Tabular Engineering with Automunge

Nicholas J. Teague
Automunge
Altamonte Springs, FL 32714
`nteague@automunge.com`

Abstract

Automunge is an open source python library that has formalized and automated the data preparations for tabular learning in between the workflow boundaries of received “tidy data” (one column per feature and one row per sample) and returned dataframes suitable for the direct application of machine learning. Under automation numeric features are normalized, categoric features are binarized, and missing data is imputed. Data transformations are fit to properties of a training set for a consistent basis on any partitioned “validation data” or additional “test data”. When preparing training data, a compact python dictionary is returned recording steps and parameters of transformation, which may then serve as a key for preparing additional corresponding data on a consistent basis. In addition to data preparations under automation, Automunge may also serve as a platform for tabular engineering, as demonstrated herein.

1 Automunge

Automunge [1] is an open source python library, available now for pip install, built on top of Pandas [2], SciKit-learn [3], Scipy [4], and Numpy [5]. It takes as input tabular data received in a tidy form [6], meaning one column per feature and one row per sample, and returns numerically encoded sets with infill to missing points, thus providing a push-button means to feed raw tabular data directly to machine learning. The extent of derivations may be minimal, such as numeric normalizations and categoric binarizations under automation, or may include more elaborate univariate transformations, including aggregated sets thereof. Generally speaking, the transformations are performed based on a “fit” to properties of features in a designated training set, and then that same basis may be used to consistently and efficiently prepare subsequent test data, as may be intended for use in inference or for additional training data preparation.

The interface is channeled through two master functions, automunge(.) and postmunge(.). The automunge(.) function receives a training set and if available also a consistently formatted test set, and returns a collection of dataframes intended for training, validation, and inference — each of these aggregations further segregated into subsets of features, index, and label sets. A validation set, if designated by ratio of partitioned data from the training set, is segregated from the training data prior to transformations and then consistently prepared on the train set basis to avoid data leakage between training and validation. The function also returns a populated python dictionary, which we call the postprocess_dict, recording steps and parameters of transformations. This dictionary may then be passed along with subsequent test data to the postmunge(.) function for consistent preparations on the train set basis, as for instance may be applied sequentially to streams of data. Because it makes use of train set properties evaluated during a corresponding automunge(.) call instead of directly evaluating properties of the test data, preparing data in the postmunge(.) function is very efficient.

2 Tabular Engineering

Tabular engineering refers to the construction of transformation sets for application to a tabular feature, which in some cases may include generations and branches of derivations to prepare features in multiple configurations of varying information content. The platform has an extensive internal library of pre-defined transformations and transformation sets, each as may be based on properties derived from training data for a consistent basis toward validation or test data. A user may assemble custom transformation sets from those defined in the library, and may even define custom functions for integration into a transformation set using a very simple template [7], which by integrating custom transforms through the library enables push button support for missing data infill and subsequent preparations of additional corresponding data on the training set basis.

The specification of transformation sets is supported by two corresponding data structures [8], the "transformdict" for specifying sets of transformation categories associated with a root transformation category by way of entries to the Automunge "family tree primitives" [9], and the "processdict" for defining properties of transformation categories - including the associated transformation functions for application towards training data / test data / inversion, any default parameters for those transformation functions, and properties associated with expected inputs and outputs. A defined root transformation category may then be assigned to a feature, and may also be used to overwrite one of the default root categories applied under automation.

Under automation, when a feature has not received a manual root category assignment, data properties of the feature are evaluated to derive a root transformation category for application. In one option, numeric features may have distribution properties evaluated to select between different types of distribution conversions. The application of feature property evaluations is modular, such that customized data property evaluations can be integrated if desired. Alternatively, features without explicit root category assignments may be returned as a pass-through without transformations, or without transformations other than missing data infill.

As transformation functions associated with transformation categories are applied, their application is recorded with the category as a suffix appender to the returned column headers, for example a z-score normalization by way of the 'nmbr' category targeting a feature with header 'column' would be returned with header 'column_nmbr'. If that transformation set included another downstream transformation, for example as 'bsor' for ordinal encoded standard deviation bins, that output would be returned as 'column_nmbr_bsor'. A retention of the intermediate stages in the returned set would be based on the configuration of family tree primitive entries.

