This package contains the data and code for “*Measuring Corporate Human Capital Disclosures: Lexicon, Data, Code, and Research Opportunities*.”

All the data and code are freely available. We request that you reference our paper if you use the data or the code. We manually collected the HC disclosures from almost 4,000 annual reports. These disclosures consist of approximately two million words. We share all the codes used to develop our word lists. We also provide additional codes, which can be used together with our word lists for constructing textual measures. You are free to use the data or tailor the codes to suit your project or setting. However, please do not share the data or code with others or make them publicly available, e.g., by hosting them on an online site or a cloud. These data and codes are stored in the online archives of the *Journal of Information Systems*, where users should obtain them.

The “Data” folder consists of the following files:

* JIS\_Data\_HC\_Disclosures.txt: This file consists of the HC disclosures from 3,953 companies. We used this corpus of HC disclosures to develop our word lists. “JIS\_HC1\_util.py” provides a function to parse these disclosures.
* final\_HC\_wordlist.csv: This file contains the final HC word lists.
* HC\_seedwords.csv: This file contains the seed words we used for developing our word lists.

The “Code” folder consists of the following files:

* requirements.txt: This file provides a list of the Python libraries required for running the code.
* Python code files:
  + JIS\_HC0\_config.py
  + JIS\_HC1\_util.py
  + JIS\_HC2\_load\_keywords.py
  + JIS\_HC3\_generate\_keyword\_variables.py
  + JIS\_HC4\_train\_phrases.py
  + JIS\_HC5\_preprocess\_for\_word2vec.py
  + JIS\_HC6\_train\_word2vec.py

Please note that the editorial system does not accept “py” and “csv” files. As a workaround, we have changed the extensions of “py” and “csv” files to “txt”. Please change these files back to their original extensions before opening them.

Please see the Appendix for a detailed description of the code files, in addition to the notes contained in the files.

**APPENDIX – PYTHON CODE DESCRIPTION**

In this appendix, we present the Python code for using our lexicon.[[1]](#footnote-1) We also provide the code for training a *word2vec* model in the event that researchers want to expand our HC word list or create an entirely new word list to capture a different topic. These codes have been tested on a computer running Windows 11. Please see the “requirements.txt” for the list of Python libraries required for running the codes.

These codes are designed to be memory efficient so that researchers can use them to process collections of documents that are too large to be loaded into the memory all at once. Moreover, our code supports parallel processing by using multiple CPU cores to cut the processing time. For example, when we generate the variables for the more than 186,000 proxy statements over 1994-2022, it takes only about 20 hours using ten logical CPU cores. These proxy statements contain 3.3 billion words and occupy a disk space of more than 20GB. Using 10 workers, it takes about 38 hours to train a new *word2vec* model on this same large corpus of documents. Most of the time (28 hours) is spent on preprocessing the documents, which includes training the phrase model.

**Code for generating keyword-based measures using our lexicon**

In what follows, we provide some technical guidance on how to apply our lexicon to create keyword-based measures. The keywords presented in Table 3 are lemmatized and contain phrases. There are two ways to use keywords in this format. First, one can rewrite each keyword into a regular expression to match the keyword and its alternative forms.[[2]](#footnote-2) Re-writing keywords into regular expressions can be cumbersome for hundreds of keywords. An alternative and more efficient approach is to tokenize the text, lemmatize the words, join certain words into phrases, and then check each word or phrase to see whether it is on the keyword list. Our code takes this second approach.

For greater readability and ease of use, we group our code into six modules (i.e., individual source code files). We briefly describe these modules below. More notes can be found in the code files.

**(1) JIS\_HC0\_config.py**

This module allows users to define parameters, options, and settings for the project – e.g., where the keyword list is located, where the source documents can be found, how many parallel processes to run, where to save preprocessed files and final outputs, and what parameters to use for phrase model training and *word2vec* training.

**(2) JIS\_HC1\_util.py**

This model contains functions for performing certain routine tasks such as generating a list of all files in a directory, reading text from a file, tokenizing a text into sentences, lemmatizing words, and outputting the results. Users may find these functions useful for many other NLP projects.

**(3) JIS\_HC2\_load\_keywords.py**

This module loads HC keywords from the CSV file and generates three objects for use in later steps: a list of all HC keywords, a list of all HC keywords that are phrases (bigrams or trigrams), and a Python data structure known as a “dictionary” for storing HC keywords for each of the five categories.

