The Synthetic data vault

Neha Patki
CSAIL, MIT
Cambridge, MA - 02139
npatki@mit.edu

Roy Wedge
LIDS, MIT
Cambridge, MA - 02139
rwedge@mit.edu

Kalyan Veeramachaneni
LIDS, MIT
Cambridge, MA- 02139
kalyanv@mit.edu

***Abstract*—The goal of this paper is to build a system that automatically creates synthetic data to enable data science endeavors. To achieve this, we present the Synthetic Data Vault (SDV), a system that builds generative models of relational databases. We are able to sample from the model and create synthetic data, hence the name SDV. When implementing the SDV, we also developed an algorithm that computes statistics at the intersection of related database tables. We then used a state-of-the-art multivariate modeling approach to model this data. The SDV iterates through all possible relations, ultimately creating a model for the entire database. Once this model is computed, the same relational information allows the SDV to synthesize data by sampling from any part of the database.**

After building the SDV, we used it to generate synthetic data for five different publicly available datasets. We then published these datasets, and asked data scientists to develop predictive models for them as part of a crowdsourced experiment. By analyzing the outcomes, we show that synthetic data can successfully replace original data for data science. Our analysis indicates that there is no significant difference in the work produced by data scientists who used synthetic data as opposed to real data. We conclude that the SDV is a viable solution for synthetic data generation.

I. Introduction

An end-to-end data science endeavor requires human intuition as well as the ability to understand data and pose hypotheses and/or variables. To expand the pool of possible ideas, enterprises hire freelance data scientists as consultants, and in some cases even crowdsource through KAGGLE, a website that hosts data science competitions. In conversations with numerous stakeholders, we found that the inability to share data due to privacy concerns often prevents enterprises from obtaining outside help. Even within an enterprise, development and testing can be impeded by factors that limit access to data.

In this paper, we posit that enterprises can sidestep these concerns and expand their pool of possible participants by generating synthetic data. This synthetic data must meet two requirements: First, it must somewhat resemble the original data statistically, to ensure realism and keep problems engaging for data scientists. Second, it must also formally and structurally resemble the original data, so that any software written on top of it can be reused.

In order to meet these requirements, the data must be statistically modeled in its original form, so that we can sample from and recreate it. In our case and in most cases, that form is the database itself. Thus, modeling must occur before any transformations and aggregations are applied. For example, one can model covariates as a multivariate distribution, and then sample and create synthetic data in the covariates space. Similarly, for online transactional data, one can extract the sequence of actions taken by customers, learn a hidden Markov model, and subsequently sample sequences from it. These strategies, while important in their own right, involve modeling data only after it has been aggregated and/or transformed with a purpose in mind. Whatever synthetic data is produced under these conditions can only aid that same purpose—for example, users working with synthetic action sequences can only explore and build solutions for problems that are defined over them. To enable a multitude of data science endeavors, we challenged ourselves to model the database directly, and to do so with no specific dataset in mind.

In the past, several researchers have focused on statistically modeling data from a relational database for the purposes of feature engineering [1] or insight generation [2]. [2] laid a Bayesian network over a relational data model and learned the parameters for the conditional data slices at the intersection of those relations. By extending these two concepts, we develop a multi-level model with a different aim—not insights, but rich synthetic data. To do this, we not only have to create a statistically validated model; we also have to pay attention to the nuances in the data, and imitate them. For instance, we treated missing values in a number of different ways, and built in the ability to handle categorical values and datetime values.

In this paper, we make the following contributions:

1) **Recursive modeling technique**: We present a method for *recursively* modeling tables in the database, allowing us to synthesize artificial data for any relational dataset. We call our approach "recursive conditional parameter aggregation". We demonstrate the applicability of our approach using 5 publicly available relational datasets.
2) **Creation of synthetic data**: We demonstrate that when a dataset and its schema are presented to our system (which we call Synthetic Data Vault), users can generate as much data as they would like post-modeling, all in the same format and structure as the original data.
3) **Enable privacy protection**: To increase privacy protection, users can simply perturb the model parameters and create many different noisy versions of the data.
4) **Demonstration of its utility**: To test whether synthetic data (and its noisy versions) can be used to create data science solutions, we hired 39 freelance data scientists to develop features for predictive models using only synthetic data. Below we present a summary of our results.

Summary of results: We modeled 5 different relational datasets, and created 3 versions of synthetic data, both with and without noise, for each of these datasets. We hired 39

data scientists and divided them into groups to solve predictive problems defined over the 5 datasets. We presented different groups with different versions of the data, always giving one group the original data. For each dataset, we compared the predictive accuracies of features generated from the original data to the accuracies of those generated by users who were given the synthetic data. Regardless of which group it came from, predictive accuracy for a feature is generated by executing that feature on the original data.

We found no significant statistical difference in the data scientists' work: For 11 out of 15 comparisons (>70%), data scientists using synthetic data performed the same or better than those using the original dataset.

The level of engagement with synthetic data was high: Even without being told that they were working with synthetic, noisy data, data scientists engaged with it just as well as they did with the original data. Data scientists working on synthetic data wrote a total of 4313 features on the 5 datasets.

The rest of the paper is organized as follows. In Section **??** we justify our focus on enabling predictive modeling through synthetic data generation. Section III provides an overview of our system and its different components. Section IV presents our modeling methodology in detail. Section V presents the algorithm to synthesize data from our model. Section VI presents our experimental setup, results, and conclusions.

II. Predictive modeling

Having figured out how to generate synthetic data for an arbitrary database, we asked whether this could enable predictive modeling. We chose predictive modeling, rather than deriving insights based on visualization and descriptive analytics, for several reasons: its impact, the potential to widen the foundational bottleneck of feature engineering, and a number of observations from our own past experiences:

- When given a database, data scientists often only look at the first few rows of each table before jumping in to define and write software for features.
- If the data relates to a data scientist's day-to-day activities, they will first attempt to understand the fields, and then quickly begin to write software.
- A data scientist is more likely to explore data in depth after their first pass at predictive modeling, especially if that pass does not give them good predictive accuracy.

Keeping these observations in mind, along with the fact that we can perturb the learned model to generate noisy synthetic data, we then asked: how much noise can we tolerate? If we can achieve a similar end result even when a lot of noise is added, this noise could ensure better protection of the data. To demonstrate the efficacy of this process, we assembled 5 publicly available datasets, created multiple synthesized sets, and employed a crowd of data scientists to build predictive models from these synthesized versions. We then examined whether there was a statistically significant difference in predictive model performance among the sets.

III. Overview

Our system, which we call Synthetic Data Vault, is broken down into four steps, as illustrated in Figure 1.

