SDV: An Open Source Library for Synthetic Data Generation

by

Andrew Montanez

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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

© Massachusetts Institute of Technology 2018. All rights reserved.
Abstract

In this thesis, I designed three open source Python libraries with the intention of creating a robust system that can accurately generate synthetic data. The goals of this thesis were to separate the different components in synthetic data generation into their own libraries. We identified these components as consisting of a way to transform the data, a way to model the data, and a way to recursively traverse the data set to model the relationships between the table as well as the data set itself.

Once the libraries were implemented and functioning, we designed a program to run the synthetic data generation process in parallel on subsets of the original data. The goal of this program was to see if the overall modeling time could be reduced by modeling subsets in parallel and then averaging the parameters. In the end, we test how close these averaged parameters are to the original to see if this is a valid modeling technique.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist
Acknowledgments

I would like to start by thanking Kalyan for all of his help and support throughout this project. Not only did he provide great feedback and knowledge through his expertise, but his mentorship and life lessons also proved invaluable.

I would like to thank Carles and Manuel for all of the help with the development of the open source libraries and for teaching me how to be a better software engineer. I would also like to thank Arash and Karol for their assistance with the graphics and tables in this paper.

Finally I want to thank my family for supporting me and my friends for keeping me company and having my back during my pursuit of a higher degree. Mom, Dad, Jack, Michael, Christian, Sam, and John, I could not thank you enough.
Contents

1 Introduction .................................................. 15
   1.1 Original synthetic data vault ................................ 15
   1.2 Motivation for redesign ..................................... 16
   1.3 Thesis roadmap ............................................. 17

2 SDV Redesign ................................................ 19
   2.1 Copulas library ............................................. 20
      2.1.1 Design considerations .................................. 21
   2.2 RDT library ................................................ 22
      2.2.1 Design considerations .................................. 24
   2.3 SDV library ................................................ 25
      2.3.1 Meta file ................................................ 25
      2.3.2 DataLoader ............................................... 30
      2.3.3 DataNavigator .......................................... 31
      2.3.4 Modeler ................................................ 33
      2.3.5 Sampler ................................................ 34
      2.3.6 SDV class .............................................. 35
   2.4 Interactions between libraries ................................ 36
      2.4.1 Initial interaction between RDT and SDV .............. 36
      2.4.2 Interaction between Copulas and SDV .................. 36
      2.4.3 Final interaction between Copulas, RDT and SDV ...... 37
   2.5 Design considerations ...................................... 37
   2.6 Further improvements ...................................... 38
3 Copulas

3.1 Univariate distributions ........................................... 41
3.2 Multivariate copulas .............................................. 42

4 Reversible Data Transforms ...................................... 45

4.1 RDT definition ...................................................... 45
  4.1.1 Transformer design .......................................... 46
4.2 Existing transformers .............................................. 47
  4.2.1 NumberTransformer ......................................... 48
4.3 DatetimeTransformer ............................................. 48
  4.3.1 CatTransformer .............................................. 49
4.4 NullTransformer ................................................... 50
4.5 Contributing to RDT .............................................. 50
  4.5.1 Create transformer ........................................... 50
  4.5.2 Write unit tests .............................................. 51
4.6 Other possible Transformers ..................................... 52
4.7 HyperTransformer and using RDT in a project .................. 52
  4.7.1 HyperTransformer design ................................... 52
  4.7.2 Using the HyperTransformer in a project .................. 54
4.8 Design challenges ................................................. 55
4.9 Future improvements .............................................. 56

5 The Synthetic Data Vault ......................................... 59

5.1 Modeling ............................................................ 59
5.2 Sampling ............................................................ 62
5.3 Sampling non-numeric data ....................................... 65
  5.3.1 Primary and foreign keys (IDs) ............................. 65
  5.3.2 Text and Faker ................................................ 66
5.4 Relationships between tables .................................... 66
  5.4.1 Isolated table .................................................. 67
  5.4.2 Linear .......................................................... 67
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Diagram of how different libraries in SDV interact with each other</td>
<td>20</td>
</tr>
<tr>
<td>2-2</td>
<td>Diagram showing process of creating <strong>DataNavigator</strong></td>
<td>31</td>
</tr>
<tr>
<td>4-1</td>
<td>Diagram of different data types</td>
<td>47</td>
</tr>
<tr>
<td>4-2</td>
<td>Diagram of <strong>NullTransformer</strong> process</td>
<td>50</td>
</tr>
<tr>
<td>5-1</td>
<td>Diagram of how tables are extended with parameters</td>
<td>61</td>
</tr>
<tr>
<td>5-2</td>
<td>Diagram of linear table relationship</td>
<td>67</td>
</tr>
<tr>
<td>5-3</td>
<td>Diagram of tables with multiple children</td>
<td>68</td>
</tr>
<tr>
<td>5-4</td>
<td>Diagram of tables with multiple parents</td>
<td>68</td>
</tr>
<tr>
<td>7-1</td>
<td>Diagram of data being sampled by year and learned by SDV in parallel</td>
<td>80</td>
</tr>
<tr>
<td>8-1</td>
<td>Airbnb data set relationships</td>
<td>84</td>
</tr>
<tr>
<td>8-2</td>
<td>Telstra data set relationships</td>
<td>84</td>
</tr>
<tr>
<td>8-3</td>
<td>Rossman data set relationships</td>
<td>84</td>
</tr>
<tr>
<td>8-4</td>
<td>Biodegradability data set relationships</td>
<td>85</td>
</tr>
<tr>
<td>8-5</td>
<td>Hillary Clinton Emails data set relationships</td>
<td>85</td>
</tr>
<tr>
<td>8-6</td>
<td>NIPS 2015 Papers data set relationships</td>
<td>86</td>
</tr>
<tr>
<td>8-7</td>
<td>World Development Indicators data set relationships</td>
<td>86</td>
</tr>
<tr>
<td>8-8</td>
<td>Walmart data set relationships</td>
<td>87</td>
</tr>
<tr>
<td>8-9</td>
<td>Coupon Purchase Prediction data set relationships</td>
<td>87</td>
</tr>
</tbody>
</table>
# List of Tables

2.1 Date Structures ................................................................. 28  
2.2 Table 2.1 – continued. ...................................................... 29  
4.1 Transformers and their data types ........................................... 47  
8.1 Timing of Modeling and Sampling for Different Data Sets ............... 88  
8.2 Results from ParallelSDV Experiment ....................................... 90  
A.1 Date Sets – airbnb and biodegradability .................................... 98  
A.2 Date Sets – hillary_clinton_emails and nips_2015_papers ................ 99  
A.3 Date Sets – rossman, telstra, and walmart ................................. 100  
A.4 Date Set – world_development_indicators ................................. 101  
A.5 Date Set – world_development_indicators (continued) ................. 102  
A.6 Date Set – coupon_purchase_prediction ................................. 104
Chapter 1

Introduction

As techniques in machine learning get more sophisticated, businesses become more eager to incorporate these techniques into their everyday operations. To get the most out of modern day machine learning algorithms, a huge volume of data is required to train these algorithms; however, when trying to capitalize on the advances of machine learning, some companies face the problem of lacking access to large amounts of data. In 2016, a paper titled "The Synthetic Data Vault: Generative Modeling for Relational Databases" proposed a solution to this very problem[11]. The Synthetic Data Vault (SDV) was a method of recursively modeling a data set so that accurate synthetic data could be generated.

The original SDV showed promising results, but acted more as prototype than official software. Since the paper’s release, interest in such a software has grown tremendously. With that interest came new requests such as other possible modeling techniques, different methods of loading data, and more options for sampling. To address these suggestions and open the way for me, we created SDV: an open source library designed to generate synthetic data.

1.1 Original synthetic data vault

The original synthetic data vault modeled data sets and provided the ability to sample from those models. It required some amount of original data from which to create
the models. After that, it could generate an infinite amount of synthetic data that would be close to indistinguishable from the original.

Simply modeling data isn’t necessarily that difficult. The synthetic data vault was uniquely interesting because it could model the relationships between tables in a data set as well. For example, imagine a data set that has two tables, one for customers of a store and another for the transactions those customers made. It may be the case that certain customers have different purchasing habits than others. Thus to generate realistic data, these purchasing habits should be modeled.

A normal sampling technique would be to model the transactions table and then sample from that model. However, the habits would be captured by modeling the transactions table for each customer, and sampling from this dependent model instead. This is roughly the strategy that the original synthetic data vault used to successfully generate synthetic data.

1.2 Motivation for redesign

Although the first synthetic data vault showed great promise, there was still room for improvements. Not every method had been optimized, and there were desires to add new features, but the code wasn’t created in a way that allowed it to be easily changed.

One major goal of this project was to redesign the code such that, in the future it would be easy to improve, optimize, expand, and contribute to. Providing these features would allow for anyone to add new modeling techniques to the code, improve the efficiency of certain methods and expand the types of sampling methods that the synthetic data vault offered.

For example, there is currently no way to synthesize only specific columns of data. This might be a desirable feature if a user has data for some but not all of the columns. The open source SDV library makes it easy for a contributor to add this feature without damaging any of the already functioning code. This is what makes it valuable.
1.3 Thesis roadmap

Chapter two describes the redesign of SDV along with the other libraries that were created to make it work. This chapter focuses on the APIs of the libraries.

Chapter three describes the technical details of the Copulas library. It focuses on the structure of the classes and the implementations of the methods involved in modeling.

Chapter four describes the data types that SDV supports. It also describes RDT in detail, as well as the specifics of the transformations used in SDV.

Chapter five covers the technical details of the sampling and modeling process in SDV. It also covers the different types of data set relationships that were discovered, and how SDV handles the modeling and sampling processes for each of them.

Chapter six describes a method for sampling a subset of data from the original data set for quicker modeling. Expanding on that, chapter seven describes a method for modeling multiple data sets in parallel. Finally, chapter eight covers the experiments we ran to test both SDV and this parallel modeling technique.
Chapter 2

SDV Redesign

One of the driving goals of this project was to make the Synthetic Data Vault more robust and extendable. To do this, it was first necessary to identify both the distinct components involved in synthetic data generation and how they interact with each other. We identified three distinct components contributing to data synthesis: a data transformer, a modeling technique, and a recursive algorithm used to model relationships between tables in the data set. Each of these modules have separate, unique roles that work together to synthesize data. Some of these modules also have practical uses outside of synthetic data generation. For this reason, we separated them and created an open-source Python library for each of the modules described above.

For the data transformation aspect, we created a library called Reversible Data Transforms (RDT). This library is responsible for cleaning data and preparing/transforming it to be fed into a modeling technique. It is also responsible for transforming the output back into its original format - hence the “reversible” in its title. Since the modeling technique originally used by SDV was a copula, we created a Copulas library. The role of this library is to take in numeric data tables and create a copula model from them from which rows can then be sampled. Finally, the portion of the code responsible for recursively traversing through the tables and applying the modeling technique resides in the core SDV library. This library has a recursive algorithm that traverses the data set and uses the other libraries to model the relationships
Figure 2-1: Diagram of how different libraries in SDV interact with each other between the tables. The exact algorithm is described in further detail in Chapter 5.

This chapter will describe each of the libraries at a high level, as well as their APIs. It will also cover the details of how these libraries interact with each other during the synthetic data generation process. Finally, this chapter describes the most important future improvements that can be made to SDV.

2.1 Copulas library

The first library that will be covered in this chapter is the Copulas library. A copula is a multivariate distribution of the marginal Cumulative Distribution Functions (CDFs) of random variables[5]. There are many different families of copulas, each of which takes a different mathematical approach to representing the data. The specifics of how they work and are implemented in this library are briefly explained in Chapter 3\(^1\). This section will focus on the API of the Copulas library.

\(^1\)Most of the development work around this library was done by Alicia (Yi) Sun and Manuel Alvarez Campo of Data to AI lab at MIT. For details about each modeling technique, API de-
As described above, copulas are used for modeling, and so the API was designed with this in mind. All instances of a copula class support the following methods:

- `fit(data (DataFrame), other (any type))`: Fits the distribution to the data and stores the fitted parameters.
- `get_pdf(x (array))`: Returns the probability density function (PDF) of the array.
- `get_cdf(x (array))`: Return the CDF of the array.
- `sample(num_rows (int))`: Samples the specified number of rows from the model. Returns as a Numpy array.

It is also important to understand that the columns of the input data each have a distribution of their own. For this reason, the Copulas library also provides a class structure for univariate distributions. Each instance of a univariate distribution has the following methods:

- `fit(data (Pandas Series))`: Fits the copula and stores fitted parameters for the distribution based on input data.
- `get_pdf(x (float))`: Returns the PDF of x.
- `get_cdf(x (float))`: Returns the CDF of x.
- `inverse_cdf(x (float))`: Takes a CDF value and returns the number that would produce that CDF given this distribution’s parameters.
- `sample(num_samples (float))`: samples the specified number of values from the distribution. Returns as Numpy array.

2.1.1 Design considerations

Below are some of the interesting considerations we took into account while designing the Copulas library.

1. The methods in the copula class only work on data that is completely numeric. If there are missing values or non-numeric values, then the methods described in this section will raise an error. This is part of the reason why RDT exists.

