CWordTM Usage on BBC News

This Jupyter notebook demonstrates how to use the package "CWordTM" on the BBC News:

1. Meta Information Features
2. Utility Features
3. Text Visualization - Word Cloud
4. Text Summarization
5. Topic Modeling - LDA and BERTopic

1. Meta Information Features

In [1]:
```
import cwordtm
from cwordtm import *
```

In [2]:
```
# Show brief module information
print(meta.get_module_info())
```

```
The member information of the module 'cwordtm'
1. Submodule meta:
   addin (func)
   addin_all (modname='cwordtm')
   addin_all_functions (submod)
   get_function (mod_name, submodules, func_name)
   get_module_info (detailed=False)
   import_module (name, package=None)
   wraps (wrapped, assigned=('__module__', '__name__', '__qualname__', '__doc__', '__annotations__'), updated=('__di
ct__',))
2. Submodule pivot:
   stat (df, chi=False, *, timing=False, code=0)
3. Submodule quot:
   extract_quotation (text, quot_marks, *, timing=False, code=0)
   match_text (target, sent_tokens, lang, threshold, n=5, *, timing=False, code=0)
   match_verse (i, ot_list, otdf, df, book, chap, verse, lang, threshold, *, timing=False, code=0)
   show_quot (target, source='ot', lang='en', threshold=0.5, *, timing=False, code=0)
   tokenize (sentence, *, timing=False, code=0)
4. Submodule ta:
   get_sent_scores (sentences, diction, sent_len, *, timing=False, code=0) -> dict
   get_sentences (docs, lang='en', *, timing=False, code=0)
   get_summary (sentences, sent_weight, threshold, sent_len, *, timing=False, code=0)
   pos_tag (tokens, tagset=None, lang='eng', *, timing=False, code=0)
   preprocess_sent (text, *, timing=False, code=0)
   sent_tokenize (text, language='english', *, timing=False, code=0)
   summary_chi (docs, weight=1.5, sent_len=8, *, timing=False, code=0)
   summary_en (docs, sent_len=8, *, timing=False, code=0)
   word_tokenize (text, language='english', preserve_line=False, *, timing=False, code=0)
5. Submodule tm:
   BTM (textfile, chi=False, num_topics=15, embed=True)
   LDA (textfile, chi=False, num_topics=15)
   NMF (textfile, chi=False, num_topics=15)
   btm_process (doc_file, source=0, text_col='text', cat=0, chi=False, group=True, eval=False, *, timing=False, code
=0)
   lda_process (doc_file, source=0, text_col='text', cat=0, chi=False, group=True, eval=False, *, timing=False, code
=0)
   load_bible (textfile, cat=0, group=True, *, timing=False, code=0)
   load_text (textfile, text_col='text', *, timing=False, code=0)
   nmf_process (doc_file, source=0, text_col='text', cat=0, chi=False, group=True, eval=False, *, timing=False, code
=0)
   pprint (object, stream=None, indent=1, width=80, depth=None, *, compact=False, sort_dicts=True, underscore_number
s=False, timing=False, code=0)
   process_text (doc, *, timing=False, code=0)
6. Submodule util:
   add_chi_vocab (*, timing=False, code=0)
   chi_sent_terms (text, *, timing=False, code=0)
   chi_stops (*, timing=False, code=0)
   clean_sentences (sentences, *, timing=False, code=0)
   clean_text (df, text_col='text', *, timing=False, code=0)
   extract (df, testament=-1, category='', book=0, chapter=0, verse=0, *, timing=False, code=0)
   extract2 (df, filter='', *, timing=False, code=0)
   get_diction (docs, *, timing=False, code=0)
   get_diction_chi (docs, *, timing=False, code=0)
   get_diction_en (docs, *, timing=False, code=0)
   get_list (df, column='book', *, timing=False, code=0)
   get_sent_terms (text, *, timing=False, code=0)
   get_text (df, text_col='text', *, timing=False, code=0)
   get_text_list (df, text_col='text', *, timing=False, code=0)
   group_text (df, column='chapter', *, timing=False, code=0)
   is_chi (*, timing=False, code=0)
   load_text (filepath, nr=0, info=False, *, timing=False, code=0)
   load_word (ver='web.csv', nr=0, info=False, *, timing=False, code=0)
   preprocess_text (text, *, timing=False, code=0)
   set_lang (lang='en', *, timing=False, code=0)
   word_tokenize (text, language='english', preserve_line=False, *, timing=False, code=0)
7. Submodule version:
8. Submodule viz:
   chi_wordcloud (docs, figsize=(15, 10), bg='white', image=0, *, timing=False, code=0)
   plot_cloud (wordcloud, figsize, *, timing=False, code=0)
   show_wordcloud (docs, clean=False, figsize=(12, 8), bg='white', image=0, *, timing=False, code=0)
```

In [3]:
```
# Show detailed module information
print(meta.get_module_info(detailed=True))
```

```
The member information of the module 'cwordtm'
1. Submodule meta:
def addin(func):
    """Adds additional features (showing timing information and source code)
    to a function at runtime. This adds two parameters ('timing' & 'code') to
    function 'func' at runtime. 'timing' is a flag indicating whether
    the execution time of the function is shown, and it is default to False.