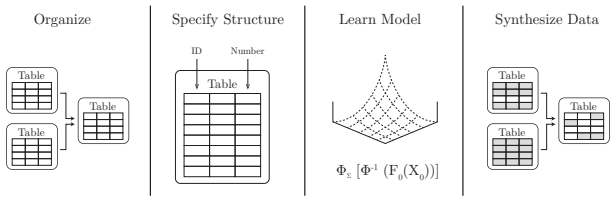

Fig. 1. The SDV workflow: The user collects and formats the data, specifies the structure and data types, runs the modeling system, and then uses the learned model to synthesize new data.

Organize: Before supplying the data to the SDV, the user must format the database's data into separate files, one for each table.

Specify Structure: The user must specify basic information about the structure of each table, and provide it as *metadata* for the database. This specification is similar to a schema in an SQL database.

Columns with ID information are special, because they contain relationship information between multiple tables. If the ID column of a table references an ID column of another table, the user must specify that table.

Learn Model: The user then invokes the SDV's script to learn the generative model. The SDV iterates through tables sequentially, using a modeling algorithm designed to account for relationships between the tables.

For each table, the SDV discovers a structure of dependence. If other tables reference the current one, dependence exists, and the SDV computes aggregate statistics for the other tables. The aggregate statistics are then added to the original table, forming an extended table. This extended table is then modeled. It captures the generating information for the original table columns, as well as all the dependencies between tables.

The SDV uses some simple optimizations to improve efficiency. It saves all the extended tables and model information to external files, so that subsequent invocations for the same database do not perform the same computations unnecessarily.

Synthesize Data: After instantiating the SDV for a database, the user is exposed to a simple API with three main functions:will

1) `database.get_table`This returns a model for a particular table in the database. Once the table has been found, the user can use it to perform the other two functions.
2) `table.synth_row`:The `synth_row` function both synthesizes rows and infers missing data.
3) `table.synth_children`:The `synth_children` function synthesizes complete tables that reference the current table. By applying this function iteratively on the newly-synthesized tables, the user can synthesize an entire database.

The results of both `synth_row` and `synth_children` match the original data exactly. The SDV takes steps to delete extended data, round values, and generate random text for textual columns. This results in rows and tables that contain fully synthesized data, which can be used in place of the original.

IV. Generative Modeling Method

This section covers the technical details of the SDV's modeling phase in the overall workflow presented in Figure 1. The goal of the generative modeling phase is to build a complete model for the entire relational database, given only meta files and tables. Ultimately, the SDV's database modeling method builds generative models for individual tables. However, it performs extra computations to account for the the relationships between them, using a method called Conditional Parameter Aggregation (CPA). A high-level overview is provided by Figure 2.

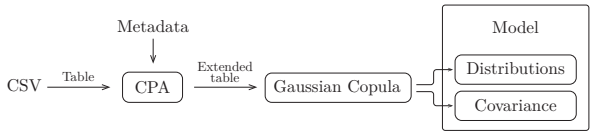

Fig. 2. An overview of the generative modeling process. Conditional Parameter Aggregation accounts for foreign key relations across multiple tables. The Gaussian Copula process calculates the overall table model.

This section is broken into five sections: Section IV-A reviews the multivariate generative modeling method we use for a table. This corresponds to the Gaussian Copula and model steps in Figure 2, and provides a foundation for our work. Section IV-B describes extending the generative model to encompass multiple tables. This is called condition parameter aggregation CPA. The next two sections provide additional adjustments necessary to make the algorithms more generalizable. Finally, Section IV-D provides the overall logic for applying our technique. This means recursively applying CPA for all tables, in order to model the entire database.

A. Standalone Table Model

We define a standalone table as a set of rows and columns that we wish to model independently of any other data. The generative model for a standalone table encompasses all columns that represent numerical data,[1] and it consists of:

- Distributions: The probability distributions of the values in each column
- Covariances: How the value of a column affects the value of another column in the same row

The distribution describes the values in a column, and the covariance describes their dependence. Together, they form a descriptive model of the entire table.

1) Distribution: A generative model relies on knowing the distribution shapes of each of its columns. The shape of the distribution is described by the *cdf* function, F, but may be expensive to calculate. A simplistic estimate is to assume the original distribution is Gaussian, so that each F is completely defined by a μ and σ^2 value. However, this is not always the case. Instead, we turn to some other common distributions shapes that are parametrized by different values:

- Truncated Gaussian Distribution: Parametrized by the mean μ, variance σ^2, min, and max values
- Uniform Distribution: Parametrized by the min and max values
- Beta Distribution: Parametrized by α and β
- Exponential Distribution: Parametrized by the decay λ

If the column's data is not Gaussian, it may be better to use a different distribution. In order to test for this fit, we use the Kolmogorov-Smirnov test [3], which returns a p-value representing the likelihood that the data matches a particular distribution. The distribution with the higher p-value is the distribution we use to determine the *cdf* function. Currently, we decide between truncated Gaussian and uniform distributions, but we provide support to add other distributions.

Note that parameters represent different statistics for each distribution. For this reason, the SDV also keeps track of the type of distribution that was used to model each column. This lets the SDV know how to interpret the parameters at a later stage. For example, if the distribution is uniform, then the parameters represent the min and max, but if it's Beta, then they represent α and β.

2) Covariance: In addition to the distributions, a generative model must also calculate the covariances between the columns. However, the shape of the distributions might unnecessarily influence the covariance estimates [4].

For this reason, we turn to the multivariate version of the Gaussian Copula. The Gaussian Copula removes any bias that the distribution shape may induce, by converting all column distributions to standard normal before finding the covariances. Steps to model a Gaussian Copula are:

1) We are given the columns of the table $0, 1, \ldots, n$, and their respective cumulative distribution functions $F_0, \ldots, F_n$.
2) Go through the table row-by-row. Consider each row as a vector $X = (x_0, x_1, \ldots, x_n)$.
3) Convert the row using the Gaussian Copula: $Y = \left[\Phi^{-1}\left(F_0\left(x_0\right)\right), \Phi^{-1}\left(F_1\left(x_1\right)\right), \ldots, \Phi^{-1}\left(F_n\left(x_n\right)\right)\right]$ where $\Phi^{-1}(F_i(x_i))$ is the inverse *cdf* of the Gaussian distribution applied to the *cdf* of the original distribution.
4) After all the rows are converted, compute the covariance matrix, Σ of the transformed values in the table.

Together, the parameters for each column distribution, and the covariance matrix Σ becomes the generative model for that table. This model contains all the information from the original table in a compact way, and can be used to synthesize new data for this table.

B. Relational Table Model

In a relational database, a table may not be standalone if there are other tables in the database that refer to it. Thus, to fully account for the additional influence a table may have on others, its generative model must encompass information from its child tables. To do this, we developed a method called Conditional Parameter Aggregation (CPA) that specifies how its children's information must be incorporated into the table. Figure 3 shows the relevant stage of the pipeline.