2. Even though the initial idea for the Copulas library was to use it for synthetic data generation, it can actually act as a stand alone library. If a user wishes to create a model of a particular data table, he or she only has to make sure that all of the values are numeric before passing it into the fit method. Once this is done, the sample method can be used to sample from that original data set.

3. The columns of the data being modeled by a copula each have a distribution. Since the type of this distribution can vary, we created a base univariate class. This way, a subclass can be made for each distribution type. Thus the Copulas library is also a library for representing univariate probability distributions of different types.

4. A base class was also made for the copulas themselves. This way, different families of copulas could be created and easily integrated into the library.

2.2 RDT library

Like the name suggests, the purpose of RDT is to provide a set of reversible transformations for data. It is also meant to provide a way to easily perform many transformations on an entire data set. The inspiration for this comes from the fact that the Copula library and many other modeling methods require the tables to be completely numeric. However, many times the data set will have columns with non-numeric values. RDT can convert all of the columns, allowing Copulas to be used on such a data set. To accomplish this, the library uses two main classes: the Transformer class and the HyperTransformer class. The types of Transformers and details of how these classes work are described in chapter 4.

The Transformer class is an abstract class that provides the methods that all Transformers are expected to have. Each individual Transformer takes in a column
of data as input. It is able to apply a transformation to the data as well as store
the necessary information to reverse that transformation. This provides an easy
way to clean or modify columns in a table based on the columns data type. Every
**Transformer** provides the following methods:

- **fit_transform**(col (*Pandas Series*), *col_meta* (*dict*)): Applies transformation
to column and stores parameters needed to reverse the transformation. The
*col_meta* argument is a dictionary containing the meta-information for the
column. This method returns a series of the transformed column.
- **reverse_transform**(col (*Pandas Series*), *col_meta* (*dict*)): Applies the re-
verse transformation to the input column using stored parameters. Returns a
series containing data in original format.

These methods can be used to transform and reverse-transform single columns at
a time, although it is more likely that users will want to transform a whole table or
even an entire data set at once. This larger-scale transformation is the purpose of the
**HyperTransformer** class. The **HyperTransformer** provides two types of methods:
those that can be applied to a table, and those that can be applied to a data set. The
table-level methods work by using instances of the Transformer class on the columns
of a table. The data-set-level methods work by applying the table-level methods on
every table in the data set. To initialize a **HyperTransformer**, a meta file must be
provided. The **HyperTransformer** API is as follows:

- **fit_transform**(tables (*dict*), *transformer_dict* (*dict*), *transformer_list*
*list*)): This method uses a Transformer to **fit_transform** every column in
every table in the data set and stores the fitted Transformers. It also re-
turns a dictionary mapping the table names to the transformed tables. The
*transformer_list* is a list of Transformer class names to use. The *transformer_dict*
is a dictionary mapping a key of the form (*table_name*, *col_name*) to the name
of a Transformer class.
- **reverse_transform**(tables (*dict*)): This method takes a dictionary mapping
table names to tables and uses the names to load up the stored Transformers
for each column and apply the reverse transformations. It returns a dictionary mapping table names to the reverse transformed tables’ DataFrames.

- **fit_transform_table** (table (DataFrame), table_meta (dict)): This method applies the fit_transform method of a Transformer to every column in a table.

- **reverse_transform_table** (table (DataFrame), table_meta (dict)): Loads the stores Transformers and runs the reverse_transform method on each column. Returns the reverse transformed table.

It is important to understand the format of the tables dictionary that both fit_transform and reverse_transform use. This dictionary maps the table name to a value of the form (table, table_meta), where the table_meta is the portion of the meta file corresponding to that table. The structure of the meta file is described in the next section. If the tables dictionary is not specified, then RDT will use the meta file to load the data and create this dictionary itself.

### 2.2.1 Design considerations

Below we list some of the interesting considerations we took into account when designing the RDT library.

1. One of the biggest challenges that came from designing the HyperTransformer was providing ways to specify which Transformers to use. In some use cases, a user might just want to apply one transformer to the whole data set; in other use cases, the user might want to specify a transformer for every column. To be flexible enough to handle these different scenarios, the HyperTransformer methods all have set parameters that allow users to tailor the transformations to their specific needs.

2. Going of the first consideration, the fit_transform method has a couple of ways to specify which transformations to use. A user can either provide a transformer_list or a transformer_dict. The list simply contains a list of Transformer subclasses. The HyperTransformer will apply those Transformers
to every column that matches the data type of the Transformer. If no list is provided, then the transformer_dict is used. This dictionary maps a key of the form (table_name, column_name) to the name of the Transformer to use. This way a Transformer is specified for each column in the data set. These two options help cover the different use cases described earlier.

2.3 SDV library

The final library spawned from the redesign of the synthetic data vault is the SDV library itself. Unlike the previous two libraries, SDV does not function completely alone. It requires access to the other two libraries. As described in Figure 2-1, SDV uses RDT to pre-process the tables and then recursively passes them into Copulas for modeling. This process continues until the root table is modeled. Then synthetic data can be sampled by recursively going through the table models and sampling from them. This implies that there are at least two separate modules necessary for SDV, one for modeling and one for sampling. Since retrieving and navigating through the data is also a large part of the data synthesis algorithm, we actually define two other modules as well: the DataLoader and DataNavigator. This section will describe the APIs of each module. The technical details can be found in chapter 5.

2.3.1 Meta file

Before we begin describing the different classes in SDV, we need to understand one of the main inputs. In order to sample and model the data, SDV requires meta information about the data like the relationships between tables and the types of the columns. This information is actually useful for many data science applications, so we created a JSON structure for how to present this information. The original structure was proposed in a thesis titled Towards Automatically Linking Data Elements[6]. We call this structure the meta file, and every data set needs one to run SDV. An example
of a properly formatted meta file is shown below.\textsuperscript{2}

```json
{
  "path": "",
  "tables": [
    {
      "path": "table.csv",
      "name": "table_name",
      "use": true,
      "headers": true,
      "primary_key": "id",
      "fields": [
        {
          "name": "id",
          "type": "id",
          "regex": "\.{10}"
        },
        {
          "name": "date_account_created",
          "type": "datetime",
          "format": "%Y-%m-%d"
        }
      ]
    }
  ]
}
```

This example shows that the meta file contains a list of the tables in the data set. Each table has a path that specifies where the CSV files for it are located. Each table also has a name and primary_key. All of the fields for each table are contained in

\textsuperscript{2}The full documentation for the meta file can be found at https://github.com/HDI-Project/MetaData.json[9]
a list as well. The fields each have a name and a type. Depending on the type, the field may have other parameters. For example, the datetime type has a format.
Table 2.1: Date Structures

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Attributes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_meta</td>
<td>Dictionary containing meta information for a column.</td>
<td>name: string defining the name of the column</td>
<td>{name: 'signup_date', type: 'datetime', format: '%Y-%m-%d'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>type: string defining the overall column type</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>other: Any other parameter the column might need</td>
<td></td>
</tr>
<tr>
<td>table_meta</td>
<td>Dictionary containing meta information for a column.</td>
<td>name: string defining name of the table</td>
<td>{name: 'users', path: 'users.csv', use: True, primary_key: 'id', fields: [col_meta1, col_meta2]}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>path: string defining path to CSV file</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>use: boolean defining whether table gets modeled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fields: list of col_meta objects</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>primary_key: string representing name of column that is the primary key</td>
<td></td>
</tr>
<tr>
<td>meta</td>
<td>Dictionary containing meta information for whole data set.</td>
<td>path: string defining path to folder containing CSVs</td>
<td>{path: 'data/', tables: [table_meta1, table_meta2]}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tables: list of table_meta objects</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>Class representing a table in the data set</td>
<td>data: the table's DataFrame</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>meta: the table's table_meta object</td>
<td></td>
</tr>
<tr>
<td>tables</td>
<td>Dictionary mapping table names to Table instances.</td>
<td>table_name: Table class instance corresponding to that name.</td>
<td>{'users': &lt;Table&gt;}</td>
</tr>
<tr>
<td>transformed_data</td>
<td>Dictionary mapping table names to the transformed</td>
<td>table_name: Numeric DataFrame for that table.</td>
<td>{'users': &lt;DataFrame&gt;}</td>
</tr>
<tr>
<td></td>
<td>DataFrames.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parent_map</td>
<td>Dictionary mapping table names to list of parent table</td>
<td>table_name: list parent table names</td>
<td>{'sessions': ['users']}</td>
</tr>
<tr>
<td></td>
<td>names.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>child_map</td>
<td>Dictionary mapping table names to list of child table</td>
<td>table_name: list child table names</td>
<td>{'users': ['sessions']}</td>
</tr>
<tr>
<td></td>
<td>names.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Attributes</td>
<td>Example</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>models</td>
<td>Dictionary mapping table names to Copula instances.</td>
<td>table_name: model class</td>
<td>{'users': &lt;GaussianCopula&gt;}</td>
</tr>
<tr>
<td>foreign_keys</td>
<td>Dictionary mapping tuple of child table name and parent table name, to tuple of primary key in parent table and foreign key in child table.</td>
<td>(child_table, parent_table): (parent_pk, child_fk)</td>
<td>{('sessions', 'users'): ('id', 'user_id')}</td>
</tr>
<tr>
<td>DataLoader</td>
<td>Class used to load data from different sources.</td>
<td>meta_file_name: string containing name of meta JSON file. meta: meta object for data set.</td>
<td></td>
</tr>
<tr>
<td>DataNavigator</td>
<td>Class used to navigate through data, store data and store relationships.</td>
<td>meta tables HyperTransformer transformed_data child_map parent_map foreign_keys</td>
<td></td>
</tr>
<tr>
<td>Modeler</td>
<td>Class used to model the data set</td>
<td>DataNavigator models model_type: name of Model class to use. model_params: list parameters for model</td>
<td>GaussianCopula' ['GaussianUnivariate']</td>
</tr>
<tr>
<td>Sampler</td>
<td>Class used to sample data from the models</td>
<td>DataNavigator Modeler sampled: dictionary mapping table name to tuple of primary key value for that generated rows and the rows themselves</td>
<td>{users': ('Bob', &lt;DataFrame&gt;)}</td>
</tr>
</tbody>
</table>
2.3.2 DataLoader

The very first step of the synthetic data generation process is loading the data. The original SDV only supported CSV files. One of the goals of the redesign of SDV was to be able to account for more use cases. For example, some users might have an SQL database instead of a folder of CSV files. This issue was the inspiration for the DataLoader class. As opposed to enforcing that users store their data as CSV files, they can instead use different DataLoader subclasses to load their data into pandas DataFrames for use in SDV.

At the core of the DataLoader module is a base DataLoader class. This is an abstract class that defines the methods that a DataLoader must support. Since the only real goal of a DataLoader is to load data from some source into Python, there is only one method that is required, the \texttt{loadData} method.

To ensure consistency, the base DataLoader class enforces that the \texttt{loadData} method returns a \texttt{DataNavigator}. It also requires that every DataLoader subclass take a meta file as an input. The way that the data is loaded and converted into a DataNavigator depends on the type or subclass of DataLoader.

To better understand how this process might work, we can look at a specific example. Since the original SDV focused on CSV files, the first DataLoader subclass that was developed was the CSVDataLoader. The steps for loading data are as follows:

1. CSVDataLoader is passed a meta file as input. When instantiated, it stores the file as an attribute called the \texttt{meta}.

2. To create the DataNavigator class, it uses the information in the \texttt{meta} to find the CSVs and load them as pandas DataFrames.

3. It then stores a dictionary mapping the table’s name to an instance of a Table class, which stores the meta information and DataFrame for that specific table.

4. This information is then used to instantiate an instance of a DataNavigator, which is then returned by the \texttt{loadData} method.

The pseudo code for this method is shown in Algorithm 1.
Algorithm 1 Loading data from CSVs

1: **procedure** `LOADDATA(metaFile)`
2:  
3:     for `csv` in `getCSVs(metaFile)` do
4:         `tableName` ← `getTableName(csv)`
5:         `dataframe` ← `loadDataframe(csv)`
6:         `table_meta` ← `metaFile(tableName)`
7:         `table` ← `Table(dataframe, table_meta)`
8:         `tables[tableName]` ← `table`
9:  
10:     `dn` ← `DataNavigator(tables, metaFile)`
11:  
12: return `dn`

---

**Figure 2-2:** Diagram showing process of creating `DataNavigator`

2.3.3 DataNavigator

As described in the previous subsection, the DataNavigator stores an attribute called `tables`, which is a dictionary mapping table names to the corresponding `Table` class instance. The DataNavigator also stores the meta file as an attribute called `meta`. This class is responsible for a few key features of SDV and also provides the standard for how to represent multiple tables in a single data structure. On top of that, the DataNavigator must also make it easy to traverse through the relationships of the table.

To explain this better, all of these data sets can be viewed as trees. A root node in this tree is a table that has no parents. It follows that a leaf node is a table that has no children. Here a child is a table that has a column that references the primary key of another table (the parent). The most important capability of the SDV library is that it can model these relationships. This is important because it means that the synthesized data is not only realistic to the table on which it was based, but realistic to the entire structure of the data set. The DataNavigator stores these relationships, making it easy to recurse through tables during the modeling phase.

