Deep Feature Synthesis

A Data Science Machine for Relational Databases For Caché

Ryan St.Pierre

Sarat Vysyaraju

Deep Feature Synthesis

Purpose

Data science involves analyzing and deriving insights from large sets of data. Such a process requires a data science team to invest a lot of time and resources in order to draw meaningful conclusions from the data. We have designed an end to end product for data residing on Caché by adapting the Deep Feature Synthesis (DFS) algorithm [1]. This product automates the process, by aiding data science teams in the discovery of hidden links, generation of meaningful features from the data, and creation of accurate predictive models in a more time efficient manner.

The rest of the paper is organized as follows. We begin by describing the DFS algorithm. Then we state the problems and solutions we encountered while integrating DFS to Caché. Next, we explain the graph optimizations that can be applied to a relational structure to decrease the complexity of the DFS algorithm. Following this, we explain the steps taken in the machine learning process to produce predictive models for the data. Lastly, we outline the possibility for future work regarding the improvement of the tool.

Deep Feature Synthesis

The Deep Feature Synthesis algorithm automatically aggregates features in a relational database structure [1]. To do such, the algorithm recursively traverses along the relationships in the data, applying mathematical functions over the features in this traversal and appending the result to a base table. The final output is an expanded base table that represents a much larger portion of the relational data.

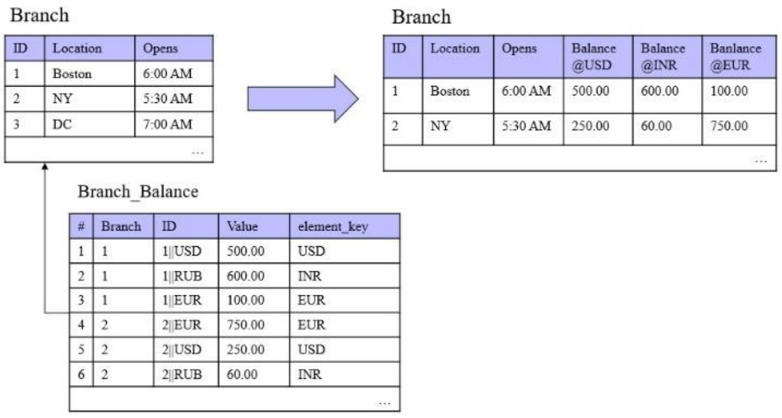
Integration Problems Encountered

The DFS algorithm is designed to work with structures compatible with MySQL convention. However, there are several Caché Object Script datatypes that are handled in a way whose storage differs from this convention. We designed several abstractions to handle these cases, each described below.

Array Denormalization

In Caché using an array in a class definition generates two tables, one for the class itself (parent table) and the other holding the information of the array, which contains a key-value pair and link to the parent table.

However, the problem with the Caché array tables is that they are immutable, i.e., additional features (columns) cannot be added. In our implementation of the DFS algorithm we dynamically append features into tables in the relational structure. This approach makes denormalizing the array structure by transferring the array information into the editable parent table a necessity. In addition, this solution reduces the depth of our structure, improving our overall efficiency.



Branch

ID	Location	Opens
1	Boston	6:00 AM
2	NY	5:30 AM
3	DC	7:00 AM
		...

Branch

ID	Location	Opens	Balance @USD	Balance @INR	Banlance @EUR
1	Boston	6:00 AM	500.00	600.00	100.00
2	NY	5:30 AM	250.00	60.00	750.00
					...

Branch_Balance

#	Branch	ID	Value	element_key
1	1	1||USD	500.00	USD
2	1	1||RUB	600.00	INR
3	1	1||EUR	100.00	EUR
4	2	2||EUR	750.00	EUR
5	2	2||USD	250.00	USD
6	2	2||RUB	60.00	INR
				...

