WEFE Documentation

Release 0.2.2

Pablo Badilla

Oct 06, 2021

GETTING STARTED

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

The following pages contain information about the formulation of 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 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.

It also provides more advanced features for:

- Running several queries on multiple embedding models and returning a DataFrame with the results.
- Plotting those results on a barplot.
- Based on the above results, calculating a bias ranking for all embedding models. This allows the user to evaluate the fairness of the embedding models according to the bias criterion (defined by the query) and the metric used.
- Plotting the ranking on a barplot.
- Correlating the rankings. This allows the user to see how the rankings of the different metrics or evaluation criteria are correlated with respect to the bias presented by the models.

1.1 Motivation and objectives

Word Embeddings models are a core component in almost all NLP systems. Several studies has 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.

In order to address the above, we first proposed WEFE as a theoretical framework that aims to formalize the main building blocks for measuring bias in word embedding models. Then, the need to conduct our case study led to the implementation of WEFE in code. 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.

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 simple interface to develop new metrics.

- To solve the two main problems that arise when comparing experiments based on different metrics:
  - Some metrics operate with different numbers of word sets as input.
  - The outputs of different metrics are incompatible with each other (their scales are different, some metrics return real numbers and others only positive ones, etc..)

1.2 The Framework

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

1.2.1 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.

1.2.2 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"*.

1.2.3 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

$$to$$

$$A_{\text{science}} = \{math, physics, chemistry, \ldots\},$$
$$A_{\text{art}} = \{poetry, dance, literature, \ldots\}.$$

$$= \{math, physics, chemistry, \ldots\}, A_{\overline{\text{art}}} \{poetry, dance, literature, \ldots\}.$$

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 applies to this and all pages of the documentation.

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

1.2.5 Fairness Measure

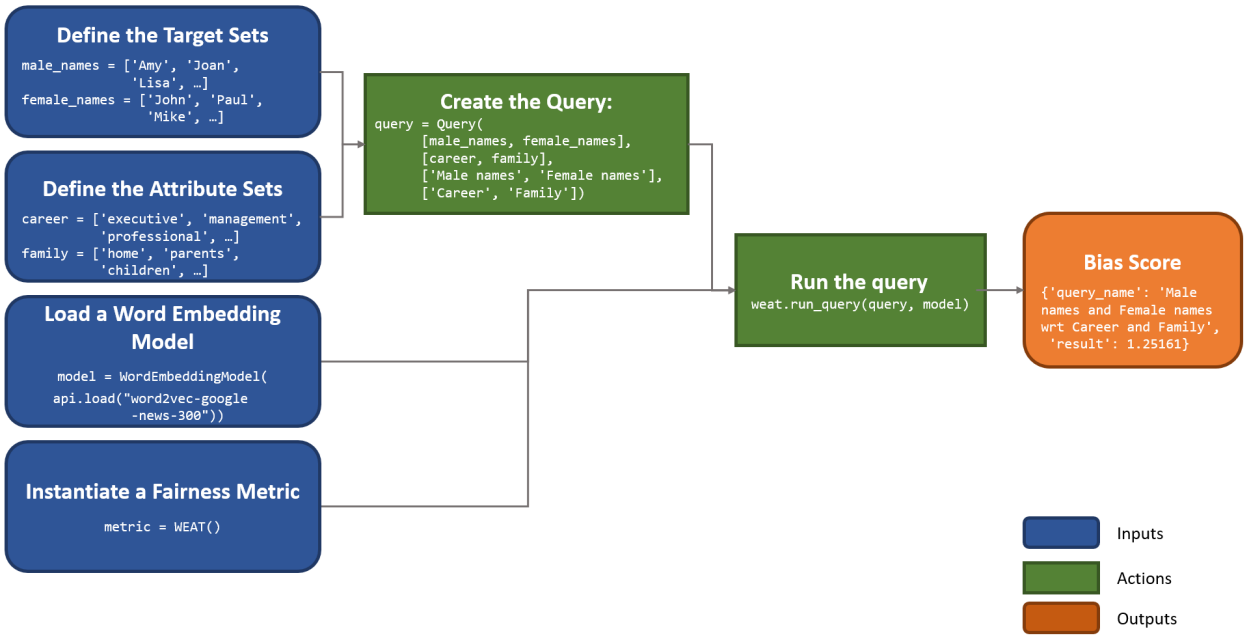
A fairness metric is a function that quantifies the degree of association between target and attribute words in a word embedding model. In our framework, every fairness metric is defined as a function that has a query and a model as input, and produces a real number as output.

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.

1.2.6 Standard usage pattern of WEFE

The following flow chart shows how to perform a bias measurement using a gender query, word2vec embeddings and the WEAT metric.



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

1.3 Metrics

The metrics implemented in the package so far are:

1.3.1 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.

1.3.2 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).

1.3.3 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.

1.3.4 MAC

Mean Average Cosine Similarity (MAC), presented in the paper "*Black is to criminals caucasian is to police: Detecting and removing multiclass bias in word embeddings*".

1.3.5 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.

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.

1.4 Changelog

- Renamed optional `run_query` parameter `warn_filtered_words` to *warn_not_found_words*.
- Added `word_preprocessor_args` parameter to `run_query` that allows to specify transformations prior to searching for words in word embeddings.
- Added `secondary_preprocessor_args` parameter to `run_query` which allows to specify a second preprocessor transformation to words before searching them in word embeddings. It is not necessary to specify the first preprocessor to use this one.
- Implemented `__getitem__` function in `WordEmbeddingModel`. This method allows to obtain an embedding from a word from the model stored in the instance using indexers.
- Removed underscore from class and instance variable names.
- Improved type and verification exception messages when creating objects and executing methods.
- Fix an error that appeared when calculating rankings with two columns of aggregations with the same name.
- Ranking correlations are now calculated using pandas `corr` method.
- Changed metric template, name and short_names to class variables.
- Implemented `random_state` in RNSB to allow replication of the experiments.
- run_query now returns as a result the default metric requested in the parameters and all calculated values that may be useful in the other variables of the dictionary.
- Fixed problem with api documentation: now it shows methods of the classes.
- Implemented p-value for WEAT

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 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 the last works about 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 de-bias module.
- Implement a visualization module.
- Implement p-values with statistic resampling to all metrics.

1.8 Licence

WEFE is licensed under the BSD 3-Clause License.

Details of the license on this link.

1.9 Team

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

Thank you very much to all our contributors!

1.10 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.

1.11 Acknowledgments

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

CHAPTER

TWO

QUICK START

In this tutorial we will 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 wefe
```

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

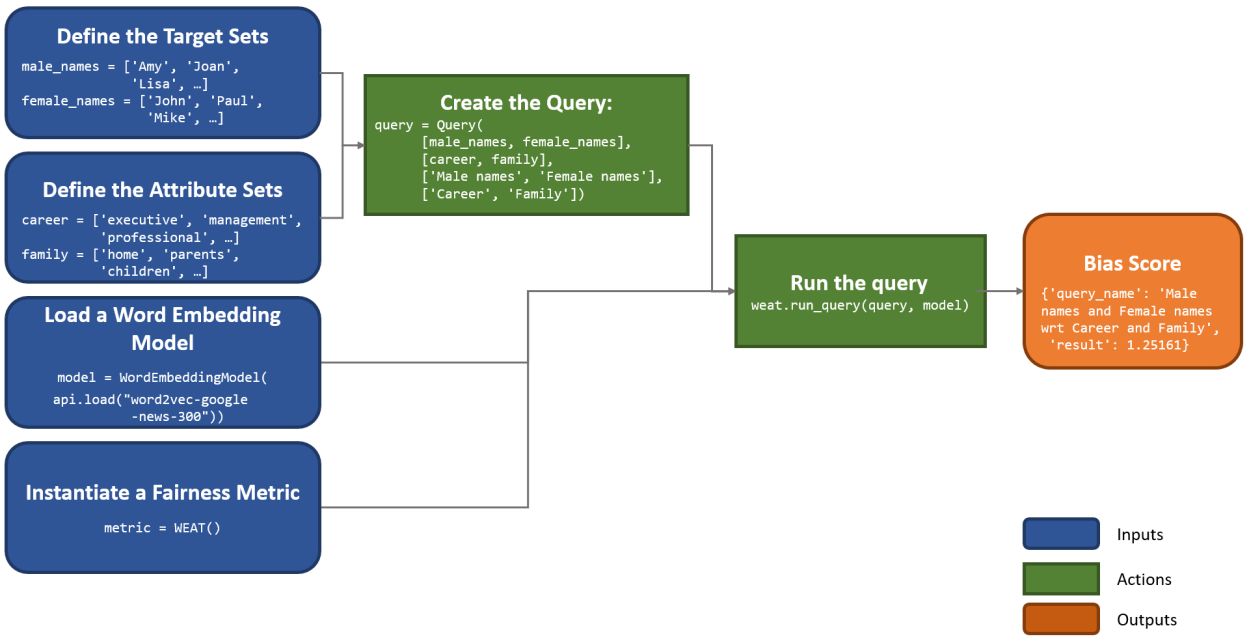
```
conda install -c pbadilla wefe
```

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 will be widely used in the following sections.

In the following code we will show how to implement the example query presented in WEFE's home page: A gender Query using WEAT metrics on the google's word2vec Word Embedding model.

The following graphic shows the flow of the query execution:



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

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

These stages will 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 will use `word2vec-google-news-300` model, but in order to speed up the execution time, the embedding model could be changed to `glove-twitter-25'`.

