Kaggler Documentation

Release 0.9.14

Jeong-Yoon Lee

Mar 06, 2022

Contents:

CHAPTER 1

About Kaggler

Kaggler is a Python package for lightweight online machine learning algorithms and utility functions for ETL and data analysis. It is distributed under the MIT License.

Its online learning algorithms are inspired by Kaggle user tinrtgu's code. It uses the sparse input format that handles large sparse data efficiently. Core code is optimized for speed by using Cython.

CHAPTER 2

Installation

`kaggler` is available on PyPI, and can be installed from `pip` or source as follows:

From `pip`:

```
pip install kaggler
```

From source:

```
git clone https://github.com/jeongyoonlee/kaggler.git
cd kaggler
python setup.py build_ext --inplace
python setup.py install
```

CHAPTER 3

kaggler package

3.1 Submodules

3.2 kaggler.ensemble module

kaggler.ensemble.**netflix**(*es*, *ps*, *e0*, *l=0.0001*)

Combine predictions with the optimal weights to minimize RMSE.

Ref: Töscher, A., Jahrer, M., & Bell, R. M. (2009). The bigchaos solution to the netflix grand prize.

Parameters

- **es** (*list of float*) – RMSEs of predictions
- **ps** (*list of np.array*) – predictions
- **e0** (*float*) – RMSE of all zero prediction
- **l** (*float*) – lambda as in the ridge regression

Returns

- (np.array): ensemble predictions
- (np.array): weights for input predictions

Return type (tuple)

3.3 kaggler.model module

class kaggler.model.**NN**(*n=5*, *h=10*, *b=100000*, *l1=0.0*, *l2=0.0*, *random_state=None*)

Bases: object

Implement a neural network with a single h layer.

fit(*X*, *y*, *X_val=None*, *y_val=None*)
Train a network with the quasi-Newton method.

Parameters

- **X** (*np.array of float*) – feature matrix for training
- **y** (*np.array of float*) – target values for training
- **X_val** (*np.array of float*) – feature matrix for validation
- **y_val** (*np.array of float*) – target values for validation

fprime(*w*, **args*)
Return the derivatives of the cost function for predictions.

Parameters

- **w** (*array of float*) – weight vectors such that: w[:-h1] – weights between the input and h layers w[-h1:] – weights between the h and output layers
- **args** – features (args[0]) and target (args[1])

Returns gradients of the cost function for predictions

func(*w*, **args*)
Return the costs of the neural network for predictions.

Parameters

- **w** (*array of float*) – weight vectors such that: w[:-h1] – weights between the input and h layers w[-h1:] – weights between the h and output layers
- **args** – features (args[0]) and target (args[1])

Returns combined cost of RMSE, L1, and L2 regularization

predict(*X*)
Predict targets for a feature matrix.

Parameters **X** (*np.array of float*) – feature matrix for prediction

Returns prediction (np.array)

predict_raw(*X*)
Predict targets for a feature matrix.

Parameters **X** (*np.array of float*) – feature matrix for prediction

class kaggler.model.**BaseAutoML**(*params*, *space*, *n_est=500*, *n_stop=10*, *sample_size=10000*, *valid_size=0.2*, *shuffle=True*, *feature_selection=True*, *n_fs=10*, *fs_th=0.0*, *fs_pct=0.0*, *hyperparam_opt=True*, *n_hpopt=100*, *minimize=True*, *n_random_col=10*, *random_state=42*)
Bases: object

Base optimized regressor class.

select_features(*X*, *y*)
Select features based on feature importances.

It adds self.n_random_col random columns to features and trains the regressor for n_eval rounds. The features ranked higher than the average rank of random columns in the best model are selected.

Parameters

- **X** (*pandas.DataFrame*) – features
- **y** (*pandas.Series*) – labels

Returns the list of selected features

Return type (list of str)

tune(*X*, *y*)

Tune the regressor with feature selection and parameter search.

