The New Iris Data: Modular Data Generators

Iris Adä Michael R. Berthold

Nycomed-Chair for Bioinformatics and Information Mining
University of Konstanz, Box 712, 78457 Konstanz, Germany.
Iris.Adae@Uni-Konstanz.DE, Michael.Berthold@Uni-Konstanz.DE

ABSTRACT

In this paper we introduce a modular, highly flexible, open-source environment for data generation. Using an existing graphical data flow tool, the user can combine various types of modules for numeric and categorical data generators. Additional functionality is added via the data processing framework in which the generator modules are embedded. The resulting data flows can be used to document, deploy, and reuse the resulting data generators. We describe the overall environment and individual modules and demonstrate how they can be used for the generation of a sample, complex customer/product database with corresponding shopping basket data, including various artifacts and outliers.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Application—*Data mining*; I.6.0 [**Simulation and Modeling**]: General

General Terms

Design, Algorithms

Keywords

data generation, pipeline tool, artificial data

1. INTRODUCTION

When it comes to teaching and testing data mining algorithms, it is often enormously helpful to use data that exhibits known effects for which the underlying mechanisms (e.g. distribution or occurrence probabilities) are known. However, for the different types of usage, rather different types of data are needed. When using data for teaching it is desirable to show students the wanted effects on a small, understandable data set with a reasonable size. On the other hand the lecturer may want to use larger, more complex data sets to challenge the students. But even then, the data given to the students should exhibit clear and controllable effects. When using data to test new algorithms, it is more important to ensure the data contains diverse types of the same class of patterns. There are, of course, other reasons for creating artificial data. Often researchers create data to illustrate their approaches or to visualize the application of a new tool.

Real world data is often not understood adequately (especially if it is sufficiently complex) and usually also only shows a few of the desired effects—in addition to effects that are not desirable. Generating artificial data could solve this problem if the data generator is flexible enough. However, this flexibility requires that data is created following a set of underlying rules, distributions and other patterns and also allows the addition of artifacts such as outliers and missing values.

A number of tools has been published to create artificial data but most are targeted at testing individual algorithms. Tools for this purpose tend to be designed to cover only the needs of that particular class of methods. Creating special data for one purpose only, is usually doable but extending the functionality later to cover other properties is often much more difficult if not impossible. Only a very small number of more general approaches exist. Still, none of them is flexible enough to truly allow for the generation of complex artificial data with several types of underlying patterns or artifacts.

This paper presents a novel, easy-to-use, general and modular data generation environment. It can be used to archive, reuse and document the generation of complex data sets with heterogeneous requirements in terms of distributions, patterns, outliers, and other artifacts. The new modules are integrated into an existing, modular open source data analysis environment. KNIME [2] offers an intuitive and graphical workflow editor, which allows the assembly of complex data processing protocols by chaining together individual processing modules, or nodes and was therefore a natural choice for the addition of data generation abilities. And, as we demonstrate later, using existing data processing nodes can also help in the data generation process itself. In addition, the analysis and visualization capabilities of KNIME help to validate that the created data sets do indeed have the desired properties. But of prime interest for this paper are the data processing nodes. In concert with the new data generation nodes introduced in this paper, complex, multi-relational data sets following various underlying distributions and patterns and exhibiting various types of outliers and other artifacts can be created. Note that modeling the data generation process as a flow also results in another positive effect: the flow can be used as an explanation of the

KDD'10, July 25–28, 2010, Washington, DC, USA.

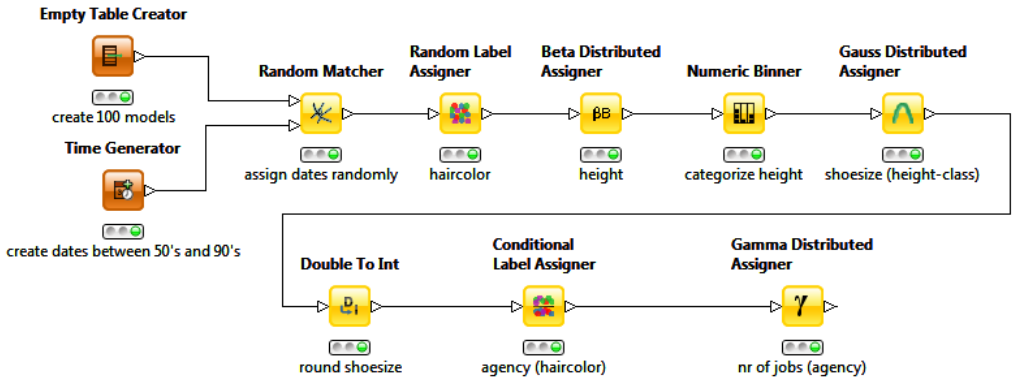


Figure 1: A simple flow demonstrating the generation of a small example data set.

generation progress, as it can be understood intuitively. Another benefit is the exchangeability of the components, the modular visual approach lends itself naturally for the testing of different routines without changing the whole generation progress every time. Figure 1 shows an example of a simple data generation flow—we explain the types of nodes used in more detail in the following, but the flow should largely be self-explanatory.

The paper is organized as follows: We first provide a brief overview of existing tools for data generation, which is followed by a description of the basic data generation nodes. Afterwards we demonstrate how these modules can be combined to create example data sets and we conclude by briefly describing the generation of a complex supermarket data set, which can be downloaded (together with the generator workflow) from our webpage[1].

