields, which also represent data dependencies, inform parts of the multivariate algorithm; the distribution of parent fields often depend on the distribution of values in their children. The schema file is integral to SDV, and incorrect types or relationships would result in incorrect models and unusable synthetic data.

Creating the schema file requires the user to completely understand and annotate the dataset. This task becomes more and more arduous the more fields and tables there are. With the Automatic Data Element Linking algorithms, a user can take a folder of csv files and automatically generate synthetic data from there, instead of first taking the time to understand all the fields and identify potential references between tables. The Automatic Data Element Linking algorithms create a schema file which feeds directly into SDV, giving the user control over the process while also relieving them of the tasks of understanding how a dataset is structured and annotating it.

7.3.2 Hard Constraints

Because the Synthetic Data Vault relies entirely on generative modeling, there are some constraints that the SDV does not enforce, including hard constraints. For example, imagine there is table of `Classes` with a field `startTime` and a field `endTime`. There is an implicit constraint, satisfied by "real" data, that the `endTime` is always after the `startTime`. While the SDV's generative modeling is unlikely to produce `endTime` values that before than the corresponding `startTime` in an entry, there is no guarantee that a violation will not be produced. Those violations are important, because applications using SDV would be expected to be able to interact with synthetic data in the same way they would interact with real data. Therefore, if they were relying on the assumption that `endTime` $>$ `startTime`, this should also hold with the synthetic data.

As the Automatic Data Element Linking process detects hard constraints, we thought to also integrate hard constraint enforcement into the SDV workflow. To

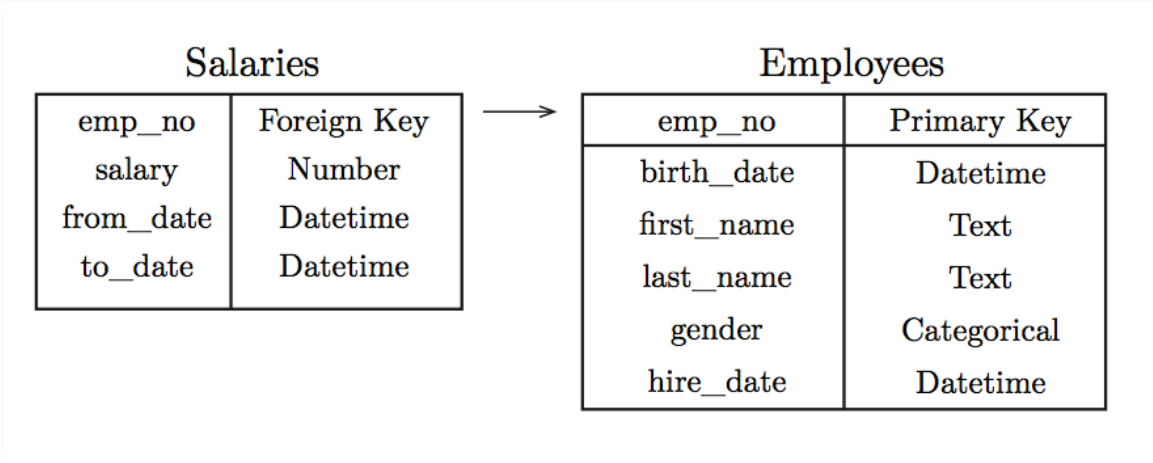

Figure 7-1: The Employees dataset, used for analyzing hard constraints.

accomplish this, we applied follow this procedure:

1. Before SDV reads in the data, the user calls `apply_pre_transform` on the dataset's `meta.json` file. This transform function will, for each table that has hard constraints, apply the corresponding transformation on the relevant fields. SDV will read in the transformed data and variables. This is where `endTime` would be mapped to `endTime - startTime`.
2. The user then uses SDV to generate models and synthesize data. SDV generates synthetic data in the transformed variables. In our example, SDV would model the difference between `endTime` and `startTime` (a duration) instead of `endTime`.
3. Before SDV returns any data, it calls `apply_post_transform` to apply the inverse transformation. As a result, the returned data that the user sees is data that corresponds to the original fields.

7.3.3 Experiment

We applied this modified SDV to a subset of an Employee dataset from the online relational dataset repository [6]. In this dataset, which is summarized in Figure 7-1, there is an `Employees` table and a `Salaries` table. In the `Salaries` table, each entry represents a salary of an employee from a certain `from_date` to a certain `to_date`. The hard constraint $A > B$ exists between the `to_date` and `from_date` fields.

	% Violation
Enforced HC	0
Control	30.6

Table 7.3: SDV Results: The percentage of violations in SDV-generated synthetic data, with the hard constraints enforcement and without.

