DataWig Documentation

Amazon

Mar 10, 2022

Contents

This is the documentation for DataWig, a framework for learning models to impute missing values in tables.

CHAPTER 1

Table of Contents

1.1 Introduction

This is the documentation for DataWig, a framework for learning models to impute missing values in tables.

Details on the underlying model can be found in Biessmann, Salinas et al. 2018.

```
@inproceedings{datawig,
 author = {Biessmann, Felix and Salinas, David and Schelter, Sebastian and Schmidt,␣
↪Philipp and Lange, Dustin},
 title = {"Deep" Learning for Missing Value Imputationin Tables with Non-Numerical␣
↪Data},
 booktitle = {Proceedings of the 27th ACM International Conference on Information and␣
↪Knowledge Management},
 series = {CIKM '18},
 year = {2018},
 isbn = {978-1-4503-6014-2},
 location = {Torino, Italy},
 pages = {2017--2025},
 numpages = {9},
 url = {http://doi.acm.org/10.1145/3269206.3272005},
 doi = {10.1145/3269206.3272005},
 keywords = {data cleaning, missing value imputation}}
```

1.2 User Guide

1.2.1 Step-by-step Examples

Setup

For installing DataWig, follow the installation instructions in the readme.

Examples

In each example, we provide a detailed description of important features along with python code that highlights these features on a public dataset. We recommend reading through the overview of DataWig and then following the below examples in order.

For additional examples and use cases, refer to the unit test cases.

Data

Unless otherwise specified, these examples will make use of the Multimodal Attribute Extraction (MAE) dataset. This dataset contains over 2.2 million products with corresponding attributes, but to make data loading and processing more manageable, we provide a reformatted subset of the validation data (for the *finish* and *color* attributes) as a .csv file.

This data contains columns for *title*, *text*, *finish*, and *color*. The title and text columns contain string data that will be used to impute the finish attribute. Note, the dataset is extremely noisy, but still provides a good example for real-world use cases of DataWig.

To speed up run-time, all examples will use a smaller version of this finish dataset that contains ~5000 samples. Run the following in this directory to download this dataset:

```
wget https://github.com/awslabs/datawig/raw/master/examples/mae_train_dataset.csv
```

To get the complete finish dataset with all data, please check instructions here.

If you'd like to use this data in your own experiments, please remember to cite the original MAE paper:

```
@article{RobertLLogan2017MultimodalAE,
  title={Multimodal Attribute Extraction},
  author={IV RobertL.Logan and Samuel Humeau and Sameer Singh},
  journal={CoRR},
  year={2017},
  volume={abs/1711.11118}
```

Overview of DataWig

Here, we give a brief overview of the internals of DataWig.

ColumnEncoder (*column_encoder.py*)

Defines an abstract super class of column encoders that transforms the raw data of a column (e.g. strings from a product title) into an encoded numerical representation.

There are a few options for ColumnEncoders (subclasses) depending on the column data type:

- `SequentialEncoder`: for sequences of string symbols (e.g. characters or words)
- `BowEncoder`: bag-of-word representation for strings, as sparse vectors
- `CategoricalEncoder`: for categorical variables (one-hot encoding)
- `NumericalEncoder`: for numerical values

Featurizer (*mxnet_input_symbol.py*)

Defines a specific featurizer for data that has been encoded into a numerical format by ColumnEncoder. The Featurizer is used to feed data into the imputation model's computational graph for training and prediction.

There are a few options for Featurizers depending on which ColumnEncoder was used for a particular column:

- `LSTMFeaturizer` maps an input representing a sequence of symbols into a latent vector using an LSTM
- `BowFeaturizer` used with `BowEncoder` on string data
- `EmbeddingFeaturizer` maps encoded catagorical data into a vector representations (word-embeddings)
- `NumericalFeaturizer` extracts features from numerical data using fully connected layers

SimpleImputer (*simple_imputer.py*)

Using `SimpleImputer` is the easiest way to deploy an imputation model on your dataset with DataWig. As the name suggests, the `SimpleImputer` is straightforward to call from a python script and uses default encoders and featurizers that usually yield good results on a variety of datasets.

Imputer (*imputer.py*)

`Imputer` is the backbone of the `SimpleImputer` and is responsible for running the preprocessing code, creating the model, executing training, and making predictions. Using the `Imputer` enables more flexibility with specifying model parameters, such as using particular encoders and featurizers rather than the default ones that `SimpleImputer` uses.

1.2.2 Introduction to `SimpleImputer`

This tutorial will teach you the basics of how to use `SimpleImputer` for your data imputation tasks. As an advanced feature the SimpleImputer supports label-shift detection and correction which is described in *Label Shift and Empirical Risk Minimization*. For now, we will use a subset of the MAE data as an example. To download this data, please refer to the previous section.

Open the SimpleImputer intro in this directory to see the code used in this tutorial.

Load Data

First, let's load the data into a pandas DataFrame and split the data into train (80%) and test (20%) subsets.

```
df = pd.read_csv('../finish_val_data_sample.csv')
df_train, df_test = random_split(df, split_ratios=[0.8, 0.2])
```

Note, the `random_split()` method is provided in `datawig.utils`. The validation set is partitioned from the train data during training and defaults to 10%.

Default `SimpleImputer`

At the most basic level, you can run the `SimpleImputer` on data without specifying any additional arguments. This will automatically choose the right `ColumnEncoder` and `Featurizer` for each column and train an imputation model with default hyperparameters.

