Don't be so One-Dimensional: How to Engineer Multi-Dimensional High Cardinality Categorical Inputs for Machine Learning

Aleksandar Nikolic, Georgia-Pacific LLC

ABSTRACT

Knowing how to process categorical inputs is a key skill that every data scientist must know. While a wide array of techniques is available to prepare categorical data for machine learning, they do not address all the problems that a data scientist could face. One such problem arises from working with transactional datasets where there could be multiple observations per level of interest. It is possible for a categorical input to not only have a high number of unique values across the entire dataset, but also multiple unique values within the level of interest. Since traditional categorical feature engineering techniques do not address the multi-dimensional aspect of this problem, this paper will outline a novel approach that does. This paper demonstrates the solution using base SAS® code, and SAS® Cloud Analytic Services (CAS) procedures found in the SAS® Viya™ 3.5 platform for machine learning model comparisons.

The SAS code and datasets from the demonstrations can be downloaded from
https://github.com/nikolicxa/multi-dimensional-high-cardinality.

INTRODUCTION

Most data scientists will eventually encounter transactional data which contains more than one observation per *level of interest*. A *level of interest* is the unit that you aggregate the data to before starting the analysis. A common example is a retail sales dataset, where the levels of interest could be either the transaction ID (which could contain multiple unique items purchased) or customer ID (which could contain multiple unique items purchased spanning multiple transactions). While there are many effective methods of summarizing numeric features to the level of interest (such as calculating the sum, minimum, maximum, mean, or range), it is not as straight forward with categorical inputs. One problem is that you would have to convert them to a numeric representation before using them to train certain machine learning models. Another is *high cardinality*, which means that the input contains a large number of distinct values [1]. When working with transactional data, one is likely to encounter two flavors of this problem. The first is where there are many distinct values spanning the entire input (categorized as *inter-level high cardinality*). The second is where there are individual levels of interest with multiple observations containing distinct values (categorized as *intra-level high cardinality*). If both are present, then it is categorized as having *multi-dimensional high cardinality*.

The objective of this paper is to introduce a new way of creating features derived from inputs with multi-dimensional high cardinality, with the aim of using them to train machine learning models. It first starts with a summary of multi-dimensional high cardinality illustrated with an example. Next, a review of some traditional categorical feature engineering techniques, and why they are not suited for predictors with multi-dimensional high cardinality. After that is an exposition of the new method, which includes a worked example. The subsequent section reviews how to perform k-fold target encoding. Following that, a worked example with SAS code that combines the concepts from the previous two sections. The penultimate section evaluates the possible benefits of including these new inputs in supervised machine learning models. The last section explores a few hazards that come with this method, along with suggestions on how to attenuate them.

SECTION 1. AN EXAMPLE OF A DATASET WITH MULTI-DIMENSIONAL HIGH CARDINALITY

To illustrate the concept of multi-dimensional high cardinality, and other ideas explored later in this paper, a small sample dataset from a fictitious office supply store called Binders will be used (along with other datasets introduced later). The Binders dataset contains a small sample of credit card transactions with three columns, *Transaction_ID*, *Product*, and *Fraud*. The *Transaction_ID* column contains the ID number for each sales transaction and is the level of interest. The *Product* column contains the descriptions of the assorted items that customers purchased. The *Fraud* column (also referred to as the target column) is an indicator of whether the credit card used for the transaction was fraudulent (a value of 1=fraudulent, 0=not fraudulent). Since the company loses money on each transaction paid with a fraudulent credit card, the data scientists at Binders want to know if they can somehow use the *Product* input to train a new machine learning model that will flag risky transactions in the future.

Is the Binders dataset in its current form in Figure 1 ready for analysis? If you look, the first thing you might notice is that there are multiple observations for every transaction ID, and they have unique *Product* values, which is a sign of intra-level high cardinality. Also, this dataset contains a large amount of distinct *Product* values (10) relative to the number of observations (15), which is an indication of inter-level high cardinality. Given that this dataset presents both conditions of multi-dimensional high cardinality, you will need to perform categorical feature engineering before training machine learning models.

Figure 1. Binders Dataset

SECTION 2. A REVIEW OF TRADITIONAL METHODS OF CATEGORICAL FEATURE ENGINEERING

To train a variety of supervised machine learning models, all the inputs would need to contain numeric representations of the data since most models cannot process categorical features (with exceptions including tree-based and naïve Bayes models) [2]. Existing methods of converting categorical inputs into numeric inputs fall under one of two taxonomies, *target-agnostic* and *target-based* techniques [3]. *Target-agnostic* methods do not use information from the target column while *target-based* techniques try to incorporate information about the target values associated with a given level. While there are methods from both taxonomies that can reduce the effects from either pillar of multi-dimensional high cardinality, there are not many that effectively mitigate both at the same time.

One of the most popular target-agnostic methods of converting categorical to numeric inputs is *one-hot encoding*. This method creates a dichotomous feature (meaning that the input only contains one of the following two values, 1 or 0) for every distinct value found in the original input [4]. All observations containing the specific column value get a value of 1 while the rest get a 0. The number of new columns created would be the same number of unique values found in the original input. While this method preserves most of the information contained in the original input, it could lead to a massive increase in the dimensionality of the entire dataset. If one were to apply one-hot encoding to the *Raw_Categorical_Input_4* column (found in the `SESUGDTA.raw_non_sum_file`), referred to in Table 1, it would result in the creation of 8,146 new columns! While effective for low cardinality inputs (containing <10 unique values), you should avoid this method for inputs like *Raw_Categorical_Input_4*.

Input Name	Cardinality of Entire Dataset	Percent of Transaction IDs with Multiple Records	Max Cardinality for Transaction IDs with Multiple Records
Raw_Categorical_Input_4	8,146	4.3%	7

Table 1. Cardinality Statistics for Raw_Categorical_Input_4

One target-based method of encoding is to consolidate the input by using a decision tree (also known as leaf encoding), done by training a decision tree using the categorical column as the sole modeling input [5]. The split search algorithm of the tree groups the input values that have similar target outcome proportions. You can either use the assigned leaf IDs to replace the levels of the input or create new dichotomous variables for each leaf denoting the leaf assignment. This is an effective way to consolidate inputs with moderate inter-level high cardinality but would not be ideal for the *Raw_Categorical_Input_4* input since it does not address the problem of intra-level high cardinality.

One target-agnostic method that that could overcome the problem of intra-level high cardinality is to create a new column that contains a concatenated string of all the distinct values grouped to the level of interest. One example from the Binders dataset is for the observations where *Transaction_ID* = "1500". If you wanted to summarize the data to a single observation, then the value of the new input would be "Chair_Notepad_Envelope". This method theoretically solves the problem of multi-dimensional high cardinality. However, this makes the inter-level high cardinality worse due to the addition of new distinct values created by the concatenated strings. On top of that, this final column is still a categorical input, and you would need to perform an additional step to convert the new column into a numeric feature.

