WEFE Documentation

Release 0.3.0

Pablo Badilla

Oct 06, 2021

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 for measuring an mitigating bias in word embedding models. It generalizes many existing fairness metrics into a unified framework and provides a standard interface for:

- Encapsulating existing fairness metrics from previous work and designing new ones.
- 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.

WEFE also standardizes the process of mitigating bias through an interface similar to the `scikit-learn` `fit-transform`. This standardization separates the mitigation process into two stages:

- The logic of calculating the transformation to be performed on the model (`fit`).
- The execution of the mitigation transformation on the model (`transform`).

1.1 Motivation and objectives

Word Embeddings 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 Measurement Framework

Here we present the main building blocks of the measuring framework and then, we present the common usage pattern of WEFE.

1.3.1 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.

1.3.2 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'*.

1.3.3 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.

1.3.4 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 $|\mathcal{A}| = a$.

1.3.5 Fairness Measure

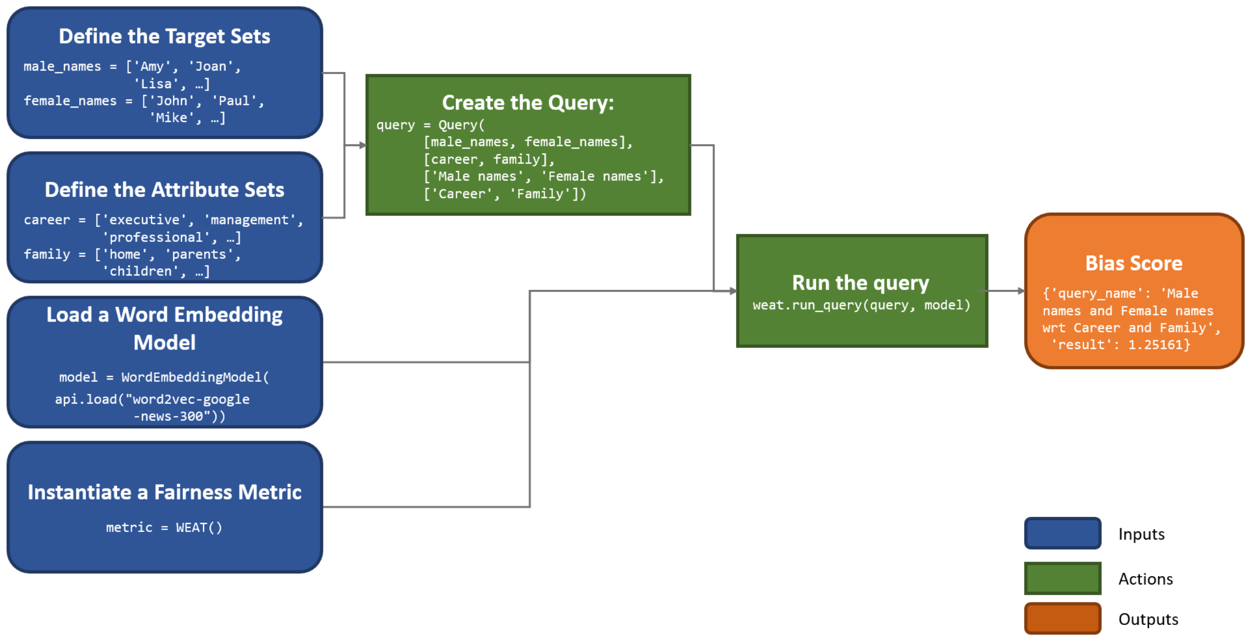
A fairness metric is a function that quantifies the degree of association between target and attribute words in a word embedding model. In our framework, every fairness metric is defined as a function that has a query and a model as input, and produces a real number as output.

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.

1.3.6 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.



```
male_names = ['Amy', 'Joan',
              'Lisa', …]
female_names = ['John', 'Paul',
                'Mike', …]
```

```
career = ['executive', 'management',
          'professional', …]
family = ['home', 'parents',
          'children', …]
```

```
model = WordEmbeddingModel(
api.load("word2vec-google
          -news-300"))
```

```
metric = WEAT()
```

```
query = Query(
      [male_names, female_names],
      [career, family],
      ['Male names', 'Female names'],
      ['Career', 'Family'])
```

```
weat.run_query(query, model)
```

```
{'query_name': 'Male
names and Female names
wrt Career and Family',
 'result': 1.25161}
```

To see the implementation of this query using WEFE, refer to the Quick start section.

1.4 Metrics

The metrics implemented in the package so far are:

1.4.1 WEAT

Word Embedding Association Test (WEAT), presented in the paper "*Semantics derived automatically from language corpora contain human-like biases*". This metric receives two sets T_1 and T_2 of target words, and two sets A_1 and A_2 of attribute words. Its objective is to quantify the strength of association of both pairs of sets through a permutation test. It also contains a variant, WEAT Effect Size. This variant represents a normalized measure that quantifies how far apart the two distributions of association between targets and attributes are.

1.4.2 RND

Relative Norm Distance (RND), presented in the paper "*Word embeddings quantify 100 years of gender and ethnic stereotypes*". RND averages the embeddings of each target set, then for each of the attribute words, calculates the norm of the difference between the word and the average target, and then subtracts the norms. The more positive (negative) the relative distance from the norm, the more associated are the sets of attributes towards group two (one).

1.4.3 RNSB

Relative Negative Sentiment Bias (RNSB), presented in the paper "*A transparent framework for evaluating unintended demographic bias in word embeddings*".

RNSB receives as input queries with two attribute sets A_1 and A_2 and two or more target sets, and thus has a template of the form $s = (N, 2)$ with $N \geq 2$. Given a query $Q = (\{T_1, T_2, \ldots, T_n\}, \{A_1, A_2\})$ and an embedding model $\mathbf{M}$, in order to compute the metric $F_{\text{RNSB}}(\mathbf{M}, Q)$ one 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. 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). Now, we construct 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 $\sum_w P(w) = 1$. The main idea behind RNSB is that the more that $P(\cdot)$ resembles a uniform distribution, the less biased the word embedding model is.

1.4.4 MAC

Mean Average Cosine Similarity (MAC), presented in the paper "*Black is to Criminal as Caucasian is to Police: Detecting and Removing Multiclass Bias in Word Embeddings*".

1.4.5 ECT

The Embedding Coherence Test, presented in "Attenuating Bias in Word vectors" 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.

1.4.6 RIPA

The Relational Inner Product Association, presented in the paper "Understanding Undesirable Word Embedding Associations", calculates bias by measuring the bias of a term by using the relation vector (i.e the first principal component of a pair of words that define the association) and calculating the dot product of this vector with the attribute word vector. RIPA's advantages are its interpretability, and its relative robustness compared to WEAT with regard to how the relation vector is defined.

1.5 Relevant Papers

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.