```
>>> # 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 word2vec word2vec-google-news-300
>>> # it can be changed to 'word2vec-google-news-300' to speed use word2vec.
>>> 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 will 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']
>>>
```

(continues on next page)

(continued from previous page)

```
>>> attribute_sets = [['math', 'physics', 'chemistry'], ['poetry','dance','literature']]
>>> attribute_sets_names = ['Science', 'Arts']
>>>
>>> # 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 will use `WEAT`.

```
>>> # instance a WEAT metric
>>> weat = WEAT()
>>> result = weat.run_query(query, model)
>>> print(result)
{'query_name': 'Male Terms and Female Terms wrt Arts and Science',
 'result': -0.010003209}
```

We close the basic tutorial on the use of the WEFE package. For more advanced examples, visit user the User Guide section.

CHAPTER

THREE

USER GUIDE

The following guide is designed to present the more general details on using the package. Below:

- We first present how to run a simple query using some embedding model.
- We then show how to run multiple queries on multiple embeddings.
- After that, we show how to compare the results obtained from running multiple sets of queries on multiple embeddings using different metrics through ranking calculation.
- Finally, we show how to calculate the correlations between the rankings obtained.

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 applies to this and all pages of the 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 will be 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 code explains how to run a gender query using Glove. embeddings and the Word Embedding Association Test (`WEAT`) as fairness metric.

Below we show the three usual steps for performing a query in `WEFE`:

```
# Load the package
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
import gensim.downloader as api
```

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

Here, we load the word embedding pretrained model using the gensim library and then we create a WordEmbedding-Model instance. For this example, we will use a 25-dimensional `Glove` embedding model trained from a Twitter dataset.

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

3.1.2 Create the query using a `Query` object

Define the target and attribute words sets and create a Query object that contains them. Some well-known word sets are already provided by the package and can be easily loaded by the user. Users can also set their own custom-made sets.

For this example, we will create a query with gender terms with respect to family and career. The words we will use will be taken from the set of words used in the `WEAT` paper (included in the package).

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

gender_query = Query([word_sets['male_terms'], word_sets['female_terms']],
              [word_sets['career'], word_sets['family']],
              ['Male terms', 'Female terms'], ['Career', 'Family'])
```

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.

The bias measurement process consists of three stages:

1. Checking the measurement parameters.
2. Transform the word sets into word embeddings.
3. Calculate the metric.

In this case we are going to use the `WEAT` metric.

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

```
{'query_name': 'Male terms and Female terms wrt Career and Family',
 'result': 0.3165841,
 'weat': 0.3165841,
 'effect_size': 0.677944,
 'p-value': None}
```

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. Also some metrics allow you to change the default value in results, which will have implications a little further down the line.

In this case, `WEAT` returns the value of `weat` and the `effect_size`, with weat as default in the results key.

3.1.4 Metric Params

Each metric allows to vary the behavior of `run_query` according to different parameters. For example: there are parameters to change the preprocessing of the words, others to warn errors or to modify what the method returns by default.

The parameters of each metric are detailed in the API documentation.

In this case, if we want `run_query` returns `effect_size` instead of `weat` in the result , when we execute `run_query` we can pass the parameter `return_effect_size` equal to `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': 'Male terms and Female terms wrt Career and Family',
 'result': 0.677944,
 'weat': 0.3165841,
 'effect_size': 0.677944,
 'p-value': None}
```

3.1.5 Word preprocessors

There may be word embeddings models whose words are not cased or that do not have accents. In `Glove`, for example, all its words in its vocabulary are lowercase. However, many words in WEAT's ethnicity dataset contain cased words.

```
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']
```

If we carelessly execute the following query, when transforming word sets to embeddings we could lose many words or the whole of several sets.

You can specify that `run_query` log the words that were lost in the transformation to vectors by using the parameter `warn_not_found_words=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(5)', 'African american names(5)'],
    ['Pleasant(5)', 'Unpleasant(5)'])
```

(continues on next page)

(continued from previous page)

```
result = weat.run_query(ethnicity_query,
                        model,
                        warn_not_found_words=True)
print(result)
```

```
WARNING:root:The following words from set 'European american names(5)' 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(5)' 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(5)' 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(5)' 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(5) and African american names(5)␣
↪wrt Pleasant(5) and Unpleasant(5)' 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(5) and African american names(5) wrt Pleasant(5)␣
↪and Unpleasant(5)', 'result': nan, 'weat': nan, 'effect_size': nan}
```

Warning

In order to give more robustness to the results, if more than 20% (by default) of the words from any of the word sets of the query are not included in the word embedding model, the result of the metric will be `np.nan`. This behavior can be changed using a float number parameter called `lost_vocabulary_threshold`.

One of the parameters of `run_query`, `preprocessor_args` allows to run a preprocessor to each word of all sets before getting its vectors. This preprocessor can specify that words be preprocessed to lowercase, remove accents or any other custom preprocessing given by the user.

The possible options for `preprocessor_args` are:

- `lowercase`: `bool`. Indicates if the words are transformed to lowercase.
- `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).

```
weat = WEAT()
result = weat.run_query(ethnicity_query,
                        model,
                        preprocessor_args={
```

(continues on next page)

(continued from previous page)

```
                            'lowercase': True,
                            'strip_accents': True
                        })
print(result)
```

```
{'query_name': 'European american names(5) and African american names(5) wrt Pleasant(5)␣
↪and Unpleasant(5)', 'result': 3.7529151, 'weat': 3.7529151, 'effect_size': 1.2746819,
↪'p-value': None}
```

It may happen that first you want to try to find the vector of a word in uppercase, (since this vector may contain more information than the one of the word lowercased) and if it is not exists in the model, then try to find its lowercase representation. This behavior can be specified by specifying preprocessing options in `secondary_preprocessor_args` and leaving the primary by default (i,e,. without providing it).

In general, the search for vectors will be done first by using the preprocessor specified in `preprocessor_args` and then the specified in `secondary_preprocessor_args` if this was provided. Therefore, any combination of these is also supported.

```
weat = WEAT()
result = weat.run_query(ethnicity_query,
                        model,
                        secondary_preprocessor_args={
                            'lowercase': True,
                            'strip_accents': True
                        })
print(result)
```

```
{'query_name': 'European american names(5) and African american names(5) wrt Pleasant(5)␣
↪and Unpleasant(5)',
 'result': 3.7529151,
 'weat': 3.7529151,
 'effect_size': 1.2746819,
 'p-value': None}
```

3.2 Running multiple Queries

We usually want to test several queries that study some criterion of bias: gender, ethnicity, religion, politics, socioeconomic, among others. Let's suppose you've created 20 queries that study gender bias on different models of embeddings. Trying to use `run_query` on each pair embedding-query can be a bit complex and will 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 will be executed using `WEAT` metric.

```
from wefe.query import Query
from wefe.word_embedding_model import WordEmbeddingModel
from wefe.metrics import WEAT, RNSB

from wefe.datasets import load_weat
```

(continues on next page)

(continued from previous page)

```
from wefe.utils import run_queries

import gensim.downloader as api
```

3.2.1 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]
```

3.2.2 Load the word sets and create the queries

Now, we will 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]
```

3.2.3 Run the queries on all Word Embeddings using WEAT.

Now, 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.
```

model_name	Male terms and Female terms wrt Career and Family	Male terms and Female terms wrt Science and Arts	Male terms and Female terms wrt Math and Arts
glove twitter dim=25	0.316584	nan	-0.0221328
glove twitter dim=50	0.363743	nan	-0.272334
glove twitter dim=100	0.385352	nan	-0.0825434

Warning: If more than 20% (by default) of the words from any of the word sets of the query are not included in the word embedding model, the metric will return `Nan`. This behavior can be changed using a float number parameter called `lost_vocabulary_threshold`.

3.2.4 Setting metric params

As you can see from the results above, 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`. Therefore, all those queries return `np.nan`. In this case, it would be very useful to use the word preprocessors seen above.