This section explains the CPA method. CPA is only necessary when the table being processed is not a leaf table.

[1] Later, we discuss how to convert other types of data, such as datetime or categorical, into numercial data

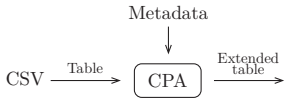

Fig. 3. Aggregating data from multiple child tables creates an extended table that accounts for the original relations.

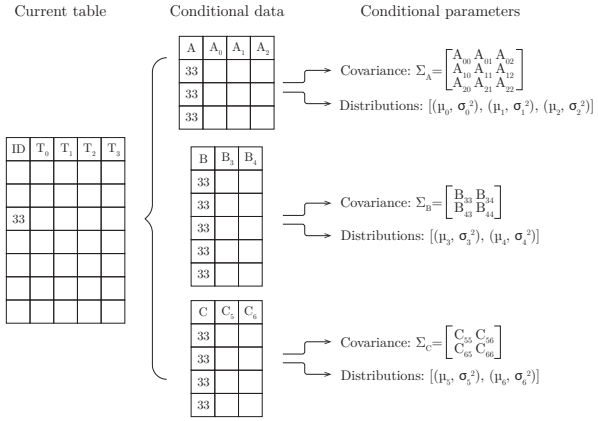

Fig. 4. An illustration of CPA for a row in table T with primary key "33". Tables A, B, and C refer to table T, so the lookup yields 3 sets of conditional data. Each is modeled using the Gaussian Copula, yielding conditional parameters.

This means there is at least one other table with a column that references rows in the current one. CPA comprises of 4 steps:

1) Iterate through each row in the table.
2) Perform a conditional primary key lookup in the entire database using the ID of that row. If there are m different foreign key columns that refer to the current table, then the lookup will yield m sets of rows. We call each set conditional data. Figure 4 illustrates such a lookup that identifies $m = 3$ sets of conditional data.
3) For each set of conditional data, perform the Gaussian Copula process. This will yield m sets of distributions, and m sets of covariance matrices, Σ. We call these values *conditional parameters*, because they represent parameters of a model for a subset of data from a child, given a parent ID. This is also shown by Figure 4.
4) Place the conditional parameters as additional values for the row in the original table.[2] The new columns are called *derived columns*, shown in Figure 5.
5) Add a new derived column that expresses the total number of children for each parent.

The extended table contains both the *original* and *derived* columns. It holds the generating information for the children of each row, so it is essentially a table containing original values and the generative models for its children. The SDV writes a the extended table as a separate CSV file, so we do not have to recalculate CPA for subsequent invocations of the same database.

[2]Some values repeat because $\Sigma = \Sigma^T$ We drop the repeats to save space.

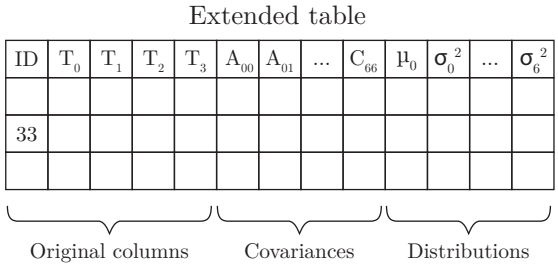

Fig. 5. The result of CPA. Every lookup for a row yields a value, such as μ_5 or B_{43}. The values form their own columns, resulting in an extended table.

Original Column Type	Replaced Column(s) Type
Categorical	Number
Datetime	Number
Number w/Missing Values	Number & Categorical
Categorical w/Missing Values	Categorical & Categorical
Datetime w/Missing Values	Datetime & Categorical

TABLE I. CONVERSIONS THAT MUST BE MADE WHEN PRE-PROCESSING. IF MULTIPLE DATA TYPES ARE LISTED, IT MEANS THAT MULTIPLE COLUMNS ARE CREATED FROM THE ORIGINAL COLUMN.

Subsequently, we can use Gaussian Copula process to create a generative model of the extended table. This model not only captures the covariances between the original columns, but the dependence of the conditional parameters on the values in the original columns. For example, it includes the covariance between original column T_0 and derived column μ_5^2.

C. Pre-Processing

Both Gaussian Copula and CPA assume there are no missing entries in the column, and that the values are numerical. When either of assumptions is false, a pre-processing step is invoked. This step ultimately converts a column of one data type into one or more columns of another data type, as summarized by Table I.

Note that some data types might require multiple rounds of pre-processing. For example, a column that is a datetime with missing values is first converted into two columns of type categorical and datetime. Then, those resulting categorical and datetime columns are further converted into number columns.

1) Missing Values: Missing values in a column cannot simply be ignored because the reasons for which they are missing may reveal some extra information about the data. As an example, consider a table representing people with a column called `weight`, which is missing for some rows. The reasons for missing data falls into one of three categories, so identified by the statistical analysis community [5]:

– Missing not at random: The data is missing because of what it's supposed to be. Perhaps people who are overweight chose not to disclose their weight, so knowing that the cell is missing probably means the weight is high.
– Missing at random:[3] The fact that the item is missing

[3]We realize that this is a confusing name. Think of *missing at random* to mean that a random subgroup decided not to supply data.

is linked with some other piece of data in that row. For example, perhaps a majority of females did not disclose their weight. So knowing that a person is female makes it more likely that the weight column will be missing.
– Missing completely at random: The fact that the item is missing tells us nothing about the structure of the rest of the data. For example, the database admin accidentally deleted some of the weights, randomly (oops).

In the first 2 cases, knowing that the value is missing provides further information about the data itself. Therefore, it is important to model missing values overall. Furthermore, a high level goal of the SDV is to model and synthesize data that mimics the format of the original. If the original data has some missing values, the synthesized must too. Modeling the null values solves this problem.

In the final case, it is not imperative that the missing values are considered from a numerical perspective, but the SDV does not know this may be the case. Hence, even though the third case is missing completely at random, the SDV must make a model.

When the SDV encounters any column that has at least 1 missing value, it replaces the column with two columns:

– A column of the same type, with missing values filled-in by randomly choosing non-missing values in the same column.
– A categorical column that contains "Yes" if the original data was present, and "No" if the data was missing for that row.

This solution ensures that the original column contains values for all rows, but also accounts for the fact that some were originally missing.

2) Categorical: Categorical columns may exist originally in the table, or may be a result pre-processing missing values. Categorical data also cannot be modeled by the Gaussian Copula or CPA.

When it encounters a categorical column, the SDV replaces it with a numerical column containing values in the range $[0, 1]$. To do this, it uses the following method:

1) Sort the categories from most frequently occurring to least.
2) Split the interval $[0, 1]$ into sections based on the cumulative probability for each category.
3) To convert a category, find the interval $[a, b] \in [0, 1]$ that corresponds to the category.
4) Chose value between a and b by sampling from a truncated Gaussian distribution with μ at the center of the interval, and $\sigma = \frac{b-a}{6}$.