31
The DataNavigator also provides a plethora of methods for accessing and modifying table information. The last key responsibility of the DataNavigator is applying the transformations required for modeling, using RDT’s HyperTransformer. The DataNavigator API is as follows:

- **get_children(table_name (string))**: Returns a list of the specified table’s children.
- **get_parents(table_name (string))**: Returns a list of the specified table’s parents.
- **get_data(table_name (string))**: Returns the table’s DataFrame.
- **get_meta_data(table_name (string))**: Returns the meta data for a table.
- **transform_data(transformers (list))**: Applies the specified transformers to the data using RDT.
- **get_relationships(tables (dict))**: Traverses the data and stores all of the important relationships so that the other methods can be computed quickly.

The majority of the work done by the DataNavigator is in the **get_relationships** method. This method is responsible for traversing through the information in the meta file, and storing all of the important relationships between tables. This includes information about parents and children as well as primary key/foreign key references. More specifically, there are three dictionaries created by this function. The **parent_map** is a dictionary that maps the table’s name to a set of the names of that table’s parents. The **child_map** is the same but for the names of the table’s children. Finally, there is the **foreign_keys** dictionary, which maps a tuple of the form (childTableName, parentTableName) to a tuple of the form (parentPrimaryKey, childForeignKey). The last dictionary is very useful during the modeling process for reasons that will be described in section 5.1.

The **get_relationships** method is executed whenever a DataNavigator class is instantiated, meaning it does not have to be explicitly called. The pseudo code for this function is shown in Algorithm 2.
Algorithm 2 Extracting relationships from meta file

1: procedure GET_RELATIONSHIPS()
2:   for table in tables do
3:     for field in table.meta do
4:       if field makes reference then
5:         parent ← referenced table
6:         parentPrimaryKey ← referenced field
7:         foreignKey ← field.name
8:         child_map[parent] ← table
9:         parent_map[table] ← parent
10:        foreign_keys[(table, parent)] ← (parentPrimaryKey, foreignKey)
11:   return (parent_map, child_map, foreign_keys)

2.3.4 Modeler

Once the DataNavigator is created, we have all of the necessary information to begin the modeling process. SDV’s modeling technique is a bit complicated because it doesn’t simply model individual tables. Instead, it attempts to model the relationships between the tables using techniques called Conditional Parameter Aggregation (CPA) and Recursive Conditional Parameter Aggregation (RCPA). These techniques create many models during the process and are described in detail in chapter 5. For this reason, the Modeler class needs to be able to store multiple models. The original SDV used a Gaussian Copula as the model type for all of its models. It is possible that there are other Copulas that might perform better in SDV. To take that a step further, future iterations of SDV might not want to use Copulas at all. This means that the Modeler class also has to be flexible in terms of which model types it uses.

These requirements guided the overall design of the Modeler class. The final architecture of the Modeler has the following methods and attributes.

- Attributes

  - extended_tables - dictionary mapping table name to extended table.
  - models - dictionary mapping table name to table model.
  - data_navigator
  - model_type - the type of model to use, eg. Gaussian Copula.
- `model_parameters` - a list of the parameters necessary for the specified model type.

- Methods
  - CPA
  - RCPA
  - `modelDatabase`

From the user’s point of view, the only method they would ever really use is `modelDatabase`. That’s because this method uses the `DataNavigator` that is passed to the Modeler to appropriately call CPA and RCPA behind the scenes. Therefore, to use the `Modeler`, the user simply needs to specify which attributes to use. The `modelDatabase` method returns a dictionary mapping the names of the tables to an instance of the model that was created for that table. Currently, only Copula models are supported, so the dictionary maps to an instance of a Copula class.

### 2.3.5 Sampler

Once the modeling is complete, the final step in the data synthesis process is to sample new data. The `Sampler` class takes in an instance of the `Modeler` class, and uses that information to generate synthetic data. The goal of the `Sampler` is to provide different methods for sampling based on the user’s needs. Currently, the following three methods are supported:

- `sample_rows(table (string), numRows (int))` - samples the specified number of rows from the specified table. Returns a DataFrame.

- `sample_table(table (string))` - samples the same number of rows as the original specified table from the specified table. Returns a DataFrame.

- `sample_all(numRows (int))` - samples the specified number of rows from every table in the data set. Returns a dictionary mapping the table names to the DataFrames that get sampled.
Thus, to use the Sampler, the user only needs to provide a Modeler instance and a DataNavigator instance to the Sampler in order for the methods above to be called to sample the data. All of the sampled data is stored in a dictionary in the Sampler. This dictionary maps the table name to all of the rows that have been sampled for that table. The reason for this is that the goal of SDV is to generate data that maintains the relationships between tables. If the sampled data is not stored, then there is no way to create relationships between sampled tables.

2.3.6 SDV class

SDV is composed of the four modules described in the previous subsections. In some cases, a user might want to simply provide a meta file and start sampling data. In this case, they might not want to concern themselves with the specifics of how each module works. For this reason, we also provided an SDV class. This class acts as a container for the four modules of SDV. A user can simply instantiate an instance of the SDV class by feeding in the necessary parameters, call the fit method and be ready to start sampling data. The parameters required to create an instance of the SDV class are:

- meta_file (required)
- data_loader_type (optional) - defaults to CSVDataLoader.
- model_type (optional) - defaults to Gaussian Copula.
- model_parameters (optional) - defaults to parameters Gaussian Copula.

An instance of the SDV class can also be saved and loaded. The attributes stored in the Modeler, Sampler, and DataNavigator can also be accessed through the SDV class. This allows user to do tasks like modifying the data or viewing and analyzing the models.
2.4 Interactions between libraries

For SDV to work, the libraries need to be able to seamlessly pass information between each other. As shown in Figure 2-1, the data gets passed from RDT to SDV in the beginning. It is then passed back and forth between SDV and Copulas for modeling. Finally, it is handed back to RDT to be reversed transformed at the end of sampling. This section will describe each of those interactions in detail.

2.4.1 Initial interaction between RDT and SDV

As described in section 2.3.3, SDV’s DataNavigator class creates an instance of RDT’s HyperTransformer class and stores it. Thus each time a DataNavigator is created, a HyperTransformer is also created. If a user is dealing with the DataNavigator class itself, then they must call the transform_data method to perform the initial transformation of all the data. They can specify the list of Transformer classes they wish to use when calling this method. If nothing is specified, then SDV’s default list of Transformers is used. If the user is working with an instance of the SDV class instead, then they do not need to specify anything. The class will implicitly transform the data using the default list. Since the DataNavigator stores the HyperTransformer, the Modeler will have access to it as well.

2.4.2 Interaction between Copulas and SDV

During the modeling phase, the Copulas library is used to form the models that get stored by the Modeler. The specific process for this is as follows:

1. When the user initializes the Modeler, he or she must provide the model type and the parameters necessary for that model type (e.g. Modeler (DataNavigator, model_type='GaussianCopula', model_params=[‘GaussianUnivariate’]))

2. Every table in the transformed_data attribute of the DataNavigator is then imputed. This means all of the missing values and infinite values are replaced
with the averages of the column. This step is necessary to ensure the model receives a completely numeric table.

3. The model then calls its fit method on the imputed DataFrame.

4. The model is then stored in the models attribute.

For the Copula models, the type of univariate distribution to use can be specified in the model parameters list. If the user is working with the SDV class, he or she can specify the model type and model parameters when instantiating the SDV instance.

2.4.3 Final interaction between Copulas, RDT and SDV

During the sampling phase, the models that were stored are used to sample the numeric data. SDV then passes this generated data into the reverse_transform_table method of the HyperTransformer that was stored. The output is a DataFrame containing data in the original format. Although storing all of the Transformer and Copula instances can be costly in terms of space, it makes it very easy for SDV generate and transform the data.

2.5 Design considerations

- Originally, the data was going to be loaded in the DataNavigator itself. When it became clear that there were many ways to store data sets, we decided to create a DataLoader class instead to take advantage of its best-in-class functionality.

- One of the challenges in designing the DataLoader class was that multiple data tables were being loaded at once. This raises the question of what format the DataLoader should return the tables in. To better illustrate this point, if the DataLoader class was just loading one table at a time, then it would be obvious to return the table as a pandas DataFrame. Since the DataLoader is actually responsible for loading up all of the tables in the data set, it was unclear whether they should be returned in a list, dictionary or some other format. To solve this
issue, we decided to return an instance of the `DataNavigator` class. The reason for this is that the `DataNavigator` provides methods to access and modify any table in the data set. It is also one of the inputs for the `Modeler` class. Given this information, it made sense to have the `DataNavigator` be SDV’s way of representing a data set.

- When creating an instance of a `Modeler`, the user can specify the model type and model parameters as a list. This was done because it is possible that model types might move away from Copulas in the future. If later someone wants to use a neural net instead, they can - they just need to make sure the model has a fit method and a way to be flattened.

### 2.6 Further improvements

One reason for re-factoring all of the SDV code and open-sourcing the libraries was so that new features could easily be added by contributors. Although SDV is currently functioning, there are a few important features that are still missing. This section lists and describes the most important missing features of SDV.

1. The first desired feature is the addition of more univariate distributions. Currently, SDV uses a normal distribution for every column. While this is good enough to sample and model data, it clearly is not the best technique. For example, many columns in the data sets we tested, did not fit a normal distribution. We are also in the process of adding a class that uses kernel density estimation (KDE) to the set of univariate distributions in Copulas.

2. Once more distributions are added, a feature should be added to the Copula class that deciphers which distribution to use for each column. This way, users will no longer have to specify the distribution to use when using Copulas.

3. Another important missing feature in the Copulas library is the ability to flatten a model into an array and to reconstruct a model from a flattened array. As
explained in chapter 5, during CPA, models are converted into an array of parameters. The row in the parent table is then extended with this array. Currently, SDV has code that can flatten a Gaussian Copula that uses normal distributions for the columns. If the modeling technique were to change at all, then SDV has no way of flattening those new models because the shape of the parameter array would be different. For this reason, it would be very convenient for every model to provide a function that flattens itself. Then, the Modeler could simply call that function during CPA. Similarly, the Sampler should be able to call a function belonging to the copula model, that takes an array of parameters as input, and constructs a new model from that.

4. Currently, there is a way to save an SDV instance as a pickle file. The problem is that, since this file is saving all of the copula models and parameters, it would be extremely large. There should be functions added to save an SDV class in a more space-efficient way. On top of that, a class method for loading should be added as well. Currently, there is no built-in function to load an SDV instance.

5. During sampling, some values are not numeric. These values end up being generated using regular expressions (a way to define the format of text). This is not ideal as it usually just generates random sequences of letters. One desired feature is to incorporate the Faker library into SDV. This library would use the meta file to find out the specific subtype of a text (eg. name) and generate a realistic example of that subset.
Chapter 3

Copulas

This chapter focuses on the technical details of the Copulas library. As described in section 2.1, a copula is a multivariate distribution of the CDFs of random variables. For this reason, the copula library is broken up into two main components: one for the univariate distributions that represent the random variable, and another for the multivariate method of representing the copula. Generally, the random variables will be represented by some univariate distribution. Then, depending on the family of copula, some transformation will be applied to the variables and a joint multivariate distribution will be made of those transformations.

3.1 Univariate distributions

There are many types of univariate probability distributions. For example, the copulas library supports both normal and uniform distributions. In order to account for the variety of distributions that might be encountered in our data, the copulas library has a base univariate distribution class. The purpose of the base class is to enforce that all subclasses have the same methods. In order to form a copula, the multivariate distribution needs to get certain functionality from each of its univariate distributions. This is why the base class enforces the following methods that were also described in section 2.1:

• fit
Each specific type of distribution inherits from this base class. The attributes will vary depending on the distribution, but the methods will always stay the same. This is because different distributions will keep track of different properties, but all of them have a probability density function (PDF) and cumulative distribution function (CDF). The fit method takes in a completely numeric array, and extracts the necessary attributes for the distribution. If we take the normal distribution for example, the only two attributes it keeps track of are the mean and standard deviation of the data[12]. Using these attributes, a PDF and CDF can be calculated.

3.2 Multivariate copulas

The second part of the library is responsible for representing the copulas themselves. We can think of a copula as a multivariate distribution that applies some changes to the random variables it takes as an input. With that in mind, we can see that the representation for a copula is not much different from that of the univariate distributions. In fact, the overall structure is almost identical.

The first thing to note is that some families of copulas are only bivariate, while others are multivariate. Furthermore, some of the multivariate copulas use bivariate copulas in their implementation. For this reason, two base classes are at the core of the copulas library, one for bivariate models and another for multivariate models. Both of them support similar functionality. The key difference is that the multivariate class provides the option to specify which bivariate class will be used in its implementation.

Similarly to the base univariate distribution class, the base copula classes enforce the following methods:
There are several differences between the methods here and those for a univariate distribution. Here we do not have an `inverseCdf` method. In general, a CDF function takes a value $x$ as input, and returns the probability that the random variable will take a value less than or equal to $x$. In multiple dimensions, the CDF function takes in multiple inputs and returns a single probability. Since multiple combinations of variable values could get the same result from the CDF of a multivariate distribution, an inverse function is impractical.