1.5.1 Measurements and Case Studies

- Caliskan, A., Bryson, J. J., & Narayanan, A. (2017). Semantics derived automatically from language corpora contain human-like biases. Science, 356(6334), 183-186..
- Garg, N., Schiebinger, L., Jurafsky, D., & Zou, J. (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes. Proceedings of the National Academy of Sciences, 115(16), E3635-E3644..
- Sweeney, C., & Najafian, M. (2019, July). A Transparent Framework for Evaluating Unintended Demographic Bias in Word Embeddings. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (pp. 1662-1667)..
- Dev, S., & Phillips, J. (2019, April). Attenuating Bias in Word vectors. In Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (pp. 879-887)..
- Ethayarajh, K., & Duvenaud, D., & Hirst, G. (2019, July). Understanding Undesirable Word Embedding Associations. Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (pp. 1696-1705)..

1.5.2 Bias Mitigation

- Bolukbasi, T., Chang, K. W., Zou, J., Saligrama, V., & Kalai, A. (2016). Quantifying and reducing stereotypes in word embeddings. arXiv preprint arXiv:1606.06121.
- 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. In Advances in neural information processing systems (pp. 4349-4357).
- Zhao, J., Zhou, Y., Li, Z., Wang, W., & Chang, K. W. (2018). Learning gender-neutral word embeddings. arXiv preprint arXiv:1809.01496.
- Zhao, J., Wang, T., Yatskar, M., Ordonez, V., & Chang, K. W. (2017). Men also like shopping: Reducing gender bias amplification using corpus-level constraints. arXiv preprint arXiv:1707.09457.
- 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 but do not remove them. arXiv preprint arXiv:1903.03862.

1.5.3 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. arXiv preprint arXiv:1908.09635.
- 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 meaning. Journal of Artificial Intelligence Research, 63, 743-788.

Bias in Contextualized Word Embeddings

- Zhao, J., Wang, T., Yatskar, M., Cotterell, R., Ordonez, V., & Chang, K. W. (2019). Gender bias in contextualized word embeddings. arXiv preprint arXiv:1904.03310.
- Basta, C., Costa-jussà, M. R., & Casas, N. (2019). Evaluating the underlying gender bias in contextualized word embeddings. arXiv preprint arXiv:1904.08783.
- Kurita, K., Vyas, N., Pareek, A., Black, A. W., & Tsvetkov, Y. (2019). Measuring bias in contextualized word representations. arXiv preprint arXiv:1906.07337.
- Tan, Y. C., & Celis, L. E. (2019). Assessing social and intersectional biases in contextualized word representations. In Advances in Neural Information Processing Systems (pp. 13209-13220).
- Stereoset: A Measure of Bias in Language Models

1.6 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 International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence (IJCAI-PRICAI 2020), Yokohama, Japan.

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},
    }
```

1.7 Roadmap

We expect in the future to:

- Implement the metrics that have come out in recent works on bias in embeddings.
- 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 visualization module.
- Implement p-values with statistic resampling to all metrics.

1.8 License

1.9 Team

- Pablo Badilla.
- Felipe Bravo-Marquez.
- Jorge Pérez.

1.9.1 Contributors

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

Thank you very much !

1.9.2 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.

1.10 Acknowledgments

This work was funded by the Millennium Institute for Foundational Research on Data (IMFD).

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:

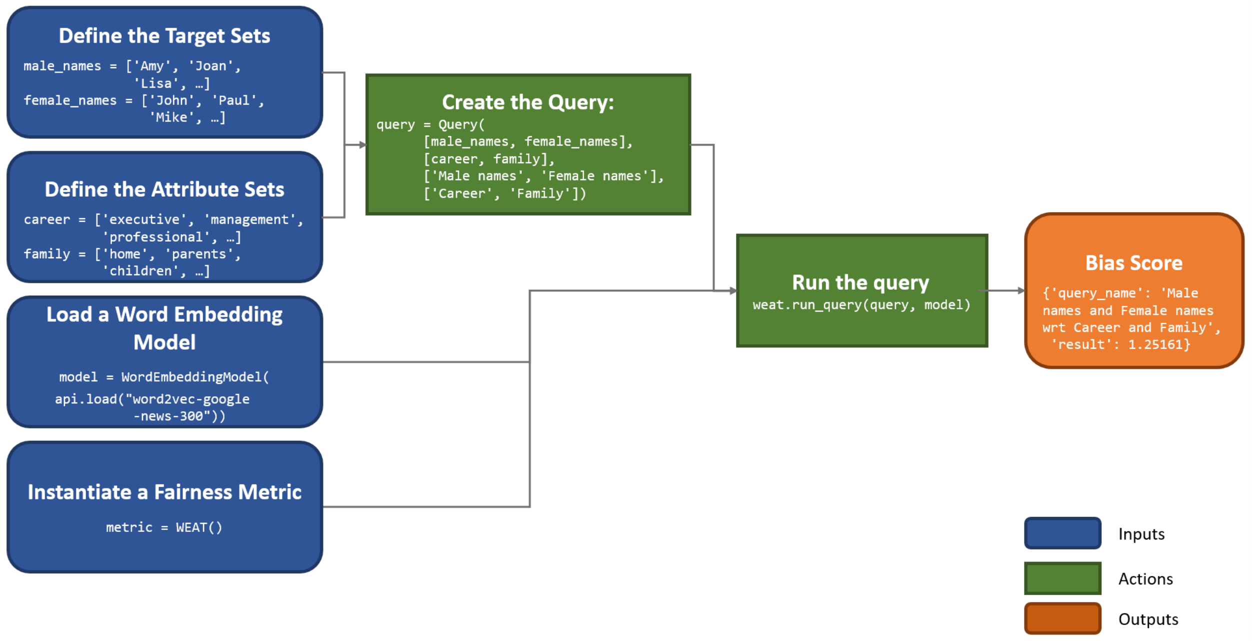
```
conda install -c pbadilla wefe
```

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:



```
male_names = ['Amy', 'Joan',
              'Lisa', …]
female_names = ['John', 'Paul',
                'Mike', …]
```

```
career = ['executive', 'management',
          'professional', …]
family = ['home', 'parents',
          'children', …]
```

```
model = WordEmbeddingModel(
api.load("word2vec-google
          -news-300"))
```

```
metric = WEAT()
```

```
query = Query(
      [male_names, female_names],
      [career, family],
      ['Male names', 'Female names'],
      ['Career', 'Family'])
```

```
weat.run_query(query, model)
```

```
{'query_name': 'Male
names and Female names
wrt Career and Family',
 'result': 1.25161}
```

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 User Guide section.

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 the framework section on the library's about page. These concepts are widely used in the following sections.

A jupyter notebook with this code is located in the following link: WEFE User Guide.

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. WEAT was originally implemented in *Semantics derived automatically from language corpora contain human-like biases* paper.

3.1.1 Load a word embeddings model as a `WordEmbedding` 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.