2. RELATED WORK

As mentioned above many authors write their own special purpose tools to create synthetic data to evaluate or demonstrate specific algorithms. More general tools exist as well but they are still centered around particular classes of patterns. The Quest synthetic tool[2], developed by the IBM Almaden Research Center, contains two programs for generating data. The first one generates associations and sequential patterns whereas the second one creates data for classification. An extension of the Quest tool was proposed in [11] where a new method for generating temporal data sets was introduced resulting in the ARtool, which is a Java implementation of the IBM generator. Another extension of the Quest tool was proposed in [3]. The authors claim that the original distributions did not follow realistic ones and suggested an alternative model, using a power law distribution for the item occurrence.

For the evaluation of clustering and outlier algorithms, a data generator was presented in [12]. It generates two dimensional data sets based on the selected density, outlier and a certain "difficulty" level. The numbers are generated following a Gaussian or a uniform distribution. Some of these and other methods have also been integrated into the Weka system [6].

Another well-known aspect of data creation is the generation of random numbers following various distributions. A well-written overview of existing methods for Gaussian distribution can be found in [14] and for beta distribution in [7].

An additional approach for synthetic data generation concerns the enrichment of data. It produces a larger amount of data based on effects in the original input data. In [5] a synthetic data definition language (SDDL) was developed. However, the required XML document quickly becomes complex. Another aspect of synthetic data generation is the anonymization of data, while yet another approach to data generation relates to benchmarking, often in the context of performance comparisons of databases.

There are, of course, many other areas requiring data generation such as the automated testing of (software) systems. A survey on test data generation can be found in [10] or [4].

It is apparent that quite a lot of work has already been done in the area of generating artificial data. However, most of this work concentrates on generating data for a special purpose or for a specific type of pattern. Also, to the best of our knowledge, no work has yet been published with the purpose of modularizing the generation progress to obtain a flexible and extendable tool.

3. MODULAR DATA GENERATION

In the following we briefly introduce the base platform before delving into details on new data generation modules. All data generation modules have been implemented as extensions to the well-known data mining workflow tool KNIME [2].

3.1 KNIME

KNIME, the Konstanz Information Miner, was developed by the Chair for Bioinformatics and Information Mining at the University of Konstanz, Germany. It is released under an open source license (GPL v3) and can be downloaded free of charge[3]. The modular data exploration platform enables the user to visually create data flows (often referred to as pipelines), selectively execute some or all analysis steps,

[1] http://www.knime.org/datagen
[2] http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html
[3] http://www.knime.org

and later investigate the results through interactive views on data and models. The base version already incorporates hundreds of processing nodes for data I/O, preprocessing and cleansing, modeling, analysis and data mining as well as various interactive views, such as scatter plots, parallel coordinates and others. It integrates all analysis modules of the well known Weka data mining environment and additional plugins allow, among others, R-scripts[4] to be run, offering access to a vast library of statistical routines.

Using KNIME has the great advantage that a lot of data processing nodes already exist, which can also be used for data generation. For instance, the ability to split data enables the simple insertion of outliers or other artifacts in a subset of the data only (which is then later reconcatenated with the unaffected data). This side effect is demonstrated later.

3.2 Getting Started

After downloading KNIME, the built-in update mechanism (File→Update KNIME) is used to install the data generation modules described in this paper. Note that for every node that is based on random number generators, the user can supply a seed. This allows "random" data sets to be recreated in a reproducibly manner. Changing (or disabling) the seed creates different data sets during each run. All nodes rely on the Java class java.util.Random, which relies on a linear congruential pseudo random number generator [9]. A description of the algorithm can be found in [8].

3.3 First Steps

The first step in modular data generation almost always consists of the creation of a table. A table can either be based on an existing file or database table (using existing nodes) or it can be generated from scratch. For the latter, the data generation plugin includes a simple module to create an empty table (*Empty Table Creator*) with a fixed number of (empty) rows. In addition to the number of rows, this node allows a pattern to be specified for each row's key—by default a standard enumeration scheme is used. The resulting table therefore contains rows which only contain the row identifier (rowkey) and no additional data.

Once one or more starting tables have been created (or read in) the user can start to add additional information, e.g. columns. These can contain classes (e.g. female/male, young/old) or numbers (e.g. age or income).

3.4 Adding nominal attributes

Several options exist to add nominal attributes to an existing data table. We can either add purely randomly distributed values or create them depending on an existing column.

The basic node, which adds a new string column to the data, is the *Random Label Assigner*. The user has to enter a set of labels $L = l_1, ..., l_n$ and for each label the probability p_i that the label l_i occurs. To ease usability of the node, the probabilities are normalized to ensure that they sum up to 1, so that the user can also enter percentages or other fractions.

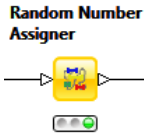
[4] http://www.r-project.org

In the event of more than just a handful of labels, manual configuration of the *Random Label Assigner* tends to become cumbersome. In this case, configuration can also be read in from a second table, e.g. from a file containing names and their occurrence frequencies. The *Random Label Assigner (File)* node is subsequently used to create a new column using the data fed into the second data port. The column containing the labels and the column containing the probabilities can be selected if more than one string or double columns exist. Here too, probabilities are normalized if they do not already add up to one.