Figure 6 shows a visual depiction of this conversion.

Note that while Gaussian distributions are completely defined by μ and σ^2 , the same is not true for these categorical distributions. Instead, they require new parameters representing the proportions of each of the c categories, $p_0, p_1, \ldots, p_c$ with $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$. These are the conditional parameters that are put in the extended table for categorical columns.[4]

[4]We save $p_0 \ldots p_{i-1}$ because the last proportion p_i can be calculated from the others.

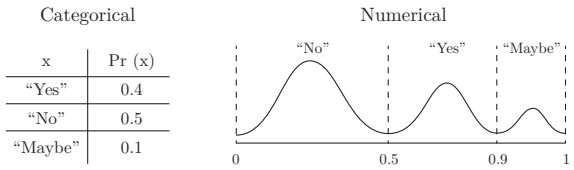

Fig. 6. The method that converts categorical variables to numerical data. Based on the proportions, "No" is assigned the interval $[0, 0.5]$; "Yes" is assigned $[0.5, 0.9]$, and "Maybe" is assigned $[0.9, 1]$. Each occupies its allocated interval with a Gaussian distribution.

Choosing a value using a Gaussian distribution gives dense areas at the center of each interval, but ensures that the numbers are essentially different. The inverse is also easy to compute: Given a value $v \in [0, 1]$, we find the interval that v corresponds to and return its category.

3) Datetime: Finally, many tables contain information containing times or dates that is represented as text. The SDV replaces such columns with numerical values. This is relatively straightforward, as any timestamp can be expressed as the number of seconds past Epoch (January 1, 1970). If timestamp represents a time before Epoch, then the numerical value is negative (number of seconds until Epoch).

D. Database Modeling

This final section describes the overall system by providing control logic for modeling an entire database. This consists of applying CPA recursively to calculate the model of the entire database.

We let D represent a database consisting of many tables, T. The relations between the tables are known, so we let $\mathcal{C}(T)$ represent the set of T's children, and $\mathcal{P}(T)$ represent the set of T's parents. Finally, we assume that our logic has access to CPA and pre-processing method we have described. Other mathematical functions include the cdf function, F, and the covariance Σ.

The CPA method works across a parent-child relationship. However, the children may have more children, so we must apply the CPA recursively down all of the parent's descendants. We call this recursive approach Recursive Conditional Parameter Aggregation, or RCPA. Algorithm 1 provides the logic for RCPA.

Algorithm 1 A recursive application of CPA to add derived columns to T.

```
1: function RCPA(T)
2:     for all C ∈ C(T) do
3:         RCPA(C)
4:     T ← CPA(T)
5:     T ← PREPROCESS(T)
```

Note that the CPA method returns the extended table. Line 4 saves the extended table as T. Finally, line 5 pre-processes T to convert the values into numerical data. The base case of this algorithm is for leaf tables, tables where $\mathcal{C}(T) = \emptyset$. Such tables are guaranteed by our non-circularity constraint.

When the SDV creates the overall model, it applies RCPA and uses the results to calculate the database model. The SDV's modeling algorithm calls the RCPA method on all tables without parents. Because RCPA is recursive, this ensures that all tables in the database ultimately go through the CPA method. Afterwards, it calculates the *cdf* functions, given by F, as well as the covariances by using the Gaussian Copula for all extended tables. The logic is given in Algorithm 2.

Algorithm 2 The overall modeling logic for the SDV for database D.

```
1: function SDV-MODEL(D)
2:     for all T ∈ D s.t. P(T) = ∅ do
3:         RCPA(T)
4:     cdf ← ∅
5:     cov ← ∅
6:     for all T ∈ D do
7:         cdf ← cdf ∪ F(T)
8:         cov ← cov ∪ Σ_{(Φ^{-1}(F(T))}
9:     return cdf, cov
```

The algorithm saves and returns all the *cdf* and covariances of the tables. The *cdf* functions are calculated using the table returned by the extend function. The covariance is calculated after applying the Gaussian Copula to that table. Together, the *cdf* and covariances form the generative model for database D. When this function returns, the user can control the amount and type of data to synthesize.

In summary, the overall database model saves the following for every table:

- The extended table (calculated by Algorithm 1)
- The *cdfs* of columns in the extended table (returned by Algorithm 2)
- The covariances of columns in the extended table (returned by Algorithm 2)

V. Data Synthesis

This chapter presents the details of the last step in the SDV's workflow: Synthesizing data based on the calculated database model.

We break up the synthesis into two categories:

- **Model-Based**: The user wishes to synthesize data relying on the model that has been computed. For example, a user may want to synthesize the entire database of their customer information.
- **Knowledge-Based**: The user already has some information about the data, and wishes to synthesize the rest it. For example, the user may want to synthesize information for particular types of customers (female, age 22, etc.).

The SDV can perform both types of synthesis. Each section of this chapter provides details for the two cases. The final section presents our API endpoints.

A. Model-Based

Model-based synthesis is based on being able to sample data from the calculated distribution and covariances. The modeling was learned using pre-processed numerical values that represent numbers, datetime, categories, and missing values. Once we sample from the model, we can factor in the primary key and foreign key relations to synthesize tables, and ultimately the entire database.

1) Sampling Numerical Values: All numerical values can be sampled from the distributions and covariances of the columns. Call the set of *cdf* functions F, and the covariance matrix Σ. The method to sample numerical values is given by algorithm 3. Assume that there are n columns, so that $|\Sigma| = |F| = n$.

Algorithm 3 Sampling numerical values from distribution and covariances of the columns.

```
1: function SAMPLE(F, Σ)
2:     v ← random n-dimensional Guassian vector
3:     Find Cholesky decomposition, LL^T = Σ
4:     u ← Lv
5:     x ← [F_0^{-1}(Φ(u_0)), F_1^{-1}(Φ(u_1)), ... F_n^{-1}(Φ(u_n))]
6:     return x
```

Line 4 of this algorithm uncovers a vector, u, in Copula space. Then, line 5 converts it back to the original space by applying the inverse of the Gaussian Copula. The returned vector, x, provides a value for all columns that were converted to numerical data (numbers, categorical, datetime, and missing values).

Once the numerical value is returned, we can post-process it to form data that looks like the original. This is accomplished by:

- Converting back from numerical values to datetime or categorical values
- Removing values for columns that were not originally in the table. This includes all derived columns from CPA.
- Making values blank if they are supposed to be missing by looking at the binary `"Yes"` or `"No"` value that is sampled.

2) Row Synthesis: Overall row synthesis relies on sampling. We use two separate methods depending on if the row does or does not have any parents.