```python
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.

```python
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",
            "salary",
            "office",
            "business",
```

(continues on next page)

(continued from previous page)

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

print(gender_query)
```

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

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)
print(result)
```

```
{'query_name': 'Female terms and Male Terms wrt Family and Careers',
 'result': 0.3165843551978469,
 'weat': 0.3165843551978469,
 'effect_size': 0.6779444653930398,
 '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 API documentation.

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.

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)
print(result)
```

```
{'query_name': 'Female terms and Male Terms wrt Family and Careers',
 'result': 0.6779444653930398,
 'weat': 0.3165843551978469,
 'effect_size': 0.6779444653930398,
 '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=15000
)
print(result)
```

```
{'query_name': 'Female terms and Male Terms wrt Family and Careers',
 'result': 0.3165843551978469,
 'weat': 0.3165843551978469,
 'effect_size': 0.6779444653930398,
 'p_value': 0.09032731151256583}
```

3.3 Out of Vocabulary Words

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()
```

(continues on next page)

(continued from previous page)

```
# 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,)
print(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,
 '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.
```

3.4 Word Preprocessors

`run_queries` 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 first to search 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.

```python
weat = WEAT()
result = weat.run_query(
    ethnicity_query,
    model,
    preprocessors=[{"lowercase": True, "strip_accents": True}],
    warn_not_found_words=True,
)
print(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': 3.752915130034089,
 'weat': 3.752915130034089,
 'effect_size': 1.2746819501134965,
 '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,
)

print(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': 3.752915130034089,
 'weat': 3.752915130034089,
 'effect_size': 1.2746819501134965,
 '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,
)

print(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': 3.752915130034089,
 'weat': 3.752915130034089,
 'effect_size': 1.2746819501134965,
 '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,
)

print(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': 3.752915130034089,
 'weat': 3.752915130034089,
 'effect_size': 1.2746819501134965,
 'p_value': nan}
```

3.5 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 the library 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
```

3.5.1 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]
```

3.5.2 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]
```

3.5.3 Run the queries on all Word Embeddings using WEAT.

To run our 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`.

```
# Run the queries
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.
```

3.5.4 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}]},
```

(continues on next page)

(continued from previous page)

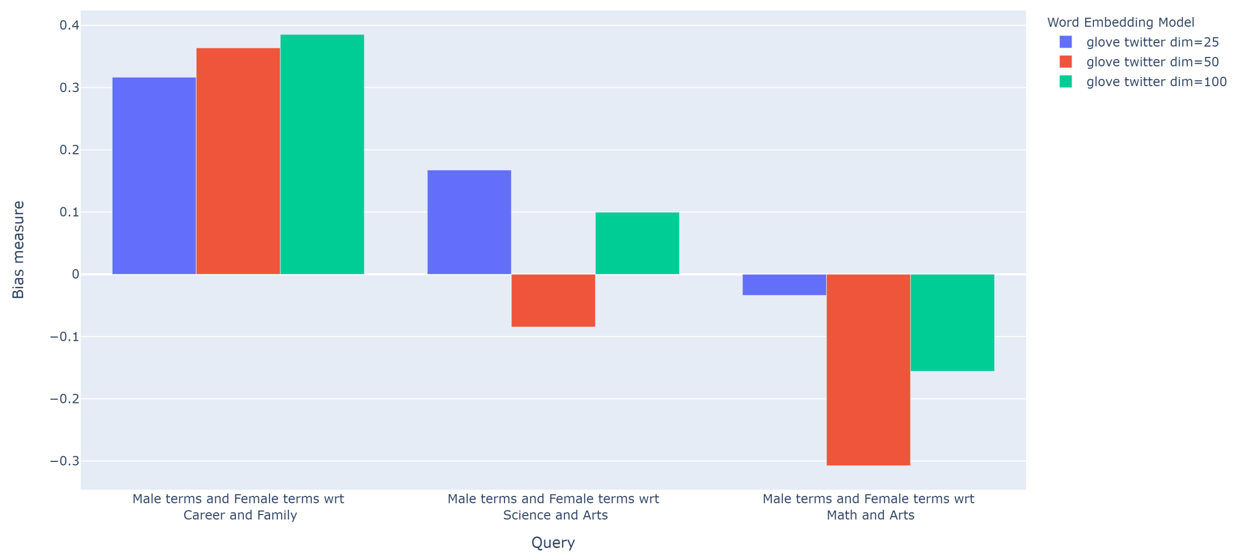
```
    queries_set_name="Gender Queries",
)

WEAT_gender_results
```

No query was null in these results.

3.5.5 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` braplot.

```python
from wefe.utils import plot_queries_results, run_queries

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



3.6 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:

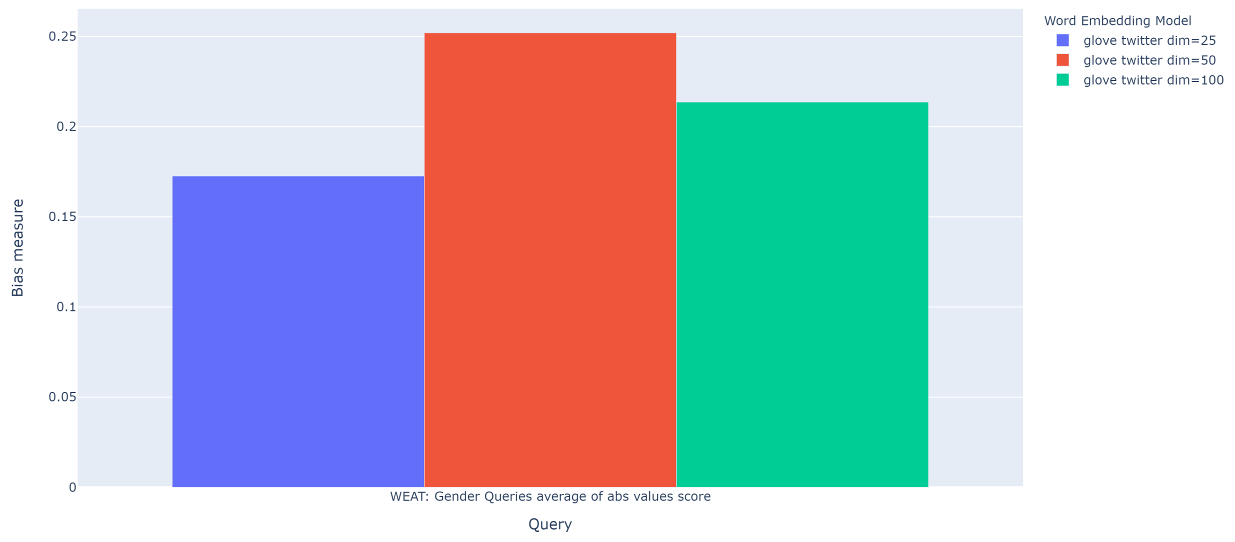
- Average `avg`.
- Average of the absolute values `abs_avg`.

- 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,
```

(continues on next page)

(continued from previous page)

```
    queries_set_name="Gender Queries",
)
WEAT_gender_results_only_agg
```

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

3.7 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:

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:

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.

