# DDR Tutorial

This tutorial explains how the ddr package can be used to generate distributed dictionary loadings for a corpus of documents.

---

---

# Introduction

The ddr method was developed to bridge the gap between cutting edge natural language processing algorithms and hypothesis driven social science. ddr involves:

1. using distributed representations of words generated by the Word2Vec algorithm (implemented via the gensim package) to represent both dictionaries of theoretically relevant terms and individual documents as high-dimensional vectors and
2. calculating the similarity (cosine similarity) between dictionary and document representations.

Accordingly, this method combines the flexibility and automation offered by distributed representations with the ability to examine precisely modeled domains of theoretical interest.

The module that this tutorial introduces (also called ddr) serves a similar goal: making ddr accessible to users with minimal programming experience.

Specifically, this tutorial provides examples of how to use the functions implemented in the ddr module. In general, these functions can be classified into two groups: a function that implements the ddr algorithm and a range of helper functions that streamline the conversion of dictionaries and corpora into the format required by ddr.

## Basic required components:

### Implementing ddr requires three components:

1. Pre-trained distributed vector representations of words (e.g. vectors trained on the Google News corpus, available here: https://github.com/mmihaltz/word2vec-GoogleNews-vectors)
2. One or more dictionaries of terms representing dimensions of interest (e.g. the Moral Foundations Dictionaries packaged with ddr)
3. A corpus of documents for which dictionary-dimension similarities (henceforth referred to as document-dictionary loadings) will be calculated.

### General Workflow

Starting with these three components, several steps must be completed in order to generate document-dictionary loadings:

1. *Generate distributed representations of each dictionary dimension.* This is done by querying the representation of each dictionary word from the set of pre-trained vectors and averaging these word representations to create a distributed representation of the dictionary space.
2. *Generate distributed representations of each document in the target corpus.* This is done by querying the representation of each word in a document from the set of pre-trained vectors and averaging these word representations to create a distributed representation of the document space.
3. *Calculate similarity between document and dictionary representations.* This is done by calculating the cosine similarity between each document and dictionary representation.

**NOTE:** The code displayed in this tutorial uses brackets to indicate where a user must supply an input. Further, if the contents of a pair of brackets is enclosed in quotes, the user must enclose her input in quotes; conversely, if the contents of a pair of brackets is not enclosed in quotes, the user should not enclose her input in quotes as this indicates an object name: For example,

>>> path = ['path']

indicates that the user should replace 'path' with a string representing a directory path.

## Pre-trained vectors

The current version of ddr uses distributed representations of words generated by training the Word2Vec algorithm on a large corpus of text. Briefly, Word2Vec is able to efficiently learn statistical relationships between words and map these relationships into high-dimensional space. Importantly, words that occur in similar contexts within the training corpus cluster together in this high-dimensional space, which means that a latent conceptual dimension can be represented by combining representations of individual words relevant to that dimension.

Once a set of vectors are trained on a corpus, they can be reused indefinitely. However, it should be kept in mind that the domain of the training corpus has a substantial effect on the representations of words. For example, a set of vectors trained on the Wikipedia corpus will be quite different from a set of vectors trained on the Google News corpus because these corpora are characterized by different kinds of language usage.

Distributed representations of words (vectors) can be obtained either by using a pre-trained set of vectors or by training Word2Vec on a sufficiently large corpus.

### Accessing pre-trained vectors

An excellent set of pre-trained vectors can be obtained {here}. These vectors were generated by training Word2Vec on the Google News corpus ..... After downloading the file containing these vectors, they can be loaded into a Python environment using either gensim or ddr's wrapper function for gensim.

### Training your own vectors

Alternatively, a new set of vectors can be generated by training Word2Vec on a new corpus. While the minimum corpus size required for reliable training has yet to be identified, it is the case that a larger corpus will yield more reliable representations. Assuming a sufficiently large training corpus is available, training a new set of vectors on from the domain of interest will likely yield better representations of the syntactical and semantic patterns characteristic of that domain.

## Generating dictionary representations

### Loading a model

In order to generate a distributed representation of a dictionary, a model (set of pre-trained vectors) must be loaded and several features must be extracted from the model. Specifically, the dimensionality of the model and the vocabulary of the model must be stored as objects. This can be accomplished using the following command:

```
>>> import ddr
>>> model, num_features, index2word_set = ddr.load_model(model_path =
    ['path_to_model'])
```

The function load_model() takes as input the path to a model and returns three objects: the model (model), the dimensionality of the model (num_features), and the vocabulary set (index2word_set).

