CS393P: Program Synthesis Final Report

1. **Idea**

As proven in the Flash-fill paper, synthesis is a very powerful tool in the context of database management and manipulation.

My goal for this project is to create a synthesizer that can derive patterns from both an existing excel database and of future entries that may include typos or less consistent data. The produced mapping serves two purposes:

1. It can be used to automatically combine two databases together, which, like flash-fill, is very useful for people who have limited experience with Excel formulas or database languages like SQL.
2. The produced mapping can be applied to an arbitrary number of entries in the new database, saving a good amount of computation time by only computing the mapping a single time (versus once for each new entry).
3. **DSL**

The DSL used for the synthesis itself is very simple; it is an index mapping of columns in the new database to columns in the existing database.

The complexity lies in the underlying classification framework which the synthesizer works with. All of the following is written in DrRacket using Rosette.

A “category” struct is used to split columns of the existing database, and has the following properties:

* Items: The items in this column.
* Label: The label assigned to this column. It can either be a generic label or a pre-set label, as described below.

These categories are used to measure similarity between themselves and potential new entries.

**Generic Categories**

A generic category takes two forms- a string generic category and a numeric generic category. In either case, potential new entries are compared against existing category entries, and the generic category which is deemed most “similar” (as defined below) becomes the assigned category for that entry.

**Defining Generic Similarity**

Similarity is defined differently between numeric and string entries respectively:

* For numeric types, the existing entries are modeled as a distribution, and the z-score of the new entry is taken. The generic numeric category with the lowest z-score is assigned the new entry.
* For string types, a “similarity score” is taken based on the levenshtein distance relative to the length of both strings, for all strings in the category (see section 6, “Additional Information”, for details). The category that contains the string with the highest similarity score relative to the new entry is assigned.

**Pre-set Categories**

The idea behind a pre-set label is there will likely be strong underlying patterns for certain kinds of categories that may not be easily recognizable with the generic approach. One such example is with names; many names are distinct, so using a similarity score will be ineffective. Thus, a pre-defined set of common names is instead defined that will be compared to for determining membership.

The currently implemented Pre-set Categories are as follows:

First Name: A category representing common first names.  
Last Name: A category representing common last names.

However, no pre-set categories are currently assigned due to the relative scarcity of their occurrence in test databases. Furthermore, they only serve to marginally improve classification, and given time constraints they are ultimately superfluous to the main objective of this project. As discussed in section 5, the expansion of pre-set categories can be a topic of future work.

1. **Algorithmic Approach**

The classifier takes in a new row of data and greedily assigns each entry to a category in ascending index order (i.e., left to right), using the criteria defined in section 2 for numeric and string categories.

A dictionary of assignments is kept and updated, and checks are made to ensure no existing column is assigned to twice for any one entry.

The algorithm terminates when either:

* All columns have been assigned to (in the case that there are more columns in the new data than the existing data), or
* All new elements of a new row have been assigned.

The algorithm returns the assignment dictionary, which can then be used quickly assign remaining entries from the new data to the existing database.

1. **Results**

Performance results are shown below for a sample database found [here](https://www.contextures.com/tablesamples/sampledatafoodsales.zip). The “training” and “test” datasets were split in half from this data, each with roughly 120 rows of 8 columns each. Column entries were

Graphical user interface, text, application, email

Description automatically generated

Accuracy is measured as the number of correct classifications over the total number of mappings.

In either case, accuracy is quite high, demonstrating the generic approach taken here works well for not just toy data, but smaller cases of real-world data.

Furthermore, the performance results indicate a large amount of time can be saved by applying a single mapping to all future entries; the accelerated performance was over 75 times faster when doing so without compromising accuracy.

1. **Limitations and Future Work**

The greedy algorithmic approach works well, but it would likely improve accuracy at the cost of performance and additional memory usage if all columns were considered at once, then assignments were made in order of “how close” the new entries are to the existing entries. This would remove the bias for the left-most entries in the new entries.

Additionally, while the generic classification approach works fairly well here, I have found two notable vulnerabilities with it:

* Different generic string columns that share identical entries are difficult to distinguish between with the given approach, which will become a more likely problem with larger databases.
* Generic numeric categories with large standard deviations and relatively close means will be misclassified often. This can be partially remedied by taking a composite mapping of the first k entries of the database, where k is some fraction of the n new entries and a hyperparameter of the classifier. This will result in a marginal loss to performance in exchange for a likely increase to classification accuracy.

The further implementation and expansion of pre-set categories could prove to marginally improve classification. Some that come to mind are phone numbers, addresses, and emails.

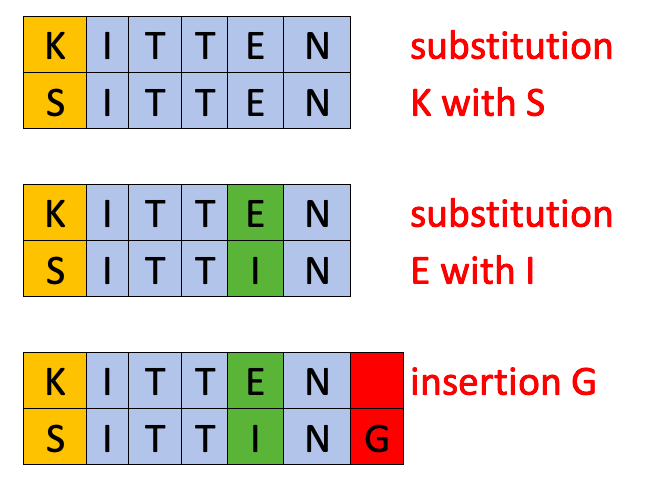
Another interesting line of future work could be the splitting/merging of column entries. Say, for instance, the existing data has the column “<Firstname>, <Lastname>”, whereas the new data has two separate columns with first names and last names. The ability to consider merging columns together in this fashion could cover a number of interesting cases, and based on existing performance metrics it wouldn’t be prohibitively expensive to do so.

Finally, because this is a prototype implementation in DrRacket, use of the application as-is would be limited as most commercial databases don’t simply use excel spreadsheets, nor would they likely fit in one. The application of this functionality to such databases would allow for testing on larger and more intricate databases and ultimately give further insight into the pros and cons of the approach.

1. **Additional Information**

**Levenshtein Distance & Similarity Score**

Levenshtein distance is like Hamming distance for strings; it measures the number of character substitutions, insertions, and deletions required to turn one string into another. The below example has a Levenshtein distance of 3:



The similarity score is defined as follows:

Similarity(s1, s2) = 1 – (levenshtein(s1, s2) / (len(s1) + len(s2))

Where a similarity of 1 means the strings are identical.

Because similarity is relative to the length of the string as well as levenshtein distance, this gives a built-in tolerance for typos in longer entries, which helps classify longer entries that may be more prone to inconsistent data.