    'code' is an indicator determining if the source code of the function
    'func' is shown and/or the function is invoked; '0' indicates the function
    is executed but its source code is not shown, '1' indicates the source code
    of the function is shown after execution, or '2' indicates the source code
    of the function is shown without execution, and it is default to 0.

    :param func: The target function for inserting additiolnal features -
        timing information and showing code, default to None
    :type func: function
    :return: The wrapper function
    :rtype: function
    """

    try:
        if "code" in inspect.signature(func).parameters:
            return
    except ValueError:
        return

    mod_name = __name__.split('.')[0]
    module = import_module(mod_name)

    # Get Submodules of cwordtm
    submodules = [name for name in dir(module)
                    if isinstance(getattr(module, name), type(module))]

    exclusion = ["files", "WordCloud"]

    def next_level(func):
        source_code = inspect.getsource(func)
        tree = ast.parse(source_code)
        for node in ast.walk(tree):
            if isinstance(node, ast.Call):
                func_call_code = ast.get_source_segment(source_code, node)
                func_name = func_call_code.split('(')[0].split(".")[-1]
                func_obj, submod = get_function(mod_name, submodules, func_name)
                if func_obj is not None and func_name not in exclusion:
                    module_name = inspect.getmodule(func_obj).__name__
                    print(">>", module_name + "." + func_name)
                    print(inspect.getsource(func_obj))

    @wraps(func)
    def wrapper(*args, timing=False, code=0, **kwargs):
        """Wrapper function to add two parameters ('timing' & 'code') to
        function 'func' at runtime.

        :param timing: The flag indicating whether the execution time of the
        function 'func' is shown, default to False
        :type timing: bool, optional
        :param code: The indicator determining if the source code of the function
            'func' is shown and/or the function is invoked; '0' indicates the function
            is executed but its source code is not shown, '1' indicates the source code
            of the function is shown after execution, or '2' indicates the source code
            of the function is shown without execution, default to 0
        :type code: bool, optional
        :return: The execution time (in seconds) of the function 'func'
        :rtype: float
        """

        if code == 0 or code == 1:
            start_time = time.perf_counter()
            value = func(*args, **kwargs)
            end_time = time.perf_counter()
            run_time = end_time - start_time

            if timing:
                print(f"Finished {func.__name__!r} in {run_time:.4f} secs")

            if code == 1:
                if isinstance(func, types.BuiltinFunctionType):
                    print("\nSource code of the funcrion '%s' cannot be retrieved!" \
                          %func.__name__)
                else:
                    print("\n" + inspect.getsource(func))
                    next_level(func)

            return value
        elif code == 2:
            if isinstance(func, types.BuiltinFunctionType):
```

```
                print("\nSource code of the funcrion '%s' cannot be retrieved!" \
                      %func.__name__)
            else:
                print("\n" + inspect.getsource(func) + "\n")
                next_level(func)

            return None

        sig = inspect.signature(func)
        params = list(sig.parameters.values())
        params.append(inspect.Parameter("timing",
                                        inspect.Parameter.KEYWORD_ONLY,
                                        default=False))
        params.append(inspect.Parameter("code",
                                        inspect.Parameter.KEYWORD_ONLY,
                                        default=0))
        wrapper.__signature__ = sig.replace(parameters=params)
        return wrapper

    def addin_all(modname='cwordtm'):
        """Applies 'addin' function to all functions of all sub-modules of
        a module at runtime.

        :param modname: The target module of which all the functions are inserted
            additional features, default to 'wordtm'
        :type modname: str, optional
        """

        module = import_module(modname)

        if hasattr(module, "__path__"):
            for _, submodname, ispkg in pkgutil.iter_modules(module.__path__):
                if not ispkg and submodname != 'meta':
                    submod = import_module(".." + submodname, \
                                           modname + "." + submodname)
                    addin_all_functions(submod)
        else:
            addin_all_functions(module)

    def addin_all_functions(submod):
        """Applies 'addin' function to all functions of a module at runtime.