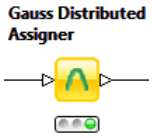
Adding independent nominal values is, indeed, only of limited interest; the *Conditional Label Assigner* inserts new nominal values depending on the labels of another, existing column. This can also be used to create simple rules (we introduce more powerful rule insertion modules later). In the configuration dialog the user enters the probability $p(l|d)$ of a new label l being inserted if the label d is seen in the existing column. These conditional probabilities are normalized again to fulfill the total sum requirement. Note that by using the column joiner as a preprocessing step, it is also possible to model dependencies on more than one column. This is also demonstrated below.

3.5 Adding continuous attributes

Obviously, nominal values are only one aspect of data generation, creating continuous attributes is at least equally important. The extensions discussed here provide different probability distributions. In each of those nodes the user has the option of using one distribution for all rows (modeling independent probabilities) or of defining the values of the distribution for each value of a nominal attribute of the given column separately (modeling conditional probabilities).



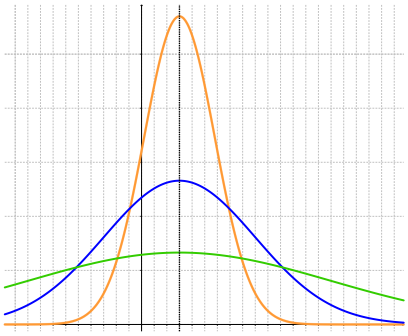
The simplest random number distribution is created by using the *Random Number Assigner* node. After defining the minimum and maximum it creates a new column containing uniformly distributed random values.



A second node creates data based on a Gaussian distribution. It needs to be configured with the mean and the variance (the square of the standard deviation) of the random variable. In Figure 2(a) the distribution function is displayed for a fixed mean and three variances. In order to generate Gaussian random numbers, we followed one of the recommendations in [14] and chose the polar method by Box, Muller and Marsaglia. The description of the algorithm can be found in [8]. As the Gaussian distribution is not bounded, a global minimum and maximum for the whole attribute has to be defined.

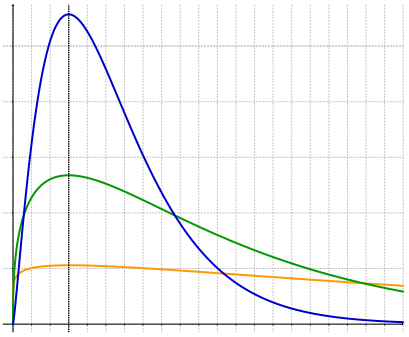
However, this type of distribution should be used with care as there is no guarantee that the average of the column will be equal to the selected mean: some values, lower than the minimum or higher than the maximum, might have been skipped.

To create bounded random values two other distributions are available, the beta and the gamma distribution. The beta distribution can be particularly helpful in generating a value that shows a peak and is bounded on both sides. If the constraint is to be imposed on only one side, gamma distribution is the node of choice.



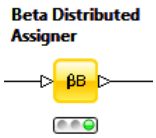
(a) Example for Gaussian distribution, using the same mean and three different variances.

(b) The distribution function $\beta(p,q)$ with one peak value and three different values for the p-value.



(c) The distribution function $\gamma(p,q)$ with a fixed peak and three scaling parameters.

Figure 2: In addition to uniformly distributed data, three other distributions are available.



The Beta distribution is typically described by the $\beta(x)_{p,q}$ function. It is configured with p and q. Modifying these parameters results in different data variations.

$$\beta(x)_{p,q} = \frac{B_{p,q}(x)}{\int_0^1 B_{p,q}(t)dt}$$

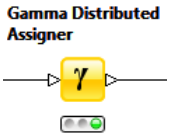
$$\text{with } B_{p,q}(x) = x^{p-1}(1-x)^{q-1}$$

To spare the user considerations about the behavior of p and q, the node asks for four more intuitive parameters: the minimum, the maximum, the peak of the value and the shape parameter p. Increasing the last parameter leads to an increase of the gradient of the distribution. The effect of the shape parameter p is depicted in Figure 2(b).

As mentioned above, the node discussed here only asks for parameter p and the peak (in addition to minimum/ maximum). The peak is set to the extrema of the density distribution. q is calculated using the transposed equation

$$q = \left(\frac{1}{\text{peak}} - 1\right)(p-1) + 1.$$

Generation of beta distributed numbers is a well studied topic in the field of stochastic modeling. Hung et al. evaluated various generation algorithms in [7]. We follow their suggestions and use the most suitable generation algorithms for the given choice of parameters.



One commonly used choice for a distribution resulting in a half-open interval, is the gamma distribution, which is defined using shape parameter p and scaling parameter b.

$$\gamma(x)_{p,b} = \frac{b^p \Gamma_{p,b}(x)}{\int_0^\infty \Gamma_{p,1}(t)dt}$$

$$\text{with } \Gamma_{p,b}(x) = x^{p-1}e^{-bx}$$

An advantage of the gamma distribution is that it is strongly connected to the chi-square distribution. A chi-square distributed value with n degrees of freedom can be generated using $p = \frac{n}{2}$ and $q = \frac{1}{2}$, as the functions can be transposed as follows : $\chi^2_n(x) = \gamma(x)_{\frac{n}{2},\frac{1}{2}}$. It can also be transformed into an exponential, a beta or a logarithmic distribution.

