WEFE Documentation

Release 0.4.0

WEFE Team

May 02, 2023

GETTING STARTED

WEFE: The Word Embeddings Fairness Evaluation Framework is an open source library for measuring and mitigating bias in word embedding models.

The following pages contain the documentation about WEFE: how to install the package, how to use it and how to contribute, as well as the detailed API documentation and extensive examples.

CHAPTER

ONE

ABOUT

Word Embedding Fairness Evaluation (WEFE) is an open source library that implements many fairness metrics and mitigation methods (debias) in a unified framework. It also provides a standard interface for designing new ones.

The main goal of the library is to provide a ready-to-use tool that allows the user to run bias measures and mitigation methods in a straightforward manner through well-designed and documented interfaces.

In bias measurement, WEFE provides a standard interface for:

- Encapsulating existing fairness metrics.
- Encapsulating the test words used by fairness metrics into standard objects called queries.
- Computing a fairness metric on a given pre-trained word embedding model using user-given queries.

On the other hand, WEFE standardizes all mitigation methods through an interface inherited from scikit-learn basic data transformations: the `fit-transform` interface. This standardization separates the mitigation process into two stages:

- The first step, `fit`, learn the corresponding mitigation transformation.
- The `transform` method applies the transformation learned in the previous step to words residing in the original embedding space.

Note: To learn more about the measurement or mitigation framework, visit *Measurement Framework* or *Mitigation Framework* respectively, in the Conceptual Guides Section.

For practical tutorials on how to measure or mitigate bias, visit *Bias Measurement* or *Bias Mitigation (Debias)* respectively in the WEFE User Guide.

1.1 Motivation and objectives

Word Embedding models are a core component in almost all NLP downstream systems. Several studies have shown that they are prone to inherit stereotypical social biases from the corpus they were built on. The common method for quantifying bias is to use a metric that calculates the relationship between sets of word embeddings representing different social groups and attributes.

Although previous studies have begun to measure bias in embeddings, they are limited both in the types of bias measured (gender, ethnic) and in the models tested. Moreover, each study proposes its own metric, which makes the relationship between the results obtained unclear.

This fact led us to consider that we could use these metrics and studies to make a case study in which we compare and rank the embedding models according to their bias.

We originally proposed WEFE as a theoretical framework that formalizes the main building blocks for measuring bias in word embedding models. The purpose of developing this framework was to run a case study that consistently compares and ranks different embedding models. Seeing the possibility that other research teams are facing the same problem, we decided to improve this code and publish it as a library, hoping that it can be useful for their studies.

We later realized that the library had the potential to cover more areas than just bias measurement. This is why WEFE is constantly being improved, which so far has resulted in a new bias mitigation module and multiple enhancements and fixes.

The main objectives we want to achieve with this library are:

- To provide a ready-to-use tool that allows the user to run bias tests in a straightforward manner.
- To provide a ready-to-use tool that allows the user to mitigate bias by means of a simple *fit-transform* interface.
- To provide simple interface and utils to develop new metrics and mitigation methods.

1.2 Similar Packages

There are quite a few alternatives that complement WEFE. Be sure to check them out!

- Fair Embedding Engine: https://github.com/FEE-Fair-Embedding-Engine/FEE
- ResponsiblyAI: https://github.com/ResponsiblyAI/responsibly

1.3 Citation

Please cite the following paper if using this package in an academic publication:

P. Badilla, F. Bravo-Marquez, and J. Pérez WEFE: The Word Embeddings Fairness Evaluation Framework In Proceedings of the 29th In

The author's version can be found at the following link.

Bibtex:

```
@InProceedings{wefe2020,
    title     = {WEFE: The Word Embeddings Fairness Evaluation Framework},
    author    = {Badilla, Pablo and Bravo-Marquez, Felipe and Pérez, Jorge},
    booktitle = {Proceedings of the Twenty-Ninth International Joint Conference on
                 Artificial Intelligence, {IJCAI-20}},
    publisher = {International Joint Conferences on Artificial Intelligence Organization}
↪,
    pages     = {430--436},
    year      = {2020},
    month     = {7},
    doi       = {10.24963/ijcai.2020/60},
    url       = {https://doi.org/10.24963/ijcai.2020/60},
    }
```

1.4 Roadmap

We expect in the future to:

- Implement measurement framework for contextualized embedding models.
- Implement new queries on different criteria.
- Create a single script that evaluates different embedding models under different bias criteria.
- From the previous script, rank as many embeddings available on the web as possible.
- Implement a simple visualization module.
- Implement p-values mixin that applies for all metrics that accept two targets.

1.5 License

1.6 Team

- Pablo Badilla.
- Felipe Bravo-Marquez.
- Jorge Pérez.
- María José Zambrano.

1.6.1 Contributors

We thank all our contributors who have allowed WEFE to grow, especially stolenpyjak and mspl13 for implementing new metrics.

We also thank alan-cueva for initiating the development of metrics for contextualized embedding models and harshvr15 for the examples of multi-language bias measurement.

Thank you very much !

1.6.2 Contact

Please write to pablo.badilla at ug.chile.cl for inquiries about the software. You are also welcome to do a pull request or publish an issue in the WEFE repository on Github.

1.7 Acknowledgments

This work was funded by the Millennium Institute for Foundational Research on Data (IMFD). It is also sponsored by National Center of Artificial Intelligence of Chile (CENIA).

CHAPTER

TWO

QUICK START

In this tutorial we show you how to install WEFE and then how to run a basic query.

2.1 Download and setup

There are two different ways to install WEFE:

- To install the package with pip, run in a console:

```
pip install --upgrade wefe
```

- To install the package with conda, run in a console:

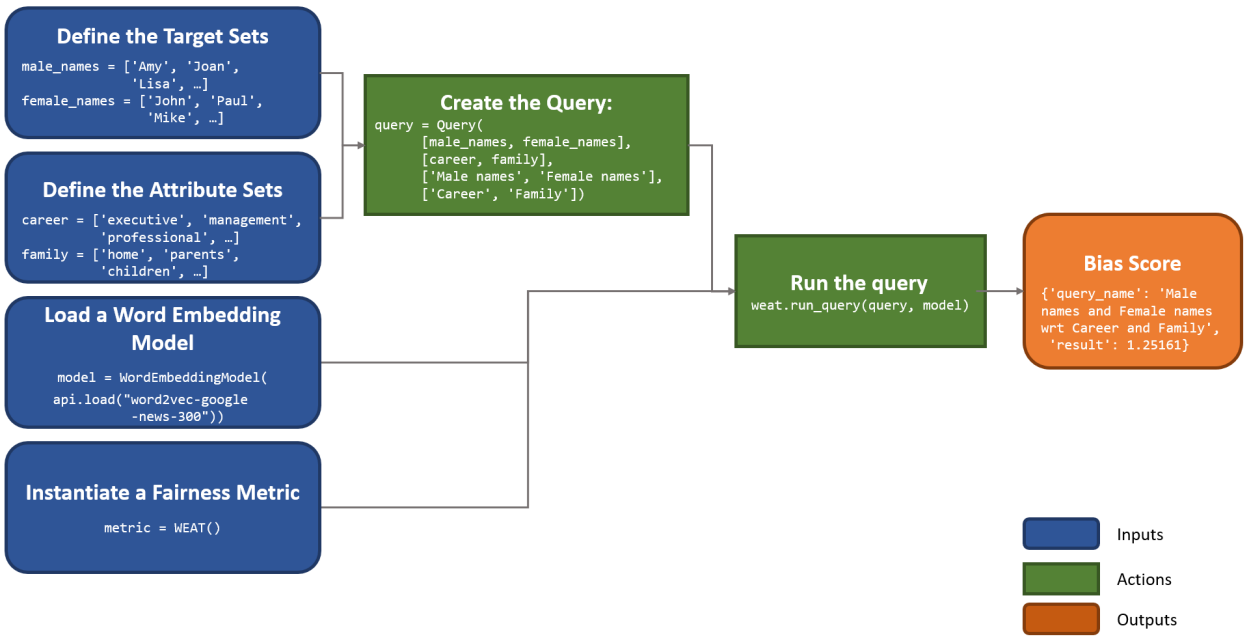
```
conda install -c pbadilla wefe
```

2.2 Run your first Query

Warning: If you are not familiar with the concepts of query, target and attribute set, please visit the the framework section on the library's about page. These concepts be widely used in the following sections.

In the following code we show how to implement the example query presented in WEFE's home page: A gender Query using WEAT metrics on the glove-twitter Word Embedding model.

The following graphic shows the flow of the query execution:



The programming of the previous flow can be separated into three steps:

- Load the Word Embedding model.
- Create the Query.
- Run the Query using the WEAT metric over the Word Embedding Model.

These stages be implemented next:

1. Load the Word Embedding pretrained model from `gensim` and then, create a :code:`` instance with it. This object took a gensim's `KeyedVectors` object and a model name as parameters. As we said previously, for this example, we use `glove-twitter-25'` embedding model.

```
>>> # import the modules
>>> from wefe.query import Query
>>> from wefe.word_embedding_model import WordEmbeddingModel
>>> from wefe.metrics.WEAT import WEAT
>>> import gensim.downloader as api
>>>
>>> # load glove
>>> twitter_25 = api.load('glove-twitter-25')
>>> model = WordEmbeddingModel(twitter_25, 'glove twitter dim=25')
```

2. Create the Query with a loaded, fetched or custom target and attribute word sets. In this case, we manually set both target words and attribute words.

```
>>> # create the word sets
>>> target_sets = [['she', 'woman', 'girl'], ['he', 'man', 'boy']]
>>> target_sets_names = ['Female Terms', 'Male Terms']
>>>
>>> attribute_sets = [['poetry','dance','literature'], ['math', 'physics', 'chemistry']]
>>> attribute_sets_names = ['Arts', 'Science']
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # create the query
>>> query = Query(target_sets, attribute_sets, target_sets_names,
>>>               attribute_sets_names)
```

3. Instantiate the metric to be used and then, execute `run_query` with the parameters created in the past steps. In this case we use the WEAT metric.

```
>>> # instance a WEAT metric
>>> weat = WEAT()
>>> result = weat.run_query(query, model)
>>> print(result)
{
  'query_name': 'Female Terms and Male Terms wrt Arts and Science',
  'result': 0.2595698336760204,
  'weat': 0.2595698336760204,
  'effect_size': 1.452482230821006,
  'p_value': nan
}
```

A score greater than 0 indicates that there is indeed a biased relationship between women and the arts with respect to men and science.

For more advanced usage, visit user the *Bias Measurement* in the User Guide.

CHAPTER

THREE

BIAS MEASUREMENT

The following guide is designed to present the more general details on using the package to measure bias. The following sections show:

- how to run a simple query using `Glove` embedding model.
- how to run multiple queries on multiple embeddings.
- how to compare the results obtained from running multiple sets of queries on multiple embeddings using different metrics through ranking calculation.
- how to calculate the correlations between the rankings obtained.

Warning: To accurately study and reduce biases contained in word embeddings, queries may contain words that could be offensive to certain groups or individuals. The relationships studied between these words DO NOT represent the ideas, thoughts or beliefs of the authors of this library. This warning applies to all documentation.

Note: If you are not familiar with the concepts of query, target and attribute set, please visit the *Measurement Framework* on the library's conceptual guides. These concepts are widely used in the following sections.

Note: For a list of metrics implemented in WEFE, refer to the *metrics section* of the API reference.

3.1 Run a Query

The following subsections explains how to run a simple query that measures gender bias on Glove. The example uses the Word Embedding Association Test (*WEAT*) metric quantifying the bias in the embeddings model. Below we show the three usual steps for performing a query in WEFE:

Note: *WEAT* is a fairness metric that quantifies the relationship between two sets of target words (sets of words intended to denote a social groups as men and women) and two sets of attribute words (sets of words representing some attitude, characteristic, trait, occupational field, etc. that can be associated with individuals from any social group).

The closer its value is to 0, the less biased the model is.

Visit the metrics documentation (*WEAT*) for more information.

3.1.1 Load a word embeddings model as a `WordEmbeddingModel` object

Load the word embedding model and then wrap it using a *`WordEmbeddingModel`* (class that allows WEFE to handle the models).

WEFE bases all its operations on word embeddings using Gensim's `KeyedVectors` interface. Any model that can be loaded using `KeyedVectors` will be compatible with WEFE. The following example uses a 25-dim pre-trained `Glove` model using a twitter dataset loaded using gensim-data.

Note: Visit gensim-data repository. to find the complete list of published pre-trained models ready to use.

```
import gensim.downloader as api

from wefe.datasets import load_weat
from wefe.metrics import WEAT
from wefe.query import Query
from wefe.word_embedding_model import WordEmbeddingModel

twitter_25 = api.load("glove-twitter-25")
# WordEmbeddingModel receives as first argument a KeyedVectors model
# and the second argument the model name.
model = WordEmbeddingModel(twitter_25, "glove twitter dim=25")
```

3.1.2 Create the query using a `Query` object

Define the target and attribute word sets and create a *`Query`* object that contains them.

For this initial example, a query is used to study the association between gender with respect to family and career. The words used are taken from the set of words used in the *Semantics derived automatically from language corpora contain human-like biases* paper, which are included in the `datasets` module.

```
gender_query = Query(
    target_sets=[
        ["female", "woman", "girl", "sister", "she", "her", "hers", "daughter"],
        ["male", "man", "boy", "brother", "he", "him", "his", "son"],
    ],
    attribute_sets=[
        [
            "home",
            "parents",
            "children",
            "family",
            "cousins",
            "marriage",
            "wedding",
            "relatives",
        ],
        [
            "executive",
            "management",
            "professional",
            "corporation",
```

(continues on next page)

(continued from previous page)

```
            "salary",
            "office",
            "business",
            "career",
        ],
    ],
    target_sets_names=["Female terms", "Male Terms"],
    attribute_sets_names=["Family", "Careers"],
)

gender_query
```

```
<Query: Female terms and Male Terms wrt Family and Careers
- Target sets: [['female', 'woman', 'girl', 'sister', 'she', 'her', 'hers', 'daughter'],␣
↪['male', 'man', 'boy', 'brother', 'he', 'him', 'his', 'son']]
- Attribute sets:[['home', 'parents', 'children', 'family', 'cousins', 'marriage',
↪'wedding', 'relatives'], ['executive', 'management', 'professional', 'corporation',
↪'salary', 'office', 'business', 'career']]>
```

3.1.3 Run the Query

Instantiate the metric that you will use and then execute `run_query` with the parameters created in the previous steps.

Any bias measurement process at WEFE consists of the following steps:

1. Metric arguments checking.
2. Transform the word sets into word embeddings.
3. Calculate the metric.

In this case we use the *WEAT* metric (proposed in the same paper of the set of words used in the query).

```
metric = WEAT()
result = metric.run_query(gender_query, model)
result
```

```
{'query_name': 'Female terms and Male Terms wrt Family and Careers',
 'result': 0.31658412935212255,
 'weat': 0.31658412935212255,
 'effect_size': 0.6779439085309583,
 'p_value': nan}
```

By default, the results are a `dict` containing the query name (in the key `query_name`) and the calculated value of the metric in the `result` key. It also contains a key with the name and the value of the calculated metric (which is duplicated in the "results" key).

Depending on the metric class used, the result `dict` can also return more metrics, detailed word-by-word values or other statistics like p-values. Also some metrics allow you to change the default value in results.

Details of all the metrics implemented, their parameters and examples of execution can be found at *metrics section*.

3.2 Run Query Arguments

Each metric allows varying the behavior of `run_query` according to different parameters. There are parameters to customize the transformation of the sets of words to sets of embeddings, others to warn errors or modify which calculation method the metric use.

Note: Each metric implements the `run_query` method with different arguments. Visit their API documentation for more information.

For example, `run_query` can be instructed to `return effect_size` in the `result` key by setting `return_effect_size` as `True`. Note that this parameter is only of the class `WEAT`.

```
weat = WEAT()
result = weat.run_query(gender_query, model, return_effect_size=True)
result
```

```
{'query_name': 'Female terms and Male Terms wrt Family and Careers',
 'result': 0.6779439085309583,
 'weat': 0.31658412935212255,
 'effect_size': 0.6779439085309583,
 'p_value': nan}
```

You can also request `run_query` to run the statistical significance calculation by setting `calculate_p_value` as `True`. This checks how many queries generated from permutations (controlled by the parameter `p_value_iterations`) of the target sets obtain values greater than those obtained by the original query.

```
weat = WEAT()
result = weat.run_query(
    gender_query, model, calculate_p_value=True, p_value_iterations=5000
)
result
```

```
{'query_name': 'Female terms and Male Terms wrt Family and Careers',
 'result': 0.31658412935212255,
 'weat': 0.31658412935212255,
 'effect_size': 0.6779439085309583,
 'p_value': 0.08418316336732654}
```

3.3 Out of Vocabulary Words and Word Preprocessors

It is common in the literature to find bias tests whose tagret sets are common names of social groups. These names are commonly cased and may contain special characters. There are several embedding models whose words are not cased or do not have accents or other special characters, as for example, in `Glove`. This implies that a query with target sets composed by names executed in `Glove` (without any preprocessing of the words) could produce erroneous results because WEFE will not be able to find the names in the model vocabulary.

Note: Some well-known word sets are already provided by the package and can be easily loaded by the user through the *datasets* module. From here on, the tutorial use the words defined in the study *Semantics derived automatically from language corpora contain human-like biases*, the same that proposed the *WEAT* metric.

```
# load the weat word sets.
word_sets = load_weat()

# print a set of european american common names.
print(word_sets["european_american_names_5"])
```

```
['Adam', 'Harry', 'Josh', 'Roger', 'Alan', 'Frank', 'Justin', 'Ryan', 'Andrew', 'Jack',
↪'Matthew', 'Stephen', 'Brad', 'Greg', 'Paul', 'Jonathan', 'Peter', 'Amanda', 'Courtney
↪', 'Heather', 'Melanie', 'Sara', 'Amber', 'Katie', 'Betsy', 'Kristin', 'Nancy',
↪'Stephanie', 'Ellen', 'Lauren', 'Colleen', 'Emily', 'Megan', 'Rachel']
```

The following query compares European-American and African-American names with respect to pleasant and unpleasant attributes.

Note: It can be indicated to `run_query` to log the words that were lost in the transformation to vectors by using the parameter `warn_not_found_words` as `True`.

```
ethnicity_query = Query(
    [word_sets["european_american_names_5"], word_sets["african_american_names_5"]],
    [word_sets["pleasant_5"], word_sets["unpleasant_5"]],
    ["European american names", "African american names"],
    ["Pleasant", "Unpleasant"],
)
result = weat.run_query(ethnicity_query, model, warn_not_found_words=True,)
result
```

```
WARNING:root:The following words from set 'European american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['Adam', 'Harry', 'Josh', 'Roger', 'Alan',
↪'Frank', 'Justin', 'Ryan', 'Andrew', 'Jack', 'Matthew', 'Stephen', 'Brad', 'Greg',
↪'Paul', 'Jonathan', 'Peter', 'Amanda', 'Courtney', 'Heather', 'Melanie', 'Sara', 'Amber
↪', 'Katie', 'Betsy', 'Kristin', 'Nancy', 'Stephanie', 'Ellen', 'Lauren', 'Colleen',
↪'Emily', 'Megan', 'Rachel']
WARNING:root:The transformation of 'European american names' into glove twitter dim=25␣
↪embeddings lost proportionally more words than specified in 'lost_words_threshold': 1.
↪0 lost with respect to 0.2 maximum loss allowed.
WARNING:root:The following words from set 'African american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['Alonzo', 'Jamel', 'Theo', 'Alphonse', 'Jerome
↪', 'Leroy', 'Torrance', 'Darnell', 'Lamar', 'Lionel', 'Tyree', 'Deion', 'Lamont',
↪'Malik', 'Terrence', 'Tyrone', 'Lavon', 'Marcellus', 'Wardell', 'Nichelle', 'Shereen',
↪'Ebony', 'Latisha', 'Shaniqua', 'Jasmine', 'Tanisha', 'Tia', 'Lakisha', 'Latoya',
↪'Yolanda', 'Malika', 'Yvette']
WARNING:root:The transformation of 'African american names' into glove twitter dim=25␣
↪embeddings lost proportionally more words than specified in 'lost_words_threshold': 1.
↪0 lost with respect to 0.2 maximum loss allowed.
ERROR:root:At least one set of 'European american names and African american names wrt␣
↪Pleasant and Unpleasant' query has proportionally fewer embeddings than allowed by the␣
↪lost_vocabulary_threshold parameter (0.2). This query will return np.nan.
```

```
{'query_name': 'European american names and African american names wrt Pleasant and␣
↪Unpleasant',
 'result': nan,
```

(continues on next page)

(continued from previous page)

```
 'weat': nan,
 'effect_size': nan}
```

Warning: If more than 20% of the words from any of the word sets of the query are lost during the transformation to embeddings, the result of the metric will be `np.nan`. This behavior can be changed using a float number parameter called `lost_vocabulary_threshold`.

3.3.1 Word Preprocessors

Any `run_query` method allows preprocessing each word before they are searched in the model's vocabulary through the parameter `preprocessors` (list of one or more preprocessor). This parameter accepts a list of individual preprocessors, which are defined below:

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before its looked up in the model vocabulary. For example, the `preprocessor {'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows searching for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}] {}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary.

By default (in case there is more than one preprocessor in the list) the first preprocessed word found in the embeddings model is used. This behavior can be controlled by the `strategy` parameter of `run_query`.

In the following example, we provide a list with only one preprocessor that instructs `run_query` to lowercase and remove all accents from every word before they are searched in the embeddings model.

```
weat = WEAT()
result = weat.run_query(
    ethnicity_query,
    model,
    preprocessors=[{"lowercase": True, "strip_accents": True}],
    warn_not_found_words=True,
)
result
```

```
WARNING:root:The following words from set 'African american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['wardell']
```

```
{'query_name': 'European american names and African american names wrt Pleasant and␣
↪Unpleasant',
 'result': 3.7529150679125456,
 'weat': 3.7529150679125456,
 'effect_size': 1.2746819330405683,
 'p_value': nan}
```

It may happen that it is more important to find the original word and in the case of not finding it, then preprocess it and look it up in the vocabulary. This behavior can be specified in `preprocessors` list by first specifying an empty preprocessor `{}` and then the preprocessor that converts to lowercase and removes accents.

```
weat = WEAT()
result = weat.run_query(
    ethnicity_query,
    model,
    preprocessors=[
        {},  # empty preprocessor, search for the original words.
        {
            "lowercase": True,
            "strip_accents": True,
        },  # search for lowercase and no accent words.
    ],
    warn_not_found_words=True,
)

result
```

```
WARNING:root:The following words from set 'European american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['Adam', 'Harry', 'Josh', 'Roger', 'Alan',
↪'Frank', 'Justin', 'Ryan', 'Andrew', 'Jack', 'Matthew', 'Stephen', 'Brad', 'Greg',
↪'Paul', 'Jonathan', 'Peter', 'Amanda', 'Courtney', 'Heather', 'Melanie', 'Sara', 'Amber
↪', 'Katie', 'Betsy', 'Kristin', 'Nancy', 'Stephanie', 'Ellen', 'Lauren', 'Colleen',
↪'Emily', 'Megan', 'Rachel']
WARNING:root:The following words from set 'African american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['Alonzo', 'Jamel', 'Theo', 'Alphonse', 'Jerome
↪', 'Leroy', 'Torrance', 'Darnell', 'Lamar', 'Lionel', 'Tyree', 'Deion', 'Lamont',
↪'Malik', 'Terrence', 'Tyrone', 'Lavon', 'Marcellus', 'Wardell', 'wardell', 'Nichelle',
↪'Shereen', 'Ebony', 'Latisha', 'Shaniqua', 'Jasmine', 'Tanisha', 'Tia', 'Lakisha',
↪'Latoya', 'Yolanda', 'Malika', 'Yvette']
```

```
{'query_name': 'European american names and African american names wrt Pleasant and␣
↪Unpleasant',
 'result': 3.7529150679125456,
 'weat': 3.7529150679125456,
 'effect_size': 1.2746819330405683,
 'p_value': nan}
```

The number of preprocessing steps can be increased as needed. For example, we can complex the above preprocessor to first search for the original words, then for the lowercase words, and finally for the lowercase words without accents.

```
weat = WEAT()
result = weat.run_query(
    ethnicity_query,
```

(continues on next page)

(continued from previous page)

```
    model,
    preprocessors=[
        {},  # first step: empty preprocessor, search for the original words.
        {"lowercase": True,},  # second step: search for lowercase.
        {
            "lowercase": True,
            "strip_accents": True,
        },  # third step: search for lowercase and no accent words.
    ],
    warn_not_found_words=True,
)

result
```

```
WARNING:root:The following words from set 'European american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['Adam', 'Harry', 'Josh', 'Roger', 'Alan',
↪'Frank', 'Justin', 'Ryan', 'Andrew', 'Jack', 'Matthew', 'Stephen', 'Brad', 'Greg',
↪'Paul', 'Jonathan', 'Peter', 'Amanda', 'Courtney', 'Heather', 'Melanie', 'Sara', 'Amber
↪', 'Katie', 'Betsy', 'Kristin', 'Nancy', 'Stephanie', 'Ellen', 'Lauren', 'Colleen',
↪'Emily', 'Megan', 'Rachel']
WARNING:root:The following words from set 'African american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['Alonzo', 'Jamel', 'Theo', 'Alphonse', 'Jerome
↪', 'Leroy', 'Torrance', 'Darnell', 'Lamar', 'Lionel', 'Tyree', 'Deion', 'Lamont',
↪'Malik', 'Terrence', 'Tyrone', 'Lavon', 'Marcellus', 'Wardell', 'wardell', 'wardell',
↪'Nichelle', 'Shereen', 'Ebony', 'Latisha', 'Shaniqua', 'Jasmine', 'Tanisha', 'Tia',
↪'Lakisha', 'Latoya', 'Yolanda', 'Malika', 'Yvette']
```

```
{'query_name': 'European american names and African american names wrt Pleasant and␣
↪Unpleasant',
 'result': 3.7529150679125456,
 'weat': 3.7529150679125456,
 'effect_size': 1.2746819330405683,
 'p_value': nan}
```

It is also possible to change the behavior of the search by including not only the first word, but all the words generated by the preprocessors. This can be controlled by specifying the parameter `strategy=all`.

```
weat = WEAT()
result = weat.run_query(
    ethnicity_query,
    model,
    preprocessors=[
        {},  # first step: empty preprocessor, search for the original words.
        {"lowercase": True,},  # second step: search for lowercase .
        {"uppercase": True,},  # third step: search for uppercase.
    ],
    strategy="all",
    warn_not_found_words=True,
)

result
```

```
WARNING:root:The following words from set 'European american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['Adam', 'ADAM', 'Harry', 'HARRY', 'Josh',
↪'JOSH', 'Roger', 'ROGER', 'Alan', 'ALAN', 'Frank', 'FRANK', 'Justin', 'JUSTIN', 'Ryan',
↪ 'RYAN', 'Andrew', 'ANDREW', 'Jack', 'JACK', 'Matthew', 'MATTHEW', 'Stephen', 'STEPHEN
↪', 'Brad', 'BRAD', 'Greg', 'GREG', 'Paul', 'PAUL', 'Jonathan', 'JONATHAN', 'Peter',
↪'PETER', 'Amanda', 'AMANDA', 'Courtney', 'COURTNEY', 'Heather', 'HEATHER', 'Melanie',
↪'MELANIE', 'Sara', 'SARA', 'Amber', 'AMBER', 'Katie', 'KATIE', 'Betsy', 'BETSY',
↪'Kristin', 'KRISTIN', 'Nancy', 'NANCY', 'Stephanie', 'STEPHANIE', 'Ellen', 'ELLEN',
↪'Lauren', 'LAUREN', 'Colleen', 'COLLEEN', 'Emily', 'EMILY', 'Megan', 'MEGAN', 'Rachel',
↪ 'RACHEL']
WARNING:root:The following words from set 'African american names' do not exist within␣
↪the vocabulary of glove twitter dim=25: ['Alonzo', 'ALONZO', 'Jamel', 'JAMEL', 'Theo',
↪'THEO', 'Alphonse', 'ALPHONSE', 'Jerome', 'JEROME', 'Leroy', 'LEROY', 'Torrance',
↪'TORRANCE', 'Darnell', 'DARNELL', 'Lamar', 'LAMAR', 'Lionel', 'LIONEL', 'Tyree', 'TYREE
↪', 'Deion', 'DEION', 'Lamont', 'LAMONT', 'Malik', 'MALIK', 'Terrence', 'TERRENCE',
↪'Tyrone', 'TYRONE', 'Lavon', 'LAVON', 'Marcellus', 'MARCELLUS', 'Wardell', 'wardell',
↪'WARDELL', 'Nichelle', 'NICHELLE', 'Shereen', 'SHEREEN', 'Ebony', 'EBONY', 'Latisha',
↪'LATISHA', 'Shaniqua', 'SHANIQUA', 'Jasmine', 'JASMINE', 'Tanisha', 'TANISHA', 'Tia',
↪'TIA', 'Lakisha', 'LAKISHA', 'Latoya', 'LATOYA', 'Yolanda', 'YOLANDA', 'Malika',
↪'MALIKA', 'Yvette', 'YVETTE']
WARNING:root:The following words from set 'Pleasant' do not exist within the vocabulary␣
↪of glove twitter dim=25: ['CARESS', 'FREEDOM', 'HEALTH', 'LOVE', 'PEACE', 'CHEER',
↪'FRIEND', 'HEAVEN', 'LOYAL', 'PLEASURE', 'DIAMOND', 'GENTLE', 'HONEST', 'LUCKY',
↪'RAINBOW', 'DIPLOMA', 'GIFT', 'HONOR', 'MIRACLE', 'SUNRISE', 'FAMILY', 'HAPPY',
↪'LAUGHTER', 'PARADISE', 'VACATION']
WARNING:root:The following words from set 'Unpleasant' do not exist within the␣
↪vocabulary of glove twitter dim=25: ['ABUSE', 'CRASH', 'FILTH', 'MURDER', 'SICKNESS',
↪'ACCIDENT', 'DEATH', 'GRIEF', 'POISON', 'STINK', 'ASSAULT', 'DISASTER', 'HATRED',
↪'POLLUTE', 'TRAGEDY', 'DIVORCE', 'JAIL', 'POVERTY', 'UGLY', 'CANCER', 'KILL', 'ROTTEN',
↪ 'VOMIT', 'AGONY', 'PRISON']
```

```
{'query_name': 'European american names and African american names wrt Pleasant and␣
↪Unpleasant',
 'result': 3.7529150679125456,
 'weat': 3.7529150679125456,
 'effect_size': 1.2746819330405683,
 'p_value': nan}
```

3.4 Running Multiple Queries

It is usual to want to test many queries of some bias criterion (gender, ethnicity, religion, politics, socioeconomic, among others) on several models at the same time. Trying to use `run_query` on each pair embedding-query can be a bit complex and could require extra work to implement.

This is why WEFE also implements a function to test multiple queries on various word embedding models in a single call: the *`run_queries`* util.

The following code shows how to run various gender queries on `Glove` embedding models with different dimensions trained from the Twitter dataset. The queries are executed using *`WEAT`* metric.

```
import gensim.downloader as api

from wefe.datasets import load_weat
from wefe.metrics import RNSB, WEAT
from wefe.query import Query
from wefe.utils import run_queries
from wefe.word_embedding_model import WordEmbeddingModel
```

3.4.1 Load the models

Load three different Glove Twitter embedding models. These models were trained using the same dataset varying the number of embedding dimensions.

```
model_1 = WordEmbeddingModel(api.load("glove-twitter-25"), "glove twitter dim=25")
model_2 = WordEmbeddingModel(api.load("glove-twitter-50"), "glove twitter dim=50")
model_3 = WordEmbeddingModel(api.load("glove-twitter-100"), "glove twitter dim=100")

models = [model_1, model_2, model_3]
```

3.4.2 Load the word sets and create the queries

Now, we load the *WEAT* word set and create three queries. The three queries are intended to measure gender bias.

```
# Load the WEAT word sets
word_sets = load_weat()

# Create gender queries
gender_query_1 = Query(
    [word_sets["male_terms"], word_sets["female_terms"]],
    [word_sets["career"], word_sets["family"]],
    ["Male terms", "Female terms"],
    ["Career", "Family"],
)

gender_query_2 = Query(
    [word_sets["male_terms"], word_sets["female_terms"]],
    [word_sets["science"], word_sets["arts"]],
    ["Male terms", "Female terms"],
    ["Science", "Arts"],
)

gender_query_3 = Query(
    [word_sets["male_terms"], word_sets["female_terms"]],
    [word_sets["math"], word_sets["arts_2"]],
    ["Male terms", "Female terms"],
    ["Math", "Arts"],
)

gender_queries = [gender_query_1, gender_query_2, gender_query_3]
```

3.4.3 Run the queries on all Word Embeddings using WEAT

To run the list of queries and models, we call *run_queries* using the parameters defined in the previous step. The mandatory parameters of the function are 3:

- a metric,
- a list of queries, and,
- a list of embedding models.