Parameters

- **X** (*pandas.DataFrame*) – features
- **y** (*pandas.Series*) – labels

Returns self

class kaggler.model.**AutoLGB**(*objective='regression'*, *metric='mae'*, *boosting='gbdt'*, *params={'bagging_freq': 1, 'feature_pre_filter': False, 'num_threads': -1, 'seed': 42, 'verbosity': -1}*, *space={'bagging_fraction': <hyperopt.pyll.base.Apply object>, 'feature_fraction': <hyperopt.pyll.base.Apply object>, 'lambda_l1': <hyperopt.pyll.base.Apply object>, 'lambda_l2': <hyperopt.pyll.base.Apply object>, 'learning_rate': <hyperopt.pyll.base.Apply object>, 'max_depth': <hyperopt.pyll.base.Apply object>, 'min_child_samples': <hyperopt.pyll.base.Apply object>, 'num_leaves': <hyperopt.pyll.base.Apply object>}*, *n_est=500*, *n_stop=10*, *sample_size=10000*, *feature_selection=True*, *n_fs=10*, *fs_th=1e-05*, *fs_pct=0.1*, *hyperparam_opt=True*, *n_hpopt=100*, *n_random_col=10*, *random_state=42*, *shuffle=True*)

Bases: kaggler.model.automl.BaseAutoML

class kaggler.model.**AutoXGB**(*objective='reg:linear'*, *metric='rmse'*, *boosting='gbtree'*, *params={'n_jobs': -1, 'random_state': 42}*, *space={'colsample_bytree': <hyperopt.pyll.base.Apply object>, 'learning_rate': <hyperopt.pyll.base.Apply object>, 'max_depth': <hyperopt.pyll.base.Apply object>, 'min_child_weight': <hyperopt.pyll.base.Apply object>, 'subsample': <hyperopt.pyll.base.Apply object>}*, *n_est=500*, *n_stop=10*, *sample_size=10000*, *feature_selection=True*, *n_fs=10*, *fs_th=1e-05*, *fs_pct=0.1*, *hyperparam_opt=True*, *n_hpopt=100*, *n_random_col=10*, *random_state=42*, *shuffle=True*)

Bases: kaggler.model.automl.BaseAutoML

3.4 kaggler.data_io module

class kaggler.data_io.**PathJoiner**(*filename='SETTINGS.json'*)

Bases: object

Load directory names from SETTINGS.json.

Originally written by Baris Umog (https://www.kaggle.com/barisumog).

Usage: # In SETTINGS.json, "data": "/path/to/data/". # To load "/path/to/data/targets.array" file to y: PATH = PathJoiner() y = load(PATH.data('targets.array'))

kaggler.data_io.**dump_svmlight_file**(*X*, *y*, *f*, ***, *zero_based=True*, *comment=None*, *query_id=None*, *multilabel=False*)

Dump the dataset in svmlight / libsvm file format.

This format is a text-based format, with one sample per line. It does not store zero valued features hence is suitable for sparse dataset.

The first element of each line can be used to store a target variable to predict.

Parameters

- **X** (*{array-like, sparse matrix} of shape (n_samples, n_features)*) – Training vectors, where *n_samples* is the number of samples and *n_features* is the number of features.
- **y** (*{array-like, sparse matrix}, shape = [n_samples (, n_labels)]*) – Target values. Class labels must be an integer or float, or array-like objects of integer or float for multilabel classifications.
- **f** (*str or file-like in binary mode*) – If string, specifies the path that will contain the data. If file-like, data will be written to f. f should be opened in binary mode.
- **zero_based** (*boolean, default=True*) – Whether column indices should be written zero-based (True) or one-based (False).
- **comment** (*str, default=None*) – Comment to insert at the top of the file. This should be either a Unicode string, which will be encoded as UTF-8, or an ASCII byte string. If a comment is given, then it will be preceded by one that identifies the file as having been dumped by scikit-learn. Note that not all tools grok comments in SVMlight files.
- **query_id** (*array-like of shape (n_samples,), default=None*) – Array containing pairwise preference constraints (qid in svmlight format).
- **multilabel** (*boolean, default=False*) – Samples may have several labels each (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)

  New in version 0.17: parameter *multilabel* to support multilabel datasets.

kaggler.data_io.**getLogger**(*name=None*)

Return a logger with the specified name, creating it if necessary.

If no name is specified, return the root logger.

kaggler.data_io.**is_number**(*s*)

Check if a string is a number or not.

kaggler.data_io.**load_csv**(*path*)

Load data from a CSV file.