Similar to the beta distribution, again the user only enters the peak of the distribution and additionally scaling value b. The shape parameter p is calculated with the given peak and scaling value b using the following equation $p = \frac{\text{peak}}{b} + 1$. The random variable x, which is drawn from $\gamma(x)_{p,1}$ is subsequently scaled by b. In order to generate the gamma random variables, we chose the algorithm of Tanizaki [13] which works for all configurations of the shape parameter p.

To summarize, it is possible to create new columns based on four different distributions. They can be open bounded (Gaussian distribution), bounded on one side (Gamma distribution) or bounded on both sides (uniform and Beta distributions). Apart from uniform distribution, all others allow the specification of a peak. In the application section, we show how these distributions can be used to model different data effects.

3.6 Adding rules

In addition to numerical and nominal value generation, another important goal in data generation is to offer methods to insert rule based patterns. When it comes to inserting realistic rules, a number of complex requirements have to be taken into account. Normal rule based dependencies can be injected using the *Rule Engine* node available in the standard KNIME release. We focus on inserting association rules in the following as this requires a more complex setup.

Association rules are based on two subsets X and Y of an item set I ($X, Y \subseteq I$). Typically Y consists of only one item and the corresponding association rule reads as $X \Rightarrow Y$, indicating that Y appears (with a minimum frequency and a minimum confidence) if X also appears in the set. The support and confidence measures are based on the overall set of transactions T. A transaction $t \in T$ can for instance represent the goods a customer bought at a particular time where the subsets X, Y are product combinations. A rule is considered to be important when the support ($\text{supp}(X)$) and confidence ($\text{conf}(X \Rightarrow Y)$) of the rule are above the predefined thresholds, see [1]:

$$\text{supp}(X) = \frac{|\{t \in T | X \subseteq t\}|}{|T|}$$

$$\text{conf}(X \Rightarrow Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X)}$$

Association rule generation allows the independent injection of such rules, usually based on an already existing set of transactions. These transactions can be generated depen-

Figure 3: Inserting association rules in existing shopping basket collections.

dent on other attributes influence e.g. shopping behavior such as income or age. Some customers with more income may buy more items or make purchases more frequently. We can now insert additional items and rules but at the same time change the overall structure of the transactions as little as possible. The rule injection is therefore divided into two steps: first we make sure the support of item sets is sufficiently high and second we make sure the confidence thresholds are reached as well.

The *Random Item Inserter* node inserts selected item i with the configured probability p_i into transactions, which are represented by lists of items. This is either achieved by adding it randomly to the list to increase the probability to the desired value, or by removing it, if the item is already contained too often. In the former case, one can specify the replacement of an existing item or add it to the list, allowing existing frequencies of other items to remain constant. This node enables the definition of the support of a specific item.

The *One Rule Inserter* inserts the items in Y into transactions, given two parameters, the support of Y and the confidence of the association rule $X_1, X_2, \cdots \Rightarrow Y$. Similar to the node above, occurrences of Y are either added or removed to existing lists depending on the frequencies in transaction. In addition we can also fix the list sizes or extend the list here.

To summarize, first, the support of the rule is ensured by using the *Random Item Inserter*. Second, the *One Rule Inserter* increases the confidence of the rule, by inserting or deleting the consequent item to transactions.

In the example discussed later, the shopping baskets are not represented as sets but as tables listing pairs of basket and product identifiers. In order to use the two nodes described above, one needs to first create the sets and later split them again. The workflow in Figure 3 depicts this process. The basket-product table is grouped to create one row per basket containing the set of contained products. The workflow then injects one rule for *paté de foie gras, chips => gummi bears*. Accordingly, the support of the two products in the antecedent is defined first. Afterwards the rule is inserted into the data. In order to convert this data back to the basket/product table we need to ungroup the product sets by splitting the column and converting it to rows using the *Unpivoting* node.

Obviously these modules can also be used to generate frequent item sets rather than complete rules. This modular way of inserting item sets and rules also enables dependencies to be created at different levels of abstraction by first injecting rules for more general product categories and later refining them to actual products.

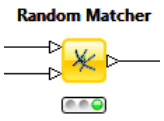
3.7 Additional Functionality

In addition to the modules described above, a number of other special purpose nodes have been added to KNIME. These nodes serve to enrich and modify information in existing data, rather than to generate new data from scratch.

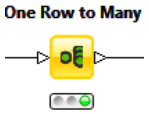
The *Stresser* node, as its name indicates, can insert artifacts into a given categorical or numerical column to create more realistic-looking data. Based on the attribute type and selected behavior two stress options are available for both types of data. In all cases, the user defines which percentage of the values of the chosen column are affected.

For stressing nominal values there are two options. The *switching* option allows one value to be changed randomly to another possible value contained in the column. The other option allows nominal values to be stressed by mutating the original string (two letters of the string are swapped as a result). This can, for instance, be used to create spelling errors in street or city names.

Adding stress to continuous values can be used to create outliers in clusters. The original value is replaced by a random number, which is either kept within the range of the column, to create moderate outliers, or outside the columns' range, to create drastic outliers. Note that the "outside the range" option does return a value from the whole range of double hence this option should be used with care.