SECTION 3. RATIONALE AND METHODOLOGY OF NEW FEATURE ENGINEERING METHOD

Converting a categorical feature with multi-dimensional high cardinality to a numeric feature will require a different approach rooted from a target-based method called *target encoding*. *Target encoding* (also known as *mean encoding*) is a method of mapping each unique categorical input value to the mean target value [6]. For a classification problem, you would calculate this by taking the sum of the target column and dividing it by the number of occurrences for every unique categorical value. For example, in the Binders dataset, there are four occurrences of the value "Desk" (Figure 2). Of the four occurrences, three have a *Fraud* value of 1. Therefore, the target encoded value for "Desk" would be .75 (3÷4). However, since the example dataset has multi-dimensional high cardinality, this section will introduce a variation of target encoding that accommodates it called *target representation encoding*.

Transaction_ID	Product	Fraud
1000	Ruler	1
1000	Binder	1
1200	Desk	0
1200	Stapler	0
1400	Desk	1
1400	Notepad	1
1500	Chair	0
1500	Notepad	0
1500	Envelope	0
2000	Desk	1
2000	Desk	1
2000	Pencil	1
2000	Ruler	1
2000	Pen	1
2000	Highlighter	1

Figure 2.

Target representation encoding is based on target representation, calculated by taking the sum of the target column grouped by the unique values of the categorical input, and then dividing it by the total of the entire target column. In Figure 2, for the value of "Desk", it has three occurrences where *Fraud* equals 1. You would then divide that by the sum of the entire *Fraud* column, which equals to ten. The target representation for "Desk" is .3 (3÷10). In other words, the input value of "Desk" represents 30% of the occurrences of fraud (Figure 3).

Product	Sum of Fraud	Target Representation
Desk	3	30%
Ruler	2	20%
Pencil	1	10%
Pen	1	10%
Notepad	1	10%
Highlighter	1	10%
Binder	1	10%
Stapler	0	0%
Notepad	0	0%
Envelope	0	0%
Desk	0	0%
Chair	0	0%
Totals	**10**	**100%**

Figure 3.

However, there is a minor problem with this calculation. The value of "Desk" appears twice for the *Transaction_ID* value of "2000". This is an instance of "double counting", which can inflate the target representation value. To solve this problem, you simply de-duplicate observations at the *Transaction_ID* and *Product* levels (Figure 4) and re-calculate the target representation column.

Transaction_ID	Product	Fraud
1000	Ruler	1
1000	Binder	1
1200	Desk	0
1200	Stapler	0
1400	Desk	1
1400	Notepad	1
1500	Chair	0
1500	Notepad	0
1500	Envelope	0
2000	Desk	1
2000	Pencil	1
2000	Ruler	1
2000	Pen	1
2000	Highlighter	1

Figure 4.

After de-duplication, the updated target representation value for "Desk" is now .22 (2÷9), shown in Figure 5.

Product	Sum of Fraud	Target Representation
Desk	2	22%
Ruler	2	22%
Pencil	1	11%
Pen	1	11%
Notepad	1	11%
Highlighter	1	11%
Binder	1	11%
Stapler	0	0%
Notepad	0	0%
Envelope	0	0%
Desk	0	0%
Chair	0	0%
Totals	**9**	**100%**

Figure 5.

Why go with target representation encoding instead of target encoding? The answer has to do with the weight that both calculations assign, especially for rarely occurring values of the categorical input. For example, "Pen" would be assigned a value of 1 and "Desk" .66 using target encoding (based on the dataset in Figure 4). However, the value of "Pen" occurs once, and has a sum target value of 1, while "Desk" occurs three times with a sum target of 2. One might think that "Desk" should carry a higher weight since the sum target value is twice the value of "Pen" and occurs more often in the dataset. The target representation encoding values for "Pen" (.11) and "Desk" (.22) are more intuitive given the higher frequencies of the latter input value.

To calculate new features using target representation encoding on a raw transactional dataset, you would need to follow the next five steps:

STEP 1: CONCATENATE THE TRANSACTION ID AND CATEGORICAL INPUT COLUMNS

Create a new column (*Transaction_ID_Product*) consisting of a concatenated string that combines the value of the *Transaction_ID* and *Product* columns (Figure 6). The level of interest is *Transaction_ID.*

Transaction_ID	Product	Fraud	Transaction_ID_Product
1000	Ruler	1	1000_Ruler
1000	Binder	1	1000_Binder
1200	Desk	0	1200_Desk
1200	Stapler	0	1200_Stapler
1400	Desk	1	1400_Desk
1400	Notepad	1	1400_Notepad
1500	Chair	0	1500_Chair
1500	Notepad	0	1500_Notepad
1500	Envelope	0	1500_Envelope
2000	Desk	1	2000_Desk
2000	Desk	1	2000_Desk
2000	Pencil	1	2000_Pencil
2000	Ruler	1	2000_Ruler
2000	Pen	1	2000_Pen
2000	Highlighter	1	2000_Highlighter

Figure 6. Resultant Table After Step 1

STEP 2: REMOVE DUPLICATE TRANSACTION_ID_PRODUCT OBSERVATIONS AND CREATE A TARGET HIT INDICATOR COLUMN

Next, remove all observations that have duplicate values of *Transaction_ID_Product*. Since the target representation column only consists of observations that had fraud associated with it, drop all records that have a value of 0 in the *Fraud* column and group by *Transaction_ID_Product*. Create a new column (*Raw_Product_TH*), which is a dichotomous variable that indicates whether there was a target value of 1 associated with the original *Product* column value (Figure 7).

Product	Raw_Product_TH	Transaction_ID_Product	Transaction_ID
Desk	1	1400_Desk	1400
Desk	1	2000_Desk	2000
Ruler	1	1000_Ruler	1000
Ruler	1	2000_Ruler	2000
Pencil	1	2000_Pencil	2000
Pen	1	2000_Pen	2000
Highlighter	1	2000_Highlighter	2000
Notepad	1	1400_Notepad	1400
Binder	1	1000_Binder	1000

Figure 7. Resultant Table After Step 2

STEP 3: CREATE TARGET REPRESENTATION COLUMN

Create a new column, *Tot_Raw_Product_TH,* by taking the sum of the *Raw_Product_TH* column grouped by the original *Product* values. Create the raw target representation column, *Raw_Product_Target_Rep*, by dividing the observation value of the *Tot_Raw_Product_TH* by the total sum of the same column (Figure 8). The sum of the *Raw_Product_Target_Rep* column should equal to 100%. Please note the "Totals" row in Figure 8 included for clarity will not be in the calculations in the SAS code in the next section.