```
import gensim.downloader as api

from wefe.datasets.datasets import load_weat
from wefe.metrics import RNSB, WEAT
from wefe.query import Query
from wefe.utils import (
    create_ranking,
    plot_ranking,
    plot_ranking_correlations,
    run_queries,
)
from wefe.word_embedding_model import WordEmbeddingModel
```

(continues on next page)

(continued from previous page)

```
# Load the models
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]

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

```
# ---------------------------------------------------------------------------------
# Gender 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"],
)

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,
```

(continues on next page)

(continued from previous page)

```
    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
```

```
# ---------------------------------------------------------------------------------
# Ethnicity 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,
```

(continues on next page)

(continued from previous page)

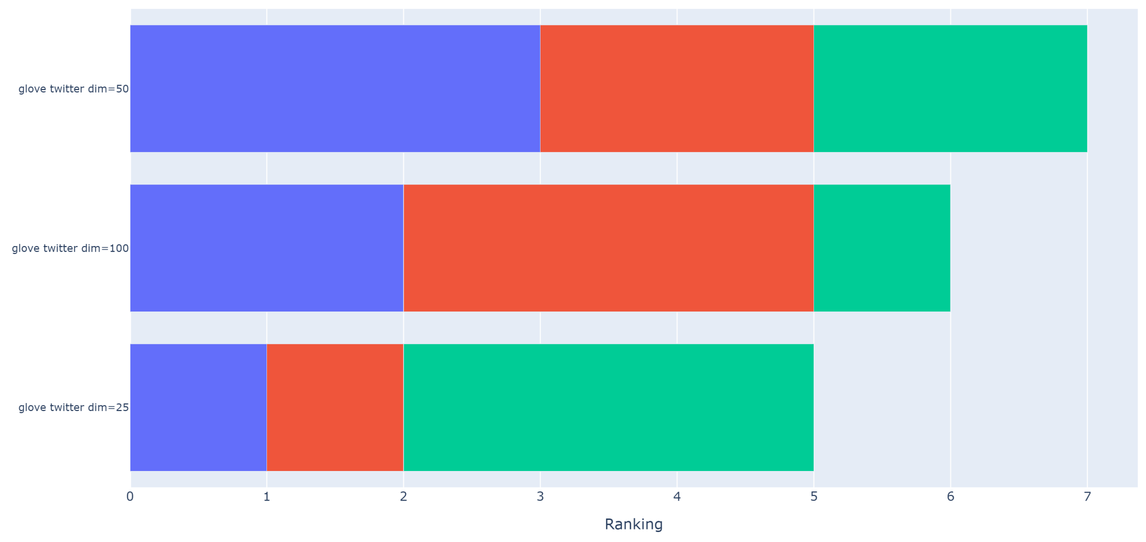
```
    ethnicity_queries,
    models,
    metric_params={"preprocessors": [{"lowercase": True}]},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name="Ethnicity Queries",
)

# 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
```

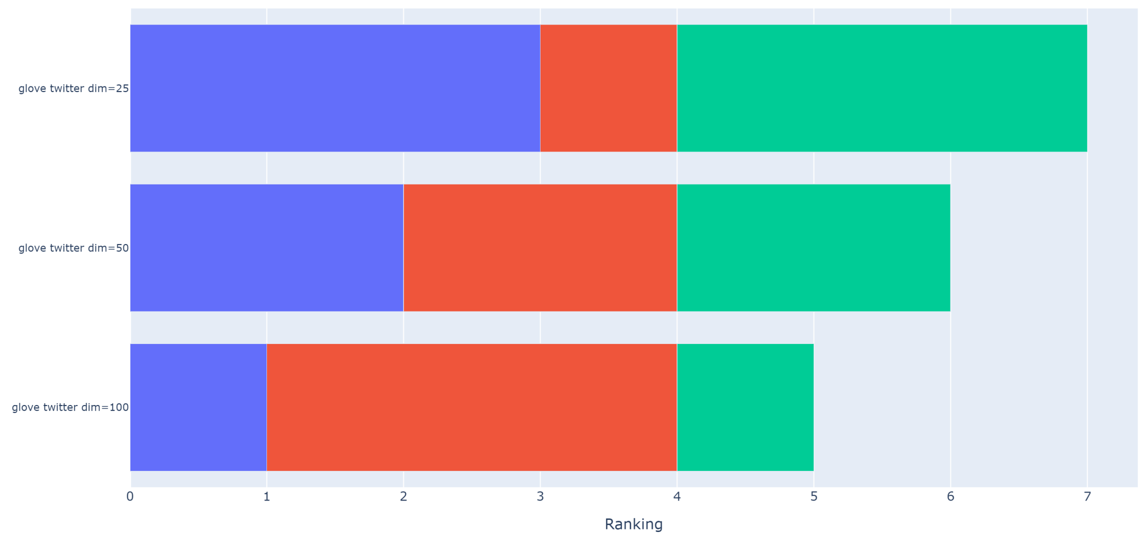
3.7.1 Plotting the rankings

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()
```



3.8 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
```

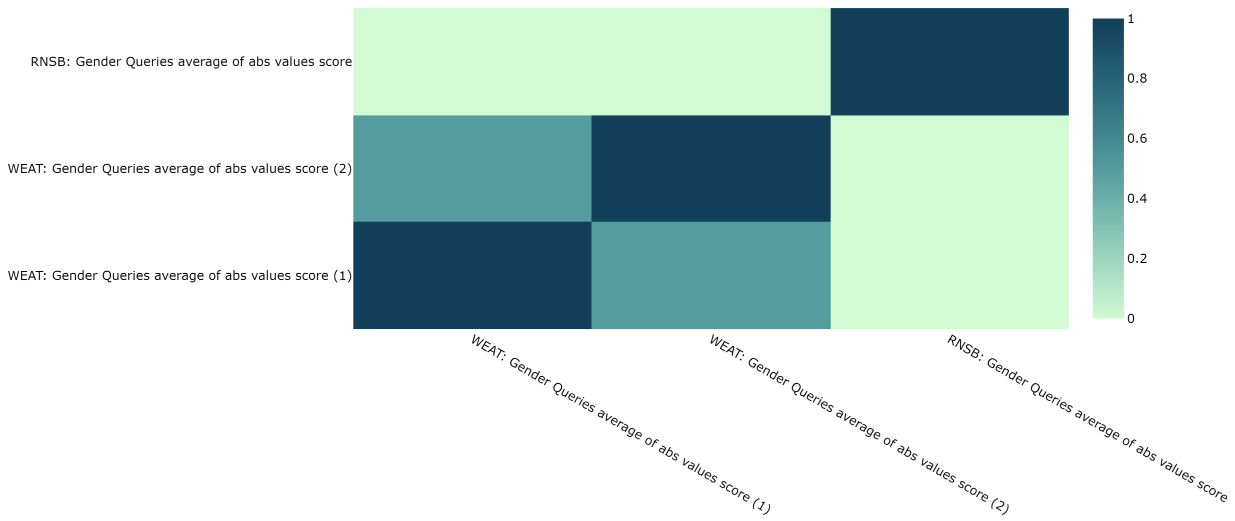
Note: `calculate_ranking_correlations` uses the `corr()` `pandas` dataframe method. The type of correlation that is calculated can be changed through the method parameter. The available options are: `'pearson'`, `'spearman'`, `'kendall'`. By default, the spearman correlation is calculated.