Additional helpful functionality is provided by the *Random Matcher*. Its functionality can best be described as a randomized joiner, as it combines the columns of two tables by matching two column values in a random fashion. If the input tables are described by t_1, t_2 and the selected columns are d_1, d_2, for each row in t_1 the algorithm tries to find another row in t_2 that has not already been used and where d_1 equals d_2. If there is no such row in t_2, a random one is added. If there are no more unused rows with the same value, all matching rows are marked as unused, and an already employed row is used. This node also offers the possibility of not choosing columns but instead just randomly adding rows of the second table and skipping the unique selection of the second row. This node can for instance be used to randomly combine first and last names from two lists of real names.



Another useful functionality is provided by the *One Row to Many* node, which multiplies the input lines based on a column specifying the number of repetitions. This can be useful to create a list of all individually purchased products from baskets where the products are listed only once with the purchase frequency.

3.8 Making Use of Existing Nodes

As already mentioned, one reason for adding the data generation modules to an existing data processing and analysis environment was the ability to make use of existing functionality. Many types of data generation tasks can be easily accomplished that way without the need for additional modules.

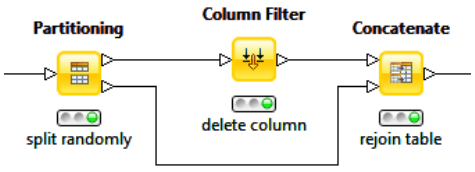


Figure 4: Inserting missing values into one column of the data.

As a first example, none of our nodes is able to directly create missing values. However, when using existing KNIME nodes, missing values ca be created very easily. We split the data, either totally randomly (*Partitioning* node) or with respect to another column, filter the selected values and finally rejoin the table. Figure 4 shows the resulting simple flow adding a certain percentage of missing values. The same mechanism can, of course, also be used to add stress or outliers or other artifacts to a subset of the overall data. A slightly more complex example of this kind of partitioning is shown in Figure 5. Here we apply three different types of modifications to three subsets of the data which, in this case, are derived based on the value of a given column using the *Nominal Value Row Filter* node.

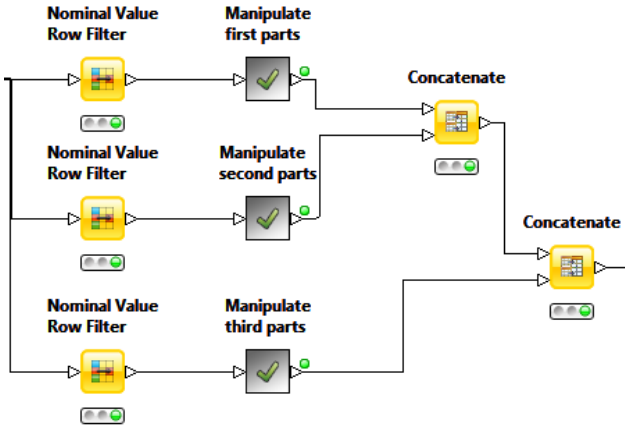


Figure 5: Splitting the data based on a nominal column value allows each branch to be manipulated individually.

Basing rules or dependencies on values of more than one column, the *Column Combiner* can be used to create a new column holding the combined values (Figure 6 shows an example). This can be helpful in modeling multidimensional dependencies, such as combining gender and occupation to model the difference between male employees and female students.

Furthermore there are other very useful nodes that can

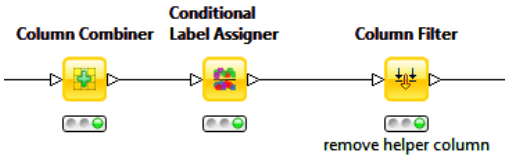


Figure 6: Combining columns allows to model multi dimensional dependencies.

be applied to execute code snippets written in different languages. KNIME currently offers nodes for executing R, Perl, Python and of course Java-Code. Additional, mathematical functions can be applied to numerical columns, and last, but not least, the *Rule Engine* node allows a user-defined set of sequential rules to be applied to the input table and insertion of the outcome as a new column.

4. EXAMPLES

Having introduced various ways to create specific types of artificial data, this section demonstrates how to use these modules together to create a more complex, real-world-like shopping basket data set. The resulting data is primarily created for the purpose of teaching and demonstrating various steps of the data analysis process—hence the main focus is on ensuring it contains real-world-like artifacts and patterns (and not so much on overly convincingly pretending to be a real world dataset itself). Due to space constraints we only show some of the particularly interesting aspects of the resulting workflow in detail. The entire workflow can be downloaded online[5].

4.1 Customer Data

The workflow fragment shown in Figure 7 creates the starting point of the customer table. After creating an empty table where the number of rows corresponds to the number of customers we add gender information using a 52 : 48 distribution. Afterwards we assign the last name based on a list of real last name distributions which is read from a file. In the full workflow we also add first names using the same process. The *Conditional Label Assigner* is then used to add the occupation dependent on the gender. In the full workflow, occupation, of course, also depends on other customer properties.