Product	Tot_Raw_Product_TH	Raw_Product_Target_Rep
Desk	2	22%
Ruler	2	22%
Pencil	1	11%
Pen	1	11%
Notepad	1	11%
Highlighter	1	11%
Binder	1	11%
Totals	**9**	**100%**

Figure 8. Resultant Table After Step 3.

STEP 4: MAP RAW TARGET REPRESENTATION COLUMN BACK TO DATASET CREATED IN STEP 2

Perform a left join to the table created in Step 3 to the table created in Step 2 using the *Product* column as the key. Create a new column called *Raw_Product_TH_Ind* and assign it a value of 1 (Figure 9). This new column indicates that the *Product* had a *Raw_Product_Target_Rep* greater than 0.

Transaction_ID	Product	Raw_Product_TH_Ind	Raw_Product_Target_Rep
1000	Ruler	1	22%
1000	Binder	1	11%
1400	Desk	1	22%
1400	Notepad	1	11%
2000	Desk	1	22%
2000	Ruler	1	22%
2000	Pencil	1	11%
2000	Pen	1	11%
2000	Highlighter	1	11%

Figure 9. Resultant Table After Step 4.

STEP 5: CREATE NEW FEATURES SUMMARIZED TO THE UNIQUE LEVEL OF INTEREST

The first column, *Product_Tot_TH*, is a sum of the *Raw_Product_TH_Ind* column. *Product_Sum_Target_Rep* is the sum of the *Raw_Product_Target_Rep* column. *Product_Enc_Prod* is a product of the *Product_Tot_TH* and *Product_Sum_Target_Rep* columns (e.g., for *Transaction_ID* = "1000", the calculation is 2 * .33 = .66). This column assumes that transactions with multiple unique products associated with the target will have a stronger relationship with the target column than single observation transactions. Group these three new columns by the level of interest (*Transaction_ID*).

Transaction_ID	Product_Tot_TH	Product_Sum_Target_Rep	Product_Enc_Prod
1000	2	33%	0.66
1400	2	33%	0.66
2000	5	78%	3.90

Figure 10. Resultant Table After Step 5.

SECTION 4. OVERVIEW OF K-FOLD TARGET ENCODING

Since the newly engineered inputs are based on the values of the target column, they can fall prey to data leakage, which is when "information is revealed to the model that gives it an unrealistic advantage to make better predictions," with the information being the *Target* column [7]. Data leakage could lead to overfitting, which is when a model fits very well to the train data but performs poorly when predicting new samples [8]. Performing cross-fold target encoding (or *k-fold target encoding*) on the training dataset can head off the effects of data leakage [9]. *K-fold target encoding* starts by dividing the training data into k folds (for this example, k=5), stratified by the target column. Each fold is a subset of the training data containing ~20% of the unique levels of interest, and ~20% of all transactions with a target column value of 1.

This demonstration will calculate the *Product_Sum_Target_Rep* column on a version of the Binders sample dataset (Figure 11). It contains 25 observations grouped to the *Transaction_ID_Product* level, along with their fold assignments (on the y-axis). This example will only use two *Product* values ("Ruler" and "Binder"). It also contains the *Raw_Product_TH* column indicating whether the transaction had a *Fraud* value of 1.

	Transaction_ID_Product	Product	Raw_Product_TH
Fold-1	500_Ruler	Ruler	1
	1000_Ruler	Ruler	1
	1000_Binder	Binder	1
	1025_Ruler	Ruler	0
	1150_Binder	Binder	0
Fold-2	1155_Binder	Binder	1
	1175_Ruler	Ruler	0
	1180_Binder	Binder	0
	1190_Ruler	Ruler	0
	1225_Binder	Binder	1
Fold-3	1250_Ruler	Ruler	1
	1300_Ruler	Ruler	0
	1350_Binder	Binder	1
	1375_Binder	Binder	1
	1550_Ruler	Ruler	0
Fold-4	1575_Binder	Binder	0
	1600_Ruler	Ruler	0
	1625_Binder	Binder	0
	1650_Binder	Binder	1
	1700_Ruler	Ruler	1
Fold-5	1775_Binder	Binder	0
	1800_Ruler	Ruler	1
	1850_Ruler	Ruler	0
	1900_Binder	Binder	1
	2000_Ruler	Ruler	1

Figure 11.

To calculate the target representation values for Fold-1, use the data from folds 2-5. Since there are ten observations with a *Raw_Product_TH* of 1 in those folds, the denominator for the calculation is ten. To get the numerator, take the sum of *Raw_Product_TH* grouped by the *Product* value (6 for "Binder", 4 for "Ruler"). The calculation yields a *Raw_Product_Target_Rep* value of 60% for "Binder" (6÷10), and 40% for "Ruler" (4÷10). Assign these values to their respective *Product* value in Fold-1 (Figure 12).

	Transaction_ID_Product	Product	Raw_Product_Target_Rep
Fold-1	500_Ruler	Ruler	**40%**
	1000_Ruler	Ruler	**40%**
	1000_Binder	Binder	**60%**
	1025_Ruler	Ruler	**40%**
	1150_Binder	Binder	**60%**
Fold-2	1155_Binder	Binder	
	1175_Ruler	Ruler	
	1180_Binder	Binder	
	1190_Ruler	Ruler	
	1225_Binder	Binder	
Fold-3	1250_Ruler	Ruler	
	1300_Ruler	Ruler	
	1350_Binder	Binder	
	1375_Binder	Binder	
	1550_Ruler	Ruler	
Fold-4	1575_Binder	Binder	
	1600_Ruler	Ruler	
	1625_Binder	Binder	
	1650_Binder	Binder	
	1700_Ruler	Ruler	
Fold-5	1775_Binder	Binder	
	1800_Ruler	Ruler	
	1850_Ruler	Ruler	
	1900_Binder	Binder	
	2000_Ruler	Ruler	

Figure 12. Assigned Target Representation for Fold-1

To calculate the target representation values for Fold-2, follow the same process as above, except calculate the values based on the data from folds 1, 3, 4, and 5. The values of *Raw_Product_Target_Rep* for this fold are .45% for "Binder" (5÷11), and 55% for "Ruler" (6÷11).

	Transaction_ID_Product	Product	Raw_Product_Target_Rep
Fold-1	500_Ruler	Ruler	**40%**
	1000_Ruler	Ruler	**40%**
	1000_Binder	Binder	**60%**
	1025_Ruler	Ruler	**40%**
	1150_Binder	Binder	**60%**
Fold-2	1155_Binder	Binder	**45%**
	1175_Ruler	Ruler	**55%**
	1180_Binder	Binder	**45%**
	1190_Ruler	Ruler	**55%**
	1225_Binder	Binder	**45%**
Fold-3	1250_Ruler	Ruler	
	1300_Ruler	Ruler	
	1350_Binder	Binder	
	1375_Binder	Binder	
	1550_Ruler	Ruler	
Fold-4	1575_Binder	Binder	
	1600_Ruler	Ruler	
	1625_Binder	Binder	
	1650_Binder	Binder	
	1700_Ruler	Ruler	
Fold-5	1775_Binder	Binder	
	1800_Ruler	Ruler	
	1850_Ruler	Ruler	
	1900_Binder	Binder	
	2000_Ruler	Ruler	

Figure 13. Assigned Target Representation for Folds 1 and 2

Repeat this process until the dataset looks like Figure 14. Please ensure that you are using the dataset from Figure 11 to calculate these values for folds 2-5, and not the newly assigned values seen in Figures 12 and 13.