When we use `run_queries`, we can also provide specific parameters for each metric. We can do this by passing a `dict` with the metric params to the `metric_params` parameter. In this case, we will use `preprocessor_args` to lower the words.

```
WEAT_gender_results = run_queries(
    WEAT,
    gender_queries,
    models,
    metric_params={'preprocessor_args': {
        'lowercase': True
    }},
    queries_set_name='Gender Queries')

WEAT_gender_results
```

model_name	Male terms and Female terms wrt Career and Family	Male terms and Female terms wrt Science and Arts	Male terms and Female terms wrt Math and Arts
glove twitter dim=25	0.316584	0.167431	-0.0339119
glove twitter dim=50	0.363743	-0.0846904	-0.307589
glove twitter dim=100	0.385352	0.0996324	-0.15579

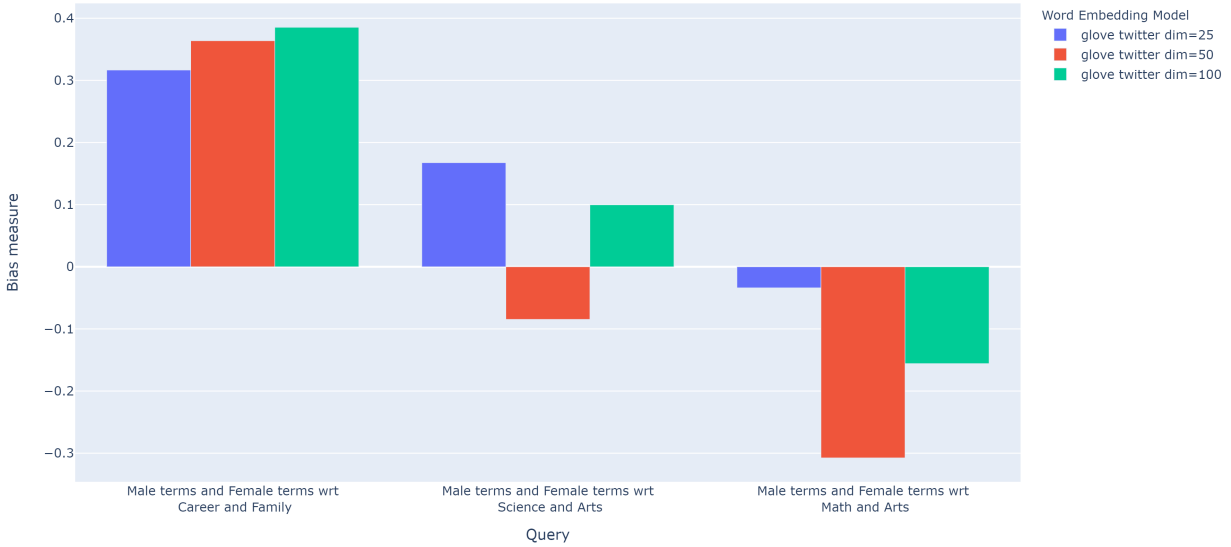
As you can see from the results table, no query was lost now.

3.2.5 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.

```
from wefe.utils import run_queries, plot_queries_results

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



3.2.6 Aggregating Results

The execution of `run_queries` in the previous step gave us many results evaluating the gender bias in the tested embeddings. However, these do not tell us much about the overall fairness of the embedding models with respect to the criteria evaluated. Therefore, we would like to have some mechanism that allows us to aggregate the results directly obtained in `run_query` so that we can evaluate the bias as a whole.

A simple way to aggregate the results would be to average their absolute values. For this, when using `run_queries`, you must set the `aggregate_results` parameter as `True`. This default value will activate the option to aggregate the results by averaging the absolute values of the results and put them in the last column.

This aggregation function can be modified through the aggregation_function parameter. Here you can specify a string that defines some of the aggregation types that are already implemented, as well as provide a function that operates in 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, for metrics returning real values (e.g., `WEAT`, `RND`, etc...), aggregation functions such as sum would make different outputs to cancel each other.

Let's aggregate the results from previous example by the average of the absolute values:

```
WEAT_gender_results_agg = run_queries(
    WEAT,
    gender_queries,
```

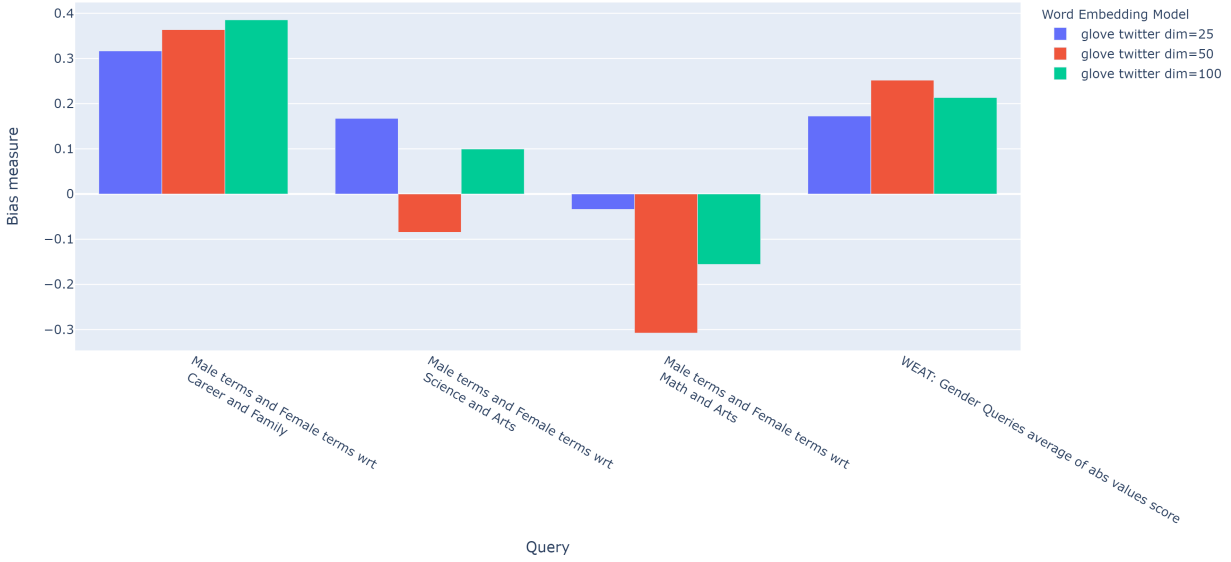
(continues on next page)

(continued from previous page)

```
    models,
    metric_params={'preprocessor_args': {
        'lowercase': True
    }},
    aggregate_results=True,
    aggregation_function='abs_avg',
    queries_set_name='Gender Queries')
WEAT_gender_results_agg
```

model_name	Male terms and Female terms wrt Career and Family	Male terms and Female terms wrt Science and Arts	Male terms and Female terms wrt Math and Arts	WEAT: Gender Queries average of abs values score
glove twitter dim=25	0.316584	0.167431	-0.0339119	0.172642
glove twitter dim=50	0.363743	-0.0846904	-0.307589	0.252008
glove twitter dim=100	0.385352	0.0996324	-0.15579	0.213591

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



Finally, we can ask the function to return only the aggregated values (through `return_only_aggregation` parameter) and then plot them.

```
WEAT_gender_results_only_agg = run_queries(
    WEAT,
    gender_queries,
    models,
```

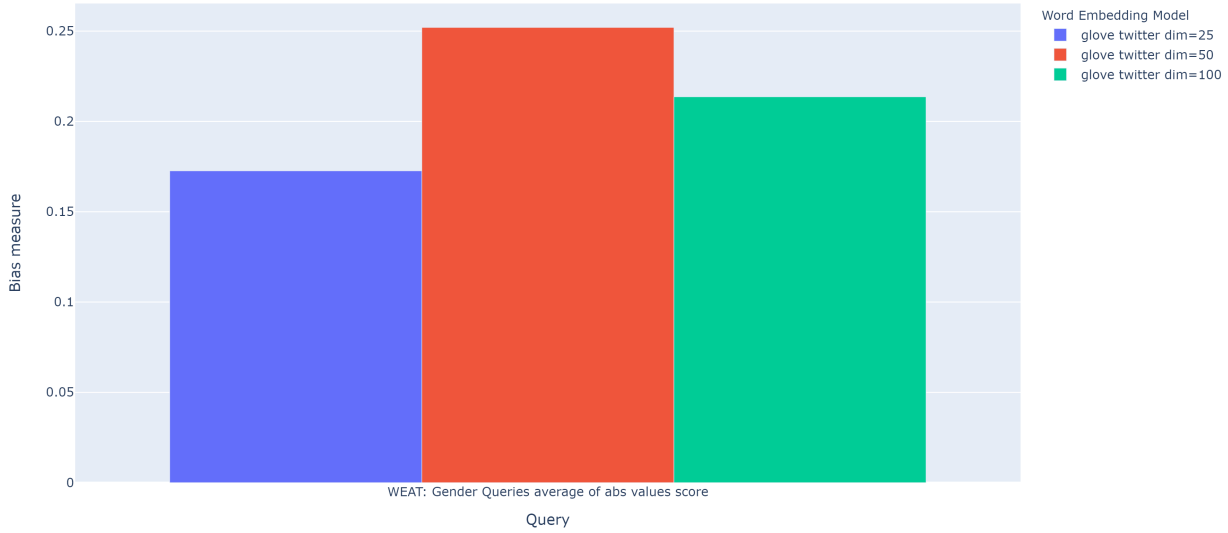
(continues on next page)

(continued from previous page)

```
    metric_params={'preprocessor_args': {
        'lowercase': True
    }},
    aggregate_results=True,
    aggregation_function='abs_avg',
    return_only_aggregation=True,
    queries_set_name='Gender Queries')
WEAT_gender_results_only_agg
```

model_name	WEAT: Gender Queries average of abs values score
glove twitter dim=25	0.172642
glove twitter dim=50	0.252008
glove twitter dim=100	0.213591

