Embedding Encoder

Release 0.0.4

CPA Ferrere | Data Analytics

Mar 09, 2022

CONTENTS

Turn categorical features to dense vector representations with a familiar scikit-learn compliant API.

CHAPTER

ONE

USER DOCUMENTATION

Brief introduction.

1.1 Overview

Embedding Encoder is a scikit-learn-compliant transformer that converts categorical variables into numeric vector representations. This is achieved by creating a small multilayer perceptron architecture in which each categorical variable is passed through an embedding layer, for which weights are extracted and turned into DataFrame columns.

While the idea is not new (it was popularized after the team that landed in the 3rd place of the Rossmann Kaggle competition used it), and although Python implementations have surfaced over the years, we are not aware of any library that integrates this functionality into scikit-learn.

1.2 Installation and dependencies

Embedding Encoder can be installed with

```
pip install embedding-encoder[tf]
```

Embedding Encoder has the following dependencies

- scikit-learn
- Tensorflow
- numpy
- pandas

Please see notes on non-Tensorflow usage at the end of this readme.

1.3 Documentation

Full documentation including this readme and API reference can be found at RTD.

1.4 Usage

Embedding Encoder works like any scikit-learn transformer, the only difference being that it requires `y` to be passed as it is the neural network's target.

Embedding Encoder will assume that all input columns are categorical and will calculate embeddings for each, unless the `numeric_vars` argument is passed. In that case, numeric variables will be included as an additional input to the neural network but no embeddings will be calculated for them, and they will not be included in the output transformation.

Please note that including numeric variables may reduce the interpretability of the final model as their total influence on the target variable can become difficult to disentangle.

The simplest usage example is

```
from embedding_encoder import EmbeddingEncoder

ee = EmbeddingEncoder(task="regression") # or "classification"
ee.fit(X=X, y=y)
output = ee.transform(X=X)
```

1.5 Compatibility with scikit-learn

Embedding Encoder can be included in pipelines as a regular transformer, and is compatible with cross-validation and hyperparameter optimization.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

from embedding_encoder import EmbeddingEncoder

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

ee = EmbeddingEncoder(task="classification")
num_pipe = make_pipeline(SimpleImputer(strategy="mean"), StandardScaler())
cat_pipe = make_pipeline(SimpleImputer(strategy="most_frequent"), ee)
col_transformer = ColumnTransformer([("num_transformer", num_pipe, numeric_vars),
                                     ("cat_transformer", cat_pipe, categorical_vars)])

pipe = make_pipeline(col_transformer,
                     LogisticRegression())
param_grid = {
    "columntransformer__cat__embeddingencoder__layers_units": [
```

(continues on next page)

(continued from previous page)

```
        [64, 32, 16],
        [16, 8],
    ]
}
cv = GridSearchCV(pipeline, param_grid)
```

In the case of pipelines, if `numeric_vars` is specificed Embedding Encoder has to be the first step in the pipeline. This is because a Embedding Encoder with `numeric_vars` requires that its `X` input be a `DataFrame` with proper column names, which cannot be guaranteed if previous transformations are applied as is.

Alternatively, previous transformations can be included provided they are held inside the `ColumnTransformerWithNames` class in this library, which retains feature names.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

from embedding_encoder import EmbeddingEncoder
from embedding_encoder.utils import ColumnTransformerWithNames

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

ee = EmbeddingEncoder(task="classification", numeric_vars=numeric_vars)
num_pipe = make_pipeline(SimpleImputer(strategy="mean"), StandardScaler())
cat_transformer = SimpleImputer(strategy="most_frequent")
col_transformer = ColumnTransformerWithNames([("num_transformer", num_pipe, numeric_
↪vars),
                                              ("cat_transformer", cat_transformer,␣
↪categorical_vars)])

pipe = make_pipeline(col_transformer,
                     ee,
                     LogisticRegression())
pipe.fit(X_train, y_train)
```

Like scikit transformers, Embedding Encoder also has a `inverse_transform` method that recomposes the original input.

1.6 Plotting embeddings

The idea behind embeddings is that categories that are conceptually similar should have similar vector representations. For example, "December" and "January" should be close to each other when the target variable is ice cream sales (here in the Southern Hemisphere at least!).

This can be analyzed with the `plot_embeddings` function, which depends on Seaborn (`pip install embedding-encoder[sns]` or `pip install embedding-encoder[full]` which includes Tensorflow).

```
from embedding_encoder import EmbeddingEncoder

ee = EmbeddingEncoder(task="classification")
```

(continues on next page)

(continued from previous page)

```
ee.fit(X=X, y=y)
ee.plot_embeddings(variable="...", model="pca")
```

1.7 Advanced usage

Embedding Encoder gives some control over the neural network. In particular, its constructor allows setting how deep and large the network should be (by modifying `layers_units`), as well as the dropout rate between dense layers. Epochs and batch size can also be modified.