It is also possible to provide a name for the criterion studied in this set of queries through the parameter `queries_set_name`.

```
WEAT_gender_results = run_queries(
    WEAT, gender_queries, models, queries_set_name="Gender Queries"
)
WEAT_gender_results
```

```
WARNING:root:The transformation of 'Science' into glove twitter dim=25 embeddings lost␣
↪proportionally more words than specified in 'lost_words_threshold': 0.25 lost with␣
↪respect to 0.2 maximum loss allowed.
ERROR:root:At least one set of 'Male terms and Female terms wrt Science and Arts' query␣
↪has proportionally fewer embeddings than allowed by the lost_vocabulary_threshold␣
↪parameter (0.2). This query will return np.nan.
WARNING:root:The transformation of 'Science' into glove twitter dim=50 embeddings lost␣
↪proportionally more words than specified in 'lost_words_threshold': 0.25 lost with␣
↪respect to 0.2 maximum loss allowed.
ERROR:root:At least one set of 'Male terms and Female terms wrt Science and Arts' query␣
↪has proportionally fewer embeddings than allowed by the lost_vocabulary_threshold␣
↪parameter (0.2). This query will return np.nan.
WARNING:root:The transformation of 'Science' into glove twitter dim=100 embeddings lost␣
↪proportionally more words than specified in 'lost_words_threshold': 0.25 lost with␣
↪respect to 0.2 maximum loss allowed.
ERROR:root:At least one set of 'Male terms and Female terms wrt Science and Arts' query␣
↪has proportionally fewer embeddings than allowed by the lost_vocabulary_threshold␣
↪parameter (0.2). This query will return np.nan.
```

3.4.4 Setting metric params

There is a whole column that has no results. As the warnings point out, when transforming the words of the sets into embeddings, there is a loss of words that is greater than the allowed by the parameter `lost_vocabulary_threshold`. In this case, it would be very useful to use the word preprocessors seen above.

run_queries, accept specific parameters for each metric. These extra parameters for the metric can be passed through `metric_params` parameter. In this case, a `preprocessor` is provided to lowercase the words before searching for them in the models' vocabularies.

```
WEAT_gender_results = run_queries(
    WEAT,
    gender_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}]},
    queries_set_name="Gender Queries",
```

(continues on next page)

(continued from previous page)

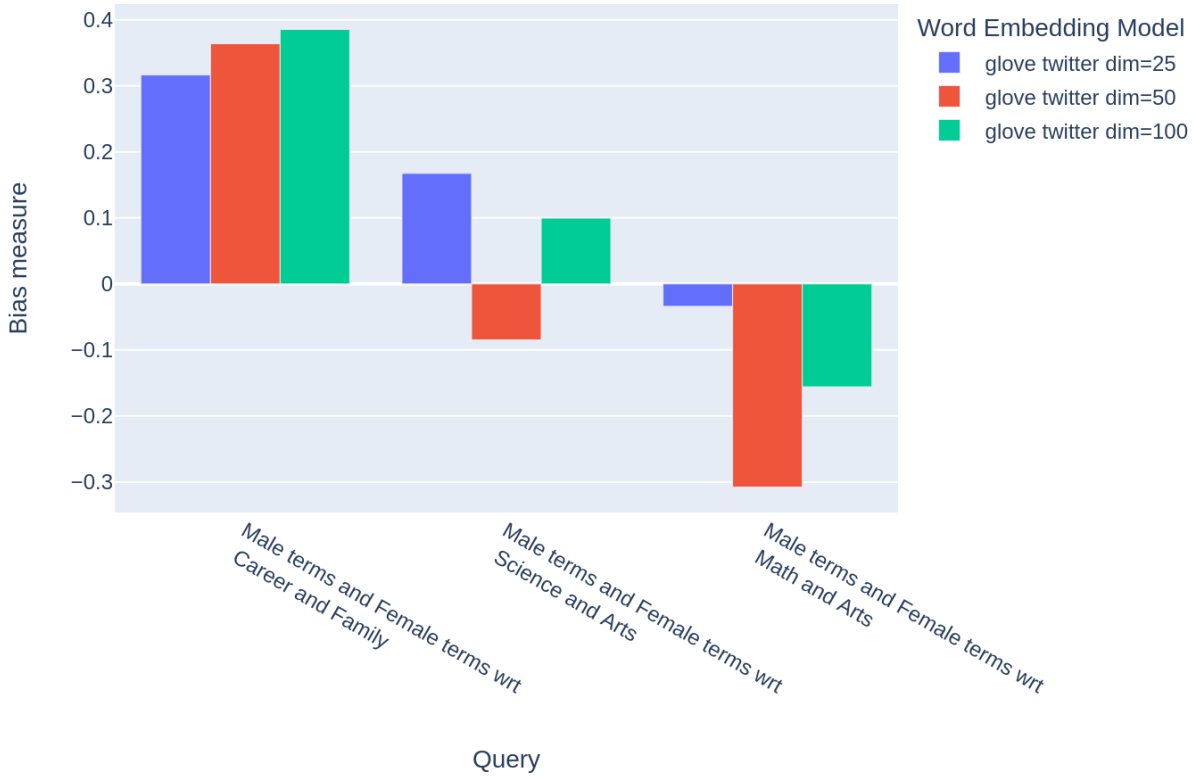
```
)

WEAT_gender_results
```

No query was null in these results.

3.4.5 Plot the results in a barplot

The library also provides an easy way to plot the results obtained from a `run_queries` execution into a plotly barplot.

```
from wefe.utils import plot_queries_results, run_queries

# Plot the results
plot_queries_results(WEAT_gender_results).show()
```



3.5 Aggregating Results

The execution of *run_queries* provided many results evaluating the gender bias in the tested embeddings. However, these results alone do not comprehensively report the biases observed in all of these queries. One way to obtain an overall view of bias is by aggregating results by model.

For WEAT, a simple way to aggregate the results is to average their absolute values. When running *run_queries*, it is possible to specify that the results be aggregated by model by setting `aggregate_results` as `True`

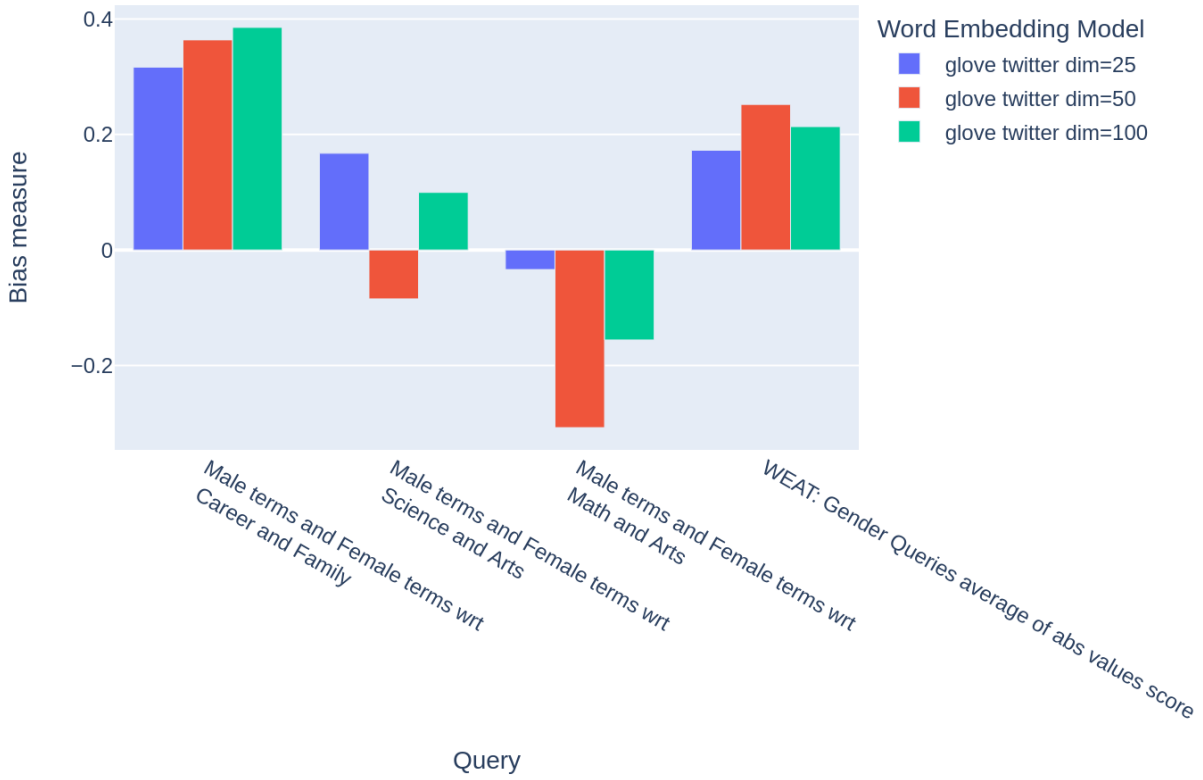
The aggregation function can be specified through the `aggregation_function` parameter. This parameter accepts a list of predefined aggregations as well as a custom function that operates on the results dataframe. The aggregation functions available are:

- Average `avg`.
- Average of the absolute values `abs_avg`.
- Sum `sum`.
- Sum of the absolute values, `abs_sum`.

Note: Notice that some functions are more appropriate for certain metrics. For metrics returning only positive numbers, all the previous aggregation functions would be OK. In contrast, metrics that return real values (e.g., *WEAT* , *RND* , etc...), aggregation functions such as sum would make positive and negative outputs to cancel each other.

```
WEAT_gender_results_agg = run_queries(
    WEAT,
    gender_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}]},
    aggregate_results=True,
    aggregation_function="abs_avg",
    queries_set_name="Gender Queries",
)
WEAT_gender_results_agg
```

```
plot_queries_results(WEAT_gender_results_agg).show()
```



It is also possible to ask the function to return only the aggregated results using the parameter `return_only_aggregation`

```
WEAT_gender_results_only_agg = run_queries(
    WEAT,
    gender_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}]},
    aggregate_results=True,
    aggregation_function="abs_avg",
    return_only_aggregation=True,
    queries_set_name="Gender Queries",
)
WEAT_gender_results_only_agg
```

```
fig = plot_queries_results(WEAT_gender_results_only_agg)
fig.show()
```

3.6 Model Ranking

It may be desirable to obtain an overall view of the bias by model using different metrics or bias criteria. While the aggregate values can be compared directly, two problems are likely to be encountered:

1. One type of bias criterion can dominate the other because of significant differences in magnitude.
2. Different metrics can operate on different scales, which makes them difficult to compare.

To show these problems, suppose we have:

- Two sets of queries: one that explores gender biases and another that explores ethnicity biases.
- Three `Glove` models of 25, 50 and 100 dimensions trained on the same twitter dataset.

Then we run *`run_queries`* on this set of model-queries using *`WEAT`*, and to corroborate the results obtained, we also use Relative Negative Sentiment Bias (*`RNSB`*).

1. The first problem occurs when the bias scores obtained from one set of queries are much higher than those from the other set, even when the same metric is used.

When executing *`run_queries`* with the gender and ethnicity queries on the models described above, the results obtained are as follows:

model_name	WEAT: Gender Queries average of abs values score	WEAT: Ethnicity Queries average of abs values score
glove twitter dim=25	0.210556	2.64632
glove twitter dim=50	0.292373	1.87431
glove twitter dim=100	0.225116	1.78469

As can be seen, the results of ethnicity bias are much greater than those of gender.

2. The second problem is when different metrics return results on different scales of magnitude.

When executing *run_queries* with the gender queries and models described above using both WEAT and RNSB, the results obtained are as follows:

model_name	WEAT: Gender Queries average of abs values score	RNSB: Gender Queries average of abs values score
glove twitter dim=25	0.210556	0.032673
glove twitter dim=50	0.292373	0.049429
glove twitter dim=100	0.225116	0.0312772

We can see differences between the results of both metrics of an order of magnitude.

One solution to this problem is to create **rankings**. Rankings focus on the relative differences reported by the metrics (for different models) instead of focusing on the absolute values.

The following guide show how to create rankings that evaluate gender bias and ethnicity.

3.6.1 Gender Bias Model Ranking

```
# define the queries
gender_query_1 = Query(
    [word_sets["male_terms"], word_sets["female_terms"]],
    [word_sets["career"], word_sets["family"]],
    ["Male terms", "Female terms"],
    ["Career", "Family"],
)
gender_query_2 = Query(
    [word_sets["male_terms"], word_sets["female_terms"]],
    [word_sets["science"], word_sets["arts"]],
    ["Male terms", "Female terms"],
    ["Science", "Arts"],
)
gender_query_3 = Query(
    [word_sets["male_terms"], word_sets["female_terms"]],
    [word_sets["math"], word_sets["arts_2"]],
    ["Male terms", "Female terms"],
    ["Math", "Arts"],
```

(continues on next page)

(continued from previous page)

```
)

gender_queries = [gender_query_1, gender_query_2, gender_query_3]

# run the queries using WEAT
WEAT_gender_results = run_queries(
    WEAT,
    gender_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}]},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name="Gender Queries",
)

# run the queries using WEAT effect size
WEAT_EZ_gender_results = run_queries(
    WEAT,
    gender_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}], "return_effect_size": True,},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name="Gender Queries",
)

# run the queries using RNSB
RNSB_gender_results = run_queries(
    RNSB,
    gender_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}]},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name="Gender Queries",
)
```

The rankings can be calculated by means of the *create_ranking* function. This function receives as input results from running *run_queries* and assumes that the last column contains the aggregated values.

```
from wefe.utils import create_ranking

# create the ranking
gender_ranking = create_ranking(
    [WEAT_gender_results, WEAT_EZ_gender_results, RNSB_gender_results]
)

gender_ranking
```

3.6.2 Ethnicity Bias Model Ranking

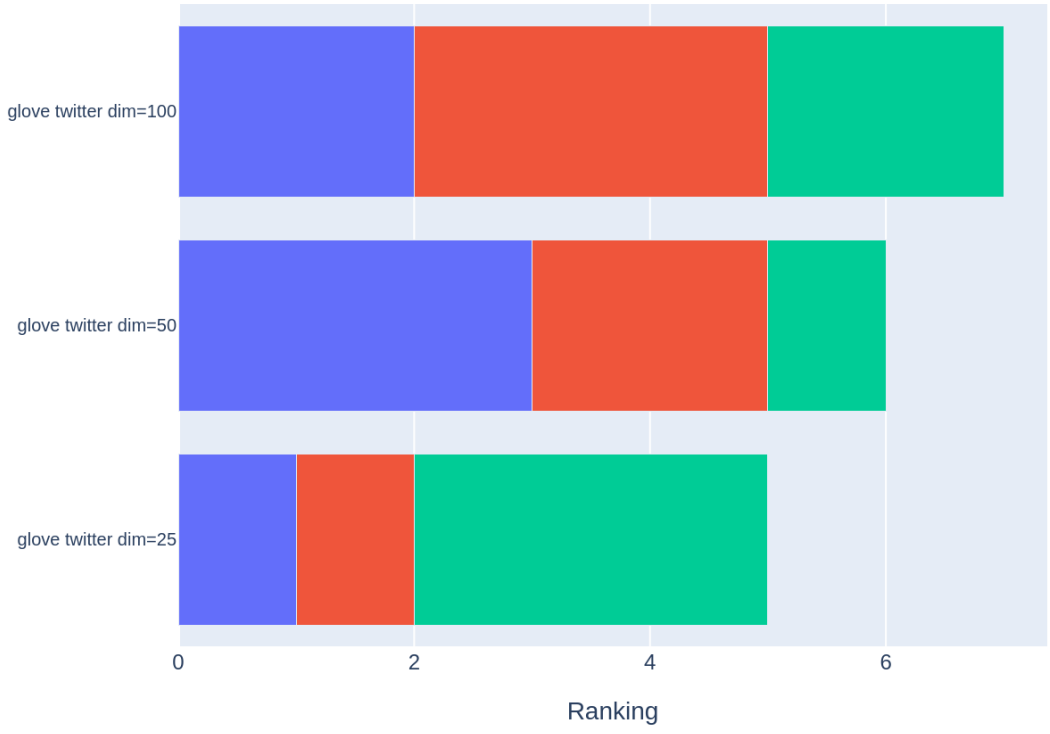
```
# define the queries
ethnicity_query_1 = Query(
    [word_sets["european_american_names_5"], word_sets["african_american_names_5"]],
    [word_sets["pleasant_5"], word_sets["unpleasant_5"]],
    ["European Names", "African Names"],
    ["Pleasant", "Unpleasant"],
)

ethnicity_query_2 = Query(
    [word_sets["european_american_names_7"], word_sets["african_american_names_7"]],
    [word_sets["pleasant_9"], word_sets["unpleasant_9"]],
    ["European Names", "African Names"],
    ["Pleasant 2", "Unpleasant 2"],
)

ethnicity_queries = [ethnicity_query_1, ethnicity_query_2]

# run the queries using WEAT
WEAT_ethnicity_results = run_queries(
    WEAT,
    ethnicity_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}]},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name="Ethnicity Queries",
)

# run the queries using WEAT effect size
WEAT_EZ_ethnicity_results = run_queries(
    WEAT,
    ethnicity_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}], "return_effect_size": True,},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name="Ethnicity Queries",
)

# run the queries using RNSB
RNSB_ethnicity_results = run_queries(
    RNSB,
    ethnicity_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}]},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name="Ethnicity Queries",
)
```

```
# create the ranking
ethnicity_ranking = create_ranking(
    [WEAT_ethnicity_results, WEAT_EZ_gender_results, RNSB_ethnicity_results]
)

ethnicity_ranking
```

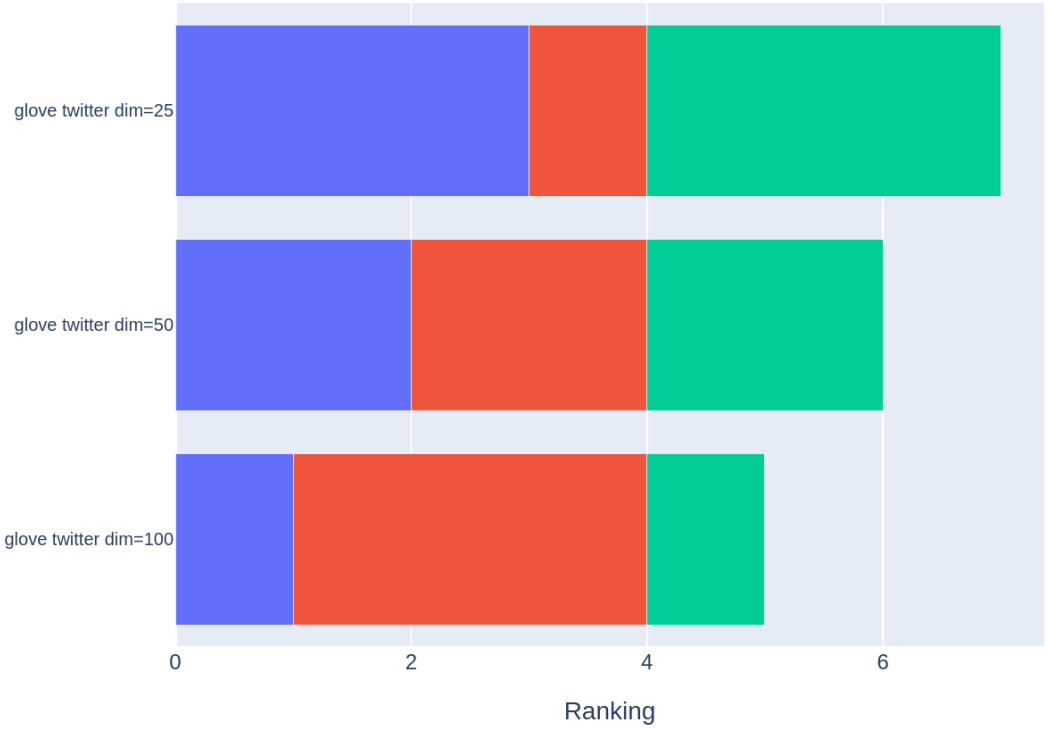
3.6.3 Plotting the rankings

It is possible to graph the rankings in barplots using the *plot_ranking* function. The generated figure shows the accumulated rankings for each embedding model. Each bar represents the sum of the rankings obtained by each embedding. Each color within a bar represents a different criterion-metric ranking.

```
from wefe.utils import plot_ranking

fig = plot_ranking(gender_ranking)
fig.show()
```




```
fig = plot_ranking(ethnicity_ranking)
fig.show()
```



3.6.4 Correlating Rankings

Having obtained rankings by metric for each embeddings, it would be ideal to see and analyze the degree of agreement between them.

A high concordance between the rankings allows us to state with some certainty that all metrics evaluated the embedding models in a similar way and therefore, that the ordering of embeddings by bias calculated makes sense. On the other hand, a low degree of agreement shows the opposite: the rankings do not allow to clearly establish which embedding is less biased than another.

The level of concordance of the rankings can be evaluated by calculating correlations.WEFE provides *calculate_ranking_correlations* to calculate the correlations between rankings.

```
from wefe.utils import calculate_ranking_correlations, plot_ranking_correlations

correlations = calculate_ranking_correlations(gender_ranking)
correlations
```

Note: `calculate_ranking_correlations` uses the `corr()` `pandas` dataframe method. The type of correlation that is calculated can be changed through the method parameter. The available options are: `'pearson'`, `'spearman'`, `'kendall'`. By default, the spearman correlation is calculated.

In this example, Kendall's correlation is used.

```
calculate_ranking_correlations(gender_ranking, method="kendall")
```

WEFE also provides a function for graphing the correlations:

```
correlation_fig = plot_ranking_correlations(correlations)
correlation_fig.show()
```

In this case, only two of the three rankings show similar results.

CHAPTER

FOUR

BIAS MITIGATION (DEBIAS)

The following guide is designed to present the more general details on using the package to mitigate (debias) bias in word embedding models. The following sections show:

- run *HardDebias* mitigation method on an embedding model to mitigate gender bias (using the `fit-transform` interface).
- apply the `target` parameter when executing the transformation.
- apply the `ignore` parameter when executing the transformation.
- apply the `copy` parameter when executing the transformation.
- run *MulticlassHardDebias* mitigation method on an word embedding model to mitigate ethnic bias.

Note: For a list of metrics implemented in WEFE, refer to the *Debias* of the API reference.

Note: If you want to know more about WEFE's standardization of debias methods, visit *Mitigation Framework* in the conceptual guides.

4.1 Hard Debias

Hard debias is a method that allows mitigating biases through geometric operations on embeddings. This method is binary because it only allows 2 classes of the same bias criterion, such as male or female.

Note: For a multiclass debias (such as for Latinos, Asians and Whites), it is recommended to visit *MulticlassHardDebias* class.

The main idea of this method is:

1. Identify a bias subspace through the defining sets. In the case of gender, these could be e.g. `[['woman', 'man'], ['she', 'he'], ...]`

2. Neutralize the bias subspace of embeddings that should not be biased.

First, we define a set of words that are correct to be related to the bias criterion: the *criterion specific gender words*. For example, in the case of gender, *gender specific* words are: `['he', 'his', 'He', 'her', 'she', 'him', 'him', 'She', 'man', 'women', 'men'...]`.

We then define that all words outside this set should have no relation to the bias criterion and thus have the possibility of being biased. (e.g. for the case of genthe bias direction, such that neither is closer to the bias direction than the other:

`['doctor', 'nurse', ...]`). Therefore, this set of words is neutralized with respect to the bias subspace found in the previous step.

The neutralization is carried out under the following operation:

- u : embedding
- v: bias direction

First calculate the projection of the embedding on the bias subspace.

$$\text{bias subspace} = \frac{v \cdot (v \cdot u)}{(v \cdot v)}$$

Then subtract the projection from the embedding.

$$u' = u - \text{bias subspace}$$

3. Equalizate the embeddings with respect to the bias direction.

Given an equalization set (set of word pairs such as `['she', 'he'], ['men', 'women'], ...`, but not limited to the definitional set) this step executes, for each pair, an equalization with respect to the bias direction. That is, it takes both embeddings of the pair and distributes them at the same distance from the bias direction, so that neither is closer to the bias direction than the other.

The fit parameters define how the neutralization will be calculated. In Hard Debias, you have to provide the the `definitional_pairs`, the `equalize_pairs` (which could be the same of definitional pairs) and optionally, a debias `criterion_name` (to name the debiased model).

The code shown below shows how to run Hard Debias from gender to the test model provided by wefe (reduced word2vec).

```
from wefe.utils import load_test_model

model = load_test_model()  # load a reduced version of word2vec
model
```

```
<wefe.word_embedding_model.WordEmbeddingModel at 0x7f3b83b3b700>
```

Load the required word sets.

```
from wefe.datasets import fetch_debiaswe
from wefe.debias.hard_debias import HardDebias

debiaswe_wordsets = fetch_debiaswe()

definitional_pairs = debiaswe_wordsets["definitional_pairs"]
equalize_pairs = debiaswe_wordsets["equalize_pairs"]
gender_specific = debiaswe_wordsets["gender_specific"]


print(f"definitional_pairs: \n{definitional_pairs}")
print(f"equalize_pairs: \n{equalize_pairs}")
print(f"gender_specific: \n{gender_specific}")
print("-" * 70, "\n")
```

```
definitional_pairs:
[['woman', 'man'], ['girl', 'boy'], ['she', 'he'], ['mother', 'father'], ['daughter',
↪'son'], ['gal', 'guy'], ['female', 'male'], ['her', 'his'], ['herself', 'himself'], [
↪'Mary', 'John']]
equalize_pairs:
[['monastery', 'convent'], ['spokesman', 'spokeswoman'], ['Catholic_priest', 'nun'], [
↪'Dad', 'Mom'], ['Men', 'Women'], ['councilman', 'councilwoman'], ['grandpa', 'grandma
↪'], ['grandsons', 'granddaughters'], ['prostate_cancer', 'ovarian_cancer'], [
↪'testosterone', 'estrogen'], ['uncle', 'aunt'], ['wives', 'husbands'], ['Father',
↪'Mother'], ['Grandpa', 'Grandma'], ['He', 'She'], ['boy', 'girl'], ['boys', 'girls'], [
↪'brother', 'sister'], ['brothers', 'sisters'], ['businessman', 'businesswoman'], [
↪'chairman', 'chairwoman'], ['colt', 'filly'], ['congressman', 'congresswoman'], ['dad',
↪ 'mom'], ['dads', 'moms'], ['dudes', 'gals'], ['ex_girlfriend', 'ex_boyfriend'], [
↪'father', 'mother'], ['fatherhood', 'motherhood'], ['fathers', 'mothers'], ['fella',
↪'granny'], ['fraternity', 'sorority'], ['gelding', 'mare'], ['gentleman', 'lady'], [
↪'gentlemen', 'ladies'], ['grandfather', 'grandmother'], ['grandson', 'granddaughter'],␣
↪['he', 'she'], ['himself', 'herself'], ['his', 'her'], ['king', 'queen'], ['kings',
↪'queens'], ['male', 'female'], ['males', 'females'], ['man', 'woman'], ['men', 'women
↪'], ['nephew', 'niece'], ['prince', 'princess'], ['schoolboy', 'schoolgirl'], ['son',
↪'daughter'], ['sons', 'daughters'], ['twin_brother', 'twin_sister']]
gender_specific:
['he', 'his', 'He', 'her', 'she', 'him', 'She', 'man', 'women', 'men', 'His', 'woman',
↪'spokesman', 'wife', 'himself', 'son', 'mother', 'father', 'chairman', 'daughter',
↪'husband', 'guy', 'girls', 'girl', 'Her', 'boy', 'King', 'boys', 'brother', 'Chairman',
↪ 'spokeswoman', 'female', 'sister', 'Women', 'Man', 'male', 'herself', 'Lions', 'Lady',
↪ 'brothers', 'dad', 'actress', 'mom', 'sons', 'girlfriend', 'Kings', 'Men', 'daughters
↪', 'Prince', 'Queen', 'teenager', 'lady', 'Bulls', 'boyfriend', 'sisters', 'Colts',
↪'mothers', 'Sir', 'king', 'businessman', 'Boys', 'grandmother', 'grandfather', 'deer',
↪'cousin', 'Woman', 'ladies', 'Girls', 'Father', 'uncle', 'PA', 'Boy', 'Councilman',
↪'mum', 'Brothers', 'MA', 'males', 'Girl', 'Mom', 'Guy', 'Queens', 'congressman', 'Dad',
↪ 'Mother', 'grandson', 'twins', 'bull', 'queen', 'businessmen', 'wives', 'widow',
↪'nephew', 'bride', 'females', 'aunt', 'Congressman', 'prostate_cancer', 'lesbian',
↪'chairwoman', 'fathers', 'Son', 'moms', 'Ladies', 'maiden', 'granddaughter', 'younger_
↪brother', 'Princess', 'Guys', 'lads', 'Ma', 'Sons', 'lion', 'Bachelor', 'gentleman',
↪'fraternity', 'bachelor', 'niece', 'Lion', 'Sister', 'bulls', 'husbands', 'prince',
↪'colt', 'salesman', 'Bull', 'Sisters', 'hers', 'dude', 'Spokesman', 'beard', 'filly',
↪'Actress', 'Him', 'princess', 'Brother', 'lesbians', 'councilman', 'actresses', 'Viagra
↪', 'gentlemen', 'stepfather', 'Deer', 'monks', 'Beard', 'Uncle', 'ex_girlfriend', 'lad
↪', 'sperm', 'Daddy', 'testosterone', 'MAN', 'Female', 'nephews', 'maid', 'daddy', 'mare
↪', 'fiance', 'Wife', 'fiancee', 'kings', 'dads', 'waitress', 'Male', 'maternal',
↪'heroine', 'feminist', 'Mama', 'nieces', 'girlfriends', 'Councilwoman', 'sir', 'stud',
↪'Mothers', 'mistress', 'lions', 'estranged_wife', 'womb', 'Brotherhood', 'Statesman',
↪'grandma', 'maternity', 'estrogen', 'ex_boyfriend', 'widows', 'gelding', 'diva',
↪'teenage_girls', 'nuns', 'Daughter', 'czar', 'ovarian_cancer', 'HE', 'Monk',
↪'countrymen', 'Grandma', 'teenage_girl', 'penis', 'bloke', 'nun', 'Husband', 'brides',
↪'housewife', 'spokesmen', 'suitors', 'menopause', 'monastery', 'patriarch', 'Beau',
↪'motherhood', 'brethren', 'stepmother', 'Dude', 'prostate', 'Moms', 'hostess', 'twin_
↪brother', 'Colt', 'schoolboy', 'eldest', 'brotherhood', 'Godfather', 'fillies',
↪'stepson', 'congresswoman', 'Chairwoman', 'Daughters', 'uncles', 'witch', 'Mommy',
↪'monk', 'viagra', 'paternity', 'suitor', 'chick', 'Pa', 'fiancé', 'sorority', 'macho',
↪'Spokeswoman', 'businesswoman', 'eldest_son', 'gal', 'statesman', 'schoolgirl',
↪'fathered', 'goddess', 'hubby', 'mares', 'stepdaughter', 'blokes', 'dudes', 'socialite
↪', 'strongman', 'Witch', 'fiancée', 'uterus', 'grandsons', 'Bride', 'studs', 'mama',
↪'Aunt', 'godfather', 'hens', 'hen', 'mommy', 'Babe', 'estranged_husband', 'fathers',
↪'elder_brother', 'boyhood', 'baritone', 'Diva', 'Lesbian', 'grandmothers', 'grandpa',
↪'boyfriends', 'feminism', 'countryman', 'stallion', 'heiress', 'queens', 'Grandpa',
↪'witches', 'aunts', 'semen', 'fella', 'granddaughters', 'chap', 'knight', 'widower',
↪'Maiden', 'salesmen', 'convent', 'KING', 'vagina', 'beau', 'babe', 'HIS', 'beards',
↪'handyman', 'twin_sister', 'maids', 'gals', 'housewives', 'Gentlemen', 'horsemen',
↪'Businessman', 'obstetrics', 'fatherhood', 'beauty_queen', 'councilwoman', 'princes',
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------
```

Instantiate and fit the parameters of the debias transformation. In the fit stage, parameters such as bias direction are calculated and embeddings are prepared for the equalization stage.

```
hd = HardDebias(verbose=False, criterion_name="gender")

hd.fit(
    model, definitional_pairs=definitional_pairs, equalize_pairs=equalize_pairs,
)
```

```
<wefe.debias.hard_debias.HardDebias at 0x7f3b810ad6d0>
```

4.1.1 Mitigation Parameters

The parameters of the transform method are relatively standard for all methods. The most important ones are `target`, `ignore` and `copy`.

In the following example we use `ignore` and `copy`, which are described below:

- `ignore` (by default, `None`):

  A list of strings that indicates that the debias method will perform the debias in all words except those specified in this list. In case it is not specified, debias will be executed on all words. In case ignore is not specified or its value is None, the transformation will be performed on all embeddings. This may cause words that are specific to social groups to lose that component (for example, leaving `'she'` and `'he'` without a gender component).

- `copy` (by default `True`):

  if the value of copy is `True`, method attempts to create a copy of the model and run debias on the copy. If `False`, the method is applied on the original model, causing the vectors to mutate.

Warning: WARNING:** Setting copy with `True` requires at least 2x RAM of the size of the model. Otherwise the execution of the debias may raise `MemoryError`.

The following transformation is executed using a copy of the model, ignoring the words contained in `gender_specific`.

```
gender_debiased_model = hd.transform(model, ignore=gender_specific, copy=True)
```

```
Copy argument is True. Transform will attempt to create a copy of the original model.␣
↪This may fail due to lack of memory.
Model copy created successfully.
```

```
100%|| 13013/13013 [00:00<00:00, 118668.18it/s]
```

4.1.2 Measuring the Decrease of Bias

Using the metrics and queries shown in the *Bias Measurement* user guide, we can measure whether there was a change in the measured gender bias between the original model and the debiased model.

```
from wefe.datasets import load_weat
from wefe.query import Query
from wefe.metrics import WEAT

weat_wordset = load_weat()
weat = WEAT()
```

Next, we measure the gender bias exposed by query 1 (Male terms and Female terms wrt Career and Family) with respect to the debiased model and the original.

```
gender_query_1 = Query(
    [weat_wordset["male_terms"], weat_wordset["female_terms"]],
    [weat_wordset["career"], weat_wordset["family"]],
    ["Male terms", "Female terms"],
    ["Career", "Family"],
)
print(gender_query_1, "\n", "-" * 70, "\n")

biased_results_1 = weat.run_query(gender_query_1, model, normalize=True)
debiased_results_1 = weat.run_query(
    gender_query_1, gender_debiased_model, normalize=True
)

print("Debiased vs Biased (absolute values)")
print(
    round(abs(debiased_results_1["weat"]), 3),
    "<",
    round(abs(biased_results_1["weat"]), 3),
)
```

```
<Query: Male terms and Female terms wrt Career and Family
- Target sets: [['male', 'man', 'boy', 'brother', 'he', 'him', 'his', 'son'], ['female',
↪'woman', 'girl', 'sister', 'she', 'her', 'hers', 'daughter']]
- Attribute sets:[['executive', 'management', 'professional', 'corporation', 'salary',
↪'office', 'business', 'career'], ['home', 'parents', 'children', 'family', 'cousins',
↪'marriage', 'wedding', 'relatives']]>
 ----------------------------------------------------------------------

Debiased vs Biased (absolute values)
0.047 < 0.463
```

The above results show that there was a decrease in the measured gender bias.

