Data Mining a Design Repository to Automatically Generate Functional Representations in Python

Alex Mikes

Oregon State University

Design Engineering Lab

School of Mechanical, Industrial & Manufacturing Engineering

mikesa@oregonstate.edu

1. Introduction and Motivation

Product design in engineering is a well-studied process [1], yet many aspects remain difficult and hard to define, especially during the fuzzy front end of concept generation. However, the concept generation phase is the one part of the design process where there is the most room for creativity and innovation [2]. During concept generation in product design, the focus is on gathering accurate customer needs, deriving the functionality of the intended product, and ideating potential form solutions. The functional model is a well-known decomposition approach that allows designers to develop a graphical representation of product functionality [3] [4]. The Functional Basis terms provide consistency in nomenclature as part of the functional model [5] [6]. However, as Nagel et al. point out, there is still inconsistency in user input in the structure of functional models [7]. This is due to the fact that building functional models is a difficult process for both novice and experienced designers. While the function of a product is critical in linking customer needs to a form solution, designers often prefer to skip a functional decomposition and move directly to component-based solutions. Designers often benchmark existing products during the concept generation phase, and they typically take inspiration from these existing products to design in terms of components rather than functionality [8].

Design repositories can provide designers with data at multiple levels of abstraction, and offer a source of inspiration for products. This software leverages research from the multi-decade long project developing a design repository, which is currently housed online through Oregon State University [9] [10] [11] [12] [13] [14] [14] [15]. The Design Repository is comprised of over 130 consumer-based electro-mechanical products.

Building consistency into the grammar and syntax in functional models, has been a prominent field in recent research [5] [6]. Similar consistency with language and syntax is built into the data in the repository, for example functions are entered using the functional basis terms, and components are populated with a component basis term developed by Bohm and Stone [16]. This terminology allows a consistent taxonomy for finding associations that remain consistent throughout each product. These discovered associations can then be used to predict functionality of a component to assist designers and students in building functional models during the concept generation phase and, ultimately to automate the process of entering additional products into design repositories.

This software mines the data in the Design Repository to find the associations between component, function, and flow. It uses these associations to automatically generate the functional representations of new products based on the constituent components and the most likely function-flow combinations for those components as they exist in the repository. For verification, the software is able to test its accuracy by comparing the results of automation with the known combinations of component-function-flow for products that already exist in the repository. Because it automatically finds functional representations, this package is titled *Autofunc*[17]. The functional representations for each component can then be connected to each other based on how the components physically connect in the product. This is the starting point for automating functional models, which requires additional rules about ordering functions and visually organizing the flows.

For example, a rule could be used that says an electric cord must import electrical energy before it can be transferred, so the functional representation for a cord would look like the one shown in Figure 1 below.

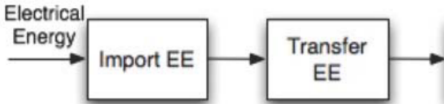

Figure 1: Example of Functional Representation of an Electric Cord

2. Methodology

This software uses two different techniques for data mining that are numerically equivalent. This provides flexibility for future explorations and a more mathematically rigorous methodology.

2.1. Probability

The first method for data mining is a probability-based counting technique that finds the number of instances of certain combinations and divides by the total number of instances to find the probability of that particular combination occurring. For example, if searching for the probabilities of the functions and flows found in a battery, the data set for mining might looks like the one in Table 1 below.

Product ID	Component	Function	Flow
1	Battery	Supply	Electrical
1	Battery	Store	Electrical
1	Battery	Transfer	Electrical
1	Blade	Separate	Solid
2	Battery	Store	Electrical
2	Battery	Supply	Electrical
2	Battery	Import	Electrical
2	Switch	Actuate	Electrical

Table 1: Example of Data Set

In this example data set, counting the combinations of Component-Function-Flow for each instance and dividing by the total number of instances would give the probability of each combination for the battery component. For example, the probability of a battery having the combination of "Battery-Store-Electrical" is the number of combinations of "Battery-Store-Electrical" divided by the number of instances of the component "Battery." In this example, it would be 2/6 or 1/3. The results of the example data set are shown in Table 2 below.