	Transaction_ID_Product	Product	Raw_Product_Target_Rep
Fold-1	500_Ruler	Ruler	**40%**
	1000_Ruler	Ruler	**40%**
	1000_Binder	Binder	**60%**
	1025_Ruler	Ruler	**40%**
	1150_Binder	Binder	**60%**
Fold-2	1155_Binder	Binder	**45%**
	1175_Ruler	Ruler	**55%**
	1180_Binder	Binder	**45%**
	1190_Ruler	Ruler	**55%**
	1225_Binder	Binder	**45%**
Fold-3	1250_Ruler	Ruler	**50%**
	1300_Ruler	Ruler	**50%**
	1350_Binder	Binder	**50%**
	1375_Binder	Binder	**50%**
	1550_Ruler	Ruler	**50%**
Fold-4	1575_Binder	Binder	**55%**
	1600_Ruler	Ruler	**45%**
	1625_Binder	Binder	**55%**
	1650_Binder	Binder	**55%**
	1700_Ruler	Ruler	**45%**
Fold-5	1775_Binder	Binder	**60%**
	1800_Ruler	Ruler	**40%**
	1850_Ruler	Ruler	**40%**
	1900_Binder	Binder	**60%**
	2000_Ruler	Ruler	**40%**

Figure 14. Assigned Target Representation for All Folds

The last step is to calculate the final *Mean_Product_Target_Rep* values, done by taking the mean of the target representation for both "Binder" and "Ruler" from all five folds (Figure 15). The reason is so you can easily map the newly created values to new data (e.g., for scoring). Leaving this step out could have you end up with one *Product* value containing five different values of the target representation from the five folds. The sum of this column will not equal to 100% (albeit close to 100%) due to the overlap of data from the k-1 folds used for the calculation of the target representation.

Product	Mean_Product_Target_Rep
Binder	**53%**
Ruler	**45%**

Figure 15. Mean Target Representations for All Folds

SECTION 5. A SAS IMPLEMENTATION OF CALCULATING TARGET REPRESENTATION USING K-FOLD TARGET ENCODING

This section applies the concepts reviewed in Sections 3 and 4 via SAS code using the `SESUGDTA.raw_non_sum_file` dataset. The following code will create the three new numeric features derived from the *Raw_Categorical_Input_4* feature. However, before applying these concepts, some pre-processing will need to be done first to set up the data.

You will first need to create a list of unique *Transaction_ID* values ❶ for the train dataset. The *Train* column ❷ contains the train dataset indicator.

```
proc sql;
create table transaction_ID_list as select
distinct Transaction_Id, ❶
Target

from SESUGDTA.raw_non_sum_file

where Train=1 ❷

order by Target;
run;
```

The next step is to assign the fold IDs to the `transaction_ID_list` dataset using PROC SURVEYSELECT. The GROUP= option specifies the value of k ❶. The STRATA statement stratifies the folds by the *Target* column ❸. The RENAME= option renames the output column containing the fold assignments to *Train_Fold* ❷.

```
proc surveyselect data=transaction_ID_list group=5 ❶ seed=220401
                              out=strat_kfold (RENAME=(groupid=Train_Fold )); ❷
strata Target; ❸
run;
```

The next step maps the `strat_kfold` dataset with the fold assignments back to the original `SESUGDTA.raw_non_sum_file` dataset using the *Transaction_ID* as the key ❶.

```
proc sql;
create table raw_data_w_folds as select
a.*,
b.Train_Fold

from SESUGDTA.raw_non_sum_file as a left join strat_kfold as b on
                                a.Transaction_Id = b.Transaction_Id; ❶
quit;
```

The next code block is where the feature engineering methodology begins. It first creates a concatenated string column that combines the values of the *Transaction_ID* and *Raw_Categorical_Input_4* columns. ❶. The `raw_data_w_folds` dataset then gets split into train (`raw_train_w_kfolds`) ❷ and validation (`raw_validation_w_kfolds`) datasets ❸.

```
data raw_train_w_kfolds raw_validation_w_kfolds ;
    set raw_data_w_folds;

Trans_ID_Raw_Categorical_Input_4 = CATX("~", Transaction_Id, Raw_Categorical_Input_4); ❶
```

13

```
        if Train = 1 then output raw_train_w_kfolds; ❷
            else output raw_validation_w_kfolds;
    run;
```

Here is where the loop for k-fold target encoding begins ❶. It creates a column containing the original values of the *Raw_Categorical_Input_4* column ❷ by splitting the concatenated string column *Trans_ID_Raw_Categorical_Input_4*. The code then creates the target indicator column ❸ on the data from four out of the five folds ❹ and drops all observations where the input value does not have a target hit ❺.

A PROC SQL statement ❻ creates a macro variable (*tot_train_cf*) ❽ before summarizing the data to the values of *Raw_Categorical_Input_4*, which contains the sum of the *Raw_Categorical_Input_4_TH* column ❼.

```
%macro kfold(fold); ❶

proc sql;
create table raw_categorical_input_dedup as select
Trans_ID_Raw_Categorical_Input_4,
substr(Trans_ID_Raw_Categorical_Input_4, index(Trans_ID_Raw_Categorical_Input_4, '~') +1)
                                           as Raw_Categorical_Input_4, ❷
case when sum(Target) > 0 then 1
    else 0 end as Raw_Categorical_Input_4_TH ❸

from raw_train_w_Kfolds

where Train_Fold ne &fold ❹

group by Trans_ID_Raw_Categorical_Input_4

having Raw_Categorical_Input_4_TH > 0; ❺
quit;

proc sql; ❻
select sum(Raw_Categorical_Input_4_TH) ❼
into :tot_train_cf ❽
from raw_categorical_input_dedup;
quit;
```