```
plot_queries_results(WEAT_gender_results_only_agg).show()
```



3.3 Calculate Rankings

When we want to measure various criteria of bias in different embedding models, two major problems arise:

1. One type of bias 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 that, suppose we have two sets of queries: one that explores gender biases and another that explores ethnicity biases, and we want to test these sets of queries on 3 Twitter Glove models of 25, 50 and 100 dimensions each, using both WEAT and Relative Negative Sentiment Bias (RNSB) as bias metrics.

1. Let's show the first problem: 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.

We executed the gender and ethnicity queries using WEAT and the 3 models mentioned above. The results obtained are:

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

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

2. For the second problem: Metrics deliver their results on different scales.

We executed the gender queries using WEAT and RNSB metrics and the 3 models mentioned above. The results obtained are:

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

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

To address these two problems, we propose to create rankings. Rankings allow us to focus on the relative differences reported by the metrics (for different models) instead of focusing on the absolute values.

Now, let's create rankings using the data used above. The following code will load the models and create the queries:

```
from wefe.query import Query
from wefe.datasets.datasets import load_weat
from wefe.word_embedding_model import WordEmbeddingModel
from wefe.metrics import WEAT, RNSB
from wefe.utils import run_queries, create_ranking, plot_ranking, plot_ranking_
↪correlations

import gensim.downloader as api

# 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]
```

(continues on next page)

(continued from previous page)

```
# 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'])

# Create ethnicity 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'])

gender_queries = [gender_query_1, gender_query_2, gender_query_3]
ethnicity_queries = [ethnicity_query_1, ethnicity_query_2]
```

Now, we will run the queries with `WEAT`, `WEAT Effect Size` and `RNSB`:

```
# Run the queries WEAT
WEAT_gender_results = run_queries(
    WEAT,
    gender_queries,
    models,
    metric_params={'preprocessor_args': {
        'lowercase': True
    }},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name='Gender Queries')

WEAT_ethnicity_results = run_queries(
    WEAT,
    ethnicity_queries,
    models,
    metric_params={'preprocessor_args': {
        'lowercase': True
    }},
    aggregate_results=True,
```

(continues on next page)

(continued from previous page)

```
    return_only_aggregation=True,
    queries_set_name='Ethnicity Queries')

# Run the queries with WEAT Effect Size

WEAT_EZ_gender_results = run_queries(WEAT,
                                     gender_queries,
                                     models,
                                     metric_params={
                                         'preprocessor_args': {
                                             'lowercase': True
                                         },
                                         'return_effect_size': True
                                     },
                                     aggregate_results=True,
                                     return_only_aggregation=True,
                                     queries_set_name='Gender Queries')

WEAT_EZ_ethnicity_results = run_queries(WEAT,
                                     ethnicity_queries,
                                     models,
                                     metric_params={
                                         'preprocessor_args': {
                                             'lowercase': True
                                         },
                                         'return_effect_size': True
                                     },
                                     aggregate_results=True,
                                     return_only_aggregation=True,
                                     queries_set_name='Ethnicity Queries')

# Run the queries using RNSB
RNSB_gender_results = run_queries(
    RNSB,
    gender_queries,
    models,
    metric_params={'preprocessor_args': {
        'lowercase': True
    }},
    aggregate_results=True,
    return_only_aggregation=True,
    queries_set_name='Gender Queries')

RNSB_ethnicity_results = run_queries(
    RNSB,
    ethnicity_queries,
    models,
    metric_params={'preprocessor_args': {
        'lowercase': True
    }},
    aggregate_results=True,
    return_only_aggregation=True,
```

(continues on next page)

(continued from previous page)

```
    queries_set_name='Ethnicity Queries')
```

To create the ranking we'll use the `create_ranking` function. This function takes all the DataFrames containing the calculated scores and uses the last column to create the ranking. It assumes that there is an aggregation in this column.

```
from wefe.utils import run_queries, create_ranking, plot_ranking, plot_ranking_
↪correlations


gender_ranking = create_ranking([
    WEAT_gender_results, WEAT_EZ_gender_results, RNSB_gender_results
])

gender_ranking
```

model_name	WEAT: Gender Queries average of abs values score (1)	WEAT: Gender Queries average of abs values score (2)	RNSB: Gender Queries average of abs values score
glove twitter dim=25	1	1	3
glove twitter dim=50	3	2	2
glove twitter dim=100	2	3	1

```
ethnicity_ranking = create_ranking([
    WEAT_ethnicity_results, WEAT_EZ_gender_results, RNSB_ethnicity_results
])

ethnicity_ranking
```

model_name	WEAT: Ethnicity Queries average of abs values score	WEAT: Gender Queries average of abs values score	RNSB: Ethnicity Queries average of abs values score
glove twitter dim=25	3	1	3
glove twitter dim=50	2	2	2
glove twitter dim=100	1	3	1

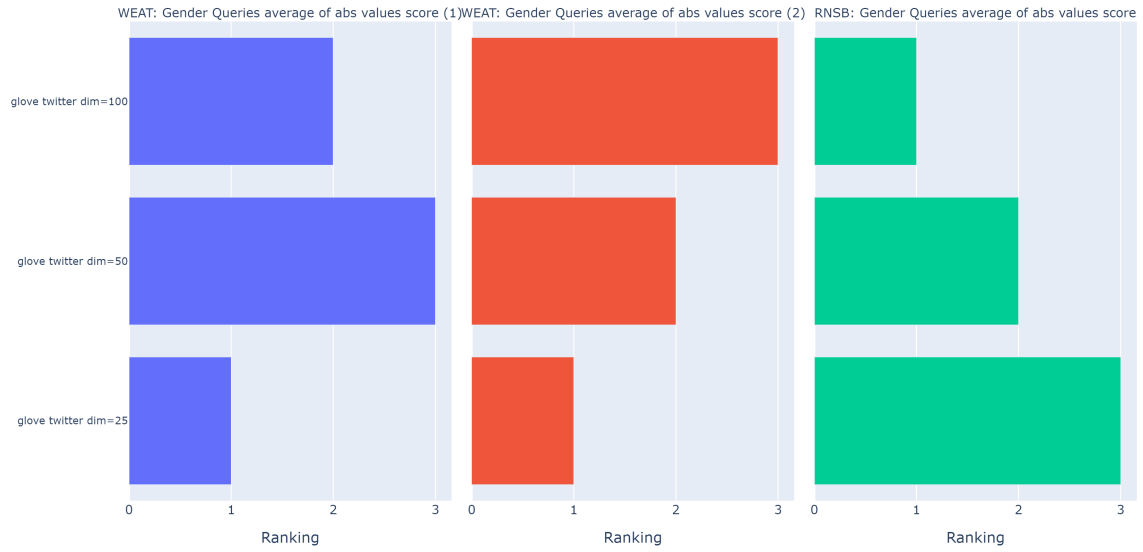
3.3.1 Plotting the rankings

Finally, we can plot the rankings in barplots using the `plot_ranking` function. The function can be used in two ways:

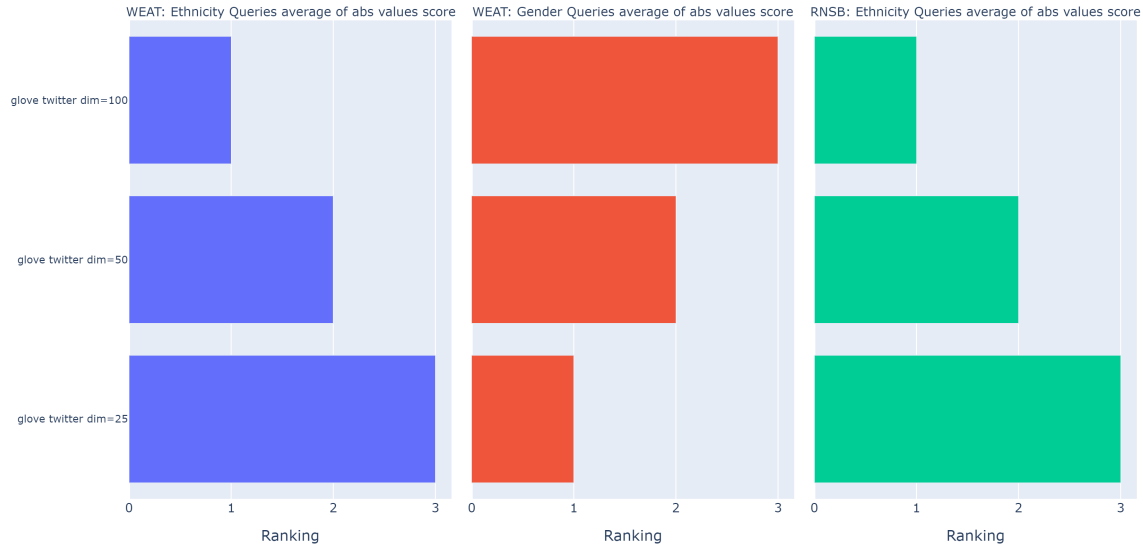
With facet by Metric and Criteria:

This image shows the rankings separated by each bias criteria and metric (i.e, by each column). Each bar represents the position of the embedding in the corresponding criterion-metric ranking.