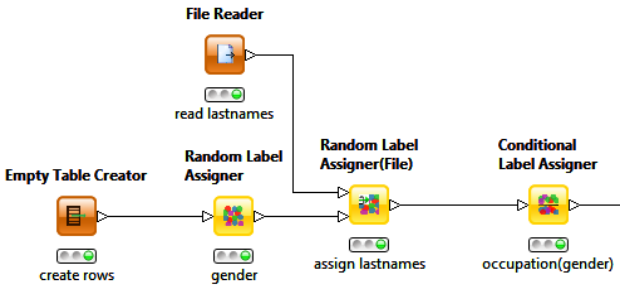


Figure 7: A workflow fragment creating names and employee status depending on the gender.

[5] http://www.knime.org/datagen

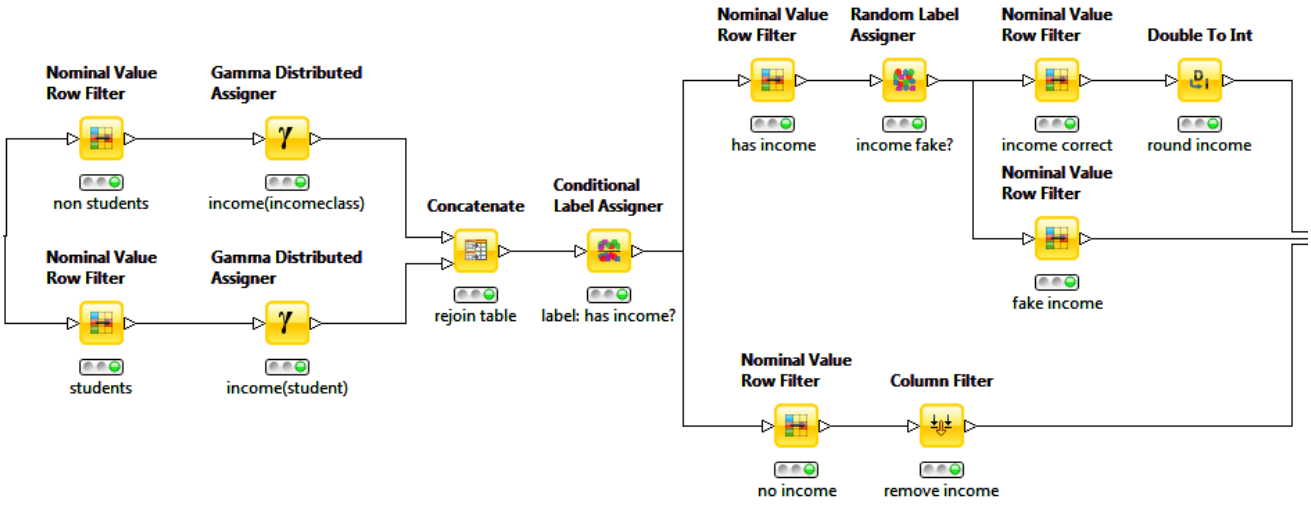


Figure 9: Generating income based on a given age class column, including additional missing values and fake/incorrect incomes.

Another aspect of customer generation relates to the modeling of complex age distributions. The workflow in Figure 8 shows the generation of a multi-modal age distribution. We first create an age category and then use different medians and variances for each category to generate the final age. The result is displayed in the histogram at the bottom of Figure 8. There is a peak of people in their forties and another, smaller peak around 70. Here, we tried to remodel the current age pyramid for the German population[6]. The colors indicate the different age categories. The various, age-group dependent means, variances, and weights should be apparent. In another example–still part of the customer creation workflow–we illustrate the generation of an income depending on various customer attributes. The input to this workflow already contains age class and job category for each customer. The workflow adds a new column containing the customer's income. Note that we may not actually include the income in the final dataset but only use it as a hidden variable for the generation of e.g. the size and value of the

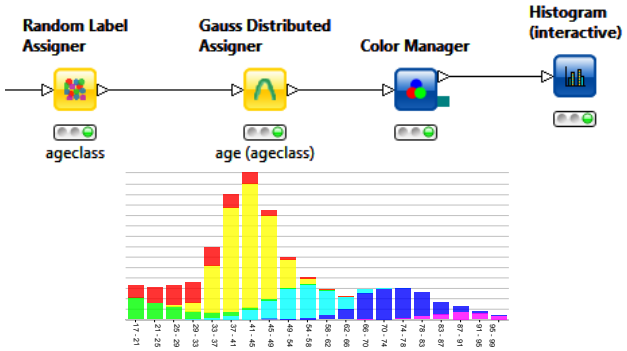


Figure 8: Multimodal age distribution: workflow and resulting distribution (colors indicate age classes).

[6]see `http://www.destatis.de/bevoelkerungspyramide/`

shopping baskets. The income is based on gamma distribution corresponding to increasing peaks with increasing age. The group of students is treated differently, assuming that their income is more or less age independent. Subsequently, missing values and fake values (in this case also the mistaken entry of monthly instead of annual incomes) are added to the column.

Figure 9 shows the corresponding workflow. First we split the whole data table into students or non-students using two row filter nodes. Afterwards, the income of the non-students is distributed based on their age (class). A gamma distributed value with a peak at 400 (euro) is used for the students. To inject missing values, we assign an additional value containing income/no_income based on the occupation of the customer, with the *Conditional Label Assigner* node. Executives, for example, may be less likely to enter their income into a poll. By filtering the income column on the "no income" values and afterwards concatenating the table, missing values are created. Additionally we set 5% of incomes as fake incomes. A fake income in this case is an income that is not rounded to a whole number, however other operations can be considered, such as entering annual instead of monthly incomes or the incorrect use of decimal separators. Finally all tables are concatenated and helper columns, such as the column containing the no_income/income labels, are removed.