Component	**Function-Flow**	**Delta %**	**Sum %**
Battery	Store Electrical	0.33	0.33
	Supply Electrical	0.33	0.66
	Transfer Electrical	0.17	0.83
	Import Electrical	0.17	1

Table 2: Probability Results of Example Data Set for Battery

2.2. Association Rules

The second method used for data mining is a well-defined technique called association rules. Association rules describe the relationship between items in item sets. The typical application of association rules is in a market-based analysis. For example in a supermarket, 90% of customers who buy bread and butter also buy milk [18]. The Apriori algorithm is often used to find these type of associations. The algorithm is well

documented to be useful for dealing with large datasets and iteratively looks for frequent itemsets [19]. This software uses the Apriori algorithm to find the associations between component and function-flow.

Association rules are defined by three measures: support, confidence and lift. Thresholds of each measure set by the user control the output of the algorithm and can influence the results. This software uses association rules to discover how often a component-function-flow combination occurs in a data set. For example, how often does component *battery* and function-flow *store electrical* occur compared to other *battery* combinations? Examples of calculating each measure for the same example data set in Table 1 are shown below.

Let C be an itemset, $C \rightarrow$ F is an association rule and T a set of transactions of a given database, where C = Component, F= Function-Flow, T = Total. From this, three measures of association are defined as:

$$Support = \frac{C \bigcup F}{T}; \tag{1}$$

$$Confidence = \frac{Support(C \bigcup F)}{Support(C)}; and \tag{2}$$

$$Lift = \frac{Confidence(C \rightarrow F)}{Support(F)}. \tag{3}$$

The results of each measure for each combination of component-function-flow are shown below in Table 3.

Component	Function-Flow	Support	Confidence	Lift
Battery	Supply Electrical	0.25	0.33	1.32
Battery	Store Electrical	0.25	0.33	1.32
Battery	Transfer Electrical	0.125	0.17	1.36
Battery	Import Electrical	0.125	0.17	1.36
Blade	Separate Solid	0.125	1	8
Switch	Actuate Electrical	0.125	1	8

Table 3: Results of Association Rules for Example Data Set

Support determines the probability of an item within all of the item sets [20]. In the fictional example seen in Table 1, the support of *battery* and *supply electrical* would be 0.25, or the number of times *battery* and *supply electrical* appear in the data set (2) divided by the total number of items in the data set (8).

Confidence determines the probability of two items appearing in the same itemset. The confidence of *battery* and *supply electrical* is 0.33 or the number of times *battery* and supply electrical appears (2) divided by number of times *battery* appears in the data set(6).

Lift is the ratio of the support of the association and the support values of the individual items within the itemset. For our example, lift is the confidence of component and function-flow divided by the support of the function and flow. A lift greater than 1 indicates that the function and flow is likely to be associated with the component. A lift value of less than one indicates that the function and flow are unlikely to be associated. Lift values provide the user with information about how likely the association is while controlling for how popular the second item is in the data set. Continuing this example, the lift for *battery-supply-electrical* is the confidence for *battery-supply-electrical* (0.33) divided by the support of *supply-electrical* (0.25).

Examining the results of the probability metrics and the confidence measure in association rules reveals their numerical equivalence. Both methods are used to allow for the simple nature of the probabilities and the robust flexibility of the apriori algorithm for potentially large datasets.

3. Implementation

The general flow of the software is shown in Figure 2 below. It imports a data set, finds the associations of component-function-flow, finds a functional representation within a certain threshold of probability or confidence (e.g. the top 70% of probability/confidence), and creates a functional representation for each component that is in the input set. For verification, the software can use products already in the repository for which the component-function-flow combinations are already known to compare the results of automation

to the existing information and generate a numerical metric of accuracy known as the match factor. This functionality is achieved through modules that correspond to each of the boxes in the flow chart, with their corresponding module names shown below.