First, we run Automatic Data Element Linking to fill the `meta.json` file, which contains the hard constraint. Then, the augmented SDV flow pre-processes the raw data according to any hard constraints. Our control is the manually-annotated `meta.json`, with no hard constraints.

The experiment:

1. Instantiate a new SDV model using respective `meta.json` files.
2. Use the SDV model to synthesize 100 new `Employees` rows, and then synthesize 500 `Salaries` entries.
3. Calculate the percentage of violations = 100 $\times$ # times hard constraint is violated / # total synthetic entries in `Salaries`.

7.3.4 Results

With the normal SDV generative modeling and the control `meta.json` annotations, the percentage of hard constraint violations is 30.6% (153 violations out of the 500 generated rows). Using the `meta.json` file generated from Automatic Data Element Linking, which does consider hard constraints, the percentage of violations drops to 0%.

Chapter 8

Conclusion and Future Work

8.1 Future Work

Moving forward, we are looking to expand the repository of datasets and corresponding manual annotations we have available for testing and validation. This will allow us to gain insight into the generalizability of our algorithms, and allow us to iterate on the existing modules. Additionally, once we have a larger repository of datasets to pull from, the training of our machine learning classification models will become more accurate. In the future, if we have a large enough dataset to select the training set from, we can either arbitrarily select or implement methods like k-folding.

In addition to expanding our repository of datasets and manual annotations, we are also looking to scrape datasets off of websites known to host datasets, such as Kaggle. We have an initial prototype of a process that automatically visits websites every day to check for new datasets. This script must:

1. Determine if datasets are relational and multi-table.
2. Determine what kind of learning problem is being asked (if part of a competition like Kaggle). This can be prediction, natural language processing, or vision.
3. If a prediction problem, determine what entity is being predicted.
4. Download and structure data files in a standardized manner.

With this information, we can not only collect statistics on metrics regarding

misclassifications, but we can continue the work done in Section 7.2, and evaluate the performance of predictions generated from automatically generated schemas.

8.2 Conclusion

In this thesis, we have presented a system to emulate human intuition in the understanding of relational datasets. We have broken down and formalized this intuition into four main steps, and have created algorithms for each one. In the end, we demonstrate the contribution of our platform by linking it to existing systems, taking one step closer to a fully automated data science process.

Appendix A

Synthetic Data Vault

The Synthetic Data Vault (SDV) is a system that automatically generates synthetic data for relational databases by creating a generative mode from the raw data, and then sampling from that model. The goal of SDV is to facilitate data science endeavors, specifically those that require access to a large amount of potentially sensitive data. SDV produces data that doesn't reveal any sensitivities of the original data, but still captures the mathematical relationships present in the original context [17].

A.1 Overview

In the Synthetic Data Vault, the user must first collect and format the relational dataset into a collection of tables in which each table entry is atomic. Then, the user specifies the structure of the provided dataset, providing information about the type and constraints of the data in each table, as well as the relationships between tables. The format and the information required by the SDV is contained in a json file called `meta.json`, which is outlined in detail in Section 3.1. Then, the SDV learns a model of the raw data, and samples from the model to synthesize data [17].

In this section, we describe the feedback collected on the original SDV API, and the implemented changes to make SDV more usable.

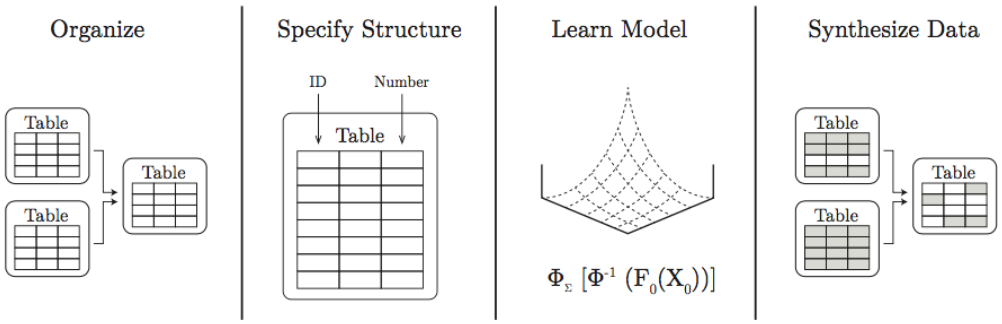

Figure A-1: The SDV workflow.