To train a model, you can simply initialize a `SimpleImputer`, specifying the input columns containing useful data for imputation, the output column that you'd like to impute values for, and the output path, which will store model data and metrics. Then, you can use the `fit()` method to train the model.

```
#Initialize a SimpleImputer model
imputer = SimpleImputer(
    input_columns=['title', 'text'],
    output_column='finish',
    output_path = 'imputer_model'
)

#Fit an imputer model on the train data
imputer.fit(train_df=df_train)
```

From here, you can this model to make predictions on the test set and return the original dataframe with an additional column containing the model's predictions.

```
predictions = imputer.predict(df_test)
```

Finally, you can determine useful metrics to gauge how well the model's predictions compare to the true values (using `sklearn.metrics`).

```
#Calculate f1 score
f1 = f1_score(predictions['finish'], predictions['finish_imputed'])

#Print overall classification report
print(classification_report(predictions['finish'], predictions['finish_imputed']))
```

HPO with `SimpleImputer`

DataWig also enables hyperparameter optimization to find the best model on a particular dataset.

The steps for training a model with HPO are identical to the default `SimpleImputer`.

```
imputer = SimpleImputer(
    input_columns=['title', 'text'],
    output_column='finish',
    output_path='imputer_model'
)

# fit an imputer model with customized hyperparameters
imputer.fit_hpo(train_df=df_train)
```

Calling HPO like this will search through some basic and usually helpful hyperparameter choices. There are two ways for a more detailed search. Firstly, `fit_hpo` offers additional arguments that can be inspected in the SimpleImputer. For even more configurations and variation of hyperparameters for the various input column types, a dictionary with ranges can be passed to `fit_hpo` as can be seen in the hpo-code. Results for any HPO run can be accessed under `imputer.hpo.results` and the model from any HPO run can then be loaded using `imputer.load_hpo_model(idx)` passing the model index.

Load Saved Model

Once a model is trained, it will be saved in the location of `output_path`, which you specified as an argument when intializing the `SimpleImputer`. You can easily load this model for further experiments or run on new datasets as follows.

```
#Load saved model
imputer = SimpleImputer.load('./imputer_model')
```

This model also contains the associated metrics (stored as a dictionary) calculated on the validation set during training.

```
#Load metrics from the validation set
metrics = imputer.load_metrics()
weighted_f1 = metrics['weighted_f1']
avg_precision = metrics['avg_precision']
# ...
```

1.2.3 Introduction to Imputer

This tutorial will teach you the basics of how to use the `Imputer` for your data imputation tasks. We will use a subset of the MAE data as an example. To download this data, please refer to README.

Open Imputer intro to see the code used in this tutorial.

Load Data

First, let's load the data into a pandas DataFrame and split the data into train (80%) and test (20%) subsets.

```
df = pd.read_csv('../finish_val_data_sample.csv')
df_train, df_test = random_split(df, split_ratios=[0.8, 0.2])
```

Note, the `random_split()` method is provided in `datawig.utils`. The validation set is partitioned from the train data during training and defaults to 10%.

Default `Imputer`

The key difference with the `Imputer` is specifying the Encoders and Featurizers used for particular columns in your dataset. Once this is done, initializing the model, training, and making predictions with the Imputer is similar to the `SimpleImputer`

```
#Specify encoders and featurizers
data_encoder_cols = [BowEncoder('title'), BowEncoder('text')]
label_encoder_cols = [CategoricalEncoder('finish')]
data_featurizer_cols = [BowFeaturizer('title'), BowFeaturizer('text')]

imputer = Imputer(
    data_featurizers=data_featurizer_cols,
    label_encoders=label_encoder_cols,
    data_encoders=data_encoder_cols,
    output_path='imputer_model'
)

imputer.fit(train_df=df_train)
predictions = imputer.predict(df_test)
```

For the input columns that contain data useful for imputation, the `Imputer` expects you to specify the particular encoders and featurizers. For the label column that your are trying to impute, only specifying the type of encoder is necessary.

Using Different Encoders and Featurizers

One of the key advantages with the `Imputer` is that you get flexibility for customizing exactly which encoders and featurizers to use, which is something you can't do with the `SimpleImputer`.

For example, let's say you wanted to use an LSTM rather than the default bag-of-words text model that the `SimpleImputer` uses. To do this, you can simply specificy the proper encoders and featurizers to initialize the `Imputer` model.

```
#Using LSTMs instead of bag-of-words
data_encoder_cols = [SequentialEncoder('title'), SequentialEncoder('text')]
label_encoder_cols = [CategoricalEncoder('finish')]
data_featurizer_cols = [LSTMFeaturizer('title'), LSTMFeaturizer('text')]

imputer = Imputer(
    data_featurizers=data_featurizer_cols,
    label_encoders=label_encoder_cols,
    data_encoders=data_encoder_cols,
    output_path='imputer_model'
)
```

Prediction with Probabilities

Beyond directly predicting values, the `Imputer` can also return the probabilities for each class on ever sample (numpy array of shape samples-by-labels). This can help with understanding what the model is predicting and with what probability for each sample.

```
prob_dict = imputer.predict_proba(df_test)
```

In addition, you can get the probabilities only for the top-k most likely predicted classes (rather than for all the classes above).

```
prob_dict_topk = imputer.predict_proba_top_k(df_test, top_k=5)
```

Get Predictions and Metrics

To get predictions (original dataframe with an extra column) and the associated metrics from the validation set during training, you can run the following:

```
predictions, metrics = imputer.transform_and_compute_metrics(df_test)
```

1.2.4 Parameters for Different Data Types

This tutorial will highlight the different parameters associated with column data types supported by DataWig. We use the `SimpleImputer` in these examples, but the same concepts apply when using the `Imputer` and other encoders/featurizers.