Parameters

- **path** (*str*) – A path to the CSV format file containing data.
- **dense** (*boolean*) – An optional variable indicating if the return matrix should be dense. By default, it is false.

Returns Data matrix X and target vector y

kaggler.data_io.**load_data**(*path*, *dense=False*)

Load data from a CSV, LibSVM or HDF5 file based on the file extension.

Parameters

- **path** (*str*) – A path to the CSV, LibSVM or HDF5 format file.
- **dense** (*boolean*) – An optional variable indicating if the return matrix should be dense. By default, it is false.

Returns Data matrix X and target vector y

kaggler.data_io.**load_hdf5**(*path*)
Load data from a HDF5 file.

Parameters

- **path** (*str*) – A path to the HDF5 format file containing data.
- **dense** (*boolean*) – An optional variable indicating if the return matrix should be dense. By default, it is false.

Returns Data matrix X and target vector y

kaggler.data_io.**load_svmlight_file**(*f*, ***, *n_features=None*, *dtype=<class 'numpy.float64'>*, *multilabel=False*, *zero_based='auto'*, *query_id=False*, *offset=0*, *length=-1*)
Load datasets in the svmlight / libsvm format into sparse CSR matrix

This format is a text-based format, with one sample per line. It does not store zero valued features hence is suitable for sparse dataset.

The first element of each line can be used to store a target variable to predict.

This format is used as the default format for both svmlight and the libsvm command line programs.

Parsing a text based source can be expensive. When repeatedly working on the same dataset, it is recommended to wrap this loader with joblib.Memory.cache to store a memmapped backup of the CSR results of the first call and benefit from the near instantaneous loading of memmapped structures for the subsequent calls.

In case the file contains a pairwise preference constraint (known as "qid" in the svmlight format) these are ignored unless the query_id parameter is set to True. These pairwise preference constraints can be used to constraint the combination of samples when using pairwise loss functions (as is the case in some learning to rank problems) so that only pairs with the same query_id value are considered.

This implementation is written in Cython and is reasonably fast. However, a faster API-compatible loader is also available at:

https://github.com/mblondel/svmlight-loader

Parameters

- **f** (*str, file-like or int*) – (Path to) a file to load. If a path ends in ".gz" or ".bz2", it will be uncompressed on the fly. If an integer is passed, it is assumed to be a file descriptor. A file-like or file descriptor will not be closed by this function. A file-like object must be opened in binary mode.
- **n_features** (*int, default=None*) – The number of features to use. If None, it will be inferred. This argument is useful to load several files that are subsets of a bigger sliced dataset: each subset might not have examples of every feature, hence the inferred shape might vary from one slice to another. n_features is only required if `offset` or `length` are passed a non-default value.
- **dtype** (*numpy data type, default=np.float64*) – Data type of dataset to be loaded. This will be the data type of the output numpy arrays `X` and `y`.
- **multilabel** (*bool, default=False*) – Samples may have several labels each (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)
- **zero_based** (*bool or "auto", default="auto"*) – Whether column indices in f are zero-based (True) or one-based (False). If column indices are one-based, they are transformed to zero-based to match Python/NumPy conventions. If set to "auto", a heuristic check is applied to determine this from the file contents. Both kinds of files occur "in the wild", but they are unfortunately not self-identifying. Using "auto" or True should always be safe when no `offset` or `length` is passed. If `offset` or `length` are passed, the

"auto" mode falls back to `zero_based=True` to avoid having the heuristic check yield inconsistent results on different segments of the file.

- **query_id** (*bool, default=False*) – If True, will return the query_id array for each file.
- **offset** (*int, default=0*) – Ignore the offset first bytes by seeking forward, then discarding the following bytes up until the next new line character.
- **length** (*int, default=-1*) – If strictly positive, stop reading any new line of data once the position in the file has reached the (offset + length) bytes threshold.

Returns

- **X** (*scipy.sparse matrix of shape (n_samples, n_features)*)
- **y** (*ndarray of shape (n_samples,), or, in the multilabel a list of*) – tuples of length n_samples.
- **query_id** (*array of shape (n_samples,)*) – query_id for each sample. Only returned when query_id is set to True.

See also:

load_svmlight_files() Similar function for loading multiple files in this format, enforcing the same number of features/columns on all of them.