        :param submod: The target sub-module of which all the functions are inserted
            additional features, default to None
        :type submod: module
        """

        for name, member in inspect.getmembers(submod):
            if callable(member) and \
               member.__name__ != 'files' and \
               name[0].islower():
                setattr(submod, name, addin(member))

    def get_function(mod_name, submodules, func_name):
        """Gets the object of the function 'func_name' if it belongs
        to one of 'submodules' of the current top-level module.

        :param mod_name: The name of the source top-level module, default to None
        :type mod_name: str
        :param submodules: The list of names of the sub-modules of the top-level module
        :type submodules: list
        :param func_name: The name of the fuunction to be looked for
        :type func_name: str
        :return: The object of the target function, if any, otherwise None
        :rtype: function
        """

        for submod in submodules:
            mod_obj = import_module(mod_name + "." + submod)
            if hasattr(mod_obj, func_name):
                return getattr(mod_obj, func_name), submod

        return None, None

    def get_module_info(detailed=False):
        """Gets the information of the module 'cwordtm'.

        :param detailed: The flag indicating whether only function signature or
            detailed source code is shown, default to False
        :type detailed: bool, optional
        :return: The information of the module 'cwordtm'
        :rtype: str
        """

        mod_name = __name__.split('.')[0]   # Current module
        module = import_module(mod_name)
```

```
        func1 = "@validate_params"
        func2 = "def cosine_similarity"

        i = 0
        mod_info = "The member information of the module '" + mod_name + "'\n"
        for _, submodname, ispkg in pkgutil.iter_modules(module.__path__):
            if not ispkg:
                i += 1
                mod_info += "%d. Submodule %s:" %(i, submodname) + "\n"
                submod = import_module(".."+submodname, mod_name+"."+submodname)
                for name, member in inspect.getmembers(submod):
                    if (inspect.isclass(member) and name in ['LDA', 'NMF', 'BTM']) or \
                       (inspect.isfunction(member) and name != 'files' and
                        not name in func1 and not name in func2):
                        if detailed:
                            mod_info += "{}\n".format(inspect.getsource(member))
                        else:
                            mod_info += "   {} {}\n".format(name, inspect.signature(member))

        return mod_info

    def import_module(name, package=None):
        """Import a module.

        The 'package' argument is required when performing a relative import. It
        specifies the package to use as the anchor point from which to resolve the
        relative import to an absolute import.

        """
        level = 0
        if name.startswith('.'):
            if not package:
                msg = ("the 'package' argument is required to perform a relative "
                       "import for {!r}")
                raise TypeError(msg.format(name))
            for character in name:
                if character != '.':
                    break
                level += 1
        return _bootstrap._gcd_import(name[level:], package, level)

    def wraps(wrapped,
              assigned = WRAPPER_ASSIGNMENTS,
              updated = WRAPPER_UPDATES):
        """Decorator factory to apply update_wrapper() to a wrapper function

           Returns a decorator that invokes update_wrapper() with the decorated
           function as the wrapper argument and the arguments to wraps() as the
           remaining arguments. Default arguments are as for update_wrapper().
           This is a convenience function to simplify applying partial() to
           update_wrapper().
        """
        return partial(update_wrapper, wrapped=wrapped,
                       assigned=assigned, updated=updated)

    2. Submodule pivot:
    def stat(df, chi=False):
        """Returns a pivot table from the DataFrame 'df' storing the input Scripture,
        with columns 'book', 'book_no', 'chapter', 'verse', 'text', 'testament',
        'category', 'cat', and 'cat_no'.