To synthesize a row with no parents (and therefore, no foreign key references), we use the overall *cdfs* and covariance computed for its table, T_F and T_Σ. To synthesize a row with a parent, we recall that its parent row, p, has conditional parameters that describe the *cdfs* and covariances for its children, p_F and p_Σ. These are the values we use to generate the child. Both methods are shown in Algorithm 4.

The first function, MakeRowFromTable expects an extended table T as input. This can be either the original extended table, or a synthetic version of the extended table. The second function MakeRowFromParent expects a single row, p, containing all values from the derived columns as input. Similar to the first function, p can be either an original row or a synthesized row. Note that both returned values require post-processing to look like the original version of the data.

3) Database Synthesis: Synthesizing the entire database just consists of synthesizing multiple rows and child rows recursively. We begin with a table that has no parents, and call

Algorithm 4 Making a row based on information in the table T or in the parent row p.

```
1: function MakeRowFromTable(T)
2:     id ← random unique ID value
3:     x ← Sample(T_F, T_Σ)
4:     return [id, x]
5:
6: function MakeRowFromParent(p)
7:     id ← random unique ID value
8:     foreign key ← ID of p
9:     x ← Sample(p_F, p_Σ)
10:    return [id, foreign key, x]
```

the MakeRowFromTable to generate rows for that table. Using the rows from that table, we can create the children. Recall that each parent row, p, also stores the number of children it contains, p_n. We can use this number to call MakeRowFromParent the appropriate number of times. Finally, we recurse to synthesize the children of those children until the entire database is synthesized. The logic is shown by Algorithm 5.

Algorithm 5 The overall database synthesis logic for the SDV.

```
1: function SDV-Synthesize(D)
2:     for all T ∈ D s.t. P(T) = ∅ do
3:         repeat
4:             row ← MakeRowFromTable(T)
5:             MakeChildrenRows(row)
6:         until reached user-defined threshold
7:
8: function MakeChildrenRows(p)
9:     if p has children then
10:        repeat
11:            child ← MakeRowFromParent(p)
12:            MakeChildrenRows(child)
13:        until reached p_n children
```

We envision that a majority of use-cases will be model-based. They will require the user to synthesize the entire database, or a subset of tables in those databases.

B. Knowledge-Based

In this section, we briefly describe algorithms we use if the user wants to synthesize data based on prior knowledge they already have. For example, if user is synthesizing data for internally testing an application, they may realize that the application needs a balance of values. As a result, the user may decide to synthesize rows for underrepresented female customers only.

This requires two modifications from the model-based method. First, the sampling method from Algorithm 3 no longer works because some of the values included in F and Σ have already been observed. This requires us to perform a special update to uncover a new F' and Σ' for just the unobserved data. Second, it requires us to infer what the parent might be based on the value that the user provides.

1) Sampling Updates: If some values are already observed and inputted by the user, then original sampling will not work by itself, because it will return synthesized values for all columns. To account for observed data, it is necessary to update update the Σ matrix, as well as the mean vector μ. Initially, $\mu = 0$ due to the Gaussian Copula process.

Let k represent all the observed (known) variables, and u represent the unobserved (unknown) variables the user wishes to synthesize. Then we can rearrange the Σ matrix and μ vector to bring all the unknown variables to the top:

$$\Sigma = \begin{bmatrix} \Sigma_{uu} & \Sigma_{uk} \\ \Sigma_{ku} & \Sigma_{kk} \end{bmatrix}$$
$$\mu = \begin{bmatrix} \mu_u \\ \mu_k \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

With this configuration, the SDV can update Σ and μ with the known observations to get a new Σ' and μ' for just the unknown.

$$\begin{aligned} \Sigma' &= \Sigma_{uu} - \Sigma_{uk}\Sigma_{kk}^{-1}\Sigma_{ku} \\ \mu' &= \mu_u + \Sigma_{uk}\Sigma_{kk}^{-1}\left(obs - \mu_k\right) \\ &= \Sigma_{uk}\Sigma_{kk}^{-1} obs \end{aligned}$$

Where obs is the user-inputted vector containing the known values. Note that the Σ' matrix has dimensions $|u| \times |u|$ and the μ' matrix has exactly $|u|$ elements. This is because they only describe the relations for the columns with unobserved values.

Now, the SDV knows the new Σ' and μ', along with the corresponding *cdf* functions for the unknown variables $F_i, i \in u$. These new values can be used in the sampling algorithm (Algorithm 3) with a slight modification: In step 4, we add the μ to the vector u. This will return all the values in the row that contain numerical information, some of which is post-processed back into categorical or datetime information. However, it does not include foreign key information, which is why we need to perform inference to find the parent.

2) Parent Inference: If the user has observed certain values for a row and the row has parents, then it is necessary for us to infer what the parent row may be.

Recall that each parent row, p contains conditional parameters that describe the covariances, p_Σ, and *cdfs*, p_F, of its children, so the problem of picking a foreign key simplifies into a log likelihood estimate. For the given data, x, the probability of x belonging to some parent p depends on p_Σ and p_F. This, in turn, is described by the Gaussian Copula:

$$-\log\left(\mathcal{L}_p(x)\right) = -\log\Phi_{p_\Sigma}\left[\Phi^{-1}\left(p_{F_0}\left(x_0\right)\right), \Phi^{-1}\left(p_{F_1}\left(x_1\right)\right), \ldots, \Phi^{-1}\left(p_{F_n}\left(x_n\right)\right)\right]$$

The SDV chooses a parent row of x from a weighted distribution of $-\log\left(\mathcal{L}_p(x)\right)$, $\forall p$. The foreign key of x is the primary key of parent p.

Note that the value $\Phi^{-1}\left(p_{F_i}(x_i)\right) = \pm\infty$ if $p_{F_i}(x_i) = 1$ or 0, making the overall log likelihood approach 0. This happens when the child's data is out of bounds for a parent. For example, if the conditional parameters in p define min and max and the observed row is not in the interval, then p is not a good candidate for a parent.

The overall SDV is able to perform many types of synthesis and inference based on a combination of all the algorithms presented in this section. Given any set of parent or children rows and columns, the SDV can ultimately synthesize the missing values and return them to the user in the same format as the original table.

C. API Endpoints

When the SDV is ready for the synthesis stage, it provides the user with a `database` object, from which the user can access individual tables with `database.get_table(name)`. The table object is used for the synthesis.

We have packed both model-based and knowledge-based synthesis in two synthesis endpoints. The first is `table.synth_row`, that allows the user to synthesize a full row based on the table or its parent rows, while also performing updates based on observed values. The second is `table.synth_children`, that allows the user to generate all children based on a parent table. This method is a convenient packaging of the MAKEROWFROMPARENT algorithm that allows the user to easily synthesize full tables and databases.