A.2 SDV API

To access the SDV functionality, the user first imports the database package from the SyntheticDataVault code repository. By instantiating a new Database object, the user loads the metadata file and generates the underlying model. `'meta.json'` is the metadata file and `'summary.json'` is where the model parameters will be written to, if the model has not yet been created, or where the model paramters will be read from, if the model already exists.

```
from database import Database
db = Database('meta.json', 'summary.db')
```

Then, the user gets a handle on the table for which they want to create synthetic data, synthesizes the desired number of rows, and then samples the data to keep the same number of rows that existed in the raw data.

```
users = db.get_table('users')
for N iterations:
    users.synth_row()
users.sample_synth_data()
```

To synthesize tables that have a foreign key into the `users` table, we would call `users.synth_children`.

In summary, the main entry points into the SDV API are

1. `database = Database('meta.json', 'model_parameters.db')`
2. `table = database.get_table('table_name')`
3. `table.synth_row(...)`
4. `table.synth_children(...)`

A.3 API Feedback

After collecting feedback from both computer scientists and statisticians, we received comments in four main areas.

First, the interface is not transparent enough; it is not obvious what the code is doing after each command. For example, instantiating a database object will also read in the metadata and create the model for the dataset, which is not obvious from the command name.

Second, the API entry points may be intuitive from a computer science viewpoint, but not from a statistician's. A statistician or data scientist is used to working with dataframe objects, and handling this new `Database` object is unfamiliar. A data scientist would need to be able to easily view how the data is being processed, as well as what data is being generated.

Third, the SDV still requires users to know the overall structure of the dataset. For example, imagine a dataset with two tables, `users` and `sessions`. The `sessions` table has entries of a web browsing session for a specific user and has a column called `user_id`. This `user_id` of the `sessions` table is a foreign key into the `id` column in the `users` table. In order to generate synthetic data for the `sessions` table, we must call |users.synth_children|, as `sessions` is a child table of `users`. However, it is unreasonable to expect everyone using SDV to know the exact structure of the underlying dataset to generate synthetic data.

Finally, the existing SDV code still requires the original raw dataset to be read in. However, this undermines the goal of SDV to provide users with the ability to reveal synthetic data without compromising the potentially sensitive original data.

A.4 API Enhancements

In our API Enhancements, we addressed each of the issues raised in the SDV Feedback.

To make the API interface more transparent and intuitive for a data science use case, we decided to view the code module as a SDV black box. This black box would have very high-level buttons that a user could push, and would return outputs in formats that are familiar to a data scientist, such as dataframe objects. Specifically, this meant packaging all functions into a `SDV` module, and adding high level steps to load the metadata, build the model, synthesize data, and view / write out the data.

Our second focus was removing reliance on the original dataset. Users can now synthesize data directly on any desired table, without having to call `synth\_children` on any parent table; instead, all of the dependencies are taken care of behind the scenes. Furthermore, all reliance on the raw data values are removed – now users can generate a model from the raw data, and pass that model around without sharing the original data.

The API now consists of the following main entry points:

1. `dataVault = SDV('meta.json')`
2. `dataVault.learn_model('model_parameters.db')`
3. `dataframe = dataVault.synth_rows('table_name', number_rows)`
4. `dataframe = dataVault.synth_table('table_name')`
5. `dataframe = dataVault.get_synth_data('table_name')`
6. `dataframe = dataVault.get_model_params()`

With these API changes, the SDV codebase aims to be more intuitive and better suited for its use cases.

Appendix B

ADEL API

In this chapter, we outline the main API endpoints necessary for training, classifying, and validating results.

B.1 Training

For the algorithms that rely on a machine learning classifier, we provide methods for training a decision tree on user-specified datasets. This applies to decision trees for both type detection and relationship detection.

- **train_type_decision_tree**: Takes in a list of manually-annotated (correct) meta.json files. Trains a type-classification decision tree on the corresponding datasets.

- **train_ref_decision_tree**: Takes in a list of manually-annotated `meta.json` files. Trains a decision tree that determines whether or not an inclusion dependency is a foreign key relationship.

B.2 Schema Generation

Once we have the necessary classifiers, we can call the following functions to incrementally generate `meta.json` files for the desired datasets. All of these functions

assume that the `meta.json` file is in the same directory as the dataset csv files.

- **classify_field_types**: Given an empty `meta.json` file for a specific directory, extracts relevant features and detects the type and subtype of each field for each table. Writes to the user-specified output file, so the user can either overwrite the old `meta.json` file or create a new one.
- **find_PKs**: Given a `meta.json` file with types and subtypes filled in, finds the most likely primary key for each table in the corresponding dataset. Writes updated metadata to the user-specified output file.
- **discover_relationships**: Given a `meta.json` file with types and PKs filled in, finds all inclusion dependencies and classifies each candidate pair as a FK reference or not. Updates metadata with detected foreign key references and writes the updated metadata to the user-specified output file.
- **discover_hard_constraints**: Given a `meta.json` file, finds if any of the pre-specified hard constraints hold within any of the dataset's tables. Updates metadata with detected hard constraints, and writes updated metadata to the user-specified output file.