The parameter tutorial contains the complete code for training models on text and numerical data. Here, we illustrate examples of relevant parameters for training models on each of these types of data.

It's important to note that your dataset can contain columns with mixed types. The `SimpleImputer` automatically determines which encoder and featurizer to use when training an imputation model!

Text Data

The key parameters associated with text data are:

- `num_hash_buckets` dimensionality of the vector for bag-of-words
- `tokens` type of tokenization used for text data (default: chars)

Here is an example of using these parameters:

```
imputer_text.fit_hpo(
    train_df=df_train,
    num_epochs=50,
    learning_rate_candidates=[1e-3, 1e-4],
    final_fc_hidden_units_candidates=[[100]],
    num_hash_bucket_candidates=[2**10, 2**15],
    tokens_candidates=['chars', 'words']
)
```

Apart from the text parameters, `final_fc_hidden_units` corresponds to a list containing the dimensionality of the fully connected layer after all column features are concatenated. The length of this list is the number of hidden fully connected layers.

Numerical Data

The key parameters associated with numerical data are:

- `latent_dim` dimensionality of the fully connected layers for creating a feature vector from numerical data
- `hidden_layers` number of fully connected layers

Here is an example of using these parameters:

```
imputer_numeric.fit_hpo(
    train_df=df_train,
    num_epochs=50,
    learning_rate_candidates=[1e-3, 1e-4],
    latent_dim_candidates=[50, 100],
    hidden_layers_candidates=[0, 2],
    final_fc_hidden_units=[[100]]
)
```

In this case, the model will use a fully connected layer size of 50 or 100, with 0 or 2 hidden layers.

1.2.5 Advanced Features

Label Shift and Empirical Risk Minimization

The SimpleImputer implements the method described by Lipton, Wang and Smola to detect and fix label shift for categorical outputs. Label shift occurs when the marginal distribution differs between the training and production setting. For instance, we might be interested in imputing the color of T-Shirts from their free-text description. Let's assume that the training data consists only of women's T-Shirts while the production data consists only of Men's T-Shirts. Then the marginal distribution of colors, p(color), is likely different while the conditional, p(description | color) may be unchanged. This is a scenario where datawig can detect and fix the shift.

Upon training a SimpleImputer, we can detect shift by calling:

```
weights = imputer.check_for_label_shift(production_data)
```

Note, that `production_data` needs to have all the relevant input columns but does not have labels. This call will log the severity of the shift and further information, as follows.

```
The estimated true label marginals are [('black', 0.62), ('white', 0.38)]
Marginals in the training data are [('black', 0.23), ('white', 0.77)]
Reweighing factors for empirical risk minimization{'label_0': 2.72, 'label_1': 0.49}
The smallest eigenvalue of the confusion matrix is 0.21 ' (needs to be > 0).
```

To fix the shift, the reweighing factors are most important. They are returned as dictionary where each key is a label and the value is the corresponding weight by which any observation's contribution to the log-likelihood must be multiplied to minimize the empirical risk. To correct the shift we need to retrain the model with a weighted likelihood which can easily be achieved by passing the weight dictionary to the `fit()` method.

```
simple_imputer.fit(train_df, class_weights=weights)
```

The resulting model will generally have improved performance on the production_data, if there was a label shift present and if the original classifier performed reasonably well. For further assumptions see the above cited paper. Note, that in extreme cases such as very high label noise, this method can lead to a decreased model performance.

Reweighing the likelihood can be useful for reasons other than label-shift. For instance we may trust certain observations more than others and wish to up-weigh their impact on the model parameters. To this end, weights can also be passed on an instance level as list with an entry for every row in the training data, for instance:

```
simple_imputer.fit(train_df, class_weights=[1, 1, 1, 2, 1, 1, 1, ...])
```

1.3 Contributing to DataWig

Please see how to contribute.

1.4 API

1.4.1 Simple Imputer

DataWig SimpleImputer: Uses some simple default encoders and featurizers that usually yield decent imputation quality

class datawig.simple_imputer.**SimpleImputer**(*input_columns: List[str], output_column: str, output_path: str = '', num_hash_buckets: int = 32768, num_labels: int = 100, tokens: str = 'chars', numeric_latent_dim: int = 100, numeric_hidden_layers: int = 1, is_explainable: bool = False*)

SimpleImputer model based on n-grams of concatenated strings of input columns and concatenated numerical features, if provided.

Given a data frame with string columns, a model is trained to predict observed values in label column using values observed in other columns.

The model can then be used to impute missing values.

Parameters

- **`input_columns`** – list of input column names (as strings)
- **`output_column`** – output column name (as string)
- **`output_path`** – path to store model and metrics
- **`num_hash_buckets`** – number of hash buckets used for the n-gram hashing vectorizer, only used for non-numerical input columns, ignored otherwise
- **`num_labels`** – number of imputable values considered after, only used for non-numerical input columns, ignored otherwise
- **`tokens`** – string, 'chars' or 'words' (default 'chars'), determines tokenization strategy for n-grams, only used for non-numerical input columns, ignored otherwise
- **`numeric_latent_dim`** – int, number of latent dimensions for hidden layer of NumericalFeaturizers; only used for numerical input columns, ignored otherwise
- **`numeric_hidden_layers`** – number of numeric hidden layers
- **`is_explainable`** – if this is True, a stateful tf-idf encoder is used that allows explaining classes and single instances

