HiDi

Release 0.0.2

Apr 27, 2017

Contents

HiDi is a library for high-dimensional embedding generation for collaborative filtering applications.

CHAPTER 1

Why HiDi?

We created HiDi because generating embeddings for collaborative filtering applications is a work intensive process that involves many data transformations, each of which requires special consideration to get a good result. HiDi makes the process more simple by breaking work into small steps, each of which can be executed in a pipeline.

The unit of work in HiDi is a Transformer. Transformers need only implement one function, *transform*.

CHAPTER 2

Ok, How Do I Use It?

This will get you started.

```
from hidi import inout, clean, matrix, pipeline


# CSV file with link_id and item_id columns
in_files = ['hidi/examples/data/user-item.csv']

# File to write output data to
outfile = 'embeddings.csv'

transforms = [
    inout.ReadTransform(in_files),       # Read data from disk
    clean.DedupeTransform(),             # Dedupe it
    matrix.SparseTransform(),            # Make a sparse user*item matrix
    matrix.SimilarityTransform(),        # To item*item similarity matrix
    matrix.SVDTransform(),               # Perform SVD dimensionality reduction
    matrix.ItemsMatrixToDFTransform(),   # Make a DataFrame with an index
    inout.WriteTransform(outfile)        # Write results to csv
]

pl = pipeline.Pipeline(transforms)
pl.run()
```

CHAPTER 3

Setup

Requirements

HiDi is tested against CPython 2.7, 3.4, 3.5, and 3.6. It may work with different version of CPython.

Installation

To install HiDi, simply run

```
$ pip install hidi
```

CHAPTER 4

API Documentation

Pipeline Module

HiDi's Pipeline module exposes functionality for creating and running pipelines.

class hidi.pipeline.**Pipeline**(*transformers*)
Bases: object

Pipeline of transforms.

Sequentially apply a list of transforms. All steps of the pipeline must be 'transforms', that is, they must implement transform method. The Pipeline abstraction is inspired by the SciKit Learn Pipeline abstraction.

Takes a list of transform instances.

add(*transform*)
Add a transform to the pipeline.

run(*io=None*, *progress=True*, ***kwargs*)
Executes the pipeline and returns the final result.

Takes an optional io parameter that will serve as input to the initial transformer.

Inout Module

HiDi's pipeline module exposes functionality for performing IO tasks.

class hidi.inout.**ReadTransform**(*infiles*, ***kwargs*)
Bases: hidi.transform.Transform

Read input csv data from disk.

Input data should be a csv file formatted with three columns: link_id, item_id, and score. If score is not provided, it we be defaulted to one. link_id represents to the "user" and *item_id* represents the "item" in the context of traditional collaborative filtering.

Parameters **infiles** (*array*) – Array of paths to csv documents to be loaded and concatenated into one DataFrame. Each csv document must have a `link_id` and a `item_id` column. An optional `score` column may also be supplied.

transform(***kwargs*)
Read in files from the `infiles` array given upon instantiation.

class hidi.inout.**WriteTransform**(*outfile*, *file_format='csv'*, *enc=None*, *link_key='link_id'*)
Bases: hidi.transform.Transform

Write output to disk in csv or json formats.

Parameters

- **outfile** (*str*) – A string that is a path to the desired output on the file system.
- **file_format** (*str*) – A string that is a file extension, either `json` or `csv`.

transform(*df*, ***kwargs*)
Write a DataFrame to a file.

Parameters **df** (*pandas.DataFrame*) – The Pandas DataFrame to be written to a file

Matrix Module

HiDi's matrix module exposes functionality for transforming matrices.

class hidi.matrix.**ApplyTransform**(*fn*)
Bases: hidi.transform.Transform

Apply a function to an input.

Takes a single argument, *fn*, which must be a function accepting one argument (the function to apply), and kwargs.

Parameters **fn** (*function*) – The function to be applied to transform input.

transform(*x*, ***kwargs*)

Parameters **x** – The input to the function `fn`.

class hidi.matrix.**SimilarityTransform**(*axis=0*)
Bases: hidi.transform.Transform

Takes the dot product of a link*item matrix.

Returns either a link*link or item*item similarity matrix. If axis is `0`, an item*item matrix is returned, if axis is `1` a link*link matrix is returned. The returned matrix represents a similarity matrix.

The transform function returns a tuple containing the similarity matrix, and the links or items, depending on the axis.

Parameters **axis** (*int[0, 1]*) – The axis to perform the dot product for.

transform(*M*, *items*, *links*, ***kwargs*)

Parameters

- **M** (*numpy ndarray-like*) – The matrix to create a similarity matrix from
- **items** (*array*) – Array of `item_ids` in the same order that they appear in `M`.
- **links** (*array*) – Array of `link_ids` in the same order that they appear in `M`.

class hidi.matrix.**ScalarTransform**(*fn=<ufunc 'log'>*)
Bases: hidi.transform.Transform

Scale the matrix using a function or class method.

ScalerTransform takes an *fn* argument that specifies the function that should be applied to the matrix. If *fn* is a string the scaler transform will try to call a function by that name on the matrix, if it is a function reference, scaler transform will call that function with the matrix as input.