Figure 1: Denormalization of array storage necessary for dynamic feature expansion

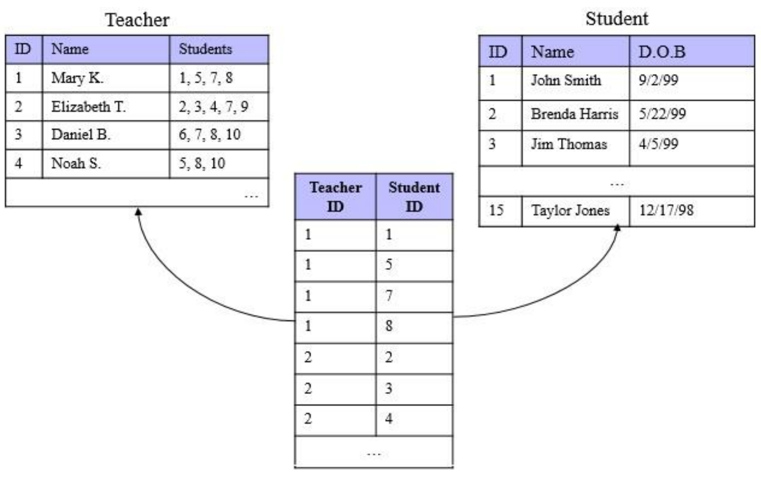
We illustrate the denormalization process in Figure 1, where a *Branch* has an array of *Balances*. In our approach, we append a column to the parent table for each unique key and transfer the values from the array table into the corresponding key column in the parent table. Thus, we are able to denormalize the array structure without losing any information.

List Handling

In MySQL each column of a given row contains one value (or foreign key). In Caché tables this may not be the case, as multiple values or object references can be saved as a list in one entry. We identified two main types of lists in Caché of importance to us: (i) lists of datatypes and (ii) lists of object references. To handle lists of datatypes we compute aggregation functions over lists of numeric types to generate meaningful features about the list itself. For example, in a Student table with a list of grades, *e-feature* expansion generates the sum, mean, minimum, and maximum of the grades for each student.

List of Object References

It is clear in Figure 2 below, that a list of object references represents a many to many relationship. In this example each teacher has reference to multiple students and it is also possible for students to have multiple teachers (such as student 5).



Teacher

ID	Name	Students
1	Mary K.	1, 5, 7, 8
2	Elizabeth T.	2, 3, 4, 7, 9
3	Daniel B.	6, 7, 8, 10
4	Noah S.	5, 8, 10
		…

Teacher ID	Student ID
1	1
1	5
1	7
1	8
2	2
2	3
2	4
…	

Student

ID	Name	D.O.B
1	John Smith	9/2/99
2	Brenda Harris	5/22/99
3	Jim Thomas	4/5/99
	…	
15	Taylor Jones	12/17/98

Figure 2: Abstraction for many to many relationships to allow the proper flow of information while avoiding an infinite loop.

However, if the relationship between teacher and student is directly handled as a many to many relationship the DFS algorithm enters an infinite traversal between these two entities. To solve this problem, we insert a buffer table into the structure which maps the IDs of the two respective tables. In addition, we established one to many relationships between the two original tables and the buffer table in order to preserve the original integrity of the relationship. This abstraction allows information and aggregations to travel in both directions without an infinite loop.

Graph Optimizations

The complexity of the DFS algorithm is exponential with respect to depth. Thus, it is crucial to decrease the depth of the relational structure whenever possible. We make one such optimization by treating one to many relationships of type unique as one to one relationships, and thus allowing us to denormalize the structure. Below, in Figure 3 we give an example of a portion of a possible relational structure for a company, where the black arrows represent one to one relationships and the red arrows represent one to many relationships.

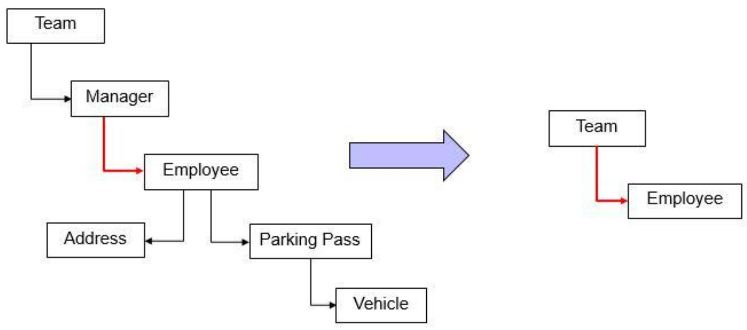


Figure 3: Denormalizing of one to one relationships to decrease DFS complexity while relational integrity.