        :param df: The input DataFrame storing the Scripture, default to None
        :type df: pandas.DataFrame
        :param chi: If the value is True, assume the input text is in Chinese,
            otherwise, the input text is in English, default to False
        :type chi: bool, optional
        :return: The pivot table of the input Scripture grouped by category ('cat_no')
        :rtype: pandas.DataFrame
        """

        stat_df = pd.pivot_table(df, index = ['book_no', 'book', 'category', 'cat_no'],
                             values = ['chapter', 'verse', 'text'],
                             aggfunc = {'chapter': lambda ch: len(ch.unique()),
                                        'verse': 'count',
                                        'text': lambda ts: sum([len(t if chi else t.split()) for t in ts])})

        stat_df = stat_df[['chapter', 'verse', 'text']].sort_index()

        stat_df2 = stat_df.groupby('cat_no').apply(lambda sub: sub.pivot_table(
                            index = ['category', 'book_no', 'book'],
                            values = ['chapter', 'verse', 'text'],
                            aggfunc = {'chapter': 'sum',
                                       'verse': 'sum',
                                       'text': 'sum'},
                            margins = True,
                            margins_name = 'Sub-Total'))
```

```
    stat_df2.loc[('', '', 'Total', '')] = stat_df2.sum() // 2
    stat_df2.index = stat_df2.index.droplevel(0)
    stat_df2.fillna('', inplace=True)
    stat_df2 = stat_df2[['chapter', 'verse', 'text']]
    return stat_df2

3. Submodule quot:
def extract_quotation(text, quot_marks):
    """Returns the text within a pair of quotation marks.

    :param text: The target text to be extracted, default to None
    :type text: str
    :param quot_marks: A pair of quotation marks, ['"', '"'] for English text
        or [' 『', '』 '] for Chinese text, default to None
    :type quot_marks: list
    :return: The text within a pair of quotation marks, if any, otherwise,
        an empty string
    :rtype: str
    """

    arr = text.split(quot_marks[0])
    if len(arr) > 1:
        return arr[1].split(quot_marks[1])[0]
    else:
        return ""

def match_text(target, sent_tokens, lang, threshold, n=5):
    """Returns a list of tuples of the cosine similarity measures of the OT verse
    with target verse and the index of that OT verse in the DataFrame storing
    the prescribed OT Scripture.

    :param target: The target verse to be matched, default to None
    :type target: str
    :param sent_tokens: The target verse to be matched, default to None
    :type sent_tokens: str
    :param lang: If the value is 'chi' , the processed language is assumed to be
        Chinese, otherwise, it is English, default to None
    :type lang: str
    :param threshold: The threshold value of the cosine similarity measure
        between the target verse and an OT verse, where the cosine similarity measure
        of a matched OT verse and the target verse should be greater this value,
        default to None
    :type threshold: float
    :param n: The upper bound of the number of matched verses, default to 5
    :type n: int, optional
    :return: The list of tuples of the cosine smilarity measure and the index
        of the OT verse
    :rtype: list
    """

    result = ''
    sent_tokens.append(target)
    if lang == 'chi':
        TfidfVec = TfidfVectorizer(tokenizer=tokenize, stop_words=chinese_stopwords)
    else:
        TfidfVec = TfidfVectorizer(analyzer='word', stop_words='english')

    tfidf = TfidfVec.fit_transform(sent_tokens)
    vals = cosine_similarity(tfidf[-1], tfidf)

    results = []
    for i in range(n):
        idx = vals.argsort()[0][-2-i]
        flat = vals.flatten()
        flat.sort()
        cos_sim = flat[-2-i]
        if (cos_sim > threshold):
            results.append((cos_sim, idx))

    return results

def match_verse(i, ot_list, otdf, df, book, chap, verse, lang, threshold):
    """Returns whether the target NT verse (book, chap, verse) can match a particular verse
    in the list of OT verses (ot_list), and prints the matched OT verse(s).