B.2.1 Hard Constraint Transformations

After hard constraints have been detected, we apply a transformation to the dataset to ensure that the constraints are satisfied during whatever analysis or manipulation that follows. We also provide a post-processing step that transforms the dataset back to its original fields.

- **apply_pre_transform**: Given a `meta.json` file for a specific directory, reads the specified hard constraints and transforms the relevant datatables. Creates new tables with the suffix '.trans.csv'
- **apply_post_transform**: Given a `meta.json` file for a specific directory, reads the specified hard constraints and transforms the relevant synthetic data. Overwrites the synthesized data csv file with the transformed data.

B.3 Testing and Validation

B.3.1 Isolating Modules

The API for generating a complete `meta.json` file was written so that a user can call only the desired modules to fill in incomplete information. For example, if all the field types are known, we can move onto detecting primary keys. To test the accuracies of individual modules, we can also employ this technique, and generate different input files with different amounts of incomplete information to pass into the respective methods.

- **gen_empty_metas**: Generate an 'empty' `meta.json` file that has contains all the obvious information – desired format, all the table and field names, as well as number of rows and number of unique values.
- **create_input_for_PK**: Generate a `meta.json` file that contains all field types and subtypes, in addition to the obvious information.
- **create_input_for_rel**: Generate a `meta.json` file that contains all field types and subtypes, primary keys for all tables, and all the obvious information.
- **create_input_for_HC**: Generate a `meta.json` file that contains all field types and subtypes, primary keys for all tables, relationships between all tables, and all of the obvious information.

B.4 Generating Results

Next, we describe the methods used to gather insight on the accuracies of the above methods.

- **compare_types**: Given a baseline `meta.json` file and a generated `meta.json` file, calculate number of incorrectly classified types, break down the number per type.

- **compare_PKs**: Given a baseline `meta.json` file and a generated `meta.json` file, calculate the number of incorrectly identified primary keys, and break it down per type.

- **compare_refs**: Given a baseline `meta.json` file and a generated `meta.json` file, calculate the number of incorrectly identified foreign key relationships, and break it down into false positives and false negatives.

- **compare_HCs**: Given a baseline `meta.json` file and a generated `meta.json` file, calculate the number of incorrectly identified hard constrains, and break it down into false positives and false negatives..

Appendix C

Collecting manual annotations for data

The goal of this web application is to collect annotations on a dataset. A dataset consists of one or more tables, each of which is represented by a *csv* file. The input to the system will be a folder of *csv* files, where each *csv* file represents a table. Each table will have many columns of data, with each column representing a field in a table. The annotations we want to collect are:

1. The types of each column in a table (*id*, *datetime*, *number*, *categorical*, or *text*).
2. The primary key of each table (one per table).
3. The foreign key relationships between tables (from one column of a table to a column in a different table).
4. Hard constraints within a table (`field_A` > `field_B`).

C.1 Type Classification

Here, we will ask a user to annotate types, with the UI shown in Figure C-1. The view will display information for one table. For each column in the table, we will display the column name, as well as the first 10 entries in that column to let the user preview the data. At the bottom of each column, there is a dropdown menu populated with the five types (*id*, *number*, *datetime*, *categorical*, *text*). There is a horizontal scroll in