In an object oriented approach one to many relationships of type unique may be used to normalize the structure for organizational purposes or to preserve clarity of information. However, taking such an approach unnecessarily increases the depth of the structure, and consequentially the complexity of our algorithm. In a one to one relationship the property of the child can be represented in the parent. Thus, as a preprocessing step to our algorithm we move the information of the child into the parent of one to one relationship, ensuring that we preserve the relational integrity of the structure. By denormalizing the structure in this way we decrease the complexity of our structure and prevent our algorithm from doing redundant aggregations over one to one relationships.

Machine Learning

The machine learning (ML) process attempts to build a predictive model for a chosen target column in the target table from the DFS algorithm. We begin by pre-processing the data which includes: numerical encoding of categorical data, removing rows/entries with incomplete data, normalizing the scale of the data (between -1 to 1), and sharding the data into multiple tables. The process of sharding entails distributing the data into two or more shards (slave machines). The benefit of sharding the resulting target table is two-fold: primarily for parallelization of complex computations and secondly for storing the data across multiple devices, which may be necessary if the data does not fit on one machine.

We also use a Spark connector which manages the communication with the Caché shard/slave machines during the machine learning process. The Spark connector is preferable to a traditional JDBC connection because it allows the Spark master to run computations hosted on the shards themselves.

ML Pipeline

Once the preprocessing is complete the data is sent through a pipeline. The pipeline is designed, utilizing the available *Spark ML* and *Spark MLlib* libraries, with three main stages: (i) feature selection, (ii) clustering, and (iii) predictive modeling.

Feature Selection

The DFS algorithm adds many columns to the target table. Our first step in the pipeline is to identify which of these columns best correlate the target column. We choose a percentage of the top ranking columns to carry through the rest of the machine learning pipeline, reducing the complexity of the following tasks.

Our current implementation of feature selection uses Singular Value Decomposition (SVD) to describe each column using a certain number of components, followed by the Fisher's Statistical test to rank these columns with respect to the target column.

Clustering

We make no assumption that the data in the target table is uniform. To identify natural clusters in the data we currently use K-Means clustering method.

Predictive Modeling

Lastly, for each of the clusters we train a predictive model that predicts the value of the target column based on the values of the other features. Our current implementation uses a Decision Tree Random Forest as the predictive model.

Hyper-parameter optimization

Each stage in the pipeline has one or more parameters. Such parameters include: the number of dimensions to use in SVD, the percentage of features retained after ranking, the number of clusters to use, the maximum depth of the decision trees, and the number of decision trees in the random forest. We attempt to tune these parameters by reducing the total error over the entirety of the pipeline, using a greedy random search heuristic.

Future Work

We have designed and implemented a tool that successfully derives insight from a set of relational tables. However, we feel there are several opportunities to improve the quality and robustness of the tool.

First, we would like to incorporate more feature expansion types. Currently we are supporting 6 SQL aggregation types: SUM, MIN, MAX, STD, COUNT, and AVG. We would like to provide the user with more options allowing them to select aggregations such as probability density function or cumulative distribution function and let them see how it changes the accuracy of the final model.

Furthermore, we believe that the accuracy of our final model can be improved by adding greater functionality to the ML pipeline. The machine learning pipeline is designed to have three modular steps, the implementation of which can be easily changed. For example, we currently support K-Means clustering for our clustering method. In the future it would be best to include other clustering methods, such as Gaussian Mixture Model (GMM). This would give the user the option to choose which techniques to use depending on the type of data being analyzed.

One necessary component for the future success of our product is an abstraction that allows entities to have more than 1000 properties, which is a hard limit for Caché class definitions. We use class definitions as containers for feature aggregation and use dynamic SQL queries to access and manipulate these features. Given data with a large amount of starting features it would be possible for the DFS algorithm to eclipse this 1000 property limit. Thus, an abstraction needs to be built in which SQL queries can be computed on an entity with more than 1000 features, represented on the backend with more than one class definition.

Lastly, we would like to build a front-end interface, most likely in the Management Portal, for the user to follow and influence the end to end process. This would include setting the input tables, target column, and machine learning pipeline parameters. In addition, the UI would give the user the ability to toggle the input values into the created model and graphically observe how it influences the prediction in real time.

References

[1] J. M. Kanter and K. Veeramachaneni, "Deep Feature Synthesis: Towards Automating Data Science Endeavors," in *IEEE International Conference on Data Science and Advanced Analytics* , Paris, 2015.