Next, we measure the gender bias exposed by query 2 (Male Names and Female Names wrt Pleasant and Unpleasant terms) with respect to the debiased model and the original.

```
gender_query_2 = Query(
    [weat_wordset["male_names"], weat_wordset["female_names"]],
    [weat_wordset["pleasant_5"], weat_wordset["unpleasant_5"]],
```

(continues on next page)

(continued from previous page)

```
    ["Male Names", "Female Names"],
    ["Pleasant", "Unpleasant"],
)

print(gender_query_2, "\n", "-" * 70, "\n")

biased_results_2 = weat.run_query(
    gender_query_2, model, normalize=True, preprocessors=[{}, {"lowercase": True}]
)
debiased_results_2 = weat.run_query(
    gender_query_2,
    gender_debiased_model,
    normalize=True,
    preprocessors=[{}, {"lowercase": True}],
)

print("Debiased vs Biased (absolute values)")
print(
    round(abs(debiased_results_2["weat"]), 3),
    "<",
    round(abs(biased_results_2["weat"]), 3),
)
```

```
<Query: Male Names and Female Names wrt Pleasant and Unpleasant
- Target sets: [['John', 'Paul', 'Mike', 'Kevin', 'Steve', 'Greg', 'Jeff', 'Bill'], ['Amy
↪', 'Joan', 'Lisa', 'Sarah', 'Diana', 'Kate', 'Ann', 'Donna']]
- Attribute sets:[['caress', 'freedom', 'health', 'love', 'peace', 'cheer', 'friend',
↪'heaven', 'loyal', 'pleasure', 'diamond', 'gentle', 'honest', 'lucky', 'rainbow',
↪'diploma', 'gift', 'honor', 'miracle', 'sunrise', 'family', 'happy', 'laughter',
↪'paradise', 'vacation'], ['abuse', 'crash', 'filth', 'murder', 'sickness', 'accident',
↪'death', 'grief', 'poison', 'stink', 'assault', 'disaster', 'hatred', 'pollute',
↪'tragedy', 'divorce', 'jail', 'poverty', 'ugly', 'cancer', 'kill', 'rotten', 'vomit',
↪'agony', 'prison']]>
 ----------------------------------------------------------------------

Debiased vs Biased (absolute values)
0.055 < 0.074
```

Again, the above results show that there was a decrease in the measured gender bias.

4.1.3 Target Parameter

If a set of words is specified in `target` parameter, the debias method is performed only on the embeddings associated with this set. In the case of providing `None`, the transformation is performed on all vocabulary words except those specified in ignore. By default `None`.

In the following example, the target parameter is used to execute the transformation only on the career and family word set:

```
targets = [
    "executive",
```

(continues on next page)

(continued from previous page)

```
    "management",
    "professional",
    "corporation",
    "salary",
    "office",
    "business",
    "career",
    "home",
    "parents",
    "children",
    "family",
    "cousins",
    "marriage",
    "wedding",
    "relatives",
]

hd = HardDebias(verbose=False, criterion_name="gender").fit(
    model, definitional_pairs=definitional_pairs, equalize_pairs=equalize_pairs,
)

gender_debiased_model = hd.transform(model, target=targets, copy=True)
```

```
Copy argument is True. Transform will attempt to create a copy of the original model.␣
↪This may fail due to lack of memory.
Model copy created successfully.
```

```
100%|| 16/16 [00:00<00:00, 9428.05it/s]
```

Next, a bias test is run on the mitigated embeddings associated with the target words.

In this case, the value of the metric is lower on the query executed on the mitigated model than on the original one. These results indicate that there was a mitigation of bias on embeddings of these words.

```
gender_query_1 = Query(
    [weat_wordset["male_terms"], weat_wordset["female_terms"]],
    [weat_wordset["career"], weat_wordset["family"]],
    ["Male terms", "Female terms"],
    ["Career", "Family"],
)
print(gender_query_1, "\n", "-" * 70, "\n")

biased_results_1 = weat.run_query(gender_query_1, model, normalize=True)
debiased_results_1 = weat.run_query(
    gender_query_1, gender_debiased_model, normalize=True
)

print("Debiased vs Biased (absolute values)")
print(
    round(abs(debiased_results_1["weat"]), 3),
    "<",
    round(abs(biased_results_1["weat"]), 3),
```

(continues on next page)

(continued from previous page)

```
)
```

```
<Query: Male terms and Female terms wrt Career and Family
- Target sets: [['male', 'man', 'boy', 'brother', 'he', 'him', 'his', 'son'], ['female',
↪'woman', 'girl', 'sister', 'she', 'her', 'hers', 'daughter']]
- Attribute sets:[['executive', 'management', 'professional', 'corporation', 'salary',
↪'office', 'business', 'career'], ['home', 'parents', 'children', 'family', 'cousins',
↪'marriage', 'wedding', 'relatives']]>
 ----------------------------------------------------------------------

Debiased vs Biased (absolute values)
0.047 < 0.463
```

However, if a bias test is run with words that were outside the `target` word set, the results are almost the same. The slight difference in the metric scores lies in the fact that the equalize sets were still equalized.

Warning: The equalization process can modify embeddings that have not been marked in the target. In Hard Debias, equalization can be deactivated by delivering an empty equalize set (`[]`).

```
gender_query_2 = Query(
    [weat_wordset["male_names"], weat_wordset["female_names"]],
    [weat_wordset["pleasant_5"], weat_wordset["unpleasant_5"]],
    ["Male Names", "Female Names"],
    ["Pleasant", "Unpleasant"],
)

print(gender_query_2, "\n", "-" * 70, "\n")

biased_results_2 = weat.run_query(
    gender_query_2, model, normalize=True, preprocessors=[{}, {"lowercase": True}]
)
debiased_results_2 = weat.run_query(
    gender_query_2,
    gender_debiased_model,
    normalize=True,
    preprocessors=[{}, {"lowercase": True}],
)

print("Debiased vs Biased (absolute values)")
print(
    round(abs(debiased_results_2["weat"]), 3),
    ">",
    round(abs(biased_results_2["weat"]), 3),
)
```

```
<Query: Male Names and Female Names wrt Pleasant and Unpleasant
- Target sets: [['John', 'Paul', 'Mike', 'Kevin', 'Steve', 'Greg', 'Jeff', 'Bill'], ['Amy
↪', 'Joan', 'Lisa', 'Sarah', 'Diana', 'Kate', 'Ann', 'Donna']]
- Attribute sets:[['caress', 'freedom', 'health', 'love', 'peace', 'cheer', 'friend',
↪'heaven', 'loyal', 'pleasure', 'diamond', 'gentle', 'honest', 'lucky', 'rainbow',
↪'diploma', 'gift', 'honor', 'miracle', 'sunrise', 'family', 'happy', 'laughter',
↪'paradise', 'vacation'], ['abuse', 'crash', 'filth', 'murder', 'sickness', 'accident',
↪'death', 'grief', 'poison', 'stink', 'assault', 'disaster', 'hatred', 'pollute',
↪'tragedy', 'divorce', 'jail', 'poverty', 'ugly', 'cancer', 'kill', 'rotten', 'vomit',
↪'agony', 'prison']]>
```

(continues on next page)

(continued from previous page)

```
 ----------------------------------------------------------------------

Debiased vs Biased (absolute values)
0.08 > 0.074
```

Note that the equalization caused the bias of the debiased model to be slightly larger than the original.

4.1.4 Saving the Debiased Model

To save the mitigated model one must access the `KeyedVectors` (the gensim object that contains the embeddings) through `wv` and then use the `save` method to store the method in a file.

```
gender_debiased_model.wv.save("gender_debiased_glove.kv")
```

4.2 Multiclass Hard Debias

Multiclass Hard Debias is a generalized version of Hard Debias that enables multiclass debiasing. Generalized refers to the fact that this method extends Hard Debias in order to support more than two types of social target sets within the definitional set.

For example, for the case of religion bias, it supports a debias using words associated with Christianity, Islam and Judaism.

The usage is very similar to Hard Debias with the difference that the `definitional_sets` can be larger than pairs.

```
from wefe.datasets import fetch_debias_multiclass
from wefe.debias.multiclass_hard_debias import MulticlassHardDebias

multiclass_debias_wordsets = fetch_debias_multiclass()
weat_wordsets = load_weat()
weat = WEAT()

ethnicity_definitional_sets = multiclass_debias_wordsets["ethnicity_definitional_sets"]
ethnicity_equalize_sets = list(
    multiclass_debias_wordsets["ethnicity_analogy_templates"].values()
)

print(f"ethnicity_definitional_sets: \n{ethnicity_definitional_sets}")
print(f"ethnicity_equalize_sets: \n{ethnicity_equalize_sets}")
print("-" * 70, "\n")

mhd = MulticlassHardDebias(verbose=False, criterion_name="ethnicity")
mhd.fit(
    model=model,
    definitional_sets=ethnicity_definitional_sets,
    equalize_sets=ethnicity_equalize_sets,
)

ethnicity_debiased_model = mhd.transform(model, copy=True)
```

```
ethnicity_definitional_sets:
[['black', 'caucasian', 'asian'], ['african', 'caucasian', 'asian'], ['black', 'white',
↪'asian'], ['africa', 'america', 'asia'], ['africa', 'america', 'china'], ['africa',
↪'europe', 'asia']]
ethnicity_equalize_sets:
[['manager', 'executive', 'redneck', 'hillbilly', 'leader', 'farmer'], ['doctor',
↪'engineer', 'laborer', 'teacher'], ['slave', 'musician', 'runner', 'criminal',
↪'homeless']]
--------------------------------------------------------------------------

copy argument is True. Transform will attempt to create a copy of the original model.␣
↪This may fail due to lack of memory.
Model copy created successfully.
```

```
100%|| 13003/13003 [00:00<00:00, 18357.20it/s]
```

4.2.1 Measuring the Decrease of Bias

The following code compares the execution of a query measuring ethnic bias in the original model vs. in the debiased model.

```
ethnicity_query = Query(
    [
        multiclass_debias_wordsets["white_terms"],
        multiclass_debias_wordsets["black_terms"],
    ],
    [
        multiclass_debias_wordsets["white_biased_words"],
        multiclass_debias_wordsets["black_biased_words"],
    ],
    ["european_american_names", "african_american_names"],
    ["white_biased_words", "black_biased_words"],
)

print(ethnicity_query, "\n", "-" * 70, "\n")

biased_results = weat.run_query(
    ethnicity_query, model, normalize=True, preprocessors=[{}, {"lowercase": True}],
)
debiased_results = weat.run_query(
    ethnicity_query,
    ethnicity_debiased_model,
    normalize=True,
    preprocessors=[{}, {"lowercase": True}],
)

print("Debiased vs Biased (absolute values)")
print(
    round(abs(debiased_results_2["weat"]), 3),
    "<",
    round(abs(biased_results_2["weat"]), 3),
```

(continues on next page)

(continued from previous page)

```
)
```

```
<Query: european_american_names and african_american_names wrt white_biased_words and␣
↪black_biased_words
- Target sets: [['america', 'caucasian', 'europe', 'white'], ['africa', 'african', 'black
↪']]
- Attribute sets:[['manager', 'executive', 'redneck', 'hillbilly', 'leader', 'farmer'], [
↪'slave', 'musician', 'runner', 'criminal', 'homeless']]>
 ----------------------------------------------------------------------

Debiased vs Biased (absolute values)
0.08 < 0.074
```

CHAPTER

FIVE

LOADING EMBEDDINGS FROM DIFFERENT SOURCES

WEFE depends on gensim's `KeyedVectors` to operate the word embeddings models. Therefore, any embedding you want to experiment with must be a model loaded through gensim's APIs or any library that extends it.

In technical terms, the minimum requirement for WEFE to operate with a model is that it extends the `BaseKeyedVectors` class.

Next we show several options to load models using different sources.

5.1 Create a example query

In this section we only create an example query (same as the query of user guide) to be used in the following sections.

```
>>> # Load the query
>>> from wefe.query import Query
>>> from wefe.word_embedding_model import WordEmbeddingModel
>>> from wefe.metrics.WEAT import WEAT
>>> from wefe.datasets.datasets import load_weat
>>>
>>> # load the weat word sets
>>> word_sets = load_weat()
>>>
>>> # create the query
>>> query = Query([word_sets['male_terms'], word_sets['female_terms']],
>>>               [word_sets['career'], word_sets['family']],
>>>               ['Male terms', 'Female terms'],
>>>               ['Career', 'Family'])
>>>
>>> # instantiate the metric
>>> weat = WEAT()
```

5.2 Load from Gensim API

Gensim provides an extensive list of pre-trained models that can be used directly. Below we show an example of use.

```
>>> import gensim.downloader as api
>>>
>>> # Load from gensim.downloader some model, for example: glove-twitter-25
>>> glove_25_keyed_vectors = api.load('glove-twitter-25')
>>>
>>> # The resulting object is already a BaseKeyedVectors subclass object.
>>> # so we can wrap directly using .
>>> glove_25_model = WordEmbeddingModel(glove_25_keyed_vectors, 'glove-25')
>>>
>>> # Execute the query
>>> result = weat.run_query(query, glove_25_model)
>>> print(result)
{'query_name': 'Male terms and Female terms wrt Career and Family', 'result': 0.33814692}
```

5.3 Using Gensim Load

As we said before, any model that is loaded with gensim and extends `BaseKeyedVectors` can be used in WEFE to measure bias. In this section we will see how to load a word2vec model and Fasttext.

Note: Gensim is not directly compatible with glove model file format. However, they provide a script that allows you to transform any glove model into a word2vec format.

5.3.1 Loading Word2vec

For example, let us load word2vec from a .bin file The procedure is quite simple: first we download word2vec binary file from its source and then we load it using the `KeyedVectors.load_word2vec_format` function.

```
>>> from gensim.models import KeyedVectors
>>>
>>> w2v_embeddings = KeyedVectors.load_word2vec_format("/path/to/your/embeddings/model",␣
↪binary=True)
>>> word2vec = WordEmbeddingModel(w2v_embeddings, 'word2vec')
>>>
>>> result = weat.run_query(query, word2vec)
>>> result
{'query_name': 'Male terms and Female terms wrt Career and Family',
 'result': 0.7280304}
```

5.3.2 Loading FastText

The same method works for `Fasttext`.

```
>>> from gensim.models import KeyedVectors
>>> fast_embeddings = KeyedVectors.load_word2vec_format('path/to/fast/embeddings.vec')
>>>
>>> fast = WordEmbeddingModel(fast_embeddings, 'fast')
>>> result = weat.run_query(query, fast)
>>>
>>> result
{'query_name': 'Male terms and Female terms wrt Career and Family',
 'result': 0.34870023}
```

While we load FastText here as `KeyedVectors` (i.e., in word2vec format), it can also be used via `FastTextKeyedVectors`.

5.4 Flair

WEFE does not support flair interfaces. However, you can use static embeddings of flair (Classic Word Embeddings) which are based on gensim's `KeyedVectors`, to load embedding models. The following code is an example of this:

```
>>> from flair.embeddings import WordEmbeddings
>>> from wefe.utils import flair_to_gensim
>>>
>>> # could be any of the Classic Word Embeddings model list.
>>> flair_model_name = "glove"
>>>
>>> flair_model = flair_to_gensim(WordEmbeddings(flair_model_name))
>>> wefe_model = WordEmbeddingModel(flair_model, flair_model_name)
>>>
>>> result = weat.run_query(query, wefe_model)
>>> print(result)
{'query_name': 'Male terms and Female terms wrt Career and Family', 'result': 1.0486683}
```

CHAPTER

SIX

WEFE API

This reference details all the utilities as well as the metrics and mitigation methods implemented so far in WEFE.

6.1 WordEmbeddingModel

wefe.word_embedding_model.WordEmbeddingModel(wv)	A wrapper for Word Embedding pre-trained models.

6.1.1 wefe.word_embedding_model.WordEmbeddingModel

class wefe.word_embedding_model.**WordEmbeddingModel**(*wv: KeyedVectors*, *name: Optional[str] = None*, *vocab_prefix: Optional[str] = None*)

A wrapper for Word Embedding pre-trained models.

It can hold gensim's KeyedVectors or gensim's api loaded models. It includes the name of the model and some vocab prefix if needed.

__init__(*wv: KeyedVectors*, *name: Optional[str] = None*, *vocab_prefix: Optional[str] = None*) → None

Initialize the word embedding model.

Parameters

wv
[BaseKeyedVectors.] An instance of word embedding loaded through gensim KeyedVector interface or gensim's api.

name
[str, optional] The name of the model, by default ''.

vocab_prefix
[str, optional.] A prefix that will be concatenated with all word in the model vocab, by default None.

Raises

TypeError
if word_embedding is not a KeyedVectors instance.

TypeError
if model_name is not None and not an instance of str.

TypeError
if vocab_prefix is not None and not an instance of str.

Examples

```
>>> from gensim.test.utils import common_texts
>>> from gensim.models import Word2Vec
>>> from wefe.word_embedding_model import WordEmbeddingModel
```

```
>>> dummy_model = Word2Vec(common_texts, window=5,
...                        min_count=1, workers=1).wv
```

```
>>> model = WordEmbeddingModel(dummy_model, 'Dummy model dim=10',
...                            vocab_prefix='/en/')
>>> print(model.name)
Dummy model dim=10
>>> print(model.vocab_prefix)
/en/
```

Attributes

wv
[BaseKeyedVectors] The model.

vocab
The vocabulary of the model (a dict with the words that have an associated embedding in the model).

model_name
[str] The name of the model.

vocab_prefix
[str] A prefix that will be concatenated with each word of the vocab of the model.

batch_update(*words: Sequence[str]*, *embeddings: Union[Sequence[ndarray], ndarray]*) → None

Update a batch of embeddings.

This method calls *update_embedding* method with each of the word-embedding pairs. All words must be in the vocabulary, otherwise an exception will be thrown. Note that both *words* and *embeddings* must have the same number of elements, otherwise the method will raise an exception.

Parameters

words
[Sequence[str]] A sequence (list, tuple or np.array) that contains the words whose representations will be updated.

embeddings
[Union[Sequence[np.ndarray], np.array],] A sequence (list or tuple) or a np.array of embeddings or an np.array that contains all the new embeddings. The embeddings must have the same size and data type as the model.

Raises

TypeError
if words is not a list

TypeError
if embeddings is not an np.ndarray

Exception
if words collection has not the same size of the embedding array.

normalize() → None

Normalize word embeddings in the model by using the L2 norm.

Use the *init_sims* function of the gensim's *KeyedVectors* class. **Warning**: This operation is inplace. In other words, it replaces the embeddings with their L2 normalized versions.

update(*word: str*, *embedding: ndarray*) → None

Update the value of an embedding of the model.

If the method is executed with a word that is not in the vocabulary, an exception will be raised.

Parameters

word
[str] The word whose embedding will be replaced. This word must be in the model's vocabulary.

embedding
[np.ndarray] An embedding representing the word. It must have the same dimensions and data type as the model embeddings.

Raises

TypeError
if word is not a1 string.

TypeError
if embedding is not an np.array.

ValueError
if word is not in the model's vocabulary.

ValueError
if the embedding is not the same size as the size of the model's embeddings.

ValueError
if the dtype of the embedding values is not the same as the model's embeddings.

6.2 Query

wefe.query.Query(target_sets, attribute_sets)	A container for attribute and target word sets.

6.2.1 wefe.query.Query

class wefe.query.**Query**(*target_sets: List[Any]*, *attribute_sets: List[Any]*, *target_sets_names: Optional[List[str]] = None*, *attribute_sets_names: Optional[List[str]] = None*)

A container for attribute and target word sets.

__init__(*target_sets: List[Any]*, *attribute_sets: List[Any]*, *target_sets_names: Optional[List[str]] = None*, *attribute_sets_names: Optional[List[str]] = None*) → None

Initializes the container. It could include a name for each word set.

Parameters

target_sets
[Union[np.ndarray, list]] Array or list that contains the target word sets.

attribute_sets
[Union[np.ndarray, Iterable]] Array or list that contains the attribute word sets.

target_sets_names
[Union[np.ndarray, Iterable], optional] Array or list that contains the word sets names, by default None

attribute_sets_names
[Union[np.ndarray, Iterable], optional] Array or list that contains the attribute sets names, by default None

Raises

TypeError
if target_sets are not an iterable or np.ndarray instance.

TypeError
if attribute_sets are not an iterable or np.ndarray instance.

Exception
if the length of target_sets is 0.

TypeError
if some element of target_sets is not an array or list.

TypeError
if some element of some target set is not an string.

TypeError
if some element of attribute_sets is not an array or list.

TypeError
if some element of some attribute set is not an string.

Examples

Construct a Query with 2 sets of target words and one set of attribute words.

```
>>> male_terms = ['male', 'man', 'boy']
>>> female_terms = ['female', 'woman', 'girl']
>>> science_terms = ['science','technology','physics']
>>> query = Query([male_terms, female_terms], [science_terms],
...               ['Male terms', 'Female terms'], ['Science terms'])
>>> query.target_sets
[['male', 'man', 'boy'], ['female', 'woman', 'girl']]
>>> query.attribute_sets
[['science', 'technology', 'physics']]
>>> query.query_name
'Male terms and Female terms wrt Science terms'
```

Attributes

target_sets
[list] Array or list with the lists of target words.

attribute_sets
[list] Array or list with the lists of target words.

template
[tuple] A tuple that contains the template: the cardinality of the target and attribute sets respectively.

target_sets_names
[list] Array or list with the names of target sets.

attribute_sets_names
[list] Array or list with the lists of target words.

query_name
[str] A string that contains the auto-generated name of the query.

`dict()` → Dict[str, Any]

Generate a dictionary from the Query data.

This includes the target and attribute sets, as well as their names, the query name generated from them and the query template.

Returns

Dict[str, Any]
The dictionary generated with the query data.

`get_subqueries`(*new_template: tuple*) → list

Generate the subqueries from this query using the given template.

6.3 Metrics

This list contains the metrics implemented in WEFE.

wefe.metrics.WEAT()	Word Embedding Association Test (WEAT).
wefe.metrics.RND()	Relative Norm Distance (RND).
wefe.metrics.RNSB()	Relative Relative Negative Sentiment Bias (RNSB).
wefe.metrics.MAC()	Mean Average Cosine Similarity (MAC).
wefe.metrics.ECT()	Embedding Coherence Test (ECT).
wefe.metrics.RIPA()	Relational Inner Product Association Test (RIPA).

6.3.1 wefe.metrics.WEAT

class wefe.metrics.**WEAT**

Word Embedding Association Test (WEAT).

The following description of the metric is WEFE's adaptation of what was presented in the original WEAT work "Semantics derived automatically from language corpora contain human-like biases" [1].

WEAT receives two sets T_1 and T_2 of target words, and two sets A_1 and A_2 of attribute words and performs a hypothesis test on the following null hypothesis: There is no difference between the two sets of target words in terms of their relative similarity to the similarity with the two sets of attribute words.

In formal terms, let T_1 and T_2 be two sets of target words of equal size, and A_1, A_2 the two sets of attribute words. Let $\cos(\vec{a}, \vec{b})$ denote the cosine of the angle between the vectors $\vec{a}$ and $\vec{b}$. The test statistic is:

$$\text{WEAT}(T_1, T_2, A_1, A_2) = \sum_{x \in T_1} s(x, A_1, A_2) - \sum_{y \in T_2} s(y, A_1, A_2)$$

where

$$s(w, A, B) = \text{mean}_{a \in A} \cos(\vec{w}, \vec{a}) - \text{mean}_{b \in B} \cos(\vec{w}, \vec{b})$$

$s(w, A, B)$ measures the association of w with the attributes, and $\text{WEAT}(T_1, T_2, A_1, A_2)$ measures the differential association of the two sets of target words with the attribute.

This metric also contains a variant: WEAT Effect Size (WEAT-ES). This variant represents a normalized measure that quantifies how far apart the two distributions of association between targets and attributes are. In practical terms, WEAT Effect Size makes the metric not dependent on the number of words used in each set.

$$\text{WEAT-ES}(T_1, T_2, A_1, A_2) = \frac{\text{mean}_{x \in T_1}\, s(x, A_1, A_2) - \text{mean}_{y \in T_2}\, s(y, A_1, A_2)}{\text{std-dev}_{w \in T_1 \cup T_2}\, s(w, A_1, A_2)}$$

The permutation test measures the (un)likelihood of the null hypothesis by computing the probability that a random permutation of the attribute words would produce the observed (or greater) difference in sample mean.

Let $(T_{1_i}, T_{2_i})_i$ denote all the partitions of $T_1 \cup T_2$ into two sets of equal size. The one-sided p-value of the permutation test is:

$$\Pr_i[s(T_{1_i}, T_{2_i}, A_1, A_2) > s(T_1, T_2, A_1, A_2)]$$

References

[1]: Aylin Caliskan, Joanna J Bryson, and Arvind Narayanan.
Semantics derived automatically from language corpora contain human-like biases.
Science, 356(6334):183–186, 2017.

__init__(**args*, ***kwargs*)

run_query(*query:* Query, *model:* WordEmbeddingModel, *return_effect_size: bool = False*, *calculate_p_value: bool = False*, *p_value_test_type: str = 'right-sided'*, *p_value_method: str = 'approximate'*, *p_value_iterations: int = 10000*, *p_value_verbose: bool = False*, *lost_vocabulary_threshold: float = 0.2*, *preprocessors: List[Dict[str, Union[str, bool, Callable]]] = [{}]*, *strategy: str = 'first'*, *normalize: bool = False*, *warn_not_found_words: bool = False*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

Calculate the WEAT metric over the provided parameters.

Parameters

query
[Query] A Query object that contains the target and attribute sets to be tested.

model
[WordEmbeddingModel] A word embedding model.

return_effect_size
[bool, optional] Specifies if the returned score in 'result' field of results dict is by default WEAT effect size metric, by default False

calculate_p_value
[bool, optional] Specifies whether the p-value will be calculated through a permutation test. Warning: This can increase the computing time quite a lot, by default False.

p_value_test_type
[{'left-sided', 'right-sided', 'two-sided}, optional] When calculating the p-value, specify the type of test to be performed. The options are 'left-sided', 'right-sided' and 'two-sided' , by default 'right-sided'

p_value_method
[{'exact', 'approximate'}, optional] When calculating the p-value, specify the method for calculating the p-value. This can be 'exact' and 'approximate'. by default 'approximate'.

p_value_iterations
[int, optional] If the p-value is calculated and the chosen method is 'approximate', it specifies the number of iterations that will be performed , by default 10000.

p_value_verbose
[bool, optional] In case of calculating the p-value, specify if notification messages will be logged during its calculation, by default False.

lost_vocabulary_threshold
[float, optional] Specifies the proportional limit of words that any set of the query is allowed to lose when transforming its words into embeddings. In the case that any set of the query loses proportionally more words than this limit, the result values will be np.nan, by default 0.2

preprocessors
[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor {'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}]` `{}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary.

strategy
[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will

include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize
[bool, optional] True indicates that embeddings will be normalized, by default False

warn_not_found_words
[bool, optional] Specifies if the function will warn (in the logger) the words that were not found in the model's vocabulary, by default False.

Returns

Dict[str, Any]
A dictionary with the query name, the resulting score of the metric, and the scores of WEAT and the effect size of the metric.

Examples

The following example shows how to run a query that measures gender bias using WEAT:

```
>>> from wefe.query import Query
>>> from wefe.utils import load_test_model
>>> from wefe.metrics import WEAT
>>>
>>> # define the query
>>> query = Query(
...     target_sets=[
...         ["female", "woman", "girl", "sister", "she", "her", "hers",
...          "daughter"],
...         ["male", "man", "boy", "brother", "he", "him", "his", "son"],
...     ],
...     attribute_sets=[
...         ["home", "parents", "children", "family", "cousins", "marriage",
...          "wedding", "relatives",
...         ],
...         ["executive", "management", "professional", "corporation", "salary",
...          "office", "business", "career",
...         ],
...     ],
...     target_sets_names=["Female terms", "Male Terms"],
...     attribute_sets_names=["Family", "Career"],
... )
>>>
>>> # load the model (in this case, the test model included in wefe)
>>> model = load_test_model()
>>>
>>> # instance the metric and run the query
>>> WEAT().run_query(query, model)
{'query_name': 'Female terms and Male Terms wrt Family and Career',
'result': 0.4634388245467562,
'weat': 0.4634388245467562,
'effect_size': 0.45076532408312986,
'p_value': nan}
```

If you want to return the effect size as result value, use *return_effect_size* parameter as *True* while running the query.

```
>>> WEAT().run_query(query, model, return_effect_size=True)
{'query_name': 'Female terms and Male Terms wrt Family and Career',
'result': 0.45076532408312986,
'weat': 0.4634388245467562,
'effect_size': 0.45076532408312986,
'p_value': nan}
```

If you want the embeddings to be normalized before calculating the metrics use the *normalize* parameter as *True* before executing the query.

```
>>> WEAT().run_query(query, model, normalize=True)
{'query_name': 'Female terms and Male Terms wrt Family and Career',
'result': 0.4634388248814503,
'weat': 0.4634388248814503,
'effect_size': 0.4507653062895615,
'p_value': nan}
```

Using the *calculate_p_value* parameter as *True* you can indicate WEAT to run the permutation test and return its p-value. The argument *p_value_method='approximate'* indicates that the calculation of the permutation test will be approximate, i.e., not all possible permutations will be generated. Instead, random permutations of the attributes to test will be generated. On the other hand, the argument *p_value_iterations* indicates the number of permutations that will be generated and tested.

```
>>> WEAT().run_query(
...     query,
...     model,
...     calculate_p_value=True,
...     p_value_method="approximate",
...     p_value_iterations=10000,
... )
{
    'query_name': 'Female terms and Male Terms wrt Family and Career',
    'result': 0.46343879750929773,
    'weat': 0.46343879750929773,
    'effect_size': 0.4507652708557911,
    'p_value': 0.1865813418658134
}
```

6.3.2 wefe.metrics.RND

class wefe.metrics.**RND**

Relative Norm Distance (RND).

Originally proposed in "Word embeddings quantify 100 years of gender and ethnic stereotypes" [1], calculates the score by:

1. Average the embeddings of each target set.
2. Then, for each attribute embedding, calculate the distance between the attribute embedding and the average of the target 1 and the distance between the embedding and target 2; and subtracts them.
3. Finally, it computes the average of the differences of the distances and returns it.

The available distances are the the difference of the normalized vectors ('norm') and the cosine distance ('cos').

The more positive (negative) the relative distance from the norm, the more associated are the sets of attributes towards group two (one).

References

[1]: Nikhil Garg, Londa Schiebinger, Dan Jurafsky, and James Zou.
Word embeddings quantify 100 years of gender and ethnic stereotypes.
Proceedings of the National Academy of Sciences, 115(16):E3635–E3644,2018.
[2]: https://github.com/nikhgarg/EmbeddingDynamicStereotypes

__init__(**args*, ***kwargs*)

run_query(*query:* Query, *model:* WordEmbeddingModel, *distance:* *str* = *'norm'*, *lost_vocabulary_threshold:* *float* = *0.2*, *preprocessors:* *List[Dict[str, Union[str, bool, Callable]]]* = *[{}]*, *strategy:* *str* = *'first'*, *normalize:* *bool* = *False*, *warn_not_found_words:* *bool* = *False*, **args:* *Any*, ***kwargs:* *Any*) → Dict[str, Any]

Calculate the RND metric over the provided parameters.

Parameters

query
[Query] A Query object that contains the target and attribute sets to be tested.

model
[WordEmbeddingModel] A word embedding model.

distance
[str, optional] Specifies which type of distance will be calculated. It could be: {norm, cos} , by default 'norm'.

preprocessors
[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor {'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True,`

`"strip_accents": True}]` `{}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary.

strategy
[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize
[bool, optional] True indicates that embeddings will be normalized, by default False

warn_not_found_words
[bool, optional] Specifies if the function will warn (in the logger) the words that were not found in the model's vocabulary, by default False.

Returns

Dict[str, Any]
A dictionary with the query name, the resulting score of the metric, and a dictionary with the distances of each attribute word with respect to the target sets means.

Examples

The following example shows how to run a query that measures gender bias using RND:

```
>>> from wefe.metrics import RND
>>> from wefe.query import Query
>>> from wefe.utils import load_test_model
>>>
>>> # define the query
>>> query = Query(
...     target_sets=[
...         ["female", "woman", "girl", "sister", "she", "her", "hers",
...          "daughter"],
...         ["male", "man", "boy", "brother", "he", "him", "his", "son"],
...     ],
...     attribute_sets=[
...         [
...             "home", "parents", "children", "family", "cousins", "marriage",
...             "wedding", "relatives",
...         ],
...     ],
...     target_sets_names=["Female terms", "Male Terms"],
...     attribute_sets_names=["Family"],
... )
>>>
>>> # load the model (in this case, the test model included in wefe)
>>> model = load_test_model()
>>>
>>> # instance the metric and run the query
>>> RND().run_query(query, model)
{'query_name': 'Female terms and Male Terms wrt Family',
 'result': 0.030381828546524048,
```

(continues on next page)

(continued from previous page)

```
 'rnd': 0.030381828546524048,
 'distance_by_word': {'wedding': -0.1056304,
                      'marriage': -0.10163283,
                      'children': -0.068374634,
                      'parents': 0.00097084045,
                      'relatives': 0.0483346,
                      'family': 0.12408042,
                      'cousins': 0.17195654,
                      'home': 0.1733501}}
>>>
```

If you want the embeddings to be normalized before calculating the metrics, use the normalize parameter as True before executing the query.

```
>>> RND().run_query(query, model, normalize=True)
{'query_name': 'Female terms and Male Terms wrt Family',
 'result': -0.006278775632381439,
 'rnd': -0.006278775632381439,
 'distance_by_word': {'children': -0.05244279,
                      'wedding': -0.04642248,
                      'marriage': -0.04268837,
                      'parents': -0.022358716,
                      'relatives': 0.005497098,
                      'family': 0.023389697,
                      'home': 0.04009247,
                      'cousins': 0.044702888}}
```

If you want to use cosine distance instead of Euclidean norm use the distance parameter as 'cos' before executing the query.

```
>>> RND().run_query(query, model, normalize=True, distance='cos')
{'query_name': 'Female terms and Male Terms wrt Family',
 'result': 0.03643466345965862,
 'rnd': 0.03643466345965862,
 'distance_by_word': {'cousins': -0.035989374,
                      'home': -0.026971221,
                      'family': -0.009296179,
                      'relatives': 0.015690982,
                      'parents': 0.051281124,
                      'children': 0.09255883,
                      'marriage': 0.09959312,
                      'wedding': 0.104610026}}
```

6.3.3 wefe.metrics.RNSB

class wefe.metrics.**RNSB**

Relative Relative Negative Sentiment Bias (RNSB).