Example usage:

from datawig.simple_imputer import SimpleImputer import pandas as pd

fn_train = os.path.join(datawig_test_path, "resources", "shoes", "train.csv.gz") fn_test = os.path.join(datawig_test_path, "resources", "shoes", "test.csv.gz")

df_train = pd.read_csv(training_data_files) df_test = pd.read_csv(testing_data_files)

output_path = "imputer_model"

# set up imputer model imputer = SimpleImputer(input_columns=['item_name', 'bullet_point'], output_column='brand')

# train the imputer model imputer = imputer.fit(df_train)

# obtain imputations imputations = imputer.predict(df_test)

`check_data_types`(*data_frame: pandas.core.frame.DataFrame*) → None

Checks whether a column contains string or numeric data

Parameters **`data_frame`** –

Returns

`check_for_label_shift`(*target_data: pandas.core.frame.DataFrame*) → dict

Detect label shift in the validation data

Parameters

- **`test_data`** – data frame that contains labels
- **`target_data`** – unlabelled data for which predictions are to be generated

Returns dictionary with labels as keys and weights as values.

`static complete`(*data_frame: pandas.core.frame.DataFrame*, *precision_threshold: float = 0.0*, *inplace: bool = False*, *hpo: bool = False*, *verbose: int = 0*, *num_epochs: int = 100*, *iterations: int = 1*, *output_path: str = '.'*)

Given a dataframe with missing values, this function detects all imputable columns, trains an imputation model on all other columns and imputes values for each missing value. Several imputation iterators can be run. Imputable columns are either numeric columns or non-numeric categorical columns; for determining whether a column is categorical (as opposed to a plain text column) we use the following heuristic: a non-numeric categorical column should have least 10 times as many rows as there were unique values

If an imputation model did not reach the precision specified in the precision_threshold parameter for a given
imputation value, that value will not be imputed; thus depending on the precision_threshold, the returned dataframe can still contain some missing values.

For numeric columns, we do not filter for accuracy. :param data_frame: original dataframe :param precision_threshold: precision threshold for categorical imputations (default: 0.0) :param inplace: whether or not to perform imputations inplace (default: False) :param hpo: whether or not to perform hyperparameter optimization (default: False) :param verbose: verbosity level, values > 0 log to stdout (default: 0) :param num_epochs: number of epochs for each imputation model training (default: 100) :param iterations: number of iterations for iterative imputation (default: 1) :param output_path: path to store model and metrics :return: dataframe with imputations

`explain`(*label: str*, *k: int = 10*, *label_column: str = None*) → dict
Return dictionary with a list of tuples for each explainable input column. Each tuple denotes one of the top k features with highest correlation to the label.

Parameters

- **`label`** – label value to explain
- **`k`** – number of explanations for each input encoder to return. If not given, return top 10 explanations.
- **`label_column`** – name of label column to be explained (optional, defaults to the first available column.)

`explain_instance`(*instance: pandas.core.series.Series*, *k: int = 10*, *label_column: str = None*, *label: str = None*) → dict
Return dictionary with list of tuples for each explainable input column of the given instance. Each entry shows the most highly correlated features to the given label (or the top predicted label of not provided).

Parameters

- **`instance`** – row of data frame (or dictionary)
- **`k`** – number of explanations (ngrams) for text inputs
- **`label_column`** – name of label column to be explained (optional)
- **`label`** – explain why instance is classified as label, otherwise explain top-label per input

`fit`(*train_df: pandas.core.frame.DataFrame*, *test_df: pandas.core.frame.DataFrame = None*, *ctx: <module 'mxnet.context' from '/home/docs/checkouts/readthedocs.org/user_builds/datawig/envs/latest/lib/python3.7/site-packages/mxnet/context.py'> = [cpu(0)]*, *learning_rate: float = 0.004*, *num_epochs: int = 100*, *patience: int = 5*, *test_split: float = 0.1*, *weight_decay: float = 0.0*, *batch_size: int = 16*, *final_fc_hidden_units: List[int] = None*, *calibrate: bool = True*, *class_weights: dict = None*, *instance_weights: list = None*) → Any
Trains and stores imputer model

Parameters

- **`train_df`** – training data as dataframe
- **`test_df`** – test data as dataframe; if not provided, a ratio of test_split of the training data are used as test data
- **`ctx`** – List of mxnet contexts (if no gpu's available, defaults to [mx.cpu()]) User can also pass in a list gpus to be used, ex. [mx.gpu(0), mx.gpu(2), mx.gpu(4)]
- **`learning_rate`** – learning rate for stochastic gradient descent (default 4e-4)

- **`num_epochs`** – maximal number of training epochs (default 10)
- **`patience`** – used for early stopping; after [patience] epochs with no improvement, training is stopped. (default 3)
- **`test_split`** – if no test_df is provided this is the ratio of test data to be held separate for determining model convergence
- **`weight_decay`** – regularizer (default 0)

:param batch_size (default 16) :param final_fc_hidden_units: list dimensions for FC layers after the final concatenation :param calibrate: Control automatic model calibration :param class_weights: Dictionary with labels as keys and weights as values.

Weighs each instance's contribution to the likelihood based on the corresponding class.

Parameters **`instance_weights`** – List of weights for each instance in train_df.

