Text Similarity

By Nalani Schumacher

Table of Contents

Introduction

The textsimilarity Python package contains functionality for text pre-processing and similarity matching between a predefined corpus of words/short phrases and target words/short phrases. This package uses large pre-trained natural language models that provide contextual and semantic text embeddings which allow for text comparison. This package has modules and classes which allow language models from the transformers library to be loaded and used for comparison without having to think about the complexity of what an embedding is, how to get them, or how to use them. Additionally, the text cleaning functionality provides a convenient way to correct spelling errors and check for profanity before applying the similarity comparison. This package wraps supporting packages, modules, and methods for ease of use and simplicity.

The package is on GitHub:
NalaniKai/TextSimilarity at 35fdb51be58899ece17d8d7c3941b4c65d945d5a (github.com)

To install this package, you can use the following command:
pip install git+https://github.com/NalaniKai/TextSimilarity

Examples of using this package can be found here:
https://github.com/NalaniKai/TextSimilarity/blob/35fdb51be58899ece17d8d7c3941b4c65d945d5a/examples/examples.ipynb

Use Cases & Users

Software engineers often have limited knowledge with machine learning and are sometimes required to apply intelligence to the product they are working on. In the scenario where software engineers are working with textual data and need to apply similarity matching, the textsimilarity package is a good starting point because the package is simple and easy to use without requiring any background knowledge of natural language modelling.

One example use case is when a product has different filters to apply to a given search. For Nordstrom Rack, the textsimilarity package could be used to show the most relevant filters when a customer types in a search query. Figure 1 below shows a customer typing in "shoes" on the left. In this case there is a predefined corpus of filters containing filters like "heel height", "heel shape", "neck style", etc. The textsimilarity package could be used to rank the similarity of each filter to the provided query and then show the filters with the highest rankings. In Figure 1, the query "shoes" returns the filters "heel height" and "heel shape." Changing the query to "jacket" on the right, the filters change to "neck style" and "sleeve length."