## Creating dictionary and document representations

In order to create representations of dictionaries the dictionary terms must be loaded into a Python dictionary with dictionary/dimension names and dictionary/dimension terms as key, [values] pairs. The ddr module offers functions that can convert dictionaries from a range of formats into the format required by ddr. These formats include:

### Load dictionary terms

1. *Text file format:* In this format, each dictionary dimension is represented by a separate text file and each text file contains the terms for that dimension. Terms should be separated only by a space. When reading from this format, the name of each text file is used as the name for the dictionary.
2. *LIWC format:* LIWC dictionaries are formatted according to specific regulations. ddr's terms_from_liwc() function converts a LIWC dictionary to the requisite Python dictionary format.
3. *CSV format*: ddr can also read dictionaries from a CSV file in which the header contains dictionary names and columns contain dictionary terms.

**Load text file format dictionaries:**

```
>>> import ddr
>>> dic_terms = ddr.terms_from_txt_(input_path = ['path to directory of dictionaries'])
```

**Load LIWC format dictionary:**

```
>>> dic_terms = ddr.terms_from_liwc(input_path = ['path to LIWC dictionary'])
```

**Load CSV format dictionary:**

```
>>> dicTerms = ddr.terms_from_csv(input_path = ['path to csv'],
        delimiter = ['type of CSV delimitation']
```

**Note:** ddr also contains a function to write the required Python dictionary format to a CSV file for which the header is dictionary dimension names and columns are dictionary terms:

```
>>> ddr.terms_to_csv(terms_dic = [dictionary object], output_path = ['path for output'],
            delimiter = ['delimiter for CSV'])
```

### Make aggregate distributed dictionary representations

Once the dictionary terms are loading into the requisite Python dictionary object, this object can be fed to the getAggDicVec() function, which returns a dictionary of aggregate vectors created by averaging the vector representations of each word in a given dictionary. In addition to the Python dictionary of dictionary terms, this function requires a model of vectors, the dimensionality of the model, and the model vocabulary as arguments. Additionally, a list of terms can be excluded from the representations by supplying a list of terms to the filter_out argument:

```
>>> agg_dic_vecs = ddr.dic_vecs(dic_terms = [dictionary],
                model = [model],
                num_features = [dimensionality],
                model_word_set = [model vocabulary])
```

These dictionary representations can then can be written to a CSV file in which the header contains dictionary names and columns contain the dictionary vector representations (shown below):

```
>>> ddr.terms_to_csv(dic_vecs = [dictionary of vectors], output_path = 'output path',
            delimiter = ['csv delimiter']
```

### Generate document representations

ddr can generate document representations for corpora in two different formats: CSV format with documents in rows and text format with each document on a separate line. The function that performs this operation generates an output file containing the distributed document representations along with unique document IDs. In this file, rows contain documents and columns contain dimensions. Thus, document representations for a corpus of 100 documents generated from a model with 300 dimensions would yield an output file with 100 rows and 300 columns.

#### Document representations from CSV format corpus

To generate document representations from documents contained in a CSV file, the doc_vecs_from_csv() function can be used. This function requires the path to the CSV corpus, the name or number of the column contain the documents, a boolean indicator of whether the CSV file contains a header, the quote character used in the CSV file (if any), an output path, the model, model dimensionality, and model vocabulary as arguments. Other optional arguments can be used to customize the CSV output and to match unique identifiers contained in the CSV corpus to the distributed document representations.

The output file can also be customized using the following arguments:

- *delimiter*: Specify the delimiter used in the output file

- *id_col*: If the document corpus already contains unique identifiers in a column, these identifiers can be paired with the matching distributed document representations by specifying either the column name or number in this argument. The default for this argument is False, which results in unique document identifiers being automatically generated and stored in the output.

**Example:**

```
>>> agg_doc_vecs_from_csv(input_path = ['path to CSV corpus'],
            output_path = ['output path'],
            model = [model],
            num_features = [model dimensionality],
            quotechar = None,
            model_word_set = [model vocabulary],
            text_col = ['name or number of column containing text'],
            header = True)
```

**Document representations from text format corpus**

ddr also offers a function that will create document representations from a text file in which each document is stored on a separate line or a directory of text files in which each document is contained in a single text file with no line breaks. Similar to doc_vecs_from_csv(), doc_vecs_from_text() takes a path to the text file(s), an output path, the model dimensionality, and the model vocabulary as arguments. Using the arguments described above, the user can also specify the delimiter used in the output file.

## Generate document-dictionary similarity measures

To generate document-dictionary similarity measures, the get_loadings() function can be used. This function takes five arguments:

- **agg_doc_vecs_path**: the path to CSV file with aggregate document vectors as rows and a unique ID as the first column.
- **agg_dic_ves_path**: the path to CSV file with aggregated dictionary vectors as columns.
- **out_path**: path for output.
- **num_features**: Dimensionality of the Word2Vec model used to generate vector representations.
- **delimiter**: Delimiter to use in output CSV

**Example:**

```
>>>get_loadings( agg_doc_vecs_path=['path'],
                 agg_dic_vecs_path=['path'],
                 out_path=['path'],
                 num_features=model_dimensionality,
                 delimiter = '\\t' )
```