4.2 Shopping Basket Data

In addition to the customer data, the resulting tables should also contain product information on product categories, product names and their prices. To emulate reality product names should also be similar to real products. We skip the generation of the product table here, as it is constructed similar to the customer data generation workflow. Additional tables should contain information about the shopping baskets such as time of purchase and identifier. The fourth and final table should store mapping of the baskets to each of the products it contains.

The existing customer and product tables already contain information that can be used for basket generation. The

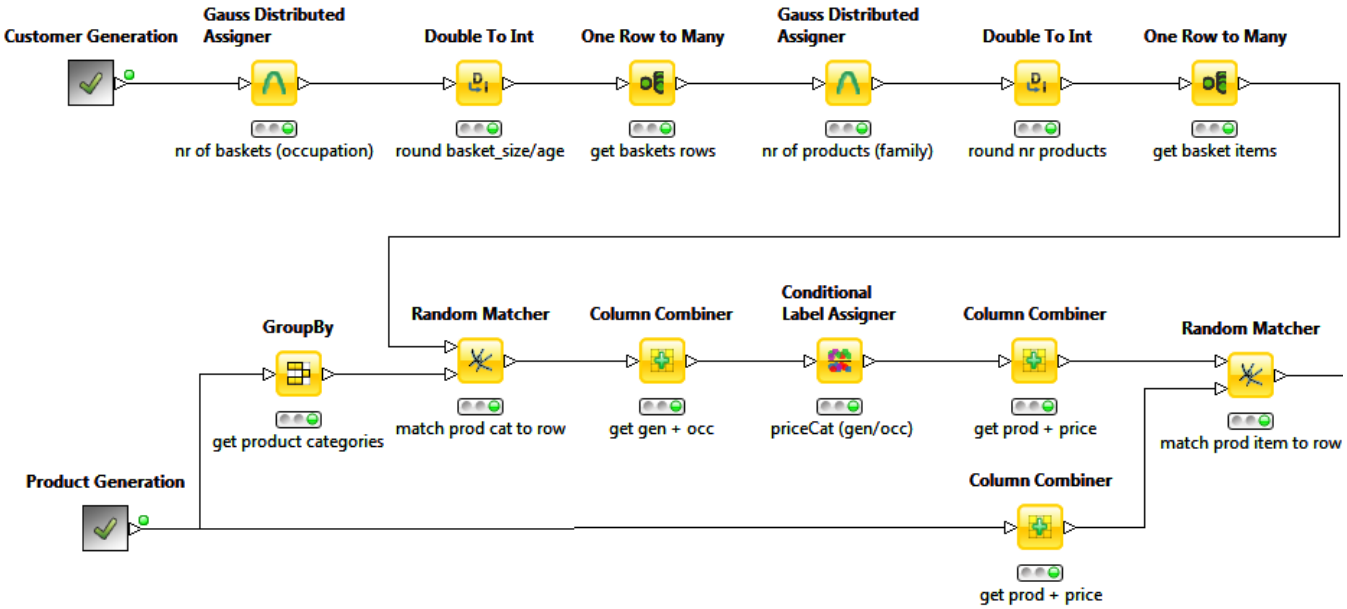
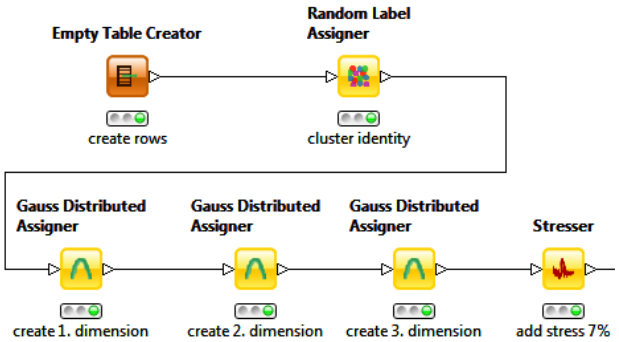


Figure 10: Generating association rules based on input.

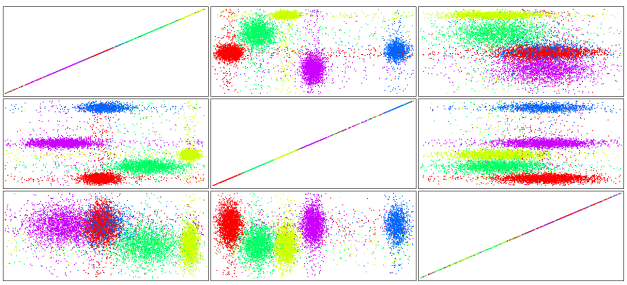
occupation of a customer should be used to determine the number of baskets bought by the respective customer in this supermarket and the family status should influence the number of products (and price categories) in each basket. As demonstrated in Figure 10, two *Gauss Distributed Assigner* nodes are used for both of these intermediate numbers. The values are rounded to whole numbers and the rows are duplicated to create one row per basket per customer. This duplication is accomplished by applying the *One Row to Many* node, which duplicates the row based on a number in another column. Afterwards one row exists for each sold product, containing information about the customer who bought it and the basket. Note that this table does not yet contain information about the actual product being sold.