The metric was originally proposed in "A transparent framework for evaluating unintended demographic bias in word embeddings" [1].

This metric is based on measuring bias through word sentiment. The main idea is that if there was no bias, all words should be equally negative. Therefore, its procedure is based on calculating how negative the words in the target sets are.

For this purpose, RNSB trains a classifier that assigns a probability to each word of belonging to the negative class (in the original work the classifier is trained using *load_bingliu* of positive and negative words). Then, it generates a probability distribution with the probabilities calculated in the previous step and compares them to the uniform distribution (case where all words have the same probability of being negative) using KL divergence.

When the negative probability distribution is equal to the uniform one (i.e., there is no bias), the KL divergence is 0.

The following description of the metric is WEFE's adaptation of what was presented in the original RNSB work.

RNSB receives as input queries with two attribute sets A_1 and A_2 and two or more target sets, thus has a template (tuple of numbers that defines the allowed target and attribute sets in the query) of the form $s = (N, 2)$ with $N \geq 2$.

Given a query $Q = (\{T_1, T_2, \ldots, T_n\}, \{A_1, A_2\})$ RNSB is calculated under the following steps:

1. First constructs a binary classifier $C_{(A_1,A_2)}(\cdot)$ using set A_1 as training examples for the negative class, and A_2 as training examples for the positive class.
2. After the training process, this classifier gives for every word w a probability $C_{(A_1,A_2)}(w)$ that can be interpreted as the degree of association of w with respect to A_2 (value $1 - C_{(A_1,A_2)}(w)$ is the degree of association with A_1).
3. Then, the metric constructs a probability distribution $P(\cdot)$ over all the words w in $T_1 \cup \cdots \cup T_n$, by computing $C_{(A_1,A_2)}(w)$ and normalizing it to ensure that $\sum_w P(w) = 1$.
4. Finally RNSB is calculated as the distance between $P(\cdot)$ and the uniform distribution $Y(\cdot)$ using the KL-divergence.

The main idea behind RNSB is that the more that $P(\cdot)$ resembles a uniform distribution, the less biased the word embedding model is. Thus, the optimal value is 0.

You can see the full paper replication in Previous Studies Replication section.

References

[1]: Chris Sweeney and Maryam Najafian. A transparent framework for evaluating unintended demographic bias in word embeddings.

In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 1662–1667, 2019.

[2]: https://github.com/ChristopherSweeney/AIFairness/blob/master/python_notebooks/Measuring_and_Mitigating_Word_Embedding_Bias.ipynb

__init__(**args*, ***kwargs*)

run_query(*query: ~wefe.query.Query, model: ~wefe.word_embedding_model.WordEmbeddingModel, estimator: ~sklearn.base.BaseEstimator = <class 'sklearn.linear_model._logistic.LogisticRegression'>, estimator_params: ~typing.Dict[str, ~typing.Any] = {'max_iter': 10000, 'solver': 'liblinear'}, n_iterations: int = 1, random_state: ~typing.Optional[int] = None, holdout: bool = True, print_model_evaluation: bool = False, lost_vocabulary_threshold: float = 0.2, preprocessors: ~typing.List[~typing.Dict[str, ~typing.Union[str, bool, ~typing.Callable]]] = [{}], strategy: str = 'first', normalize: bool = False, warn_not_found_words: bool = False, *args: ~typing.Any, **kwargs: ~typing.Any*) → Dict[str, Any]

Calculate the RNSB metric over the provided parameters.

Note if you want to use with Bing Liu dataset, you have to pass the positive and negative words in the first and second place of attribute set array respectively. Scores on this metric vary with each run due to different instances of classifier training. For this reason, the robustness of these scores can be improved by repeating the test several times and returning the average of the scores obtained. This can be indicated in the n_iterations parameter.

Parameters

query
[Query] A Query object that contains the target and attribute word sets to be tested.

model
[WordEmbeddingModel] A word embedding model.

estimator
[BaseEstimator] A scikit-learn classifier class that implements predict_proba function, by default None,

estimator_params
[dict] Parameters that will use the classifier, by default { 'solver': 'liblinear', 'max_iter': 10000, }

n_iterations
[int, optional] When provided, it tells the metric to run the specified number of times and then average its results. This functionality is indicated to strengthen the results obtained. Note that you cannot specify random_state next to n_iterations as this would always produce the same results, by default 1.

random_state
[Union[int, None], optional] Seed that allows making the execution of the query reproducible. Warning: if a random_state other than None is provided along with n_iterations, each iteration will split the dataset and train a classifier associated to the same seed, so the results of each iteration will always be the same, by default None.

holdout: bool, optional
True indicates that a holdout (split attributes in train/test sets) will be executed before running the model training. This option allows for evaluating the performance of the classifier (can be printed using print_model_evaluation=True) at the cost of training the classifier with fewer examples. False disables this functionality. Note that holdout divides into 80% train and 20% test, performs a shuffle and tries to maintain the original ratio of the classes via stratify param, by default True.

print_model_evaluation
[bool, optional] Indicates whether the classifier evaluation is printed after the training process is completed, by default False

preprocessors
[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor {'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}] {}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary.

strategy
[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize
[bool, optional] True indicates that embeddings will be normalized, by default False

warn_not_found_words
[bool, optional] Specifies if the function will warn (in the logger) the words that were not found in the model's vocabulary, by default False.

Returns

Dict[str, Any]
A dictionary with the query name, the calculated kl-divergence, the negative probabilities for all tested target words and the normalized distribution of probabilities.

Examples

The following example shows how to run a query that measures gender bias using RNSB. Note that by default the RNSB score is returned plus the negative class probabilities for each word and its distribution (the above probabilities normalized to 1).

```
>>> from wefe.query import Query
>>> from wefe.utils import load_test_model
>>> from wefe.metrics import RNSB
>>>
>>> # define the query
>>> query = Query(
```

(continues on next page)

(continued from previous page)

```
...     target_sets=[
...         ["female", "woman", "girl", "sister", "she", "her", "hers",
...          "daughter"],
...         ["male", "man", "boy", "brother", "he", "him", "his", "son"],
...     ],
...     attribute_sets=[
...         ["home", "parents", "children", "family", "cousins", "marriage",
...          "wedding", "relatives",],
...         ["executive", "management", "professional", "corporation", "salary",
...          "office", "business", "career", ],
...     ],
...     target_sets_names=["Female terms", "Male Terms"],
...     attribute_sets_names=["Family", "Careers"],
... )
>>>
>>> # load the model (in this case, the test model included in wefe)
>>> model = load_test_model()
>>>
>>> # instance the metric and run the query
>>> RNSB().run_query(query, model)
{
    "query_name": "Female terms and Male Terms wrt Family and Careers",
    "result": 0.10769060995617141,
    "rnsb": 0.10769060995617141,
    "negative_sentiment_probabilities": {
        "female": 0.5742192708509877,
        "woman": 0.32330898567978306,
        "girl": 0.17573260129841273,
        "sister": 0.15229835340332343,
        "she": 0.3761328719677399,
        "her": 0.35739995104539557,
        "hers": 0.2911542159275662,
        "daughter": 0.11714195753410628,
        "male": 0.4550779245077232,
        "man": 0.39826729589696475,
        "boy": 0.17445392462199483,
        "brother": 0.16517694979156405,
        "he": 0.5044892468050808,
        "him": 0.45426103796811057,
        "his": 0.5013980699813614,
        "son": 0.1509265229834842,
    },
    "negative_sentiment_distribution": {
        "female": 0.11103664779476699,
        "woman": 0.0625181838962096,
        "girl": 0.033981372529543176,
        "sister": 0.02944989742595416,
        "she": 0.07273272658861042,
        "her": 0.06911034599602082,
        "hers": 0.0563004234950174,
        "daughter": 0.022651713275710483,
        "male": 0.08799831316676666,
```

(continues on next page)

(continued from previous page)

```
        "man": 0.07701285503210042,
        "boy": 0.033734115115920706,
        "brother": 0.031940228635376634,
        "he": 0.09755296914839981,
        "him": 0.08784035200525282,
        "his": 0.09695522899987082,
        "son": 0.029184626894479145,
    },
}
```

If you want to perform a holdout to evaluate (defualt option) the model and print the evaluation, use the params `holdout=True` and `print_model_evaluation=True`

```
>>> RNSB().run_query(
...     query,
...     model,
...     holdout=True,
...     print_model_evaluation=True)
"Classification Report:"
"              precision    recall  f1-score   support"
"                                                     "
"        -1.0       1.00      1.00      1.00         2"
"         1.0       1.00      1.00      1.00         2"
"                                                      "
"    accuracy                           1.00         4"
"   macro avg       1.00      1.00      1.00         4"
"weighted avg       1.00      1.00      1.00         4"
{
    "query_name": "Female terms and Male Terms wrt Family and Careers",
    "result": 0.09400726375514418,
    "rnsb": 0.09400726375514418,
    "negative_sentiment_probabilities": {
        "female": 0.5583010801302075,
        "woman": 0.3159147912504866,
        "girl": 0.20753840501109977,
        "sister": 0.16020059726421976,
        "she": 0.4266765171984158,
        "her": 0.4066467259229203,
        "hers": 0.32435655424005905,
        "daughter": 0.13318012193912765,
        "male": 0.44129601598998147,
        "man": 0.42681869843678866,
        "boy": 0.21830517614567535,
        "brother": 0.2037443178553553,
        "he": 0.5655603842644314,
        "him": 0.512466010254818,
        "his": 0.5689713390373838,
        "son": 0.18364286185769785,
    },
    "negative_sentiment_distribution": {
        "female": 0.09875108690481095,
        "woman": 0.05587832464521873,
```

(continues on next page)

(continued from previous page)

```
        "girl": 0.0367089439708001,
        "sister": 0.02833593497428311,
        "she": 0.07546961904547295,
        "her": 0.07192679290859644,
        "hers": 0.05737148541506475,
        "daughter": 0.023556611770367462,
        "male": 0.0780554843555203,
        "man": 0.07549476775523993,
        "boy": 0.038613347150078775,
        "brother": 0.03603785404499525,
        "he": 0.1000350969111323,
        "him": 0.09064387893111858,
        "his": 0.10063841921015712,
        "son": 0.032482352007143285,
    },
}
```

If you want to disable the holdout, use the param `holdout=False`.

```
>>> # instance the metric and run the query
>>> RNSB().run_query(
...     query,
...     model,
...     holdout=False,
...     print_model_evaluation=True)
"Holdout is disabled. No evaluation was performed."
{
    "query_name": "Female terms and Male Terms wrt Family and Careers",
    "result": 0.12921977967420623,
    "rnsb": 0.12921977967420623,
    "negative_sentiment_probabilities": {
        "female": 0.5815344717945401,
        "woman": 0.28951392462851366,
        "girl": 0.16744925298532254,
        "sister": 0.1365690846140981,
        "she": 0.40677635339222296,
        "her": 0.3861243765483825,
        "hers": 0.2794312043966708,
        "daughter": 0.1035870893754135,
        "male": 0.45492464330345805,
        "man": 0.4143325974603802,
        "boy": 0.18150440132198242,
        "brother": 0.1607078872193466,
        "he": 0.5656269380025241,
        "him": 0.5025663479575841,
        "his": 0.5657745694122852,
        "son": 0.14803033326417403,
    },
    "negative_sentiment_distribution": {
        "female": 0.10881083995606983,
        "woman": 0.05417091306830872,
        "girl": 0.03133140811261611,
```

(continues on next page)

(continued from previous page)

```
        "sister": 0.025553423794525788,
        "she": 0.0761118709786697,
        "her": 0.07224768225706711,
        "hers": 0.05228433658715302,
        "daughter": 0.019382166922557734,
        "male": 0.08512089128923325,
        "man": 0.07752571883094242,
        "boy": 0.03396126510372403,
        "brother": 0.030070032034284336,
        "he": 0.10583438336150637,
        "him": 0.09403512449772648,
        "his": 0.1058620066555313,
        "son": 0.027697936550083888,
    },
}
```

Since each run of RNSB may give a different result due to the random `train_test_split` and random initializations, RNSB can be requested to run many times and returns the average of all runs through the parameter `n_iterations`. This makes it potentially more stable and robust to outlier runs.

```
>>> RNSB().run_query(query, model, n_iterations=1000)
{
    "query_name": "Female terms and Male Terms wrt Family and Careers",
    "result": 0.09649701346914233,
    "rnsb": 0.09649701346914233,
    "negative_sentiment_probabilities": {
        "female": 0.5618993210534083,
        "woman": 0.31188456697468364,
        "girl": 0.1968846981458747,
        "sister": 0.1666990161087616,
        "she": 0.4120315698794307,
        "her": 0.3956786125532543,
        "hers": 0.3031550094192968,
        "daughter": 0.13259627603249385,
        "male": 0.45579890258209677,
        "man": 0.4210218238530363,
        "boy": 0.2104231329680286,
        "brother": 0.18879207133574177,
        "he": 0.5473770682025214,
        "him": 0.4924664455586234,
        "his": 0.5479209229372095,
        "son": 0.1770764765027373,
    },
    "negative_sentiment_distribution": {
        "female": 0.10176190651838826,
        "woman": 0.056483371593163176,
        "girl": 0.035656498409823205,
        "sister": 0.030189767202716926,
        "she": 0.07462033949087124,
        "her": 0.07165876247453706,
        "hers": 0.05490241858857015,
        "daughter": 0.024013643264435475,
```

(continues on next page)

(continued from previous page)

```
        "male": 0.08254675451251275,
        "man": 0.07624850551663455,
        "boy": 0.03810835568595322,
        "brother": 0.034190895761652934,
        "he": 0.09913187640146649,
        "him": 0.08918737310881396,
        "his": 0.09923037037111067,
        "son": 0.03206916109934998,
    },
}
```

If you want the embeddings to be normalized before calculating the metrics use the `normalize=True` before executing the query.

```
>>> RNSB().run_query(query, model, normalize=True)
{
    "query_name": "Female terms and Male Terms wrt Family and Careers",
    "result": 0.00957187793390364,
    "rnsb": 0.00957187793390364,
    "negative_sentiment_probabilities": {
        "female": 0.5078372178028085,
        "woman": 0.4334357574118245,
        "girl": 0.3764103216054252,
        "sister": 0.35256834229924383,
        "she": 0.4454357087596428,
        "her": 0.4390986149718311,
        "hers": 0.41329577968574494,
        "daughter": 0.33427930165282493,
        "male": 0.470250420503012,
        "man": 0.4577545228416623,
        "boy": 0.3698438702135818,
        "brother": 0.35380575403374315,
        "he": 0.49962008627445753,
        "him": 0.47052126448152776,
        "his": 0.49505591114138436,
        "son": 0.34192683607526553,
    },
    "negative_sentiment_distribution": {
        "female": 0.07511118533317328,
        "woman": 0.06410690741777263,
        "girl": 0.05567261405091151,
        "sister": 0.05214628855999085,
        "she": 0.06588174891831232,
        "her": 0.0649444670309599,
        "hers": 0.061128122983393235,
        "daughter": 0.04944126523085684,
        "male": 0.0695519454840733,
        "man": 0.06770375151119586,
        "boy": 0.054701409243182085,
        "brother": 0.05232930677698083,
        "he": 0.07389583823474007,
        "him": 0.06959200440758956,
```

(continues on next page)

(continued from previous page)

```
        "his": 0.07322077821098569,
        "son": 0.050572366605882095,
    },
}
```

RNSB accepts more than 2 sets of target words. This example shows how to measure words representing different nationalities with respect to positive and negative words.

Note this is one of the tests that was proposed in the RNSB paper. You can see the full paper replication in Previous Studies Replication.

```
>>> import gensim.downloader as api
>>>
>>> from wefe.word_embedding_model import WordEmbeddingModel
>>> from wefe.query import Query
>>> from wefe.datasets import load_bingliu
>>> from wefe.metrics import RNSB
>>>
>>> # Load the model
>>> model = WordEmbeddingModel(
...         api.load('glove-wiki-gigaword-300'), 'Glove wiki'
...     )
>>>
>>> RNSB_words = [
...     ["swedish"],
...     ["irish"],
...     ["mexican"],
...     ["chinese"],
...     ["filipino"],
...     ["german"],
...     ["english"],
...     ["french"],
...     ["norwegian"],
...     ["american"],
...     ["indian"],
...     ["dutch"],
...     ["russian"],
...     ["scottish"],
...     ["italian"],
... ]
>>>
>>> bing_liu = load_bingliu()
>>>
>>> # Create the query
>>> query = Query(
...     RNSB_words,
...     [bing_liu["positive_words"], bing_liu["negative_words"]],
...     attribute_sets_names=["Positive words", "Negative words"],
... )
>>>
>>> results = RNSB().run_query(
...     query,
...     model,
```

(continues on next page)

(continued from previous page)

```
...     preprocessors=[{"lowercase": True}],
...     holdout=True,
...     print_model_evaluation=True,
...     n_iterations=500,
... )
>>> results
{
    "query_name": (
        "Target set 0, Target set 1, Target set 2, Target set 3, "
        "Target set 4, Target set 5, Target set 6, Target set 7, "
        "Target set 8, Target set 9, Target set 10, Target set 11, "
        "Target set 12, Target set 13 and Target set 14 "
        "wrt Positive words and Negative words"
    ),
    "result": 0.6313118439654091,
    "rnsb": 0.6313118439654091,
    "negative_sentiment_probabilities": {
        "swedish": 0.03865446713798508,
        "irish": 0.12266387930214015,
        "mexican": 0.5038405165657709,
        "chinese": 0.01913990969357335,
        "filipino": 0.08074140612507152,
        "german": 0.0498762435972975,
        "english": 0.058042779461913364,
        "french": 0.08030917713203162,
        "norwegian": 0.12177903128690087,
        "american": 0.22908203952254658,
        "indian": 0.7836948288757486,
        "dutch": 0.22748838866881654,
        "russian": 0.4877408793080844,
        "scottish": 0.027805085889223837,
        "italian": 0.007885923500742055,
    },
    "negative_sentiment_distribution": {
        "swedish": 0.01361674725376778,
        "irish": 0.04321060837966024,
        "mexican": 0.17748709213331876,
        "chinese": 0.006742385345191273,
        "filipino": 0.02844264587050847,
        "german": 0.017569824481278706,
        "english": 0.020446636995867174,
        "french": 0.028290385255119174,
        "norwegian": 0.04289890438595362,
        "american": 0.08069836330742804,
        "indian": 0.27607092269031136,
        "dutch": 0.08013697047258364,
        "russian": 0.17181569869170976,
        "scottish": 0.009794853090881381,
        "italian": 0.0027779616464207076,
    },
}
```

6.3.4 wefe.metrics.MAC

class wefe.metrics.**MAC**

Mean Average Cosine Similarity (MAC).

The metric, presented in the paper "Black is to Criminal as Caucasian is to Police: Detecting and Removing Multiclass Bias in Word Embeddings" [1], calculate the score as follows:

```
Embed all target and attribute words.
For each target set:
For each word embedding in the target set:
    For each attribute set:
        Calculate the cosine similarity of the target embedding and
        each attribute embedding of the set.
        Calculate the mean of the cosines similarities and store it in a
        array.
Average all the mean cosine similarities and return the calculated score.
```

The closer the value is to 1, the less biased the query will be.

References

[1]: Thomas Manzini, Lim Yao Chong,Alan W Black, and Yulia Tsvetkov.

Black is to Criminal as Caucasian is to Police: Detecting and Removing Multiclass Bias in Word Embeddings.

In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 615–621,

Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[2]: https://github.com/TManzini/DebiasMulticlassWordEmbedding/blob/master/Debiasing/evalBias.py

__init__(**args*, ***kwargs*)

run_query(*query:* Query, *model:* WordEmbeddingModel, *lost_vocabulary_threshold:* *float* = *0.2*, *preprocessors:* *List[Dict[str, Union[str, bool, Callable]]]* = *[{}]*, *strategy:* *str* = *'first'*, *normalize:* *bool* = *False*, *warn_not_found_words:* *bool* = *False*, **args:* *Any*, ***kwargs:* *Any*) → Dict[str, Any]

Calculate the MAC metric over the provided parameters.

Parameters

query
[Query] A Query object that contains the target and attribute word sets for be tested.

model
[WordEmbeddingModel] A word embedding model.

preprocessors
[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor {'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}]` `{}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary.

strategy
[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize
[bool, optional] True indicates that embeddings will be normalized, by default False

warn_not_found_words
[bool, optional] Specifies if the function will warn (in the logger) the words that were not found in the model's vocabulary, by default False.

Returns

Dict[str, Any]
A dictionary with the query name, the resulting score of the metric, and a dictionary with the distances of each attribute word with respect to the target sets means.

Examples

The following example shows how to run a query that measures gender bias using MAC. Note that the results return both the result of the metric and the cosine distance of each target embedding with respect to the average embedding of each attribute set.

```
>>> from wefe.metrics import MAC
>>> from wefe.query import Query
>>> from wefe.utils import load_test_model
>>>
>>> query = Query(
...     target_sets=[
...         ["female", "woman", "girl", "sister", "she", "her", "hers",
...          "daughter"],
...         ["male", "man", "boy", "brother", "he", "him", "his", "son"],
...     ],
...     attribute_sets=[
```

(continues on next page)

(continued from previous page)

```
...         ["home", "parents", "children", "family", "cousins", "marriage",
...          "wedding", "relatives",
...         ],
...         ["executive", "management", "professional", "corporation", "salary",
...          "office", "business", "career",
...         ],
...     ],
...     target_sets_names=["Female terms", "Male Terms"],
...     attribute_sets_names=["Family", "Careers"],
... )
>>>
>>> # load the model (in this case, the test model included in wefe)
>>> model = load_test_model()
>>>
>>> # instance the metric and run the query
>>> MAC().run_query(query, model)
{
    "query_name": "Female terms and Male Terms wrt Family and Careers",
    "result": 0.8416415235615204,
    "mac": 0.8416415235615204,
    "targets_eval": {
        "Female terms": {
            "female": {"Family": 0.9185737599618733, "Careers": 0.
↪916069650076679},
            "woman": {"Family": 0.752434104681015, "Careers": 0.
↪9377805145923048},
            "girl": {"Family": 0.707457959651947, "Careers": 0.9867974997032434}
↪,
            "sister": {"Family": 0.5973392464220524, "Careers": 0.
↪9482253392925486},
            "she": {"Family": 0.7872791914269328, "Careers": 0.9161583095556125}
↪,
            "her": {"Family": 0.7883057091385126, "Careers": 0.9237247597193345}
↪,
            "hers": {"Family": 0.7385367527604103, "Careers": 0.
↪9480051446007565},
            "daughter": {"Family": 0.5472579970955849, "Careers": 0.
↪9277344475267455},
        },
        "Male Terms": {
            "male": {"Family": 0.8735092766582966, "Careers": 0.
↪9468009045813233},
            "man": {"Family": 0.8249392118304968, "Careers": 0.9350165261421353}
↪,
            "boy": {"Family": 0.7106057899072766, "Careers": 0.9879048476286698}
↪,
            "brother": {"Family": 0.6280269809067249, "Careers": 0.
↪9477180293761194},
            "he": {"Family": 0.8693044614046812, "Careers": 0.8771287016716087},
            "him": {"Family": 0.8230192996561527, "Careers": 0.888683641096577},
            "his": {"Family": 0.8876195731572807, "Careers": 0.8920885202242061}
↪,
```

(continues on next page)

(continued from previous page)

```
            "son": {"Family": 0.5764635019004345, "Careers": 0.9220191016211174}
↪,
        },
    },
}
```

MAC was originally designed to accept more than two target sets. The following example shows how to run a Query that measures ethnic bias (black, white and Asian) with respect to biased concepts commonly associated with these groups:

```
>>> from wefe.word_embedding_model import WordEmbeddingModel
>>> from wefe.query import Query
>>> from wefe.metrics import MAC
>>>
>>> ethnicity_query = Query(
...     target_sets=[
...         ["black", "africa"],
...         ["caucasian", "america"],
...         ["asian", "asia"]],
...     attribute_sets=[
...         ["slave", "musician", "runner", "criminal", "homeless"],
...         ["manager", "executive", "redneck", "hillbilly", "leader", "farmer
↪"],
...         ["doctor", "engineer", "laborer", "teacher"],
...     ],
...     target_sets_names=["Black words", "White words", "Asian words"],
...     attribute_sets_names=[
...         "Black biased words",
...         "White biased words",
...         "Asian biased words",
...     ],
... )
>>>
>>> # load word2vec from gensim api.
>>> model = WordEmbeddingModel(api.load("word2vec-google-news-300"), "word2vec")
>>>
>>> # instance the metric and run the query
>>> MAC().run_query(ethnicity_query, model)
{
    "query_name": (
        "Black words, White words and Asian words wrt Black "
        "biased words, White biased words and Asian biased "
        "words"
    ),
    "result": 0.9462675075454171,
    "mac": 0.9462675075454171,
    "targets_eval": {
        "Black words": {
            "black": {
                "Black biased words": 0.887664645537734,
                "White biased words": 0.8847345071844757,
                "Asian biased words": 0.9767352174967527,
```

(continues on next page)

(continued from previous page)

```
            },
            "africa": {
                "Black biased words": 0.9635645077563822,
                "White biased words": 0.9766113509734472,
                "Asian biased words": 1.0130788520909846,
            },
        },
        "White words": {
            "caucasian": {
                "Black biased words": 0.8768320322036743,
                "White biased words": 0.8743524154027303,
                "Asian biased words": 0.9000119741540402,
            },
            "america": {
                "Black biased words": 0.9680319786071777,
                "White biased words": 0.9214469141637286,
                "Asian biased words": 0.994440613896586,
            },
        },
        "Asian words": {
            "asian": {
                "Black biased words": 0.965478494623676,
                "White biased words": 0.9149698968976736,
                "Asian biased words": 0.9534151484258473,
            },
            "asia": {
                "Black biased words": 0.972597711533308,
                "White biased words": 0.9664777771686204,
                "Asian biased words": 1.0223710977006704,
            },
        },
    },
}
```

We can also test words that represent religious groups and beliefs:

```
>>> from wefe.word_embedding_model import WordEmbeddingModel
>>> from wefe.query import Query
>>> from wefe.metrics import MAC
>>> from wefe.utils import load_test_model
>>> import gensim.downloader as api
>>>
>>> # load word2vec from gensim api.
>>> model = WordEmbeddingModel(api.load("word2vec-google-news-300"), "word2vec")
>>>
>>> # create the query
>>> religion_query = Query(
...     target_sets=[
...         ["judaism", "jew", "synagogue", "torah", "rabbi"],
...         ["christianity", "christian", "church", "bible", "priest"],
...         ["islam", "muslim", "mosque", "quran", "imam"],
...     ],
```

(continues on next page)

(continued from previous page)

```
...     attribute_sets=[
...         ["greedy", "cheap", "hairy", "liberal"],
...         ["judgmental", "conservative", "familial"],
...         ["violent", "terrorist", "dirty", "uneducated"],
...     ],
...     target_sets_names=["judaism", "christianity", "islam"],
...     attribute_sets_names=[
...         "jew biased words",
...         "christian biased words",
...         "musilm biased words",
...     ],
... )
>>>
>>> # instance the metric and run the query
>>> MAC().run_query(religion_query, model, warn_not_found_words=True)
{
    "query_name": (
        "judaism, christianity and islam wrt jew biased words,"
        " christian biased words and musilm biased words"
    ),
    "result": 0.8589896201628209,
    "mac": 0.8589896201628209,
    "targets_eval": {
        "judaism": {
            "judaism": {
                "jew biased words": 0.8744675349444151,
                "christian biased words": 0.815421904126803,
                "musilm biased words": 0.8894469570368528,
            },
            "jew": {
                "jew biased words": 0.7810277417302132,
                "christian biased words": 0.8705306425690651,
                "musilm biased words": 0.8410659478977323,
            },
            "synagogue": {
                "jew biased words": 0.9586692564189434,
                "christian biased words": 0.8717945317427317,
                "musilm biased words": 0.9161230166791938,
            },
            "torah": {
                "jew biased words": 0.9311909799580462,
                "christian biased words": 0.8741760378082594,
                "musilm biased words": 0.9664547641441459,
            },
            "rabbi": {
                "jew biased words": 0.9022729225689545,
                "christian biased words": 0.8595656901597977,
                "musilm biased words": 0.9270578834693879,
            },
        },
        "christianity": {
            "christianity": {
```

(continues on next page)

(continued from previous page)


```
                "jew biased words": 0.8192066270858049,
                "christian biased words": 0.783344641327858,
                "musilm biased words": 0.808249220252037,
            },
            "christian": {
                "jew biased words": 0.8092729989439249,
                "christian biased words": 0.7565138414502144,
                "musilm biased words": 0.7822588048875332,
            },
            "church": {
                "jew biased words": 0.934008174444898,
                "christian biased words": 0.8065384129683176,
                "musilm biased words": 0.8915035352110863,
            },
            "bible": {
                "jew biased words": 0.8507496938109398,
                "christian biased words": 0.8642359959194437,
                "musilm biased words": 0.8490688409656286,
            },
            "priest": {
                "jew biased words": 0.9257305036298931,
                "christian biased words": 0.861459826429685,
                "musilm biased words": 0.8620996568351984,
            },
        },
        "islam": {
            "islam": {
                "jew biased words": 0.8377434946596622,
                "christian biased words": 0.8127042253812155,
                "musilm biased words": 0.7525370791554451,
            },
            "muslim": {
                "jew biased words": 0.8212915528565645,
                "christian biased words": 0.8246404901146889,
                "musilm biased words": 0.7299829311668873,
            },
            "mosque": {
                "jew biased words": 0.9514421001076698,
                "christian biased words": 0.8898302918920914,
                "musilm biased words": 0.8566081328317523,
            },
            "quran": {
                "jew biased words": 0.913289612159133,
                "christian biased words": 0.8723569065332413,
                "musilm biased words": 0.8311764020472765,
            },
            "imam": {
                "jew biased words": 0.9434488633705769,
                "christian biased words": 0.8907990877827009,
                "musilm biased words": 0.8431751518510282,
            },
        },
```


(continues on next page)

(continued from previous page)

```
    },
}
```

6.3.5 wefe.metrics.ECT

class wefe.metrics.**ECT**

Embedding Coherence Test (ECT).

The metric was originally proposed in [1] and implemented in [2]. It calculates the average target group vectors, measures the cosine similarity of each to a list of attribute words and calculates the correlation of the resulting similarity lists.

Values closer to 1 are better as they represent less bias.

The general steps of the test, as defined in [1], are as follows:

1. Embed all given target and attribute words with the given embedding model.
2. Calculate mean vectors for the two sets of target word vectors.
3. Measure the cosine similarity of the mean target vectors to all of the given attribute words.
4. Calculate the Spearman r correlation between the resulting two lists of similarities.
5. Return the correlation value as score of the metric (in the range of -1 to 1); higher is better.

Values closer to 1 are better as they represent less bias.

References

[1]: Dev, S., & Phillips, J. (2019, April). Attenuating Bias in Word vectors.
[2]: https://github.com/sunipa/Attenuating-Bias-in-Word-Vec

__init__(**args*, ***kwargs*)

run_query(*query:* Query, *model:* WordEmbeddingModel, *lost_vocabulary_threshold:* *float* = *0.2*, *preprocessors:* *List[Dict[str, Union[str, bool, Callable]]]* = *[{}]*, *strategy:* *str* = *'first'*, *normalize:* *bool* = *False*, *warn_not_found_words:* *bool* = *False*, **args:* *Any*, ***kwargs:* *Any*) → Dict[str, Any]

Run ECT with the given query with the given parameters.

Parameters

query
[Query] A Query object that contains the target and attribute word sets to be tested.

model
[WordEmbeddingModel] A word embedding model.

preprocessors
[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor`

`{'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}]` `{}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary.

strategy
[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize
[bool, optional] True indicates that embeddings will be normalized, by default False

warn_not_found_words
[bool, optional] Specifies if the function will warn (in the logger) the words that were not found in the model's vocabulary, by default False.

Returns

Dict[str, Any]
A dictionary with the query name and the result of the query.

Examples

```
>>> from wefe.metrics import ECT
>>> from wefe.query import Query
>>> from wefe.utils import load_test_model
>>>
>>> # define the query
>>> query = Query(
...     target_sets=[
...         ["female", "woman", "girl", "sister", "she", "her", "hers",
...          "daughter"],
...         ["male", "man", "boy", "brother", "he", "him", "his", "son"],
...     ],
...     attribute_sets=[
```

(continues on next page)

(continued from previous page)

```
...         [
...             "home", "parents", "children", "family", "cousins", "marriage",
...             "wedding", "relatives",
...         ],
...     ],
...     target_sets_names=["Female terms", "Male Terms"],
...     attribute_sets_names=["Family"],
... )
>>> # load the model (in this case, the test model included in wefe)
>>> model = load_test_model()
>>>
>>> # instance the metric and run the query
>>> ECT().run_query(query, model)
{'query_name': 'Female terms and Male Terms wrt Family',
'result': 0.6190476190476191,
'ect': 0.6190476190476191}
>>> # if you want the embeddings to be normalized before calculating the metrics
>>> # use the normalize parameter as True before executing the query.
>>> ECT().run_query(query, model, normalize=True)
{'query_name': 'Female terms and Male Terms wrt Family',
'result': 0.7619047619047621,
'ect': 0.7619047619047621}
```

6.3.6 wefe.metrics.RIPA

class wefe.metrics.**RIPA**

Relational Inner Product Association Test (RIPA).

This metric was originally proposed in [1][2].

RIPA is most interpretable with a single pair of target words, although this function returns the values for every attribute averaged across all base pairs.

NOTE: As the variance tends to be high depending on the base pair chosen, it is recommended that only a single pair of target words is used as input to the function.

This metric follows the following steps:

1. The input is the word vectors for a pair of target word sets, and an attribute set. Example: Target Set A (Masculine), Target Set B (Feminine), Attribute Set (Career).
2. Calculate the difference between the word vector of a pair of target set words.
3. Calculate the dot product between this difference and the attribute word vector.
4. Return the average RIPA score across all attribute words, and the average RIPA score for each target pair for an attribute set.

Note: RIPA is most interpretable with a single pair of target words, although this function returns the values for every attribute averaged across all base pairs.

Note: Note 2: As the variance tends to be high depending on the base pair chosen, it is recommended that only

a single pair of target words is used as input to the function.

References

[1]: Ethayarajh, K., & Duvenaud, D., & Hirst, G. (2019, July). Understanding Undesirable Word Embedding Associations.