The next code block summarizes the `raw_categorical_input_dedup` dataset ❸ to the values of *Raw_Categorical_Input_4* ❹ using the data from k - 1 folds. It then maps to the dataset containing the fold left out by the WHERE statement ❷ using *Raw_Categorical_Input_4* as the key ❺. Please note that the columns calculated for this new dataset ❶ are only for observations where the *Raw_Categorical_Input_4* had a *Fraud* value of 1 ❻.

```
proc sql;
create table Raw_Categorical_Input_4_f_&fold as select ❶
a.Transaction_Id,
a.Raw_Categorical_Input_4,
case when b.Tot_Raw_Categorical_Input_4_TH > 0 then 1
    else 0 end as Raw_Categorical_Input_4_TH,
b.Raw_Categorical_Input_4_Sum_Rep

from (select Trans_ID_Raw_Categorical_Input_4,
       substr(Trans_ID_Raw_Categorical_Input_4,
       index(Trans_ID_Raw_Categorical_Input_4, '~')  +1) as Raw_Categorical_Input_4,
       scan(Trans_ID_Raw_Categorical_Input_4,1,'~') as Transaction_Id,
       count(*) as total_line_items

       from raw_train_w_Kfolds
```

```
            where Train_Fold = &fold ❷

            group by Trans_ID_Raw_Categorical_Input_4) as a left join (select
                                        Raw_Categorical_Input_4,
                                        sum(Raw_Categorical_Input_4_TH) as
                                        Tot_Raw_Categorical_Input_4_TH,
                                        sum(Raw_Categorical_Input_4_TH)/&tot_train_cf. as
                                        Raw_Categorical_Input_4_Sum_Rep

                                        from raw_categorical_input_dedup ❸

                                        group by Raw_Categorical_Input_4 ❹) as b
                                        on a.Raw_Categorical_Input_4 =
                                        b.Raw_Categorical_Input_4 ❺

    having Raw_Categorical_Input_4_TH > 0; ❻
    quit;

    %mend;

    %kfold(1);
    %kfold(2);
    %kfold(3);
    %kfold(4);
    %kfold(5);
```

The next step appends all five datasets created by the macro ❶.

```
    data raw_categorical_input_4_train;
        set raw_categorical_input_4_f_1
                raw_categorical_input_4_f_2
                raw_categorical_input_4_f_3
                raw_categorical_input_4_f_4
                raw_categorical_input_4_f_5; ❶
    run;
```

The code then summarizes the `raw_categorical_input_4_train` dataset to the unique values found in the *Raw_Categorical_Input_4* input ❸. It calculates the target representation ❶ and target indicator ❷ columns using data from all five folds.

```
    proc sql;
    create table Raw_Categorical_Input_4_trn_avg as select
    Raw_Categorical_Input_4,
    mean(Raw_Categorical_Input_4_Sum_Rep) as Mean_Categorical_Input_4_Sum_Rep, ❶
    case when sum(Raw_Categorical_Input_4_TH) >0 then 1
        else 0 end as Raw_Categorical_Input_4_TH_Ind ❷

    from raw_categorical_input_4_train

    group by Raw_Categorical_Input_4 ❸

    having Raw_Categorical_Input_4_TH_Ind ne 0;
    quit;
```

Next, the `Raw_Categorical_Input_4_trn_avg` dataset maps to the train ❶ and validation ❸ datasets using *Raw_Categorical_Input_4* as the key (❷ and ❹).

```
    proc sql;
    create table categorical_input_init_train as select
    a.Transaction_Id,
    a.Raw_Categorical_Input_4,
```

15

```
          b.Mean_Categorical_Input_4_Sum_Rep,
          b.Raw_Categorical_Input_4_TH_Ind

     from (select Trans_ID_Raw_Categorical_Input_4,
              substr(Trans_ID_Raw_Categorical_Input_4,index(Trans_ID_Raw_Categorical_Input_4,'~') +1)
                   as Raw_Categorical_Input_4,
              scan(Trans_ID_Raw_Categorical_Input_4,1,'~') as Transaction_Id,
              count(*) as Total_Line_Products

              from raw_train_w_Kfolds ❶

              group by Trans_ID_Raw_Categorical_Input_4) as a left join Raw_Categorical_Input_4_trn_avg
              as b on a.Raw_Categorical_Input_4 = b.Raw_Categorical_Input_4 ❷

     where b.Raw_Categorical_Input_4_TH_Ind > 0;
     quit;


     proc sql;
     create table categorical_input_init_val as select
     a.Transaction_Id,
     a.Raw_Categorical_Input_4,
     b.Mean_Categorical_Input_4_Sum_Rep,
     b.Raw_Categorical_Input_4_TH_Ind

     from (select Trans_ID_Raw_Categorical_Input_4,
              substr(Trans_ID_Raw_Categorical_Input_4,
              index(Trans_ID_Raw_Categorical_Input_4,'~')+1) as Raw_Categorical_Input_4,
              scan(Trans_ID_Raw_Categorical_Input_4,1,'~') as Transaction_Id,
              count(*) as Total_Line_Products

              from raw_validation_w_Kfolds ❸

              group by Trans_ID_Raw_Categorical_Input_4) as a left join
              Raw_Categorical_Input_4_trn_avg as b on a.Raw_Categorical_Input_4 =
              b.Raw_Categorical_Input_4 ❹

     where b.Raw_Categorical_Input_4_TH_Ind > 0;
     quit;
```

Finally, the last two PROC SQL statements summarizes the raw train ❷ and validation ❺ datasets to the *Transaction_ID* level (❸ and ❻), creating the final datasets (`sum_categorical_input_4_train` ❶ and `sum_categorical_input_4_val` ❸) used for training machine learning models.

```
     proc sql;
     create table sum_categorical_input_4_train as select ❶
     Transaction_Id,
     sum(Raw_Categorical_Input_4_TH_Ind) as Categorical_Input_4_Tot_TH,
     sum(Mean_Categorical_Input_4_Sum_Rep) as Categorical_Input_4_Sum_Tgt_Rep,
     case when calculated Categorical_Input_4_Tot_TH > 0 then
        calculated Categorical_Input_4_Tot_TH * calculated Categorical_Input_4_Sum_Tgt_Rep
     else 0 end as Categorical_Input_4_Enc_Prod

     from categorical_input_init_train ❷

     group by Transaction_Id; ❸
     quit;

     proc sql;
     create table sum_categorical_input_4_val as select ❹
     Transaction_Id,
     sum(Raw_Categorical_Input_4_TH_Ind) as Categorical_Input_4_Tot_TH,
     sum(Mean_Categorical_Input_4_Sum_Rep) as Categorical_Input_4_Sum_Tgt_Rep,
     case when calculated Categorical_Input_4_Tot_TH > 0 then
        calculated Categorical_Input_4_Tot_TH * calculated Categorical_Input_4_Sum_Tgt_Rep
     else 0 end as Categorical_Input_4_Enc_Prod
```

```
from categorical_input_init_val ❺

group by Transaction_Id; ❻
quit;

proc datasets; delete transaction_ID_list strat_kfold
raw_data_w_folds raw_categorical_input_dedup
raw_categorical_input_4_f_1 raw_categorical_input_4_f_2
raw_categorical_input_4_f_3 raw_categorical_input_4_f_4
raw_categorical_input_4_f_5 raw_categorical_input_4_train
raw_train_w_Kfolds  raw_validation_w_Kfolds
categorical_input_init_train
categorical_input_init_val;
quit;
```