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

WEFE also provides several methods to mitigate the bias of the embedding models. In the following section:

- We present how to reduce binary bias (such as gender bias) using Hard Debias.
- We present how to reduce multiclass bias (such as ethnic having classes like black, white, Latino, etc…) using Multiclass Hard Debias.

```python
import gensim.downloader as api

from wefe.datasets import fetch_debiaswe, load_weat
from wefe.debias.hard_debias import HardDebias
from wefe.metrics import WEAT
from wefe.query import Query
from wefe.word_embedding_model import WordEmbeddingModel

twitter_25 = api.load("glove-twitter-25")
model = WordEmbeddingModel(twitter_25, "glove-twitter-dim=25")
```

4.1 Hard Debias

This method allow reducing the bias of an embedding model through geometric operations between embeddings. This method is binary because it only allows two 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. **Neutralize the bias subspace on the 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 gender: `{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.

- projection = v • (v • u) / (v • v)

Then subtract the projection from the embedding.

- u' = u - projection

3. **Equalize 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 a pair of embeddings and distributes them both at the same distance from the bias direction, so that neither is closer to the bias direction than the other.

4.1.1 Fit-Transform Interface

WEFE implements all debias methods through an interface inspired by the transformers of `scikit-learn`. That is, the execution of a debias method involves two steps: - First a training through the `fit` method where the transformation that will be applied on the embeddings is calculated - Second, a `transform` that applies the trained transformation.

Each of these stages defines its own parameters.

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).

```
debiaswe_wordsets = fetch_debiaswe()

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

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

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

Next, the 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.
INFO:gensim.models.keyedvectors:precomputing L2-norms of word weight vectors
Model copy created successfully.
100%|| 1193514/1193514 [00:18<00:00, 65143.18it/s]
INFO:gensim.models.keyedvectors:precomputing L2-norms of word weight vectors
INFO:gensim.models.keyedvectors:precomputing L2-norms of word weight vectors
```

Using the metrics displayed in the first section of this user guide, we can measure whether or not there was a change in the measured bias between the original model and the debiased model.

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

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

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

print(round(debiased_results_1["weat"], 3),"<",round(biased_results_1["weat"], 3),
    "=",debiased_results_1["weat"] < biased_results_1["weat"],)
```

```
-0.06 < 0.317 = True
```

```
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(
    round(debiased_results_2["weat"], 3),"<",round(biased_results_2["weat"], 3),
    "=",debiased_results_2["weat"] < biased_results_2["weat"],)
```

```
-1.033 < -0.949 = True
```

4.1.2 Target Parameter

- target: If a set of words is specified in target, the debias method will be performed only on the word embeddings associated with this set. In the case of providing `None`, the transformation will be 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',
           '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, 10754.63it/s]
INFO:gensim.models.keyedvectors:precomputing L2-norms of word weight vectors
INFO:gensim.models.keyedvectors:precomputing L2-norms of word weight vectors
```

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.

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

```

(continues on next page)

(continued from previous page)

```
print(round(debiased_results_1["weat"], 3),"<",round(biased_results_1["weat"], 3)
      ,"=",debiased_results_1["weat"] < biased_results_1["weat"],)
```

```
-0.06 < 0.317 = True
```

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. Equalization can be deactivated by delivering an empty equalize set (`[]`)

```
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(round(debiased_results_2["weat"], 3),"<",round(biased_results_2["weat"], 3),
    "=",debiased_results_2["weat"] < biased_results_2["weat"],)
```

```
-0.941 < -0.949 = False
```

4.1.3 Save 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')
```

```
INFO:gensim.utils:saving Word2VecKeyedVectors object under gender_debiased_glove.kv,␣
↪separately None
INFO:gensim.utils:storing np array 'vectors' to gender_debiased_glove.kv.vectors.npy
INFO:gensim.utils:not storing attribute vectors_norm
DEBUG:smart_open.smart_open_lib:{'uri': 'gender_debiased_glove.kv', 'mode': 'wb',
↪'buffering': -1, 'encoding': None, 'errors': None, 'newline': None, 'closefd': True,
↪'opener': None, 'ignore_ext': False, 'compression': None, 'transport_params': None}
INFO:gensim.utils:saved 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()
)

mhd = MulticlassHardDebias(verbose=True, 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)
```

```
INFO:wefe.debias.multiclass_hard_debias:PCA variance explaned: [4.0089381e-01 2.3377793e-
↪01 1.7155512e-01 7.3547199e-02 5.5353384e-02
3.5681739e-02 2.2261711e-02 6.9290772e-03 2.4344339e-15 2.4052477e-15]
Obtaining definitional sets.
Word(s) found: ['black', 'caucasian', 'asian'], not found: []
Word(s) found: ['african', 'caucasian', 'asian'], not found: []
Word(s) found: ['black', 'white', 'asian'], not found: []
Word(s) found: ['africa', 'america', 'asia'], not found: []
Word(s) found: ['africa', 'america', 'china'], not found: []
Word(s) found: ['africa', 'europe', 'asia'], not found: []
6/6 sets of words were correctly converted to sets of embeddings
Identifying the bias subspace.
Obtaining equalize pairs.
Word(s) found: ['manager', 'executive', 'redneck', 'hillbilly', 'leader', 'farmer'], not␣
↪found: []
Word(s) found: ['doctor', 'engineer', 'laborer', 'teacher'], not found: []
Word(s) found: ['slave', 'musician', 'runner', 'criminal', 'homeless'], not found: []
3/3 sets of words were correctly converted to sets of embeddings
Executing Multiclass Hard Debias on glove-twitter-dim=25
copy argument is True. Transform will attempt to create a copy of the original model.␣
↪This may fail due to lack of memory.


INFO:gensim.models.keyedvectors:precomputing L2-norms of word weight vectors

Model copy created successfully.
Normalizing embeddings.
Neutralizing embeddings

100%|| 1193504/1193504 [01:38<00:00, 12108.73it/s]
INFO:gensim.models.keyedvectors:precomputing L2-norms of word weight vectors
```

(continues on next page)

(continued from previous page)

```
DEBUG:wefe.debias.multiclass_hard_debias:Equalizing embeddings..
INFO:gensim.models.keyedvectors:precomputing L2-norms of word weight vectors

Normalizing embeddings.
Normalizing embeddings.
Done!
```

```
# test with weat

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"],
)

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}],
)
```