Each specific type of copula must inherit from one of these base classes. As an example, we can look at the Gaussian Copula, since it is the default copula used by SDV. A Gaussian Copula can be built using the following steps[4].

1. First start by calling the columns in the table $X_0, X_1, ..., X_n$, where $n$ is the number of columns.

2. For each column $X_i$, we will create an instance of a univariate distribution.

3. From these distributions, we can obtain the CDFs, $F_0, F_1, ..., F_n$.

4. For each row in the table, we can transform the values using the following transformation: $\phi^{-1}(F_i(x_i))$, where $\phi^{-1}$ is the inverse CDF of a standard normal distribution, and $F_i$ is the CDF of the distribution obtained for $X_i$.

5. This will yield our copula, $Y = \phi^{-1}(F_0(x_0)) + \phi^{-1}(F_1(x_1)) + ... + \phi^{-1}(F_n(x_n))$.

6. Finally, the covariance matrix $\Sigma$, can be computed for these transformed values.

All of these steps are completed in the `fit` method of the class. In order to avoid any type errors during the fitting process, the `fit` method requires that the input
table be completely numeric. Once this is done, data can be sampled from this new distribution. When the sample method is called, data is first generated using the multivariate Gaussian distribution. Then the data is reverted back into a form fitting the original distribution by applying the following transformation to each column, $F_i^{-1}(\phi(X_i))$. In this case, $F_i^{-1}$ is the inverse CDF of the original univariate distribution and $\phi$ is the CDF of a standard normal distribution. The implementation of the \texttt{fit} and \texttt{sample} methods differ quite a bit for each copula. This is why the base multivariate copula class exists. Any contributor could add a new type of copula to the library so long as these methods exist. The original implementation of SDV was not designed with this in mind, so adding new copulas would have been challenging.
Chapter 4

Reversible Data Transforms

This chapter describes the RDT library and the different types of transformations it supports. It also covers the technical details of how RDT actually transforms these different data types. We will start by explaining what a reversible data transform is, and then dive into existing Transformer classes. We will also discuss challenges in developing this library as well as further improvements that can be made.

4.1 RDT definition

Much like the name suggests, reversible data transforms are transformations that can take a column of some type and convert it to a completely numeric column. Furthermore, this transformation must have a way to reverse a numeric column into a column of the original type and format. The need for a reversible data transform is motivated by the fact that many data science algorithms and processes require strictly numeric tables. The example for SDV is sampling from a copula. That being said, sometimes it is desirable for the output of such a process to be converted back into the original format. For example, in SDV, it would be worthless to return data that was full of random floats. Those values need to be converted back into their column types in order for the data to be valuable.

A simple example of a reversible data transformation is a datetime transformation. Assume we have a column of type datetime. Each value can be converted to a unique
number by taking the number of seconds that date is away from a reference date. Similarly, the numeric column can be converted back by mapping the number to a datetime object of a specific format.

On the other hand, an irreversible data transform is one that cannot be converted back into the original format. For example, if we were to take strings in categories and map them to the number of letters in the word, we could not get the category back. More specifically, if we had the categories red, green and black, both green and black would map to five. Thus the reverse transformation would not always be correct.

Now we that we have a solid understanding of a reversible data transform, we can describe the architecture and design of the Transformer class.

4.1.1 Transformer design

As explained in section 2.2, RDT is composed of two main classes, the Transformer class and the HyperTransformer class. This subsection will explain how each class works in more detail. Since the Transformer class is used in the HyperTransformer class, we will start with the former.

Many different types of transformations can be needed to make a column strictly numeric. To account for this, the RDT library provides a base Transformer class. This is an abstract class that defines the methods that every Transformer must provide. Similar to the base uniform distribution class described in chapter 3, the attributes may differ for every Transformer subclass are but the methods remain basically the same. For example, a datetime transformer needs to store the format of the date as an attribute. This is unique to that type of transformer. The only attribute that is enforced for all transformers is the type. As the name suggests, this attribute defines the type of the column it is meant to transform. It is enforced because the HyperTransformer uses the type to infer which Transformer instances to use on specific columns. The methods enforced by the base Transformer are listed below.
Table 4.1: Transformers and their data types

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>NumberTransformer</th>
<th>CatTransformer</th>
<th>DatetimeTransformer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float, Int</td>
<td>Categorical</td>
<td>Datetime</td>
<td></td>
</tr>
<tr>
<td>Output Data Type</td>
<td>Float</td>
<td>Float in range (0,1)</td>
<td>Float</td>
</tr>
</tbody>
</table>

Figure 4-1: Diagram of different data types

- `fit_transform`
- `reverse_transform`

As an example, if a user wanted to create an instance of a `Transformer` and fit it to data, he or she could do the following:

```python
data = pd.read_csv('data.csv')
transformer = CatTransformer()
transformer.fit_transform(data)
```

Similarly, if they want to reverse a numeric column they could do the following:

```python
data = pd.read_csv('numeric_data.csv')
transformer.reverse_transform(data)
```

4.2 Existing transformers

There are many different data types supported by RDT. The exact set can be seen in Figure 4-1. Of these different types, RDT has `Transformer` classes for the follow-
ing: categorical, number and datetime. This section will describe these supported transforms as well as the specifics of the transformation process.

4.2.1 NumberTransformer

This section talks about the NumberTransformer and how it handles different numeric types. The first numeric type that needs to be handled by the NumberTransformer is an integer. Since integers are already in a format that can be modeled by a copula, not much needs to be done by RDT. The only transformation is that the integers are converted to floats of the same value. When rows are sampled from a copula model, each value in the row will be a float. Since the type we are concerned with is an integer, RDT will have to round the number during the reverse transformation process. So the overall flow for values in an integer column is to convert to a float using fit_transform, model in a copula, sample from the copula and round back to an integer using reverse_transform.

The other numeric type is a float. Floats are the easiest numeric type to handle since the transformations described in the previous section do not apply. The copula model can take in values in a float column directly as they are. When values are sampled, no reverse transformations are necessary either, since the desired output is a float. The only transformation necessary is if there are null values, and the process for this situation is described in section 4.4.

4.3 DatetimeTransformer

A column of type datetime is a bit more challenging to handle. This is because although any date can be converted to a float, in most tables, datetime values are represented as a formatted string. This means that in order to give the values in a datetime column to a copula model, they must first be converted to a float. Every value in the column is converted into a float by taking the time value and calculating its distance in seconds from the epoch. For most Unix systems, the epoch used is January 1st, 1970[1].
It is important to note that the distance is not an absolute value - i.e., that time values before the epoch are negative and time values after the epoch are positive. After this conversion is made, the values can be given to a copula model. The sampled values are then converted back into the original datetime format using the \texttt{reverse\_transform} method. In order to get the correct format, the \texttt{DatetimeTransformer} stores the format used by each column.

### 4.3.1 CatTransformer

The final data type handled by RDT is the categorical type. This is accomplished by the \texttt{CatTransformer} class. Categorical types are typically represented as strings in the original table. In order to get these columns into an appropriate format for a copula model, we have come up with the following procedure.

1. Collect all unique values of categories for the column and sort them from most frequent to least frequent.

2. Split the interval \([0, 1]\) into \(k\) sections where \(k\) is the number of unique categories. The size of the section should correspond to the probability of that value occurring. For example, if a category has a 40\% chance of occurring, then it might receive the range \([0.2, 0.6]\).

3. RDT can transform a category into a float by finding the range corresponding to that category, i.e. \([a, b]\)\(\epsilon[0, 1]\).

4. The upper and lower bounds of this range can be used to create a normal distribution with \(\mu = \frac{(b-a)}{2}\) and \(\sigma = \frac{(b-a)}{6}\). A float can then be sampled from this distribution and passed into the copula model.

After this procedure is applied to the column and the column is modeled, values can then be sampled. These sampled values will of course be floats that need to be converted back into the string category values. Since the \texttt{CatTransformer} stores the ranges corresponding to each category, we can simply find which range the sampled value belongs to, and replace it with the category.
4.4 NullTransformer

RDT provides the option to have missing values appropriately synthesized if the user so chooses. To do this, the pattern of missing values in numeric columns must also be modeled. This is achieved using a NullTransformer. If the user specifies that they want missing values to be generated, then the NullTransformer applies fit_transform to every numeric column on top of the transformations described in the previous sections.

This Transformer adds a new boolean column to the output of fit_transform. If the value was missing in the original data, then the new column will have a 1 in the corresponding row; if not, it will have a 0. With this information, the copula can model the likelihood of a value being missing. Then, after sampling the data, the reverse_transform method of the NullTransformer will determine if the numeric value should be outputted or if the value should be missing.

4.5 Contributing to RDT

RDT is an open source library and thus anyone can make contributions to it. To do this, however, a certain sequence of steps must be followed. The steps are described in this section.

4.5.1 Create transformer

The first thing a contributor has to do is create a new branch for their code. Then they can create a class that inherits from the Transformer class. As described before,
this class must have a `fit_transform` and `reverse_transform` method. The class must also have a type attribute that has the same value as one of the types described earlier in this chapter (e.g. categorical). The code might look something like what is shown below.

```
class NewTransformer(BaseTransformer):
    def __init__(self):
        self.type == 'categorical'

    def fit_transform(self, col, col_meta):
        ...

    def reverse_transform(self, col, col_meta):
        ...
```

Once this class is appropriately defined, the python file must be saved to the transformers folder. The directory structure is as follows:

```
\rdt
\transformers
    DTTTransformer.py
    NewTransformer.py
```

### 4.5.2 Write unit tests

Once the `Transformer` is created, unit tests must be added to the project before it can be accepted. The unit tests should at least cover the `fit_transform` method and `reverse_transform` method. Once the tests are written, the must be added to the proper folder. The directory location for tests is shown below.

```
\rdt
\tests
\rdt
\transformers
    test_NewTransformer.py
```

Once the unit tests are added and all tests are passing, a pull request can be made to merge the branch into the master branch for RDT.
4.6 Other possible Transformers

This section will cover ideas for desired Transformer classes. A good place to start is by examining the types that currently aren’t supported by RDT.

1. Some of the text subtypes have a format that can be converted into a number and reversed quite easily. For example, a phone number has a pre-defined number of values that can easily be converted to a number.

2. Similar to previous example, it could also be useful to have a credit card Transformer class.

3. The IP address subtype of the digital location type can also be modeled. Certain columns might represent IP addresses in a certain range, in which an IP Transformer class would prove useful.

4. The same thing goes for the MAC address subtype.

5. If the addresses of a column follow a particular format, they might also be able to be modeled. For example, addresses in some cities are all numeric, and have a certain range they fall in. For these areas, an address Transformer class could be implemented.

4.7 HyperTransformer and using RDT in a project

This section will describe the HyperTransformer class and then how that class can be used to use RDT for practical purposes.

4.7.1 HyperTransformer design

Unlike the Transformer, the HyperTransformer class is not an abstract class. Instead of gaining flexibility through inheritance, the HyperTransformer methods have many parameters that give users some leeway. More specifically, this class allows for multiple different inputs that define the Transformer classes that should be used.
Overall, users can specify the Transformer classes they wish to use in one of three ways.

1. A list of Transformers they want to use.
   - In this case, as the HyperTransformer loops through the columns, it will check to see if the type matches that of the Transformer. If so, it will be applied.

2. A dictionary mapping the column name to the Transformer type.

3. In the meta file, they can define the Transformer type for each field.

Once this information is provided, the user can use one of the four HyperTransformer methods described in section 2.2. The four methods divide into two categories: operations on tables and operations on data sets. We will explain the technical details for both categories. The algorithms for fit_transform_table and reverse_transform_table are all very similar, so to understand how they work, we can just look at the pseudo code for one. In this example, we assume that the user is passing in a list of Transformers that they want to use across the table.

Algorithm 3 Fitting and applying the hyper transformer to a table

1: procedure FIT_TRANSFORM_TABLE(TABLE, TRANSFORMER_LIST)
2:   for column in table do
3:       for transformer in transformer_list do
4:         if type(column) = type(transformer) then
5:           newColumn ← transformer.fit_transform(column)
6:           transformers[(tableName, columnName)] ← transformer
7:           outputTable.append(newColumn)
8:   return transformedTable

Since the entire purpose of RDT is to support transformations that are reversible, the HyperTransformer is also responsible for storing information to remember which columns in the data set correspond to which transformer. To do this it has an attribute called transformers, which is a dictionary mapping a key of the form (tableName, columnName) to the Transformer used on that column. By storing the
Transformer instances for each column, reverse transformations can easily be done by grabbing that Transformer instance and applying it to an input column.

The data set level transformation functions are also all very similar in implementation. To understand how they work, we can just look at the pseudo code for fit_transform in Algorithm 4.