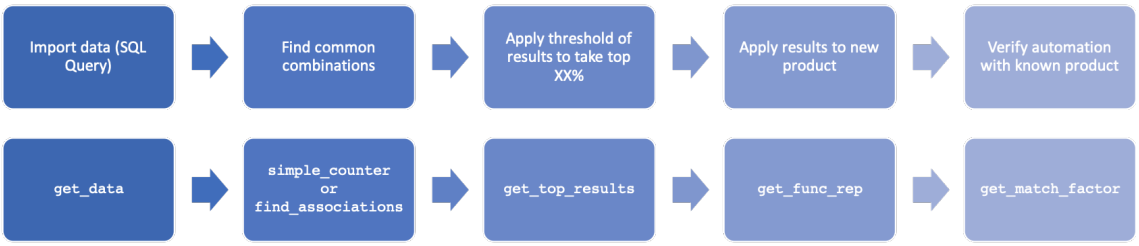

Figure 2: Flow of Software Functionality

3.1. Dependencies

This module interacts with data in UTF-8 .csv file format. It depends on the external package *Pandas* for data storage and analysis, as well as *apyori* to implement the Apriori algorithm for finding association rules.

3.2. Module: get_data

This module reads a .csv file and exports a *Pandas* data frame and a list of the contents. It is only used in conjunction with the association rules method of analysis. Using the probability method to find the associations does not require this module.

3.3. Module: simple_counter

This module implements the math described above in section 2.1 and shown in Table 2. The input is the file name and relative file path for the .csv file containing the data from the repository. The output is a dictionary of function and flow combinations sorted by prevalence. The key is the component and the value is a list with the function-flow and the probability. The format is: {component1: [[function-flow1, probability1], [function-flow2, probability2]]}.

3.4. Module: find_associations

This module uses the *apyori* package to implement the Apriori algorithm for finding the association rules between the items in the data set that was extracted with the *get_data* module. The inputs are the results of the *get_data* module, as well as optional thresholds for support, confidence, and lift. The algorithm will not find or output associations below these thresholds. Lower thresholds are used to give a richer output set, and higher thresholds are used to only find strong associations. The outputs of the module are two dictionaries: a dictionary of function and flow combinations sorted by confidence (the same dictionary output as the *simple_counter* package), and a dictionary that contains the full results of the association rules. The full association rule results are in the form: {component1: [[function-flow1, support1, confidence1, lift1], [function-flow2, support2, confidence2, lift2]]}.

3.5. Module: get_top_results

This module uses a threshold to take the top percentage of probability or confidence values (depending on the method used) of function-flow combinations for each component. It will sum the probability/confidence values of each function-flow combination until the threshold is reached and store the combinations that are within the threshold. This is the set of the top results of data mining for each component. The inputs are the results of either *simple_counter* or *find_associations* and a threshold with a default of 0.7. The output is

a dictionary of the same format as the results of *simple_counter* or *find_associations* but only containing the function-flow combinations that are within the threshold.

Using the example in Table 2, if the threshold was set to 75%, the module would return the results: {'battery': [['Store Electrical', 0.33], ['Supply Electrical', 0.33], [Transfer Electrical, 0.17]]}. The sum of the first two probabilities is 66%, so it adds the third to reach the threshold. This highlights a flaw in the method, which is that the third function-flow combination has an equivalent probability to the fourth combination, but they are sorted by which one the program saw first when sorting. An analysis could find how often this happens to determine if it is a large problem.

3.6. Module: get_func_rep

This module returns the functional representation of a product for a given set of constituent components. The inputs are the results of *get_top_results* and the name of the .csv file containing the components of a new product. It uses the output of *get_top_results* to refine the entire dictionary to only contain the components that are in the new product. The outputs are the new dictionary and a list of components that were in the input set but did not exist in the data mining results and for which it could not learn the functions and flows. It also has an optional Boolean input that toggles the results containing the confidence values for each function-flow combination. If only the functional representation is desired, set this to **False**.