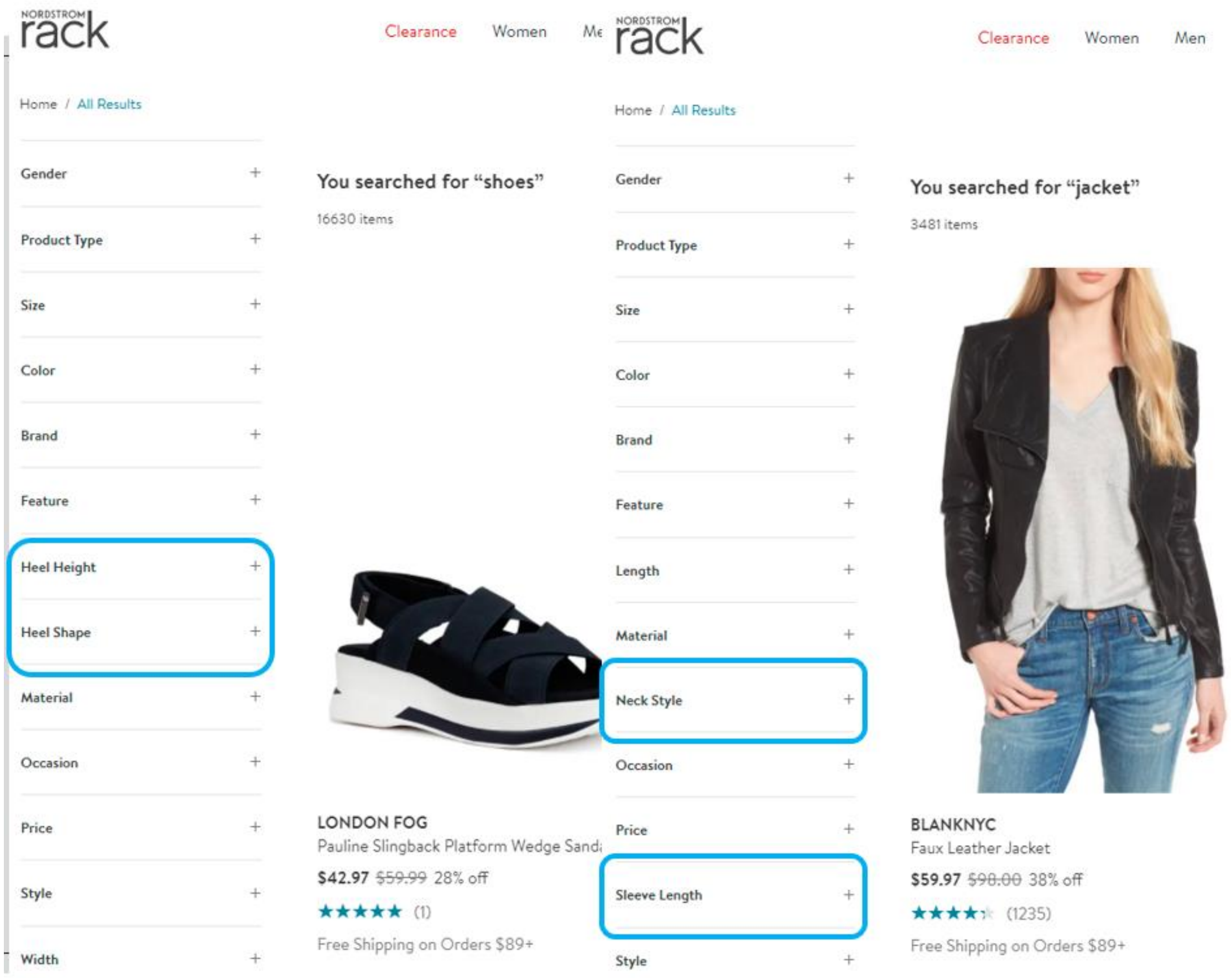


Figure 1: Nordstrom Rack search provides different filter options based on the provided search query.

Another example where textsimilarity can be used is to match a search query to different image titles. For example, Figure 2 demonstrates the similarity in image titles to the query "Korean food" to show the most relevant images of Korean food.

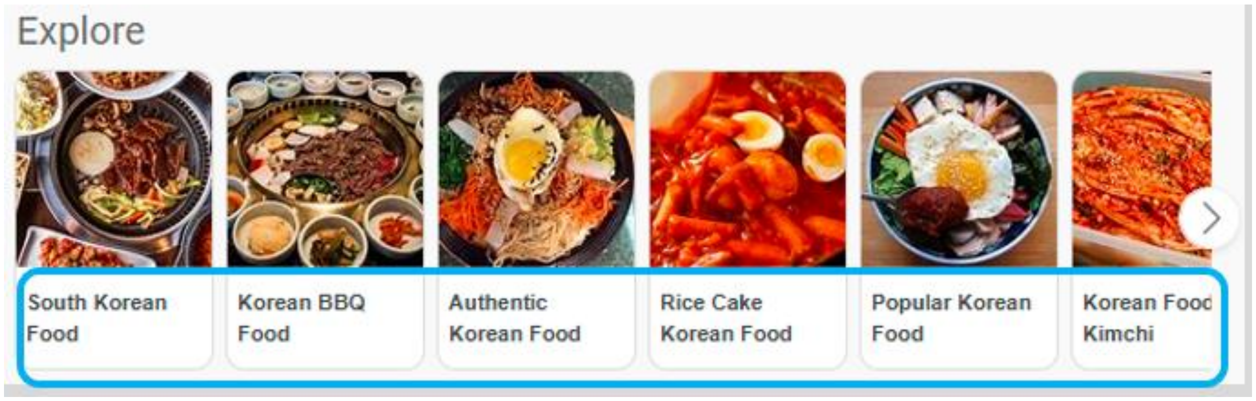


Figure 2: Search query “Korean food” brings up images with similar titles such as “Korean BBQ Food.”

Software Modules & Components

The textsimilarity package has three modules which are separated by functionality and purpose:

- clean_text
- text_models
- rankers

The clean_text module has a CleanText class which loads supporting dictionaries and instances once during initialization. This class contains the public methods spelling_correction() which corrects spelling errors and determine_text_profanity() which provides insight into whether a text contains profanity. The clean_text class also has one private method _calculate_jaccard_distance() which is a helper method for spelling_correction().