[2]: https://kawine.github.io/assets/acl2019_bias_slides.pdf

[3]: https://kawine.github.io/blog/nlp/2019/09/23/bias.html

__init__(**args*, ***kwargs*)

run_query(*query:* Query, *model:* WordEmbeddingModel, *lost_vocabulary_threshold: float = 0.2*, *preprocessors: List[Dict[str, Union[str, bool, Callable]]] = [{}]*, *strategy: str = 'first'*, *normalize: bool = False*, *warn_not_found_words: bool = False*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

Calculate the Example Metric metric over the provided parameters.

Parameters

query
[Query] A Query object that contains the target and attribute sets to be tested.

model
[WordEmbeddingModel] A word embedding model.

lost_vocabulary_threshold
[float, optional] Specifies the proportional limit of words that any set of the query is allowed to lose when transforming its words into embeddings. In the case that any set of the query loses proportionally more words than this limit, the result values will be np.nan, by default 0.2

preprocessors
[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor {'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}]` `{}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary.

strategy
[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize
[bool, optional] True indicates that embeddings will be normalized, by default False

warn_not_found_words
[bool, optional] Specifies if the function will warn (in the logger) the words that were not found in the model's vocabulary, by default False.

Returns

Dict[str, Any]
A dictionary with the query name, the resulting score of the metric, and other scores.

Examples

```
>>> from wefe.metrics import RIPA
>>> from wefe.query import Query
>>> from wefe.utils import load_test_model
>>>
>>> # define the query
... query = Query(
...     target_sets=[
...         ["female", "woman", "girl", "sister", "she", "her", "hers",
...         "daughter"],
...         ["male", "man", "boy", "brother", "he", "him", "his", "son"],
...     ],
...     attribute_sets=[
...         [
...             "home", "parents", "children", "family", "cousins", "marriage",
...             "wedding", "relatives",
...         ],
...     ],
...     target_sets_names=["Female terms", "Male Terms"],
...     attribute_sets_names=["Family"],
... )
>>>
>>> # load the model (in this case, the test model included in wefe)
>>> model = load_test_model()
>>>
>>> # instance the metric and run the query
>>> RIPA().run_query(query, model)
{
    'query_name': 'Female terms and Male Terms wrt Family',
    'result': 0.18600442,
```

(continues on next page)

(continued from previous page)

```
    'ripa': 0.18600442,
    'word_values': {
        'home': {'mean': 0.008022693, 'std': 0.07485135},
        'parents': {'mean': 0.2038254, 'std': 0.30801567},
        'children': {'mean': 0.30370313, 'std': 0.2787335},
        'family': {'mean': 0.07277942, 'std': 0.20808344},
        'cousins': {'mean': -0.040398445, 'std': 0.27916202},
        'marriage': {'mean': 0.37971354, 'std': 0.31494072},
        'wedding': {'mean': 0.39682359, 'std': 0.28507116},
        'relatives': {'mean': 0.16356598, 'std': 0.21053126}
    }
}
```

6.4 Debias

This list contains the debiasing methods implemented so far in WEFE.

wefe.debias.hard_debias.HardDebias([...])	Hard Debias debiasing method.
wefe.debias.multiclass_hard_debias.MulticlassHardDebias([...])	Generalized version of Hard Debias that enables multiclass debiasing.
wefe.debias.repulsion_attraction_neutralization.RepulsionAttractionNeutralization([...])	Repulsion Attraction Neutralization method.
wefe.debias.double_hard_debias.DoubleHardDebias([...])	Double Hard Debias Method.
wefe.debias.half_sibling_regression.HalfSiblingRegression([...])	Half Sibling Debias method.

6.4.1 wefe.debias.hard_debias.HardDebias

class wefe.debias.hard_debias.**HardDebias**(*pca_args: Dict[str, Any] = {'n_components': 10}*, *verbose: bool = False*, *criterion_name: Optional[str] = None*)

Hard Debias debiasing method.

Hard debias is a method that allows mitigating biases through geometric operations on embeddings.

This method is binary because it only allows 2 classes of the same bias criterion, such as male or female.

Note: For a multiclass debias (such as for Latinos, Asians and Whites), it is recommended to visit *MulticlassHardDebias* class.

The main idea of this method is:

1. Identify a bias subspace through the defining sets. In the case of gender, these could be e.g. `[['woman', 'man'], ['she', 'he'], ...]`

2. Neutralize the bias subspace of embeddings that should not be biased. First, it is defined a set of words that are correct to be related to the bias criterion: the *criterion specific gender words*. For example, in the case of gender, *gender specific* words are: `['he', 'his', 'He', 'her', 'she', 'him', 'him', 'She', 'man', 'women', 'men', ...]`.

Then, it is defined that all words outside this set should have no relation to the bias criterion and thus have the possibility of being biased. (e.g. for the case of genthe bias direction, such that neither is closer to the bias direction than the other: `['doctor', 'nurse', ...]`). Therefore, this set of words is neutralized with respect to the bias subspace found in the previous step.

The neutralization is carried out under the following operation:

- u : embedding
- v : bias direction

First calculate the projection of the embedding on the bias subspace.

$$\text{bias subspace} = \frac{v \cdot (v \cdot u)}{(v \cdot v)}$$

Then subtract the projection from the embedding.

$$u' = u - \text{bias subspace}$$

3. Equalizate the embeddings with respect to the bias direction. Given an equalization set (set of word pairs such as `['she', 'he'], ['men', 'women'], ...`, but not limited to the definitional set) this step executes, for each pair, an equalization with respect to the bias direction. That is, it takes both embeddings of the pair and distributes them at the same distance from the bias direction, so that neither is closer to the bias direction than the other.

References

[1]: Bolukbasi, T., Chang, K. W., Zou, J. Y., Saligrama, V., & Kalai, A. T. (2016).
Man is to computer programmer as woman is to homemaker? debiasing word embeddings.
Advances in Neural Information Processing Systems.
[2]: https://github.com/tolga-b/debiaswe

Examples

Note: For more information on the use of mitigation methods, visit *Bias Mitigation (Debias)* in the User Guide.

To run the bias debiasing specified in the original paper, run:

```
>>> from wefe.datasets import fetch_debiaswe
>>> from wefe.debias.hard_debias import HardDebias
>>> from wefe.utils import load_test_model
>>>
>>> model = load_test_model()  # load a reduced version of word2vec
>>>
>>> # load the definitional and equalize pairs. Also, the gender specific words
>>> # that should be ignored in the debias process.
>>> debiaswe_wordsets = fetch_debiaswe()
>>>
>>> definitional_pairs = debiaswe_wordsets["definitional_pairs"]
>>> equalize_pairs = debiaswe_wordsets["equalize_pairs"]
>>> gender_specific = debiaswe_wordsets["gender_specific"]
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # instance the debias object that will perform the mitigation
>>> hd = HardDebias(verbose=False, criterion_name="gender")
>>>
>>> # fits the transformation parameters (bias direction, etc...)
>>> hd.fit(
...     model, definitional_pairs=definitional_pairs, equalize_pairs=equalize_pairs,
... )
>>>
>>> # perform the transformation (debiasing) on the embedding model
>>  # note that words specified in ignore will not be mitigated (see exception
>>  # to this in the transform documentation).
>>> gender_debiased_model = hd.transform(model, ignore=gender_specific, copy=True)
```

If you only want to run debias on a limited set of words, you can use the target parameter when running transform.

```
>>> targets = [
...     "executive",
...     "management",
...     "professional",
...     "corporation",
...     "salary",
...     "office",
...     "business",
...     "career",
...     "home",
...     "parents",
...     "children",
...     "family",
...     "cousins",
...     "marriage",
...     "wedding",
...     "relatives",
... ]
>>>
>>> hd = HardDebias(verbose=False, criterion_name="gender").fit(
...     model, definitional_pairs=definitional_pairs, equalize_pairs=equalize_pairs,
>>> )
>>>
>>> gender_debiased_model = hd.transform(model, target=targets, copy=True)
```

__init__(*pca_args: Dict[str, Any] = {'n_components': 10}*, *verbose: bool = False*, *criterion_name: Optional[str] = None*) → None

Initialize a Hard Debias instance.

Parameters

pca_args
[Dict[str, Any], optional] Arguments for the PCA that is calculated internally in the identification of the bias subspace, by default {"n_components": 10}

verbose
[bool, optional] True will print informative messages about the debiasing process, by default False.

criterion_name
[Optional[str], optional] The name of the criterion for which the debias is being executed, e.g., 'Gender'. This will indicate the name of the model returning transform, by default None

`fit`(*model:* WordEmbeddingModel, *definitional_pairs: List[List[str]]*, *equalize_pairs: Optional[List[List[str]]] = None*, ***fit_params*) → BaseDebias

Compute the bias direction and obtains the equalize embedding pairs.

Parameters

model
[WordEmbeddingModel] The word embedding model to debias.

definitional_pairs
[List[List[str]]] A sequence of string pairs that will be used to define the bias direction. For example, for the case of gender debias, this list could be [['woman', 'man'], ['girl', 'boy'], ['she', 'he'], ['mother', 'father'], ...].

equalize_pairs
[Optional[List[List[str]]], optional] A list with pairs of strings, which will be equalized. In the case of passing None, the equalization will be done over the word pairs passed in definitional_pairs, by default None.

Returns

BaseDebias
The debias method fitted.

`transform`(*model:* WordEmbeddingModel, *target: Optional[List[str]] = None*, *ignore: Optional[List[str]] = None*, *copy: bool = True*) → *WordEmbeddingModel*

Execute hard debias over the provided model.

Parameters

model
[WordEmbeddingModel] The word embedding model to debias.

target
[Optional[List[str]], optional] If a set of words is specified in target, the debias method will be performed only on the word embeddings of this set. If *None* is provided, the debias will be performed on all words (except those specified in ignore). Note that some words that are not in target may be modified due to the equalization process. By default *None*.

ignore
[Optional[List[str]], optional] If target is *None* and a set of words is specified in ignore, the debias method will perform the debias in all words except those specified in this set. Note that some words that are in ignore may be modified due to the equalization process. By default *None*.

copy
[bool, optional] If *True*, the debias will be performed on a copy of the model. If *False*, the debias will be applied on the same model delivered, causing its vectors to mutate. **WARNING:** Setting copy with *True* requires RAM at least 2x of the size of the model, otherwise the execution of the debias may raise to *MemoryError*, by default True.

Returns

WordEmbeddingModel
The debiased embedding model.

6.4.2 wefe.debias.multiclass_hard_debias.MulticlassHardDebias

class wefe.debias.multiclass_hard_debias.**MulticlassHardDebias**(*pca_args: Dict[str, Any] = {'n_components': 10}*, *verbose: bool = False*, *criterion_name: Optional[str] = None*)

Generalized version of Hard Debias that enables multiclass debiasing.

Generalized refers to the fact that this method extends Hard Debias in order to support more than two types of social target sets within the definitional set. For example, for the case of religion bias, it supports a debias using words associated with Christianity, Islam and Judaism.

References

[1]: Manzini, T., Chong, L. Y., Black, A. W., & Tsvetkov, Y. (2019, June).
Black is to Criminal as Caucasian is to Police: Detecting and Removing Multiclass Bias in Word Embeddings.
In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers) (pp. 615-621).
[2]: https://github.com/TManzini/DebiasMulticlassWordEmbedding

Examples

Note: For more information on the use of mitigation methods, visit *Bias Mitigation (Debias)* in the User Guide.

The following example shows how to run an ethnicity debias based on Black, Asian and Caucasian groups.

```
>>> from wefe.datasets import fetch_debias_multiclass, load_weat
>>> from wefe.debias.multiclass_hard_debias import MulticlassHardDebias
>>> from wefe.utils import load_test_model
>>>
>>> model = load_test_model()  # load a reduced version of word2vec
>>>
>>> # obtain the sets of words that will be used in the debias process.
>>> multiclass_debias_wordsets = fetch_debias_multiclass()
>>> weat_wordsets = load_weat()
>>>
>>> ethnicity_definitional_sets = (
...     multiclass_debias_wordsets["ethnicity_definitional_sets"]
... )
>>> ethnicity_equalize_sets = list(
...     multiclass_debias_wordsets["ethnicity_analogy_templates"].values()
... )
>>>
>>> # instance the debias object that will perform the mitigation
>>> mhd = MulticlassHardDebias(verbose=False, criterion_name="ethnicity")
>>> # fits the transformation parameters (bias direction, etc...)
>>> mhd.fit(
```

(continues on next page)

(continued from previous page)

```
...     model=model,
...     definitional_sets=ethnicity_definitional_sets,
...     equalize_sets=ethnicity_equalize_sets,
... )
>>>
>>> # perform the transformation (debiasing) on the embedding model
>>> ethnicity_debiased_model = mhd.transform(model, copy=True)
```

__init__(*pca_args: Dict[str, Any] = {'n_components': 10}*, *verbose: bool = False*, *criterion_name: Optional[str] = None*) → None

Initialize a Multiclass Hard Debias instance.

Parameters

pca_args
[Dict[str, Any], optional] Arguments for the PCA that is calculated internally in the identification of the bias subspace, by default {"n_components": 10}

verbose
[bool, optional] True will print informative messages about the debiasing process, by default False.

criterion_name
[Optional[str], optional] The name of the criterion for which the debias is being executed, e.g. 'Gender'. This will indicate the name of the model returning transform, by default None

fit(*model: WordEmbeddingModel*, *definitional_sets: List[List[str]]*, *equalize_sets: List[List[str]]*) → BaseDebias

Compute the bias direction and obtains the equalize embedding pairs.

Parameters

model
[WordEmbeddingModel] The word embedding model to debias.

definitional_sets
[List[List[str]]] A sequence of string pairs that will be used to define the bias direction. For example, for the case of gender debias, this list could be [['woman', 'man'], ['girl', 'boy'], ['she', 'he'], ['mother', 'father'], ...]. Multiclass hard debias also accepts lists of sets of more than two words, such as religion where sets of words representing Christianity, Islam and Judaism can be used. See the example for more information.

equalize_pairs
[Optional[List[List[str]]], optional] A list with pairs of strings, which will be equalized. In the case of passing None, the equalization will be done over the word pairs passed in definitional_sets, by default None.

Returns

BaseDebias
The debias method fitted.

transform(*model: WordEmbeddingModel*, *target: Optional[List[str]] = None*, *ignore: Optional[List[str]] = None*, *copy: bool = True*) → *WordEmbeddingModel*

Execute Multiclass Hard Debias over the provided model.

Parameters

model
[WordEmbeddingModel] The word embedding model to debias.

target
[Optional[List[str]], optional] If a set of words is specified in target, the debias method will be performed only on the word embeddings of this set. If *None* is provided, the debias will be performed on all words (except those specified in ignore). Note that some words that are not in target may be modified due to the equalization process. by default *None*.

ignore
[Optional[List[str]], optional] If target is *None* and a set of words is specified in ignore, the debias method will perform the debias in all words except those specified in this set. Note that some words that are in ignore may be modified due to the equalization process. by default *None*.

copy
[bool, optional] If *True*, the debias will be performed on a copy of the model. If *False*, the debias will be applied on the same model delivered, causing its vectors to mutate. **WARNING:** Setting copy with *True* requires RAM at least 2x of the size of the model, otherwise the execution of the debias may raise to *MemoryError*, by default True.

Returns

WordEmbeddingModel
The debiased embedding model.

6.4.3 wefe.debias.repulsion_attraction_neutralization.RepulsionAttractionNeutralization

class wefe.debias.repulsion_attraction_neutralization.**RepulsionAttractionNeutralization**(*pca_args: Dict[str, Any] = {'n_components': 10}*, *verbose: bool = False*, *criterion_name: Optional[str] = None*, *epochs: int = 300*, *theta: float = 0.05*, *n_neighbours: int = 100*, *learning_rate: float = 0.01*, *weights: List[float] = [0.33, 0.33, 0.33]*)

Repulsion Attraction Neutralization method.

Warning: This method only works if Pytorch is installed. If you do not have it installed, check https://pytorch.org/get-started/locally/ for further information.

This method allow reducing the bias of an embedding model creating a transformation such that the stereotypical information is minimized with minimal semantic offset. This transformation bases its operations on:

1. Repelling embeddings from neighbours with a high value of indirect bias (indicating a strong association due to bias), to minimize the bias based illicit associations.
2. Attracting debiased embeddings to the original representation, to minimize the loss of semantic meaning.
3. Neutralizing the bias direction of each word, minimizing its bias to any particular group.

This method is binary because it only allows 2 classes of the same bias criterion, such as male or female.

Note: For a multiclass debias (such as for Latinos, Asians and Whites), it is recommended to visit *MulticlassHardDebias* class.

The steps followed to perform the debias are:

1. Identify a bias subspace through the defining sets. In the case of gender, these could be e.g. `[['woman', 'man'], ['she', 'he'], ...]`
2. A multi-objective optimization is performed. For each vector w in the target set it is found its debias counterpart w_d by solving:

$$argmin(F_r(w_d), F_a(w_d), F_n(w_d))$$

where Fr, Fa, Fn are repulsion, attraction and neutralization functions defined as the following:

$$F_r(w_d) = \sum |cos(w_d, n_i)|/|S|$$

$$F_a(w_d) = |cos(w_d, w) - 1|/2$$

$$F_n(w_d) = |cos(w_d, g)|$$

The optimization is performed by formulating a single objective:

$$F(w_d) = \lambda_1 F_r(w_d) + \lambda_2 F_a(w_d) + \lambda_3 F_n(w_d)$$

In the original implementation is define a preserve set (V_p) corresponding to words for which gender carries semantic importance, this words are not included in the debias process.

In WEFE this words would be the ones included in the ignore parameter of the transform method. The words that are not present in V_p are the ones to be included in the debias process and form part of the debias set (V_d), in WEFE this words can be specified in the target parameter of the transform method.

References

[1]: Kumar, Vaibhav, Tenzin Singhay Bhotia y Tanmoy Chakraborty: Nurse is Closer to Woman than Surgeon? Mitigating Gender-Biased Proximities in Word Embeddings. CoRR,abs/2006.01938, 2020.
https://arxiv.org/abs/2006.01938
[2]: https://github.com/TimeTraveller-San/RAN-Debias

Examples

The following example shows how to execute Repulsion Attraction Neutralization method that reduces bias in a word embedding model:

```
>>> from wefe.debias.repulsion_attraction_neutralization import (
...   RepulsionAttractionNeutralization
... )
>>> from wefe.utils import load_test_model
>>> from wefe.datasets import fetch_debiaswe
>>>
>>> # load the model (in this case, the test model included in wefe)
```

(continues on next page)

(continued from previous page)

```
>>> model = load_test_model()
>>> # load definitional pairs, in this case definitinal pairs included in wefe
>>> debiaswe_wordsets = fetch_debiaswe()
>>> definitional_pairs = debiaswe_wordsets["definitional_pairs"]
>>>
>>> # instance and fit the method
>>> ran = RepulsionAttractionNeutralization().fit(
...     model = model,
...     definitional_pairs= definitional_pairs
...   )
>>> # execute the debias passing words over a set of target words
>>> debiased_model = ran.transform(
...    model = model, target = ['doctor','nurse','programmer']
... )
Copy argument is True. Transform will attempt to create a copyof the original model.
This may fail due to lack of memory.
Model copy created successfully.
>>> # if you don't want a set of words to be debiased include them in the ignore set
>>> gender_specific = debiaswe_wordsets["gender_specific"]
>>> debiased_model = ran.transform(
...    model = model, ignore= gender_specific
... )
```

__init__(*pca_args: Dict[str, Any] = {'n_components': 10}*, *verbose: bool = False*, *criterion_name: Optional[str] = None*, *epochs: int = 300*, *theta: float = 0.05*, *n_neighbours: int = 100*, *learning_rate: float = 0.01*, *weights: List[float] = [0.33, 0.33, 0.33]*) → None

Initialize a Repulsion Attraction Neutralization Debias instance.

Parameters

pca_args
[Dict[str, Any], optional] Arguments for the PCA that is calculated internally in the identification of the bias subspace, by default {"n_components": 10}

verbose
[bool, optional] True will print informative messages about the debiasing process, by default False.

criterion_name
[Optional[str], optional] The name of the criterion for which the debias is being executed, e.g., 'Gender'. This will indicate the name of the model returning transform, by default None

epochs
[int, optional] number of times that the minimization is done. By default 300

theta: float, optional
Indirect bias threshold to select neighbours for the repulsion set. By default 0.05

n_neighbours: int, optional
Number of neighbours to be consider for the repulsion set. By default 100

learning_rate: float, optional
Learning rate to be used by the optimizer during the optimization. By default 0.01

weights: List[float], optional
List of the 3 initial weights to be used. By default [0.33,0.33,0.33]

fit(*model:* WordEmbeddingModel, *definitional_pairs: Sequence[Sequence[str]]*) → BaseDebias

Compute the bias direction.

Parameters

model
[WordEmbeddingModel] The word embedding model to debias.

definitional_pairs
[Sequence[Sequence[str]]] A sequence of string pairs that will be used to define the bias direction. For example, for the case of gender debias, this list could be [['woman', 'man'], ['girl', 'boy'], ['she', 'he'], ['mother', 'father'], ...].

Returns

BaseDebias
The debias method fitted.

transform(*model:* WordEmbeddingModel, *target: Optional[List[str]] = None*, *ignore: Optional[List[str]] = []*, *copy: bool = True*) → *WordEmbeddingModel*

Execute Repulsion Attraction Neutralization Debias over the provided model.

Parameters

model
[WordEmbeddingModel] The word embedding model to debias.

target
[Optional[List[str]], optional] If a set of words is specified in target, the debias method will be performed only on the word embeddings of this set. If *None* is provided, the debias will be performed on all words (except those specified in ignore). by default *None*.

ignore
[Optional[List[str]], optional] If target is *None* and a set of words is specified in ignore, the debias method will perform the debias in all words except those specified in this set, by default *None*.

copy
[bool, optional] If *True*, the debias will be performed on a copy of the model. If *False*, the debias will be applied on the same model delivered, causing its vectors to mutate. **WARNING:** Setting copy with *True* requires RAM at least 2x of the size of the model, otherwise the execution of the debias may raise to *MemoryError*, by default True.

WordEmbeddingModel
The debiased embedding model.

6.4.4 wefe.debias.double_hard_debias.DoubleHardDebias

class wefe.debias.double_hard_debias.**DoubleHardDebias**(*pca_args: Dict[str, Any] = {'n_components': 10}*, *verbose: bool = False*, *criterion_name: Optional[str] = None*, *incremental_pca: bool = True*, *n_words: int = 1000*, *n_components: int = 4*)

Double Hard Debias Method.

This method allow reducing the bias of an embedding model through geometric operations between embeddings. This method is binary because it only allows 2 classes of the same bias criterion, such as male or female.

The main idea of this method is:

1. Identify a bias subspace through the defining sets. In the case of gender, these could be e.g. `[['woman', 'man'], ['she', 'he'], ...]`

2. Find the dominant directions of the entire set of vectors by doing a Principal components analysis over it.

3. Get the target words by finding the most biased words, this is the words tha are closest to the representation of each bias group. In case of gender `'he'` and `'she'`.

3. Try removing each component resulting of PCA and remove also the bias direction to every vector in the target set and find which component reduces bias the most.

4. Remove the dominant direction that most reduces bias and remove also the bias direction of the vectors in the target set.

References

[1]: Wang, Tianlu, Xi Victoria Lin, Nazneen Fatema Rajani, Bryan McCann, Vicente Or-donez y Caiming Xiong.

Double-Hard Debias: Tailoring Word Embeddings for GenderBias Mitigation.

CoRR, abs/2005.00965, 2020.https://arxiv.org/abs/2005.00965.

[2]: https://github.com/uvavision/Double-Hard-Debias

Examples

The following example shows how to execute Double Hard Debias method that reduces bias in a word embedding model:

```
>>> from wefe.debias.double_hard_debias import DoubleHardDebias
>>> from wefe.utils import load_test_model
>>> from wefe.datasets import fetch_debiaswe
>>>
>>> # load the model (in this case, the test model included in wefe)
>>> model = load_test_model()
>>> # load definitional pairs, in this case definitinal pairs included in wefe
>>> debiaswe_wordsets = fetch_debiaswe()
>>> definitional_pairs = debiaswe_wordsets["definitional_pairs"]
>>>
>>> # instance and fit the method including bias representation words,
>>> # in case of gender definitional_pairs=[['he', 'she'], ...]
>>> dhd = DoubleHardDebias().fit(
...     model=model,
...     definitional_pairs=definitional_pairs,
...     bias_representation=['he','she'])
>>> # execute the debias, if you don't want a set of words to be debiased
>>> # include them in the ignore set.
>>> gender_specific = debiaswe_wordsets["gender_specific"]
>>>
>>> debiased_model = dhd.transform(
...     model=model, ignore=gender_specific
... )
Copy argument is True. Transform will attempt to create a copy of the original
model. This may fail due to lack of memory.
Model copy created successfully.
```

If you want the debiased to be performed over a specific set of words you can specify them in the target parameter

```
>>> debiased_model = dhd.transform(
... model=model, target = ['doctor','nurse','programmer','teacher']
... )
Copy argument is True. Transform will attempt to create a copy of the original
model. This may fail due to lack of memory.
Model copy created successfully.
```

__init__(*pca_args: Dict[str, Any] = {'n_components': 10}*, *verbose: bool = False*, *criterion_name: Optional[str] = None*, *incremental_pca: bool = True*, *n_words: int = 1000*, *n_components: int = 4*) → None

Initialize a Double Hard Debias instance.

Parameters

pca_args
[Dict[str, Any], optional] Arguments for the PCA that is calculated internally in the identification of the bias subspace, by default {"n_components": 10}

verbose
[bool, optional] True will print informative messages about the debiasing process, by default False.

criterion_name
[Optional[str], optional] The name of the criterion for which the debias is being executed, e.g., 'Gender'. This will indicate the name of the model returning transform, by default None

incremental_pca: bool, optional
If *True*, incremental pca will be used over the entire set of vectors. If *False*, pca will be used over the entire set of vectors. **WARNING:** Running pca over the entire set of vectors may raise to *MemoryError*, by default True.

n_words: int, optional
Number of target words to be used for each bias group. By default 1000

n_components: int, optional
Numbers of components of PCA to be used to explore the one that reduces bias the most. Usually the best one is close to embedding dimension/100. By default 4.

fit(*model: WordEmbeddingModel*, *definitional_pairs: List[List[str]]*, *bias_representation: List[str]*, ***fit_params*) → BaseDebias

Compute the bias direction and get the principal components of the vectors.

Parameters

model
[WordEmbeddingModel] The word embedding model to debias.

definitional_pairs
[List[List[str]]] A sequence of string pairs that will be used to define the bias direction. For example, for the case of gender debias, this list could be [['woman', 'man'], ['girl', 'boy'], ['she', 'he'], ['mother', 'father'], ...].

bias_representation: List[str]
Two words that represents each bias group. In case of gender "he" and "she".

Returns

BaseDebias
The debias method fitted.

`get_target_words`(*model:* WordEmbeddingModel, *target: List[str]*, *ignore: List[str]*, *n_words: int*, *bias_representation: List[str]*) → List[str]

Obtain target words to be debiased.

This is done by searching the "n_words" most biased words by obtaining the words closest to each word in the bias_representation set (in case of gender "he" and "she").

Parameters

model: WordEmbeddingModel
The word embedding model to debias.

ignore: List[str]
Set of words to be ignored from the debias process.

n_words: int
number of target words to use.

bias_representation: List[str]
Two words that represents de bias groups.

Returns

List[str]
List of target words for each bias group

`transform`(*model:* WordEmbeddingModel, *target: Optional[List[str]] = None*, *ignore: Optional[List[str]] = []*, *copy: bool = True*) → *WordEmbeddingModel*

Execute hard debias over the provided model.

Parameters

model
[WordEmbeddingModel] The word embedding model to debias.

ignore
[List[str], optional] If set of words is specified in ignore, the debias method will perform the debias in all target words except those specified in this set, by default [].

copy
[bool, optional] If *True*, the debias will be performed on a copy of the model. If *False*, the debias will be applied on the same model delivered, causing its vectors to mutate. **WARNING:** Setting copy with *True* requires RAM at least 2x of the size of the model, otherwise the execution of the debias may raise to *MemoryError*, by default True.

Returns

WordEmbeddingModel
The debiased embedding model.

6.4.5 wefe.debias.half_sibling_regression.HalfSiblingRegression

class wefe.debias.half_sibling_regression.**HalfSiblingRegression**(*verbose: bool = False*, *criterion_name: Optional[str] = None*)

Half Sibling Debias method.

This method proposes to learn spurious gender information via causal inference by utilizing the statistical dependency between gender-biased word vectors and gender definition word vectors. The learned spurious gender information is then subtracted from the gender-biased word vectors to achieve gender-debiasing as the following where V_n are the debiased word vectors, Vn are non gender definition and G is the approximated gender information:

$$V'_n := V_n - G$$

G is obtained by predicting Non gender definition word vectors (V_n) using the gender-definition word vectors (V_d):

$$G := E[V_n|V_d]$$

The Prediction is done by a Ridge Regression following the next steps:

1. Compute the weight matrix of a Ridge Regression using two sets of words

$$W = ((V_d)^T V_d + \alpha I)^{-1}(V_d)^T V_n$$

2. Compute the gender information:

$$G = V_d W$$

3. Subtract gender information from non gender definition words:

$$V'_n = V_n - G$$

This method is binary because it only allows 2 classes of the same bias criterion, such as male or female. For a multiclass debias (such as for Latinos, Asians and Whites), it is recommended to visit MulticlassHardDebias class.

Warning: This method requires three times the memory of the model when a copy of the model is made and two times the memory of the model if not. Make sure this much memory is available.

References

[1]: Yang, Zekun y Juan Feng: A causal inference method for reducing gender bias in word embedding relations.
In Proceedings of the AAAI Conference on Artificial Intelligence, volumen 34, pages 9434–9441, 2020

[2]: https://github.com/KunkunYang/GenderBiasHSR

[3]: Bernhard Sch olkopf, David W. Hogg, Dun Wang, Daniel Foreman-Mackey, Dominik Jan-zing, Carl-Johann Simon-Gabriel, and Jonas Peters.
Modeling confounding by half-sibling regression.
Proceedings of the National Academy of Sciences, 113(27):7391–7398, 2016

Examples

The following example shows how to execute Half Sibling Regression Debias method that reduces bias in a word embedding model:

```
>>> from wefe.debias.half_sibling_regression import HalfSiblingRegression
>>> from wefe.utils import load_test_model
>>> from wefe.datasets import fetch_debiaswe
>>>
>>> # load the model (in this case, the test model included in wefe)
>>> model = load_test_model()
>>> # load gender specific words, in this case the ones included in wefe
>>> debiaswe_wordsets = fetch_debiaswe()
>>> gender_specific = debiaswe_wordsets["gender_specific"]
>>>
>>> # instance and fit the method
>>> hsr = HalfSiblingRegression().fit(
...     model=model, definitional_words=gender_specific
... )
>>> # execute the debias on the words not included in the gender definition set
>>> debiased_model = hsr.transform(model = model)
Copy argument is True. Transform will attempt to create a copy of the original
model. This may fail due to lack of memory.
Model copy created successfully.
>>>
>>>
>>> # if you want the debias over a specific set of words  you can
>>> #include them in the target parameter
>>> debiased_model = hsr.transform(
...     model=model, target=["doctor", "nurse", "programmer"]
... )
Copy argument is True. Transform will attempt to create a copy of the original
model. This may fail due to lack of memory.
Model copy created successfully.
>>>
>>> # if you want to exclude a set of words from the debias process
>>> # you can include them in the ignore parameter
>>> debiased_model = hsr.transform(
...     model=model, ignore=["dress", "beard", "niece", "nephew"]
... )
Copy argument is True. Transform will attempt to create a copy of the original
model. This may fail due to lack of memory.
Model copy created successfully.
```

__init__(*verbose: bool = False*, *criterion_name: Optional[str] = None*) → None

Initialize a Half Sibling Regression Debias instance.

Parameters

verbose
[bool, optional] True will print informative messages about the debiasing process, by default False.

criterion_name
[Optional[str], optional] The name of the criterion for which the debias is being executed,

e.g., 'Gender'. This will indicate the name of the model returning transform, by default None

fit(*model:* WordEmbeddingModel, *definitional_words: List[str]*, *alpha: float = 60*) → BaseDebias

Compute the weight matrix and the bias information.

Parameters

model: WordEmbeddingModel
The word embedding model to debias.

definitional_words: List[str]
List of strings. This list contains words that embody bias information by definition.

alpha: float
Ridge Regression constant. By default 60.

Returns

BaseDebias
The debias method fitted.

transform(*model:* WordEmbeddingModel, *target: Optional[List[str]] = None*, *ignore: Optional[List[str]] = None*, *copy: bool = True*) → *WordEmbeddingModel*

Substracts the gender information from vectors.

Parameters

model
[WordEmbeddingModel] The word embedding model to mitigate.

target
[Optional[List[str]], optional] If a set of words is specified in target, the debias method will be performed only on the word embeddings of this set. If *None* is provided, the debias will be performed on all non gender specific words (except those specified in ignore). Target words must not be included in the gender specific set. by default *None*.

ignore
[Optional[List[str]], optional] If target is *None* and a set of words is specified in ignore, the debias method will perform the debias in all non gender specific words except those specified in this set, by default *[]*.

copy
[bool, optional] If *True*, the debias will be performed on a copy of the model. If *False*, the debias will be applied on the same model delivered, causing its vectors to mutate. **WARNING:** Setting copy with *True* requires RAM at least 2x of the size of the model, otherwise the execution of the debias may raise to *MemoryError*, by default True.

Returns

WordEmbeddingModel
The debiased embedding model.

6.5 Datasets

The following functions allow you to load sets of words used in previous studies.

wefe.datasets.load_bingliu()	Load the Bing-Liu sentiment lexicon.
wefe.datasets.fetch_debias_multiclass()	Fetch the word sets used in the paper Black Is To Criminals as Caucasian
wefe.datasets.fetch_debiaswe()	Fetch the word sets used in the paper Man is to Computer Programmer as Woman is to Homemaker? from the source.
wefe.datasets.fetch_eds([occupations_year, ...])	Fetch the sets of words used in the experiments of the _Word Embeddings
wefe.datasets.load_weat()	Load the word sets used in the experiments of the

6.5.1 wefe.datasets.load_bingliu

wefe.datasets.**load_bingliu**() → Dict[str, List[str]]

Load the Bing-Liu sentiment lexicon.