Examples

To use joblib.Memory to cache the svmlight file:

```
from joblib import Memory
from .datasets import load_svmlight_file
mem = Memory("./mycache")

@mem.cache
def get_data():
    data = load_svmlight_file("mysvmlightfile")
    return data[0], data[1]

X, y = get_data()
```

kaggler.data_io.**open**()

Open file and return a stream. Raise OSError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for creating and writing to a new file, and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation. The 'x' mode implies 'w' and raises an *FileExistsError* if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

'U' mode is deprecated and will raise an exception in future versions of Python. It has no effect in Python 3. Use newline to control universal newlines mode.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on *io.DEFAULT_BUFFER_SIZE*. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which isatty() returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding errors are to be handled—this argument should not be used in binary mode. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for codecs.register or run 'help(codecs.Codec)' for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text mode). It can be None, '', 'n', 'r', and 'rn'. It works as follows:

- On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in 'n', 'r', or 'rn', and these are translated into 'n' before being returned to the caller. If it is '', universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if newline is None, any 'n' characters written are translated to the system default line separator, os.linesep. If newline is '' or 'n', no translation takes place. If newline is any of the other legal values, any 'n' characters written are translated to the given string.

If closefd is False, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *opener* must return an open file descriptor (passing os.open as *opener* results in functionality similar to passing None).

open() returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a TextIOWrapper. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a BufferedReader; in write binary and append binary modes, it returns a BufferedWriter, and in read/write mode, it returns a BufferedRandom.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings StringIO can be used like a file opened in a text mode, and for bytes a BytesIO can be used like a file opened in a binary mode.

kaggler.data_io.**read_sps**(*path*)
Read a LibSVM file line-by-line.

Parameters **path** (*str*) – A path to the LibSVM file to read.

Yields data (list) and target (int).

kaggler.data_io.**save_csv**(*X*, *y*, *path*)
Save data as a CSV file.

Parameters

- **X** (*numpy or scipy sparse matrix*) – Data matrix
- **y** (*numpy array*) – Target vector.
- **path** (*str*) – Path to the CSV file to save data.

kaggler.data_io.**save_data**(*X*, *y*, *path*)
Save data as a CSV, LibSVM or HDF5 file based on the file extension.

Parameters

- **X** (*numpy or scipy sparse matrix*) – Data matrix
- **y** (*numpy array*) – Target vector. If None, all zero vector will be saved.
- **path** (*str*) – Path to the CSV, LibSVM or HDF5 file to save data.

kaggler.data_io.**save_hdf5**(*X*, *y*, *path*)
Save data as a HDF5 file.

Parameters

- **X** (*numpy or scipy sparse matrix*) – Data matrix
- **y** (*numpy array*) – Target vector.
- **path** (*str*) – Path to the HDF5 file to save data.

kaggler.data_io.**save_libsvm**(*X*, *y*, *path*)
Save data as a LibSVM file.

Parameters

- **X** (*numpy or scipy sparse matrix*) – Data matrix
- **y** (*numpy array*) – Target vector.
- **path** (*str*) – Path to the CSV file to save data.

3.5 kaggler.feature_selection module

class kaggler.feature_selection.**DropInactive**(*lowest=25*)
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Drop all zero features.

Originally written by Baris Umog (https://www.kaggle.com/barisumog).

class kaggler.feature_selection.**DropLowInfo**(*margin=0.02*, *weighted=True*)
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Drop features with low information.

Originally written by Baris Umog (https://www.kaggle.com/barisumog).

3.6 kaggler.preprocessing module

class kaggler.preprocessing.**DAE**(*cat_cols=[]*, *num_cols=[]*, *embedding_dims=[]*, *encoding_dim=128*, *n_layer=1*, *n_encoder=1*, *noise_std=0.0*, *swap_prob=0.2*, *mask_prob=0.0*, *dropout=0.2*, *min_obs=1*, *n_epoch=10*, *batch_size=1024*, *learning_rate=0.004*, *random_state=42*, *label_encoding=True*, *pretrained_model=None*, *freeze_embedding=True*)
Bases: sklearn.base.BaseEstimator

Denoising AutoEncoder feature transformer.