Transformation functions that accept parameters may have selections designated targeting specific transformation category applications among different features or even among different generations applied to a common feature. In cases where the same transformation category is applied in different generations of transformations targeting a common input feature, the applications may be distinguished by the column header configuration including suffix appenders associated with input to the intended application. In order of precedence, parameter assignments may be designated targeting a transformation category as applied to a specific column header with suffix appenders, a transformation category as applied to an input column header (which may include multiple instances), all instances of a specific transformation category, all transformation categories, or may be initialized as default parameters when defining a transformation category.

An extensive library of transformations includes options like numeric translations [10], bin aggregations, date-time encodings, categoric encodings, and even parsed categoric encodings [11] in which categoric strings are vectorized based on shared grammatical structure between entries. Noise injection transforms [10] are available for both numeric and categoric features, as may benefit data augmentation and differential privacy. Dimensionality reductions may be applied, such as by principle component analysis [12], feature importance rankings, or categoric consolidations. Categoric consolidations refers to aggregating sets of two, more, or all categoric features or labels into a common categoric feature or label. A consolidation records a single encoding for each activation set, for example if two features each have unique entries {0,1}, a consolidation could aggregate unique entries {00, 01, 10, 11} and then encode them. Consolidated categoric labels can be used to train a single classification model for multiple labels. Inversion can recover the form of input to transformations and consolidations after an inference. Missing data receives ML infill [13], in which models are trained for a feature to impute missing entries based on properties of the surrounding features.

3 Demonstrations

The interface is channeled through two master functions, automunge(.) for preparing data and postmunge(.) for preparing additional corresponding data. As an example, for a training set df_train which includes a label feature 'labels', automunge(.) can be applied under automation as:

```
!pip install Automunge
from Automunge import *
am = AutoMunge()

train, train_ID, labels, \
val, val_ID, val_labels, \
test, test_ID, test_labels, \
postprocess_dict = \
am.automunge(df_train,
             labels_column = 'labels')
```

Some of the returned sets may be empty based on parameter selections. Using the returned dictionary postprocess_dict, corresponding data can then be prepared on a consistent basis with postmunge(.).

```
test, test_ID, test_labels, \
postreports_dict = \
am.postmunge(postprocess_dict,
             df_test)
```

To engineer a custom set of transformations, one can populate a transformdict and processdict entry for a new transformation category we'll call 'newt'. Here the functionpointer is used to match 'newt' to the processdict entries applied for 'nmbr', which is for z-score normalization. The transformdict is used to populate transformation category entries to the family tree primitives [Table 1] associated with a root category. The first four primitives are for upstream transforms. Since parents is a primitive with offspring, after applying transforms for the 'newt' entry, the downstream primitives from newt's family tree will be inspected to apply 'bsor' for ordinal encoded standard deviation bins to the output of the upstream transform. The upstream 'NArw' is used to aggregate missing data markers. The assigncat parameter is used to assign 'newt' as a root category to a target input column 'targetcolumn'.

```
processdict =  {'newt' : {'functionpointer'   : 'nmbr'}}

transformdict =  {'newt' : {'parents'       : ['newt'],
                            'siblings'      : [],
                            'auntsuncles'   : [],
                            'cousins'       : ['NArw'],
                            'children'      : [],
                            'niecesnephews' : [],
                            'coworkers'     : [],
                            'friends'       : ['bsor']}}

assigncat = {'newt' : ['targetcolumn']}
```

This transformation set will return columns with headers logging the applied transformation categories as: 'column_newt' (z-score normalization), 'column_newt_bsor' (ordinal encoded standard deviation bins), and 'column_NArw' (missing data markers). In an alternate configuration 'bsor' could be entered to an upstream primitive, this is just an example to demonstrate applying generations of transformations. Since friends is a supplement primitive, the upstream output 'column_newt' to which the 'bsor' transform is applied is retained in the returned data. And since cousins and friends are primitives without offspring, no further generations are inspected after applying their entries.