```
plot_ranking(gender_ranking, use_metric_as_facet=True)
```



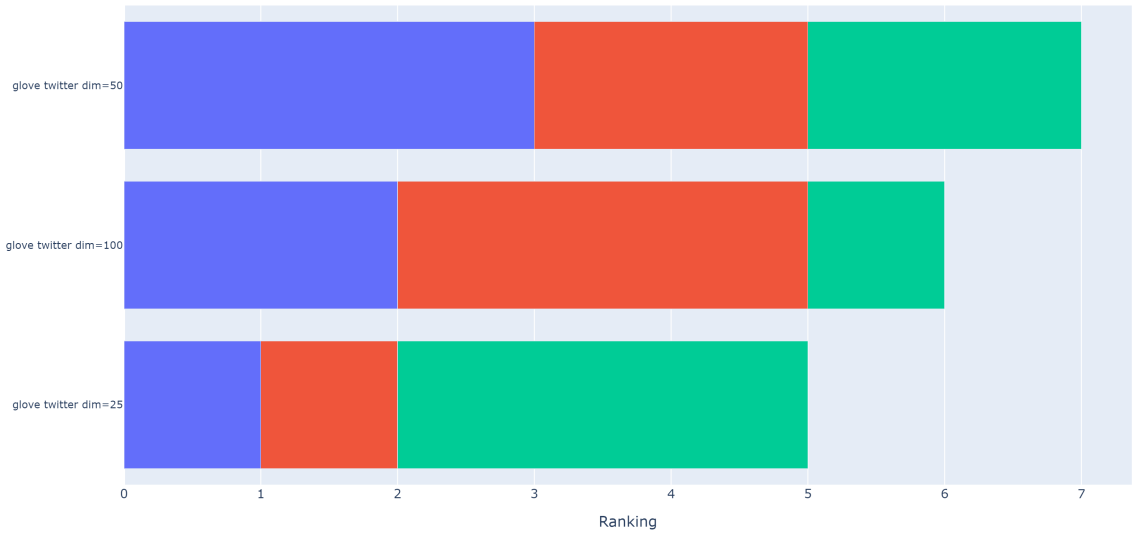

```
plot_ranking(ethnicity_ranking, use_metric_as_facet=True)
```



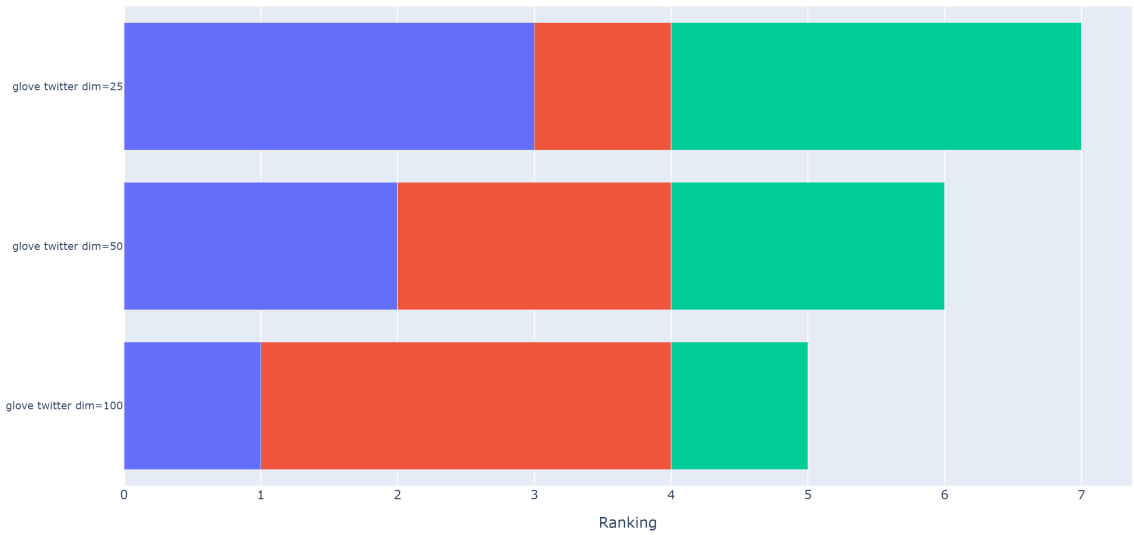
Without facet

This image 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.

```
plot_ranking(gender_ranking)
```




```
plot_ranking(ethnicity_ranking)
```



3.4 Ranking Correlations

We can see how the rankings obtained in the previous section relate to each other by using a correlation matrix. To do this we provide a function called `calculate_ranking_correlations`. This function takes the rankings as input and calculates the Spearman correlation between them.

```
from wefe.utils import calculate_ranking_correlations, plot_ranking_correlations
correlations = calculate_ranking_correlations(gender_ranking)
correlations
```

Model	WEAT: Gender Queries average of abs values score (1)	WEAT: Gender Queries average of abs values score (2)	RNSB: Gender Queries average of abs values score
WEAT: Gender Queries average of abs values score (1)	1	0.5	-0.5
WEAT: Gender Queries average of abs values score (2)	0.5	1	-1
RNSB: Gender Queries average of abs values score	-0.5	-1	1

This function uses the corr() method of the ranking dataframe. This allows you to change the correlation calculation method to: 'pearson', 'spearman', 'kendall'.

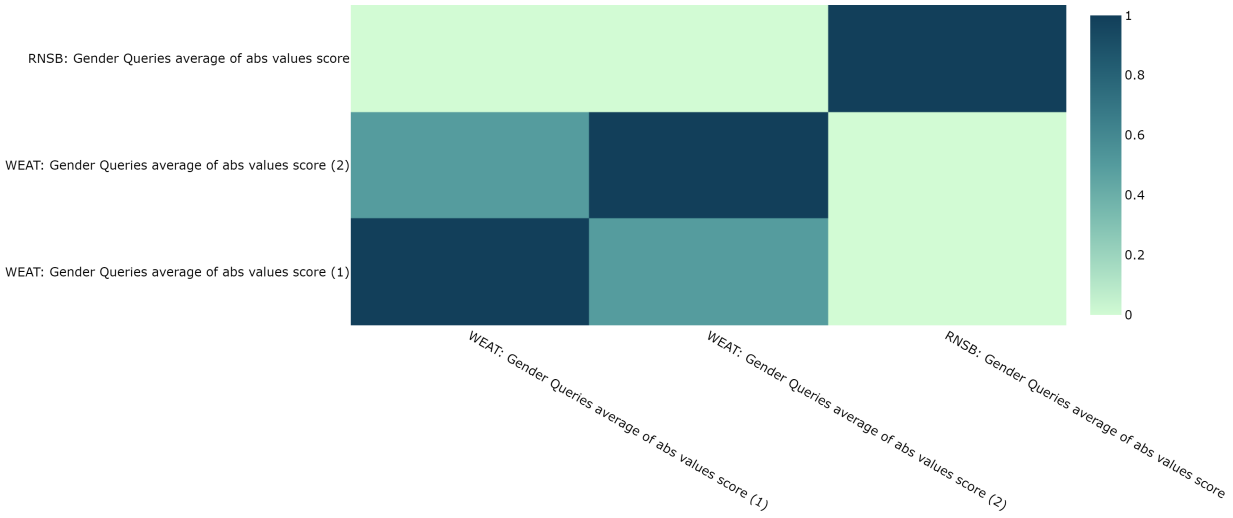
In the following example we use the kendall correlation.

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

Model	WEAT: Gender Queries average of abs values score (1)	WEAT: Gender Queries average of abs values score (2)	RNSB: Gender Queries average of abs values score
WEAT: Gender Queries average of abs values score (1)	1	0.333333	-0.333333
WEAT: Gender Queries average of abs values score (2)	0.333333	1	-1
RNSB: Gender Queries average of abs values score	-0.333333	-1	1

Finally, we also provide a function to graph the correlations. This function enables us to visually analyze how the rankings relate to each other.

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


RNSB: Gender Queries average of abs values score
WEAT: Gender Queries average of abs values score (2)
WEAT: Gender Queries average of abs values score (1)
1
0.8
0.6
0.4
0.2
0
WEAT: Gender Queries average of abs values score (1)
WEAT: Gender Queries average of abs values score (2)
RNSB: Gender Queries average of abs values score

CHAPTER

FOUR

HOW TO IMPLEMENT YOUR OWN METRIC

The following guide will show you how to implement a metric using WEFE. You can find a notebook version of this tutorial at the following link.

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 tagret 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 set 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'
```

4.2 Implement run_query method

The second step is to implement run_query method. This method is in charge of storing 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:

4.2.1 Validate the parameters:

To do this, execute the function :code:run_query from the BaseMetric class. This call checks the 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 (only the defaults).
super().run_query(query, word_embedding, lost_vocabulary_threshold,
                  preprocessor_args, secondary_preprocessor_args,
                  warn_not_found_words, *args, **kwargs)
```

4.2.2 Transform the Query to Embeddings.

This call transforms all the word sets of a query into embeddings.

```
# transform query word sets into embeddings
embeddings = word_embedding.get_embeddings_from_query(
    query=query,
    lost_vocabulary_threshold=lost_vocabulary_threshold,
    preprocessor_args=preprocessor_args,
    secondary_preprocessor_args=secondary_preprocessor_args,
    warn_not_found_words=warn_not_found_words)
```

This step could return either:

- None None if for at least one of the word sets in the query there are more words without embedding vector than those specified in the lost_vocabulary_threshold parameter (specified as percentage float).
- A tuple otherwise. This tuple contains two values:
  - A dictionary that maps each target set name to a dictionary containing its words and embeddings.
  - A dictionary that maps each attribute set name to a dictionary containing its words and embeddings.

We can illustrate what the outputs of the previous transformation look like using the following query:

```
from wefe.word_embedding_model import WordEmbeddingModel
from wefe.query import Query
from wefe.utils import load_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']
weapons = weat['weapons']
pleasant = weat['pleasant_5']
```

(continues on next page)

(continued from previous page)

```
 query = Query([flowers, weapons], [pleasant],
               ['Flowers', 'Weapons'], ['Pleasant'])

embeddings = model.get_embeddings_from_query(query=query)

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 the words and not just returning the embeddings is because that there are some metrics that can calculate per-word measurements and deliver useful information from these.