fit(*X*, *y=None*, *validation_data=None*)
Train DAE

Parameters

- **X** (*pandas.DataFrame*) – features to encode
- **y** (*pandas.Series, optional*) – not used
- **validation_data** (*list of pandas.DataFrame and pandas.Series*) – validation features and target

Returns None

fit_transform(*X*, *y=None*, *validation_data=None*)
Train DAE and encode features using the DAE trained

Parameters

- **X** (*pandas.DataFrame*) – features to encode
- **y** (*pandas.Series, optional*) – not used
- **validation_data** (*list of pandas.DataFrame and pandas.Series*) – validation features and target

Returns Encoding matrix for features

load_dae(*model*, *freeze_embedding=True*)
Load weights for self.dae from another DAE model.

Parameters

- **model** (DAE) – a DAE model with the same init parameters

- **freeze_embedding** (*bool*) – whether to freeze categorical embedding layers (True) or not (False)

Returns None

transform(*X*)

Encode features using the DAE trained

Parameters **X** (*pandas.DataFrame*) – features to encode

Returns Encoding matrix for features

class kaggler.preprocessing.**SDAE**(*cat_cols=[]*, *num_cols=[]*, *embedding_dims=[]*, *encoding_dim=128*, *n_layer=1*, *n_encoder=1*, *noise_std=0.0*, *swap_prob=0.2*, *mask_prob=0.0*, *dropout=0.2*, *min_obs=1*, *n_epoch=10*, *batch_size=1024*, *learning_rate=0.004*, *random_state=42*, *label_encoding=True*, *pretrained_model=None*, *freeze_embedding=True*, *n_class=None*, *output_layer_size=1024*, *output_activation='sigmoid'*, *output_loss='binary_classification'*)

Bases: kaggler.preprocessing.autoencoder.DAE

Supervised Denoising AutoEncoder feature transformer.

fit(*X*, *y*, *validation_data=None*)

Train supervised DAE

Parameters

- **X** (*pandas.DataFrame*) – features to encode
- **y** (*pandas.Series*) – target variable
- **validation_data** (*list of pandas.DataFrame and pandas.Series*) – validation features and target

Returns None

class kaggler.preprocessing.**OneHotEncoder**(*min_obs=10*)

Bases: sklearn.base.BaseEstimator

One-Hot-Encoder that groups infrequent values into one dummy variable.

min_obs

minimum number of observation to create a dummy variable

Type int

label_encoders

label encoders and their maximums for columns

Type list of (dict, int)

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

Encode categorical columns into sparse matrix with one-hot-encoding.

Parameters **X** (*pandas.DataFrame*) – categorical columns to encode

Returns sparse matrix encoding categorical variables into dummy variables

transform(*X*)

Encode categorical columns into sparse matrix with one-hot-encoding.

Parameters **X** (*pandas.DataFrame*) – categorical columns to encode

Returns

sparse matrix encoding categorical variables into dummy variables

Return type (scipy.sparse.coo_matrix)

class kaggler.preprocessing.**LabelEncoder**(*min_obs=10*)

Bases: sklearn.base.BaseEstimator

Label Encoder that groups infrequent values into one label.

min_obs

minimum number of observation to assign a label.

Type int

label_encoders

label encoders for columns

Type list of dict

label_maxes

maximum of labels for columns

Type list of int

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

Encode categorical columns into label encoded columns

Parameters **X** (*pandas.DataFrame*) – categorical columns to encode

Returns label encoded columns

Return type (pandas.DataFrame)

transform(*X*)

Encode categorical columns into label encoded columns

Parameters **X** (*pandas.DataFrame*) – categorical columns to encode

Returns label encoded columns

Return type (pandas.DataFrame)

class kaggler.preprocessing.**TargetEncoder**(*smoothing=1*, *min_samples=10*, *cv=KFold(n_splits=5*, *random_state=42*, *shuffle=True)*)

Bases: sklearn.base.BaseEstimator

Target Encoder that encode categorical values into average target values.

Smoothing and min_samples are added based on olivier's kernel at Kaggle: https://www.kaggle.com/ogrellier/python-target-encoding-for-categorical-features

, which is based on Daniele Micci-Barreca (2001): https://dl.acm.org/citation.cfm?id=507538

target_encoders

target encoders for columns

Type list of dict

fit(*X*, *y*)

Encode categorical columns into average target values.