Algorithm 4 Fitting and applying the hyper transformer to a data set

1: procedure fit_transform(metaFile)
2: tables ← getTablesFromMeta(metaFile)
3: for table in tables do
4:   transformedTable ← fit_transform_table(table)
5:   transformedTables[tableName] ← transformedTable
6: return transformedTables

4.7.2 Using the HyperTransformer in a project

Now that we understand how the HyperTransformer works, we can discuss how to incorporate it into a project. After the user has installed RDT, they should be able to import the HyperTransformer module using the following line of code.

```python
from rdt.hyper_transformer import HyperTransformer
```

Once this is done, the user can then decide how they want to use it. As discussed in the previous subsection, the HyperTransformer is useful for either transforming a table or an entire data set at a time. When initializing the HyperTransformer, the user should specify the meta_file location as a string argument. The HyperTransformer will then use that to locate the meta_file and load the tables. Notice that the HyperTransformer does not use the DataLoader to create a DataNavigator. Instead it loads the data on its own from CSV files. This means that no other methods of storing data are supported by RDT.

That being said, if users are able to load the data themselves, they can pass it directly into the methods. To do this however, the tables must be formatted in a certain way. More specifically, the fit_transform method accepts the data as a dictionary of the following format: 
```python
{table_name: (DataFrame, table_meta)}
```  
An example of using this argument is shown below.

54
ht = HyperTransformer(meta_file)
transformer_list = ['DTTransformer', 'NumberTransformer']
users_dataframe = <whatever code required to get data>
users_meta = <code to get table_meta for 'users'>
tables = {'users': (users_dataframe, users_meta)}
ht.fit_transform(tables=tables, transformer_list=transformer_list)

The only other thing the user need to understand is how to specify the Transformer classes to use. This can be done by passing a list of class names to the fit_transform method like in the example above. The user can also pass a dictionary of the form
{(table_name, column_name): Transformer}. An example of this is shown below.

transformer_dict = {('users', 'date'): DatetimeTransformer}
ht.fit_transform(transformer_dict=transformer_dict)

4.8 Design challenges

RDT is a very complex library and posed quite a few interesting design challenges. This section will discuss those in some detail.

- One of the biggest challenges was understanding how to account for the different use cases. Some users might want to be very specific about which Transformer classes to use for each column. Other users might just want to specify two Transformer classes and apply them to the whole data set. This is why the HyperTransformer methods support different inputs that specify this information.

- Another challenge was finding a way to decide when to apply a Transformer if only a list is provided. This is the reason that all Transformer classes must have a type attribute. This way, the HyperTransformer can use the type to decide whether or not to apply the Transformer in the list to a column.

- Another issue discovered is that many edge cases show up during the transformations. For example, all the Transformer classes should be able to handle both null and infinite values that work their way into the reverse_transform
methods. This is why all of the unit tests for Transformer class instances provide a test for null values.

- Some edge cases are type specific. This is one benefit of having a Transformer subclass for each type. For example, the DatetimeTransformer has a range of dates that it supports due to a limitation in Pandas[14]. This means that certain values are too large for the standard reverse transformation. In this case, an extra attribute was stored in the class called the default_val. This attribute is used whenever a date is out of bounds.

4.9 Future improvements

This section discusses the desired features of RDT that were not able to be implemented for this thesis.

- RDT’s HyperTransformer methods have the ability to take in a dictionary of the tables. The issue is that the dictionary’s format is different from how the SDV class stores its tables. This makes it difficult to pass tables between the two libraries. Currently, RDT simply loads the data itself using the meta file. This is not ideal as it means that both SDV and RDT are loading the data. It also means the RDT can only be used on CSV data.

- Some of the Transformer classes take a long time to run fit_transform. Unfortunately, we were unable to do any performance testing for these classes. One of the next steps for the library should be to improve the efficiency of every Transformer.

- Another desired improvement is unit tests for the HyperTransformer. Currently, only a couple of the possible use cases for the HyperTransformer have been tested. The class is supposed to be able to create a transformer_dict from the meta_file if neither the transformer_dict or transformer_list is provided to fit_transform. This has never actually been tested though.
• SDV provides a DataLoader class that could make RDT more useful. This would allow the library to load data from different sources as well as make it more compatible with SDV.

• The NullTransformer class has also never been properly tested.
Chapter 5

The Synthetic Data Vault

This chapter will cover the technical details of the modeling and sampling processes for SDV. This includes describing the core algorithms of the Modeler class, which are CPA and RCPA [11]. We then look at the different types of data set schemas and relationships that SDV can handle. These relationships take the form of structural relationships between tables and also relationships between fields. This chapter discusses how each of these relationships provides new challenges to SDV as well. The SDV library is the second version of a previous project titled The Synthetic Data Vault: Generative Modeling for Relational Databases[11]. The new iteration is able to cover the same relationships as the first one.

5.1 Modeling

As mentioned before, the key to SDV’s modeling technique is that it is capable of modeling relationships between tables in the data set. As described in section 2.3.4, the Modeler provides three main methods: CPA, RCPA, and modelDatabase. The majority of the hard work is done by CPA. CPA can be explained in the following four steps:

1. Iterate through each row in the input table

2. Iterate through the children of the input table. For every child, there should be
a column that references the primary key of the input table. This implies that there may be some rows in that child table that have the same value in this column, as the value of the primary key in the current row of the input table. The set of these rows is called conditional data. If there are m children, then we should have m sets of conditional data.

3. For each set of conditional data, we want to create a model using the type specified by the user. Each of these models will have parameters that we call conditional parameters.

4. We can then extend the current row of the input table with these parameters. The new columns that get added are called derived columns.

The idea behind this process is that for every value of the primary key of a table, there is a subset of rows in the children of this table that reference that specific value. By creating models of these subsets, and extending the original row with the parameters of these models, we are storing the influence that the parent table has over its children. This is better explained through an example. Suppose that we have a data set representing online transactions. The parent table stores information about the customers, and the child table stores information about the transactions. For every customer, there is potentially some subset of rows in the transactions table, representing the transactions that customer made. Those rows are defined as the conditional data for that customer. By modeling that conditional data, and extending the row of that customer with the model parameters, we will create a new table. This new extended customers table has information not only about the customers, but about how to create a transactions model for that customer. This is pivotal to generating accurate synthetic data, because the models that we sample from will now be taking into account the way that the parent tables influence their values.

**CPA** only aggregates the conditional parameters for a single table. In order for this modeling technique to really be useful, this process has to be done on every table in the data set in a particular order. This is where **RCPA** comes in. **RCPA** recursively runs **CPA** on all of the descendants of the node it is originally called on. This means
that CPA is initially run on the leaf nodes. By the time it gets back to the root node, all of the descendants have been extended with the necessary conditional parameters. This assures that all of the conditional parameters are modeled. The pseudo code for RCPA is shown in Algorithm 5.

**Algorithm 5 RCPA**

1: procedure RCPA(table)
2:    children ← get_children(table)
3:    for child in children do
4:        RCPA(child)
5:    extendedTable = CPA(table)
6:    extendedTables[tableName] ← preProcess(extendedTable)

Note that on line 6 of Algorithm 5, the extended table needs to be preprocessed. This is because it is possible for values in the extension to end up being null. To handle this, all extended tables are imputed before being stored by the Modeler. This imputation is done by replacing null or infinite values with the mean of the column. As described in section 2.1, the Copulas library requires a completely numeric table.
as input, thus these null values must be replaced.

RCPA allows for the modeling to be done recursively, but there still needs to be a way to know where to start the process and a functionality for storing the models as they are created. This is all done in the modelDatabase method. The goal of this method is to find the roots of the data set, run RCPA on them, and store the models in an attribute in the Modeler. At this stage, the data has already been transformed, meaning that the modeling will only happen on columns that have a Transformer class. This means that columns of type text, ID, and digital location are ignored during this process. The pseudo code for RCPA is shown in Algorithm 6.

**Algorithm 6** Pseudo code for modeling entire data set

1: procedure modelDatabase()
2:    for table in data set do
3:        if get_parents(table) = ∅ then
4:            RCPA(table)
5:            RCPA(child)
6:        for extendedTableName in extendedTables do
7:            extendedTable ← getDataFrame(extendedTableName)
8:            model = model_type().fit(extendedTable)
9:            models[extendedTableName] = model

---

### 5.2 Sampling

This section describes how the sampling process is implemented once the models are created and stored and SDV has everything it needs to generate synthetic data. In section 2.3.5, it is mentioned that the Sampler has three main methods: sample_rows, sample_table and sample_all. Although there are three different methods, most of the functionality is done in the sample_rows method, so we will start by explaining how that works.

Much like the modeling process, the sampling process needs to capitalize on the information about the relationships of the table to be effective. This means that although a model is formed for every table, those models should not always be used directly for sampling. We know that during CPA, all tables with children are extended
with the parameters of the models for each child. This means that a parent table has the necessary information to reconstruct a model for each child. When sampling, we therefore start by sampling the roots of the data set, use the parameters in the sampled rows of the root tables to construct models for the children, and then use these constructed models for sampling. This process can be repeated all the way down to the leaf nodes.

Following this logic, we can see that sampling rows comes down to two cases. Either the table we are sampling from has parents or it doesn’t. If it doesn’t, then we can sample using the model stored in the Modeler. If it does, we have to get a sampled row from the parent, and use the parameters in that row to create a model. This is essentially what the sample_rows function of the Sampler does. One other thing to keep in mind is that the models are only capable of generating the numeric columns of a table. This means that all of the non-numeric columns, including foreign and primary keys, must be synthesized using a different technique that is described in the next section. The pseudo code for sample_rows is shown in Algorithm 7.1

**Algorithm 7** Pseudo code for sampling rows from a table

```plaintext
procedure sample_rows(tableName, numRows)
    parents ← get_parents(tableName)
    primaryKey ← getPrimaryKey(tableName)
    if parents = ∅ then
        model ← Modeler.models[tableName]
    else
        randomParent ← getRandomParent(parents)
        parentRow ← sample_rows(randomParent, 1)
        model ← makeModelFromParameters(parentRow)
        foreignKeyVal ← parentRow[primaryKey]
    primaryKeyVal ← generateRandomVal(primaryKey)
    synthesizedRows ← model.sample(numRows)
    synthesizedRows.append(generateTextColumns(synthesizedRows))
    synthesizedRows.append([primaryKeyVal, foreignKeyVal])
    synthesizedRows ← reverse_transform(synthesizedRows)
    return synthesizedRows
```

As shown in the pseudo code, the process for sampling rows boils down to sampling

---

1Not all methods in the pseudo code are described in this paper. Refer to the SDV repository for full details.[10]
the numeric columns using the model, adding values for all the non-numeric columns, and then reverse-transforming the numeric columns using RDT. In line 11, we see that a random value for the primary key must be generated. This is because most primary keys are random assortments of characters that can’t be modeled. Therefore, to get the value, the format for that primary key must be specified in the meta file. This format is used to randomly generate a value. Line 10 shows that the foreign key value is always set to be the same as the primary key value of the parent row it was sampled from. This ensures that the sampled rows actually reference sampled rows from the parent table, and not just random values.

The other two sampling methods use `sample_rows` within them. The `sample_table` method simply calls `sample_rows` and passes in the number of rows of the original table in as a parameter. The pseudo code for the `sample_all` method is shown in Algorithm 8.

```
Algorithm 8 Pseudo code for sampling all tables in a data set
1: procedure sample_all(numRows)
2:   for tableName in tables do
3:     parents ← get_parents(tableName)
4:     if parents = ∅ then
5:       sample_rows(tableName, numRows)
6:       sampleChildRows(tableName, numRows)
```

The `sampleChildRows` method on line 6 of Algorithm 8 just calls `sample_rows` on all of the children of the input table. This guarantees that child tables are only sampled after their parents are sampled. Although it is not shown in any of the pseudo code, the Sampler actually stores all of the data that it generates. By storing sampled rows, we guarantee that new sampled data actually has relationships to old sampled data. It also means that parent rows do not always have to be randomly generated to sample children. Instead we can randomly pick from one of the already generated parent rows.
5.3 Sampling non-numeric data

As described in chapter 4, there are no Transformers for columns of type text, digital location, or ID. This means that they cannot be converted into a numeric table and modeled by a copula. Therefore, fields belonging to these types are handled primarily in two different ways during sampling. The method used depends on whether or not the field is a foreign or primary key (ID type).

5.3.1 Primary and foreign keys (IDs)

The meta file format supports a type titled ID. The subtype for ID is either primary or non-primary. The reason for this type is that IDs are handled differently from other types of text in SDV. Since IDs are usually a random unique set of characters, the meta file requires that any ID type provide a regular expression (regex) field. This is used to figure out the format of the ID.

During sampling, the primary key column is identified, and the corresponding regex retrieved from the meta file. This regex value is used to randomly generate an ID that fits the specified format. This is how unique primary key values are synthesized in SDV.

Foreign keys are handled in a similar manner. As we will see in section 5.5.1, the most common relationship between a foreign key and primary key of another table is one to one. In these scenarios, the value of the foreign key in the child table is usually the same as the value of the primary key in the parent table. For this reason, when sampling a child table row, the foreign key column is assigned the value of the primary key column in the parent row that is used to synthesize that child row. If there are multiple foreign key columns that reference the same primary key, then each of those columns will synthesize a new parent row and use the primary key value from there.
5.3.2 Text and Faker

All of the other types are supposed to be handled using the Faker library. This works by first extracting the specific subtype of the column. The Faker library can then takes this subtype as an input, and generates a random value for it. Examples of supported subtypes include address, name, email, url, phone number and social security number. These subtype values are provided in the meta file.

