embedeval

Release 1.0.2

Jan 20, 2020

Contents

CHAPTER 1

Documentation

This part of the documentation guides you through all of embedeval's usage patterns.

1.1 Usage

embedeval is deployed as Command Line Interface Tool for Linux, Mac OS X and Windows. It can be downloaded from PyPI using pip:

```
pip install embedeval
```

1.1.1 Task Management

An important part of *embedeval* is the management of Tasks. The CLI supports two commands to either list the available Tasks or create new Tasks.

List Tasks

To list all available built-in Tasks use the following command:

```
embedeval tasks
```

It'll list the names of all available Tasks. If there is an additional directory of Tasks where *embedeval* should look at use the `-p` or `--tasks-path` option:

```
# also look in the relative tasks/ directory for Tasks
embedeval tasks -p tasks/

# also look in the absolute /etc/tasks directory for Tasks
embedeval tasks --tasks-path /etc/tasks
```

Create Tasks

The second CLI command can be used to create new Tasks.

This command is described in detail in the *A new Task from the CLI* section.

1.1.2 Task Evaluation

The most important part of **embedeval**, however, is the evaluation of Word Embeddings using Tasks.

The following few section go into detail on how to execute an evaluation and how to interpret its reports.

To execute an evaluation the `embedeval` default command can be used or it can be specified explicitly using the `embedeval eval` command:

```
# run an evaluation on embeddding.vec
# with built-in task word-analogy
embedeval embedding.vec -t word-analogy

# same as above, but use the ``eval`` command explicitly
embedeval eval embedding.vec -t word-analogy
```

The `-t` or `--task` CLI option can be used one or multiple times to specify which Tasks should be run. The available Tasks can be listed with the *tasks command*.

In case the Tasks to run are not built-in Tasks the `-p` or `--tasks-path` CLI option can be used to specify the directory where the Tasks are located:

```
# run an evaluation on embeddding.vec
# with task word-analogy from the tasks/ directory
embedeval embedding.vec -t word-analogy -p tasks/
```

1.2 Task Implementation Tutorial

Tasks are the heart pieces of embedeval. They are used to evaluate Word Embeddings on their quality with respect to certain measures given certain baselines.

The Task API can be found in detail in the *Task API* section of this documentation.

There exists two ways of creating a new Taks. The most simplest one is to create one using the `embedeval create-task` command. The other one is from scratch and can be seen as a reference for the details of a Task Implementation.

1.2.1 A new Task from the CLI

The following section describes how to create a new Task using the `embedeval` command line interface.

Task from the built-in skeleton

`embedeval` comes with a built-in skeleton Task which can be used as a base for a new Task:

```
embedeval create-task word-similarity
```

This command will create a new Task called `word-similarity` based on the skeleton Task and placed in a Python module called `word_similarity.py` in the current directory.

Often that's not the place the module should be placed, thus with the `--target-task-path` option a target directory for the new Task can be specified:

```
embedeval create-task word-similarity --target-task-path tasks/
```

The skeleton doesn't provide much, therefore the new Task can be based on another Task known to `embedeval` using the `--based-on` option:

```
embedeval create-task word-similarity --based-on odd-one-out
```

The above command will create the Task based on the `embedeval` `odd-one-out` built-in Taks.

In case the Task should be created based on a non built-in Task the `--tasks-path` option can be used:

```
embedeval create-task word-similarity-v2 \
   --based-on word-similarity \
   --tasks-path tasks/ \
```

1.2.2 A new Task from scratch

The following sections will guide through the steps which need to be done to implement a Task and how to do an evaluation with them.

Step 1: Task Anatomy

First we'll see how a Task is represented in code.

A Task is a subclass of the following Python Interface:

```
class Task(ABC):
    """Base Class for the Task API

    Subclass this Task to automatically register
    an evaluation Task to the Task Registry.

    The Task Evaluation Algorithm must be implemented
    in the ``evaluate()`` method.
    """

    #: Holds the name for this Task.
    #:  This name is used for the discovery.
    NAME: str = ""

    def __init_subclass__(cls, **kwargs):
        """Register subclasses to the Task Registry

        This registration makes it possible
        to later discover and run the Tasks.
        """
        super().__init_subclass__(**kwargs)
        task_registry.register(cls)

    @abstractmethod
```

(continues on next page)

(continued from previous page)

```
    def evaluate(self, embedding: WordEmbedding) -> TaskReport:
        """Evaluate this Task on the given Word Embedding

        The evaluation algorithm should always produce some kind of
        comparable statistics or measures which can be
        provided to the user to verify the quality of the
        given Word Embedding.

        This measure must be returned as a string from this method.

        It should contain everything needed by the user
        to verify the Embedding.
        """
        ...  # pragma: no cover
```

Don't mind the `__init__subclass__` magic method as it's only used internally to register the Tasks in the Task Registry.

Important is `evaluate()` method. that's the method used to implement the evaluation algorithm of a Task.

Step 2: Define a Task

The first thing to think about is the name of the Task. The name is used to reference the Task from the command line to use it for the evaluation.