Absolute value is used here because the closer the value is to zero, the less biased the model is.

```
import numpy as np

print(
    '| -',
    round(np.abs(debiased_results["weat"]), 3),
    "| < | -",
    round(np.abs(biased_results["weat"]), 3),
    "| =",
    np.abs(debiased_results["weat"]) < np.abs(biased_results["weat"]),
)
```

```
| - 0.027 | < | - 0.088 | = True
```

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 yet 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
>>>
>>> glove_embedding = WordEmbeddings('glove') # 100 dim glove
>>>
>>> # extract KeyedVectors object
>>> glove_keyed_vectors = glove_embedding.precomputed_word_embeddings
>>> glove_100 = WordEmbeddingModel(glove_keyed_vectors, 'glove-100')
>>>
>>> result = weat.run_query(query, glove_100)
>>> print(result)
{'query_name': 'Male terms and Female terms wrt Career and Family', 'result': 1.0486683}
```

CHAPTER

SIX

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!

6.1 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 PEP8 coding style.

6.2 Testing

All unit tests are located in the wefe/test folder and are based on the `pytest` framework. In order to run tests, you will first need to install `pytest` and `pytest-cov`:

```
pip install -U pytest
pip install pytest-cov
```

To run the tests, execute:

```
pytest wefe
```

To check the coverage, run:

```
py.test wefe --cov-report xml:cov.xml --cov wefe
```

And then:

```
coverage report -m
```

6.3 Build the documentation

The documentation is created using sphinx. It can be found in the doc 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
```

6.4 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.

6.4.1 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'
```

6.4.2 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:

```python
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 first to search 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)

```
        }
        """
```

6.4.3 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 first to search 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.

6.5 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*.

6.5.1 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*.

6.5.2 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 rise
        `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 give rise 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

SEVEN

WEFE API

This is the documentation of the API of WEFE.

7.1 WordEmbeddingModel

`WordEmbeddingModel`(wv[, name, vocab_prefix])	A wrapper for Word Embedding pre-trained models.

7.1.1 `wefe`.WordEmbeddingModel

`class wefe.`**`WordEmbeddingModel`**(*wv: gensim.models.keyedvectors.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: gensim.models.keyedvectors.KeyedVectors*, *name: Optional[str] = None*, *vocab_prefix: Optional[str] = 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[numpy.ndarray], numpy.ndarray]*)

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()

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: numpy.ndarray*)

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.

7.2 Query

`Query`(target_sets, attribute_sets[, ...])	A container for attribute and target word sets.

7.2.1 `wefe`.Query

`class wefe.`**`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*)

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.

get_subqueries(*new_template: tuple*) → list

Generate the subqueries from this query using the given template

7.3 Metrics

This list contains the metrics implemented in WEFE.

WEAT()	Word Embedding Association Test (WEAT).

7.3.1 `wefe`.WEAT

`class wefe.`**`WEAT`**

Word Embedding Association Test (WEAT).

The metric was originally proposed in [1]. It measures the degree of association between two sets of target words and two sets of attribute words through a permutation test.

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: wefe.query.Query*, *model: wefe.word_embedding_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 first to search 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

```
>>> 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"],
...     ],
```

(continues on next page)

(continued from previous page)

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

RND()	Relative Norm Distance (RND).

7.3.2 wefe.RND

class wefe.**RND**

Relative Norm Distance (RND).

It measures the relative strength of association of a set of neutral words with respect to two groups.

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: wefe.query.Query*, *model: wefe.word_embedding_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 first to search 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

```
>>> 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,
 'rnd': 0.030381828546524048,
 'distances_by_word': {'wedding': -0.1056304,
                       'marriage': -0.10163283,
                       'children': -0.068374634,
                       'parents': 0.00097084045,
```

(continues on next page)

(continued from previous page)

```
                       '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,
 'distances_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,
 'distances_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}}
```

RNSB()	Relative Relative Negative Sentiment Bias (RNSB).

7.3.3 wefe.RNSB

class wefe.**RNSB**

Relative Relative Negative Sentiment Bias (RNSB).

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: Dict[str, Any] = {'max_iter': 10000, 'solver': 'liblinear'}, num_iterations: int = 1, random_state: Optional[int] = None, print_model_evaluation: 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 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 num_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, optional] A scikit-learn classifier class that implements predict_proba function, by default None,

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

num_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, by default 1.

random_state [Union[int, None], optional] Seed that allow making the execution of the query reproducible. Warning: if a random_state other than None is provided along with num_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.

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` `{'lowercase': 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 first to search 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

```
>>> from wefe.query import Query
>>> from wefe.utils import load_test_model
>>> from wefe.metrics import RNSB
>>>
>>> # 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",],
```

(continues on next page)

(continued from previous page)

```
...         ["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": 0.09223875552506647,
    "kl-divergence": 0.09223875552506647,
    "clf_accuracy": 1.0,
    "negative_sentiment_probabilities": {
        "female": 0.5543954373912665,
        "woman": 0.3107589242224508,
        "girl": 0.18710587484907013,
        "sister": 0.1787081823837198,
        "she": 0.4172419154977331,
        "her": 0.4030950036121549,
        "hers": 0.3126640373120572,
        "daughter": 0.14249529344431694,
        "male": 0.4422224610164615,
        "man": 0.4194123616222211,
        "boy": 0.20556697141459176,
        "brother": 0.19801831727151584,
        "he": 0.5577524826493919,
        "him": 0.514179075019818,
        "his": 0.5544435993736733,
        "son": 0.18711536982098712,
    },
    "negative_sentiment_distribution": {
        "female": 0.09926195811727109,
        "woman": 0.05563995884577796,
        "girl": 0.0335004479837668,
        "sister": 0.0319968796973831,
        "she": 0.0747052496243332,
        "her": 0.07217230999250153,
        "hers": 0.05598106059906622,
        "daughter": 0.02551312816774791,
        "male": 0.07917790162647549,
        "man": 0.07509385803950792,
        "boy": 0.03680582257831352,
        "brother": 0.0354542707060297,
        "he": 0.09986302166025017,
        "him": 0.09206140304753956,
        "his": 0.09927058129913385,
        "son": 0.03350214801490194,
```

(continues on next page)

(continued from previous page)

```
    },
}
```

`MAC()`	Mean Average Cosine Similarity (MAC).

7.3.4 `wefe`.MAC

class `wefe.MAC`

Mean Average Cosine Similarity (MAC).

The general steps of the test are as follows [1].

1. Embed all target and attribute words.
2. 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 save it in a array.
3. 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: wefe.query.Query*, *model: wefe.word_embedding_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 first to search 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

```
>>> 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},
    'son': {'Family': 0.5764635019004345, 'Careers': 0.9220191016211174}}}}
```

ECT()	Embedding Coherence Test [1].

7.3.5 wefe.ECT

class wefe.**ECT**

Embedding Coherence Test [1].

The metric was originally proposed in [1] and implemented in [2]. 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.