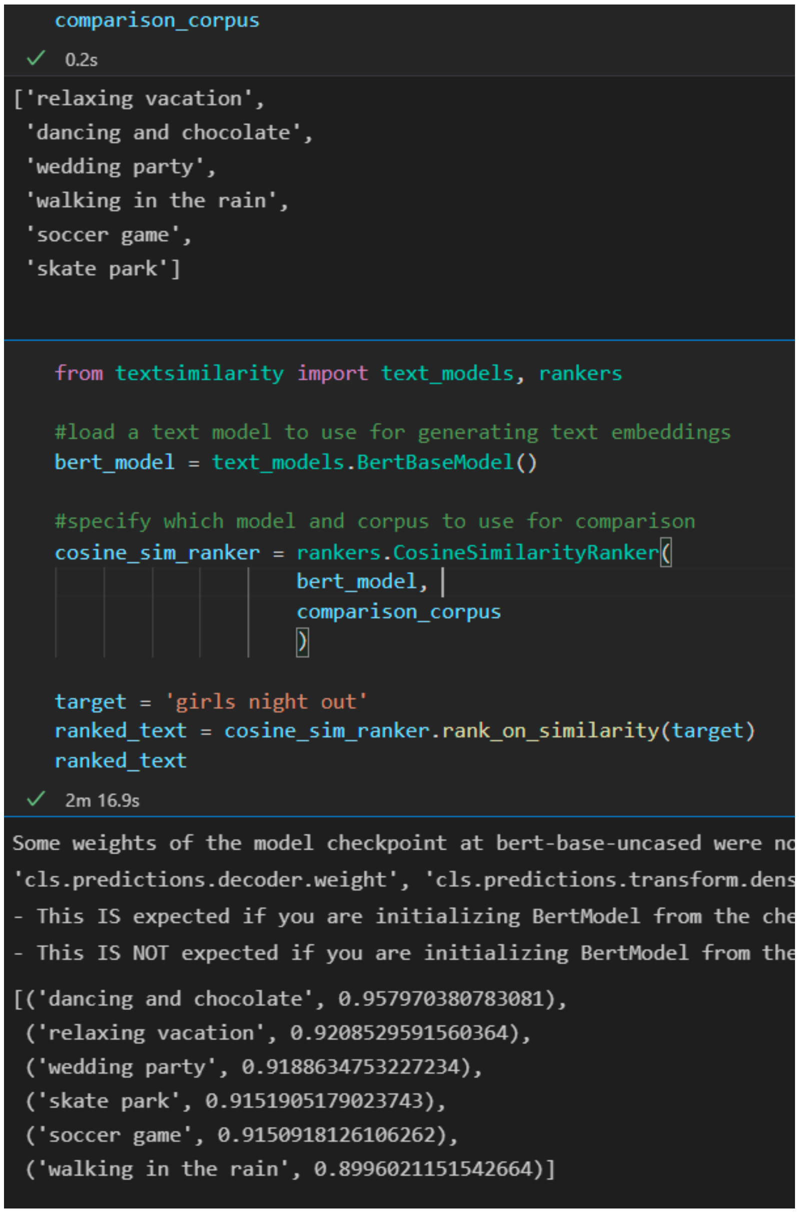
The text_models module has a BertBaseModel class which loads a pre-trained BERT model. The class contains two private methods. One is _tokenize() for tokenizing the data and the other is _get_embedding() for retrieving the text embeddings from the model.

The rankers module has a CosineSimilarityRanker class which contains two private methods and one public method. The private method_get_embeddings_dict() loads all text embeddings from a predefined corpus into a dictionary on object instantiation for quick access during runtime. The private method _calculate_cosine_similarity() calculates the similarity between two texts using the cosine of the angle between two vector embeddings. The public method rank_on_similarity() calculated the ranked list of predefined texts based on the given target text.

The textsimilarity package also has a module which contains constants used in the package as well as a tests folder which has one test script per module. Each test script uses the unittest package and contains a test class. The test classes contain methods for setting up and tearing down after each test is run to avoid any influence the tests may have on each other. There is also at least one test per method in each class.

Design Decisions

The textsimilarity package was designed to be extremely simple for ease of use and to quickly enable users to compare text. For example, the BertBaseModel class in the text_models module has pre-set parameters for initialization, so users don't need to worry about specifying the parameters to get started. Given this simplicity, Figure 3 demonstrates how a text corpus can be ranked by a target text in just a few lines of code. All a user has to do is specify a corpus as a list of texts, import the text_models and rankers modules, create instances of a model and ranker, and then the corpus can be ranked based on the given input text. Using the textsimilarity package, a user can easily compare text without having to deal with tokenization, embeddings, or similarity metrics to compare vectors.



```
    comparison_corpus
  ✓ 0.2s
['relaxing vacation',
 'dancing and chocolate',
 'wedding party',
 'walking in the rain',
 'soccer game',
 'skate park']
```

```
    from textsimilarity import text_models, rankers

    #load a text model to use for generating text embeddings
    bert_model = text_models.BertBaseModel()

    #specify which model and corpus to use for comparison
    cosine_sim_ranker = rankers.CosineSimilarityRanker(
                                    bert_model,
                                    comparison_corpus
                                    )

    target = 'girls night out'
    ranked_text = cosine_sim_ranker.rank_on_similarity(target)
    ranked_text
  ✓ 2m 16.9s
Some weights of the model checkpoint at bert-base-uncased were no
'cls.predictions.decoder.weight', 'cls.predictions.transform.dens
- This IS expected if you are initializing BertModel from the che
- This IS NOT expected if you are initializing BertModel from the

[('dancing and chocolate', 0.957970380783081),
 ('relaxing vacation', 0.9208529591560364),
 ('wedding party', 0.9188634753227234),
 ('skate park', 0.9151905179023743),
 ('soccer game', 0.9150918126106262),
 ('walking in the rain', 0.8996021151542664)]
```