Returns

dict
A dictionary with the positive and negative words.

References

Minqing Hu and Bing Liu. "Mining and Summarizing Customer Reviews." Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2004), Aug 22-25, 2004, Seattle, Washington, USA.

6.5.2 wefe.datasets.fetch_debias_multiclass

wefe.datasets.**fetch_debias_multiclass**() → Dict[str, Union[List[str], list]]

Fetch the word sets used in the paper Black Is To Criminals as Caucasian
Is To Police: Detecting And Removing Multiclass Bias In Word Embeddings.

This dataset contains gender (male, female), ethnicity (asian, black, white) and religion (christianity, judaism and islam) word sets. This helper allow accessing independently to each of the word sets (to be used as target or attribute sets in metrics) as well as to access them in the original format (to be used in debiasing methods). The dictionary keys whose names contain definitional sets and analogies templates are the keys that point to the original format focused on debiasing.

Returns

dict
A dictionary in which each key correspond to the name of the set and its values correspond to the word set.

References

[1]: Thomas Manzini, Lim Yao Chong,Alan W Black, and Yulia Tsvetkov.
Black is to Criminal as Caucasian is to Police: Detecting and Removing Multiclass
Bias in Word Embeddings.
In Proceedings of the 2019 Conference of the North American Chapter of the
Association for Computational Linguistics:
Human Language Technologies, Volume 1 (Long and Short Papers), pages 615–621,
Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.
[2]: https://github.com/TManzini/DebiasMulticlassWordEmbedding/blob/master/Debiasing/evalBias.py

6.5.3 wefe.datasets.fetch_debiaswe

wefe.datasets.**fetch_debiaswe**() → Dict[str, Union[List[str], list]]

Fetch the word sets used in the paper Man is to Computer Programmer as Woman is to Homemaker? from the source. It includes gender (male, female) terms and related word sets.

Returns

Dict[str, Union[List[str], list]]
A dictionary in which each key correspond to the name of the set and its values correspond to the word set.

References

[1]: Man is to Computer Programmer as Woman is to Homemaker?
Debiasing Word Embeddings by Tolga Bolukbasi, Kai-Wei Chang, James Zou, Venkatesh Saligrama, and Adam Kalai.
Proceedings of NIPS 2016.

6.5.4 wefe.datasets.fetch_eds

wefe.datasets.**fetch_eds**(*occupations_year: int = 2015*, *top_n_race_occupations: int = 10*) → Dict[str, List[str]]

Fetch the sets of words used in the experiments of the _Word Embeddings
Quantify 100 Years Of Gender And Ethnic **Stereotypes_** work.

This dataset includes the following word sets: - gender: male, female. - ethnicity: asian, black, white. - religion: christianity, judaism and islam. - adjetives: appearence, intelligence, otherization, sensitive.

Parameters

occupations_year
[int, optional] The year of the census for the occupations file. Available years: {'1850', '1860', '1870', '1880', '1900', '1910', '1920', '1930', '1940', '1950', '1960', '1970', '1980', '1990', '2000', '2001', '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013', '2014', '2015'} , by default 2015

top_n_race_occupations
[int, optional] The year of the census for the occupations file. The number of occupations by race, by default 10

Returns

dict
A dictionary with the word sets.

References

[1]: Word Embeddings quantify 100 years of gender and ethnic stereotypes.
Garg, N., Schiebinger, L., Jurafsky, D., & Zou, J. (2018).
Proceedings of the National Academy of Sciences, 115(16), E3635-E3644.

6.5.5 wefe.datasets.load_weat

`wefe.datasets.load_weat() → Dict[str, List[str]]`

Load the word sets used in the experiments of the
Semantics Derived Automatically From Language Corpora Contain Human-Like **Biases** work.

It includes the following word sets: - gender (male, female) - ethnicity (black, white) - pleasant, unpleasant - among others.

Returns

word_sets_dict
[dict] A dictionary with the word sets.

References

[1]: Semantics derived automatically from language corpora contain human-like biases.
Caliskan, A., Bryson, J. J., & Narayanan, A. (2017).
Science, 356(6334), 183-186.

6.6 Preprocessing

The following functions allow transforming sets of words and queries to embeddings. The documentation of the functions in this section are intended as a guide for WEFE developers.

wefe.preprocessing.preprocess_word(word[, ...])	pre-processes a word before it is searched in the model's vocabulary.
wefe.preprocessing.get_embeddings_from_set(...)	Transform a sequence of words into dictionary that maps word - word embedding.
wefe.preprocessing.get_embeddings_from_tuples(...)	Given a sequence of word sets, obtain their corresponding embeddings.
wefe.preprocessing.get_embeddings_from_query(...)	Obtain the word vectors associated with the provided Query.

6.6.1 wefe.preprocessing.preprocess_word

wefe.preprocessing.**preprocess_word**(*word: str*, *options: Dict[str, Union[str, bool, Callable]] = {}*, *vocab_prefix: Optional[str] = None*) → str

pre-processes a word before it is searched in the model's vocabulary.

Parameters

word
[str] Word to be preprocessed.

options
[Dict[str, Union[str, bool, Callable]], optional] Dictionary with arguments that specifies how the words will be preprocessed, The available word preprocessing options are as follows:

- `lowercase`: bool. Indicates if the words are transformed to lowercase.
- `uppercase`: bool. Indicates if the words are transformed to uppercase.
- `titlecase`: bool. Indicates if the words are transformed to titlecase.
- `strip_accents`: *bool*, *{'ascii', 'unicode'}*: Specifies if the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: Callable. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

By default, no preprocessing is generated, which is equivalent to {}

Returns

str
The pre-processed word according to the given parameters.

6.6.2 wefe.preprocessing.get_embeddings_from_set

wefe.preprocessing.**get_embeddings_from_set**(*model: WordEmbeddingModel*, *word_set: Sequence[str]*, *preprocessors: List[Dict[str, Union[str, bool, Callable]]] = [{}]*, *strategy: str = 'first'*, *normalize: bool = False*, *verbose: bool = False*) → Tuple[List[str], Dict[str, ndarray]]

Transform a sequence of words into dictionary that maps word - word embedding.

The method discard out words that are not in the model's vocabulary (according to the rules specified in the preprocessors).

Parameters

model
[WordEmbeddingModel] A word embeddding model

word_set
[Sequence[str]] A sequence with the words that this function will convert to embeddings.

preprocessors
[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor`

`{'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}]` `{}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary. by default [{}]

strategy
[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize
[bool, optional] True indicates that embeddings will be normalized, by default False

verbose
[bool, optional] Indicates whether the execution status of this function is printed, by default False

Returns

Tuple[List[str], Dict[str, np.ndarray]]
A tuple containing the words that could not be found and a dictionary with the found words and their corresponding embeddings.

6.6.3 wefe.preprocessing.get_embeddings_from_tuples

wefe.preprocessing.**get_embeddings_from_tuples**(*model:* WordEmbeddingModel, *sets:* *Sequence[Sequence[str]]*, *sets_name:* *Optional[str]* = *None*, *preprocessors:* *List[Dict[str, Union[str, bool, Callable]]]* = *[{}]*, *strategy:* *str* = *'first'*, *normalize:* *bool* = *False*, *discard_incomplete_sets:* *bool* = *True*, *warn_lost_sets:* *bool* = *True*, *verbose:* *bool* = *False*) → List[Dict[str, ndarray]]

Given a sequence of word sets, obtain their corresponding embeddings.

Parameters

model

sets

[Sequence[Sequence[str]]] A sequence containing word sets. Example: *[['woman', 'man'], ['she', 'he'], ['mother', 'father'] ...]*.

sets_name

[Union[str, optional]] The name of the set of word sets. Example: *definning sets*. This parameter is used only for printing. by default None

preprocessors

[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor {'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}]` `{}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True, "strip_accents": True}` is executed on these words and then they are searched in the model vocabulary. by default [{}]

strategy

[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize

[bool, optional] True indicates that embeddings will be normalized, by default False

discard_incomplete_sets

[bool, optional] True indicates that if a set could not be completely converted, it will be discarded., by default True

warn_lost_sets

[bool, optional] Indicates whether word sets that cannot be fully converted to embeddings are warned in the logger, by default True

verbose

[bool, optional] Indicates whether the execution status of this function is printed, by default False

Returns

List[EmbeddingDict]
A list of dictionaries. Each dictionary contains as keys a pair of words and as values their associated embeddings.

6.6.4 wefe.preprocessing.get_embeddings_from_query

wefe.preprocessing.**get_embeddings_from_query**(*model:* WordEmbeddingModel, *query:* Query, *lost_vocabulary_threshold: float = 0.2*, *preprocessors: List[Dict[str, Union[str, bool, Callable]]] = [{}]*, *strategy: str = 'first'*, *normalize: bool = False*, *warn_not_found_words: bool = False*, *verbose: bool = False*) → Optional[Tuple[Dict[str, Dict[str, ndarray]], Dict[str, Dict[str, ndarray]]]]

Obtain the word vectors associated with the provided Query.

The words that does not appears in the word embedding pretrained model vocabulary under the specified preprocessing are discarded. If the remaining words percentage in any query set is lower than the specified threshold, the function will return None.

Parameters

query
[Query] The query to be processed.

lost_vocabulary_threshold
[float, optional, by default 0.2] Indicates the proportional limit of words that any set of the query is allowed to lose when transforming its words into embeddings. In the case that any set of the query loses proportionally more words than this limit, this method will return None.

preprocessors
[List[Dict[str, Union[str, bool, Callable]]]] A list with preprocessor options.

A `preprocessor` is a dictionary that specifies what processing(s) are performed on each word before it is looked up in the model vocabulary. For example, the `preprocessor {'lowecase': True, 'strip_accents': True}` allows you to lowercase and remove the accent from each word before searching for them in the model vocabulary. Note that an empty dictionary `{}` indicates that no preprocessing is done.

The possible options for a preprocessor are:

- `lowercase`: `bool`. Indicates that the words are transformed to lowercase.
- `uppercase`: `bool`. Indicates that the words are transformed to uppercase.
- `titlecase`: `bool`. Indicates that the words are transformed to titlecase.
- `strip_accents`: `bool`, `{'ascii', 'unicode'}`: Specifies that the accents of the words are eliminated. The stripping type can be specified. True uses 'unicode' by default.
- `preprocessor`: `Callable`. It receives a function that operates on each word. In the case of specifying a function, it overrides the default preprocessor (i.e., the previous options stop working).

A list of preprocessor options allows you to search for several variants of the words into the model. For example, the preprocessors `[{}, {"lowercase": True, "strip_accents": True}] {}` allows searching first for the original words in the vocabulary of the model. In case some of them are not found, `{"lowercase": True,`

"strip_accents": True} is executed on these words and then they are searched in the model vocabulary. by default [{}]

strategy
[str, optional] The strategy indicates how it will use the preprocessed words: 'first' will include only the first transformed word found. 'all' will include all transformed words found, by default "first".

normalize
[bool, optional] True indicates that embeddings will be normalized, by default False

warn_not_found_words
[bool, optional] A flag that indicates if the function will warn (in the logger) the words that were not found in the model's vocabulary, by default False.

verbose
[bool, optional] Indicates whether the execution status of this function is printed, by default False

Returns

Union[Tuple[EmbeddingSets, EmbeddingSets], None]
A tuple of dictionaries containing the targets and attribute sets or None in case there is a set that has proportionally less embeddings than it was allowed to lose.

6.7 Utils

Collection of assorted utils.

wefe.utils.load_test_model()	Load a Word2vec subset to test metrics and debias methods.
wefe.utils.generate_subqueries_from_queries_list(...)	Generate a list of subqueries from queries.
wefe.utils.run_queries(metric, queries, models)	Run several queries over a several word embedding models using a specific metic.
wefe.utils.plot_queries_results(results[, by])	Plot the results obtained by a run_queries execution.
wefe.utils.create_ranking(results_dataframes)	Create a ranking form the aggregated scores of the provided dataframes.
wefe.utils.plot_ranking(ranking[, ...])	
wefe.utils.calculate_ranking_correlations(...)	Calculate the correlation between the calculated rankings.
wefe.utils.plot_ranking_correlations(...[, ...])	
wefe.utils.flair_to_gensim(flair_embedding)	

6.7.1 wefe.utils.load_test_model

wefe.utils.**load_test_model**() → *WordEmbeddingModel*

Load a Word2vec subset to test metrics and debias methods.

Returns

WordEmbeddingModel
The loaded model

6.7.2 wefe.utils.generate_subqueries_from_queries_list

wefe.utils.**generate_subqueries_from_queries_list**(*metric: BaseMetric*, *queries: List[*Query*]*) → List[*Query*]

Generate a list of subqueries from queries.

Parameters

metric
[BaseMetric] Some metric.

queries
[List[Query]] A list with queries.

Returns

List[Query]
A list with all the generated subqueries.

6.7.3 wefe.utils.run_queries

wefe.utils.**run_queries**(*metric: Type[BaseMetric]*, *queries: List[*Query*]*, *models: List[*WordEmbeddingModel*]*, *queries_set_name: str = 'Unnamed queries set'*, *lost_vocabulary_threshold: float = 0.2*, *metric_params: dict = {}*, *generate_subqueries: bool = False*, *aggregate_results: bool = False*, *aggregation_function: Union[str, Callable] = 'abs_avg'*, *return_only_aggregation: bool = False*, *warn_not_found_words: bool = False*) → DataFrame

Run several queries over a several word embedding models using a specific metic.

Parameters

metric
[Type[BaseMetric]] A metric class.

queries
[list] An iterable with a set of queries.

word_embeddings_models
[list] An iterable with a set of word embedding pretrianed models.

queries_set_name
[str, optional] The name of the set of queries or the criteria that will be tested, by default 'Unnamed queries set'

lost_vocabulary_threshold
[float, optional] The threshold that will be passed to the , by default 0.2

metric_params
[dict, optional] A dict with custom params that will passed to run_query method of the respective metric, by default {}

generate_subqueries: bool, optional
It indicates if the program, when detecting queries with a bigger template than the metric, should try to generate subqueries compatible with it. If any query is compatible with the metric template, then it appends the same query. DANGER: This may cause some comparisons to become meaningless when comparing biases that are not compatible with each other. By default, False.

aggregate_results
[bool, optional] A boolean that indicates if the results must be aggregated with some function.

aggregation_function
[Union[str, Callable], optional] The function that will be applied row by row to add the results. It must be pandas row compatible operation. Implemented functions: 'sum', 'abs_sub', 'avg' and 'abs_avg', by default 'abs_avg'.

return_only_aggregation
[bool, optional] If return_only_aggregation is True, only the column with the added queries is returned, by default False.

Returns

pd.DataFrame
A dataframe with the results. The index contains the word embedding model name and the columns the experiment name. Each cell represents the result of run a metric using a specific word embedding model and query.

6.7.4 wefe.utils.plot_queries_results

wefe.utils.**plot_queries_results**(*results: DataFrame*, *by: str = 'query'*) → Figure

Plot the results obtained by a run_queries execution.

Parameters

results
[pd.DataFrame] A dataframe that contains the result of having executed run_queries with a set of queries and word embeddings.

by
[{'query', 'model'}, optional] The aggregation function , by default 'query

Returns

plotly.Figure
A Figure that contains the generated graphic.

Raises

TypeError
if results is not a instance of pandas DataFrame.

6.7.5 wefe.utils.create_ranking

wefe.utils.**create_ranking**(*results_dataframes: List[DataFrame]*, *method: str = 'first'*, *ascending: bool = True*) → DataFrame

Create a ranking form the aggregated scores of the provided dataframes.

The function will assume that the aggregated scores are in the last column of each result dataframe. It uses pandas.DataFrame.rank to generate the ranks.

Parameters

results_dataframes
[List[pd.DataFrame]] A list or array of dataframes returned by the run_queries function.

method
[str, optional] How to rank the group of records that have the same value, by default 'first'. The options are: - average: average rank of the group - min: lowest rank in the group - max: highest rank in the group - first: ranks assigned in order they appear in the array - dense: like 'min', but rank always increases by 1 between groups.

ascending
[bool, optional] Whether or not the elements should be ranked in ascending order, by default True.

Returns

pd.DataFrame
A dataframe with the ranked scores.

Raises

Exception
If there is no average column in some result Dataframe.

TypeError
If some element of results_dataframes is not a pandas DataFrame.

6.7.6 wefe.utils.plot_ranking

wefe.utils.**plot_ranking**(*ranking: DataFrame*, *use_metric_as_facet: bool = False*) → Figure

6.7.7 wefe.utils.calculate_ranking_correlations

wefe.utils.**calculate_ranking_correlations**(*rankings: DataFrame*, *method: str = 'spearman'*) → DataFrame

Calculate the correlation between the calculated rankings.

It uses pandas corr() method to calculate the correlations. The method parameter documentarion was copied from the documentation of the pandas DataFrame.corr() method. To see the updated documentation, visit: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

Parameters

rankings
[pd.DataFrame] DataFrame that contains the calculated rankings.

method
[{'pearson', 'kendall', 'spearman'} or callable] Correlation type: - pearson : standard correlation coefficient - kendall : Kendall Tau correlation coefficient - spearman : Spearman rank correlation - callable: callable with input two 1d ndarrays and returning a float.

Returns

pd.DataFrame
A dataframe with the calculated correlations.

6.7.8 wefe.utils.plot_ranking_correlations

wefe.utils.**plot_ranking_correlations**(*correlation_matrix: DataFrame*, *title: str = ''*) → Figure

6.7.9 wefe.utils.flair_to_gensim

wefe.utils.**flair_to_gensim**(*flair_embedding*) → KeyedVectors

CHAPTER

SEVEN

MEASUREMENT FRAMEWORK

Below we present the main aspects of the measurement framework developed at WEFE.

Note: If you want to see tutorials on how to apply queries, visit *Bias Measurement* in the User Guide.

7.1 Target set

A target word set (denoted by T) corresponds to a set of words intended to denote a particular social group,which is defined by a certain criterion. This criterion can be any character, trait or origin that distinguishes groups of people from each other e.g., gender, social class, age, and ethnicity. For example, if the criterion is gender we can use it to distinguish two groups, *women and men*. Then, a set of target words representing the social group "*women*" could contain words like *'she'*, *'woman'*, *'girl'*, etc. Analogously a set of target words the representing the social group *'men'* could include *'he'*, *'man'*, *'boy'*, etc.

7.2 Attribute set

An attribute word set (denoted by A) is a set of words representing some attitude, characteristic, trait, occupational field, etc. that can be associated with individuals from any social group. For example, the set of *science* attribute words could contain words such as *'technology'*, *'physics'*, *'chemistry'*, while the *art* attribute words could have words like *'poetry'*, *'dance'*, *'literature'*.

7.3 Query

Queries are the main building blocks used by fairness metrics to measure bias of word embedding models. Formally, a query is a pair $Q = (\mathcal{T}, \mathcal{A})$ in which T is a set of target word sets, and A is a set of attribute word sets. For example,

consider the target word sets:

$$to$$

$$T_{\text{women}} = \{she, woman, girl, \ldots\},$$
$$T_{\text{men}} = \{he, man, boy, \ldots\},$$

$$= \{she, woman, girl, \ldots\}, T_{\text{men}} \{he, man, boy, \ldots\},$$

and the attribute word sets

$$to$$

$$A_{\text{science}} = \{math, physics, chemistry, \ldots\},$$
$$A_{\text{art}} = \{poetry, dance, literature, \ldots\}.$$

$$= \{math, physics, chemistry, \ldots\}, A_{\text{art}} \{poetry, dance, literature, \ldots\}.$$

Then the following is a query in our framework

$Q = (\{T_{\text{women}}, T_{\text{men}}\}, \{A_{\text{science}}, A_{\text{art}}\})$.

When a set of queries $\mathcal{Q} = Q_1, Q_2, \ldots, Q_n$ is intended to measure a single type of bias, we say that the set has a **Bias Criterion**. Examples of bias criteria are gender, ethnicity, religion, politics, social class, among others.

Warning: To accurately study the biases contained in word embeddings, queries may contain words that could be offensive to certain groups or individuals. The relationships studied between these words DO NOT represent the ideas, thoughts or beliefs of the authors of this library. This warning applies to all documentation.

7.4 Query Template

A query template is simply a pair $(t, a) \in \mathbb{N} \times \mathbb{N}$. We say that query $Q = (\mathcal{T}, \mathcal{A})$ satisfies a template (t, a) if $|\mathcal{T}| = t$ and $|\mathcal{A}| = a$.

7.5 Fairness Measure

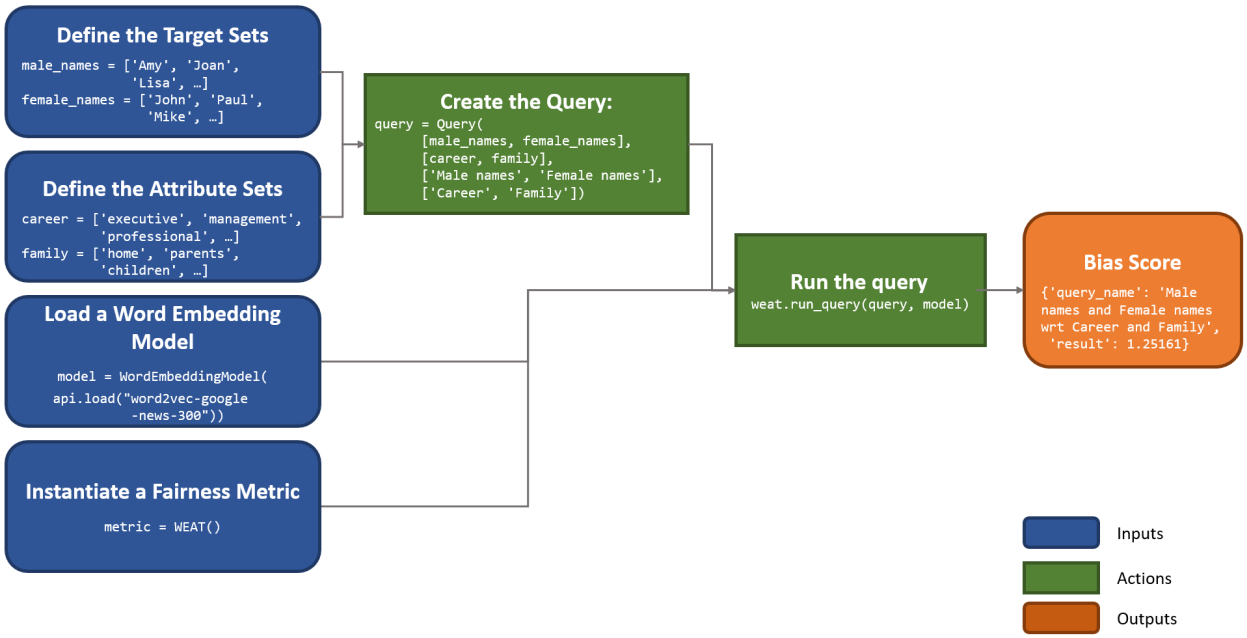
A fairness metric is a function that quantifies the degree of association between target and attribute words in a word embedding model. In our framework, every fairness metric is defined as a function that has a query and a model as input, and produces a real number as output.

Several fairness metrics have been proposed in the literature. But not all of them share a common input template for queries. Thus, we assume that every fairness metric comes with a template that essentially defines the shape of the input queries supported by the metric.

Formally, let F be a fairness metric with template $s_F = (t_F, a_F)$. Given an embedding model $\mathbf{M}$ and a query Q that satisfies s_F, the metric produces the value $F(\mathbf{M}, Q) \in \mathbb{R}$ that quantifies the degree of bias of $\mathbf{M}$ with respect to query Q.

7.6 Standard usage pattern of WEFE

The following flow chart shows how to perform a bias measurement using a gender query, word2vec embeddings and the WEAT metric.



To see the implementation of this query using WEFE, refer to the *Quick start* section.

7.7 Metrics Implemented So Far

WEFE implements the following bias measurement metrics:

- Word Embedding Association Test (*WEAT*)
- Relative Norm Distance (*RND*)
- Relative Negative Sentiment Bias (*RNSB*)
- Mean Average Cosine Similarity (*MAC*)
- Embedding Coherence Test (*ECT*)
- Relational Inner Product Association (*RIPA*)

CHAPTER

EIGHT

MITIGATION FRAMEWORK

Note: If you want to see tutorials on how to mitigate (debias) bias in word embedding models, visit *Bias Mitigation* in the User Guide.

WEFE standardizes all mitigation methods through an interface inherited from `scikit-learn` basic data transformations: the `fit-transform` interface.

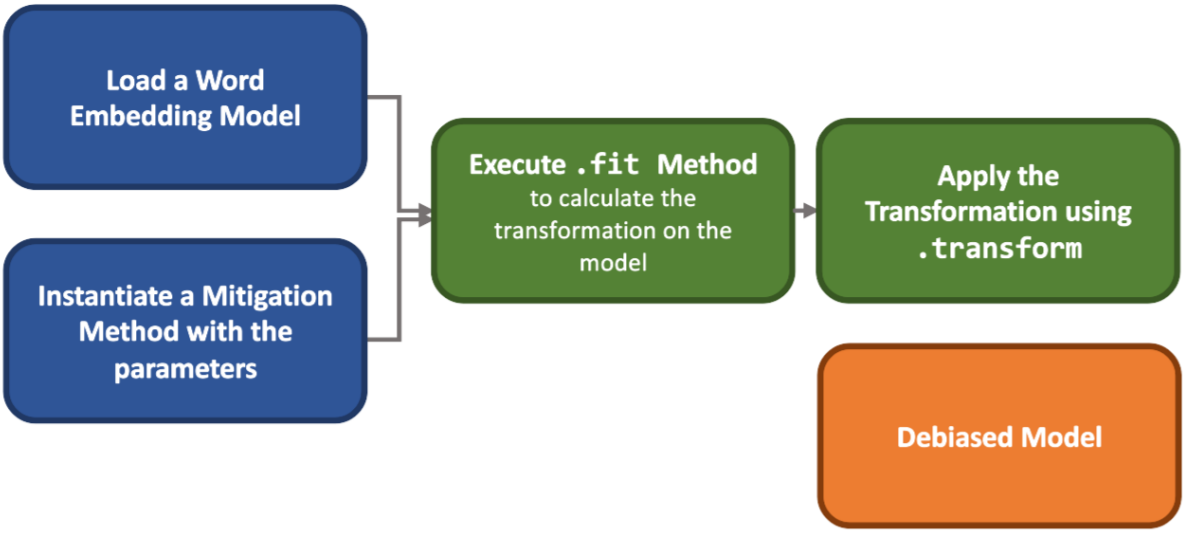
8.1 Fit method

The first step `fit`, consists in learning the corresponding mitigation transformation, which usually corresponds to a matrix projection of the embedding space. This method is quite flexible: it can accept multiple sets of words and other parameters.

8.2 Transform method

The `transform` method applies the transformation learned in the previous step to words residing in the original embedding space. The method is rigid and only accepts lists of words that should be mitigated (`target`) or words that should be omitted (`ignore`).

The process by which debiasing methods are used is shown in the following Figure.



8.3 Mitigation Methods Implemented So Far

WEFE implements the following bias mitigation (debias) metrics:

- *HardDebias*,
- *MulticlassHardDebias*,
- *DoubleHardDebias*,
- *RepulsionAttractionNeutralization*,
- *HalfSiblingRegression*,

Except for `MulticlassHardDebias`, all methods are limited to binary criteria, such as gender.

CHAPTER

NINE

REFERENCES

The intention of this section is to provide a list of the works on which WEFE relies as well as a rough reference of works on measuring and mitigating bias in word embeddings.

9.1 Measurements and Case Studies

- Caliskan, A., Bryson, J. J., & Narayanan, A. (2017). Semantics derived automatically from language corpora contain human-like
- Garg, N., Schiebinger, L., Jurafsky, D., & Zou, J. (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes.
- Sweeney, C., & Najafian, M. (2019, July). A Transparent Framework for Evaluating Unintended Demographic Bias in Word Emb
- Dev, S., & Phillips, J. (2019, April). Attenuating Bias in Word vectors. In Proceedings of the 22nd International Conference on A
- Ethayarajh, K., & Duvenaud, D., & Hirst, G. (2019, July). Understanding Undesirable Word Embedding Associations. Proceedin

9.2 Bias Mitigation

- Bolukbasi, T., Chang, K. W., Zou, J., Saligrama, V., & Kalai, A. (2016). Quantifying and reducing stereotypes in word embeddin
- Bolukbasi, T., Chang, K. W., Zou, J. Y., Saligrama, V., & Kalai, A. T. (2016). Man is to computer programmer as woman is to ho
- Zhao, J., Zhou, Y., Li, Z., Wang, W., & Chang, K. W. (2018). Learning gender-neutral word embeddings. arXiv preprint arXiv:18
- Zhao, J., Wang, T., Yatskar, M., Ordonez, V., & Chang, K. W. (2017). Men also like shopping: Reducing gender bias amplificatio
- Black is to Criminal as Caucasian is to Police: Detecting and Removing Multiclass Bias in Word Embeddings.
- Gonen, H., & Goldberg, Y. (2019). Lipstick on a pig: Debiasing methods cover up systematic gender biases in word embeddings

9.3 Surveys and other resources

A Survey on Bias and Fairness in Machine Learning

- Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., & Galstyan, A. (2019). A survey on bias and fairness in machine learning. a
- Bakarov, A. (2018). A survey of word embeddings evaluation methods. arXiv preprint arXiv:1801.09536.
- Camacho-Collados, J., & Pilehvar, M. T. (2018). From word to sense embeddings: A survey on vector representations of meanin

Bias in Contextualized Word Embeddings

- Zhao, J., Wang, T., Yatskar, M., Cotterell, R., Ordonez, V., & Chang, K. W. (2019). Gender bias in contextualized word embeddi

- Basta, C., Costa-jussà, M. R., & Casas, N. (2019). Evaluating the underlying gender bias in contextualized word embeddings. arX
- Kurita, K., Vyas, N., Pareek, A., Black, A. W., & Tsvetkov, Y. (2019). Measuring bias in contextualized word representations. arX
- Tan, Y. C., & Celis, L. E. (2019). Assessing social and intersectional biases in contextualized word representations. In Advances
- Stereoset: A Measure of Bias in Language Models

CHAPTER

TEN

WEFE CASE STUDY REPLICATION

The following code replicates the case study presented in our paper:

P. Badilla, F. Bravo-Marquez, and J. Pérez WEFE: The Word Embeddings Fairness Evaluation Framework In Proceedings of the 29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence (IJCAI-PRICAI 2020), Yokohama, Japan.

In this study we evaluate:

- Multiple queries grouped according to different criteria (gender, ethnicity, religion)
- Multiple embeddings (`word2vec-google-news`, `glove-wikipedia`, `glove-twitter`, `conceptnet`, `lexvec`, `fasttext-wiki-news`)
- Multiple metrics (`WEAT` and its variant, `WEAT effect size`, `RND`, `RNSB`).

After grouping the results by each criterion and metric, the rankings of the bias scores of each embedding model are calculated and plotted. An overall ranking is also computed, which is simply the sum of all rankings by model and metric.

Finally, the matrix of correlations between these rankings is calculated and plotted.

The code for this experiment is relatively long to run. A Jupyter Notebook with the code is provided in the following link.

CHAPTER

ELEVEN

PREVIOUS STUDIES REPLICATION

All replications of other studies that WEFE has currently implemented are in the Examples folder.

Below we list some examples:

11.1 Semantics derived automatically from language corpora contain human-like biases (WEAT)

The following notebook reproduces the experiments performed in the following paper:

> Semantics derived automatically from language corpora contain human-like biases. Aylin Caliskan, Joanna J. Bryson, Arvind Narayanan

Note: Due to the formulation of the metric and the methods to transform the word to embeddings, our results are not exactly the same as those reported in the original paper. However, our results are still very similar to those in the original paper.

11.2 A transparent framework for evaluating unintended demographic bias in word embeddings (RNSB)

The following notebook replicates the experiments carried out in the following paper:

> Chris Sweeney and Maryam Najafian. A transparent framework for evaluating unintended demographic bias in word embeddings. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 1662–1667, 2019.