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: wefe.query.Query*, *model: wefe.word_embedding_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 first to search 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=[
...         [
...             "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}
```

RIPA()	An implementation of the Relational Inner Product Association Test, proposed by [1][2].

7.3.6 `wefe`.RIPA

class `wefe.RIPA`

An implementation of the Relational Inner Product Association Test, proposed by [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.

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: wefe.query.Query*, *model: wefe.word_embedding_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 first to search 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.

7.4 Debias

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

`HardDebias`([pca_args, verbose, criterion_name])	Hard Debias debiasing method.

7.4.1 `wefe`.HardDebias

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

Hard Debias debiasing 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. 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 gender: *{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. - bias_subspace = v • (v • u) / (v • v)

Then subtract the projection from the embedding. - u' = u - 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, such 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

`__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: wefe.word_embedding_model.WordEmbeddingModel*, *definitional_pairs: Sequence[Sequence[str]]*, *equalize_pairs: Optional[Sequence[Sequence[str]]] = None*, ***fit_params*) → wefe.debias.base_debias.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.

Returns

BaseDebias The debias method fitted.

`transform`(*model: wefe.word_embedding_model.WordEmbeddingModel*, *target: Optional[List[str]] = None*, *ignore: Optional[List[str]] = None*, *copy: bool = True*) → *wefe.word_embedding_model.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 give rise to *MemoryError*, by default True.

Returns

WordEmbeddingModel The debiased embedding model.

`MulticlassHardDebias`([pca_args, verbose, ...])	Generalized version of Hard Debias that enables multiclass debiasing.

7.4.2 `wefe`.MulticlassHardDebias

class `wefe.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

`__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:* wefe.word_embedding_model.WordEmbeddingModel, *definitional_sets: Sequence[Sequence[str]]*, *equalize_sets: Sequence[Sequence[str]]*) → wefe.debias.base_debias.BaseDebias

Compute the bias direction and obtains the equalize embedding pairs.

Parameters

model [WordEmbeddingModel] The word embedding model to debias.

definitional_sets [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_sets, by default None.

Returns

BaseDebias The debias method fitted.

`transform`(*model:* wefe.word_embedding_model.WordEmbeddingModel, *target: Optional[List[str]] = None*, *ignore: Optional[List[str]] = None*, *copy: bool = True*) → *wefe.word_embedding_model.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). 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 give rise to *MemoryError*, by default True.

Returns

WordEmbeddingModel The debiased embedding model.

7.5 Dataloaders

The following functions allow one to load word sets used in previous works.

load_bingliu()	Load the Bing-Liu sentiment lexicon.

7.5.1 wefe.load_bingliu

wefe.**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.

fetch_debias_multiclass()	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.

7.5.2 wefe.fetch_debias_multiclass

wefe.**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

`fetch_debiaswe`()	Fetch the word sets used in the paper Man is to Computer Programmer as Woman is to Homemaker? from the source.

7.5.3 `wefe`.fetch_debiaswe

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

References

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.

`fetch_eds`([occupations_year, ...])	Fetch the word sets used in the experiments of the work *Word Embeddings *Quantify 100 Years Of Gender And Ethnic Stereotypes*.

7.5.4 `wefe`.fetch_eds

`wefe.`**`fetch_eds`**`(`*occupations_year: int = 2015*, *top_n_race_occupations: int = 15*`)` → Dict[str, List[str]]

Fetch the word sets used in the experiments of the work *Word Embeddings *Quantify 100 Years Of Gender And Ethnic Stereotypes*.

This dataset includes gender (male, female), ethnicity (asian, black, white) and religion (christianity and islam) and adjetives (appearence, intelligence, otherization, sensitive) word sets.

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

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.

load_weat()	Load the word sets used in the paper *Semantics Derived Automatically From Language Corpora Contain Human-Like Biases*.

7.5.5 wefe.load_weat

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

Load the word sets used in the paper *Semantics Derived Automatically From Language Corpora Contain Human-Like Biases*. It includes gender (male, female), ethnicity (black, white) and pleasant, unpleasant word sets, among others.

Returns

word_sets_dict [dict] A dictionary with the word sets.

References

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

7.6 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.

preprocess_word(word[, options, vocab_prefix])	pre-processes a word before it is searched in the model's vocabulary.

7.6.1 wefe.preprocess_word

`wefe.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.

`get_embeddings_from_set`(model, word_set[, ...])	Transform a sequence of words into dictionary that maps word - word embedding.

7.6.2 `wefe`.get_embeddings_from_set

`wefe.`**`get_embeddings_from_set`**(*model:* wefe.word_embedding_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, numpy.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 first to search 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.

`get_embeddings_from_sets`(model, sets[, ...])	Given a sequence of word sets, obtain their corresponding embeddings.

7.6.3 `wefe`.get_embeddings_from_sets

`wefe.`**`get_embeddings_from_sets`**(*model: wefe.word_embedding_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, numpy.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 first to search 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.

get_embeddings_from_query(model, query[, ...])	Obtain the word vectors associated with the provided Query.

7.6.4 `wefe`.get_embeddings_from_query

`wefe.`**get_embeddings_from_query**(*model: wefe.word_embedding_model.WordEmbeddingModel, query: wefe.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, numpy.ndarray]], Dict[str, Dict[str, numpy.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 first to search 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.

CHAPTER

EIGHT

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

NINE

REPLICATION OF PREVIOUS STUDIES

All replications of other studies that WEFE has currently implemented are in the Examples folder.

Below we list some examples:

9.1 WEAT Replication

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.

9.2 RNSB Replication

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.

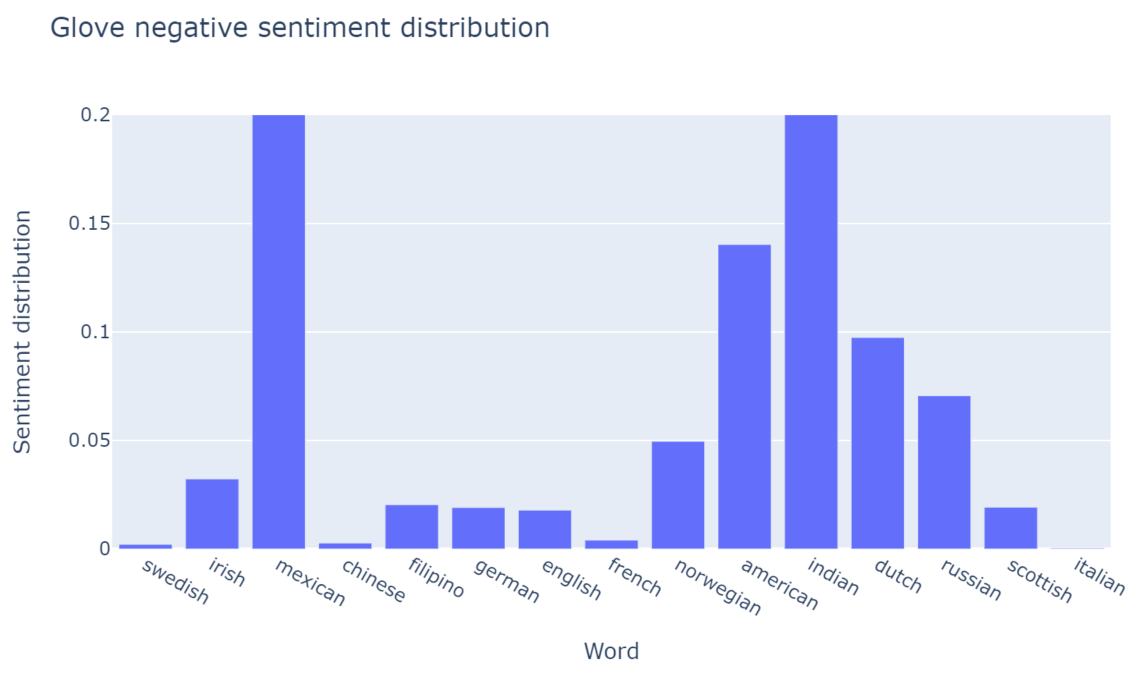
Note: Due to the formulation of the metric (it trains a logistic regression in each execution) our results are not exactly the same as those reported in the original paper. However, our results are still very similar to those in the original paper.