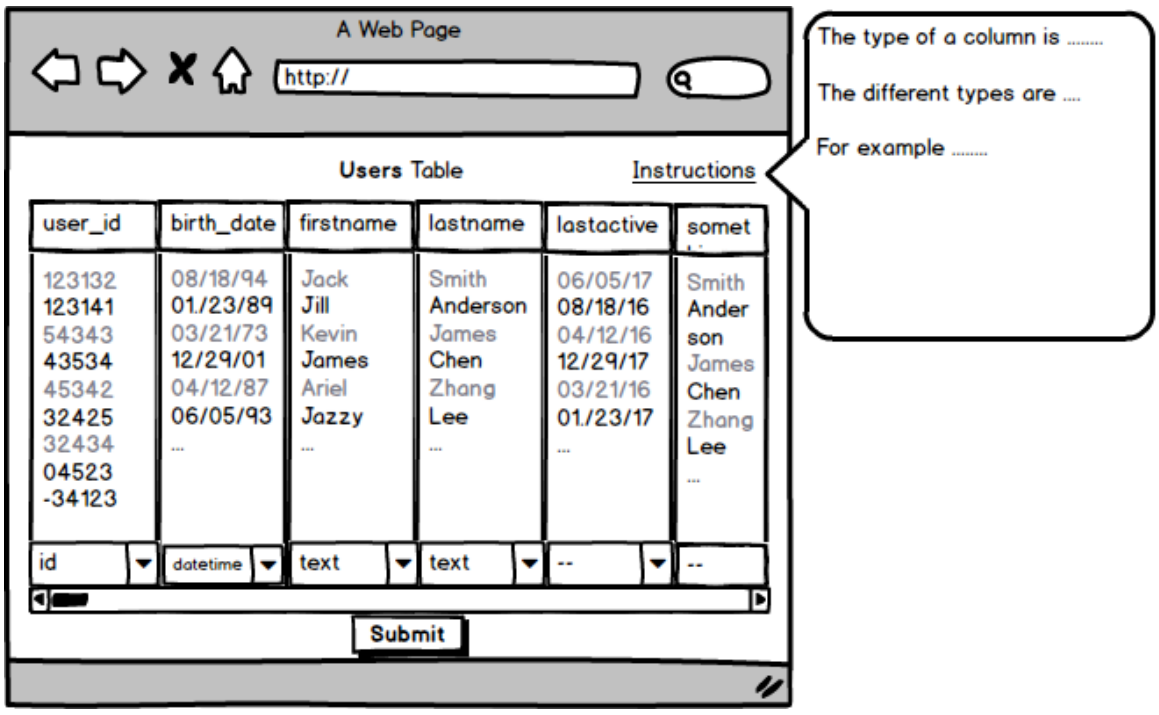

Figure C-1: UI to collect annotations for a table's types.

type_annotation_id	timestamp	dataset	table	field	chosen_type

Table C.1: The information that should be captured in a "type_classification" database when an annotation is submitted.

case all of the columns do not fit across the width of the screen.

The user must select types for **all** columns before they can submit.

On the top right, there is an "Instructions" link that opens up a popover or pop up that displays an explanation of what types are and how to complete the task.

When the user clicks submit, we want the information in Table C.1 to be entered into a `type_classification` database.

- `type_annotation_id` : ID of the database entry
- `timestamp`: time the user submitted
- `dataset`: name of the directory that was provided
- `table`: name of the table

pk_annotation_id	timestamp	dataset	table	pk

Table C.2: The information that should be captured in a "pk_classification" database when an annotation is submitted.

- `field`: name of the column
- `chosen_type`: which of the five types the user chose for this column

C.2 Primary Key Detection

Here, we will ask a user to select the primary key for a table with the UI shown in Figure C-2. The view will display information for one table. For each column in the table, we will display the column name, as well as the first 10 entries in that column to let the user preview the data. There is a horizontal scroll in case all of the columns do not fit across the width of the screen. The dropdown at the bottom of the string is populated with all the column names in the table, and "None". Once the user selects the name of the column that he/she thinks is the primary key (could be "None"), the user can submit.

On the top right, there is an “Instructions" link that opens up a popover or pop up that displays an explanation of what primary keys are and how to complete the task.

When the user clicks submit, we want the information in Table C.2 to be entered into a `pk_detection` database.

- `pk_annotation_id`: ID of the database entry
- `timestamp`: time the user submitted
- `dataset`: name of the directory that was provided
- `table`: name of the table
- `pk`: which column the user chose to be the primary key