    :param i: The number of matched verses so far, default to None
    :type i: int
    :param ot_list: The list of OT verses (str) to be matched, default to None
    :type ot_list: list
    :param otdf: The DataFrame storing the prescribed OT verses to be matched,
        default to None
    :type otdf: pandas.DataFrame
    :param df: The DataFrame storing the collection of the target NT verses
        to be matched, default to None
    :type df: pandas.DataFrame
```

```
        :param book: The Bible book short name (3 characters) of the target NT verse
            to be matched, default to None
        :type book: str
        :param chap: The chapter number of the target NT verse to be matched,
            default to None
        :type chap: int
        :param verse: The verse number of the target NT verse to be matched,
            default to None
        :type verse: int
        :param lang: If the value is 'chi' , the processed language is assumed to be Chinese
            otherwise, it is English, default to None
        :type lang: str
        :param threshold: The threshold value of the cosine similarity measure
            between the target verse and an OT verse, where that measure for successful match
            should be greater this value, default to None
        :type threshold: float
        :return: True if the target verse matched an OT verse, False otherwise
        :rtype: bool
        """

        global book_cat
        if book_cat is None:
            book_cat = util.load_word('book_categories.csv')

        if lang == 'en':
            book_sname = book
            quot_marks = ['"', '"']
        else:
            book_sname = book_cat[book_cat.book_s == book].book_chi.iloc[0]
            quot_marks=['〝', '〞']

        averse = util.extract(df, book=book, chapter=chap, verse=verse)
        vtext = util.get_text(averse)

        quot = extract_quotation(vtext, quot_marks)
        if quot == "": return False

        nt_str = book_sname + ' ' + str(chap) + ':' + str(verse)
        print("(%2d) %-6s %s" %(i+1, nt_str, vtext))  # NT Verse

        results = match_text(quot, ot_list, lang, threshold)
        ot_list.remove(quot)

        for cos_sim, idx in results:
            sv = otdf.iloc[idx]

            if lang == 'en':
                book_sname = book
            else:
                book_sname = book_cat[book_cat.book_s == sv.book].book_chi.iloc[0]

            ot_str = book_sname + ' ' + str(sv.chapter) + ':' + str(sv.verse)
            print("     -> %.4f %-9s %s" %(cos_sim, ot_str, sv.text))  # OT Verse

        return True

    def show_quot(target, source='ot', lang='en', threshold=0.5):
        """Shows a collection of matched OT verses, if any, based on the prescribed
        collection of target NT verse and the threshold value.

        :param target: The collection of target NT verses to be matched, default to None
        :type target: pandas.DataFrame
        :param source: The string representing the collection of all or subset of OT verses
            to be matched, default to 'ot'
        :type source: str, optional
        :param lang: If the value is 'en', the processed language is assumed to be English
            otherwise, it is Chinese, default to 'en
        :type lang: str, optional
        :param threshold: The threshold value of the cosine similarity measure
            between the target verse and an OT verse, where that measure for successful match
            should be greater this value, default to 0.5
        :type threshold: str, optional
        :return: The list of tuples of the cosine smilarity measure and the index
            of the OT verse
        :rtype: list
        """

        util.set_lang(lang)
        ot_cat = ['tor', 'oth', 'ket', 'map', 'mip']

        if lang == 'en':
            df = util.load_word()
        else:
            df = util.load_word('cuv.csv')

        if source in ot_cat:
```

```
            otdf = util.extract(df, category=source)
        else:
            otdf = util.extract(df, testament=0)

        ot_list = util.get_text_list(otdf)

        i = 0
        for _, row in target.iterrows():
            if match_verse(i, ot_list, otdf, target, row.book, row.chapter, row.verse, lang, threshold):
                i += 1

    def tokenize(sentence):
        """Returns a list of tokens from a Chinese sentence.

        :param sentence: The target text to be tokenized, default to None
        :type sentence: str
        :return: The generator object that storing the list of tokens
            extracted from the sentence
        :rtype: generator
        """

        without_duplicates = re.sub(r'(.)\1+', r'\1\1', sentence)
        without_punctuation = re.sub(r'[^\w]','', without_duplicates)
        return jieba.cut(without_duplicates)

    4. Submodule ta:
    def get_sent_scores(sentences, diction, sent_len) -> dict:
        """Returns the dictionary of a list of sentences with their scores
        computed by their words.

        :param sentences: The list of sentences for computing their scores,
            default to None
        :type sentences: list
        :param diction: The dictionary storing the collection of tokenized words
            with their frequencies
        :type diction: collections.Counter object
        :param sent_len: The upper bound of the number of sentences to be processed,
            default to None
        :type sent_len: int
        :return: The list of sentences tokenized from the collection of document
        :rtype: pandas.DataFrame
        """

        sent_weight = dict()

        for sentence in sentences:
            sent_wordcount_net = 0
            for word_weight in diction:
                if word_weight in sentence.lower():
                    sent_wordcount_net += 1
                    if sentence[:sent_len] in sent_weight:
                        sent_weight[sentence[:sent_len]] += diction[word_weight]
                    else:
                        sent_weight[sentence[:sent_len]] = diction[word_weight]

            if sent_weight != dict() and sent_weight.get(sentence[:sent_len], '') != '' \
                and sent_wordcount_net > 0:
                sent_weight[sentence[:sent_len]] = sent_weight[sentence[:sent_len]] / \
                                                   sent_wordcount_net

        return sent_weight

    def get_sentences(docs, lang='en'):
        """Returns the list of sentences tokenized from the collection of documents (df).