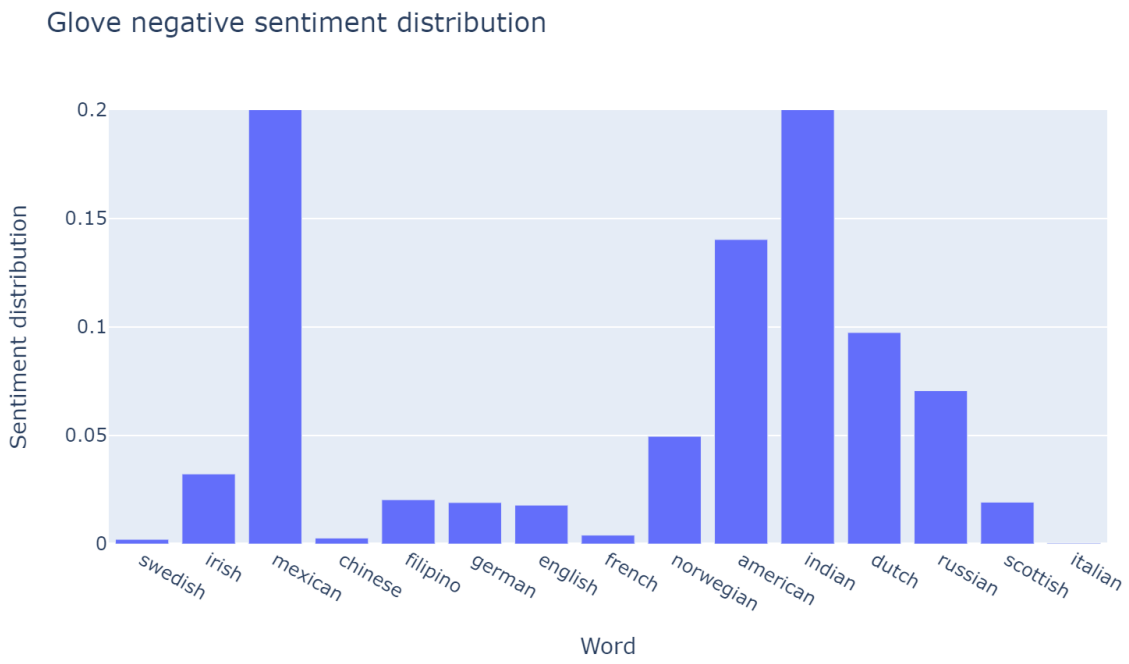
Note: Due to the formulation of the metric (it trains a logistic regression in each execution) our results are not exactly the same as those reported in the original paper. However, our results are still very similar to those in the original paper.

```
>>> from wefe.datasets import load_bingliu
>>> from wefe.metrics import RNSB
>>> from wefe.query import Query
>>> from wefe.word_embedding import
>>>
>>> import pandas as pd
>>> import plotly.express as px
```

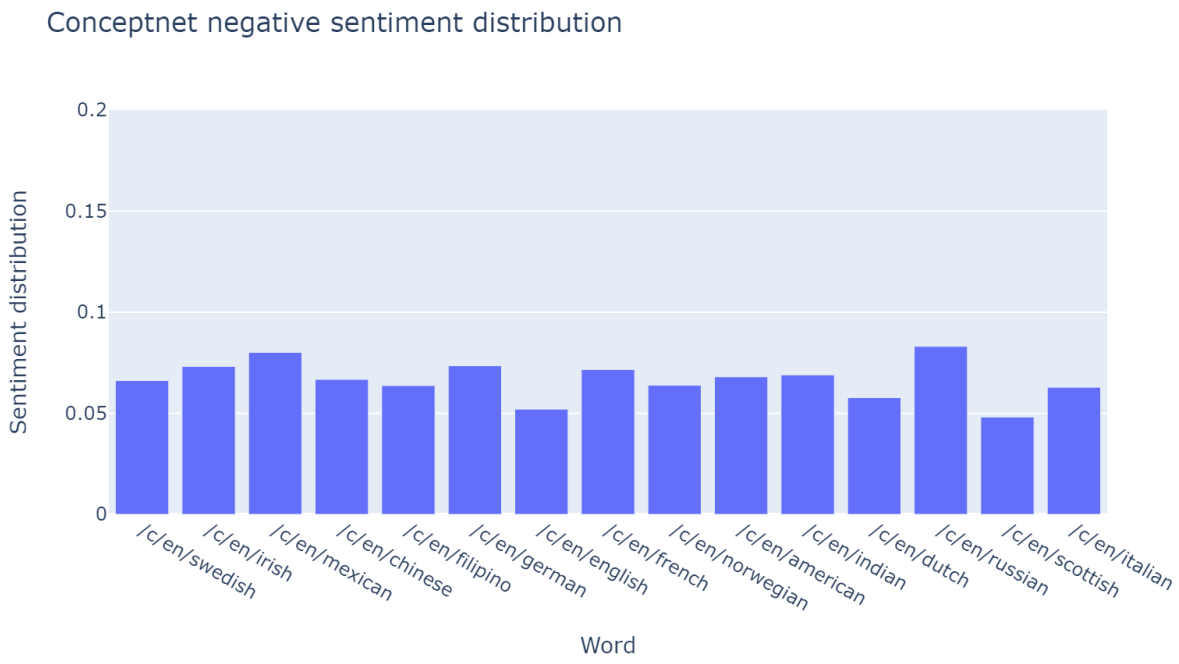
(continues on next page)

(continued from previous page)

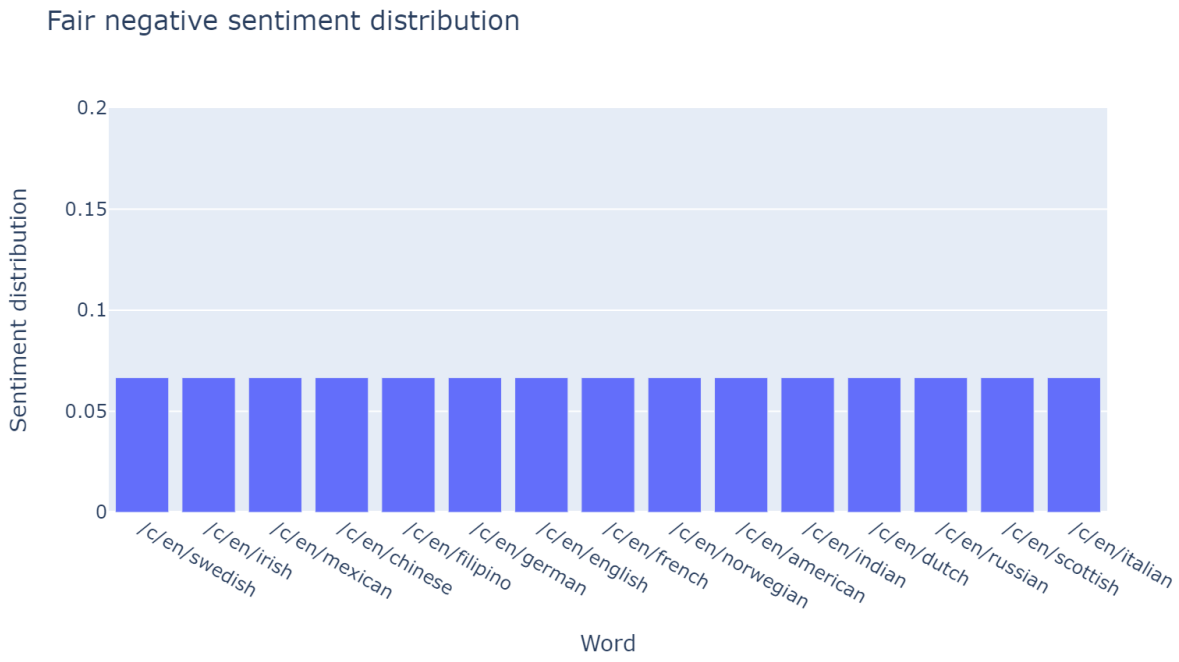
```
>>> import gensim.downloader as api
>>>
>>> # load the target word sets.
>>> # In this case each word is an objective set because each of them represents a␣
↪different social group.
>>> RNSB_words = [
>>>     ['swedish'], ['irish'], ['mexican'], ['chinese'], ['filipino'], ['german'], [
↪'english'],
>>>     ['french'], ['norwegian'], ['american'], ['indian'], ['dutch'], ['russian'],
>>>     ['scottish'], ['italian']
>>> ]
>>>
>>> bing_liu = load_bingliu()
>>>
>>> # Create the query
>>> query = Query(RNSB_words,
>>>               [bing_liu['positive_words'], bing_liu['negative_words']])
>>>
>>> # Fetch the models
>>> glove = (api.load('glove-wiki-gigaword-300'),
>>>                           'glove-wiki-gigaword-300')
>>> # note that conceptnet uses a /c/en/ prefix before each word.
>>> conceptnet = (api.load('conceptnet-numberbatch-17-06-300'),
>>>                                 'conceptnet-numberbatch-17',
>>>                                 vocab_prefix='/c/en/')
>>>
>>> # Run the queries
>>> glove_results = RNSB().run_query(query, glove)
>>> conceptnet_results = RNSB().run_query(query, conceptnet)
>>>
>>>
>>> # Show the results obtained with glove
>>> glove_fig = px.bar(
>>>     pd.DataFrame(glove_results['negative_sentiment_distribution'],
>>>                  columns=['Word', 'Sentiment distribution']), x='Word',
>>>     y='Sentiment distribution', title='Glove negative sentiment distribution')
>>> glove_fig.update_yaxes(range=[0, 0.2])
>>> glove_fig.show()
```




```
>>> # Show the results obtained with conceptnet
>>> conceptnet_fig = px.bar(
>>>     pd.DataFrame(conceptnet_results['negative_sentiment_distribution'],
>>>                  columns=['Word', 'Sentiment distribution']), x='Word',
>>>     y='Sentiment distribution',
>>>     title='Conceptnet negative sentiment distribution')
>>> conceptnet_fig.update_yaxes(range=[0, 0.2])
>>> conceptnet_fig.show()
```




```
>>> # Finally, we show the fair distribution of sentiments.
>>> fair_distribution = pd.DataFrame(
>>>     conceptnet_results['negative_sentiment_distribution'],
>>>     columns=['Word', 'Sentiment distribution'])
>>> fair_distribution['Sentiment distribution'] = np.ones(
>>>     fair_distribution.shape[0]) / fair_distribution.shape[0]
>>>
>>> fair_distribution_fig = px.bar(fair_distribution, x='Word',
>>>                                y='Sentiment distribution',
>>>                                title='Fair negative sentiment distribution')
>>> fair_distribution_fig.update_yaxes(range=[0, 0.2])
>>> fair_distribution_fig.show()
```



Note: This code is not executed when compiling the documentation due to the long processing time. Instead, the tables and plots of these results were embedded. The code is available for execution in the following notebook.

CHAPTER

TWELVE

MULTILINGUAL GENDER BIAS MEASUREMENT EXAMPLES

The notebooks located in multilingual examples folder show how to measure gender bias in static embeddings from gender queries in different languages.

The word embedding models in different languages are obtained from the flair library. The notebooks are self-contained: they contain everything necessary to load the embeddings and execute the queries and allow flair to be installed in case it is not already installed.

Available languages are English, Spanish, French, German, Italian, Spanish, Swedish, Dutch.

Warning: The words sets used in the notebooks were translated using google translator. Therefore, it is possible that some concepts may have been mistranslated and may require some correction. The original English concepts can be loaded using load_weat util. Read the instructions carefully and use the notebooks and its results with caution!

CHAPTER

THIRTEEN

BENCHMARK

To the best of our knowledge, there are only three other libraries besides WEFE that implement bias measurement and mitigation methods for word embeddings: Fair Embedding Engine (FEE), Responsibly, and EmbeddingBiasScores.

According to its authors, Fair Embedding Engine is defined as "A library for analyzing and mitigating gender bias in word embeddings", Responsibly is defined as "A toolkit for auditing and mitigating bias and fairness of machine learning systems". Finally, EmbeddingBiasScores describes itself as a collection of implementations and wrappers of bias scores for text embeddings.

The documentation for these three libraries can be found at the following links:

- https://github.com/FEE-Fair-Embedding-Engine/FEE
- https://docs.responsibly.ai/
- https://github.com/HammerLabML/EmbeddingBiasScores

The benchmark presented here compares these three libraries against WEFE according to the following criteria:

1. Ease of installation.
2. Quality of the package and documentation.
3. Ease of loading models.
4. Ease of running bias measurements.
5. Ease of running bias mitigation algorithms.
6. Implemented metrics and mitigation methods.

13.1 1. Ease of installation

This comparison aims to evaluate how easy it is to install the library.

13.1.1 WEFE

According to the documentation, WEFE is available for installation using the Python Package Index (via pip) as well as via conda.

```
pip install --upgrade wefe
# or
conda install -c pbadilla wefe
```

13.1.2 Fair Embedding Engine

In the case of FEE, neither the documentation nor the repository indicates how to install the package. Therefore, the easiest thing to do in this case is to clone the repository and then install the requirements , as described in the following steps:

1. Clone the repo

```
$ git clone https://github.com/FEE-Fair-Embedding-Engine/FEE
```

2. Install the requirements.

```
$ pip install -r FEE/requirements.txt
$ pip install sympy
$ pip install -U gensim==3.8.3
```

13.1.3 Responsibly

According to its documentation, responsibly is hosted in the Python Package Index so it can be installed using pip.

```
$ pip install responsibly
```

13.1.4 EmbeddingBiasScores

In the case of EmbeddingBiasScores, the documentation indicates that the repository can be cloned and then installed locally.

```
$ git clone https://github.com/HammerLabML/EmbeddingBiasScores.git
$ pip install -r EmbeddingBiasScores/requirements.txt
```

13.1.5 Conclusion

Both WEFE and Responsibly are hosted in the Python package index, which simplifies their installation and dependency handling, lowering the barrier to entry. FEE and EmbeddingBiasScores, on the other hand, require ad hoc installation procedures that require more advanced knowledge of Python and Pip.

13.2 2. Source Code Quality and Documentation

This benchmark seeks to compare the quality of documentation as well as other software quality features such as testing and continuous integration.

13.2.1 WEFE

WEFE has a complete documentation site that explains in detail how to use the package: an about page with the motivation and goals of the project, a quick start page showing how to install the library, several user guides on how to measure and mitigate bias in word embeddings, a detailed API of the implemented methods, theoretical background in the area, and finally implementations of previous case studies.

In addition, most of the code is tested and developed using continuous integration mechanisms (through a linter and testing mechanisms in Github Actions), which are well-established practices in software development.

13.2.2 Fair Embedding Engine

FEE' documentation covers only the basic aspects of the API and a flowchart showing the main concepts of the library. The documentation does not include user guides, code examples, or theoretical background on the implemented methods.

In terms of software engineering practices and standards, no tests, linter, or continuous integration mechanisms could be identified.

13.2.3 Responsibly

Responsibly has a complete documentation site that explains how to use the package: an index page with the main project information and a quick start page that shows how to install the library, demos that act as user manuals, and a detailed API of the implemented methods.

In addition, most of the code is tested and developed using continuous integration mechanisms (through a linter and testing in Github Actions).

13.2.4 EmbeddingBiasScores

It was not possible to find formal documentation explaining how to run bias tests in EmbeddingBiasScores. There is only a small Jupyter notebook with some use cases, which at the time of writing had several flaws that made it difficult to understand and use.

No testing, linter, or continuous integration mechanisms could be identified.

13.2.5 Conclusion

In terms of documentation, WEFE contains much more detailed documentation than the other libraries, with more extensive manuals and replications of previous case studies. Responsibly has sufficient documentation to execute its main functionalities without major problems, however, it is not as exhaustive as that of WEFE. FEE, only provides API documentation, which in our opinion is not sufficient for new users to use it without problems. Finally, EmbeddingBiasScores only presents a Jupyter notebook with some implementation examples.

With respect to software quality, both FEE and Responsibly comply with well-established software development practices (i.e., testing, continuous integration, linter). FEE and EmbeddingBiasScores, on the other hand, do not have any of these practices in place

13.3 3. Ease of loading models

In this section we will compare how easy it is to load a pre-trained word embedding (WE) model from each library. Two settings are compared: loading a model from Gensim's API (`glove-twitter-25`) and loading a model from a binary file (`word2vec`).

The second setting requires downloading a WE model trained with the original word2vec implementation, which can be obtained as follows:

```
# !wget https://github.com/RaRe-Technologies/gensim-data/releases/download/word2vec-
↪google-news-300/word2vec-google-news-300.gz
# !gzip -dv word2vec-google-news-300.gz
```

13.3.1 WEFE

In WEFE, WE models are represented internally by wrapping Gensim models. This means that the model loading process (either from the API or from a file) is handled by Gensim loaders, while the class that generates the objects that allow access to the embeddings is managed by WEFE.

The following code shows how to load a glove model using the Gensim API from within WEFE:

```
from wefe.word_embedding_model import WordEmbeddingModel
import gensim.downloader as api

# load glove
twitter_25 = api.load("glove-twitter-25")
model = WordEmbeddingModel(twitter_25, "glove twitter dim=25")
```

The following code shows how to load a word2vec model trained with the original implementation.

```
from wefe.word_embedding_model import WordEmbeddingModel
from gensim.models.keyedvectors import KeyedVectors

# load word2vec
word2vec = api.load("word2vec-google-news-300")
model = WordEmbeddingModel(word2vec, "word2vec-google-news-300")
```

13.3.2 FEE

FEE also offers direct support for loading WE models from its API through the following code. In this case, model loading is coupled to the WE class, which provides the methods to access the embeddings.

```
from FEE.fee.embedding.loader import WE

fee_model = WE().load(ename="glove-twitter-25")
```

```
from FEE.fee.embedding.loader import WE

fee_model = WE().load(fname="word2vec-google-news-300", format="bin")
```

13.3.3 Responsibly and EmbeddingBiasScores

Neither Responsibly nor EmbeddingBiasScores implement their own interfaces to handle WE models. Users must rely on Gensim or other external libraries for this purpose. This can be expressed as shown in the following script:

```
# load twitter_25 model from gensim api
twitter_25 = api.load("glove-twitter-25")

# load word2vec model from file
word2vec = KeyedVectors.load_word2vec_format("word2vec-google-news-300", binary=True)
```

13.3.4 Conclusion

As discussed above, both WEFE and FEE implement their own interfaces to internally manage access to WE models. Responsibly and EmbeddingBiasScores lack such functionalities, which may complicate their use.

13.4 4. Ease of running bias measurements.

The following section aims to compare the execution of fairness metrics in the libraries included in this study. To make the benchmark as objective as possible, the set of words and the WE model are kept fixed throughout the comparison, and only the metrics are allowed to vary.

```
# words to evaluate

female_terms = ["female", "woman", "girl", "sister", "she", "her", "hers", "daughter"]
male_terms = ["male", "man", "boy", "brother", "he", "him", "his", "son"]

family_terms = [
    "home",
    "parents",
    "children",
    "family",
    "cousins",
    "marriage",
    "wedding",
    "relatives",
]
career_terms = [
    "executive",
    "management",
    "professional",
    "corporation",
    "salary",
    "office",
    "business",
    "career",
]

# optional, only for wefe usage.
target_sets_names = ["Female terms", "Male terms"]
attribute_sets_names = ["Family terms", "Career terms"]
```

13.4.1 WEFE

WEFE defines a standardized framework for executing metrics: in short, it is necessary to define a query that will act as a container for the words to be tested and then, together with the model, will be provided as input to some metric.

The outputs of the metrics are contained in dictionaries that allow additional metadata to be included to the output.

```
# import the modules
from wefe.query import Query

# 1. create the query
query = Query(
    [female_terms, male_terms],
    [family_terms, career_terms],
    target_sets_names,
    attribute_sets_names,
)
query
```

```
<Query: Female terms and Male terms wrt Family terms and Career terms
- Target sets: [['female', 'woman', 'girl', 'sister', 'she', 'her', 'hers', 'daughter'],␣
↪['male', 'man', 'boy', 'brother', 'he', 'him', 'his', 'son']]
- Attribute sets:[['home', 'parents', 'children', 'family', 'cousins', 'marriage',
↪'wedding', 'relatives'], ['executive', 'management', 'professional', 'corporation',
↪'salary', 'office', 'business', 'career']]>
```

```
from wefe.metrics.WEAT import WEAT

# 2. instance a WEAT metric and pass the query plus the model.
weat = WEAT()
result = weat.run_query(query, model)
result
```

```
{'query_name': 'Female terms and Male terms wrt Family terms and Career terms',
 'result': 0.46343881433131173,
 'weat': 0.46343881433131173,
 'effect_size': 0.4507652792646716,
 'p_value': nan}
```

Since the `run_query` method is independent of the query and the model, it can receive additional parameters that customize the process. In this case, we show how to normalize the words before searching for them in the model (i.e., lowercase them and remove their accents).

```
weat = WEAT()
result = weat.run_query(
    query,
    model,
    preprocessors=[{"lowercase": True, "strip_accents": True}],
)
result
```

```
{'query_name': 'Female terms and Male terms wrt Family terms and Career terms',
 'result': 0.46343881433131173,
```

(continues on next page)

(continued from previous page)

```
 'weat': 0.46343881433131173,
 'effect_size': 0.4507652792646716,
 'p_value': nan}
```

Next, we show how to report the corresponding p-value through a permutation test.

```
weat = WEAT()
result = weat.run_query(
    query,
    model,
    calculate_p_value=True,
)
result
```

```
{'query_name': 'Female terms and Male terms wrt Family terms and Career terms',
 'result': 0.46343881433131173,
 'weat': 0.46343881433131173,
 'effect_size': 0.4507652792646716,
 'p_value': 0.19068093190680932}
```

This interface allows us to easily switch to similar metrics (i.e., supporting the same number of number of word sets).

```
from wefe.metrics import RNSB

rnsb = RNSB()
result = rnsb.run_query(query, model)
result
```

```
{'query_name': 'Female terms and Male terms wrt Family terms and Career terms',
 'result': 0.09051558681296493,
 'rnsb': 0.09051558681296493,
 'negative_sentiment_probabilities': {'female': 0.5285811053851917,
  'woman': 0.3031782770423851,
  'girl': 0.20810547466232254,
  'sister': 0.17327510211466302,
  'she': 0.4165425516161486,
  'her': 0.3895078245770702,
  'hers': 0.31412920848479164,
  'daughter': 0.13146512364633123,
  'male': 0.42679205714649815,
  'man': 0.43079499436045987,
  'boy': 0.21701323144255624,
  'brother': 0.19983034212661,
  'he': 0.5645185337599223,
  'him': 0.49470907399126185,
  'his': 0.552712793795697,
  'son': 0.17457869573293805},
 'negative_sentiment_distribution': {'female': 0.09565807331470504,
  'woman': 0.054866603359974946,
  'girl': 0.03766114329405169,
  'sister': 0.031357841309175544,
  'she': 0.07538229712572722,
```

(continues on next page)

(continued from previous page)

```
  'her': 0.07048978417965314,
  'hers': 0.05684840897525258,
  'daughter': 0.02379143012863325,
  'male': 0.07723716469755836,
  'man': 0.0779615819300061,
  'boy': 0.03927319268906782,
  'brother': 0.036163580806998274,
  'he': 0.10216172076480977,
  'him': 0.0895282036894233,
  'his': 0.10002521923736822,
  'son': 0.03159375449759469}}
```

```
from wefe.metrics import MAC

mac = MAC()
result = mac.run_query(query, model)
result
```

```
{'query_name': 'Female terms and Male terms wrt Family terms and Career terms',
 'result': 0.8416415235615204,
 'mac': 0.8416415235615204,
 'targets_eval': {'Female terms': {'female': {'Family terms': 0.9185737599618733,
    'Career terms': 0.916069650076679},
   'woman': {'Family terms': 0.752434104681015,
    'Career terms': 0.937780514592304 8},
   'girl': {'Family terms': 0.707457959651947,
    'Career terms': 0.9867974997032434},
   'sister': {'Family terms': 0.5973392464220524,
    'Career terms': 0.9482253392925486},
   'she': {'Family terms': 0.7872791914269328,
    'Career terms': 0.9161583095556125},
   'her': {'Family terms': 0.7883057091385126,
    'Career terms': 0.9237247597193345},
   'hers': {'Family terms': 0.7385367527604103,
    'Career terms': 0.9480051446007565},
   'daughter': {'Family terms': 0.5472579970955849,
    'Career terms': 0.9277344475267455}},
  'Male terms': {'male': {'Family terms': 0.8735092766582966,
    'Career terms': 0.9468009045813233},
   'man': {'Family terms': 0.8249392118304968,
    'Career terms': 0.9350165261421353},
   'boy': {'Family terms': 0.7106057899072766,
    'Career terms': 0.9879048476286698},
   'brother': {'Family terms': 0.6280269809067249,
    'Career terms': 0.9477180293761194},
   'he': {'Family terms': 0.8693044614046812,
    'Career terms': 0.8771287016716087},
   'him': {'Family terms': 0.8230192996561527,
    'Career terms': 0.888683641096577},
   'his': {'Family terms': 0.8876195731572807,
    'Career terms': 0.8920885202242061},
```

(continues on next page)

(continued from previous page)

```
   'son': {'Family terms': 0.5764635019004345,
    'Career terms': 0.9220191016211174}}}}
```

13.4.2 2. Fair Embedding Engine

In the case of Fair Embedding Engine, the WE model is passed in the metric instantiation. Then, the output value of the metric is computed using the `compute` method of the metric object.

FEE differs somewhat from the WEFE standardization by making mandatory to provide the model when instantiating each metric, making the metric object model dependent. This makes it difficult to test several models at once since you have to instantiate a different metric object for each model.

On the other hand, FEE does not establish a clear mechanism for passing sets of words of different sizes to the computation method: sets of words are delivered directly with a star parameter *, which defines an arbitrary number of positional arguments. This lack of definition makes it difficult for the user to understand how many and which word sets to pass.

```
from FEE.fee.metrics import WEAT as FEE_WEAT

fee_weat = FEE_WEAT(fee_model)

fee_weat.compute(female_terms, male_terms, family_terms, career_terms)
```

```
0.39821118
```

The FEE implementation of WEAT also allows the calculation of the p-value.

```
fee_weat.compute(female_terms, male_terms, family_terms, career_terms, p_val=True)
```

```
(0.39821118, 0.0)
```

Finally, the implementation of the metric does not support the execution of more complex actions, such as preprocessing word sets. We could not find any other metric that was easily replaceable using the same or a similar interface (with respect to the WEFE standardization layer).

13.4.3 Responsibly

Similar to WEFE, responsibly has a function that takes the model and word sets as input and returns the WEAT score as output.

```
from responsibly.we.weat import calc_single_weat

calc_single_weat(
    twitter_25,
    first_target={"name": "female_terms", "words": female_terms},
    second_target={"name": "male_terms", "words": male_terms},
    first_attribute={"name": "family_terms", "words": family_terms},
    second_attribute={"name": "career_terms", "words": career_terms},
)
```

```
{'Target words': 'female_terms vs. male_terms',
 'Attrib. words': 'family_terms vs. career_terms',
 's': 0.31658393144607544,
 'd': 0.67794365,
 'p': 0.09673659673659674,
 'Nt': '8x2',
 'Na': '8x2'}
```

The p-value can also be obtained from the same function by setting the `with_pvalue` parameter to `True`.

```
calc_single_weat(
    twitter_25,
    first_target={"name": "female_terms", "words": female_terms},
    second_target={"name": "male_terms", "words": male_terms},
    first_attribute={"name": "family_terms", "words": family_terms},
    second_attribute={"name": "career_terms", "words": career_terms},
    with_pvalue=True,
)
```

```
{'Target words': 'female_terms vs. male_terms',
 'Attrib. words': 'family_terms vs. career_terms',
 's': 0.31658393144607544,
 'd': 0.67794365,
 'p': 0.09673659673659674,
 'Nt': '8x2',
 'Na': '8x2'}
```

The implementation of this metric does not include the ability to perform more complex actions such as preprocessing word sets.

In addition, we were unable to find any metrics in this library other than WEAT that are directly comparable to those implemented by WEFE.

13.4.4 EmbeddingBiasScores

EmbeddingBiasScores formalizes how bias is measured in a different way than WEFE: it classifies the methods into clustering or geometric methods (note that WEFE only implements the geometric equivalents).

As part of their standardization, each geometric metric must first define the direction of the bias using the `define_bias_space` function with attribute_embeddings (attribute words) as input; and then use the `group_bias` or `mean_individual_bias` methods to compute the value of the metric.

Examples of use are shown below:

```
# the embeddings to be used must be transformed by hand from words to arrays.
target_embeddings = [
    [model[word] for word in female_terms],
    [model[word] for word in male_terms],
]
attribute_embeddings = [
    [model[word] for word in family_terms],
    [model[word] for word in career_terms],
]
```

```
from EmbeddingBiasScores.geometrical_bias import WEAT

weat = WEAT()
weat.define_bias_space(attribute_embeddings)
# group bias returns the effect size.
weat.group_bias(target_embeddings)
```

```
0.4364516797305417
```

This implementation of WEAT returns the effect size by default. There is no way to parameterize the metric to compute the WEAT score or the p-value.

Similar to WEFE, the standardization implemented by EmbeddingBiasScores allows to easily change the used metric to another with the same input word sets.

```
from EmbeddingBiasScores.geometrical_bias import MAC

mac = MAC()
mac.define_bias_space(attribute_embeddings)

# mac does not accept more than one target set, so we have to calculate it manually.
target_0_mac = mac.mean_individual_bias(target_embeddings[0])
target_1_mac = mac.mean_individual_bias(target_embeddings[1])
(target_0_mac + target_1_mac) / 2
```

```
0.8416415235615204
```

EmbeddingBiasScores includes metrics that WEFE does not yet implement, such as `GeneralizedWEAT` and `SAME`.

```
from EmbeddingBiasScores.geometrical_bias import GeneralizedWEAT

gweat = GeneralizedWEAT()
gweat.define_bias_space(attribute_embeddings)
gweat.group_bias(target_embeddings)
```

```
0.02896493
```

```
from EmbeddingBiasScores.geometrical_bias import SAME

same = SAME()
same.define_bias_space(attribute_embeddings)
same.mean_individual_bias(target_embeddings[0])
```

```
0.2677120929221758
```

Finally, EmbeddingBiasScores does not allow any of its metrics to perform more complex actions, such as preprocessing word set or customizing some performance settings.

13.4.5 Conclusion

In WEFE, having the input words as query objects decoupled from the execution of metrics allows both parameterization of metric execution and easy exchange of one metric for another. In addition, the clean and unified interface for all metrics makes the execution of bias measurements intuitive.

Responsibly and FEE share a similar interface, in which the metric arguments are sets of words (which lack the expressiveness of WEFE queries to declare the number of sets of words supported by each metric), making it difficult to standardize inputs across metrics. We were unable to find any metrics other than WEAT to include in the benchmarking of FEE and Responsibly.

On the other hand, EmbeddingBiasScores also presents its own mathematical standardization for each metric as well as some metrics that WEFE does not yet implement. While the standardization they present may be a bit more specific, it makes it more complex to use.

The increased difficulty is mainly due to two factors: users have to manually define the bias space (using the `define_bias_space` parameter) and then investigate whether to use the parameters `group_bias` or `mean_individual_bias`, which is not clear at first sight unless the basics of the standardization proposed by this library have been previously studied.

Finally, we highlight WEFE's `run_query` method, which allows the user to customize the execution of metrics, such as word preprocessing, normalization of embeddings, and calculation of submetrics or statistical tests.

13.5 5. Ease of Running Bias Mitigation Algorithms

Next we will compare how to run bias mitigation methods on the libraries included in the benchmark. In order to make the comparison as objective as possible, the set of words and the embedding model remain fixed; only the algorithms executed vary. Furthermore, to evaluate the performance of the implemented methods, we will use the same query defined in the previous section using WEAT (female vs. male terms with respect to family vs. career).

```
from wefe.datasets import fetch_debiaswe
from wefe.utils import load_test_model

# word sets to be used
debiaswe_wordsets = fetch_debiaswe()

definitional_pairs = debiaswe_wordsets["definitional_pairs"]
gender_specific = debiaswe_wordsets["gender_specific"]

targets = [
    "executive",
    "management",
    "professional",
    "corporation",
    "salary",
    "office",
    "business",
    "career",
    "home",
    "parents",
    "children",
    "family",
    "cousins",
    "marriage",
```

(continues on next page)

(continued from previous page)

```
    "wedding",
    "relatives",
]
```

13.5.1 1. WEFE

WEFE defines a standardized framework for executing bias mitigation algorithms based on the scikit-learn fit transform interface.

The fit-transform interface allows the user to select the sets of words and parameters that will be used to learn the debiasing transformation (`fit`), as well as to select the words that will be effectively debiased by the method (`transform`).

This allows the user to change the words used to define the bias criterion (which is usually gender, but could be easily changed), as well as the vocabulary word to which the mitigation is applied. This software design pattern is useful for comparing different de-biasing methods, as the user can ensure that the same parameters are used across methods.

Below we show how to execute a mitigation method with WEFE:

```
from wefe.debias.hard_debias import HardDebias
from wefe.debias.hard_debias import HardDebias
from wefe.word_embedding_model import WordEmbeddingModel
from gensim import downloader as api

# load glove model
twitter_25 = api.load("glove-twitter-25")
model = WordEmbeddingModel(twitter_25, "glove twitter dim=25")

# 1. instance Hard Debias algortihm
hd = HardDebias(
    verbose=False,
    criterion_name="gender",
)

# 2. apply fit method and pass the model and definitional pairs.
hd.fit(model, definitional_pairs=definitional_pairs)

# 3. apply transform method passing the model, target and ignore word sets resulting in␣
↪the debiased model
hd_debiased_model = hd.transform(
    model,
    target=targets,
    ignore=gender_specific,
    copy=True,
)
```

```
Copy argument is True. Transform will attempt to create a copy of the original model.␣
↪This may fail due to lack of memory.
Model copy created successfully.
```

```
100%|| 16/16 [00:00<00:00, 21809.84it/s]
```

Next, we show how to change the debiasing method while keeping a very similar parameter configuration.

```
from wefe.debias.repulsion_attraction_neutralization import (
    RepulsionAttractionNeutralization,
)

ran = RepulsionAttractionNeutralization().fit(
    model=model,
    definitional_pairs=definitional_pairs,
)

ran_debiased_model = ran.transform(
    model=model,
    target=targets,
    ignore=gender_specific,
    copy=True,
)
```

```
Copy argument is True. Transform will attempt to create a copyof the original model.␣
↪This may fail due to lack of memory.
Model copy created successfully.
```

```
100%|| 16/16 [00:03<00:00,  5.23it/s]
100%|| 16/16 [00:00<00:00, 45964.98it/s]
```

As can be seen, the fit-transform standardization implemented in WEFE allows to easily execute and exchange the different bias mitigation methods implemented in the library.

```
from wefe.metrics import WEAT

weat = WEAT()
result = weat.run_query(
    query,
    model,
)
print("Original model WEAT evaluation: ", result["weat"])

weat = WEAT()
result = weat.run_query(
    query,
    hd_debiased_model,
)
print("Hard Debias debiased model WEAT evaluation: ", result["weat"])


weat = WEAT()
result = weat.run_query(
    query,
    ran_debiased_model,
)
print(
    "Repulsion Attraction Neutralization debiased model WEAT evaluation: ",
    result["weat"],
)
```

```
Original model WEAT evaluation:  0.31658415612764657
Hard Debias debiased model WEAT evaluation:  0.0023205252364277784
Repulsion Attraction Neutralization debiased model WEAT evaluation:  0.26007230998948216
```

13.5.2 1. Fair Embedding Engine

The Fair Embedding Engine (FEE) requires the embedding model to be passed during instantiation of the algorithm. It currently does not support user-given definitional pairs, as the word sets used are fixed in this implementation, focusing only on gender bias at the moment.