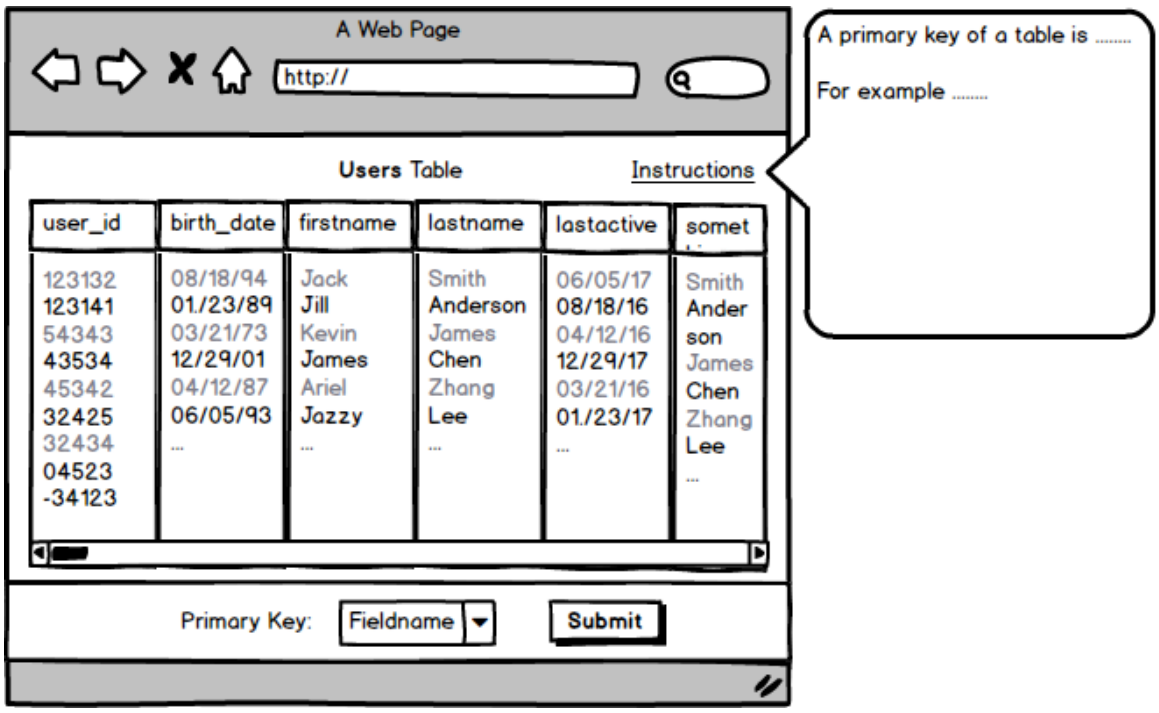

Figure C-2: UI to collect annotations for a table's primary key.

C.3 Relationship Discovery

Here, we will ask a user to identify any foreign key relationships between two tables with the UI shown in Figure C-3. A foreign key relationship is between a column of one table and a column of a different table. The view will display information for two tables. The tables' info are presented side by side, and initially looks like a list of each table's column names. If a user clicks on a column name, that one entry expands (an accordion) to display the first 5 entries from the column data.

At the bottom of the screen, there is a fixed section where the user can enter any foreign key relationships they identify. In this section, there is are two dropdowns so they can select the two columns for which they identify a foreign key (FK) relationship. The values in each dropdown will be all the column names in both tables, formatted as "`field_name` (`table_name`)." Because the FK must be between columns of different tables, once the user selects one dropdown, we can change the options in the second dropdown to only contain field names of the other table.

Once the user adds a FK, it will show up in the "Added FKs" section in the format