SECTION 6. COMPARING THE PERFORMANCE OF MACHINE LEARNING MODELS WITH AND WITHOUT NEWLY CREATED FEATURES

To assess whether the newly created inputs can improve model performance, ten supervised classification models were trained using the following datasets: `SESUGDTA.model_train` and `SESUGDTA.model_train_val` (see APPENDIX A: OVERVIEW OF DATA USED IN EXAMPLES for descriptions). Five were trained using a list of inputs that included the new features and the other five without them. The models trained were decision tree, logistic regression, random forest, gradient boosting, and neural network. The performance metric used for comparison was *recall* at the top scored percentile. *Recall* for this example is the proportion of all transactions with a *Target* value of 1 scored in the top 1% of the validation dataset based on the values of the *P_Target1* column. You can find the code used to fit the models and create the model metrics table in APPENDIX B: SAS CODE FOR MODEL TRAINING AND EVALUATION.

As you will see in Figure 16 (note that the x-axis starts at 70%), all models that included the four newly created features (with recall percent enclosed in blue circles) clearly outperformed the models without these features (recall enclosed in grey circles) on the validation dataset. For tables containing additional model performance statistics, see APPENDIX C: ADDITIONAL MODEL PERFORMANCE TABLES.
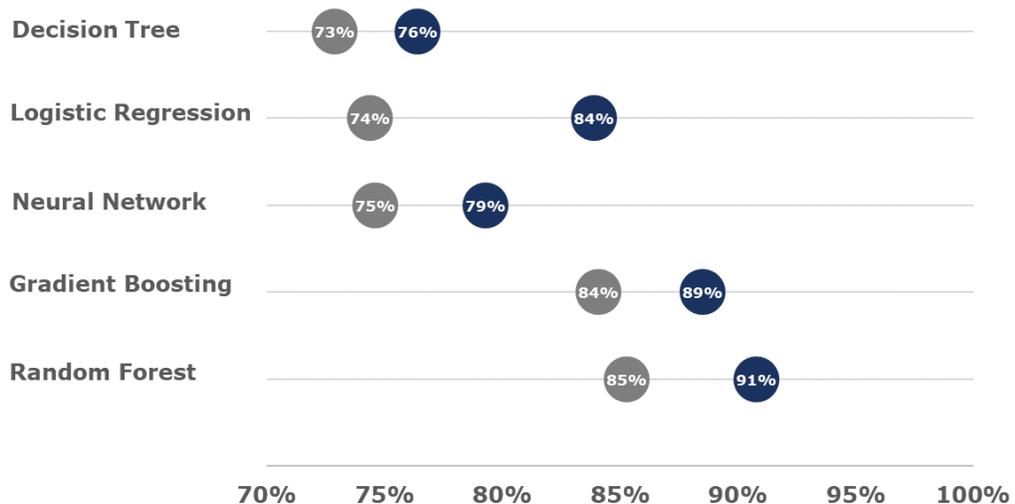
Figure 16. Recall on Validation Dataset

SECTION 7. WATCHOUT!

While the test described in the previous section demonstrated the benefits of including these newly created features in your models, there are pitfalls you should watch out for. Here are a few, along with suggestions to address them:

- **Dirty data** – Raw categorical data are notorious for being messy, where you must deal with issues such as misspellings, formatting irregularities, and mismatched cases. Without proper treatment, the new features will contain inaccurate representations of the original input.
    - o **How to address** – Do extensive data cleaning on all categorical features that you want to use for your analyses. Review results with a subject matter expert before proceeding with feature engineering.

- **Overfitting** – Engineering new features tied to the target increases the likelihood of data leakage, which could then cause overfitting.
    - o **How to address** – Apply k-fold target encoding on the train dataset, as demonstrated in SECTION 5. A SAS IMPLEMENTATION OF CALCULATING TARGET REPRESENTATION AND K-FOLD TARGET ENCODING.
        - ▪ You can also apply smoothing to the target representation calculation, which considers the total occurrences for each unique input value [10]. Feel free to experiment with the different data leakage/overfitting mitigation measures to see what works best for your analysis.

- **Scoring data with previously unseen categorical values** – When scoring new data, it is possible for a transaction to contain values that were not in the original training data. There is no information on whether this new value is associated with the target or not.
    - o **How to address** – Apply additive smoothing, which would assign a value close to the overall mean of the target column to the new unseen value.

- **Bias towards transactions with multiple observations** – Transactions with multiple unique feature values associated with the target can have an overall stronger association with the target compared to records with a single observation. This is especially true for the total target hit (e.g., *Tot_Product_TH*) and product inputs (e.g., *Product_End_Prod*).
    - o **How to address** – If your dataset contains records with mostly single observation transactions, consider only using the sum target representation feature (e.g., *Sum_Product_Target_Rep*). Even if you have mostly single observation transactions, having a bias towards the multiple observation transactions might not be an issue depending on the problem you want to solve. Consult with a subject matter expert for guidance.

CONCLUSION

Working with categorical inputs in transactional datasets can present many challenges. The problem of multi-dimensional high cardinality is one of them, and there are not many clear solutions, or at least ones that are publicly available. While the new method introduced in this paper proved to be effective on a single dataset, additional research is necessary to confirm whether these results would generalize. In the end, a data scientist should always be willing to experiment with multiple methods, whether traditional or new, to see what works best for the problem at hand.

APPENDIX A: OVERVIEW OF DATA USED IN EXAMPLES

The methods summarized in this paper were demonstrated on masked data from an invoice error detection project at Georgia-Pacific LLC. The data contained 917,699 unique invoices, of which 1,158 of them had an error associated with them. The target column is *Target* (1=error, 0=no error).

The first dataset, `SESUGDTA.raw_non_sum_file`, contains all transactions and their multiple line items in their non-summarized form. The dataset contains 1,082,788 million observations and the following four inputs:

1. *Transaction_ID*: Transaction ID number

2. *Raw_Categorical_Input_1*: Raw categorical input containing masked versions of the original input, done by assigning each distinct value a prefix of "Raw_Categorical_Value_", followed by a number representing the unique value

3. *Target*: Dichotomous variable indicating whether the distinct transaction ID had an error (*Target*=1) associated with it

4. *Train*: Dichotomous variable indicating whether the distinct transaction ID is a part of the training dataset (*Train*=1)

Two datasets were used for the model training and comparison portion of this paper (SECTION 6. COMPARING THE PERFORMANCE OF MACHINE LEARNING MODELS WITH AND WITHOUT NEWLY CREATED FEATURES). The names of the datasets are `SESUGDTA.model_train` and `SESUGDTA.model_train_val`, and the observation and target counts can be found in Table 2 below. The four new columns tested were: *Categorical_Input_1_Enc_Prod, Categorical_Input_2_Sum_Tgt_Rep*, *Categorical_Input_3_Enc_Prod*, *and Categorical_Input_4_Sum_Tgt_Rep.* Please note that the `SESUGDTA.model_train_val` dataset contains both training and validation data. The training data indicator column is *Train*, which has a 70/30 train/validation split.