When SDV is sampling a new row, it will check all of the columns that were not modeled by the copula library. If these columns are labeled as a primary key or foreign key, then they will be handled as described in the previous section. Otherwise, they will be mapped to the corresponding type in the Faker library.

Algorithm 9 Synthesize Text Columns

1: procedure SYNTHESIZETEXTCOLUMNS(row, table_meta)
2: for column in row do
3: if table_meta[column] is primary key then
4: \(\text{regex} \leftarrow \text{table_meta[column]}.\text{regex}\)
5: \(\text{syntheticRow[column]} \leftarrow \text{generateID(regex)}\)
6: else
7: \(\text{type} \leftarrow \text{table_meta[column]}.\text{type}\)
8: \(\text{syntheticRow[column]} \leftarrow \text{faker(type)}\)
9: return syntheticRow

Currently, Faker is not integrated into SDV, so columns of these types must have a regex specified. This data can be generated in a similar fashion to how IDs are generated. This illustrates the need for Faker integration: data generated from regex does not look realistic. Thus these columns are not as useful as SDV intends for them to be.

5.4 Relationships between tables

Now that the data sampling and modeling techniques are defined, we can see the different type of data set schemas they must work on. The different types of relationships that can be found in a data set usually fall into two categories: relationships
between tables and relationships between fields. This section describes relationships found between tables.

5.4.1 Isolated table

The very simplest architecture to deal with is one in which there are no relationships between tables. We call a table with no children or parents an isolated table. In this case, SDV is able to simply create one Copula model for the table. No recursion is needed since there are no relationships that need to be modeled.

5.4.2 Linear

The most basic example of an actual relationship between two tables is a linear relationship. This is when the tables in the data set only have one parent/child. A structure like this is easy to deal with because no special precautions are needed when modeling. In this scenario, SDV first models the leaf node (table with no children), and then flattens the parameters of that model. These parameters are then used to extend the parent of the leaf node. This process is repeated until the root node (table with no parents) is modeled.

5.4.3 Multiple children

Another possible structure that arises is when tables have multiple children. SDV is able to handle modeling these relationships in the same fashion as linear relationships. The only added challenge is that a parent of multiple tables must be extended with
the parameters for all of its children. This means that it must have a way of keeping track of which columns correspond to the parameters of a specific child.

The Modeler in SDV currently handles this by storing a variable that holds the index locations of every child table’s parameters. As the parent table is extended with the parameters of its child’s conditional data model, the column indices that the parameters take up in the extended table are stored.

Overall, the process starts at the leaves in the data set. Each leaf node gets modeled and passes its flattened parameters up to its parent. The parent extends itself with the columns for the parameters of its children. After that, this extended table is modeled and the process is repeated until the root node is modeled. When sampling, a child table is able to retrieve its parameters from its parent since the indices of the relevant columns were stored. This implies that a parent table must be sampled before any of its children can be, since the parameters to create the child table’s model are stored in the extended parent table.

### 5.4.4 Multiple parents

Perhaps the most difficult table relationship to handle is one in which a table has multiple parents. This means that it has columns that reference the primary keys
of multiple different tables. This situation is difficult because the standard sampling
technique for SDV does not account for it. More specifically, when sampling from a
child table that has just one parent, the parameters stored in the extended parent
table are used to recreate the child table’s model. As described above, this means
that SDV samples a row of the parent and extracts the relevant columns to rebuild
the model. The problem is that if a table has multiple parents, it becomes unclear
which parameters to use.

We can illustrate this with an example. Imagine we have a data set pertaining
to the transactions of an online clothing store. This data set has three tables: one
for the users, another for the transactions, and the last for the items being sold. The
transactions table has a field for the item that was purchased as well as the user that
purchased it. This means that it references the primary key of the users table, as well
as the primary key of the items table. When SDV models this data set, it will start at
the transactions table. For each user, the rows in the transaction table corresponding
to that user will be modeled, and the parameters of that model will be appended to
the corresponding row of the users table. Similarly, for each item, the rows in the
transaction table corresponding to that item will be modeled and the parameters will
be appended. In the end, both the users table and items table will have different
parameters to recreate the transactions table’s model.

This raises the question of how these parameters should be used. There are a
couple of options. One is to randomly pick a parent each time the child table is
sampled from. This means that the sampled data will get a mix of rows that are
either generated completely from one parent’s perspective or completely from the
others. The other option is to do some sort of weighted averaging of the parameters
stored in each of the parents. This would ensure that the model being created for
the child table is as informed as possible, but on the other hand, it also requires that
each parent be sampled before the child table is sampled. Currently, SDV uses the
first method. One of the parent tables is randomly selected with equal probability.
Then a row from that parent is randomly selected, and the parameters in that row
are used to create a model and sample a row for the child table. This method is not
the best in terms of assuring the most informed model, but it is easier to handle and quicker since only one parent needs to be sampled beforehand.

5.4.5 Multiple children and parents

The final possible structure is one that puts everything together. Luckily, a data set that contains tables with multiple children and parents provides no new obstacles other than the ones described in the previous two subsections. Child tables will still pass up their parameters to each parent, and each parent will store the necessary information for each of its children.

5.5 Relationships between fields

5.5.1 One foreign key to one primary key

We have defined a child table to be a table with a field that references the primary key of the other table. All of the examples that have been given so far, have a one to one foreign key to primary key relationship. This means that one column in the child table references the primary key of another table. This is the situation that was used for each of the table relationships described above, so there aren’t any other technical details involved.

5.5.2 Many foreign keys to one primary key

Another possibility that hasn’t yet been described is one in which a child table has multiple columns that each reference the primary key of the same table. For example, if we have a data set representing basketball games, we might have a table for the teams and a table for the outcomes of games. In the table for the game outcomes, there will be a column for each team involved. This means that the table has two columns that reference the primary key of the teams table.

This scenario poses a new challenge to the way that SDV samples data because, when sampling a child table with a normal one-to-one relationship, the value of the
primary key from the parent row is used as the value of the foreign key in the sampled child row. In this case, that wouldn’t quite make sense since every row for the games table that gets sampled would say that a team played itself. In these cases, it usually makes sense for each foreign key column to have a different value of the parent table’s primary key. To handle this, only one of the foreign key columns is designated the role of foreign key. This column will get the same value as its parent’s primary key just like in the one to one example. The rest of the columns that reference the parent table’s primary key will be assigned a random possible value for that key. More specifically, for each of the other columns, a new parent row is generated and the value of the primary key in that new row is used.

A problem that still remains with the way SDV handles this type of relationship is that sometimes it is alright for multiple foreign keys to have the same value and sometimes it isn’t. In the basketball game example, it doesn’t make sense for the same team to play itself. If instead we have a data set representing chemical bonds, it might be alright for multiple atoms in the bonds table to have the same value. It is difficult to infer whether or not it is alright for a table to have duplicate foreign key values in the same row, so currently SDV does not worry about it. In future iterations, however, this is something that should be considered.
Chapter 6

Data subsets

This chapter describes the purpose and technical details of the data subsetting method we designed. The overall goal of “subsetting” is to extract subsets of data from a database such that each subset can be individually modeled by SDV. This can be useful if a user wishes to model a subset of the data based on a certain criteria - for example, modelling the data between a certain start and stop time. Being able to easily create subsets of data also makes it easy to divide the modeling of a database into multiple parallel jobs.

6.1 Overview of data subsetting challenge

There are two key challenges in developing approaches to create subsets. First, the users may want to create subsets based on different criteria or axis, and second, since we are dealing with multiple tables with relationships, the algorithm has to traverse the relationships and identify data elements from multiple tables.

There are many ways a user may want to subset their data for analytic purposes; thus key challenge to be solved by data subsetting is building a system robust enough to account for the different types of features which a user might want to subset. This is because different information is needed for each feature. For example, let’s say we are looking at a database of online customer transactions, for which there is one table of users and another of transactions. In this case, a user might wish to use SDV to
model of all transactions occurring in 2015, which would require sampling the data based on time, which requires knowing the date-time format. However, a different user might want an SDV model for customers from the United States only. In this case, the subsetting would be location-based.

Beyond the challenge of accounting for different features, another issue is that SDV is concerned with modeling multiple tables at a time. If only one table is being subsetted, then knowing the feature information would be enough. However, since SDV is designed to model an entire data set at once, there needs to be a way to subset from the rest of the data set based on the parent table. For example, if we have a data set with two tables, and the parent table has a time-based column, then a user may want to sample by year for that column. The issue is that the child table may not have a time based column, so there needs to be a way to infer which rows for the child table are relevant.

6.2 Design

At the core of the design for the data sampler is a base \texttt{DataSampler} class. The goal of this class is to be able to account for the different axes that users may want to use to create subsets. By having an abstract \texttt{DataSampler} class, a child class can be created to handle the unique properties that come with any specific axis.

6.2.1 Data sampler class

The base \texttt{DataSampler} class is an abstract class that defines the methods and attributes that any \texttt{DataSampler} should have. A \texttt{DataSampler} takes in as input, a meta file and a data loader type. This information is used to create a \texttt{DataNavigator} object that will allow the \texttt{DataSampler} to navigate the data easily.

The most important attribute of a \texttt{DataSampler} is the \texttt{index_map}. This is a dictionary that maps each unique value of the axis that the subset is being created upon, to another dictionary that maps table names to the indices pertaining to that subset. For example, if we are creating a subset of the data by year, the \texttt{index_map}
might look like the following:

```json
{2014:
  {
    users : [1,2,3,4],
    transactions: [2,5,6,12,13,15,18,25]
  },
2015:
  {
    users : [5,6,7,8],
    transactions: [1,3,4,7,8,9]
  }
}
```

In the example above, the users table is a parent of the transactions table. The indices 1, 2, 3, and 4 correspond to rows in the users table that from the year 2014. The indices for the transactions table are obtained by extracting the rows in the transactions table that reference the rows in the users table from the year 2014. The index_map only stores a list of relevant indices because this is more space-efficient than storing the cropped data tables instead. A user can take the information from the index_map and use it to easily extract the relevant rows from a table, so all of the necessary information is provided without creating a storage problem. Some of these data sets get very large, so storing all of the cropped tables could be problematic in some scenarios.

It is also important to note that each DataSampler subclass has its own method for populating the index_map, as different information is required to do this procedure for each type of axis.

### 6.2.2 Time based subset

One example of a subclass is the time-based subsetting. This DataSampler allows users to divide their database by any unit of time ranging from microseconds to weeks.
It inherits from the base `DataSampler` class, but provides extra methods. The process of sampling the data by time is broken into three main methods:

1. `getTimeRanges()`: Uses the unit of time and duration specified by the user to divide the data into different time ranges.

2. `sampleByTime()`: Populates the `index_map` for a specific table using the time ranges found by the `getTimeRanges` method.

3. `sampleChildren()`: Populates the `index_map` for the descendants of the table specified in `sampleByTime()`. For each unique value of time, the values of the primary key for the parent table are grabbed. These values are then used to grab the appropriate rows from the child table and store the indices of those rows in the `index_map`.

The process of subsetting data by time starts with the `getTimeRanges` method. In this step, the user specifies the table and column from which they want to get time values, the format of the datetime values in that column, and the duration they want to use for subsetting. There is also an option to specify the number of samples to use. If this parameter is specified, then that many time ranges will be randomly selected from the data. Otherwise, the time ranges created will span the entire length of the data.

To illustrate how this method works, we can use the example from before. We have a data set of online transactions consisting of a customers table and a transactions table. The customers table has a column for the date that the account was first created, and we want to sample by month over that column. We can do this by first calling `getTimeRanges` with the following parameters:

```python
table=’customers’
column=’date-account-created’
duration={weeks:4}
format=’%Y-%m-%d’
```
The duration argument is a dictionary that maps the unit of time to the desired duration. The maximum time unit that can be specified is a week, but months and years can be represented as four weeks or 52 weeks respectively. Calling the method with those parameters will return a list of four week long time ranges that make up the total range of the values in the column. For example, if the data has values from May 1st, 2012 to July 1st, 2012 then the method would return 


After the time ranges are retrieved, the `sampleByTime` method can be used to create the `index_map` for the `DataSampler`. This method essentially boils down to looping over every row in the table and placing the row’s index into the `index_map` under the time range it belongs to. The pseudo code for this method is shown in Algorithm 10.

**Algorithm 10 Sampling by time**

1: procedure `sampleByTime(table, column, timeRanges)`
2:   `numRows ← number of rows in table`
3:   `for i in range numRows do`
4:     `row ← table[i]`
5:     `for timeRange in timeRanges do`
6:       `date ← column[i]`
7:       `if date is in timeRange then`
8:         `index_map[timeRanges][table].append(i)`
9:     `sampleChildren()`

At the end of the algorithm, the `sampleChildren` method is called to recursively populate the rest of the `index_map`.