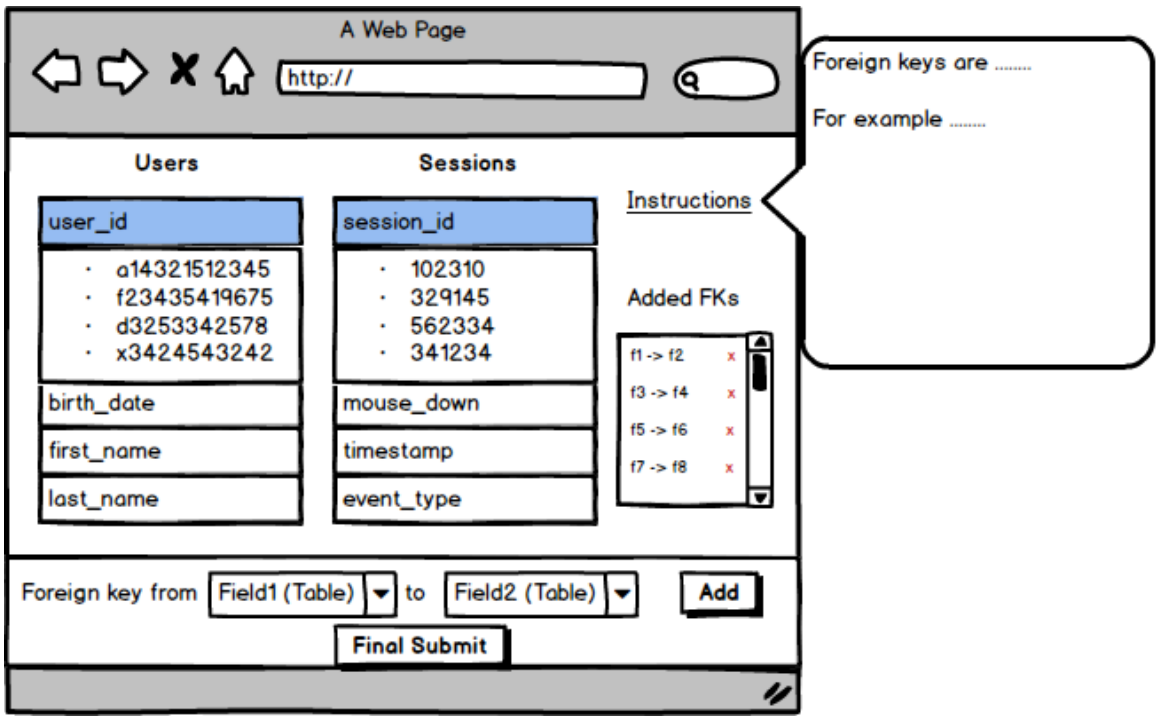

Figure C-3: UI to collect annotations for foreign keys between two tables.

fk_annotation_id	timestamp	dataset	field1	table1	field2	table2

Table C.3: The information that should be captured in a "fk_classification" database when an annotation is submitted.

"`field1(table1)` → `field2(table2)`." There should be an option to remove this entry. The "Final Submit" button must only be clicked when the user has found all FKs they think exist - to ensure this, we should add a confirmation pop up when they click "Final Submit" to say *"Have you entered all the FKs you can find? If not, click cancel and use the Add button to add more."*

On the top right, there is a "Instructions" link that opens up a popover or pop up that displays an explanation of what primary keys are and how to complete the task.

When the user clicks submit, we want the information represented in Table C.3 to be entered into a `fk_discovery` database.

- `fk_annotation_id`: ID of the database entry

- `timestamp`: time the user submitted
- `dataset`: name of the directory that was provided
- `field1`: name of the field the FK is from
- `table1`: name of the table containing field1
- `field1`: name of the field the FK is to
- `table1`: name of the table containing field2

C.4 Hard Constraints Discovery

Here, we will ask a user to identify any hard constraints within a table with the UI shown in Figure C-4. A hard constraint is between either two or three columns in a table. The view will display information for one tables. For each column in the table, we will display the column name, as well as the first 10 entries in that column to let the user preview the data. There is a horizontal scroll in case all of the columns do not fit across the width of the screen.

The types of constraints we allow for are:

I. A $>$ B

II. A + B = C

At the bottom of the screen, there is a fixed section where the user can enter any hard constraints they identify. In this section, there are two types of hard constraints that can be specified. If a user completely fills out either of these constraints and pushes the corresponding 'Add' button, it will show up in the 'Added Constraints' section in the format `fieldA > fieldB` or `fieldA + fieldB = fieldC`, depending on which type of constraint was added. There should be an option to remove this entry. The "Final Submit" button must only be clicked when the user has found all hard constraints they think exist – to ensure this, we should add a confirmation pop up when they click "Final Submit" to say *"Have you entered all the hard constraints you can find? If not, click cancel and use the Add button to add more."*