In the next step we assign individual products to each row. One constraint is that the price category should be based on gender and occupation. Female executives buy more expensive products than male students, for example. Each item row is assigned a product category, obtained by grouping this column, using the *Random Matcher*. We then pick the price category based on gender and occupation. First these two columns are combined to one identifying string which is then used in the *Conditional Label Assigner* to create a price category. The result of this part of the process is a product and price category for each purchased product. To create the final product these two are combined and the *Random Matcher* is used to find a pseudo random product for each selection. As mentioned in Section 3.7, the *Random Matcher* node finds a unique row with the same string for each input row of the first table. After the creation of the initial shopping basket data, we can inject additional dependencies using association rules as described earlier.

The full workflow demonstrates how shopping baskets can be generated based on the income and family status of the customer and how different aspects of product categories can also be included in the generation process. The final workflow can be used to easily generate tens or hundreds of thousands of customers buying from an arsenal of millions of products incorporating various types of outliers, missing entries and other artifacts.



(a) The workflow generating three-dimensional data.



(b) The created data (colors indicate original cluster membership).

Figure 11: Creating five clusters in three dimensions with added noise.

4.3 Cluster Generation

We want to illustrate one other aspect of data generation by briefly demonstrating the creation of noisy cluster data. In Figure 11(a) a three dimensional data set is created, containing five clusters. For this purpose the *Empty Table Creator* creates the basic empty table and using the *Random Label Assigner* each line is attributed to one cluster. Subsequently one *Gaussian Distributed Assigner* is used for each dimension. Note that this node uses the cluster identifier as dependent variable to create different normal distributions for each cluster. As a last step we add outliers to some percentage of the three dimensional values, using the *Stresser* node, configured to stay inside the input range for each column. The scatter matrix shown in Figure 11(b) illustrates the created data. The color indicates the original cluster identity.

5. MISCELLANEOUS

In this section we briefly mention some additional benefits of using KNIME as the base platform:

- **Reproducibility.** By using a workflow tool we provide the ability to intuitively document and reproduce the data generation process. To give others access to the exact same generation progress, the workflow can be exported and made available for download, since the underlying platform is freely available. In addition to simply re-creating the same data set, the workflow can than also be modified and extended and hence adopted to individual needs.

- **Parametrization.** KNIME offers the possibility of defining parameters, or variables, for the generation progress. Such variables can either be defined for the whole workflow or read from a file or table. The variables can be used for the random seed value or to control other aspects of the data generation process such as the number of customers, clusters, or occupation groups. Starting the flow several times with different variable settings results in different data sets, but with the same underlying motifs. This can, for example, be applied for teaching. To give each student their own individual data set, the flow is started several times, only with a varying seed value. Another option is to start the flow with different parameters to further modify the nature of the data set.

- **Batch Mode.** Finally KNIME can also be launched in a batch mode. In this mode the user can, for example, start the workflow with different variables and create data sets based on different conditions automatically without having to launch the workflow GUI every time.

6. CONCLUSION

In this paper we have introduced an environment for modular data generation. The entire generation progress is split into simple steps, breaking the process into clear and understandable modules. By integrating the new data generation modules into an existing data processing and analysis platform, existing modules can be used easily. We demonstrate how this environment can be used to create complex, multi relational data that adheres to different distributions and patterns and constrains various outliers and other artifacts. We plan to enhance the functionality of the KNIME data generation modules continuously, most notably we will add more nodes to support time series data in the future. Of course, contributions from the community are very welcome as well.

7. ACKNOWLEDGMENTS

This work was partly supported by the DFG Research Training Group GK-1042 "Explorative Analysis and Visualization of Large Information Spaces".

8. REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, 1993.

[2] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel. KNIME: The Konstanz Information Miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer, 2007.

[3] C. Cooper and M. Zito. Realistic Synthetic Data for Testing Association Rule Mining Algorithms for Market Basket Databases. *Data Engineering*, pages 398–405, 2007.

[4] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28. Citeseer, 1999.

[5] J. Eno and C. Thompson. Generating Synthetic Data to Match Data Mining Patterns. *IEEE Internet Computing*, pages 78–82, 2008.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[7] Y.-C. Hung, N. Balakrishnan, and Y.-T. Lin. Evaluation of Beta Generation Algorithms. *Communications in Statistics - Simulation and Computation*, 38(4):750–770, 2009.

[8] D. Knuth. The art of computer programming: Seminumerical algorithms, volume 2, 1981.

[9] D. Lehmer. Random number generation on the BRL highspeed computing machines. *Math. Rev*, 15:559, 1954.

[10] P. McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[11] A. Omari, R. Langer, and S. Conrad. TARtool: A Temporal Dataset Generator for Market Basket Analysis. *Analysis*, pages 400–410, 2008.

[12] Y. Pei and O. Zaïane. A Synthetic Data Generator for Clustering and Outlier Analysis. Technical report, Citeseer, 2006.

[13] H. Tanizaki. A Simple Gamma Random Number Generator for Arbitrary Shape Parameters. *Economics Bulletin*, 3(7):1–10, 2008.

[14] D. B. Thomas, W. Luk, P. H. Leong, and J. D. Villasenor. Gaussian random number generators. *ACM Comput. Surv.*, 39(4):11, 2007.