        :param docs: The input documents storing the Scripture, default to None
        :type docs: pandas.DataFrame
        :param lang: If the value is 'chi' , the processed language is assumed to be Chinese
            otherwise, it is English, default to 'en'
        :type lang: str, optional
        :return: The list of sentences tokenized from the collection of document
        :rtype: list
        """

        join_str = '' if lang == 'chi' else ' '
        if isinstance(docs, pd.DataFrame):
            text = join_str.join(list(docs.text))
        elif isinstance(docs, pd.Series):
            text = join_str.join(list(docs))
        elif isinstance(docs, list) or isinstance(docs, np.ndarray):
            text = join_str.join(docs)
        else:
            text = docs

        if lang == 'chi':
            text = text.replace('\u3000', '')
```

```
            sentences = text.split('。')
            if sentences[-1] == '':
                sentences = np.delete(sentences, -1)
        else:
            text = text.replace('\n', '')
            text = re.sub(r'[0-9]', '', text)
            sentences = sent_tokenize(text)
            sentences = [re.sub(r'[.:•]', '', sent) for sent in sentences]
            sentences = [sent for sent in sentences if len(sent) > 10]

        return sentences

    def get_summary(sentences, sent_weight, threshold, sent_len):
        """Returns the summary of the collection of sentences.

        :param sentences: The list of target sentences for summarization, default to None
        :type sentences: list
        :param sent_weight: The dictionary of a list of sentences with their scores
            computed by their words
        :type sent_weight: collections.Counter object
        :param threshold: The minimum value of sentence weight for extracting that sentence
            as part of the final summary, default to None
        :type threshold: float
        :param sent_len: The upper bound of the number of sentences to be processed,
            default to None
        :type sent_len: int
        :return: The list of sentences of the extractive summary
        :rtype: list
        """

        sent_counter = 0
        summary = []

        for sentence in sentences:
            if sentence[:sent_len] in sent_weight and \
                sent_weight[sentence[:sent_len]] >= (threshold):
                summary.append(sentence)

        return summary

    def pos_tag(tokens, tagset=None, lang="eng"):
        """
        Use NLTK's currently recommended part of speech tagger to
        tag the given list of tokens.

            >>> from nltk.tag import pos_tag
            >>> from nltk.tokenize import word_tokenize
            >>> pos_tag(word_tokenize("John's big idea isn't all that bad.")) # doctest: +NORMALIZE_WHITESPACE
            [('John', 'NNP'), ("'s", 'POS'), ('big', 'JJ'), ('idea', 'NN'), ('is', 'VBZ'),
            ("n't", 'RB'), ('all', 'PDT'), ('that', 'DT'), ('bad', 'JJ'), ('.', '.')]
            >>> pos_tag(word_tokenize("John's big idea isn't all that bad."), tagset='universal') # doctest: +NORMALIZE_
    WHITESPACE
            [('John', 'NOUN'), ("'s", 'PRT'), ('big', 'ADJ'), ('idea', 'NOUN'), ('is', 'VERB'),
            ("n't", 'ADV'), ('all', 'DET'), ('that', 'DET'), ('bad', 'ADJ'), ('.', '.')]

        NB. Use `pos_tag_sents()` for efficient tagging of more than one sentence.

        :param tokens: Sequence of tokens to be tagged
        :type tokens: list(str)
        :param tagset: the tagset to be used, e.g. universal, wsj, brown
        :type tagset: str
        :param lang: the ISO 639 code of the language, e.g. 'eng' for English, 'rus' for Russian
        :type lang: str
        :return: The tagged tokens
        :rtype: list(tuple(str, str))
        """
        tagger = _get_tagger(lang)
        return _pos_tag(tokens, tagset, tagger, lang)

    def preprocess_sent(text):
        """Preprocesses English text by tokenizing text into sentences of words,
        converting text to lower case, removing stopwords, lemmatize text, and
        tagging text with Part-of-Speech (POS).

        :param text: The text to be preprocessed, default to None
        :type text: str
        :return: The list of preprocessed and tagged sentences (word, pos)
        :rtype: list of tuples (str, str)
        """

        if isinstance(text, list) or isinstance(text, np.ndarray):
            text = ' '.join(text)

        # print("Preprocessing text ...")

        # Tokenize text into sentences
```

```
        sentences = sent_tokenize(text)

        # Convert text to lowercase and tokenize text into words
        sentences = [word_tokenize(sentence.lower()) for sentence in sentences]

        # Remove stopwords
        stop_words = set(stopwords.words('english'))
        sentences = [[word for word in sentence if word not in stop_words] for sentence in sentences]

        # Lemmatize text
        lemmatizer = WordNetLemmatizer()
        sentences = [[lemmatizer.lemmatize(word) for word in sentence] for sentence in sentences]

        # Tag text with POS
        sentences = [pos_tag(sentence) for sentence in sentences]

        return sentences

    def sent_tokenize(text, language="english"):
        """
        Return a sentence-tokenized copy of *text*,
        using NLTK's recommended sentence tokenizer
        (currently :class:`.PunktSentenceTokenizer`
        for the specified language).