Parameters

- **X** (*pandas.DataFrame*) – categorical columns to encode

- **y** (*pandas.Series*) – the target column

Returns encoded columns

Return type (pandas.DataFrame)

fit_transform(*X*, *y*)
Encode categorical columns into average target values.

Parameters

- **X** (*pandas.DataFrame*) – categorical columns to encode
- **y** (*pandas.Series*) – the target column

Returns encoded columns

Return type (pandas.DataFrame)

transform(*X*)
Encode categorical columns into average target values.

Parameters **X** (*pandas.DataFrame*) – categorical columns to encode

Returns encoded columns

Return type (pandas.DataFrame)

class kaggler.preprocessing.**EmbeddingEncoder**(*cat_cols*, *num_cols=[]*, *n_emb=[]*, *min_obs=10*, *n_epoch=10*, *batch_size=1024*, *cv=None*, *random_state=42*)
Bases: sklearn.base.BaseEstimator

EmbeddingEncoder encodes categorical features to numerical embedding features.

Reference: 'Entity embeddings to handle categories' by Abhishek Thakur at https://www.kaggle.com/abhishek/entity-embeddings-to-handle-categories

fit(*X*, *y*)
Train a neural network model with embedding layers.

Parameters

- **X** (*pandas.DataFrame*) – categorical features to create embeddings for
- **y** (*pandas.Series*) – a target variable

Returns A trained EmbeddingEncoder object.

class kaggler.preprocessing.**Normalizer**
Bases: sklearn.base.BaseEstimator

Normalizer that transforms numerical columns into normal distribution.

ecdfs
empirical CDFs for columns

Type list of empirical CDF

fit_transform(*X*, *y=None*)
Normalize numerical columns.

Parameters **X** (*pandas.DataFrame*) – numerical columns to normalize

Returns normalized numerical columns

Return type (pandas.DataFrame)

transform(*X*)
Normalize numerical columns.

Parameters **X** (*pandas.DataFrame*) – numerical columns to normalize

Returns normalized numerical columns

Return type (pandas.DataFrame)

class kaggler.preprocessing.**QuantileEncoder**(*n_label=10*, *sample=100000*, *random_state=42*)
Bases: sklearn.base.BaseEstimator

QuantileEncoder encodes numerical features to quantile values.

ecdfs
empirical CDFs for columns

Type list of empirical CDF

n_label
the number of labels to be created.

Type int

fit(*X*, *y=None*)
Get empirical CDFs of numerical features.

Parameters **X** (*pandas.DataFrame*) – numerical features to encode

Returns A trained QuantileEncoder object.

fit_transform(*X*, *y=None*)
Get empirical CDFs of numerical features and encode to quantiles.

Parameters **X** (*pandas.DataFrame*) – numerical features to encode

Returns Encoded features (pandas.DataFrame).

transform(*X*)
Encode numerical features to quantiles.

Parameters **X** (*pandas.DataFrame*) – numerical features to encode

Returns Encoded features (pandas.DataFrame).

class kaggler.preprocessing.**FrequencyEncoder**(*cv=None*)
Bases: sklearn.base.BaseEstimator

Frequency Encoder that encode categorical values by counting frequencies.

frequency_encoders
frequency encoders for columns

Type list of dict

fit(*X*, *y=None*)
Encode categorical columns into frequency.

Parameters

- **X** (*pandas.DataFrame*) – categorical columns to encode
- **y** (*pandas.Series, optional*) – the target column

Returns encoded columns

Return type (pandas.DataFrame)

fit_transform(*X*, *y=None*)
Encode categorical columns into feature frequency counts.

Parameters

- **X** (*pandas.DataFrame*) – categorical columns to encode
- **y** (*pandas.Series, optional*) – the target column

transform(*X*)
Encode categorical columns into feature frequency counts.