`fit_hpo`(*train_df: pandas.core.frame.DataFrame, test_df: pandas.core.frame.DataFrame = None, hps: dict = None, num_evals: int = 10, max_running_hours: float = 96.0, hpo_run_name: str = None, user_defined_scores: list = None, num_epochs: int = None, patience: int = None, test_split: float = 0.2, weight_decay: List[float] = None, batch_size: int = 16, num_hash_bucket_candidates: List[float] = [4096, 32768, 262144], tokens_candidates: List[str] = ['words', 'chars'], numeric_latent_dim_candidates: List[int] = None, numeric_hidden_layers_candidates: List[int] = None, final_fc_hidden_units: List[List[int]] = None, learning_rate_candidates: List[float] = None, normalize_numeric: bool = True, hpo_max_train_samples: int = None, ctx: <module 'mxnet.context' from '/home/docs/checkouts/readthedocs.org/user_builds/datawig/envs/latest/lib/python3.7/site-packages/mxnet/context.py'> = [cpu(0)]*) → Any

Fits an imputer model with hyperparameter optimization. The parameter ranges are searched randomly.

Grids are specified using the *_candidates arguments (old) or with more flexibility via the dictionary hps.

Parameters

- **`train_df`** – training data as dataframe
- **`test_df`** – test data as dataframe; if not provided, a ratio of test_split of the training data are used as test data
- **`hps`** – nested dictionary where hps[global][parameter_name] is list of parameters. Similarly, hps[column_name][parameter_name] is a list of parameter values for each input column. Further, hps[column_name]['type'] is in ['numeric', 'categorical', 'string'] and is inferred if not provided.
- **`num_evals`** – number of evaluations for random search
- **`max_running_hours`** – Time before the hpo run is terminated in hours.
- **`hpo_run_name`** – string to identify the current hpo run.
- **`user_defined_scores`** – list with entries (Callable, str), where callable is a function accepting **kwargs true, predicted, confidence. Allows custom scoring functions.

Below are parameters of the old implementation, kept to ascertain backwards compatibility. :param num_epochs: maximal number of training epochs (default 10) :param patience: used for early stopping; after [patience] epochs with no improvement,

training is stopped. (default 3)

Parameters

- **`test_split`** – if no test_df is provided this is the ratio of test data to be held separate for determining model convergence
- **`weight_decay`** – regularizer (default 0)

:param batch_size (default 16) :param num_hash_bucket_candidates: candidates for gridsearch hyperparameter

optimization (default [2**10, 2**13, 2**15, 2**18, 2**20])

Parameters

- **`tokens_candidates`** – candidates for tokenization (default ['words', 'chars'])
- **`numeric_latent_dim_candidates`** – candidates for latent dimensionality of numerical features (default [10, 50, 100])
- **`numeric_hidden_layers_candidates`** – candidates for number of hidden layers of
- **`final_fc_hidden_units`** – list of lists w/ dimensions for FC layers after the final concatenation (NOTE: for HPO, this expects a list of lists)
- **`learning_rate_candidates`** – learning rate for stochastic gradient descent (default 4e-4) numerical features (default [0, 1, 2])
- **`learning_rate_candidates`** – candidates for learning rate (default [1e-1, 1e-2, 1e-3])
- **`normalize_numeric`** – boolean indicating whether or not to normalize numeric values
- **`hpo_max_train_samples`** – training set size for hyperparameter optimization. use is deprecated.
- **`ctx`** – List of mxnet contexts (if no gpu's available, defaults to [mx.cpu()]) User can also pass in a list gpus to be used, ex. [mx.gpu(0), mx.gpu(2), mx.gpu(4)] This parameter is deprecated.

Returns pd.DataFrame with with hyper-parameter configurations and results

`static load`(*output_path: str*) → Any

Loads model from output path

Parameters **`output_path`** – output_path field of trained SimpleImputer model

Returns SimpleImputer model

`load_hpo_model`(*hpo_name: int = None*)

Load model after hyperparameter optimisation has ran. Overwrites local artifacts of self.imputer.

Parameters **`hpo_name`** – Index of the model to be loaded. Default, load model with highest weighted precision or mean squared error.

Returns imputer object

`load_metrics`() → Dict[str, Any]

Loads various metrics of the internal imputer model, returned as dictionary :return: Dict[str,Any]

`predict`(*data_frame: pandas.core.frame.DataFrame, precision_threshold: float = 0.0, imputation_suffix: str = '_imputed', score_suffix: str = '_imputed_proba', inplace: bool = False*)

Imputes most likely value if it is above a certain precision threshold determined on the validation set

Precision is calculated as part of the *datawig.evaluate_and_persist_metrics* function.

Returns original dataframe with imputations and respective likelihoods as estimated by imputation model; in additional columns; names of imputation columns are that of the label suffixed with *imputation_suffix*, names of respective likelihood columns are suffixed with *score_suffix*

Parameters

- **`data_frame`** – data frame (pandas)
- **`precision_threshold`** – double between 0 and 1 indicating precision threshold
- **`imputation_suffix`** – suffix for imputation columns
- **`score_suffix`** – suffix for imputation score columns
- **`inplace`** – add column with imputed values and column with confidence scores to data_frame, returns the modified object (True). Create copy of data_frame with additional columns, leave input unmodified (False).

Returns data_frame original dataframe with imputations and likelihood in additional column

`save`()

Saves model to disk; mxnet module and imputer are stored separately

1.4.2 Imputer

DataWig Imputer: Imputes missing values in tables

class `datawig.imputer.`**`Imputer`**(*data_encoders: List[datawig.column_encoders.ColumnEncoder], data_featurizers: List[datawig.mxnet_input_symbols.Featurizer], label_encoders: List[datawig.column_encoders.ColumnEncoder], output_path=”*)

Imputer model based on deep learning trained with MxNet

Given a data frame with string columns, a model is trained to predict observed values in one or more column using values observed in other columns. The model can then be used to impute missing values.