Figure 3: Example of ranking a text corpus based on a given target text query.

Principles of abstraction and information hiding are leveraged throughout the package to reduce complexity and make both maintenance and modification easier. The private method _calculate_jaccard_distance() in the CleanText class hides the input transformations needed to use nltk's jaccard_distance() method in the distance module and also hides the algorithm used to support spelling correction. Additionally, code that uses spelling_correction() would remain functional when changes are made to any supporting functionality for spelling_correction() because the method call and expected output would remain the same. Further, the design decision to have the CosineSimilarityRanker take in a language model as an input parameter allows the language model complexity to remain abstracted from the user by calling the private methods in the model. This way the user does not have to think about tokenization or embeddings.

Another design decision was to separate the functionality into three separate modules. The clean_text module is separate because if a user already knows their text data is clean or if the data had already been pre-processed, then the user can save time and not use the functionality provided in the module. They can instead go straight to using the text_models and rankers modules like in Figure 3. Having separate modules for the cleaning text, language models, and rankers also follows the principle of encapsulation which enables easier maintenance, extensibility, and discoverability. The separation and naming provide a clear sense of the functionality each module contains and also reduces the amount of data shared across components.

Comparison to Existing Package

The textsimilarity package is like the semantic_sh package https://github.com/KeremZaman/semantic-sh which also enables text to text similarity using large pre-trained natural language models; however, the design approaches between the two packages are different. The semantic_sh package has only one module with one class which has all the functionality including loading text models, ranking a list of text based on a target text and removing stop words which is a text cleaning/pre-processing step. In contrast the textsimilarity package separates the functionality of loading models, ranking, and text cleaning into different modules which makes textsimilarity cleaner, easier to extend, and easier to modify. For instance, the textsimilarity package has more text cleaning capability built in with spelling correction and profanity checking. By having a separate module, additional text cleaning capability can easily be added in a cleaner manner. On the other hand, making changes to semantic_sh likely requires more time and effort. This is because everything in semantic_sh in in one .py file so there is more scrolling required to scan the code. Additionally, there are calls to methods within methods that call other methods which adds cognitive load to those trying to extend or modify the code. Further, all supporting packages and modules will be loaded regardless of whether they are used since all functionality is specified in the same file.

The textsimilarity package also differs from the semantic_sh module because there is a text_models module which is structured to have a separate class per model type and currently only supports BERT based models from the transformers library. The semantic_sh module on the other hand supports fasttext, glove, word2vec, and BERT models from the transformers library and loads the specified model in the same class. The separate class approach allows for more clarity in the model being created; however, the number of classes would grow with the number of models. The single class approach could provide more simplicity in having one class when there are only a few models, but as the number of models grows this approach makes the code harder to maintain because more if-else statements are required to check for the model type and to make the appropriate assignments. This makes the code harder to debug, maintain, and modify because the methods are longer, and it becomes harder to quickly understand what the code is doing.

The semantic_sh class contains functionality such as saving the state of the object, loading the state of an object, adding text to be compared, and comparing two text strings together rather than comparing a text string to an entire list. These functionalities would also be good to add to the textsimilarity package in CosineSimilarityRanker because they are directly applicable to the use cases the package was built for.

Extensibility

The textsimilarity package was designed to be easily extensible. This can be done through adding more classes and/or more methods. Additional text cleaning and pre-processing functionality can be added as new methods in the CleanText class. For instance, removing stop words and removing special characters could be two methods to extend the functionality. Additional models and rankers can also be added in the text_models and rankers modules as their own classes. The current BertBaseModel and CosineSimilarityRanker classes are simple and compact so a base class could be created for more structure if additional model and rankers were to be added. The classes could also be extended by adding methods such as comparing two text strings, adding text to the corpus, saving the state of the ranker instance, and loading the state of the ranker instance. Furthermore, the clean_text and text_models modules could also be extended to support other languages besides English.