Parameters **X** (*pandas.DataFrame*) – categorical columns to encode

Returns encoded columns

Return type (pandas.DataFrame)

3.7 kaggler.online_model module

class kaggler.online_model.**FTRL**
Bases: object

FTRL online learner with the hasing trick using liblinear format data.

inspired by Kaggle user tinrtgu's code at http://goo.gl/K8hQBx original FTRL paper is available at http://goo.gl/iqIaH0

n
number of features after hashing trick

Type int

epoch
number of epochs

Type int

a
alpha in the per-coordinate rate

Type double

b
beta in the per-coordinate rate

Type double

l1
L1 regularization parameter

Type double

l2
L2 regularization parameter

Type double

w
feature weights

Type array of double

c
counters for weights

Type array of double

z
lazy weights

Type array of double

interaction
whether to use 2nd order interaction or not

Type boolean

fit()
Update the model with a sparse input feature matrix and its targets.

Parameters

- **X** (*scipy.sparse.csr_matrix*) – a list of (index, value) of non-zero features
- **y** (*numpy.array*) – targets

Returns updated model weights and counts

predict()
Predict for a sparse matrix X.

Parameters **X** (*scipy.sparse.csr_matrix*) – a sparse matrix for input features

Returns predictions for input features

Return type p (numpy.array)

read_sparse()
Apply hashing trick to the libsvm format sparse file.

Parameters **path** (*str*) – a file path to the libsvm format sparse file

Yields *x (list of int)* – a list of index of non-zero features y (int): target value

class kaggler.online_model.**FM**
Bases: object

Factorization Machine online learner.

n
number of input features

Type int

epoch
number of epochs

Type int

k
size of factors for interactions

Type int

a
initial learning rate

Type double

w0
weight for bias

Type double

c0
counters

Type double

w
feature weights

Type array of double

c
counters for weights

Type array of double

V
feature weights for factors

Type array of double

fit()
Update the model with a sparse input feature matrix and its targets.

Parameters

- **X** (*scipy.sparse.csr_matrix*) – a list of (index, value) of non-zero features
- **y** (*numpy.array*) – targets

Returns updated model weights and counts

predict()
Predict for a sparse matrix X.

Parameters **X** (*scipy.sparse.csr_matrix*) – a sparse matrix for input features

Returns predictions for input features

Return type p (numpy.array)

predict_one()
Predict for features.

Parameters **x** (*list of tuple*) – a list of (index, value) of non-zero features

Returns a prediction for input features

Return type p (double)

read_sparse()
Apply hashing trick to the libsvm format sparse file.

Parameters **path** (*str*) – a file path to the libsvm format sparse file

Yields *idx (list of int)* – a list of index of non-zero features val (list of double): a list of values of non-zero features y (int): target value

update_one()
Update the model.

Parameters

- **idx** (*list of int*) – a list of index of non-zero features

- **val** (*list of double*) – a list of values of non-zero features
- **e** (*double*) – error between the prediction of the model and target

Returns updated model weights and counts

class kaggler.online_model.**NN**

Bases: object

Neural Network with a single ReLU hidden layer online learner.

n

number of input units

Type int

epoch

number of epochs

Type int

h

number of hidden units

Type int

a

initial learning rate

Type double

l2

L2 regularization parameter

Type double

w0

weights between the input and hidden layers

Type array of double

w1

weights between the hidden and output layers

Type array of double

z

hidden units

Type array of double

c

counter

Type double

c1

counters for hidden units

Type array of double

fit()

Update the model with a sparse input feature matrix and its targets.

Parameters

- **X** (*scipy.sparse.csr_matrix*) – a list of (index, value) of non-zero features

- **y** (*numpy.array*) – targets

Returns updated model weights and counts

predict()
Predict for a sparse matrix X.

Parameters **X** (*scipy.sparse.csr_matrix*) – a sparse matrix for input features

Returns predictions for input features

Return type p (numpy.array)

predict_one()
Predict for features.

Parameters **x** (*list of tuple*) – a list of (index, value) of non-zero features

Returns a prediction for input features

Return type p (double)

read_sparse()
Read a libsvm format sparse file line by line.

Parameters **path** (*str*) – a file path to the libsvm format sparse file

Yields *idx (list of int)* – a list of index of non-zero features val (list of double): a list of values of non-zero features y (int): target value

update_one()
Update the model with one observation.