1) `table.synth_row`: If they are synthesizing a full row, the user can just call the `synth_row` function without any arguments. This generates all of the modeled data. The SDV generates a unique primary key, as well as any textual data that is not modeled. As a final step, the SDV formats the data to mimic the original. This means performing the following checks and transformations:

1) If the column <x> has a corresponding categorical column ?<x> check its value. If ?<x> = "NO" then the value should be missing. Set the value of <x> to null.
2) Remove all columns that were not in the original table.
3) If the original column was a datetime, take the numerical value and convert it back to a datetime with a user-provided time regex.
4) If the original column was a category, perform the transform from Section IV-C2 in reverse to recover the correct category.

As keyword arguments, the user can input any observed values for columns that exist in the table. The SDV performs the appropriate inference to synthesize a full row based on the input. These can include derived columns too, because derived columns are modeled by the SDV. Table II shows some examples.

2) `table.synth_children`: When calling the `synth_children` function, the SDV synthesize entire tables that represent children of the current table. The number of children generated for each unique primary key of the table are based on the value of the derived `count` column.

Command	English Description
`customer.synth_row()`	Synthesize a completely new customer
`customer.synth_row(gender=F)`	Synthesize a female customer
`customer.synth_row(?weight=No)`	Synthesize customer with missing weight

TABLE II. EXAMPLE COMMANDS USING THE `SYNTH_ROW` FUNCTION TO CREATE NEW STORES. ORIGINAL COLUMNS AND DERIVED COLUMNS CAN BE INPUTS TO THE SYSTEM.

This function completely generates the all the columns of the children table, including any other foreign key constraints that the children may have. This function is intended to help the user generate entirely new databases. The user first calls `synth_row` on every row in the parent table, and the `synth_children` recursively until the entire database is synthesized.

This meets our usability for the SDV: Provide a simple interface for the user that gives them control to synthesize data at any granularity. The cell and row granularities are covered by the `synth_row` method, while the table and database granularities are covered by `synth_children`.

VI. EXPERIMENTAL SETUP

In this section, we describe the experimental setup we used to validate the SDV's ability to synthesize realistic data. Our experiments evaluate the SDV in terms of its effectiveness at scaling Data Science Efforts without the need to share real data.

To demonstrate this, we designed a crowdsourcing experiment. The overall goal was to test whether data scientists working with synthesized data could generate valuable features just as easily as they would on the original data. In order to test this, we found publicly available relational datasets with prediction problems for a particular column. For each dataset, we performed the following steps:

1) Run the SDV on the dataset to create the generative model.
2) Use the model to synthesize data with varying degrees of noise.
3) Hire data scientists to solve prediction problem with a particular version of the dataset (synthesized or the original).

This section describes the experimental process. We provide details about the datasets and the methods used to synthesize the data, and we describe the experimental setup, with 4 conditions.

A. Datasets

We used a total of 5 relational datasets in the experiment. Two came from an online relational dataset repository [6], and three came from Kaggle [7]. Table III provides a summary of each dataset. The prediction problem for each of the datasets was turned into a classification problem by discretizing the target column's values, if they weren't originally categorical.

The rest of this section provides details about the data and the prediction problems for each of the datasets.

Biodegradability: The first dataset describes various chemical compounds in terms of molecules, atoms, and bonds [8]. The

Dataset Name	Source	# Tables	# Classes	# Exemplars
Biodegradability	Relational Repo	5	5	249
Mutagenesis	Relational Repo	3	2	145
Airbnb	Kaggle	4	12	5000
Rossmann	Kaggle	2	8	1017209
Telstra	Kaggle	5	3	7381

TABLE III. SUMMARIES OF THE FIVE RELATIONAL DATASETS USED FOR THE CROWDSOURCING EXPERIMENT. THE FINAL COLUMN REFERS TO THE NUMBER OF CLASSES THAT THE PREDICTION PROBLEM ENCOMPASSES.

prediction problem is the biodegradability of each molecule in water, as measured by the column `logp` in the `Molecule` table. The `logp` value describes the half-life of the biodegradation for the molecule. For this experiment the `logp` values were discretized into 5 classes, and the objective was to predict the class to which the molecule belongs. Figure 7 presents the schema of this data. To create a synthetic database for

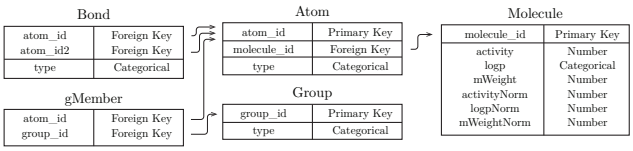

Fig. 7. The schema for the biodegradability dataset. Molecules consist of multiple atoms. Two atoms are joined by bonds, and multiple atoms can be part of an atom group.

this prediction problem, the SDV synthesizes each group in turn: first new molecules, then new atoms, and finally new bonds and group members. (Note that is it is not necessary to synthesize new groups, because a row in `Group` is not a child of molecule.)

Mutagenesis: Similar to the biodegradability dataset, the mutagenesis dataset [9] is also related to chemical compounds described by molecules, atoms, and bonds as shown by Figure 8.

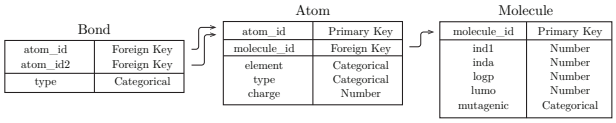

Fig. 8. The schema for the mutagenesis dataset. The overall structure is the same as for biodegradability, but there is no `gMember` or `Group` tables associated with the dataset.

The objective of this prediction problem was to predict the `mutagenic` column in the `Molecule` table. "Mutagenicity" refers to the ability of a chemical to cause mutations in a strand of DNA. Thus, the `mutagenic` column is binary, and contains either a `'yes'` or `'no'` value. Creating synthetic data was straightforward for this SDV: create new molecules, new atoms for those molecules, and new bonds for those atoms. Thus, all three tables needed to be synthesized for the prediction problem.

Airbnb: The Airbnb datasets comes from a Kaggle competition [10] hosed by the lodging site Airbnb [11]. It consists of web access log data from each of its users, as described in Figure 9.