This name can be used as the class name suffixed with `Task` and must be used as the value for the `NAME` class attribute:

```
from embedeval.task import Task


class WordSimilarityTask(Task):
    """Compare two words in regards of how similar they are"""

    NAME = "word-similarity"

    def evaluate(self, embedding):
        ...
```

This Task can be implemented in an appropriatly named Python module within the embedeval source code at: `src/embedeval/tasks/`. A good name module for this Task would be `src/embedeval/tasks/word_similarity.py`.

Step 3: Implement Task Algorithm

The most important step is to implement the evaluation algorithm for the new Task.

The `evaluate()` method is the place for that. It is given a `WordEmbedding` instance and should return an evauation report as a human readable string:

```
import colorful as cf

from embedeval.task import Task


class WordSimilarityTask(Task):
```

(continues on next page)

(continued from previous page)

```
    """Compare two words in regards of how similar they are"""

    NAME = "word-similarity"

    def evaluate(self, embedding):
        woman_vector = embedding.get_word_vector("woman")
        man_vector = embedding.get_word_vector("man")

        # TODO: ``cosine_similarity`` must be implemented
        similarity = cosine_similarity(woman_vector, man_vector)

        return f"""
            {cf.bold}The Task{cf.reset}:
                How similar is the word woman to the word man?
            was {cf.underlined}successful{cf.reset}.
            The similarity was measured to be
                {cf.bold}{similarity:.2}{cf.reset}
        """
```

Step 4: Run the Task

Now that the evaluation algorithm is implemented we can run the Task on a Word Embedding from the command line using `embedeval`:

```
embedeval embedding.vec -t word-similarity
```

CHAPTER 2

API Reference

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

2.1 API

This part of the documentation lists the full API reference of all public classes and functions.

2.1.1 Word Embedding Representation

Embedeval uses a generic internal representation for Word Embeddings. This representation *is not* dependent on the source format of the Embedding itself, to give a `Task` the chance to be *Embedding format independent*. However, a Task may reference and therefore depend on the source format of the Embedding.

class `embedeval.embedding.`**`WordEmbedding`**

Representation of a loaded immutable Word Embedding

This interface should be implemented to represent concrete parsed Word Embeddings of a particular type.

A Word Embedding always consists of a one-dimensional vector of words and a n-dimensional vector representing the position in the vector space for each word.

`get_word_vector`(*word: str*) → numpy.array

Get the word vector for the given word from Word Embedding

`get_words`() → List[str]

Get a list of all words in the Word Embedding

`path`

Get the path to the Word Embedding file

`shape`

Get the shape of the Embedding.

The first value in the tuple is the amount of words and the second value the vector size of each word.

2.1.2 Task API

The Tasks are the heart piece of Embedeval. The `Task` must be subclassed by a concrete implementation of an evaluation Task to implement the evaluation algorithm.

class `embedeval.task.`**`Task`**

Base Class for the Task API

Subclass this Task to automatically register an evaluation Task to the Task Registry.

The Task Evaluation Algorithm must be implemented in the `evaluate()` method.

`NAME = ''`

Holds the name for this Task. This name is used for the discovery.

`evaluate`(*embedding: embedeval.embedding.WordEmbedding*) → embedeval.taskreport.TaskReport

Evaluate this Task on the given Word Embedding

The evaluation algorithm should always produce some kind of comparable statistics or measures which can be provided to the user to verify the quality of the given Word Embedding.

This measure must be returned as a string from this method.

It should contain everything needed by the user to verify the Embedding.

2.1.3 Implemented Word Embedding Parsers

Embedeval implements parsers for some well-known Word Embedding formats.

class `embedeval.parsers.word2vec_gensim.`**`KeyedVectorsWordEmbedding`**(*path*, *keyed_vectors*)

Represents a word2vec KeyedVectors specific Word Embedding

The word2vec file will be parsed by `gensim`.

The gensim `KeyedVectors` instance is made available in the `self.keyed_vectors` attribute.

`get_word_vector`(*word: str*) → numpy.array

Get the word vector for the given word from Word Embedding

`get_words`() → List[str]

Get a list of all words in the Word Embedding

`keyed_vectors = None`

Holds the gensim KeyedVectors instance

`path`

Get the path to the Word Embedding file

`shape`

Get the shape of the Embedding.

The first value in the tuple is the amount of words and the second value the vector size of each word.

class `embedeval.parsers.word2vec_simple.`**`SimpleWordEmbedding`**(*path*, *word_vectors*)

Represents a word2vec specific Word Embedding

This Word Embedding should only be used for small datasets as it's purely implemented in Python and therefore somewhat slow.

`get_word_vector`(*word: str*) → numpy.array

Get the word vector for the given word from Word Embedding

get_words() → List[str]
Get a list of all words in the Word Embedding

path
Get the path to the Word Embedding file

shape
Get the shape of the Embedding.

The first value in the tuple is the amount of words and the second value the vector size of each word.

2.1.4 Top-Level Package Exports

The `embedeval` Python package exports the Task API as top-level names:

```python
from embedeval import (
    Task,
    TaskReport,
    EmbedevalError
)
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index