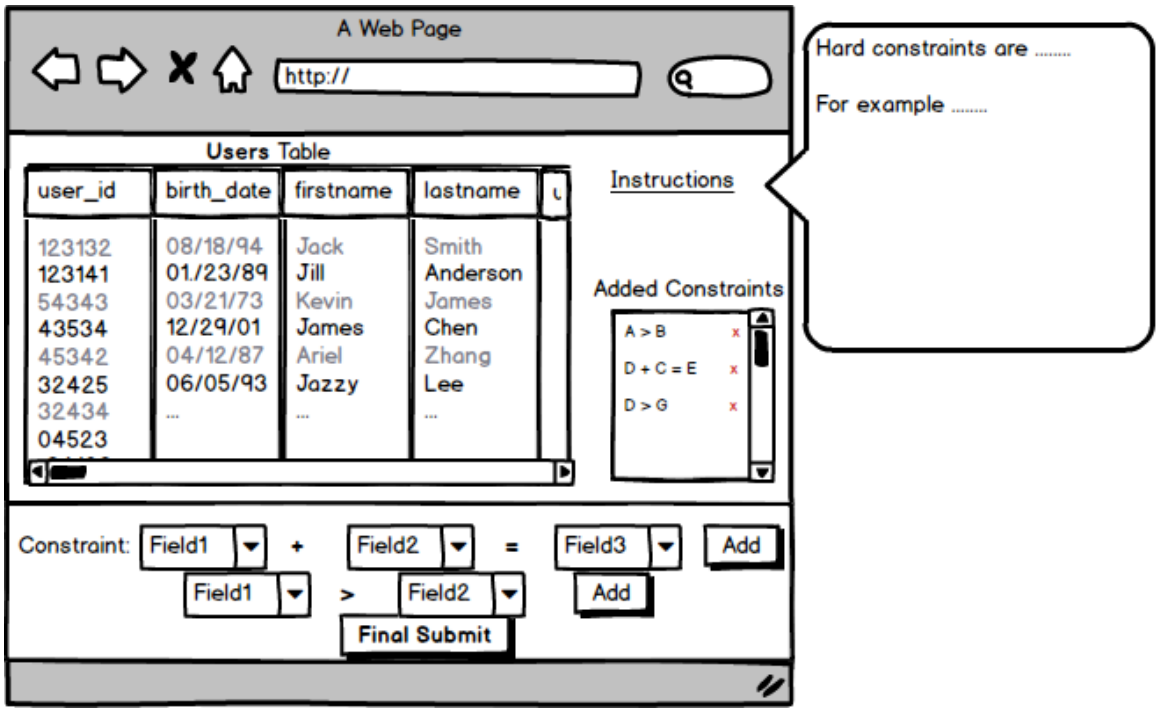

Figure C-4: UI to collect annotations for hard constraints within a table.

hc_annotation_id	timestamp	dataset	fieldA	fieldB	fieldC	type

Table C.4: The information that should be captured in a "hc_classification" database when an annotation is submitted.

On the top right, there is a “Instructions” link that opens up a popover or pop up that displays an explanation of what primary keys are and how to complete the task.

When the user clicks submit, we want the information shown in Table C.4 to be entered into a `hard_constraints_discovery` database.

- `hc_annotation_id`: ID of the database entry
- `timestamp`: time the user submitted
- `dataset`: name of the directory that was provided
- `fieldA1`: name of the first field
- `fieldB`: name of the second field
- `fieldC`: name of the third field, if is the type A + B = C, otherwise None

- `type`: 0 if the constraint is type A $>$ B, 1 if it is of type A + B = C

Bibliography

[1] Apache spark. https://spark.apache.org/docs/1.1.0/api/python/pyspark.sql.sqlcontext-class.html. accessed: 2017-04-26.

[2] pandas. https://pandas.pydata.org/pandas-docs/stable/generated/pandas.dataframe.dtypes.html. accessed: 2017-04-24.

[3] Kaggle airbnb new user bookings. https://www.kaggle.com/c/airbnb-recruiting-new-user-bookings. Accessed: 2016-10-23.

[4] Amir Hassan Bahmani, Mahmoud Naghibzadeh, and Behnam Bahmani. Automatic database normalization and primary key generation. *Canadian Conference on Electrical and Computer Engineering*, 2008.

[5] Hendrik Blockeel, Boris Kompare SasÌŇo DzÌŇerosk and, Stefan Kramer, and Bernhard Pfahringer. Experiments in predicting biodegradability. *Applied Arti cial In- telligence*, 2004.

[6] Employee dataset. https://relational.fit.cvut.cz/dataset/Employee. Accessed: 2017-4-3.

[7] A. K. Debnath, R. L. Lopez de Compadre, A. J. Shusterman G. Debnath, and C. Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 1991.

[8] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. The data civilizer system. *CIDR*, 2017.

[9] Y. V. Dongare, P. S. Dhabe, and S. V. Deshmukh. Rdbnorma: - a semi-automated tool for relational database schema normalization up to third normal form. *International Journal of Database Management Systems (IJDMS)*, 2011.

[10] Peter A. Flach and Iztok Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 1999.

[11] Kenneth HoukjÃęr, Kristian Torp, and Rico Wind. Simple and realistic data generation. *VLDB*, 2006.

[12] MongoDB. https://docs.mongodb.com/manual/core/data-modeling-introduction/. Accessed: 2017-5-3.

[13] Kaggle. https://www.kaggle.com/. Accessed: 2016-10-23.

[14] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. *Data Science and Advanced Analytics*, 2015.

[15] Jan Mot and Oliver Schulte. The ctu prague relational learning repository. *abs/1511.03086*, 2005.

[16] Telstra network disruptions. https://www.kaggle.com/c/telstra-recruiting-network. Accessed: 2016-10-26.

[17] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. *Data Science and Advanced Analytics*, 2016.

[18] Kaggle rossman store sales. https://www.kaggle.com/c/rossmann-store-sales. Accessed: 2016-10-23.

[19] Alexandra Rostin, Felix Naumann, Jana Bauckmann, Oliver Albrecht, and Ulf Leser. A machine learning approach to foreign key discovery. *Research Gate*, 2009.