Parameters

- **`data_encoders`** – list of datawig.mxnet_input_symbol.ColumnEncoders, output_column name must match field_name of data_featurizers
- **`data_featurizers`** – list of Featurizer;
- **`label_encoders`** – list of CategoricalEncoder or NumericalEncoder
- **`output_path`** – path to store model and metrics

`calibrate`(*test_iter: datawig.iterators.ImputerIterDf*)

Cecks model calibration and fits temperature scaling. If the fit improves model calibration, the temperature parameter is assigned as property to self and used for all further predictions in self.predict_mxnet_iter(). Saves calibration information to dictionary.

Parameters **`test_iter`** – iterator, see ImputerIter in iterators.py

Returns None

`explain`(*label: str*, *k: int = 10*, *label_column: str = None*) → dict

Return dictionary with a list of tuples for each explainable input column. Each tuple denotes one of the top k features with highest correlation to the label.

Parameters

- **`label`** – label value to explain
- **`k`** – number of explanations for each input encoder to return. If not given, return top 10 explanations.
- **`label_column`** – name of label column to be explained (optional, defaults to the first available column.)

`explain_instance`(*instance: pandas.core.series.Series*, *k: int = 10*, *label_column: str = None*, *label: str = None*) → dict

Return dictionary with list of tuples for each explainable input column of the given instance. Each entry shows the most highly correlated features to the given label (or the top predicted label of not provided).

Parameters

- **`instance`** – row of data frame (or dictionary)
- **`k`** – number of explanations (ngrams) for text inputs
- **`label_column`** – name of label column to be explained (optional)
- **`label`** – explain why instance is classified as label, otherwise explain top-label per input

`fit`(*train_df: pandas.core.frame.DataFrame, test_df: pandas.core.frame.DataFrame = None, ctx: <module 'mxnet.context' from '/home/docs/checkouts/readthedocs.org/user_builds/datawig/envs/latest/lib/python3.7/site-packages/mxnet/context.py'> = [cpu(0)], learning_rate: float = 0.001, num_epochs: int = 100, patience: int = 3, test_split: float = 0.1, weight_decay: float = 0.0, batch_size: int = 16, final_fc_hidden_units: List[int] = None, calibrate: bool = True*)

Trains and stores imputer model

Parameters

- **`train_df`** – training data as dataframe
- **`test_df`** – test data as dataframe; if not provided, [test_split] % of the training data are used as test data
- **`ctx`** – List of mxnet contexts (if no gpu's available, defaults to [mx.cpu()]) User can also pass in a list gpus to be used, ex. [mx.gpu(0), mx.gpu(2), mx.gpu(4)]
- **`learning_rate`** – learning rate for stochastic gradient descent (default 1e-4)
- **`num_epochs`** – maximal number of training epochs (default 100)
- **`patience`** – used for early stopping; after [patience] epochs with no improvement, training is stopped. (default 3)
- **`test_split`** – if no test_df is provided this is the ratio of test data to be held separate for determining model convergence
- **`weight_decay`** – regularizer (default 0)
- **`batch_size`** – default 16
- **`final_fc_hidden_units`** – list of dimensions for the final fully connected layer.
- **`calibrate`** – whether to calibrate predictions

Returns trained imputer model

`static load`(*output_path: str*) → Any

Loads model from output path

Parameters **`output_path`** – output_path field of trained Imputer model

Returns imputer model

predict(*data_frame: pandas.core.frame.DataFrame*, *precision_threshold: float = 0.0*, *imputation_suffix: str = '_imputed'*, *score_suffix: str = '_imputed_proba'*, *inplace: bool = False*) → pandas.core.frame.DataFrame

Computes imputations for numerical or categorical values

For categorical imputations, most likely values are imputed if values are above a certain precision threshold computed on the validation set Precision is calculated as part of the *datawig.evaluate_and_persist_metrics* function.

For numerical imputations, no thresholding is applied.

Returns original dataframe with imputations and respective likelihoods as estimated by imputation model in additional columns; names of imputation columns are that of the label suffixed with *imputation_suffix*, names of respective likelihood columns are suffixed with *score_suffix*

Parameters

- **data_frame** – pandas data_frame
- **precision_threshold** – double between 0 and 1 indicating precision threshold for each imputation
- **imputation_suffix** – suffix for imputation columns
- **score_suffix** – suffix for imputation score columns
- **inplace** – add column with imputed values and column with confidence scores to data_frame, returns the modified object (True). Create copy of data_frame with additional columns, leave input unmodified (False).

Returns dataframe with imputations and their likelihoods in additional columns

predict_above_precision(*data_frame: pandas.core.frame.DataFrame*, *precision_threshold=0.95*) → dict

Returns the probabilities for each class, filtering out predictions below the precision threshold.