Debiasing is performed by executing the run method. The list of target words to be debiased must be provided in this implementation.

```
import copy
from FEE.fee.embedding.loader import WE

# load model
fee_model = WE().load(ename="glove-twitter-25")
# model must be normalized
fee_model.normalize()
```

```
from FEE.fee.debias import HardDebias

# instance the algortihm and apply it to the embedding model
fee_hd_debiased_model = HardDebias(copy.deepcopy(fee_model)).run(word_list=targets)
```

FEE allows easy use of different debiasing methods with a similar interface

```
from FEE.fee.debias import RANDebias

# instance the algortihm and apply it to the embedding model
ran_hd_debiased_model = RANDebias(copy.deepcopy(fee_model)).run(words=targets)
```

```
# in the case, we generate a custom weat calculation using the fee debiasing methods.
result = WEAT()._calc_weat(
    [fee_model.v(word) for word in query.target_sets[0]],
    [fee_model.v(word) for word in query.target_sets[1]],
    [fee_model.v(word) for word in query.attribute_sets[0]],
    [fee_model.v(word) for word in query.attribute_sets[1]],
)

print("Original model WEAT evaluation: ", result)
result = WEAT()._calc_weat(
    [fee_hd_debiased_model.v(word) for word in query.target_sets[0]],
    [fee_hd_debiased_model.v(word) for word in query.target_sets[1]],
    [fee_hd_debiased_model.v(word) for word in query.attribute_sets[0]],
    [fee_hd_debiased_model.v(word) for word in query.attribute_sets[1]],
)
print("Hard Debias debiased model WEAT evaluation: ", result)
result = WEAT()._calc_weat(
    [ran_hd_debiased_model.v(word) for word in query.target_sets[0]],
    [ran_hd_debiased_model.v(word) for word in query.target_sets[1]],
```

(continues on next page)

(continued from previous page)

```
    [ran_hd_debiased_model.v(word) for word in query.attribute_sets[0]],
    [ran_hd_debiased_model.v(word) for word in query.attribute_sets[1]],
)
print("Repulsion Attraction Neutralization debiased model WEAT evaluation: ", result)
```

```
Original model WEAT evaluation:  0.31658416730351746
Hard Debias debiased model WEAT evaluation:  -0.061893132515251637
Repulsion Attraction Neutralization debiased model WEAT evaluation:  0.17548414319753647
```

13.5.3 1. Responsibly

In Responsibly the embedding model is provided during the instantiation of the `GenderBiasWe` class. Definitional pairs cannot be provided by the user, as the bias being mitigated is set specifically to gender bias. To perform the debiasing process, one simply needs to execute the `debias` method.

However, it should be noted that the mitigation method cannot be run on the benchmark model chosen, as it is not compatible with uncased models such as `twitter-25`.

```
from responsibly.we import GenderBiasWE

# does not work with twitter_25.
gender_bias_we = GenderBiasWE(word2vec)  # instance the GenderBiasWE
gender_bias_we.debias(neutral_words=targets)  # apply the debias
```

13.5.4 4. EmbeddingBiasScore

The library does not implement mitigation methods, so it is not included in this comparison.

13.5.5 Conclusion

All three libraries offer a simple way to apply bias mitigation algorithms in a similar way and all of them are able to mitigate bias in the word embedding model by similar amounts, depending on the metric used.

The main difference between them is that WEFE offers more flexibility to users, allowing them to choose the bias criteria through the words used to learn the transformation and the words that are mitigated. On the other hand, FEE and Responsibly only work with gender bias because the set of words is fixed by default.

Finally, WEFE includes more mitigation algorithms than the other two frameworks.

13.6 6. Metrics and Mitigation Methods Implemented

The following tables provide a comparison of the libraries included in this benchmarking, with respect to the bias metrics and mitigation methods they implement to date.

13.6.1 Fairness Metrics

Metric	WEFE	FEE	Responsibly	EmbeddingBiasScores
WEAT	✓	✓	✓	✓
WEAT ES	✓	×	×	×
RNSB	✓	×	×	×
RIPA	✓	×	×	✓
ECT	✓	×	×	×
RND	✓	×	×	×
MAC	✓	×	×	✓
Direct Bias	×	✓	✓	✓
SAME	×	×	×	✓
Generalized WEAT	×	×	×	✓

The table exclusively focuses on metrics that directly compute from word embeddings (WE) using predefined word sets. As a result, it omits metrics that are not compatible with the wefe interface such as:

- IndirectBias, a metric that accepts as input only two words and the gender direction, previously calculated in a distinct operation.
- GIPE, PMN, and Proximity Bias, which evaluate WE models before and after debiasing with auxiliary mitigation methods.
- SemBias, which is an analogy evaluation dataset.

13.6.2 Mitigation algorithms

Algorithm	WEFE	FEE	Responsibly	EmbeddingBiasScores
Hard Debias	✓	✓	✓	×
Double Hard Debias	✓	×	×	×
Half Sibling Regression	✓	✓	×	×
RAN	✓	✓	×	×
Multiclass HD	✓	×	×	×

13.7 Conclusion

The following table summarizes the main differences between the libraries analyzed in this benchmark study.

	WEFE	FEE	Responsibl y	Embeddi ngBi-asS cores
Implemented Metrics	7	7	3	6
Implemented Mitigation Algorithms	5	3	1	0
Extensible	Easy	Easy	Difficult, not very modular.	Easy
Well-define d interface for metrics	✓	×	×	✓
Well-define d interface for mitigation algorithms	✓	×	×	×
Lastest update	January 2023	October 2020	April 2021	April 2023
Installatio n	Easy: pip or conda	No instructions. It can be installed from the repository	Only with pip. Presents problems	Only from the reposit ory
Documentati on	Extensive documenta tion with examples	Almost no documentation	Limited documentat ion with some examples	No documen tation, only example s.

CHAPTER

FOURTEEN

DIFFERENCES BETWEEN IJCAI VERSION AND CURRENT VERSION

An initial iteration of the present software was constructed to execute the experiments in our previous IJCAI publication titled "WEFE: The word embeddings fairness evaluation framework" authored by Badilla, P., Bravo-Marquez, F., & Pérez, J. (2020) presented at the International Joint Conferences on Artificial Intelligence.

It is pertinent to note that the primary focus of the IJCAI publication was the conceptual framework of evaluating bias rather than the software's development. The main differences between the previous version and the current one are discussed below.

The most noticeable change we can mention with respect to the IJCAI version and the current version is the full implementation of a new debiasing methods module. It includes 5 methods of debiasing: `HardDebias`, `MulticlassHardDebias`, `DoubleHardDebias`, `RepulsionAttractionNeutralization` and `HalfSiblingRegression`.

Regarding metrics: The original version of WEFE published in IJCAI contained 4 metrics: `WEAT`, `WEAT-ES`, `RND` and `RNSB`. Currently and thanks to contributions, WEFE also implements `MAC`, `RIPA` and `ECT`.

Also, the original version contained very rudimentary `Query` and `WordEmbeddingModel` wrapper routines.

In the actual version, the wrappers are much more complete and allow better interaction with the user and with WEFE's internal APIs.

For example, the implementation of `__repr__` for `Query` and `WordEmbeddingModel` contain short descriptions of each object for the user. We have also included a `dict` method in `Query` that allows to transform a query into a dictionary and the `update` in `WordEmbeddingModel` that allows to update an embedding associated to a word by a new one.

The `preprocessing` module has also been improved to cover a wider range of operations (such as different preprocessing steps) that have been modularized and generalized so that any metric or mitigation method can use it.

The documentation has been significantly improved from the original release. These improvements include the addition of new user guides, conceptual guides explaining the theoretical framework, multi-language tutorials, and detailed API documentation covering metrics and mitigation methods, including theoretical details. It is also worth noting that there have been notable improvements in both testing and code quality compared to the original release.

CHAPTER

FIFTEEN

CONTRIBUTING

There are many ways to contribute to the library:

- Implement new metrics.
- Implement new mitigation methods.
- Create more examples and use cases.
- Help improve the documentation.
- Create more tests.

All contributions are welcome!

15.1 Get the repository

You can download the library by running the following command

```
git clone https://github.com/dccuchile/wefe
```

To contribute, simply create a pull request. Verify that your code is well documented, to implement unit tests and follow the black coding style.

15.1.1 Development Requirements

To install the necessary dependencies for the development, testing and compilation of WEFE documentation, run

```
pip install -r requirements-dev.txt
```

15.2 Testing

All unit tests are located in the wefe/tests folder and are based on the `pytest` framework.

To run the tests, execute:

```
pytest tests
```

To check the coverage, run:

```
py.test wefe --cov-report xml:cov.xml --cov wefe
```

And then:

```
coverage report -m
```

15.3 Build the documentation

The documentation is created using sphinx. It can be found in the docs folder at the project's root folder. The documentation includes the API description and some tutorials. To compile the documentation, run the following commands:

```
cd doc
make html
```

Then, you can visit the documentation at `docs/_build/html/index.html`

15.4 How to implement your own metric

The following guide is intended to show how to implement a metric using WEFE. You can find a notebook version of this tutorial at the following link.

15.4.1 Create the class

The first step is to create the class that will contain the metric. This class must extend the `BaseMetric` class.

In the new class you must specify the template (explained below), the name and an abbreviated name or acronym for the metric as class variables.

A **template** is a tuple that defines the cardinality of the target and attribute sets of a query that can be accepted by the metric. It can take integer values, which require that the target or attribute sets have that cardinality or 'n' in case the metric can operate with 1 or more word sets. Note that this will indicate that all queries that do not comply with the template will be rejected when executed using this metric.

Below are some examples of templates:

```
# two target sets and one attribute set required to execute this metric.
template_1 = (2, 1)

# two target sets and two attribute sets required to execute this metric.
template_2 = (2, 2)

# one or more (unlimited) target sets and one attribute set required to execute this␣
↪metric.
template_3 = ('n', 1)
```

Once the template is defined, you can create the metric according to the following code scheme:

```
from wefe.metrics.base_metric import BaseMetric

class ExampleMetric(BaseMetric):
    metric_template = (2, 1)
    metric_name = 'Example Metric'
    metric_short_name = 'EM'
```

15.4.2 Implement `run_query` method

The second step is to implement `run_query` method. This method is in charge of coordinating all the operations to calculate the scores from a `query` and the `word_embedding` model. It must perform 2 basic operations before executing the mathematical calculations:

Validate the parameters

This call checks the main parameters provided to the `run_query` and will raise an exception if it finds a problem with them.

```
# check the types of the provided arguments.
self._check_input(query, model)
```

Transform the Query to Embeddings

This call transforms all the word sets of a query into embeddings.

```
# transform query word sets into embeddings
embeddings = get_embeddings_from_query(
    model=model,
    query=query,
    lost_vocabulary_threshold=lost_vocabulary_threshold,
    preprocessors=preprocessors,
    strategy=strategy,
    normalize=normalize,
    warn_not_found_words=warn_not_found_words,
)
```

This step could return either:

- `None` if any of the sets lost percentage more words than the number of words allowed by `lost_vocabulary_threshold` parameter (specified as percentage float). In this case the metric would be expected to return nan in its results.

```
# if there is any/some set has less words than the allowed limit,
# return the default value (nan)
if embeddings is None:
    return {
        "query_name": query.query_name,
        "result": np.nan,
        "metrica_default_value": np.nan,
    }
```

- A tuple otherwise. This tuple contains two values:
  - A dictionary that maps each target set name to a dictionary containing its words and embeddings.
  - A dictionary that maps each attribute set name to a dictionary containing its words and embeddings.

We can illustrate what the outputs of the previous transformation look like using the following query:

```
from wefe.word_embedding_model import WordEmbeddingModel
from wefe.query import Query
from wefe.utils import load_test_model # a few embeddings of WEAT experiments
from wefe.datasets.datasets import load_weat # the word sets of WEAT experiments
from wefe.preprocessing import get_embeddings_from_query


weat = load_weat()
model = load_test_model()

flowers = weat['flowers']
weapons = weat['weapons']
pleasant = weat['pleasant_5']
query = Query([flowers, weapons], [pleasant],
              ['Flowers', 'Weapons'], ['Pleasant'])

embeddings = get_embeddings_from_query(
    model=model,
    query=query,
    # other params...
)
target_sets, attribute_sets = embeddings
```

If you inspect `target_sets`, it would look like the following dictionary:

```
{
    'Flowers': {
        'aster': array([-0.22167969, 0.52734375, 0.01745605, ...], dtype=float32),
        'clover': array([-0.03442383, 0.19042969, -0.17089844, ...], dtype=float32),
        'hyacinth': array([-0.01391602, 0.3828125, -0.21679688, ...], dtype=float32),
        ...
    },
    'Weapons': {
        'arrow': array([0.18164062, 0.125, -0.12792969. ...], dtype=float32),
        'club': array([-0.04907227, -0.07421875, -0.0390625, ...], dtype=float32),
        'gun': array([0.05566406, 0.15039062, 0.33398438, ...], dtype=float32),
        'missile': array([4.7874451e-04, 5.1953125e-01, -1.3809204e-03, ...],␣
↪dtype=float32),
        ...
    }
}
```

And `attribute_sets` would look like:

```
{
    'Pleasant': {
        'caress': array([0.2578125, -0.22167969, 0.11669922], dtype=float32),
        'freedom': array([0.26757812, -0.078125, 0.09326172], dtype=float32),
        'health': array([-0.07421875, 0.11279297, 0.09472656], dtype=float32),
        ...
    }
}
```

The idea of keeping a mapping between set names, words and their embeddings is that there are some metrics that can

calculate sub-metrics at different levels and that can be useful for further use.

Example Metric

Using the steps previously seen, a sample metric is implemented:

```
from typing import Any, Dict, Union, List, Callable

import numpy as np

from wefe.metrics.base_metric import BaseMetric
from wefe.query import Query
from wefe.word_embedding_model import WordEmbeddingModel


class ExampleMetric(BaseMetric):

    # replace with the parameters of your metric
    metric_template = (2, 1) # cardinalities of the targets and attributes sets that␣
↪your metric will accept.
    metric_name = 'Example Metric'
    metric_short_name = 'EM'

    def run_query(
        self,
        query: Query,
        model: WordEmbeddingModel,
        lost_vocabulary_threshold: float = 0.2,
        preprocessors: List[Dict[str, Union[str, bool, Callable]]] = [{}],
        strategy: str = "first",
        normalize: bool = False,
        warn_not_found_words: bool = False,
        *args: Any,
        **kwargs: Any,
    ) -> Dict[str, Any]:
        """Calculate the Example Metric metric over the provided parameters.

        Parameters
        ----------
        query : Query
            A Query object that contains the target and attribute word sets to
            be tested.

        word_embedding : WordEmbeddingModel
            A WordEmbeddingModel object that contains certain word embedding
            pretrained model.

        lost_vocabulary_threshold : float, optional
            Specifies the proportional limit of words that any set of the query is
            allowed to lose when transforming its words into embeddings.
            In the case that any set of the query loses proportionally more words
            than this limit, the result values will be np.nan, by default 0.2
```

(continues on next page)

(continued from previous page)

```
        preprocessors : List[Dict[str, Union[str, bool, Callable]]]
            A list with preprocessor options.

            A ``preprocessor`` is a dictionary that specifies what processing(s) are
            performed on each word before it is looked up in the model vocabulary.
            For example, the ``preprocessor``
            ``{'lowecase': True, 'strip_accents': True}`` allows you to lowercase
            and remove the accent from each word before searching for them in the
            model vocabulary. Note that an empty dictionary ``{}`` indicates that no
            preprocessing is done.

            The possible options for a preprocessor are:

            *   ``lowercase``: ``bool``. Indicates that the words are transformed to
                lowercase.
            *   ``uppercase``: ``bool``. Indicates that the words are transformed to
                uppercase.
            *   ``titlecase``: ``bool``. Indicates that the words are transformed to
                titlecase.
            *   ``strip_accents``: ``bool``, ``{'ascii', 'unicode'}``: Specifies that
                the accents of the words are eliminated. The stripping type can be
                specified. True uses 'unicode' by default.
            *   ``preprocessor``: ``Callable``. It receives a function that operates
                on each word. In the case of specifying a function, it overrides the
                default preprocessor (i.e., the previous options stop working).

            A list of preprocessor options allows searching for several
            variants of the words into the model. For example, the preprocessors
            ``[{}, {"lowercase": True, "strip_accents": True}]``
            ``{}`` allows searching first for the original words in the vocabulary of
            the model. In case some of them are not found,
            ``{"lowercase": True, "strip_accents": True}`` is executed on these words
            and then they are searched in the model vocabulary.

        strategy : str, optional
            The strategy indicates how it will use the preprocessed words: 'first' will
            include only the first transformed word found. 'all' will include all
            transformed words found, by default "first".

        normalize : bool, optional
            True indicates that embeddings will be normalized, by default False

        warn_not_found_words : bool, optional
            Specifies if the function will warn (in the logger)
            the words that were not found in the model's vocabulary
            , by default False.

        Returns
        -------
        Dict[str, Any]
            A dictionary with the query name, the resulting score of the metric,
            and other scores.
```

(continues on next page)

(continued from previous page)

```
        """
        # check the types of the provided arguments (only the defaults).
        self._check_input(query, model)

        # transform query word sets into embeddings
        embeddings = get_embeddings_from_query(
            model=model,
            query=query,
            lost_vocabulary_threshold=lost_vocabulary_threshold,
            preprocessors=preprocessors,
            strategy=strategy,
            normalize=normalize,
            warn_not_found_words=warn_not_found_words,
        )

        # if there is any/some set has less words than the allowed limit,
        # return the default value (nan)
        if embeddings is None:
            return {
                'query_name': query.query_name, # the name of the evaluated query
                'result': np.nan, # the result of the metric
                'em': np.nan, # result of the calculated metric (recommended)
                'other_metric' : np.nan, # another metric calculated (optional)
                'results_by_word' : np.nan, # if available, values by word (optional)
                # ...
            }

        # get the targets and attribute sets transformed into embeddings.
        target_sets, attribute_sets = embeddings

        # commonly, you only will need the embeddings of the sets.
        # this can be obtained by using:
        target_embeddings = list(target_sets.values())
        attribute_embeddings = list(attribute_sets.values())


        """
        # From here, the code can vary quite a bit depending on what you need.
        # It is recommended to calculate the metric operations in another method(s).
        results = calc_metric()

        # The final step is to return query and result.
        # You can return other scores, metrics by word or metrics by set, etc.
        return {
                'query_name': query.query_name, # the name of the evaluated query
                'result': results.metric, # the result of the metric
                'em': results.metric # result of the calculated metric (recommended)
                'other_metric' : results.other_metric # Another metric calculated␣
↪(optional)
                'another_results' : results.details_by_set # if available, values by word␣
↪(optional),
                ...
```

(continues on next page)

(continued from previous page)

```
            }
        """
```

15.4.3 Implement the logic of the metric

Suppose we want to implement an extremely simple three-step metric, where:

1. We calculate the average of all the sets,
2. Then, calculate the cosine distance between the target set averages and the attribute average.
3. Subtract these distances.

To do this, we create a new method :code:`_calc_metric` in which, using the array of embedding dict objects as input, we will implement the above.

```
from typing import Any, Dict, Union, List, Callable

from scipy.spatial import distance
import numpy as np

from wefe.metrics import BaseMetric
from wefe.query import Query
from wefe.word_embedding_model import WordEmbeddingModel
from wefe.preprocessing import get_embeddings_from_query

class ExampleMetric(BaseMetric):

    # replace with the parameters of your metric
    metric_template = (
        2, 1
    )  # cardinalities of the targets and attributes sets that your metric will accept.
    metric_name = 'Example Metric'
    metric_short_name = 'EM'

    def _calc_metric(self, target_embeddings, attribute_embeddings):
        """Calculates the metric.

         Parameters
         ----------
         target_embeddings : np.array
             An array with dicts. Each dict represents an target set.
             A dict is composed with a word and its embedding as key, value respectively.
         attribute_embeddings : np.array
             An array with dicts. Each dict represents an attribute set.
             A dict is composed with a word and its embedding as key, value respectively.

         Returns
         -------
         np.float
             The value of the calculated metric.
         """
```

(continues on next page)

(continued from previous page)

```
        # get the embeddings from the dicts
        target_embeddings_0 = np.array(list(target_embeddings[0].values()))
        target_embeddings_1 = np.array(list(target_embeddings[1].values()))

        attribute_embeddings_0 = np.array(
            list(attribute_embeddings[0].values()))

        # calculate the average embedding by target and attribute set.
        target_embeddings_0_avg = np.mean(target_embeddings_0, axis=0)
        target_embeddings_1_avg = np.mean(target_embeddings_1, axis=0)
        attribute_embeddings_0_avg = np.mean(attribute_embeddings_0, axis=0)

        # calculate the distances between the target sets and the attribute set
        dist_target_0_attr = distance.cosine(target_embeddings_0_avg,
                                             attribute_embeddings_0_avg)
        dist_target_1_attr = distance.cosine(target_embeddings_1_avg,
                                             attribute_embeddings_0_avg)

        # subtract the distances
        metric_result = dist_target_0_attr - dist_target_1_attr
        return metric_result

    def run_query(
        self,
        query: Query,
        model: WordEmbeddingModel,
        lost_vocabulary_threshold: float = 0.2,
        preprocessors: List[Dict[str, Union[str, bool, Callable]]] = [{}],
        strategy: str = "first",
        normalize: bool = False,
        warn_not_found_words: bool = False,
        *args: Any,
        **kwargs: Any,
    ) -> Dict[str, Any]:
        """Calculate the Example Metric metric over the provided parameters.

        Parameters
        ----------
        query : Query
            A Query object that contains the target and attribute word sets to
            be tested.

        word_embedding : WordEmbeddingModel
            A WordEmbeddingModel object that contains certain word embedding
            pretrained model.

        lost_vocabulary_threshold : float, optional
            Specifies the proportional limit of words that any set of the query is
            allowed to lose when transforming its words into embeddings.
            In the case that any set of the query loses proportionally more words
            than this limit, the result values will be np.nan, by default 0.2
```

(continues on next page)

(continued from previous page)

```
        preprocessors : List[Dict[str, Union[str, bool, Callable]]]
            A list with preprocessor options.

            A ``preprocessor`` is a dictionary that specifies what processing(s) are
            performed on each word before its looked up in the model vocabulary.
            For example, the ``preprocessor``
            ``{'lowecase': True, 'strip_accents': True}`` allows you to lowercase
            and remove the accent from each word before searching for them in the
            model vocabulary. Note that an empty dictionary ``{}`` indicates that no
            preprocessing is done.

            The possible options for a preprocessor are:

            *   ``lowercase``: ``bool``. Indicates that the words are transformed to
                lowercase.
            *   ``uppercase``: ``bool``. Indicates that the words are transformed to
                uppercase.
            *   ``titlecase``: ``bool``. Indicates that the words are transformed to
                titlecase.
            *   ``strip_accents``: ``bool``, ``{'ascii', 'unicode'}``: Specifies that
                the accents of the words are eliminated. The stripping type can be
                specified. True uses 'unicode' by default.
            *   ``preprocessor``: ``Callable``. It receives a function that operates
                on each word. In the case of specifying a function, it overrides the
                default preprocessor (i.e., the previous options stop working).

            A list of preprocessor options allows searching for several
            variants of the words into the model. For example, the preprocessors
            ``[{}, {"lowercase": True, "strip_accents": True}]``
            ``{}`` allows searching first for the original words in the vocabulary of the␣
↪model.
            In case some of them are not found, ``{"lowercase": True, "strip_accents":␣
↪True}``
            is executed on these words and then they are searched in the model␣
↪vocabulary.

        strategy : str, optional
            The strategy indicates how it will use the preprocessed words: 'first' will
            include only the first transformed word found. 'all' will include all
            transformed words found, by default "first".

        normalize : bool, optional
            True indicates that embeddings will be normalized, by default False

        warn_not_found_words : bool, optional
            Specifies if the function will warn (in the logger)
            the words that were not found in the model's vocabulary
            , by default False.

        Returns
        -------
        Dict[str, Any]
```

(continues on next page)

(continued from previous page)

```
            A dictionary with the query name, the resulting score of the metric,
            and other scores.
        """
        # check the types of the provided arguments (only the defaults).
        self._check_input(query, model)

        # transform query word sets into embeddings
        embeddings = get_embeddings_from_query(
            model=model,
            query=query,
            lost_vocabulary_threshold=lost_vocabulary_threshold,
            preprocessors=preprocessors,
            strategy=strategy,
            normalize=normalize,
            warn_not_found_words=warn_not_found_words,
        )

        # if there is any/some set has less words than the allowed limit,
        # return the default value (nan)
        if embeddings is None:
            return {
                'query_name': query.query_name, # the name of the evaluated query
                'result': np.nan, # the result of the metric
                'em': np.nan, # result of the calculated metric (recommended)
                'other_metric' : np.nan, # another metric calculated (optional)
                'results_by_word' : np.nan, # if available, values by word (optional)
                # ...
            }

        # get the targets and attribute sets transformed into embeddings.
        target_sets, attribute_sets = embeddings

        # commonly, you only will need the embeddings of the sets.
        # this can be obtained by using:
        target_embeddings = list(target_sets.values())
        attribute_embeddings = list(attribute_sets.values())

        result = self._calc_metric(target_embeddings, attribute_embeddings)

        # return the results.
        return {"query_name": query.query_name, "result": result, 'em': result}
```

Now, let us try it out:

```
from wefe.query import Query
from wefe.utils import load_weat_w2v  # a few embeddings of WEAT experiments
from wefe.datasets.datasets import load_weat  # the word sets of WEAT experiments

weat = load_weat()
model = WordEmbeddingModel(load_weat_w2v(), 'weat_w2v', '')

flowers = weat['flowers']
```

(continues on next page)

(continued from previous page)

```
weapons = weat['weapons']
pleasant = weat['pleasant_5']
query = Query([flowers, weapons], [pleasant], ['Flowers', 'Weapons'],
                ['Pleasant'])


results = ExampleMetric().run_query(query, model)
print(results)
```

```
{'query_name': 'Flowers and Weapons wrt Pleasant', 'result': -0.10210171341896057, 'em':␣
↪-0.10210171341896057}
```

We have completely defined a new metric. Congratulations!

Note

Some comments regarding the implementation of new metrics:

- Note that the returned object must necessarily be a `dict` instance containing the `result` and `query_name` key-values. Otherwise you will not be able to run query batches using utility functions like `run_queries`.
- `run_query` can receive additional parameters. Simply add them to the function signature. These parameters can also be used when running the metric from the `run_queries` utility function.
- We recommend implementing the logic of the metric separated from the `run_query` function. In other words, implement the logic in a `calc_your_metric` function that receives the dictionaries with the necessary embeddings and parameters.
- The file where `ExampleMetric` is located can be found inside the distances folder of the repository.

15.5 Mitigation Method Implementation Guide

The main idea when implementing a mitigation method is that it has to follow the logic of the transformations in scikit-learn. That is, you must separate the logic of the calculation of the mitigation transformation (*fit*) with the application of the transformation on the model (*transform*).

In practical terms, every WEFE transformation must extend the *BaseDebias* class. *BaseDebias* has two abstract methods that must be implemented: *fit* and *transform*.

15.5.1 Fit

fit is the method in charge of calculating the bias mitigation transformation that will be subsequently applied to the model. *BaseDebias* implements it as an abstract method that requires only one argument: *model*, which expects a *WordEmbeddingModel* instance.

```
@abstractmethod
def fit(
    self,
    model: WordEmbeddingModel,
    **fit_params,
) -> "BaseDebias":
    """Fit the transformation.

```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    model : WordEmbeddingModel
        The word embedding model to debias.
    """
    raise NotImplementedError()
```

The idea of requesting model at this point is that the calculation of the transformation commonly requires some words from the model vocabulary.

As each bias mitigation method is different, it is expected that these can receive more parameters than those listed above. In, *HardDebias*, *fit* is defined using the default parameter *model* plus *definitional_pairs* and *equalize_pairs*, which are specific to *HardDebias*:

```
def fit(
    self,
    model: WordEmbeddingModel,
    definitional_pairs: Sequence[Sequence[str]],
    equalize_pairs: Optional[Sequence[Sequence[str]]] = None,
    **fit_params,
) -> BaseDebias:
    """Compute the bias direction and obtains the equalize embedding pairs.

    Parameters
    ----------
    model : WordEmbeddingModel
        The word embedding model to debias.
    definitional_pairs : Sequence[Sequence[str]]
        A sequence of string pairs that will be used to define the bias direction.
        For example, for the case of gender debias, this list could be [['woman',
        'man'], ['girl', 'boy'], ['she', 'he'], ['mother', 'father'], ...].
    equalize_pairs : Optional[Sequence[Sequence[str]]], optional
        A list with pairs of strings which will be equalized.
        In the case of passing None, the equalization will be done over the word
        pairs passed in definitional_pairs,
        by default None.
    criterion_name : Optional[str], optional
        The name of the criterion for which the debias is being executed,
        e.g. 'Gender'. This will indicate the name of the model returning transform,
        by default None

    Returns
    -------
    BaseDebias
        The debias method fitted.
    """
    self._check_sets_size(definitional_pairs, "definitional")
    self.definitional_pairs_ = definitional_pairs

    # -------------------------------------------------------------------------
    # Obtain the embedding of each definitional pairs.
    if self.verbose:
        print("Obtaining definitional pairs.")
```

(continues on next page)

(continued from previous page)

```
    self.definitional_pairs_embeddings_ = get_embeddings_from_sets(
        model=model,
        sets=definitional_pairs,
        sets_name="definitional",
        warn_lost_sets=self.verbose,
        normalize=True,
        verbose=self.verbose,
    )

    # ------------------------------------------------------------------------:
    # Identify the bias subspace using the definning pairs.
    if self.verbose:
        print("Identifying the bias subspace.")

    self.pca_ = self._identify_bias_subspace(
        self.definitional_pairs_embeddings_, self.verbose,
    )
    self.bias_direction_ = self.pca_.components_[0]
    # code was cut for simplicity.
    # you can visit the missing code in the file debias/HardDebias
    ...
    return self
```

Note: Note that *get_embeddings_from_sets* is used to transform word sets to embeddings sets. This function, as well as the one to transform queries to embeddings, are available in the *preprocessing* module.

Once *fit* has calculated the transformation, the method should return *self*.

15.5.2 Transform

This method is intended to implement the application of the transformation calculated in *fit* on the embedding model. It must always receive the same 4 arguments:

- *model*: The model on which the transformation will be applied
- *target*: A set of words or None. If it is specified, the debias method will be performed only on the word embeddings of this set. If *None* is provided, the debias will be performed on all words (except those specified in ignore). by default *None*.
- *ignore*: A set of words or None. If target is *None* and a set of words is specified
- in ignore, the debias method will perform the debias in all words except those
- specified in this set, by default *None*.
- *copy*: If *True*, the debias will be performed on a copy of the model. If *False*, the debias will be applied on the same model delivered, causing its vectors to mutate.

```
@abstractmethod
def transform(
    self,
    model: WordEmbeddingModel,
    target: Optional[List[str]] = None,
```

(continues on next page)

(continued from previous page)

```
    ignore: Optional[List[str]] = None,
    copy: bool = True,
) -> WordEmbeddingModel:
    """Perform the debiasing method over the model provided.

    Parameters
    ----------
    model : WordEmbeddingModel
        The word embedding model to debias.
    target : Optional[List[str]], optional
        If a set of words is specified in target, the debias method will be performed
        only on the word embeddings of this set. If `None` is provided, the
        debias will be performed on all words (except those specified in ignore).
        by default `None`.
    ignore : Optional[List[str]], optional
        If target is `None` and a set of words is specified in ignore, the debias
        method will perform the debias in all words except those specified in this
        set, by default `None`.
    copy : bool, optional
        If `True`, the debias will be performed on a copy of the model.
        If `False`, the debias will be applied on the same model delivered, causing
        its vectors to mutate.
        **WARNING:** Setting copy with `True` requires at least 2x RAM of the size
        of the model. Otherwise the execution of the debias may raise
        `MemoryError`, by default True.

    Returns
    -------
    WordEmbeddingModel
        The debiased word embedding model.
    """
    raise NotImplementedError()
```

As can be seen, the embeddings that will be modified by the transformation are determined by the words delivered in the *target* and *ignore* sets or the absence of both (apply on all words). The idea is that this convention is maintained during the creation of a new debias method.

Some useful initial checks and operations for this method:

- The arguments can be checked through the *_check_transform_args BaseDebias* method.
- You can also check whether the method is trained or not using the *check_is_fitted* method. This is a wrapper of the original scikit-learn that can be imported from the utils module.
- In case *copy* argument is *True*, you must duplicate the model and work on the replica. It is recommended to use *deepcopy* of the *copy* module for such purposes.

The following code segment (obtained from *HardDebias*) shows an example of how to execute the points mentioned above:

```
def transform(
    self,
    model: WordEmbeddingModel,
    target: Optional[List[str]] = None,
    ignore: Optional[List[str]] = None,
```

(continues on next page)

(continued from previous page)

```
    copy: bool = True,
    ) -> WordEmbeddingModel:
    """Execute hard debias over the provided model.

    Parameters
    ----------
    model : WordEmbeddingModel
        The word embedding model to debias.
    target : Optional[List[str]], optional
        If a set of words is specified in target, the debias method will be performed
        only on the word embeddings of this set. If `None` is provided, the
        debias will be performed on all words (except those specified in ignore).
        by default `None`.
    ignore : Optional[List[str]], optional
        If target is `None` and a set of words is specified in ignore, the debias
        method will perform the debias in all words except those specified in this
        set, by default `None`.
    copy : bool, optional
        If `True`, the debias will be performed on a copy of the model.
        If `False`, the debias will be applied on the same model delivered, causing
        its vectors to mutate.
        **WARNING:** Setting copy with `True` requires RAM at least 2x of the size
        of the model, otherwise the execution of the debias may raise to
        `MemoryError`, by default True.

    Returns
    -------
    WordEmbeddingModel
        The debiased embedding model.
    """
    # ------------------------------------------------------------------------------
    # Check types and if the method is fitted

    self._check_transform_args(
        model=model, target=target, ignore=ignore, copy=copy,
    )

    # check if the following attributes exist in the object.
    check_is_fitted(
        self,
        [
            "definitional_pairs_",
            "definitional_pairs_embeddings_",
            "pca_",
            "bias_direction_",
        ],
    )

    # Copy
    if copy:
        print(
            "Copy argument is True. Transform will attempt to create a copy "
```

(continues on next page)

(continued from previous page)

```
            "of the original model. This may fail due to lack of memory."
        )
        model = deepcopy(model)
        print("Model copy created successfully.")

    else:
        print(
            "copy argument is False. The execution of this method will mutate "
            "the original model."
        )
```

Unfortunately it is impossible to cover much more without losing generality. However, we recommend checking the code structure shown in *HardDebias* or *MulticlassHardDebias* classes to guide you through the process of implementing a new mitigation method. You can also open an issue in the repository to comment on any questions you may have in the implementation.

CHAPTER

SIXTEEN

REPOSITORY

You can find the project repository at the following link: WEFE repository on Github.

INDEX

Symbols

B

C

D

E

F

G

H

L

M

N

P

Q

R

T

U

W