The prediction problem for this dataset is `country_destination` from the `Users` column. This represents the country that a particular user booked a lodging for. A total of 10 popular countries are labeled using a shortform (for example `'ES'` for Spain), while an 11th category called `'other'` encompassed all non-popular countries. Finally, a 12th category, labeled `'NDF'` (No Destination Found) indicated that the user did not end up booking lodging using the site. To create synthetic data for this

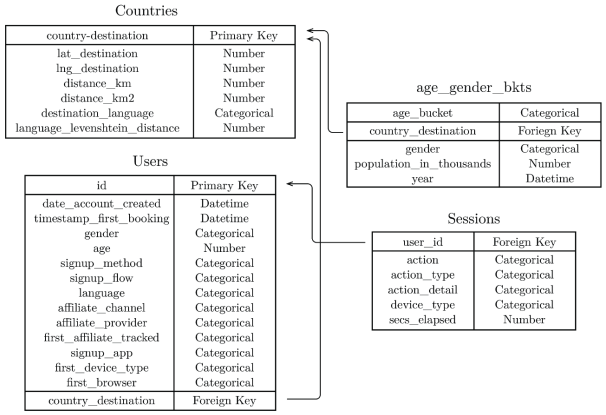

Fig. 9. The schema for the Airbnb dataset. Each "user" is an account made on Airbnb, and each session describes a particular access made to the website. The `Countries` table provides general information about the country, while `age_gender_bkt` provides information about people traveling to those countries.

prediction problem, the SDV synthesized new users, and then synthesized new sessions for those users. It was not necessary to synthesize `Countries` because it was the parent table of the table containing the prediction problem. It was also unnecessary to synthesize `age_gender_bkts` because it was not a child of `Users`.

Rossmann: Kaggle's Rossman Store Sales dataset comes from another competition [12] based on historic sales data for different stores in the franchise [13]. The Rossmann franchise is one of the largest drug store companies in Germany, and the dataset provided information about each individual store, as well as weekly details about it. This is described in Figure 10.

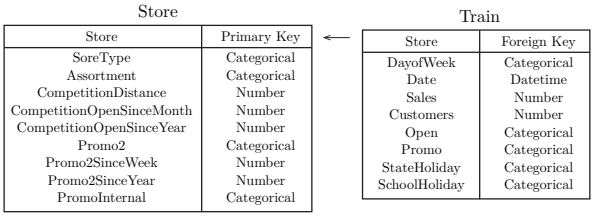

Fig. 10. The schema from the Rossmann Store dataset. Each store is a different store location of the Rossmann franchise, and each row in `Train` corresponds to a particular day in the store.

The prediction problem was the `'Sales'` field in the `Train` table, that represented the total revenue made by the store in that day. Because this was a continuous variable, it was discretized into 8 bins. Creating a synthetic version of the data

Group	Biodegradability	Mutagenesis	Airbnb	Rossmann	Telstra
0	control	table noise	key noise	no noise	control
1	no noise	key noise	control	table noise	no noise
2	table noise	control	no noise	key noise	table noise
3	key noise	no noise	table noise	control	key noise

TABLE IV. THE VERSIONS OF EACH DATASET THAT WERE AVAILABLE TO EACH EXPERIMENT GROUP. WHILE THIS SETUP MAY BE BIASED TO SOME ORDERING EFFECTS, IT ENSURES THAT A SINGLE GROUP RECEIVES DIFFERENTLY SYNTHESIZED VERSIONS OF DIFFERENT DATASETS.

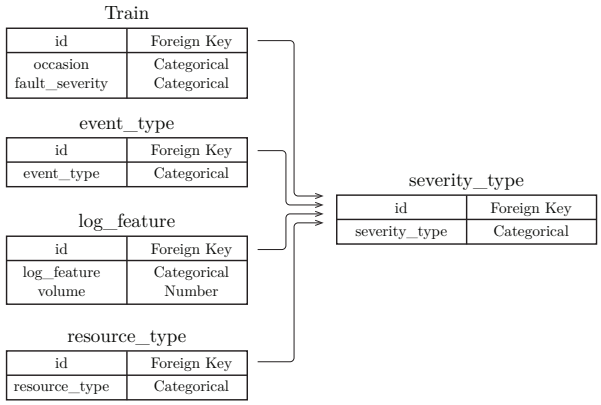

Fig. 11. The schema for the Telstra dataset. Each column named of `'id'` represents a location and time. The information is split up by tables with meta-information about the event, log, resources, and severity of a possible network outage.

meant synthesizing different stores first, and then synthesizing the rows in `Train` for each of those stores.

Telstra:The final dataset was from a Kaggle competition focused on a dataset from Telstra [14], a telecommunications service from Australia that provides mobile phones and broadband internet. The layout of the dataset is described by Figure 11.

The prediction problem is to classify the `'fault_severity'` column of the `Train` table. This is either `'0'` for no network outage, `'1'` for a few outages, or `'2'` for many outages.

To create a synthesized version for this data, it was only necessary to synthesize new rows of the `Train` table, because this table had no children.

B. Crowdsourcing Experiment Setup

For each dataset, the SDV created three versions of data, each with a different noisy condition. These conditions were:

- **Control**: The data scientist is given the original version of the dataset.
- **No Noise (Synthesized)**: The data scientist is given the synthesized output from the SDV's algorithm.
- **Table Noise (Synthesized)**: The data scientist is given synthesized output from the SDV's algorithm, with noise introduced by taking every covariance value, σ_{ij}, $i \neq j$ and halving it, effectively reducing the strength of the covariance.
- **Key Noise (Synthesized)**: The data scientist is given synthesized output from the SDV's algorithm, with noise introduced by randomly sampling a primary key for the foreign key relation instead of performing an inference.

Data scientists with some experience analyzing data were hired for the experiment. These data scientists were assigned to one of four groups, which determined the versions of the datasets. This is specified by Table IV.

All data scientists were given a briefing in which they were told to write complex features for each of the datasets. We used Feature Factory[5] as the interface for conducting the experiment. Each dataset was exposed to the subjects as a separate iPython notebook. The notebook contained some background information about the domain, as well as access to a variable `dataset` that contained a list of table objects belonging to the dataset.

Data scientists were not told which version of the data they were given. In order to test their features, they were provided with a method called `cross_validate` that automatically computed their features and returned an accuracy score based on their version of the dataset. Feature Factory saved finished and submitted scripts, which we used for our analysis.

VII. RESULTS

We recruited a total of 34 data scientists via the freelance website UPWORK. They were divided into groups, and were presented the five datasets as per Table IV. Each group had at least one control dataset. In Table VII, we present the total number of features written by the data scientists in a group for a dataset.

The two questions we wished to explore when interpreting these features were:

- Accuracy: was there a difference in the crowd's predictive accuracy when they were given original vs. synthetic data?
- Qualitative findings: Did the data scientists using synthetic data describe themselves as feeling confused?

	Group 0		Group 1		Group 2		Group 3	
Datasets	f	s	f	s	f	s	f	s
Airbnb	374	9	309	8	38	7	209	6
Telstra	2369	6	122	5	413	2	2085	6
Biodegrad.	26	9	48	6	8	5	35	5
Mutagenisis	216	8	150	5	7708	6	286	3
Rossman	176	6	118	5	35	6	218	10

TABLE V. NUMBER OF DATA SCIENTISTS, s, WHO WROTE FEATURES FOR A PARTICULAR DATASET AND THE TOTAL NUMBER OF FEATURES, f, THEY WROTE.