Parameters

- **data_frame** – data frame
- **precision_threshold** – don't predict if predicted class probability is below this precision threshold

Returns dict of {'column_name': array}, array is a numpy array of shape samples-by-labels

predict_proba(*data_frame: pandas.core.frame.DataFrame*) → dict

Returns the probabilities for each class :param data_frame: data frame :return: dict of {'column_name': array}, array is a numpy array of shape samples-by-labels

predict_proba_top_k(*data_frame: pandas.core.frame.DataFrame*, *top_k: int = 5*) → dict

Returns tuples of (label, probability) for the top_k most likely predicted classes

Parameters

- **data_frame** – pandas data frame
- **top_k** – number of most likely predictions to return

Returns dict of {'column_name': list} where list is a list of (label, probability) tuples

save()

Saves model to disk, except mxnet module which is stored separately during fit

transform(*data_frame: pandas.core.frame.DataFrame*) → dict

Imputes values given an mxnet iterator (see iterators) :param data_frame: pandas data frame (pandas) :return: dict of {'column_name': list} where list contains the string predictions

transform_and_compute_metrics(*data_frame: pandas.core.frame.DataFrame*, *metrics_path=None*) → dict
Returns predictions and metrics (average and per class)

Parameters

- **data_frame** – data frame
- **metrics_path** – if not None and exists, metrics are serialized as json to this path.

Returns

1.4.3 Column Encoders

Column Encoders: used for translating values of a table into numerical representation such that Featurizers can operate on them

class datawig.column_encoders.**BowEncoder**(*input_columns: Any*, *output_column: str = None*, *max_tokens: int = 262144*, *tokens: str = 'chars'*, *ngram_range: tuple = None*, *prefixed_concatenation: bool = True*)
Bag-of-Words encoder for text data, using sklearn's HashingVectorizer

Parameters

- **input_columns** – List[str] with column names to be used as input for this ColumnEncoder
- **output_column** – Name of output field, used as field name in downstream MxNet iterator
- **max_tokens** – Number of hash buckets (dimensionality of sparse ngram vector). default 2**18
- **tokens** – How to tokenize the input data, supports 'words' and 'chars'.
- **ngram_range** – length of ngrams to use as features
- **prefixed_concatenation** – whether or not to prefix values with column name before concat

decode(*col: pandas.core.series.Series*) → pandas.core.series.Series
Raises NotImplementedError, hashed bag-of-words cannot be decoded due to hash collisions

Parameters **token_index_sequence** –

Returns

fit(*data_frame: pandas.core.frame.DataFrame*)
Does nothing, HashingVectorizers do not need to be fit.

Parameters **data_frame** –

Returns

is_fitted() → bool
Returns true if the column encoder does not require fitting (anymore or at all)

Parameters **self** –

Returns True if the encoder is fitted

transform(*data_frame: pandas.core.frame.DataFrame*) → numpy.core.multiarray.array
Transforms one or more string columns into Bag-of-words vectors, hashed into a max_features dimensional feature space. Nans and missing values will be replaced by zero vectors.

Parameters **data_frame** – pandas DataFrame with text columns

Returns numpy array (rows by max_features)

class datawig.column_encoders.**CategoricalEncoder**(*input_columns: Any*, *output_column: str = None*, *token_to_idx: Dict[str, int] = None*, *max_tokens: int = 10000*)
Transforms categorical variable from string representation into number

Parameters

- **input_columns** – List[str] with column names to be used as input for this ColumnEncoder
- **output_column** – Name of output field, used as field name in downstream MxNet iterator
- **token_to_idx** – token to index mapping, 0 is reserved for missing tokens, 1 ... max_tokens for most to least frequent tokens
- **max_tokens** – maximum number of tokens

decode(*col: pandas.core.series.Series*) → pandas.core.series.Series
Decodes a pandas Series of token indices

Parameters **col** – pandas Series of token indices

Returns pandas Series of tokens

decode_token(*token_idx: int*) → str
Decodes a token index into a token

Parameters **token_idx** – token index

Returns token

fit(*data_frame: pandas.core.frame.DataFrame*)
Fits a CategoricalEncoder by extracting the value histogram of a column and capping it at max_tokens. Issues warning if less than 100 values were observed.

Parameters **data_frame** – pandas data frame

is_fitted()
Checks if ColumnEncoder (still) needs to be fitted to data

Returns True if the column encoder does not require fitting (anymore or at all)

transform(*data_frame: pandas.core.frame.DataFrame*) → numpy.core.multiarray.array
Transforms string column of pandas dataframe into categoricals

Parameters **data_frame** – pandas data frame

Returns numpy array (rows by 1)

static transform_func_categorical(*col: pandas.core.series.Series, token_to_idx: Dict[str, int], missing_token_idx: int*) → Any
Transforms categorical values into their indices

Parameters

- **col** – pandas Series with categorical values

- **token_to_idx** – Dict[str, int] with mapping from token to token index
- **missing_token_idx** – index for missing symbol

Returns

class datawig.column_encoders.**ColumnEncoder**(*input_columns: List[str], output_column=None, output_dim=1*)

Abstract super class of column encoders. Transforms value representation of columns (e.g. strings) into numerical representations to be fed into MxNet.

Options for ColumnEncoders are:

SequentialEncoder: for sequences of symbols (e.g. characters or words), BowEncoder: bag-of-word representation, as sparse vectors CategoricalEncoder: for categorical variables NumericalEncoder: for numerical values

Parameters

- **input_columns** – List[str] with column names to be used as input for this ColumnEncoder
- **output_column** – Name of output field, used as field name in downstream MxNet iterator
- **output_dim** – dimensionality of encoded column values (1 for categorical, vocabulary size for sequential and BoW)

decode(*col: pandas.core.series.Series*) → pandas.core.series.Series

Decodes a pandas Series of token indices

Parameters **col** – pandas Series of token indices

Returns pandas Series of tokens

fit(*data_frame: pandas.core.frame.DataFrame*)

Fits a ColumnEncoder if needed (i.e. vocabulary/alphabet)

Parameters **data_frame** – pandas DataFrame

Returns

is_fitted()