**(4) JIS\_HC3\_generate\_keyword\_variables.py**

This module generates variables using our HC keyword lists. Users do not need to run the previous three modules separately. They are imported into this module automatically when this module is run. This module defines some additional functions. It performs the following steps by using these additional functions as well as the functions or objects from the previous three modules:

1. Load keywords from the CSV file. This is done by importing the three objects from “JIS\_HC1\_util.py.”Users do not need to run“JIS\_HC1\_util.py” separately.
2. Traverse the specified directory (including sub-directories) to find all txt files that need to be processed.
3. Open each txt file and extract the text.
4. Tokenize the text into sentences and further tokenize each sentence into words.
5. Convert all words into lower cases and lemmatize them.
6. Join words that are part of a phrase with an underline “\_”, according to the list of phrases in our keyword list. For example, “affirmative action” is converted to “affirmative\_action”.
7. Count the frequencies for all keywords in each category.
8. Identify sentences containing any of the keywords and count the number of such sentences as well as the word count of all such sentences.
9. Get the word count and sentence count of the entire document.
10. Output the txt file name (i.e., the document identifier) and variables as a new row in a CSV file.

Steps 3 through 10 are performed in parallel. Users can specify the number of workers in “JIS\_HC0\_config.py”. By default, this module uses all available logical CPU cores, less two.

For tokenization, we mostly use tools from NLTK, which is known for its flexibility, a large support community, and great performance for this task.[[3]](#footnote-3) For lemmatization, we provide two options, NLTK and Spacy.[[4]](#footnote-4) The default is the “WordNetLemmatizer” available with NLTK. Users can choose to use the Spacy lemmatizer by changing the setting in “JIS\_HC0\_config.py.” Although the Spacy lemmatizer is slightly quicker, users should be aware that the Spacy lemmatizer converts all pronouns to “-PRON-”. In addition, the Spacy lemmatizer keeps proper pronouns such as “Native Americans” unchanged (i.e., instead of converting it to “Native American”). To use the Spacy lemmatizer, some keywords may need to be adapted.

This module takes a list of text files from the specified directory as inputs. It is ideal for the use case where one works with a large corpus of documents, for example, 10-K or 10-Q filings, each of which is saved in a separate txt file. Users can try out the code by using cleaned 10-K filings available at <https://sraf.nd.edu/sec-edgar-data/cleaned-10x-files/>. The code can be easily adapted for use in cases where the text is stored in another format, for example, in CSV files.

The next few modules are for training a *word2vec* model.

**Code for training a *word2vec* model**

**(5)** **JIS\_HC4\_train\_phrases.py**

This module trains a phrase model using Gensim, a popular Python library for NLP.[[5]](#footnote-5) As part of the preprocessing for *word2vec* training, we use the phrase model to generate phrases so that the *word2vec* algorithmcanidentify not only single words but also phrases that are similar to the seed words. This module can be used for many NLP projects that benefit from phrase training, such as LDA topic modelling.

Training a phrase model is a time-consuming process for a large corpus because the algorithm needs to go through the entire corpus multiple times. To reduce the processing time, our code saves pre-processed sentences to a folder specified by the user in “JIS\_HC0\_config.py.” It then loads the pre-processed sentences from the folder when they are needed again in later steps. To discover both two-word phrases (bigrams) and three-word phrases (trigrams), this module performs the following steps:

1. Preprocess the text (by tokenizing the text into sentences, tokenizing the sentences into words, lemmatizing the words, and further normalizing the words), and save these sentences consisting of unigrams (i.e., single words, without phrases being tagged).

For the normalization procedure, we convert words containing numbers to “1” and convert words containing punctuations (excluding “-” and “\_”) to “,”.[[6]](#footnote-6) This can reduce the vocabulary size and speed up the later training process. Note that we do not simply drop non-alphabetic words, punctuations, or stop words, because doing so will break the flow of a sentence and result in phrases consisting of words that are separated by a number, punctuation, or stop word in a sentence.

1. Load the preprocessed unigram sentences from the disk and feed them into the algorithm to train a bigram model.
2. Obtain all bigrams discovered by the model and generate a list of valid two-word phrases (bigrams) by removing those containing stop words, numbers, punctuations (other than “\_” and “-”) or single-letter words.[[7]](#footnote-7)
3. Process the unigram sentences to form phrases using the bigram model (e.g., converting “workplace safety” to “workplace\_safety”), break the phrases not in the list of valid phrases from (c), and save the results (bigram sentences) to the disk.
4. Load the bigram sentences and feed them into the algorithm again to train a trigram model. Save the model to the disk.
5. Obtain all the bigrams and trigrams identified by the model and generate a list of valid two-word phrases and three-word phrases by excluding those bigrams/trigrams containing stop words, numbers, punctuations (other than “\_” and “-”) or single-letter words. Export the list of bigrams and trigrams into a CSV file for inspection and evaluation.

**(6)** **JIS\_HC5\_preprocess\_for\_word2vec.py**

This module generates the final inputs for training a *word2vec* model. The final inputs are sentences that are tokenized, lemmatized, phrased, and have all stop words, numbers, and punctuations removed. Building on the previous phrase modelling process, it performs the following steps:

1. Load the bigram sentences, the trigram model, and the list of valid phrases.
2. Form phrases using the trigram model and break those not in the valid phrase list.[[8]](#footnote-8)
3. Remove stop words, one-letter words, and words containing digits or punctuations (excluding “\_” and “-”).
4. Save the results (i.e., trigram sentences) to the disk.