These can be optimized with regular scikit-learn hyperparameter optimization techiniques.

The training loop includes an early stopping callback that restores the best weights (by default, the ones that minimize the validation loss).

1.8 Non-Tensorflow usage

Tensorflow can be tricky to install on some systems, which could make Embedding Encoder less appealing if the user has no intention of using TF for modeling.

There are actually two partial ways of using Embedding Encoder without a TF installation.

1. Because TF is only used and imported in the `EmbeddingEncoder.fit()` method, once EE or the pipeline that contains EE has been fit, TF can be safely uninstalled; calls to methods like `EmbeddingEncoder.transform()` or `Pipeline.predict()` should raise no errors.
2. Embedding Encoder can save the mapping from categorical variables to embeddings to a JSON file which can be later imported by setting `pretrained=True`, requiring no TF whatsoever. This also opens up the opportunity to train embeddings for common categorical variables on common tasks and saving them for use in downstream tasks.

Installing EE without Tensorflow is as easy as removing "[tf]" from the install command.

```
pip install embedding-encoder
```

CHAPTER

TWO

API DOCUMENTATION

Or read the API documentation (automatically generated from source code) for the specifics.

2.1 API documentation

2.1.1 EmbeddingEncoder class

class embedding_encoder.core.**EmbeddingEncoder**(*task*, *numeric_vars=None*, *dimensions=None*, *layers_units=None*, *dropout=0.2*, *classif_classes=None*, *classif_loss=None*, *optimizer='adam'*, *epochs=5*, *batch_size=32*, *validation_split=0.2*, *verbose=0*, *mapping_path=None*, *pretrained=False*, *keep_model=False*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.TransformerMixin`

Obtain numeric embeddings from categorical variables.

Embedding Encoder trains a small neural network with categorical inputs passed through embedding layers. Numeric variables can be included as additional inputs by setting `numeric_vars`.

Embedding Encoder returns (unique_values + 1) / 2 vectors per categorical variable, with a minimum of 2 and a maximum of 50. However, this can be changed by passing a list of integers to `dimensions`.

The neural network architecture and training loop can be partially modified. `layers_units` takes an array of integers, each representing an additional dense layer, i.e, *[32, 24, 16]* will create 3 hidden layers with the corresponding units, with dropout layers interleaved, while `dropout` controls the dropout rate.

While Embedding Encoder will try to infer the appropiate number of units for the output layer and the model's loss for classification tasks, these can be set with `classif_classes` and `classif_loss`. Regression tasks will always have 1 unit in the output layer and mean squared error loss.

`optimizer` and `batch_size` are passed directly to Keras.

`validation_split` is also passed to Keras. Setting it to something higher than 0 will use validation loss in order to decide whether to stop training early. Otherwise train loss will be used.

`mapping_path` is the path to a JSON file where the embedding mapping will be saved. If `pretrained` is set to True, the mapping will be loaded from this file and no model will be trained.

Parameters

- **task** (`str`) – "regression" or "classification". This determines the units in the head layer, loss and metrics used.

- **numeric_vars** (Optional[List[str]]) – Array-like of strings containing the names of the numeric variables that will be included as inputs to the network.
- **dimensions** (Optional[List[int]]) – Array-like of integers containing the number of embedding dimensions for each categorical feature. If none, the dimension will be *min(50, int(np.ceil((unique + 1) / 2)))*
- **layers_units** (Optional[List[int]]) – Array-like of integers which define how many dense layers to include and how many units they should have. By default None, which creates two hidden layers with 24 and 12 units.
- **dropout** (float) – Dropout rate used between dense layers.
- **classif_classes** (Optional[int]) – Number of classes in *y* for classification tasks.
- **classif_loss** (*Optional[str], optional*) – Loss function for classification tasks.
- **optimizer** (str) – Optimizer, default "adam".
- **epochs** (int) – Number of epochs, default 3.
- **batch_size** (int) – Batches size, default 32.
- **validation_split** (float) – Passed to Keras *Model.fit*.
- **verbose** (int) – Verbosity of the Keras *Model.fit*, default 0.
- **mapping_path** (Union[str, Path, None]) – Path to a JSON file where the mapping from categorical variables to embeddings will be saved. If pretrained is True, the mapping will be loaded from this file and no model will be trained.
- **pretrained** (bool) – Whether to use pretrained embeddings found in the JSON at mapping_path.
- **keep_model** (bool) – Whether to assign the Tensorflow model to *_model*. Setting to True will prevent the EmbeddingEncoder from being pickled. Default False. Please note that the model's *history* dict is available at *_history*.