        :param text: text to split into sentences
        :param language: the model name in the Punkt corpus
        """
        tokenizer = load(f"tokenizers/punkt/{language}.pickle")
        return tokenizer.tokenize(text)

    def summary_chi(docs, weight=1.5, sent_len=8):
        """Returns an extractive summary of a collection of Chinese sentences.

        :param docs: The collection of target documents for summarization,
            default to None
        :type docs: pandas.DataFrame or pandas.Series or numpy.ndarray or list
        :param weight: The factor to be multiplied to the threshold, which
            determines the sentences as the summary, default to 1.5
        :type weight: float, optional
        :param sent_len: The upper bound of the number of sentences to be processed,
            default to 8
        :type sent_len: int, optional
        :return: The list of sentences of the extractive summary
        :rtype: list
        """

        lang = 'chi'
        util.set_lang(lang)
        diction = util.get_diction(docs)
        sentences = get_sentences(docs, lang)

        sent_scores = get_sent_scores(sentences, diction, sent_len)
        threshold = np.mean(list(sent_scores.values()))
        return get_summary(sentences, sent_scores, weight * threshold, sent_len)

    def summary_en(docs, sent_len=8):
        """Returns an extractive summary of a collection of English sentences.

        :param docs: The collection of target documents for summarization,
            default to None
        :type docs: pandas.DataFrame or pandas.Series or numpy.ndarray or list or text
        :param sent_len: The upper bound of the number of sentences to be processed,
            default to 8
        :type sent_len: int, optional
        :return: The list of sentences of the extractive summary
        :rtype: list
        """

        join_str = ' '
        if isinstance(docs, pd.DataFrame):
            text = join_str.join(list(docs.text))
        elif isinstance(docs, pd.Series):
            text = join_str.join(list(docs))
        elif isinstance(docs, list) or isinstance(docs, np.ndarray):
            text = join_str.join(docs)
        else:
            text = docs

        tagged_sentences = preprocess_sent(text)

        # Compute sentence scores
        sentence_scores = []
        for sentence in tagged_sentences:
            score = 0
            for word, pos in sentence:
```

```
                # Filter with nouns and verbs
                if pos.startswith('NN') or pos.startswith('VB'):
                    score += 1
            sentence_scores.append(score)

        # Extract top scoring sentences
        top_sentences = sorted(range(len(sentence_scores)), \
                               key=lambda i: sentence_scores[i], \
                               reverse=True)[:sent_len]

        # Build a summary
        sentences = sent_tokenize(text)
        # summary = ' '.join([sentences[i] for i in top_sentences])
        summary = [sentences[i] for i in top_sentences]

        return summary

    def word_tokenize(text, language="english", preserve_line=False):
        """
        Return a tokenized copy of *text*,
        using NLTK's recommended word tokenizer
        (currently an improved :class:`.TreebankWordTokenizer`
        along with :class:`.PunktSentenceTokenizer`
        for the specified language).

        :param text: text to split into words
        :type text: str
        :param language: the model name in the Punkt corpus
        :type language: str
        :param preserve_line: A flag to decide whether to sentence tokenize the text or not.
        :type preserve_line: bool
        """
        sentences = [text] if preserve_line else sent_tokenize(text, language)
        return [
            token for sent in sentences for token in _treebank_word_tokenizer.tokenize(sent)
        ]

    5. Submodule tm:
    class BTM:
        """The BTM object for BERTopic modeling.