Using the above, you can already implement the run_query method

```
from typing import Any, Dict, Union

import numpy as np

from wefe.metrics.base_metric import BaseMetric
from wefe.query import Query
from wefe.word_embedding_model import WordEmbeddingModel, PreprocessorArgs


class ExampleMetric(BaseMetric):

```

(continues on next page)

(continued from previous page)

```
    # 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,
                  word_embedding: WordEmbeddingModel,
                  # any parameter that you need
                  # ...,
                  lost_vocabulary_threshold: float = 0.2,
                  preprocessor_args: PreprocessorArgs = {
                      'strip_accents': False,
                      'lowercase': False,
                      'preprocessor': None,
                  },
                  secondary_preprocessor_args: PreprocessorArgs = None,
                  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

        secondary_preprocessor_args : PreprocessorArgs, optional
            Dictionary with the arguments that specify how the pre-processing of the
            words will be done, by default {}
            The possible arguments for the function are:
            - lowercase: bool. Indicates if the words are transformed to lowercase.
            - 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 { 'strip_accents': False, 'lowercase': False, 'preprocessor': None,␣
↪}
```

(continues on next page)

(continued from previous page)

```
        secondary_preprocessor_args : PreprocessorArgs, optional
            Dictionary with the arguments that specify how the secondary pre-processing
            of the words will be done, by default None.
            Indicates that in case a word is not found in the model's vocabulary
            (using the default preprocessor or specified in preprocessor_args),
            the function performs a second search for that word using the preprocessor
            specified in this parameter.

        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.
        """
        # check the types of the provided arguments (only the defaults).
        super().run_query(query, word_embedding, lost_vocabulary_threshold,
                          preprocessor_args, secondary_preprocessor_args,
                          warn_not_found_words, *args, **kwargs)

        # transform query word sets into embeddings
        embeddings = word_embedding.get_embeddings_from_query(
            query=query,
            lost_vocabulary_threshold=lost_vocabulary_threshold,
            preprocessor_args=preprocessor_args,
            secondary_preprocessor_args=secondary_preprocessor_args,
            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())
```

(continues on next page)

(continued from previous page)

```
        """
        # From here, the code can vary quite a bit depending on what you need.
        # metric operations. It is recommended to calculate it in another method(s).
        results = calc_metric()

        # You must return query and result.
        # However, you can return other calculated metrics, 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),
                ...
            }
        """
```

This is what the transformed :code:`target_embeddings_dict` would look like:

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 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

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'
```

(continues on next page)

(continued from previous page)

```
    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.
         """

        # 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,
            word_embedding: WordEmbeddingModel,
            # any parameter that you need
            # ...,
            lost_vocabulary_threshold: float = 0.2,
            preprocessor_args: PreprocessorArgs = {
                'strip_accents': False,
                'lowercase': False,
                'preprocessor': None,
```

(continues on next page)

(continued from previous page)

```
            },
            secondary_preprocessor_args: PreprocessorArgs = None,
            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

        secondary_preprocessor_args : PreprocessorArgs, optional
            A dictionary with the arguments that specify how the pre-processing of the
            words will be done, by default {}
            The possible arguments for the function are:
            - lowercase: bool. Indicates if the words are transformed to lowercase.
            - 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 { 'strip_accents': False, 'lowercase': False, 'preprocessor': None,␣
↪}

        secondary_preprocessor_args : PreprocessorArgs, optional
            A dictionary with the arguments that specify how the secondary pre-processing
            of the words will be done, by default None.
            Indicates that in case a word is not found in the model's vocabulary
            (using the default preprocessor or specified in preprocessor_args),
            the function performs a second search for that word using the preprocessor
            specified in this parameter.

        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
```

(continues on next page)

(continued from previous page)

```
        -------
        Dict[str, Any]
            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).
        super().run_query(query, word_embedding, lost_vocabulary_threshold,
                          preprocessor_args, secondary_preprocessor_args,
                          warn_not_found_words, *args, **kwargs)

        # transform query word sets into embeddings
        embeddings = word_embedding.get_embeddings_from_query(
            query=query,
            lost_vocabulary_threshold=lost_vocabulary_threshold,
            preprocessor_args=preprocessor_args,
            secondary_preprocessor_args=secondary_preprocessor_args,
            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

        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's 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', '')
```

(continues on next page)

(continued from previous page)

```
flowers = weat['flowers']
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.

4.4 Contribute

If you want to contribute your own metric, please follow the conventions, document everything, create specific tests for the metric, and make a pull request to the project's Github repository. We would really appreciate it!

You can visit the Contributing section for more information.

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 import
>>> 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 = (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's 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 = (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 = (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 = (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:

- Implementing new metrics. A relatively extensive guide can be found in the section Creating your own metrics.
- Create more examples and use cases.
- Help to 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 follows 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
```

CHAPTER

SEVEN

WEFE API

This is the documentation of the API of WEFE.

7.1 WordEmbeddingModel

WordEmbeddingModel(model[, model_name, ...])	A container for Word Embedding pre-trained models.

7.1.1 wefe.WordEmbeddingModel

class wefe.**WordEmbeddingModel**(*model: gensim.models.keyedvectors.KeyedVectors*, *model_name: Optional[str] = None*, *vocab_prefix: Optional[str] = None*)

A container 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__(*model: gensim.models.keyedvectors.KeyedVectors*, *model_name: Optional[str] = None*, *vocab_prefix: Optional[str] = None*)

Initializes the container.

Parameters

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

model_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 instance of str.

TypeError if vocab_prefix is not None and not 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.model_name)
Dummy model dim=10
>>> print(model.vocab_prefix)
/en/
```

Attributes

model [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.

get_embeddings_from_query(*query:* wefe.query.Query, *lost_vocabulary_threshold: float = 0.2*, *preprocessor_args: Dict[str, Optional[Union[bool, str, Callable]]] = {}*, *secondary_preprocessor_args: Optional[Dict[str, Optional[Union[bool, str, Callable]]]] = None*, *warn_not_found_words: 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 pre-processing 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.

preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the pre-processing of the words will be done, by default {} The possible arguments for the function are: - lowercase: bool. Indicates if the words are transformed to lowercase. - 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).

secondary_preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the secondary pre-processing of the words will be done, by default None. Indicates that in case a word is not found in the model's vocabulary (using the default preprocessor or specified in preprocessor_args), the function performs a second search for that word using the preprocessor specified in this parameter. Example: Suppose we have the word "John" in the query and only the lowercase version "john" is found in the model's vocabulary. If we use preprocessor_args by default (so as not to affect the search for other words that may exist in capital letters in the model), the function will not be able to extract the representation of "john" even if it exists in lower case. However, we can use {'lowecase' : True} in preprocessor_args to specify that it also looks for the lower case version of "juan", without affecting the first preprocessor. Thus, this preprocessor will only remain as an alternative in case the first one does not work.

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.

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.

Raises

TypeError If query is not an instance of Query

TypeError If lost_vocabulary_threshold is not float

TypeError If preprocessor_args is not a dictionary

TypeError If secondary_preprocessor_args is not a dictionary

TypeError If warn_not_found_words is not a boolean

`get_embeddings_from_word_set`(*word_set: List[str]*, *preprocessor_args: Dict[str, Optional[Union[bool, str, Callable]]] = {}*, *secondary_preprocessor_args: Optional[Dict[str, Optional[Union[bool, str, Callable]]]] = None*) → Tuple[List[str], Dict[str, numpy.ndarray]]

Transforms a set of words into their respective embeddings and discard out words that are not in the model's vocabulary (according to the rules specified in the preprocessors).

Parameters

word_set [List[str]] The list/array with the words that will be transformed

preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the pre-processing of the words will be done, by default {} The options for the dict are: - lowercase: bool. Indicates if the words are transformed to lowercase. - 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).

secondary_preprocessor_args [PreprocessorArgs, optional] Dictionary with arguments for pre-processing words (same as the previous parameter), by default None. Indicates that in case a word is not found in the model's vocabulary (using the default preprocessor or specified in preprocessor_args), the function performs a second search for that word using the preprocessor specified in this parameter.

Returns

Tuple[List[str], Dict[str, np.ndarray]] A tuple with a list of missing words and a dictionary that maps words to 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 BaseMetric

metrics.BaseMetric()	A base class to implement any metric following the framework described by WEFE.

7.3.1 wefe.metrics.BaseMetric

class wefe.metrics.**BaseMetric**
A base class to implement any metric following the framework described by WEFE.

It contains the name of the metric, the templates (cardinalities) that it supports and the abstract function run_query, which must be implemented by any metric that extends this class.