Options for root categories applied to unspecified features under automation are by the powertransform parameter, True means numeric features have conditional normalizations based on distribution properties. 'excl' is for direct passthrough, 'exc2' for passthrough as numeric, 'infill' for passthrough as numeric with ML infill. Defaults as False for normalized numeric and binarized categoric.

```
powertransform = True
```

Table 1: Family Tree Primitives

Primitive	Upstream / Downstream	Applied to Generation	Column Action	Downstream Offspring
parents	upstream	first	replace	yes
siblings	upstream	first	supplement	yes
auntsuncles	upstream	first	replace	no
cousins	upstream	first	supplement	no
children	downstream parents	offspring	replace	yes
niecesnephews	downstream siblings	offspring	supplement	yes
coworkers	downstream auntsuncles	offspring	replace	no
friends	downstream cousins	offspring	supplement	no

Parameters can be passed to the transformations through assignparam, as demonstrated here for updating a parameter setting so that the number of standard deviation bins for 'bsor' as applied to column 'column' is increased from the default of 6 to 7, where since this is an odd number will result in the center bin straddling the mean.

```
assignparam = {'bsor' : {'column' : {'bincount' : 7}}}
```

To consolidate categoric features, specification is by the Binary parameter. Consolidation takes place after family tree applications and infill, and accepts single column and multiple column integer encoded target categoric features, such as those encodings returned from the library of transforms. Target specifications can be via input headers to include all derived categoric sets from that feature, or via returned column headers with suffix appenders to target specific returned categoric sets. Demonstrated here is consolidating one set of features to return in a binarized form with retention of target columns, and one set returned in an ordinal encoded form with replacement. A first entry encoded in set brackets is used to specify the returned form, or when set bracket specification is omitted it defaults to returning as binarized with replacement. Consolidations can be specified to either replace or supplement the associated targets. Label consolidations are specified by the passing the labels_column parameter as a list of categoric labels with similar set bracket specification.

```
Binary = [ [{'retain'}, 'feature1', 'feature2'],
           [{'ordinal'}, 'feature3', 'feature4'] ]
```

Putting it all together in an automunge(.) call simply means passing our parameter specifications.

```
train, train_ID, labels, \
val, val_ID, val_labels, \
test, test_ID, test_labels, \
postprocess_dict = \
am.automunge(df_train,
             labels_column = 'labels',
             processdict = processdict,
             transformdict = transformdict,
             assigncat = assigncat,
             powertransform = powertransform,
             assignparam = assignparam,
             Binary = Binary)
```

One can then save the returned postprocess_dict, such as by downloading with the pickle library, to use as a key for preparing additional corresponding data on a consistent basis with postmunge(.).

4 Intellectual Property Disclaimer

Acknowledgments

A thank you owed to those facilitators behind Stack Overflow, Python, Numpy, Scipy Stats, PyPI, GitHub, Colaboratory, Anaconda, VSCode, and Jupyter. Special thanks to Scikit-Learn and Pandas.

References

[1] N. Teague. Automunge. `https://github.com/Automunge/AutoMunge`, 2021.

[2] W. McKinney. Data structures for statistical computing in python. Proceedings of the 9th Python in Science Conference, pages 51–56, 2010.

[3] F. Pedregosa, G. Pedregosa, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. Journal of Machine Learning Research, 12:2825–2830, 2011.

[4] P. Virtanen, R. Gommers, T. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, I. Polat, Y. Feng, E. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. Quintero, C. Harris, A. Archibald, A. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17:261–272, 2020.

[5] S. van der Walt, S. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. Computing in Science & Engineering, 13:22–30, 2011.

[6] H. Wickham. Tidy data. Journal of Statistical Software, 59(10), 2014.

[7] N. Teague. Custom Transformations with Automunge. `https://medium.com/automunge/custom-transformations-with-automunge-ae694c635a7e`, 2021.

[8] N. Teague. Data Structure. `https://medium.com/automunge/data-structure-59e52f141dd6`, 2021.

[9] N. Teague. Specification of Derivations with Automunge. `https://medium.com/automunge/specification-of-derivations-with-automunge-6174ca227184`, 2020.

[10] N. Teague. Numeric Encoding Options with Automunge. `https://medium.com/automunge/a-numbers-game-b68ac261c40d`, 2020.

[11] N. Teague. Parsed Categoric Encodings with Automunge. `https://medium.com/automunge/string-theory-acbd208eb8ca`, 2020.

[12] I. Jolliffe and J. Cadima. Principal component analysis: a review and recent developments. Philos Trans A Math Phys Eng Sci, 374:2065, 2016.

[13] N. Teague. Missing Data Infill with Automunge. `https://medium.com/automunge/missing-data-infill-with-automunge-ec94d6b13433`, 2020.