3.7. Module: get_match_factor

This module compares the results of data mining to a product that is already in the repository for which the actual results are known and outputs a "match factor" based on how well the automation matched the actual combinations. This is used for testing different data sets used for mining and different thresholds for the top percentage of probabilities or confidence.

The inputs are the results of *get_top_results* for the data set used for mining as well as the results of *get_data* for a product in the repository that has been extracted to a .csv file. The module uses the known function-flow combinations for each component and compares them to the results learned from data mining. For each combination of component-function-flow, it labels it as either matched, overmatched, or unmatched. Matched means that the combination in the verification case was found in data mining. Overmatched means that a combination was found in data mining but it did not exist in the verification case. Unmatched means that a combination exists in the verification case but it was not found in the data mining. Each case is stored in a dictionary, the number of each case is tracked, and the match factor is determined by the following equation:

$$Match\ Factor = \frac{\#\ of Matched}{\#\ of Unmatched + \#\ of Overmatched} \tag{4}$$

The match factor is a ratio of correct to incorrect automation, and is a simple metric for validating the data mining results.

3.8. Module: write_results

This module writes the results to a .csv file. The format it shown in Table 5 below. The inputs are the dictionary to be written to a .csv file and the name of the file to write. This module is optional if a .csv file is desired at the end.

3.9. Usage

Autofunc is run from within a Python script that calls each module and passes along the information required, all starting with a .csv file of the data from the Design Repository. Examples are provided for various configurations, including using both methods of probability and association rules as well as automating functional representations from a list of components or generating a match factor from a verification case.

4. Results

To test the results of the software, I used a data set of 21 products, each containing the component *blade*, for data mining. I verified these results with another product with a blade, a jigsaw, and found the match factor for different threshold values. The plot in Figure 3 shows a maximum match factor at a threshold of 0.5, indicating this is the optimum threshold for this data set and this verification product.

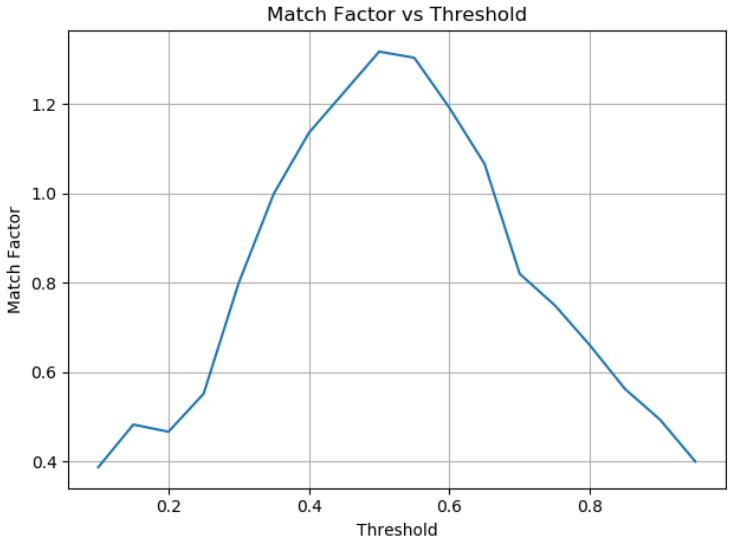

Figure 3: Match Factor vs Threshold for sample data set and verification case

Using the same data set and verification product, I automatically generated the functional representations of each of the components in the jigsaw. This is similar to the computation done for the match factor, but this assumes that only the names of the components for the jigsaw are known and automatically generates the functions and flows using the modules described previously. The list used for input components is shown in Table 4.

Component
electric switch
gear
electric motor
screw
electric wire
housing
blade
cap
guiders
spring
washer
cam
battery
cheese
bread

Table 4: Input Components for Data Mining

The full results are shown below in Table 5. The input components of *cheese* and *bread* are used to verify that the unmatched functionality works, and the results of that list include only those two false components.