```
>>> 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
>>> import gensim.downloader as api
>>>
>>> # load the target word sets.
```

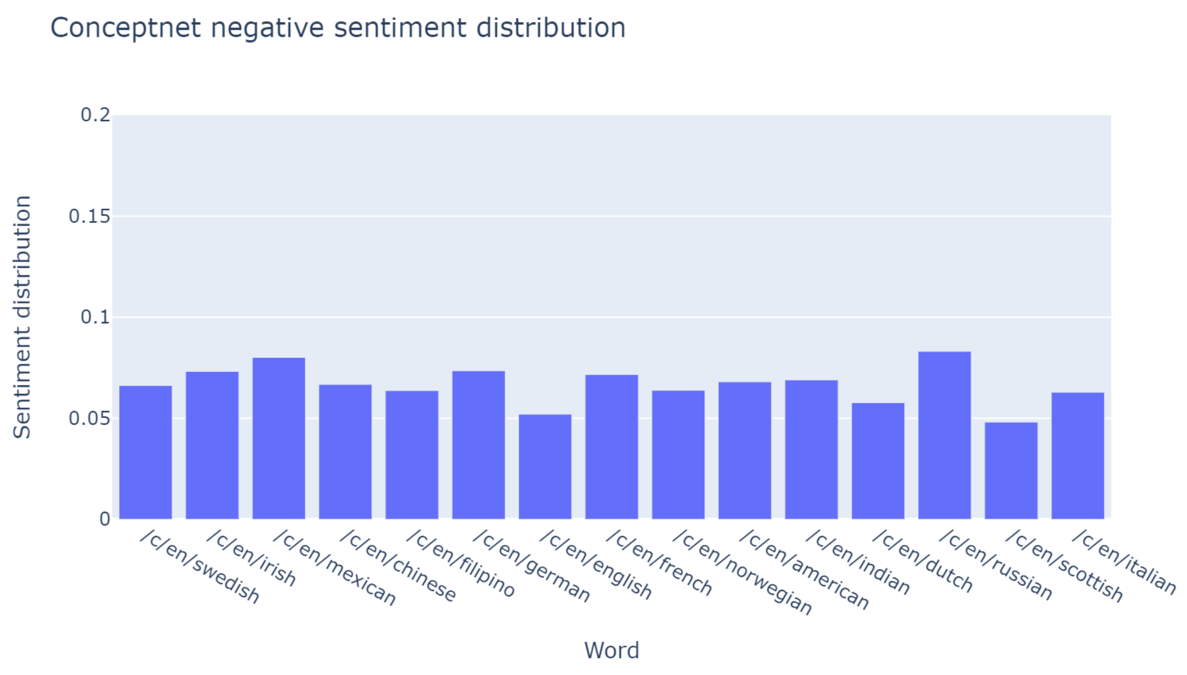
(continues on next page)

(continued from previous page)

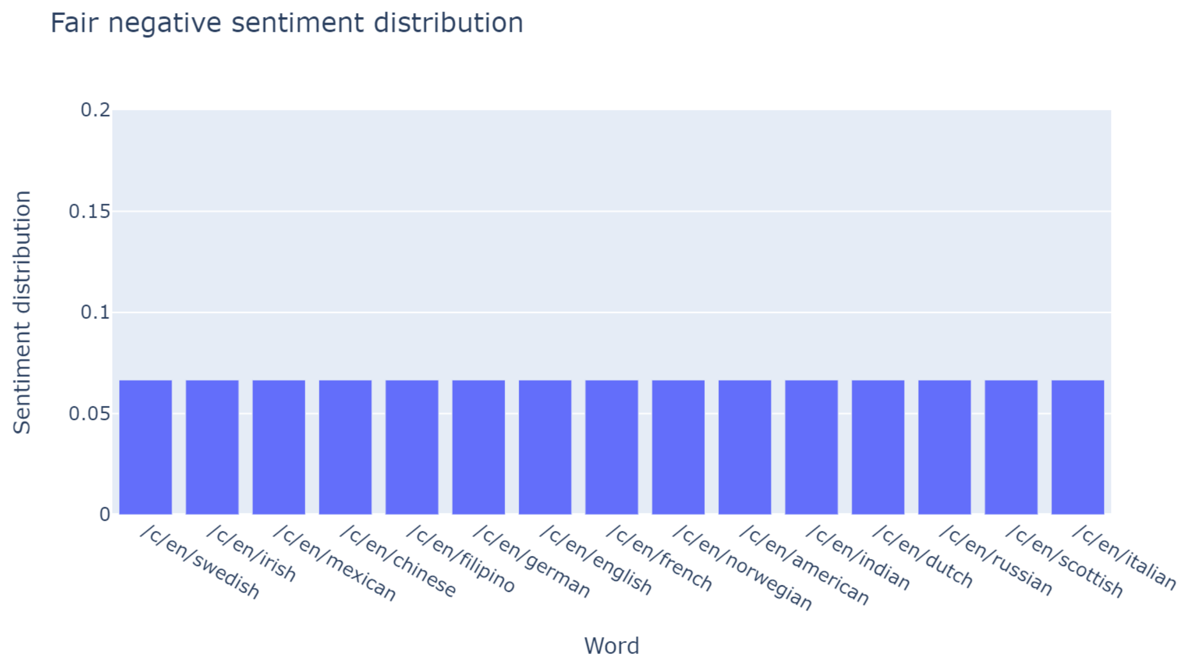
```
>>> # In this case each word is an objective set because each of them represents a␣
↪different social group.
>>> RNSB_words = [
>>>     ['swedish'], ['irish'], ['mexican'], ['chinese'], ['filipino'], ['german'], [
↪'english'],
>>>     ['french'], ['norwegian'], ['american'], ['indian'], ['dutch'], ['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 TEN

REPOSITORY

You can find the project repository at the following link: WEFE repository on Github.

INDEX