Parameters **fn** (*str | function*) – The scalar function to use. If fn is a string then an attribute of that name will be looked up and called. If fn is a function, that function will be called with the input given to transform.

transform(*matrix_to_scale*, ***kwargs*)
Takes a matrix_to_scale as a numpy ndarray-like object and performs scaling on it, then returns the result.

class hidi.matrix.**SparseTransform**
Bases: hidi.transform.Transform

Make a sparse item*link matrix using SciPy's sparse compressed row matrix implementation.

transform(**func_args*, ***func_kwargs*)
Takes a dataframe that has link_id, item_id and score columns.

Returns a SciPy csr_matrix.

Parameters **df** (*pandas.DataFrame*) – The DataFrame to make a sparse matrix from. Must have link_id, item_id, and score columns.

Return type scipy.sparse.csr_matrix

class hidi.matrix.**DenseTransform**
Bases: hidi.transform.Transform

Transform a sparse matrix to its dense representation.

transform(*M*, ***kwargs*)
Takes a sparse matrix and transform it into its dense representation

Parameters **M** (*scipy.sparse classes*) – a sparse matrix

class hidi.matrix.**ItemsMatrixToDFTransform**
Bases: hidi.transform.Transform

Create a Pandas DataFrame object with items as the index.

transform(*M*, *items*, ***kwargs*)
Takes a numpy ndarray-like object and a list of item identifiers to be used as the index for the DataFrame.

class hidi.matrix.**KerasEvaluationTransform**(*keras_model*, *validation_matrix*, *tts_seed=42*, *tt_split=0.25*, ***keras_kwargs*)
Bases: hidi.transform.Transform

Generalized transform for Keras algorithm

This transform takes a Keras sequential model, a validation matrix and its keyword arugments upon initialization.

Parameters

- **keras_model** (*Keras Sequential model*) – a Keras sequential model which is documented here: https://keras.io/getting-started/sequential-model-guide/

- **validation_matrix** (*pandas.DataFrame*) – A validation matrix is a dataframe that has item_id index, other 'label' columns. It will be inner joined with the M matrix and then fed into the Keras sequential model.
- **tts_seed** (*int*) – random state seed for train_test_split
- **tt_split** (*float*) – the proportion of the dataset to include in the test split for train_test_split

transform(*M*, ***kwargs*)

Takes a Takes a dataframe that has item_id index, other 'features' columns for prediction, and applies a Keras sequential model to it.

Parameters **M** – a dataframe that has item_id index, other

'features' columns :type M: pandas.DataFrame :rtype: a tuple with trained Keras model and its keyword

arguments

class hidi.matrix.**KerasKfoldTransform**(*keras_model*, *validation_matrix*, *kfold_n_splits=10*, *kfold_seed=42*, *kfold_shuffle=True*, *classification=False*, ***keras_kwargs*)

Bases: hidi.transform.Transform

Generalized transform for Keras algorithm with k fold cross validation evaluation

Parameters

- **keras_model** (*Keras Sequential model*) – a Keras sequential model which is documented here: https://keras.io/getting-started/sequential-model-guide/
- **validation_matrix** (*pandas.DataFrame*) – A validation matrix is a dataframe that has item_id index, other 'label' columns. It will be inner joined with the M matrix and then fed into the Keras sequential model.
- **kfold_n_splits** (*int*) – Number of folds for kfold. Must be at least 2.
- **kfold_seed** (*None, int or RandomState*) – random state seed for kfold
- **kfold_shuffle** (*boolean*) – Whether to shuffle the data before splitting into batches for kfold

transform(*M*, ***kwargs*)

Takes a Takes a dataframe that has item_id index, other 'features' columns for prediction, and applies a Keras sequential model to it.

Parameters **M** (*pandas.DataFrame*) – a dataframe that has item_id index, other 'features' columns

Return type a tuple with trained Keras model and its keyword arguments

class hidi.matrix.**KerasPredictionTransform**(*model*)

Bases: hidi.transform.Transform

Generalized transform for Keras model prediction

This transform takes a trained Keras model. It applies the train model to the input when transform is called.

Param model: trained keras model

transform(*M*, ***kwargs*)

Takes a numpy ndarray-like object and applies a trained Keras model to it.

Returns the predictions from the trained Keras model

Param M: a dataframe that has item_id index, other 'features' columns

Param M: pandas.DataFrame

Return type ndarray-like object with its kwargs

class hidi.matrix.**SkLearnTransform**(*SkLearnAlg*, ***sklearn_args*)
Bases: hidi.transform.Transform

Generalized transform for SciKit Learn algorithms.

This transform takes a SciKit Learn algorithm, and its keyword arguments upon initialization. It applies the algorithm to the input when transform is called.

The algorithm to be applied is likely, but not necessarily a sklearn.decomposition algorithm.

transform(*M*, ***kwargs*)
Takes a numpy ndarray-like object and applies a SkLearn algorithm to it.

class hidi.matrix.**SVDTransform**(***svd_kwargs*)
Bases: *hidi.matrix.SkLearnTransform*

Perform Truncated SVD on the matrix.