Component	Function-Flow	Probability
electric switch	actuate electrical	0.1318
electric switch	secure solid	0.1008
electric switch	convert human energy	0.0930
electric switch	import human material	0.0930
electric switch	import human energy	0.0853
gear	guide solid	0.2662
gear	change mechanical	0.2086
gear	transfer mechanical	0.2086
electric motor	convert electrical	0.2609
electric motor	transfer electrical	0.1594
electric motor	guide solid	0.1449
screw	couple solid	0.9286
electric wire	transfer electrical	0.6324
housing	position solid	0.2476
housing	secure solid	0.1714
housing	guide solid	0.1571

Component	Function-Flow	Probability
blade	separate solid	0.1461
blade	export solid	0.1348
blade	import solid	0.1124
blade	transfer mechanical	0.0787
blade	export mechanical	0.0787
cap	position solid	0.2667
cap	secure solid	0.2667
guiders	guide solid	0.3333
guiders	position solid	0.2029
spring	position solid	0.3333
spring	guide solid	0.1538
spring	store mechanical	0.1538
washer	position solid	0.4000
washer	secure solid	0.3143
cam	transfer mechanical	0.4000
cam	position solid	0.4000
battery	store electrical	0.2500
battery	supply electrical	0.2500
battery	transfer electrical	0.2000

Table 5: Full Results of Data Mining with Threshold 0.5

Unmatched
cheese
bread

Table 6: Unmatched Components from Data Mining

5. Conclusions and Future Work

The *autofunc* package uses data mining techniques to find the most common combinations of function and flow for components in products within a data set from a design repository. This simplifies the process of generating functional representations and is a step toward automating functional modeling. This will allow designers to reap the benefits of rigorous early phase design practices without the tedious task of creating functional models, and also enable more thorough and consistent data entry for new products in design repositories.

Future work includes finding different data sets that have a certain percentage of similar components as the input product and optimizing a similarity based on the resulting match factor. For example, it may be found that that the match factor is the highest if the products in a data set have at least 50% of the same components as the product for which the functional model is being automated. This will be used in conjunction with the optimum threshold and a 3D surface can be used to find the best combination of threshold and percent similarity of components.

Additional future work includes graphical visualization of the results using a package like *PyGraphViz*. This will require many of the rules for ordering and visualization to be implemented in the automation results, but it will complete the task of fully automating functional models based on an input set of components.

References

[1] K. Otto, Product design: techniques in reverse engineering and new product development, 2003.

[2] M. C. Yang, Observations on concept generation and sketching in engineering design, Research in Engineering Design 20 (1) (2009) 1–11.

[3] D. G. Ullman, The mechanical design process, Vol. 2, McGraw-Hill New York, 1992.

[4] W. Beitz, G. Pahl, Engineering design: a systematic approach, MRS BULLETIN 71.

[5] R. B. Stone, K. L. Wood, Development of a Functional Basis for Design, Journal of Mechanical Design 122 (4) (2000) 359. doi:10.1115/1.1289637.
URL http://mechanicaldesign.asmedigitalcollection.asme.org/article.aspx?articleid=1446060

[6] J. Hirtz, R. Stone, D. McAdams, S. Szykman, K. Wood, A functional basis for engineering design: Reconciling and evolving previous efforts, Research in Engineering Design 13 (2002) 65–82.

[7] R. L. Nagel, J. P. Vucovich, R. B. Stone, D. A. McAdams, A Signal Grammar to Guide Functional Modeling of Electromechanical Products, Journal of Mechanical Design 130 (5) (2008) 051101. doi:10.1115/1.2885185.
URL http://mechanicaldesign.asmedigitalcollection.asme.org/article.aspx?articleid=1449717

[8] S. R. Miller, B. P. Bailey, Searching for Inspiration: An In-Depth Look at Designers Example Finding Practices, in: Volume 7: 2nd Biennial International Conference on Dynamics for Design; 26th International Conference on Design Theory and Methodology, ASME, 2014, p. V007T07A035. doi:10.1115/DETC2014-35450.
URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?doi=10.1115/DETC2014-35450