Checks if ColumnEncoder (still) needs to be fitted to data

Returns True if the column encoder does not require fitting (anymore or at all)

transform(*data_frame: pandas.core.frame.DataFrame*) → numpy.core.multiarray.array

Transforms values in one or more columns of DataFrame into a numpy array that can be fed into a Featurizer

Parameters **data_frame** –

Returns List of integers

exception datawig.column_encoders.**NotFittedError**

Error thrown when unfitted encoder is used

class datawig.column_encoders.**NumericalEncoder**(*input_columns: Any, output_column: str = None, normalize=True*)

Numerical encoder, concatenates columns in field_names into one vector fills nans with the mean of a column

Parameters

- **`input_columns`** – List[str] with column names to be used as input for this ColumnEncoder
- **`output_column`** – Name of output field, used as field name in downstream MxNet iterator
- **`normalize`** – whether to normalize by the standard deviation or not, default True

`decode`(*col: pandas.core.series.Series*) → pandas.core.series.Series
Undoes the normalization, scales by scale and adds the mean

Parameters **`col`** – pandas Series (normalized)

Returns pandas Series (unnormalized)

`fit`(*data_frame: pandas.core.frame.DataFrame*)
Does nothing or fits the normalizer, if normalization is specified

Parameters **`data_frame`** – DataFrame with numerical columns specified when instantiating NumericalEncoder

`is_fitted`()
Returns true if the column encoder does not require fitting (anymore or at all)

Parameters **`self`** –

Returns True if the encoder is fitted

`transform`(*data_frame: pandas.core.frame.DataFrame*) → numpy.core.multiarray.array
Concatenates the numerical columns specified when instantiating the NumericalEncoder Normalizes features if specified in the NumericalEncoder

Parameters **`data_frame`** – DataFrame with numerical columns specified in NumericalEncoder

Returns np.array with numerical features (rows by number of numerical columns)

`class` `datawig.column_encoders.`**`SequentialEncoder`**(*input_columns: Any*, *output_column: str = None*, *token_to_idx: Dict[str, int] = None*, *max_tokens: int = 1000*, *seq_len: int = 500*)
Transforms sequence of characters into sequence of numbers

Parameters

- **`input_columns`** – List[str] with column names to be used as input for this ColumnEncoder
- **`output_column`** – Name of output field, used as field name in downstream MxNet iterator
- **`token_to_idx`** – token to index mapping 0 is reserved for missing tokens, 1 ... max_tokens-1 for most to least frequent tokens
- **`max_tokens`** – maximum number of tokens
- **`seq_len`** – length of sequence, shorter sequences get padded to, longer sequences truncated at seq_len symbols

`decode`(*col: pandas.core.series.Series*) → pandas.core.series.Series
Decodes a pandas Series of token indices

Parameters **`col`** – pandas Series of token index iterables

Returns pd.Series of strings

decode_seq(*token_index_sequence: Iterable[int]*) → str
Decodes a sequence of token indices into a string

Parameters **token_index_sequence** – an iterable of token indices

Returns str the decoded string

fit(*data_frame: pandas.core.frame.DataFrame*)
Fits a SequentialEncoder by extracting the character value histogram of a column and capping it at max_tokens

Parameters **data_frame** – pandas data frame

is_fitted() → bool
Checks if ColumnEncoder (still) needs to be fitted to data

Returns True if the column encoder does not require fitting (anymore or at all)

transform(*data_frame: pandas.core.frame.DataFrame*) → numpy.core.multiarray.array
Transforms column of pandas dataframe into sequence of tokens

Parameters **data_frame** – pandas DataFrame

Returns numpy array (rows by seq_len)

static transform_func_seq_single(*string: str, token_to_idx: Dict[str, int], seq_len: int, missing_token_idx: int*) → List[int]
Transforms a single string into a sequence of token ids

Parameters

- **string** – a sequence of symbols as string
- **token_to_idx** – Dict[str, int] with mapping from token to token index
- **seq_len** – length of sequence
- **missing_token_idx** – index for missing symbol

Returns List[int] with transformed values

class datawig.column_encoders.**TfIdfEncoder**(*input_columns: Any*, *output_column: str = None*, *max_tokens: int = 262144*, *tokens: str = 'chars'*, *ngram_range: tuple = None*, *prefixed_concatenation: bool = True*)
TfIdf bag of word encoder for text data, using sklearn's TfidfVectorizer

Parameters

- **input_columns** – List[str] with column names to be used as input for this ColumnEncoder
- **output_column** – Name of output field, used as field name in downstream MxNet iterator
- **max_tokens** – Number of feature buckets (dimensionality of sparse ngram vector). default 2**18
- **tokens** – How to tokenize the input data, supports 'words' and 'chars'.
- **ngram_range** – length of ngrams to use as features
- **prefixed_concatenation** – whether or not to prefix values with column name before concat

decode(*col: pandas.core.series.Series*) → pandas.core.series.Series
Given a series of indices, decode it to input tokens

Parameters **`col`** –

Returns pd.Series of tokens

`fit`(*data_frame: pandas.core.frame.DataFrame*)

Parameters **`data_frame`** –

Returns

`is_fitted`() → bool

Parameters **`self`** –

Returns True if the encoder is fitted

`transform`(*data_frame: pandas.core.frame.DataFrame*) → numpy.core.multiarray.array

Transforms one or more string columns into Bag-of-words vectors.

Parameters **`data_frame`** – pandas DataFrame with text columns

Returns numpy array (rows by max_features)

Python Module Index

Index