Dataset	Total Transactions with an Error	Total Transactions without an Error	Target %
model_train	811	641,579	.13%
model_train_val	1,158	916,541	.13%

Table 2.

Both datasets contain nineteen columns, sixteen of which were used as inputs for model training.

1. *Transaction_ID*: Transaction ID number

2. *Categorical_Input_1_Enc_Prod*: Sum target representation feature based on methods outlined in SECTION 3. RATIONALE AND METHODOLOGY OF NEW FEATURE ENGINEERING METHOD

3. *Categorical_Input_2_Sum_Tgt_Rep*: Feature containing the product of the sum target representation and total target hit indicator features, based on methods outlined in SECTION 3. RATIONALE AND METHODOLOGY OF NEW FEATURE ENGINEERING METHOD

4. *Categorical_Input_3_Enc_Prod*: Sum target representation feature based on methods outlined in SECTION 3. RATIONALE AND METHODOLOGY OF NEW FEATURE ENGINEERING METHOD

5. *Categorical_Input_4_Sum_Tgt_Rep*: Feature containing the product of the sum target representation and total target hit indicator features based on methods outlined in SECTION 3. RATIONALE AND METHODOLOGY OF NEW FEATURE ENGINEERING METHOD

6. *Numeric_Input_1*: Numeric input

7. *Numeric_Input_2*: Numeric input

8. *Numeric_Input_3*: Numeric input

9. *Numeric_Input_4*: Numeric input

10. *Numeric_Input_5*: Numeric input

11. *Numeric_Input_6*: Numeric input

12. *Numeric_Input_7*: Numeric input

13. *Numeric_Input_8*: Numeric input

14. *Numeric_Input_9*: Numeric input

15. *Numeric_Input_10*: Numeric input

16. *Numeric_Input_11*: Numeric input

17. *Numeric_Input_12*: Numeric input

18. *Target*: Dichotomous variable indicating whether the distinct transaction ID had an error (*Target*=1) associated with it

19. *Train*: Dichotomous variable indicating whether the distinct transaction ID is a part of the training dataset (*Train*=1)

The SAS code files (along with their Python Jupyter Notebook equivalents) and datasets can be downloaded from https://github.com/nikolicxa/multi-dimensional-high-cardinality.