[9] S. Szykman, R. Sriram, C. Bochenek, J. Racz, J. Senfaute, Design repositories: engineering design's new knowledge base, IEEE Intelligent Systems 15 (3) (2000) 48–55. doi:10.1109/5254.846285.
URL http://ieeexplore.ieee.org/document/846285/

[10] M. R. Bohm, R. B. Stone, T. W. Simpson, E. D. Steva, Introduction of a data schema to support a design repository, Computer-Aided Design 40 (7) (2008) 801–811. doi:10.1016/J.CAD.2007.09.003.
URL https://www.sciencedirect.com/science/article/pii/S0010448507002151?via{%}3Dihub

[11] M. R. Bohm, J. P. Vucovich, R. B. Stone, An Open Source Application for Archiving Product Design Information, Volume 2: 27th Computers and Information in Engineering Conference, Parts A and B (2007) 253–263doi:10.1115/DETC2007-35401.
URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1603963

[12] C. R. Bryant, D. A. McAdams, R. B. Stone, J. Johnson, V. Rajagopalan, T. Kurtoglu, M. I. Campbell, Creation of Assembly Models to Support Automated Concept Generation, in: ASME 2005 International Design Engineering Technical Conferences, 2008, pp. 259–266. doi:10.1115/detc2005-85302.
URL https://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1572136

[13] M. R. Bohm, J. P. Vucovich, R. B. Stone, Using a Design Repository to Drive Concept Generation, Journal of Computing and Information Science in Engineeringdoi:10.1115/1.2830844.
URL http://www.asme.org/about-asme/terms-of-use

[14] T. Kurtoglu, M. I. Campbell, C. R. Bryant, R. B. Stone, D. A. McAdams, Deriving a Component Basis for Computational Functional Synthesis, ICED 05: 15th International Conference on Engineering Design: Engineering Design and the Global Economy (2005) 4061.
URL https://search.informit.com.au/documentSummary;dn=389890219001721;res=IELENG

[15] P. Sridharan, M. I. Campbell, A study on the grammatical construction of function structures, Artificial Intelligence for Engineering Design, Analysis and Manufacturing: AIEDAM 19 (3) (2005) 139–160. doi:10.1017/S0890060405050110.
URL http://alvarestech.com/temp/PDP2011/emc6605.ogliari.prof.ufsc.br/Restrito/Astudyonthegrammaticalconstructionoffunctionstructures.pdf

[16] M. R. Bohm, R. B. Stone, A Natural Language to Component Term Methodology: Towards a Form Based Concept Generation Tool, in: Volume 2: 29th Computers and Information in Engineering Conference, Parts A and B, ASME, 2009, pp. 1341–1350. doi:10.1115/DETC2009-86581.
URL http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1649365

[17] A. Mikes, Autofunc v0.1.1 (2019). doi:10.5281/zenodo.3243689.
URL https://github.com/AlexMikes/AutoFunc

[18] R. Agrawal, T. Imieliński, A. Swami, R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the 1993 ACM SIGMOD international conference on Management of data - SIGMOD '93, Vol. 22, ACM Press, New York, New York, USA, 1993, pp. 207–216. doi:10.1145/170035.170072.
URL http://portal.acm.org/citation.cfm?doid=170035.170072

[19] S. Orlando, P. Palmerini, R. Perego, Enhancing the Apriori algorithm for frequent set counting, Data Warehousing and Knowledge Discovery. 2114 (2001) 71–82. doi:10.1007/3-540-44801-2_8.
URL http://link.springer.com/10.1007/3-540-44801-2{_}8

[20] A. Lora-Michiels, C. Salinesi, R. Mazo, A Method based on Association Rules to Construct Product Line Model, 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMos) (2010) 50.
URL https://hal.archives-ouvertes.fr/hal-00707527/