The module will first look for preprocessed trigram sentences in the folder specified in “JIS\_HC0\_config.py.” If they are not found, it will generate these inputs by going through the whole process. In this way, users do not need to generate the inputs again when they try out new parameters for the *word2vec* model. However, if users want to use new parameters for phrase modelling, they should delete the previously saved trigram sentences and bigram/trigram models so that new ones will be generated to reflect the desired changes.

**(7) JIS\_HC6\_train\_word2vec.py**

This module trains a *word2vec* model, saves the model, and outputs the similar words of a given list of words into a CSV file. It performs the following steps:

1. Load the preprocessed trigram sentences from the specified folder. If no files are found, it will generate these sentences using the txt files in the specified folder by calling related functions from “JIS\_HC4\_train\_phrases.py” and “JIS\_HC5\_final\_preprocess\_for\_word2vec.py.”

Users do not need to run these two modules (i.e., “JIS\_HC4” and “JIS\_HC5”) before they run the current module. However, users are recommended to first run the phrase training module (“JIS\_HC4\_train\_phrases.py”) and inspect the results. Based on the results, users can fine-tune the parameters and determine whether to exclude certain phrases discovered by the algorithm. Once the decision is made, users can remove the preprocessed files and saved models, and re-run this module so that a new model will be trained using new parameters.

1. Train a *word2vec* model using Gensim based on the parameters from “JIS\_HC0\_config.py” and save the model to the specified folder.
2. Load the seed words from the CSV file specified in “JIS\_HC0\_config.py.” Each column of the CSV file should contain the seed words for one category. The top row of each column should contain the category name (e.g., “Compensation&Benefits”). If a seed word is a phrase, it should be separated by spaces, for example, “human resource”, instead of “human\_resource.”
3. Generate similar words and export them to a CSV file.

The CSV file contains the following columns: “Category,” “SeedWord,” “SeedWordFreq,” “SimWord,” “SimScore,” “SimWordFreq,” “SimScore\_ABS,” and “AvgCatSim\_ABS.” “SeedWordFreq” represents the number of times the seed word appears in the training corpus. “SimScore” represents the similarity score between a seed word and a similar word (“SimWord”), with a higher value indicating greater similarity. “SimWordFreq” represents the number of times that the similar word appears in the corpus. “SimScore\_ABS” is the absolute value of “SimScore,” and is useful for considering similar words with both positive and negative similarity scores. “AvgCatSim\_ABS” is the average absolute similarity score of a word with all the seed words in a category. When a word is found to be highly similar to seedwords from multiple categories, the word is assigned to the category with the highest “AvgCatSim\_ABS.”

For each seed word, users can choose how many similar words to keep, for example, the top 300 words or top 500 words with the highest similarity scores. Alternatively, researchers can choose to keep similar words whose similarity scores meet a certain threshold, such as +0.5 or an absolute value of 0.5. Users can change the settings in “JIS\_HC0\_config.py.” Under either approach, a word may be found to be highly similar to multiple seed words that belong to different categories. This gives rise to a situation where a word can be classified into multiple categories. One way to avoid this is to compute the average similarity score between the word and seed words in each category and re-assign the word to the category having the highest average score. The program outputs another list of similar words reclassified in this way to a CSV file. Ultimately, however, the researcher should determine which category or categories the word belongs to.

1. Although our dictionary is intended for general purpose use, as suggested by Bochkay et al. (2022), researchers are encouraged to make their own adaptations to our lexicon to tailor it to their particular setting. In information retrieval or classification, a proper balance needs to be struck between “precision” and “recall,” which are analogous to Type I and Type II errors. In the current context, this would require that a researcher examine the sample of results generated by our lexicon, and then make whatever changes may be necessary to the keywords to suit their specific research context (e.g., excluding certain keywords, including new keywords, and/or modifying the forms of certain keywords). [↑](#footnote-ref-1)
2. For example, “equality” may appear in a plural form or in upper case, or have one or more of its letters capitalized. To capture these variants, one needs to write a regular expression as “(?i)equalit(y|ies)”. To make the whole regular expression ignore case, a flag (e.g., “re.I” for Python) can be passed to the regex engine instead of using the inline flag, “(?i)”. [↑](#footnote-ref-2)
3. https://www.nltk.org/ [↑](#footnote-ref-3)
4. https://pypi.org/project/spacy/ [↑](#footnote-ref-4)
5. https://pypi.org/project/gensim/ [↑](#footnote-ref-5)
6. If any word containing punctuations or numbers is known to be important (e.g., “COVID-19”), it should be added to the whitelist. [↑](#footnote-ref-6)
7. If a phrase containing these elements is known to be important, it should be added to the whitelist. [↑](#footnote-ref-7)
8. If one of the two words is a two-word phrase, this will result in a three-word phrase (e.g., “workplace\_safety\_program”). If both words are two-word phrases, then this will result in an occasional four-word phrase. [↑](#footnote-ref-8)