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

metric_name: str

metric_short_name: str

metric_template: Tuple[Union[int, str], Union[int, str]]

abstract run_query(*query:* wefe.query.Query, *word_embedding:* wefe.word_embedding_model.WordEmbeddingModel, *lost_vocabulary_threshold: float = 0.2*, *preprocessor_options: Dict[str, Optional[Union[bool, str, Callable]]] = {'lowercase': False, 'preprocessor': None, 'strip_accents': False}*, *secondary_preprocessor_options: Optional[Dict[str, Optional[Union[bool, str, Callable]]]] = None*, *warn_not_found_words: bool = False*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

7.4 WEAT

WEAT()	A implementation of Word Embedding Association Test (WEAT).

7.4.1 wefe.WEAT

class wefe.**WEAT**

A implementation of Word Embedding Association Test (WEAT).

It measures the degree of association between two sets of target words and two sets of attribute words through a permutation test.

References

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*)

metric_name: str = 'Word Embedding Association Test'

metric_short_name: str = 'WEAT'

metric_template: Tuple[Union[int, str], Union[int, str]] = (2, 2)

run_query(*query:* wefe.query.Query, *word_embedding:* wefe.word_embedding_model.WordEmbeddingModel, *return_effect_size: bool = False*, *calculate_p_value: bool = False*, *p_value_test_type: Literal['left-sided', 'right-sided', 'two-sided'] = 'right-sided'*, *p_value_method: Literal['approximate', 'exact'] = 'approximate'*, *p_value_iterations: int = 10000*, *p_value_verbose: bool = False*, *lost_vocabulary_threshold: float = 0.2*, *preprocessor_args: Dict[str, Optional[Union[bool, str, Callable]]] = {'lowercase': False, 'preprocessor': None, 'strip_accents': False}*, *secondary_preprocessor_args: Optional[Dict[str, Optional[Union[bool, str, Callable]]]] = None*, *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 word sets to be tested.

word_embedding [WordEmbeddingModel] A WordEmbeddingModel object that contains a word embedding pretrained 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] In case of 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] In case of 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

preprocessor_args [PreprocessorArgs, optional] A dictionary with the arguments that specify how the pre-processing of the words will be done. The possible arguments for the function are: - lowercase: bool. Indicates if the words are transformed to lowercase. - 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 { 'strip_accents': False, 'lowercase': False, 'preprocessor': None, }

secondary_preprocessor_args [PreprocessorArgs, optional] A dictionary with the arguments that specify how the secondary pre-processing of the words will be done, by default None. Indicates that in case a word is not found in the model's vocabulary (using the default preprocessor or specified in preprocessor_args), the function performs a second search for that word using the preprocessor specified in this parameter.

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.

7.5 RND

RND()	A implementation of Relative Norm Distance (RND).

7.5.1 wefe.RND

class wefe.**RND**

A implementation of Relative Norm Distance (RND).

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

References

Nikhil Garg, Londa Schiebinger, Dan Ju-rafsky, 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.

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

metric_name: str = 'Relative Norm Distance'

metric_short_name: str = 'RND'

metric_template: Tuple[Union[int, str], Union[int, str]] = (2, 1)

run_query(*query:* wefe.query.Query, *word_embedding:* wefe.word_embedding_model.WordEmbeddingModel, *distance_type: str = 'norm'*, *average_distances: bool = True*, *lost_vocabulary_threshold: float = 0.2*, *preprocessor_args: Dict[str, Optional[Union[bool, str, Callable]]] = {'lowercase': False, 'preprocessor': None, 'strip_accents': False}*, *secondary_preprocessor_args: Optional[Dict[str, Optional[Union[bool, str, Callable]]]] = None*, *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 word sets for be tested.

word_embedding : A object that contain certain word embedding pretrained model.

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

average_distances [bool, optional] Specifies wheter the function averages the distances at the end of the calculations, by default True

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

preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the pre-processing of the words will be done, by default {} The possible arguments for the function are: - lowercase: bool. Indicates if the words are transformed to lowercase. - 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 { 'strip_accents': False, 'lowercase': False, 'preprocessor': None, }

secondary_preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the secondary pre-processing of the words will be done, by default None. Indicates that in case a word is not found in the model's vocabulary (using the default preprocessor or specified in preprocessor_args), the function performs a second search for that word using the preprocessor specified in this parameter.

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.

7.6 RNSB

RNSB()	A implementation of Relative Relative Negative Sentiment Bias (RNSB).

7.6.1 wefe.RNSB

class wefe.**RNSB**

A implementation of 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.

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

metric_name: str = 'Relative Negative Sentiment Bias'

metric_short_name: str = 'RNSB'

metric_template: Tuple[Union[int, str], Union[int, str]] = ('n', 2)

run_query(*query: wefe.query.Query*, *word_embedding: 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*, *preprocessor_args: Dict[str, Optional[Union[bool, str, Callable]]] = {'lowercase': False, 'preprocessor': None, 'strip_accents': False}*, *secondary_preprocessor_args: Optional[Dict[str, Optional[Union[bool, str, Callable]]]] = None*, *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.

word_embedding [WordEmbeddingModel] A WordEmbeddingModel object that contains certain word embedding pretrained 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 allows to make 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

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

preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the pre-processing of the words will be done, by default {} The possible arguments for the function are: - lowercase: bool. Indicates if the words are transformed to lowercase. - 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 { 'strip_accents': False, 'lowercase': False, 'preprocessor': None, }

secondary_preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the secondary pre-processing of the words will be done, by default None. Indicates that in case a word is not found in the model's vocabulary (using the default preprocessor or specified in preprocessor_args), the function performs a second search for that word using the preprocessor specified in this parameter.

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.

7.7 ECT

ECT()	An implementation of the Embedding Coherence Test.

7.7.1 wefe.ECT

class wefe.**ECT**

An implementation of the Embedding Coherence Test.

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

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

1. Embedd 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*)

metric_name: str = 'Embedding Coherence Test'

metric_short_name: str = 'ECT'

metric_template: Tuple[Union[int, str], Union[int, str]] = (2, 1)

run_query(*query:* wefe.query.Query, *word_embedding:* wefe.word_embedding_model.WordEmbeddingModel, *lost_vocabulary_threshold: float = 0.2*, *preprocessor_args: Dict[str, Optional[Union[bool, str, Callable]]] = {'lowercase': False, 'preprocessor': None, 'strip_accents': False}*, *secondary_preprocessor_args: Optional[Dict[str, Optional[Union[bool, str, Callable]]]] = None*, *warn_not_found_words: bool = False*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

Runs 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.

word_embedding : A 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

preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the pre-processing of the words will be done, by default {} The possible arguments for the function are: - lowercase: bool. Indicates if the words are transformed to lowercase. - 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 { 'strip_accents': False, 'lowercase': False, 'preprocessor': None, }

secondary_preprocessor_args [PreprocessorArgs, optional] Dictionary with the arguments that specify how the secondary pre-processing of the words will be done, by default None. Indicates that in case a word is not found in the model's vocabulary (using the default preprocessor or specified in preprocessor_args), the function performs a second search for that word using the preprocessor specified in this parameter.

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.

7.8 RIPA

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

7.8.1 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*)

metric_name: str = 'Relational Inner Product Association'

metric_short_name: str = 'RIPA'

metric_template: Tuple[Union[int, str], Union[int, str]] = (2, 1)

run_query(*query:* wefe.query.Query, *word_embedding:* wefe.word_embedding_model.WordEmbeddingModel, *lost_vocabulary_threshold: float = 0.2*, *preprocessor_args: Dict[str, Optional[Union[bool, str, Callable]]] = {'lowercase': False, 'preprocessor': None, 'strip_accents': False}*, *secondary_preprocessor_args: Optional[Dict[str, Optional[Union[bool, str, Callable]]]] = None*, *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

secondary_preprocessor_args [PreprocessorArgs, optional] A dictionary with the arguments that specify how the pre-processing of the words will be done, by default {} The possible arguments for the function are: - lowercase: bool. Indicates if the words are transformed to lowercase. - 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 { 'strip_accents': False, 'lowercase': False, 'preprocessor': None, }

secondary_preprocessor_args [PreprocessorArgs, optional] A dictionary with the arguments that specify how the secondary pre-processing of the words will be done, by default None. Indicates that in case a word is not found in the model's vocabulary (using the default preprocessor or specified in preprocessor_args), the function performs a second search for that word using the preprocessor specified in this parameter.

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.

7.9 Dataloaders

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

7.9.1 Load BingLiu

load_bingliu()	Load the bing-liu sentiment lexicon.

wefe.load_bingliu

wefe.**load_bingliu**()

Load the bing-liu sentiment lexicon.

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.

Returns

dict A dictionary with the positive and negative words.

7.9.2 Fetch Debias Multiclass Word sets

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

wefe.fetch_debias_multiclass

wefe.**fetch_debias_multiclass**() → dict

Fetch the word sets used in the paper *Black Is To Criminals Caucasian Is To Police: Detecting And Removing Multiclass Bias In Word Embeddings*. It includes gender (male, female), ethnicity(asian, black, white) and religion(christianity, judaism and islam) target and attribute word sets.

References: Thomas Manzini, Lim Yao Chong,Alan W Black, and Yulia Tsvetkov. Black is to criminals 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. As-sociation for Computational Linguistics.

Returns

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

7.9.3 Fetch Debias Word Embedding Word Sets

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

wefe.fetch_debiaswe

wefe.**fetch_debiaswe**() → dict

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.

Reference: 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.