[5]An interactive i-python based platform hosted on Amazon. The discussion of the platform specifics is beyond the scope of this paper

A. Accuracy

A chief question we wanted to address was: did the synthetic dataset affect the quality of the features generated in terms of predictive accuracy? Was there any significant difference between the features generated with synthetic data and those generated with the original dataset? To test this, we followed these steps:

- We computed every feature script written by a data scientist on the original training dataset (the control).
- Once the feature was extracted, we computed its predictive accuracy by training a classifier with the feature alone and evaluating 10-fold cross validation accuracy on the training dataset.
- We then grouped these accuracy numbers into 4 subsets. The first set contained all the accuracy numbers for features written using the control data. We call this S_0. The second set contained accuracy numbers for features written on top of no-noise data S_1, the third set was table noise S_2 and the fourth set was key noise S_3. (Note that all feature values were calculated on the control dataset.)
- We then performed the two-sample `t-test` between three pairs (S_0, S_1), (S_0, S_2) and (S_0,S_3). The two sample `t-test` returns a test decision for the null hypothesis that the data in vectors S_0 and S_i comes from independent random samples from normal distributions with *equal* means and *equal* but unknown variances. The alternative hypothesis is that the data comes from populations with *unequal* means. The result is a logical value.
  - If Result = True, it indicates the rejection of the null hypothesis at the α significance level.
  - If Result = False, it indicates a failure to reject the null hypothesis at the α significance level

In the table VII-A, each row presents the results of tests performed for one of the 5 datasets. We present the $p-value$, as well as the *lower* and *upper* bounds of the confidence interval. If the result is $True$, it implies that there was a difference in the feature accuracies. We employ a total of 15 tests across different datasets.

Key findings

- For **7 out of 15** comparisons, we found no significant difference between the accuracy of features developed on the control dataset vs. those developed on some version of the synthesized data; that is, the result of the test was $False$.
- When we examined the confidence intervals for the remaining 8 tests, we found that for *half*, the mean of accuracies for features written over synthesized data was higher then for those written on the control dataset.

Overall, we can say that there is no statistically significant difference between the accuracy scores of data scientists with control data and data scientists with synthesized data. This confirms our belief that scientists can be as productive with synthesized data as they can with control data. It remains to be seen how the level of noise in the synthesized data affects the accuracy.

B. Qualitative Findings

Finally, we consider the subjective feedback provided to us by the data scientists. In particular, we observed the questions different data scientists asked to determine whether they were confused by the data they were provided.

A majority of the questions dealt with technical issues (such as with Feature Factory or the experimental setup) that were unrelated to the synthesized data. Many subjects were also unhappy that they did not have access to the target column's data. Although we purposefully designed the experiment in this way, so that users could focus on writing features without knowing the answers, we may consider allowing data scientists to see the target column of the training data in the future, in order to mimic a typical workflow more closely.

Some data scientists were confused about the relationships that existed between the tables. One user in particular did not understand how two columns of the `bond` table could be foreign keys to the `atom` table in the Mutagenesis dataset (Figure 8). Other users defaulted to writing features from the target table only. We explicitly encouraged the data scientists to join the tables to explore the entire dataset.

Only 1 question was related to the actual values in the dataset. A data scientist in group 3 indicated that the `ages` column in the `Users` table of the Airbnb dataset (Figure 9) had unrealistic ages. This data was synthesized with table noise. However, upon closer inspection, it appears that the original data for Airbnb also had unrealistic data in the `ages` column (max age was 2004). The SDV synthesized data within the correct bounds when compared to the control data.

However, we did realize that the SDV's bounds will not always be correct if we are synthesizing children from a noised parent table. This is because the parent holds the min and max values of their children's columns. If the parent is noised, then the min and max may represent unrealistic data.

Ultimately, we found that the SDV successfully modeled each of the relational datasets, and used the generative models to synthesize data that data scientists could realistically work with.

REFERENCES

[1] S. Kramer, N. Lavrač, and P. Flach, *Propositionalization approaches to relational data mining*. Springer, 2001.

[2] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer, "Learning probabilistic relational models," in *IJCAI*, vol. 99, 1999, pp. 1300–1309.

[3] F. J. Massey, "The Kolmogorov-Smirnov test for goodness of fit," *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[4] L. Rüschendorf, *Mathematical Risk Analysis*. Springer, 2013, ch. 1.

[5] R. Little and D. Rubin, *Statistical Analysis with Missing Data*. Wiley, 2002.

[6] J. Motl and O. Schulte, "The CTU prague relational learning repository," *CoRR*, vol. abs/1511.03086, 2015. [Online]. Available: http://arxiv.org/abs/1511.03086

[7] "Kaggle," https://www.kaggle.com/, accessed: 2015-05-16.

[8] H. Blockeel, S. Džeroski, B. Kompare, S. Kramer, and B. Pfahringer, "Experiments In Predicting Biodegradability," *Applied Artificial Intelligence*, vol. 18, no. 2, pp. 157–181, 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.3797

	Control vs. No Noise (Synthesized)				Control vs. Table noise (Synthesized)				Control vs. Key Noise(Synthesized)			
Dataset	Result	$p-value$	l_b	u_b	Result	$p-value$	l_b	u_b	Result	$p-value$	l_b	u_b
Airbnb	True	9e -5	-0.1203	-0.0403	False	0.693	-0.0244	0.0162	False	0.2986	-0.0266	0.0082
Telstra	False	0.3191	-0.0514	0.0168	False	0.9657	-0.0191	0.0200	True	2e-19	0.0390	0.0606
Biodegrad.	False	0.3867	-0.0511	0.1305	False	0.6718	-0.1743	0.1138	True	0.0095	0.0291	0.2003
Mutagenesis	True	6e-61	-0.1584	-0.1250	True	2e-2645	-0.2973	-0.2558	True	7.3e-61	-0.2173	-0.1714
Rossman	True	2.29e-10	0.0866	0.1615	True	0.0147	0.0115	0.1046	False	0.1029	-0.0130	0.1404

[9] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity." *Journal of medicinal chemistry*, vol. 34, no. 2, pp. 786–797, 1991.

[10] "Kaggle airbnb new user bookings," https://www.kaggle.com/c/airbnb-recruiting-new-user-bookings, accessed: 2015-05-16.

[11] "Airbnb," https://www.airbnb.com/, accessed: 2015-05-16.

[12] "Kaggle rossman store sales," https://www.kaggle.com/c/rossmann-store-sales, accessed: 2015-05-16.

[13] "Rossmann mein drogeriemarkt," https://www.rossmann.de/verbraucherportal.html, accessed: 2015-05-16.

[14] "Telstra network disruptions," https://www.kaggle.com/c/telstra-recruiting-network, accessed: 2015-05-16.