This uses SciKit Learn's Tuncated SVD implementation, which is documented here: http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

All kwargs given to SVDTransform's initialization function will be given to sklearn.decomposition.TruncatedSVD.

Please reference the sklearn docs when using this transform.

class hidi.matrix.**NimfaTransform**(*NimfaAlg*, ***nimfa_kwargs*)
Bases: hidi.transform.Transform

Generalized Nimfa transform.

This transform takes a nimfa algorithm, and its keyword arguments upon initialization. It applies the algorithm to the input when transform is called.

class hidi.matrix.**SNMFTransform**(***snmf_kwargs*)
Bases: *hidi.matrix.NimfaTransform*

Perform Sparse Nonnegative Matrix Factorization.

This wraps nimfa's snmf function, which is documented here: http://nimfa.biolab.si/nimfa.methods.factorization.snmf.html

All kwargs given to SNFMTransform's initialization function will be given to nimfa.Snmf.

Please reference the nimfa docs when using this transform.

Clean Module

HiDi's clean module exposes functionality for cleaning data.

class hidi.clean.**DedupeTransform**(*skip_dedupe=False*)
Bases: hidi.transform.Transform

Deduplicate link-item tall skinny DataFrame

transform(*df*, ***kwargs*)
Takes a df that has link_id and item_id columns, and deduplicates them so that each pair is represented at most once.

Forking Module

HiDi's forking module exposes functionality for concurrent pipelines. Forking us done with ordinary Transforms that take lists of pipelines upon initialization.

class hidi.forking.**ThreadForkTransform**(*pipelines*, *progress=False*)
Bases: hidi.forking.ExecutorFork

Fork a pipeline using concurrent.futures.ThreadPoolExecutor as a backend for execution.

This is useful if you have several transforms that perform well when running in concurrent threads such as IO heavy or CPU heavy tasks that execute outside the Python runtime.

The forked transform will return a list of Pipeline outputs, in the same order as the forked pipelines were given.

Parameters

- **pipelines** (*list[hidi.pipeline.Pipeline]*) – An array of pipelines to fork execution to.
- **progress** (*bool*) – When True, progress of the forked pipelines will be logged.

class hidi.forking.**ProcessForkTransform**(*pipelines*, *progress=False*)
Bases: hidi.forking.ExecutorFork

Fork a pipeline using concurrent.futures.ProcessesPoolExecutor as a backend for execution.

This method is useful if you have several transforms that can be executed concurrently and are CPU intensive.

The forked pipeline will now return a list of pipeline ouputs, in the same order as the forked pipelines were given.

Special care must be taken as each transform must be pickled to a new process.

Parameters

- **pipelines** (*list[hidi.pipeline.Pipeline]*) – An array of pipelines to fork execution to.
- **progress** (*bool*) – When True, progress of the forked pipelines will be logged.

class hidi.forking.**TrivialForkTransform**(*pipelines*, *progress=False*)
Bases: hidi.transform.Transform

Trivial Fork Transform using an ordinary loop.

Parameters

- **pipelines** (*list[hidi.pipeline.Pipeline]*) – An array of pipelines to fork execution to.
- **progress** (*bool*) – When True, progress of the forked pipelines will be logged.

Example

Here is an example of using a ProcessForkTransform:

```
import numpy as np

from hidi import pipeline, inout, matrix, forking


def to_float32(df, **kwargs):
```

```
    return df.astype(np.int32).astype(np.float32)


def create_pipeline(infiles):
    pl = pipeline.Pipeline([
        inout.ReadTransform(infiles),
        matrix.SparseTransform(),
        matrix.SimilarityTransform(),
        matrix.ApplyTransform(fn=to_float32),
        matrix.ScalarTransform(fn='log1p')
    ])

    left = pipeline.Pipeline([
        matrix.SNMFTransform(rank=32, max_iter=2),
        matrix.DenseTransform(),
        matrix.ItemsMatrixToDFTransform(),
        inout.WriteTransform('snmf-embeddings.csv')
    ])

    right = pipeline.Pipeline([
        matrix.SVDTransform(n_components=32, n_iter=2),
        matrix.ItemsMatrixToDFTransform(),
        inout.WriteTransform('svd-embeddings.csv')
    ])

    pl.add(forking.ProcessForkTransform([left, right], progress=False))

    return pl


def run_pipeline():
    pl = create_pipeline(['hidi/examples/data/user-item.csv'])

    return pl.run(progress=False)


if __name__ == '__main__':
    run_pipeline()
```

Writing Custom Transforms

Writing a custom transform is simple and straighforward. A transformer must only implement one function, `transform`. After initialization, transformers should be stateless so they may be used in multiple pipelines, and each pipeline can be executed many times. Keeping transformers stateless also helps with memory consumption, which can become a problem as the size of input grows.

Here is an example transform class implementation:

```
import hidi


class TimesTwoTransform(object):
    def transform(self, inp, **kwargs):
        # Transform input
        return inp*2, kwargs
```

```
pipeline = hidi.pipeline.Pipeline([
    ...,
    TimesTwoTransform(),
    ...
])

pipeline.run()
```

Python Module Index

h

Index