Returns

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

7.9.4 Fetch Embedding Dynamic Stereotypes Word Sets

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*.

wefe.fetch_eds

wefe.**fetch_eds**(*occupations_year: int = 2015*, *top_n_race_occupations: int = 15*) → dict

Fetch the word sets used in the experiments of the work *Word Embeddings *Quantify 100 Years Of Gender And Ethnic Stereotypes*. It includes gender (male, female), ethnicity (asian, black, white) and religion(christianity and islam) and adjetives (appearence, intelligence, otherization, sensitive) word sets.

Reference: 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.

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.

7.9.5 Load Word Embedding Association Test Word Sets

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

wefe.load_weat

wefe.**load_weat**()

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.

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

Returns

word_sets_dict [dict] A dictionary with the word sets.

CHAPTER

EIGHT

REPLICATION OF PREVIOUS STUDIES

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

Below we list some examples:

8.1 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.

8.2 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 Associationfor Computational Linguistics, pages 1662–1667, 2019.

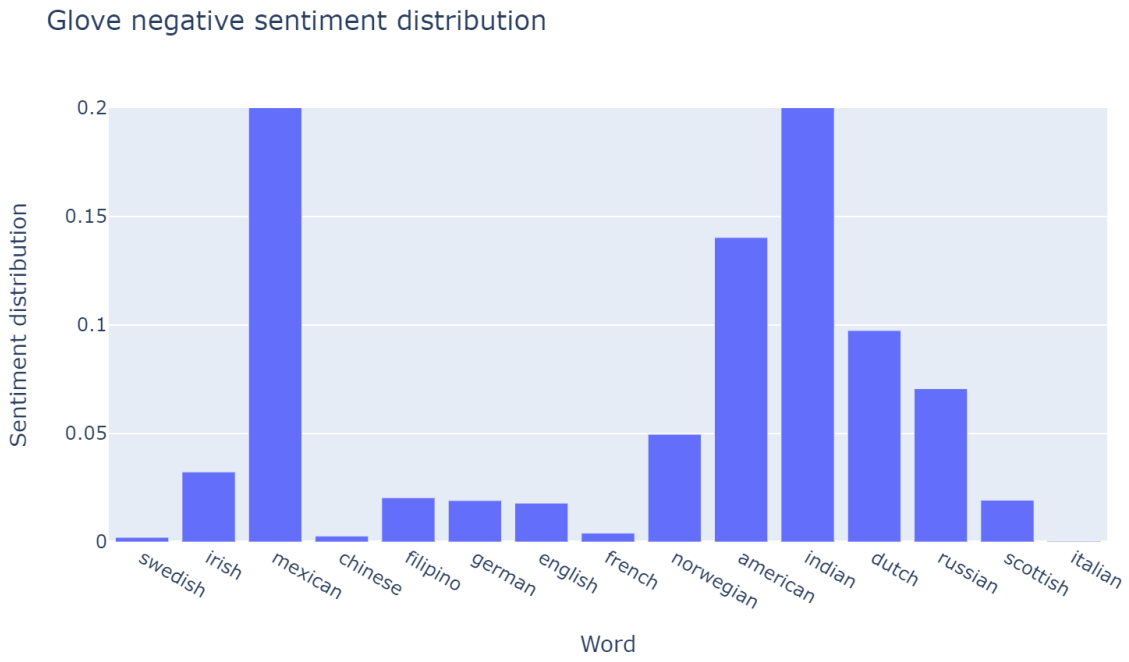
Note: Due to the formulation of the metric (it trains a logistic regression in each execution) our results are not exactly the same as those reported in the original paper. However, our results are still very similar to those in the original paper.

```
>>> 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.
```

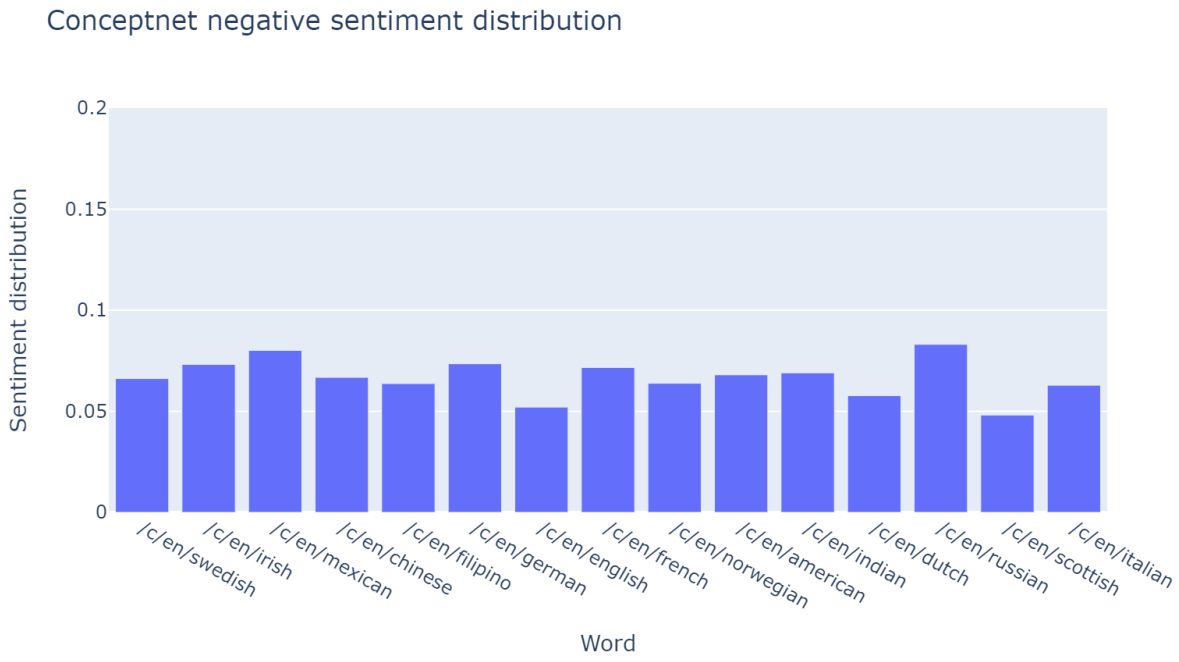
(continues on next page)

(continued from previous page)

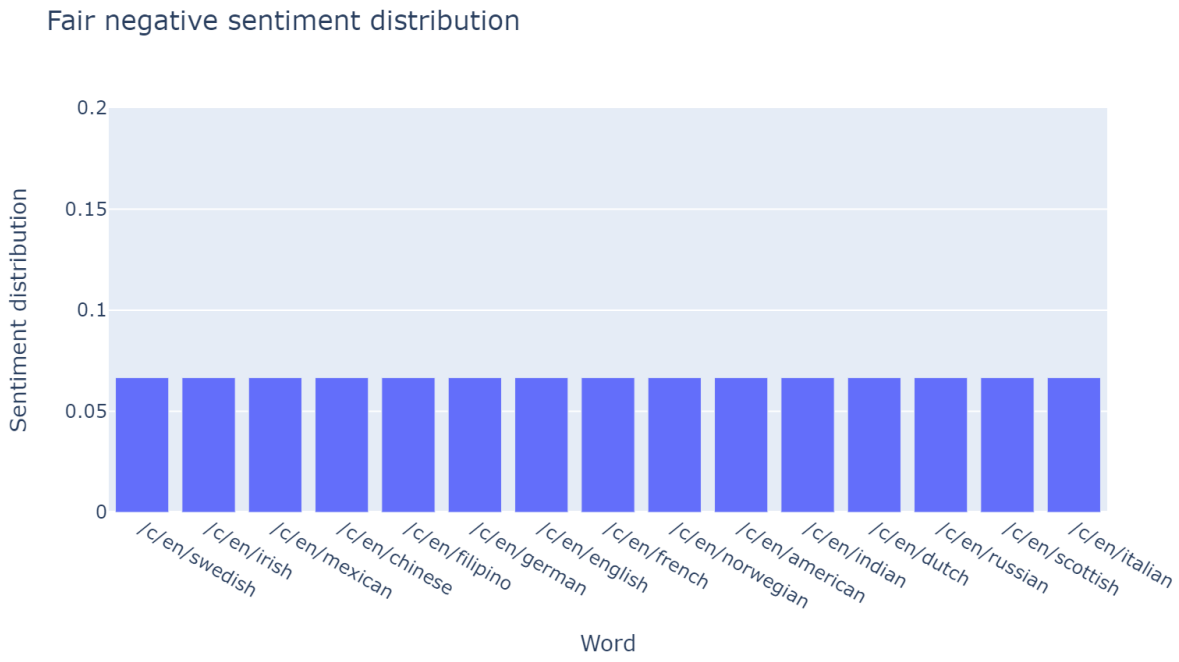
```
>>> # In this case each word is an objective set because each of them represents a␣
↪different social group.
>>> RNSB_words = [
>>>     ['swedish'], ['irish'], ['mexican'], ['chinese'], ['filipino'], ['german'], [
↪'english'],
>>>     ['french'], ['norwegian'], ['american'], ['indian'], ['dutch'],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

CHAPTER

NINE

RANK WORD EMBEDDINGS FAIRNESS USING SEVERAL METRICS AND QUERIES

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

TEN

REPOSITORY

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

INDEX

Symbols

B

E

F

G

L

M

Q

R

W