Parameters

- **x** (*list of tuple*) – a list of (index, value) of non-zero features
- **e** (*double*) – error between the prediction of the model and target

Returns updated model weights and counts

class kaggler.online_model.**NN_H2**
Bases: object

Neural Network with 2 ReLU hidden layers online learner.

n
number of input units

Type int

epoch
number of epochs

Type int

h1
number of the 1st level hidden units

Type int

h2
number of the 2nd level hidden units

Type int

a
initial learning rate

Type double

l2
L2 regularization parameter

Type double

w0
weights between the input and 1st hidden layers

Type array of double

w1
weights between the 1st and 2nd hidden layers

Type array of double

w2
weights between the 2nd hidden and output layers

Type array of double

z1
1st level hidden units

Type array of double

z2
2nd level hidden units

Type array of double

c
counter

Type double

c1
counters for 1st level hidden units

Type array of double

c2
counters for 2nd level hidden units

Type array of double

fit()
Update the model with a sparse input feature matrix and its targets.

Parameters

- **X** (*scipy.sparse.csr_matrix*) – a list of (index, value) of non-zero features
- **y** (*numpy.array*) – targets

Returns updated model weights and counts

predict()
Predict for a sparse matrix X.

Parameters **X** (*scipy.sparse.csr_matrix*) – a sparse matrix for input features

Returns predictions for input features

Return type p (numpy.array)

predict_one()
Predict for features.

Parameters **x** (*list of tuple*) – a list of (index, value) of non-zero features

Returns a prediction for input features

Return type p (double)

read_sparse()
Read the libsvm format sparse file line by line.

Parameters **path** (*str*) – a file path to the libsvm format sparse file

Yields *idx (list of int)* – a list of index of non-zero features val (list of double): a list of values of non-zero features y (int): target value

update_one()
Update the model.

Parameters

- **x** (*list of tuple*) – a list of (index, value) of non-zero features
- **e** (*double*) – error between the prediction of the model and target

Returns updated model weights and counts

class kaggler.online_model.**SGD**
Bases: object

Simple online learner using a hasing trick.

epoch
number of epochs

Type int

n
number of features after hashing trick

Type int

a
initial learning rate

Type double

l1
L1 regularization parameter

Type double

l2
L2 regularization parameter

Type double

w
feature weights

Type array of double

c
counters for weights

Type array of double

interaction
whether to use 2nd order interaction or not

Type boolean

fit()
Update the model with a sparse input feature matrix and its targets.

Parameters

- **X** (*scipy.sparse.csr_matrix*) – a list of (index, value) of non-zero features
- **y** (*numpy.array*) – targets

Returns updated model weights and counts

predict()
Predict for a sparse matrix X.

Parameters **X** (*scipy.sparse.csr_matrix*) – a sparse matrix for input features

Returns predictions for input features

Return type p (numpy.array)

predict_one()
Predict for features.

Parameters **x** (*list of int*) – a list of index of non-zero features

Returns a prediction for input features

Return type p (double)

read_sparse()
Apply hashing trick to the libsvm format sparse file.

Parameters **path** (*str*) – a file path to the libsvm format sparse file

Yields *x (list of int)* – a list of index of non-zero features y (int): target value

update_one()
Update the model.

Parameters

- **x** (*list of int*) – a list of index of non-zero features
- **e** (*double*) – error between the prediction of the model and target

Returns updates model weights and counts

3.8 kaggler.util module

kaggler.util.**getLogger**(*name=None*)
Return a logger with the specified name, creating it if necessary.

If no name is specified, return the root logger.

kaggler.util.**get_downsampled_index**()
Return the index that downsamples a vector x by the rate.

kaggler.util.**get_downsampled_index0**()
Return the index that downsamples 0s of a vector x by the rate.

kaggler.util.**point**()
Calculate Kaggle points to earn after a competition.

Parameters

- **rank** (*int*) – final ranking in the private leaderboard.
- **n_team** (*int*) – the number of teams participated in the competition.
- **n_teammate** (*int*) – the number of team members in my team.
- **t** (*int*) – the number of days since the competition ends.

Returns returns Kaggle points to earn after a compeittion.

kaggler.util.**rank**()
Rank a vector x. Ties will be averaged.

kaggler.util.**set_column_width**()
Set the column width of a matrix X to n_col.

3.9 Module contents

CHAPTER 4

Changelog

Please check the GitHub release notes at https://github.com/jeongyoonlee/Kaggler/releases.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

G

H

I

K

L

M

N

O

P