Automated Feature Engineering for Event Prediction

CPSC 490 Project Report

Felipe Pires

Advised by Benedict Brown

April 30, 2019
Abstract

We developed a program that automates much of the repetitive programming involved in the feature engineering for certain problems related to event prediction. Nowadays, with the help of frameworks such as XGBoost and LightGBM, data scientists are able to easily create machine learning models to predict the future, if only they are able to, given a dataset of past events, generate the training features that appropriately expose patterns to the model being trained. Much of this feature engineering requires the data scientist to write repetitive, error-prone code that looks very similar across completely different applications. Examples of this repetitive work include the generation of so-called lag features and window features. Both of these are popular techniques for capturing historical trends in datasets but require a lot of boilerplate code that could, but rarely ever is, automated away.

With this opportunity in mind, we set out to create a program capable of assembling lag, window and other kinds of features that are useful to create models that predict the future. First, we devised a grammar for describing these features, and then a program that takes instructions in this grammar and puts together the desired columns. Our solution has proven powerful and versatile, and we are finding new uses for it each day. We were able, for instance, to completely automate a couple of top-ranking solutions in the data science competition website Kaggle.
Motivation

The jumping-off point for this project was the Kaggle competition "Predict Future Sales", created by the website’s team in partnership with the Russian company 1C.\(^1\) The challenge was to create a machine learning model to predict how much of a product would be sold in a store, one month in the future. Competitors were provided a sizeable dataset for training, containing sales data of about 22,100 products over 60 different stores in Russia. Information was provided about a total of almost three million transactions, corresponding to sales between January 2013 and October 2015.

Figure 1. Schema of the data provided by 1C on Kaggle

\(^1\) https://www.kaggle.com/c/competitive-data-science-predict-future-sales
The schema of the provided data can be seen in Figure 1. The dataset was normalized across four different tables, without redundancies. Each row in the `day_sales` table contained the information of how much (item.ctn_day) of a certain product (product_id) was sold at a store (shop_id) each day of the year (date).

The first simple feature that competing data scientists had to engineer had to do with how the format of the original data compares to the format of the prediction in the challenge. The rows in `day_sales` inform of how much was sold daily. The competition, however, was to find how much would be sold monthly. For this purpose, training data would have to associate months, shops and products to the total sold that month (of that product, in that shop), like the following table:

<table>
<thead>
<tr>
<th>Month</th>
<th>Shop features</th>
<th>Product features</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>City</td>
<td>(others)</td>
</tr>
<tr>
<td>Dec 2018</td>
<td>Target</td>
<td>New York</td>
<td></td>
</tr>
<tr>
<td>Jan 2019</td>
<td>Target</td>
<td>New York</td>
<td></td>
</tr>
<tr>
<td>Feb 2019</td>
<td>Target</td>
<td>New York</td>
<td></td>
</tr>
</tbody>
</table>

This aggregation of daily data into monthly data, if only needed once, could be best accomplished by writing a few lines of Python (especially if using a library for data manipulation, such as the popular pandas). However, to train any reasonable model, aggregations like these need to be programmed several times, sometimes once per feature.
Transformations like these are one of the factors that caused most top-ranking solutions to the sales forecasting challenge to need several hundreds of lines of code. Data scientists are used to doing them "by hand", causing feature engineering to be a tedious and error-prone task at times. What if there was a way to automate these common data transformations to a third-party library instead?

**Defining a Grammar**

Consider the following expression

\[
\text{Day\_sales.\text{SUM}(item\_cnt\_day|\text{CMONTH}(date), shop, item)}
\]

It describes the transformation of daily sales data into monthly sales, by summing the `item\_cnt\_day` field for each month, shop and item. The `Day\_sales` term that begins the expression indicates all column names (i.e. `item\_cnt\_day`, `date`, `shop`, `item`) are with respect to the table `day\_sales`. The vertical sign, or "pipe", inside the parenthesis indicates that the aggregation should group the result of `SUM` over all combinations of `CMONTH(date)`, `shop` and `item`.

In this project, we drew inspiration from the syntax used in programming and statistics to create a grammar that could express elements native to the world of data science and feature engineering.

- Tables and fields were modeled after the traditional Object-oriented syntax for classes and objects. The following points to the `name` field of the `category` field of each element of the `Items` table.

  "Items.category.name"
- Element-wise transformations of columns were modeled after the traditional syntax for function in mathematics. The following returns a boolean value for each item, depending on whether the name field of the item is null.

"ISNULL(item.name)"

- The syntax for data frame indices, or “group by’s”, was inspired by the use of the pipe character in statistics. The following represents a column indexed by the columns shop and item, containing the average item_cnt_day for each combination of these values.

"AVG(item_cnt_day|shop,item)"

Put together, these three elements can express an infinite (though not necessarily all) number of historical features that may be useful for training a machine learning model, including the aforementioned lag features. Consider, for instance, the following expression:

\[ \text{FWD(Day_sales.SUM(item_cnt_day|CMONTH(date),shop,item),1)} \]

Like SUM, the FWD function is part of the standard library of transformations standardized in this project. It receives as the first argument an indexed data frame (ie. Day_sales.SUM(...)) containing a date-bound index (ie. CMONTH(date)) and advances the count of the date index by a certain value, passed in as the second argument (ie. 1).
Figure X. Example JSON describing input tables and composed features to generate

**Program Description**

Our solution was a program written in Python that takes a set of datasets and a list of feature engineering instructions, as described above, and outputs a new dataset with the desired features. Two main parts are involved in making this happen: the parser and the assembler. Our engine approaches one instruction at a time, in a loop (see `src/index.py`), first parsing it, and then sending the result to the assembler, which puts together the one column corresponding to the parsed instruction.

**Grammar Parser**

To create a parser for our instructions, we opted to use the *lark* library for Python, instead of writing a parser from scratch using string manipulation. Lark allowed us to
define a grammar for our language using a syntax very similar to EBNF. After a handful of iterations, we found a clear way of describing all the elements we wanted for our language using a concise set of rules (see Figure X).

Figure X. Code excerpt from src/parser/grammar.lark

The parser was almost done once the grammar was described in lark’s syntax. Only an extra bit of processing was required to transform the default tree-like output of the library into an intermediary structure of nested dictionaries that could be best understood by the next steps of the program (see src/parser/parse.py).

Assembler

The assembler is the second half of the engine pipeline. It takes one parsed instruction and a set of input datasets, and it outputs the column described. It does
so by recursively calling function `assemble_column()` (see file `src/assembler/assembler.py`).

**Conclusion**

The biggest challenge of this project was to come up with an appropriate grammar for representing