_history
Keras *model.history.history* containing training data.

Type *dict*

_model
Keras model. Only available if keep_model is True.

Type *keras.Model*

_embeddings_mapping
Dictionary mapping categorical variables to their embeddings.

Type dict

Raises

- **ValueError** – If *task* is not "regression" or "classification".
- **ValueError** – If *classif_classes* or *classif_loss* are specified for regression tasks.
- **ValueError** – If *classif_classes* is specified but *classif_loss* is not.

Parameters

- **task** (*str*) –
- **numeric_vars** (*Optional[List[str]]*) –

- **dimensions** (*Optional[List[int]]*) –
- **layers_units** (*Optional[List[int]]*) –
- **dropout** (*float*) –
- **classif_classes** (*Optional[int]*) –
- **classif_loss** (*Optional[str]*) –
- **optimizer** (*str*) –
- **epochs** (*int*) –
- **batch_size** (*int*) –
- **validation_split** (*float*) –
- **verbose** (*int*) –
- **mapping_path** (*Optional[Union[str, Path]]*) –
- **pretrained** (*bool*) –
- **keep_model** (*bool*) –

fit(*X*, *y*)

Fit the EmbeddingEncoder to X.

Parameters

- **X** (DataFrame) – The data to process. It can include numeric variables that will not be encoded but will be used in the neural network as additional inputs.
- **y** (Union[DataFrame, Series]) – Target data. Used as target in the neural network.

Returns **self** – Fitted Embedding Encoder.

Return type object

mapping_to_json()

Return type None

mapping_from_json()

Return type Dict[str, DataFrame]

transform(*X*)

Transform X using computed variable embeddings.

Parameters **X** (DataFrame) – The data to process.

Returns Vector embeddings for each categorical variable.

Return type embeddings

inverse_transform(*X*)

Inverse transform X using computed variable embeddings.

Parameters **X** (Union[DataFrame, ndarray]) – The data to process.

Return type Original DataFrame.

get_feature_names_out(*input_features=None*)

get_feature_names(*input_features=None*)

plot_embeddings(*variable*, *model='pca'*)

Create a 2D scatterplot of a variable's embeddings. Each dot represents a category.

Parameters

- **variable** (str) – Variable to plot. Please note that scikit-learn's Pipeline might strip column names.
- **model** (*str, optional*) – Dimensionality reduction model. Either "tsne" or "pca". Default "pca".

Returns Seaborn scatterplot (Matplotlib axes)

Return type matplotlib.axes._subplots.AxesSubplot

Raises

- **ValueError** – If selected variable has less than 3 unique values.
- **ValueError** – If selected model is not "tsne" or "pca".
- **ImportError** – If seaborn is not installed.

2.1.2 Utilities

class embedding_encoder.utils.compose.**ColumnTransformerWithNames**(*transformers*, *, *remainder='drop'*, *sparse_threshold=0.3*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*, *verbose_feature_names_out=True*)

Bases: sklearn.compose._column_transformer.ColumnTransformer

A ColumnTransformer that retains DataFrame column names. Obtained from https://stackoverflow.com/questions/61079602/how-do-i-get-feature-names-using-a-column-transformer/68671424#68671424

get_feature_names()

Get feature names from all transformers.

Returns **feature_names** – Names of the features produced by transform.

Return type List[str]

transform(*X*)

Transform X separately by each transformer, concatenate results.

Parameters **X** (*{array-like, dataframe} of shape (n_samples, n_features)*) – The data to be transformed by subset.

Returns **X_t** – Horizontally stacked results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers. If any result is a sparse matrix, everything will be converted to sparse matrices.

Return type {array-like, sparse matrix} of shape (n_samples, sum_n_components)

fit_transform(*X*, *y=None*)

Fit all transformers, transform the data and concatenate results.

Parameters

- **X** (*{array-like, dataframe} of shape (n_samples, n_features)*) – Input data, of which specified subsets are used to fit the transformers.

- **y** (*array-like of shape (n_samples,), default=None*) – Targets for supervised learning.

Returns **X_t** – Horizontally stacked results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers. If any result is a sparse matrix, everything will be converted to sparse matrices.

Return type {array-like, sparse matrix} of shape (n_samples, sum_n_components)

steps: `List[Any]`

PYTHON MODULE INDEX

e

INDEX

Symbols

C

E

F

G

I

M

P

S

T