The code for the `sampleChildren` method is a little more confusing. There is a nested loop, since we must cover every child of the parent table and every unique time range in the `index_map`. For each of these, we extract the parent table’s relevant rows based on the indices stored in the `index_map`. Since each child table has a foreign key that references the parent table’s primary key, we extract all of the unique values of the parent’s primary key that appear in the sample. We then extract the indices of the rows from the child table that reference these primary key values. These indices get stored in the `index_map` under the corresponding unique time value. Once this is done...
Algorithm 11 Recursively sampling child tables

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td><code>procedure sampleChildren(tableName)</code></td>
</tr>
<tr>
<td>2:</td>
<td><code>childTables ← get_children(tableName)</code></td>
</tr>
<tr>
<td>3:</td>
<td><code>for child in children do</code></td>
</tr>
<tr>
<td>4:</td>
<td><code>for splitVal in index_map do</code></td>
</tr>
<tr>
<td>5:</td>
<td><code>parentIndices ← index_map.get(splitVal).get(tableName)</code></td>
</tr>
<tr>
<td>6:</td>
<td><code>parentTable ← table[parentIndices]</code></td>
</tr>
<tr>
<td>7:</td>
<td><code>uniquePrimaryKeys ← getUnique(parentTable[primaryKey])</code></td>
</tr>
<tr>
<td>8:</td>
<td><code>childIndices ← child[foreignKey] in uniquePrimaryKeys</code></td>
</tr>
<tr>
<td>9:</td>
<td><code>index_map.get(splitVal)[child] ← childIndices</code></td>
</tr>
<tr>
<td>10:</td>
<td><code>sampleChildren()</code></td>
</tr>
</tbody>
</table>

done, the `sampleChildren` method is called under the outer for the loop to continue recursing down the tree of data tables.

### 6.2.3 Further improvements

As mentioned before, each of the `DataSampler` subclasses require certain information to work. In the future, we would like to be able to infer this information as opposed to having the user provide it. We would also like to expand the different types of `DataSampler` classes beyond the feature based types we currently have. For example, a user might just want to randomly sample 10% of the database. Due to the current design of the `DataSampler`, this could be represented as a subclass with different methods for how to randomly pick rows.
Chapter 7

Parallel SDV Learning

As "big data" becomes ubiquitous, some data sets can get unbelievably large. One of the issues that SDV still faces is that it takes a long time to model a large data set with many relationships. One proposed solution to this problem was to break the original data set into smaller subsets, and model them in parallel. At the end, the parameters of the models could be averaged. This raises the question of whether or not that averaged model would perform the same as if the entire data set was modeled by SDV. To answer this question, we developed a Parallel SDV Learning module.

The goal of this module is to split the original data into smaller samples, and run SDV on all of these samples in parallel. It does this by leveraging the DataSampler class described in chapter 6, as well as new class called ParallelSDV.

7.1 The ParallelSDV class

The ParallelSDV class provides methods that use the index_map provided by a DataSampler to create multiple DataNavigator instances from the original data and run the SDV modeling process on them in parallel. This process happens in three main steps.

1. Provide a method that takes in one set of indices from the index_map, and creates a DataNavigator from it.
Figure 7-1: Diagram of data being sampled by year and learned by SDV in parallel.

2. Provide a method that takes a `DataNavigator` and models it using SDV’s `Modeler` class.

3. Provide a method that loops over all the sets of indices in the `index_map`, and calls the two methods described above to create a list of models.

Using the steps above, a user can create an `index_map` using the `DataSampler`, and then create an instance of `ParallelSDV` using that `index_map`. Then by simply calling the last method, they can model the whole data set in parallel.

The first method used is the `createDataNavigatorFromIndices` method. This method takes in a dictionary mapping the tables in the data set to the indices used in a particular sample. It then performs the following steps to create a `DataNavigator` instance that uses only those indices.

1. First, it uses the `DataLoader` class to load the original data in the form of a `DataNavigator`.

2. Then, it loops over every table in the data set.

3. Finally, for every table, it grabs the relevant indices provided by the `DataSampler`, and splices the data in the `DataNavigator` to match those indices.
This process returns a `DataNavigator` instance that only stores the data relevant to the subset. With this information, it is fairly straightforward to implement the method that models this `DataNavigator`. The only difficulty comes from the fact that the data must be transformed manually, as opposed to calling the built-in `transform_data` method. This is because the `HyperTransformer` belonging to the `DataNavigator` performs all of its hyper methods on the original data, not the data that was spliced using the `createDataNavigatorFromIndices` method.

To handle all of this, we created a `transformModelSample` method that transforms the data and then models the sample or subset. The pseudo code for this method is shown in Algorithm 12.

**Algorithm 12 Transforming and Modeling Sampled Data**

```
1: procedure transformModelSample(DataNavigator)
2:     for table in DataNavigator.tables do
3:         transformedTable ← DataNavigator.HyperTransformer.transform_table(table)
4:         transformed_data[tableName] ← transformedTable
5:     DataNavigator.transformed_data ← transformed_data
6:     modeler ← Modeler(DataNavigator)
7:     modeler.modelDatabase()
8:     return modeler.models
```

The final step involves running these methods in parallel on all of the sets of index values provided by the `index_map`. To accomplish this, we leverage a python library called Dask. Dask allows us to simply map the two methods described above to each other[3]. By passing in the full `index_map`, we can quickly compute the two methods on every set of indices and provide a list of models as output. These models can then be averaged to see if the parallel modeling technique can be used in place of modeling the whole data set at once. The specific steps for using Dask are as follows:

1. Create an instance of the dask.bag class by passing in a list containing all of the values in the `index_map`. That should have the form

   ```
   [{customers: [1, 2, 3], sessions: [1,2,8,9,12,13]},
    {customers: [4, 5, 6], sessions: [3, 5, 6, 7, 11]},
    ...
   ```
2. Map the dask.bag instance to the createDataNavigatorFromIndices method.
3. Map the dask.bag instance to the transformModelSample method.
4. Run dask.bag.compute()

Running the steps above will output a list of Modeler.models objects. Each of these stores the models for every table in that Modeler instance’s respective sample. The final step is to average these models. This is actually quite simple. The Modeler class provides a flattenModel method. Since the roots of the data set are the only models that actually get used for sampling, we can simply flatten the models of the roots, average them and reconstruct a model from the averaged parameters. The pseudo code for that is shown in Algorithm 13.

Algorithm 13 Averaging Models

1: procedure AVERAGEMODELS(MODELS)
2:     flattenedModels ← []
3:     for model in models do
4:         if model belongs to a root then
5:             flattenedModels.append(flattenModel(model))
6:     averagedParameters ← average(flattenedModels)
7:     averageModel ← makeModelFromParams(averagedParameters)
8:     return averageModel

The average model can then be compared to the model that is created when modeling the entire data set as whole. This comparison will tell us if modeling in parallel is a viable alternative to modeling the whole data set. The comparisons are covered in detail in the next chapter.
Chapter 8

Experiments

This project was largely about improving the code base of SDV. One of the issues SDV had in the past was that it was fairly inefficient when dealing with large data sets. In this chapter, we conduct a couple of experiments that put the refactored code’s efficiency to the test. On top of that, we would like to see if that parallel modeling method discussed in chapter 7, is a valid and reliable way to do modeling. We are able to test these features with the following two experiments:

1. Test SDV on 10 different data sets with different schemas and time it.

2. Use the `DataSampler` to sample from the Airbnb data set with different durations and with different sample sizes. Then, model those samples in parallel, time the modeling, average the parameters, and compare them to the parameters for the original Airbnb model.

8.1 Running SDV on different data sets

The purpose of this first experiment is to test both the speed and breadth of SDV. If we can generate data for 10 different data sets and sample from the models, then we show that SDV is able to handle a variety complex schemas. By timing these experiments, we also see which types of schemas are more time consuming that others. Before examining the results, we will first describe the different data sets.
Airbnb The Airbnb data set is one of the more straightforward examples we work with. It has only two tables to model, and the relationship between the two is linear. The data has a 'users' table representing the different Airbnb users, and a 'sessions' table representing web application sessions by different users. The biggest challenge from this data is that it has the most rows, meaning that a lot of conditional data sets are generated.

Telstra The Telstra data set is slightly more interesting because it contains a multiple-children relationship. The data consists of four tables: 'event_type', 'log_feature', 'resource_type', and 'severity_type'. Three of the tables are children of the 'severity_type' table. For SDV to model this, it is going to have to extend the 'severity_type' table with flattened models for each of the other three tables, meaning that the number of columns in that model can get quite large.

Rossman The Rossman data set is another example of just one linear relationship.
Biodegradability The Biodegradability data set contains two linear relationships, so this is the first data set with an ancestor greater than just a parent. It contains three tables: 'bond,' 'atom,' and 'molecule.' The 'bond' table actually has two columns that reference the 'atom' table, and the 'atom' table is just a normal child of the 'molecule' table.

Correctly sampling from this data set shows SDV’s ability to preserve relationships in synthetic data. It is also tricky because the two foreign keys in the 'bond' table should not always be the same, creating an additional sampling challenge for SDV.

Mutagenesis The structure for this data set is identical to the Biodegradability data set.

Hillary Clinton Emails This data set contains information from the Hillary Clinton email scandal. It is one of the more structurally interesting data sets because it contains multiple children, multiple parents and a lineage longer than two. The data set has four tables that get modeled: 'Emails,' 'Persons,' 'EmailReceivers,' and 'Aliases.' The 'Persons' table is a parent of the rest of them and the 'EmailReceivers' table is a child of both the 'Persons' table and the 'Emails' table.

The fact that the 'EmailReceivers' table has multiple parents means that SDV
NIPS 2015 Papers This data set contains a multiple parent relationship. It has three tables: 'Papers,' 'Authors,' and 'PaperAuthors'. The 'PaperAuthors' table is a child of the other two.

World Development Indicators This data set is challenging because it has six tables on top of every type of relationship. The 'Indicators' table is the child of the 'Country' table. The 'CountryNotes' table is the child of both the 'Country' table and the 'Series' table, and the same applies to the 'Footnotes' table. The 'SeriesNotes' table is another child of the 'Series' table.
Walmart  This is another example of a multiple children schema. The Walmart data contains three tables: 'features,' 'train,' and 'stores.' The 'training' and 'features' tables are both children of the 'stores' table as the training data and features both contain the store id in them.

Coupon Purchase Prediction  The final data set covered is Coupon Purchase Prediction, which contains six tables. Both the 'coupon_detail_train' table and the 'coupon_visit_train' table are children to 'user_list' and 'coupon_list_train'. The 'coupon_area_train' table is a child of both the 'prefecture_locations' table and the 'coupon_list_train' table. Finally, the 'user_list' table is a child of the 'prefecture_locations' table.

This data set is actually the most stressful on SDV because the model that gets created for 'prefecture_locations' has a lot of columns. This is because it has multiple children that it needs to extend its row with, and one of those children has multiple children of its own. Thus the rows contain the flattened parameters for four tables.
Table 8.1: Timing of Modeling and Sampling for Different Data Sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airbnb</td>
<td>3036.65</td>
</tr>
<tr>
<td>Telstra</td>
<td>210.84</td>
</tr>
<tr>
<td>Rossman</td>
<td>428.19</td>
</tr>
<tr>
<td>Biodegradability</td>
<td>2.47</td>
</tr>
<tr>
<td>Mutagenesis</td>
<td>2.41</td>
</tr>
<tr>
<td>Hillary Clinton Emails</td>
<td>17.11</td>
</tr>
<tr>
<td>NIPS 2015 Papers</td>
<td>0.89</td>
</tr>
<tr>
<td>World Development Indicators</td>
<td>692.16</td>
</tr>
<tr>
<td>Walmart</td>
<td>90.75</td>
</tr>
<tr>
<td>Coupon Purchase Prediction</td>
<td>3598.5</td>
</tr>
</tbody>
</table>

8.1.1 SDV experiment results

The results of running SDV on all of these data sets is shown in Table 8.1 There are a couple interesting takeaways from the results. The first is that, rather than the data set with the most rows producing the slowest result, it was Coupon Purchase Prediction - the data set with many relationships between tables which themselves have many columns. This probably happens because it requires extensive recursive modeling which can lead to extended tables with many columns.

Another observation worth noting is that if a table has very few transformations, then it will be modeled quickly. The NIPS 2015 Papers data set has only one column that gets modeled, which is why it is the fastest. The Hillary Clinton Emails data set is also very fast because there is no recursive modeling. Only one table has values that get modeled and that is the 'Emails' table.

8.2 ParallelSDV experiment

The other experiment we ran was to test if modeling samples of a data set in parallel and then averaging the parameters was a valid substitution for modeling the entire data set. Besides the effectiveness of the method, we also wanted to test how certain variables affected the outcome. More specifically, we wanted to test how the number
of samples used affected the success of the modeling as well as the duration used.

8.2.1 Experiment setup

To run this experiment, we took the following steps.

1. Install the DataSampler and ParallelSDV modules on an Amazon EC2 m5.24xlarge instance. This machine has 96 cores[2].

2. Use the sampleByTime method to sample the Airbnb data set at duration window sizes of 2 weeks, 4 weeks and 8 weeks as well as with 12 samples, 24 samples, 36 samples and 48 samples (12 sets of experiments in total).

3. For each of the combinations of number of samples and duration, run ParallelSDV, and time it.

4. Average the parameters of the models in the output of the previous step.

5. Compute the Root Relative Square Error (RRSE) between the parameters in the averaged models and the parameters of the model formed from running SDV on the entire Airbnb data set.