APPENDIX B: SAS CODE FOR MODEL TRAINING AND EVALUATION

```
/***********************************************************************/
/*              Preliminary notes                                   */
/***********************************************************************/

/* To create the recall summary chart found in the COMPARING THE PERFORMANCE
OF MACHINE LEARNING MODELS WITH AND WITHOUT NEWLY CREATED FEATURES section,
you will need to have access to SAS Viya 3.5 or above, and load the model_train
and model_train_val datasets into a SAS directory.
*/

/***********************************************************************/
/*              Assigning libname and opening a CAS session         */
/***********************************************************************/

libname SESUGDTA ''; ***** Insert path to where datasets are stored *****;

%let outdir = ; ***** Insert path to where you want to store models fitted by proc treesplit,
proc logselect, and proc nnet *****;

***** Create a CAS session *****;
cas mySession sessopts=(caslib=casuser timeout=1800 locale="en_US");
caslib _all_ assign;

/***********************************************************************/
/*              Creating macro variables containing input lists for    */
/*              model comparisons                                       */
/***********************************************************************/

%let all = Categorical_Input_1_Enc_Prod Numeric_Input_1 Categorical_Input_2_Sum_Tgt_Rep
Numeric_Input_2 Categorical_Input_3_Enc_Prod Numeric_Input_3 Numeric_Input_4 Numeric_Input_5
Numeric_Input_6 Numeric_Input_7 Numeric_Input_8 Numeric_Input_9 Numeric_Input_10
Categorical_Input_4_Sum_Tgt_Rep Numeric_Input_11 Numeric_Input_12;

%let noenc = Numeric_Input_1 Numeric_Input_2 Numeric_Input_3 Numeric_Input_4 Numeric_Input_5
Numeric_Input_6 Numeric_Input_7 Numeric_Input_8 Numeric_Input_9 Numeric_Input_10 Numeric_Input_11
Numeric_Input_12;

/***********************************************************************/
/*              Loading model_train and model_train_val datasets into  */
/*              memory                                                  */
/***********************************************************************/

data casuser.model_train;
      set SESUGDTA.model_train;
run;

data casuser.model_train_val;
      set SESUGDTA.model_train_val;
run;

/***********************************************************************/
/*              Training five different models for both input lists    */
/***********************************************************************/

%macro varlist(list,name);

***** Decision Tree *****;
proc treesplit data= casuser.model_Train;
   class Target;
   model Target = &list;
   code file="&outdir./DT_Model_&name..sas";
run;

data casuser.scored_DT_&name (keep=P_Target1 P_Target0 Target Train);
  set casuser.model_Train_val (keep=&list Target Train);
  %include "&outdir./DT_Model_&name..sas";
run;

***** Logistic Regression *****;
proc logselect data=casuser.model_Train;
```

```sas
   model Target(event='1')= &list;
    code file="&outdir./LR_Model_&name..sas" pcatall;
run;

data casuser.scored_LR_&name (keep=P_Target1 P_Target0 Target Train);
  set casuser.model_Train_val (keep=&list Target Train);
   %include "&outdir./LR_Model_&name..sas";
run;


***** Random Forest *****;
proc forest data=casuser.model_Train outmodel=casuser.RF_&name;
      input &list/ level = interval;
    Target Target/ level = nominal;
run;

proc forest data=casuser.model_Train_val inmodel=casuser.RF_&name;
  output out=casuser.scored_RF_&name copyvars=(_ALL_);
run;

***** Gradient Boosting *****;
proc gradboost data=casuser.model_Train  outmodel=casuser.GB_&name;
  input  &list   / level = interval;
  Target Target/ level=nominal;
run;

proc gradboost  data=casuser.model_Train_val   inmodel=casuser.GB_&name noprint;
  output out=casuser.scored_GB_&name copyvars=(_ALL_);
run;

***** Neural Network *****;
proc nnet data=casuser.model_Train;
  Target Target/ level=nom;
  input &list / level=int;
      hidden 3;
  Train outmodel=casuser.nnet_model;
  ods exclude OptIterHistory;
  code file="&outdir./NN_&name..sas";
run;

data casuser.scored_NN_&name (keep=P_Target1 P_Target0 Target Train);
  set casuser.model_Train_val  (keep=&list Target Train);
   %include "&outdir./NN_&name..sas";
run;

%mend;


%varlist(&all,W_ENC);
%varlist(&noenc,WO_ENC);

/************************************************************************/
/*            Calculating recall at the highest scored percentile      */
/*            on the validation dataset                                 */
/************************************************************************/

/* Please note that the recall statistics will be slightly different
compared to Table 4 for tree-based models */

%macro dataset(name, model, list);

      proc rank data=casuser.scored_&name out=ranked_&name groups=100 descending;
      var p_target1;
      ranks target_score_rank;
      where train = 0; /* Change to 1 to calculate train data performance metrics */
      run;

      proc sql;
      select sum(target)
      into :tot_val_cf
      from ranked_&name;
      quit;

      proc sql;
      create table top_1_pct_rank_&name (drop=target_score_rank) as select
```

```
        target_score_rank,
        &model as Model,
        sum(target)  as Tot_target_recall_&list,
        calculated Tot_target_recall_&list/&tot_val_cf. as Pct_target_recall_&list format
        percent8.1

        from ranked_&name

        where target_score_rank = 0

        group by target_score_rank;
        quit;
%mend;

%dataset(DT_W_ENC, 'Decision Tree', W_ENC);
%dataset(DT_WO_ENC, 'Decision Tree', WO_ENC);
%dataset(LR_W_ENC, 'Logistic Regression', W_ENC);
%dataset(LR_WO_ENC, 'Logistic Regression', WO_ENC);
%dataset(RF_W_ENC, 'Random Forest', W_ENC);
%dataset(RF_WO_ENC, 'Random Forest', WO_ENC);
%dataset(GB_W_ENC, 'Gradient Boosting', W_ENC);
%dataset(GB_WO_ENC, 'Gradient Boosting', WO_ENC);
%dataset(NN_W_ENC, 'Neural Network', W_ENC);
%dataset(NN_WO_ENC, 'Neural Network', WO_ENC);

/**********************************************************************/
/*              Appending and joining recall datasets               */
/*              to create final comparison dataset                  */
/**********************************************************************/

data top_1_pct_recall_stats_w_enc;
format Model $32.;
        set top_1_pct_rank_DT_w_Enc
                top_1_pct_rank_LR_w_Enc
                top_1_pct_rank_RF_w_Enc
                top_1_pct_rank_GB_w_Enc
                top_1_pct_rank_NN_w_Enc;
run;

data top_1_pct_recall_stats_wo_enc;
format Model $32.;
        set top_1_pct_rank_DT_wo_Enc
                top_1_pct_rank_LR_wo_Enc
                top_1_pct_rank_RF_wo_Enc
                top_1_pct_rank_GB_wo_Enc
                top_1_pct_rank_NN_wo_Enc;
run;

proc sql;
create table top_1_pct_recall_summary as select
a.Model,
a.Tot_target_recall_w_Enc,
a.Pct_target_recall_w_Enc,
b.Tot_target_recall_wo_Enc,
b.Pct_target_recall_wo_Enc

from top_1_pct_recall_stats_w_enc as a left join top_1_pct_recall_stats_wo_enc as b on a.model =
b.model;
quit;

proc datasets; delete top_1_pct_rank_DT_w_Enc top_1_pct_rank_LR_w_Enc top_1_pct_rank_RF_w_Enc
                    top_1_pct_rank_GB_w_Enc top_1_pct_rank_NN_w_Enc top_1_pct_rank_DT_wo_Enc
                    top_1_pct_rank_LR_wo_Enc top_1_pct_rank_RF_wo_Enc top_1_pct_rank_GB_wo_Enc
                    top_1_pct_rank_NN_wo_Enc ranked_DT_W_ENC ranked_DT_WO_ENC ranked_LR_W_ENC
                    ranked_LR_WO_ENC ranked_RF_W_ENC ranked_RF_WO_ENC ranked_GB_W_ENC
                    ranked_GB_WO_ENC ranked_NN_W_ENC ranked_NN_WO_ENC
                    top_1_pct_recall_stats_w_enc top_1_pct_recall_stats_wo_enc;
run;
```

APPENDIX C: ADDITIONAL MODEL PERFORMANCE TABLES

Model	Total Transactions with Target Hit Scored in Top 1% - With Encoded Variables	Recall With Encoded Inputs	Total Transactions with Target Hit Scored in Top 1% - Without Encoded Variables	Recall Without Encoded Inputs
Decision Tree	750	92.5%	684	84.3%
Logistic Regression	674	83.1%	626	77.2%
Random Forest	811	100.0%	798	98.4%
Gradient Boosting	786	96.9%	752	92.7%
Neural Network	702	86.6%	643	79.3%

Table 3. Recall stats detail for train dataset

Model	Total Transactions with Target Hit Scored in Top 1% - With Encoded Variables	Recall With Encoded Inputs	Total Transactions with Target Hit Scored in Top 1% - Without Encoded Variables	Recall Without Encoded Inputs
Decision Tree	291	83.90%	258	74.40%
Logistic Regression	265	76.40%	253	72.90%
Random Forest	315	90.80%	296	85.30%
Gradient Boosting	307	88.50%	292	84.10%
Neural Network	275	79.30%	259	74.60%

Table 4. Recall stats detail for validation dataset

REFRENCES

[1]Laramore, Jay. et al. 2017. *Feature Engineering and Data Preparation for Analytics Course Notes*. Cary, NC: SAS Institute Inc.

[2][8]Kuhn, Max., Johnson, Kjell. 2020. *Feature Engineering and Selection: A Practical Approach for Predictive Models*. Boca Raton, FL: CRC Press.

[3][5]Pargent, F., Pfisterer, F., Thomas, J. et al. (2022) "Regularized target encoding outperforms traditional methods in supervised machine learning with high cardinality features." *Computational Statistics*, https://doi.org/10.1007/s00180-022-01207-6.

[4][7]Zheng, Alice., Casari, Amanda. 2018. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. Sebastopol, CA: O'Reilly Media, Inc.

[6]Holbrook, R., Cook, A. (2022) "Target Encoding". Available at https://www.kaggle.com/code/ryanholbrook/target-encoding

[9]Hasz, Brandon. (2019) "Representing Categorical Data with Target Encoding." Date Accessed: August 31, 2022. Available at https://brendanhasz.github.io/2019/03/04/target-encoding.

[10]Halford, Max. (2018) "Target encoding done the right way." *Max Halford*. Date Accessed: August 31, 2022. Available at https://maxhalford.github.io/blog/target-encoding/.

ACKNOWLEDGMENTS

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Aleksandar Nikolic
Senior Data Scientist
Georgia-Pacific LLC
nikolicxa@gmail.com