We ran 10 trials of the experiment described above. We collected the average time, standard deviation of time, average RRSE, and standard deviation of the RRSE for each trial. Note that we used RRSE to calculate the error instead of regular Mean Squared Error (MSE). This is because the parameters of the models vary in size quite a bit, with some being close to 0 and others being in the billions. We used RRSE to normalize these values.

The MSE would be calculated by taking the average square distance between parameters in the averaged (predicted) model and the actual model [7]. In comparison, RRSE is the squared error relative to the average of the parameters in the actual model [13]. To be more specific, the RRSE can be calculated with the following steps:
1. For every parameter $T_i$ in the actual model, compute its squared error with the corresponding parameter in the predicted model $P_i$. Divide this error by the squared error between $T_i$ and the average value of parameters in the actual model, $\bar{T}$.

2. Sum all of the values computed in the step 1.

3. Take the square root of the sum.

This process is also described by equation 8.1, where $\bar{T}$ is the average value of parameters in the actual model, $P_i$ is the $i^{th}$ parameter of the predicted model, and $T_i$ is the $i^{th}$ parameter of the actual model.

$$\sqrt{\frac{\sum (P_i - T_i)^2}{\sum (T - T_i)^2}}$$ (8.1)

### 8.2.2 ParallelSDV Results

<table>
<thead>
<tr>
<th>Window Size</th>
<th>Number of Samples</th>
<th>Average Time(s)</th>
<th>Average RRSE</th>
<th>Time SD</th>
<th>RRSE SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>164.860909</td>
<td>0.001611</td>
<td>5.961371</td>
<td>0.000192</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>196.028182</td>
<td>0.00143</td>
<td>4.8721</td>
<td>0.000036</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
<td>228.489091</td>
<td>0.001463</td>
<td>6.126736</td>
<td>0.000036</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>268.972727</td>
<td>0.001493</td>
<td>8.074161</td>
<td>0.000126</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>311.257273</td>
<td>0.001508</td>
<td>7.337269</td>
<td>0.000246</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>367.794545</td>
<td>0.001341</td>
<td>11.283282</td>
<td>0.000097</td>
</tr>
<tr>
<td>4</td>
<td>36</td>
<td>427.877273</td>
<td>0.001397</td>
<td>13.363546</td>
<td>0.00014</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>518.785455</td>
<td>0.001307</td>
<td>11.493122</td>
<td>0.000049</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>604.001818</td>
<td>0.001219</td>
<td>17.567324</td>
<td>0.000149</td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>705.308182</td>
<td>0.001116</td>
<td>24.159135</td>
<td>0.000118</td>
</tr>
</tbody>
</table>

The results are displayed as a chart representing the different combinations of window sizes and number of samples, as well as the average time, standard deviation of time, average RRSE and standard deviation of RRSE for each combination. There is a graph for each metric collected.
Average RRSE vs Number of Samples for 4 Week Window

Average Time vs Number of Samples for 8 Week Window

Average RRSE vs Number of Samples for 8 Week Window
The experiment results show exactly what you would expect to see and can be summarized as follows:

1. As the duration increases, the error decreases. This makes sense because the Modeler is using more data at once and thus more likely to be accurate to the full model.

2. As the number of samples increases, the error decreases. Again, this is likely because more data is being modeled at once.

3. Time increases with both duration and the number of samples; however, the duration has a larger effect.

4. Better accuracy can be achieved by using a small duration with many samples. For example, a duration of two weeks with 48 samples had lower error than a duration of four weeks with 12 samples. This is promising because the former took less time and had less error.

It is important to notice that there are no results for a duration of eight weeks and 36 or 48 samples. This is because those processes demanded too much memory from the machine and thus could not be completed. This means that in the future, SDV modeling will have to be made more space efficient.
Chapter 9

Conclusion

9.1 Key findings

In this second iteration of SDV, we made some interesting discoveries.

By running the modeling and sampling process on 10 data sets with different architectures, we were able to come up with a better understanding of which relationships caused SDV the most trouble. This is valuable because it points us to the algorithms that need to be improved. For example, we saw that the Customer Coupon Purchase Prediction data set took the longest due to the fact that a large number of columns are being created during the CPA process. This means the CPA method may need improvements.

Another interesting discovery is that the parallel modeling of subsets strategy shows promise. This is important for a couple of reasons. First, it shows that large data sets can be modeled much more quickly by subsetting and using ParallelSDV. Second, it shows that the subset doesn’t necessarily have to be particularly large, because sampling repeated values still seems to work.

Finally, we found that separating the different components involved in synthetic data generation into their own libraries did not hinder the success of the system. We now have a version of SDV that works and is extendable.
9.2 Contributions

In the second iteration of SDV we achieved the following.

- Created a library for modeling and sampling using copulas and defined how it works.

- Created a library for applying transformations and reverse transformations to data and defined how it works.

- Categorized the different types of relational data set structures and identified the challenges each one provided to SDV.

- Created a library for SDV that can successfully model and sample from different relational data set structures and defined how it works.

- Created a module for subsetting data based on different features.

- Created a module for running SDV on subsets in parallel, and then combining the models.

- Tested the legitimacy of parallel SDV modeling and found positive results.

These contributions make it possible for SDV to be easily improved and expanded on in the future, without the loss of any functionality. Thus, the SDV open source library achieves its primary goal.
Appendix A

Tables
<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Table Name</th>
<th>Column Name</th>
<th>Column Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>airbnb</td>
<td>sessions</td>
<td>user_id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>action</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>action_type</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>action_detail</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>device_type</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>secs_elapsed</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>date_account_created</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>timestamp_first_active</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>date_first_booking</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gender</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>age</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signup_method</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signup_flow</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>language</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>affiliate_channel</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>affiliate_provider</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first_affiliate_tracked</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signup_app</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first_device_type</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first_browser</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td>users</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>date_account_created</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>timestamp_first_active</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>date_first_booking</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gender</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>age</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signup_method</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signup_flow</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>language</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>affiliate_channel</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>affiliate_provider</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first_affiliate_tracked</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signup_app</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first_device_type</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first_browser</td>
<td>categorical</td>
</tr>
<tr>
<td>biodegradability</td>
<td>atom</td>
<td>atom_id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>molecule_id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>type</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td>bond</td>
<td>atom_id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>atom_id2</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>type</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td>molecule</td>
<td>molecule_id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>activity</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mweight</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>activitynorm</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>logpnorm</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mweightnorm</td>
<td>number</td>
</tr>
</tbody>
</table>
Table A.2: Date Sets – hillary_clinton_emails and nips_2015_papers

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Table Name</th>
<th>Column Name</th>
<th>Column Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hillary_clinton_emails</td>
<td>Aliases</td>
<td>Id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alias</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PersonId</td>
<td>id</td>
</tr>
<tr>
<td>Email Receivers</td>
<td>Id</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EmailId</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PersonId</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td>Emails</td>
<td>Id</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DocNumber</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MetadataSubject</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MetadataTo</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MetadataFrom</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SenderPersonId</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MetadataDateSent</td>
<td>datetime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MetadataDateReleased</td>
<td>datetime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MetadataPdfLink</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MetadataCaseNumber</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MetadataDocumentClass</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedSubject</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedTo</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedFrom</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedCc</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedCaseNumber</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedDocNumber</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedDateReleased</td>
<td>datetime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedReleaseInPartOrFull</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExtractedBodyText</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RawText</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>Persons</td>
<td>Id</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>nips_2015_papers</td>
<td>Authors</td>
<td>Id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>Paper Authors</td>
<td>Id</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PaperId</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AuthorId</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td>Papers</td>
<td>Id</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Title</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EventType</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PdfName</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Abstract</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PaperText</td>
<td>text</td>
<td></td>
</tr>
</tbody>
</table>
Table A.3: Date Sets – rossman, telstra, and walmart

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Table Name</th>
<th>Column Name</th>
<th>Column Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>rossman</td>
<td>store</td>
<td>Store</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StoreType</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assortment</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CompetitionDistance</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CompetitionOpenSinceMonth</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CompetitionOpenSinceYear</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promo2</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promo2SinceWeek</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promo2SinceYear</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PromoInterval</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td>train</td>
<td>Id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Store</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DayOfWeek</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Date</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Customers</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Open</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promo</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StateHoliday</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SchoolHoliday</td>
<td>categorical</td>
</tr>
<tr>
<td>telstra</td>
<td>event_type</td>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>event_type</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td>log_feature</td>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log_feature</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>volume</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>resource_type</td>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>resource_type</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td>severity_type</td>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>severity_type</td>
<td>categorical</td>
</tr>
<tr>
<td>walmart</td>
<td>features</td>
<td>Store</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Date</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Temperature</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fuel_Price</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MarkDown1</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MarkDown2</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MarkDown3</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MarkDown4</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MarkDown5</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPI</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unemployment</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IsHoliday</td>
<td>categorical</td>
</tr>
<tr>
<td>stores</td>
<td>Store</td>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>train</td>
<td>Store</td>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>Dept</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Date</td>
<td>datetime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Weekly_Sales</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IsHoliday</td>
<td>categorical</td>
<td></td>
</tr>
</tbody>
</table>

100
<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Table Name</th>
<th>Column Name</th>
<th>Column Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>world_development_indicators</td>
<td>Country</td>
<td>CountryCode</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ShortName</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TableName</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LongName</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alpha2Code</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CurrencyUnit</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpecialNotes</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Region</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IncomeGroup</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wb2Code</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NationalAccountsBaseYear</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NationalAccountsReferenceYear</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SnaPriceValuation</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LendingCategory</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OtherGroups</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SystemOfNationalAccounts</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AlternativeConversionFactor</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PppSurveyYear</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BalanceOfPaymentsManualInUse</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ExternalDebtReportingStatus</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SystemOfTrade</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GovernmentAccountingConcept</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ImfDataDisseminationStandard</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LatestPopulationCensus</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LatestHouseholdSurvey</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SourceOfMostRecentIncomeAndExpenditureData</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VitalRegistrationComplete</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LatestAgriculturalCensus</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LatestIndustrialData</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LatestTradeData</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LatestWaterWithdrawalData</td>
<td>number</td>
</tr>
<tr>
<td>CountryNotes</td>
<td>Countrycode</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seriescode</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>Footnotes</td>
<td>Countrycode</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seriescode</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Year</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>Indicators</td>
<td>CountryName</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CountryCode</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IndicatorName</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IndicatorCode</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Year</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>
### Table A.5: Date Set – world_development_indicators (continued)

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Table Name</th>
<th>Column Name</th>
<th>Column Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td>SeriesCode</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Topic</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IndicatorName</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ShortDefinition</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LongDefinition</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UnitOfMeasure</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Periodicity</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BasePeriod</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OtherNotes</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AggregationMethod</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LimitationsAndExceptions</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NotesFromOriginalSource</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GeneralComments</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Source</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>StatisticalConceptAndMethodology</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DevelopmentRelevance</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RelatedSourceLinks</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OtherWebLinks</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RelatedIndicators</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LicenseType</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td>SeriesNotes</td>
<td>SeriesCode</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Year</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>text</td>
<td></td>
</tr>
</tbody>
</table>
### Table A.6: Date Set – coupon_purchase_prediction

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Table Name</th>
<th>Column Name</th>
<th>Column Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>coupon_purchase_prediction</td>
<td>coupon_area_train</td>
<td>SMALL_AREA_NAME</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PREF_NAME</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COUPON_ID_hash</td>
<td>id</td>
</tr>
<tr>
<td>coupon_detail_train</td>
<td>ITEM_COUNT</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I_DATE</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMALL_AREA_NAME</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PURCHASEID_hash</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USER_ID_hash</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COUPON_ID_hash</td>
<td>id</td>
</tr>
<tr>
<td>coupon_list_train</td>
<td>CAPSULE_TEXT</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GENRE_NAME</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRICE_RATE</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CATALOG_PRICE</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DISCOUNT_PRICE</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DISPFROM</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DISPEND</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DISPPERIOD</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VALIDFROM</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VALIDEND</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VALIDPERIOD</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_MON</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_TUE</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_WED</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_THU</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_FRI</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_SAT</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_SUN</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_HOLIDAY</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USABLE_DATE_BEFORE_</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>large_area_name</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ken_name</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>small_area_name</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COUPON_ID_hash</td>
<td>id</td>
</tr>
<tr>
<td>coupon_visit_train</td>
<td>PURCHASE_FLG</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I_DATE</td>
<td>datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAGE_SERIAL</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REFERER_hash</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VIEW_COUPON_ID_hash</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USER_ID_hash</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SESSION_ID_hash</td>
<td>categorical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PURCHASEID_hash</td>
<td>categorical</td>
</tr>
<tr>
<td>prefecture_locations</td>
<td>PREF_NAME</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PREFECTUAL_OFFICE</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LATITUDE</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGITUDE</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>user_list</td>
<td>REG_DATE</td>
<td>datetime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SEX_ID</td>
<td>categorical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AGE</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WITHDRAW_DATE</td>
<td>datetime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PREF_NAME</td>
<td>id</td>
<td></td>
</tr>
<tr>
<td></td>
<td>USER_ID_hash</td>
<td>id</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