        :cvar num_topics: The number of topics to be built from the modeling,
            default to 10
        :vartype num_topics: int
        :ivar textfile: The filename of the text file to be processed
        :vartype textfile: str
        :ivar chi: The flag indicating whether the processed text is in Chinese or not,
            True stands for Traditional Chinese or False for English
        :vartype chi: bool
        :ivar num_topics: The number of topics set for the topic model
        :vartype num_topics: int
        :ivar docs: The collection of the original documents to be processed
        :vartype docs: pandas.DataFrame or list
        :ivar pro_docs: The collection of documents, in form of list of lists of words
            after text preprocessing
        :vartype pro_docs: list
        :ivar dictionary: The dictionary of word ids with their tokenized words
            from preprocessed documents ('pro_docs')
        :vartype dictionary: gensim.corpora.Dictionary
        :ivar corpus: The list of documents, where each document is a list of tuples
            (word id, word frequency in the particular document)
        :vartype corpus: list
        :ivar model: The BERTopic model object
        :vartype model: bertopic.BERTopic
        :ivar embed: The flag indicating whether the BERTopic model is trained
            with the BERT pretrained model
        :vartype embed: bool
        :ivar bmodel: The BERT pretrained model
        :vartype bmodel: transformers.BertModel
        :ivar bt_vectorizer: The vectorizer extracted from the BERTopic model
            for model evaluation
        :vartype bt_vectorizer: sklearn.feature_extraction.text.CountVectorizer
        :ivar bt_analyzer: The analyzer extracted from the BERTopic model
            for model evaluation
        :vartype bt_analyzer: functools.partial
        :ivar cleaned_docs: The list of documents (string) built by grouping
            the original documents by the topics created from the BERTopic model
        :vartype cleaned_docs: list
        """

        def __init__(self, textfile, chi=False, num_topics=15, embed=True):
            """Constructor method.
            """

            self.textfile = textfile
            self.chi = chi
```

```
        self.num_topics = num_topics
        self.docs = None
        self.pro_docs = None
        self.dictionary = None
        self.corpus = None
        self.model = None

        self.embed = embed
        self.bmodel = None
        self.bt_vectorizer = None
        self.bt_analyzer = None
        self.cleaned_docs = None


    def preprocess(self):
        """Process the original English documents (cwordtm.tm.BTM.docs)
        by invoking cwordtm.tm.process_text, and build a dictionary and
        a corpus from the preprocessed documents for the BERTopic model.
        """

        self.pro_docs = [process_text(doc) for doc in self.docs]

        # Create a dictionary and corpus for the BERTopic model
        self.dictionary = corpora.Dictionary(self.pro_docs)
        self.corpus = [self.dictionary.doc2bow(doc) for doc in self.pro_docs]

    def preprocess_chi(self):
        """Process the original Chinese documents (cwordtm.tm.BTM.docs)
        by tokenizing text, removing stopwords, and building a dictionary
        and a corpus from the preprocessed documents for the BERTopic model.
        """

        # Build stop words
        stop_file = files('cwordtm.data').joinpath("tc_stopwords_2.txt")
        stopwords = [k[:-1] for k in open(stop_file, encoding='utf-8')\
                     .readlines() if k != '']

        # Tokenize"the text using Jieba
        dict_file = files('cwordtm.data').joinpath("user_dict_4.txt")
        jieba.load_userdict(str(dict_file))
        docs = [jieba.cut(doc) for doc in self.docs]

        # Replace special characters
        docs = [[word.replace('\u3000', ' ') for word in doc] \
                                    for doc in docs]

        # Remove stop words
        self.pro_docs = [' '.join([word for word in doc if word not in stopwords]) \
                                       for doc in docs]

        self.pro_docs = [doc.split() for doc in self.pro_docs]

        # Create a dictionary and corpus
        self.dictionary = corpora.Dictionary(self.pro_docs)
        self.corpus = [self.dictionary.doc2bow(doc) for doc in self.pro_docs]


    def fit(self):
        """Build the BERTopic model for English text with the created corpus
        and dictionary.
        """

        j_pro_docs = [" ".join(doc) for doc in self.pro_docs]

        if self.embed:
            self.bmodel = BertModel.from_pretrained('bert-base-uncased')
            self.model = BERTopic(language='english',
                                  calculate_probabilities=True,
                                  embedding_model=self.bmodel,
                                  nr_topics=self.num_topics)
        else:
            self.model = BERTopic(language='english',
                                  calculate_probabilities=True,
                                  nr_topics=self.num_topics)

        _, _ = self.model.fit_transform(j_pro_docs)


    def fit_chi(self):
        """Build the BERTopic model for Chinese text with the created corpus
        and dictionary.
        """

        j_pro_docs = [" ".join(doc) for doc in self.pro_docs]

        if self.embed:
```

```
            self.bmodel = BertModel.from_pretrained('bert-base-chinese')
            self.model = BERTopic(language='chinese (traditional)',
                                  calculate_probabilities=True,
                                  embedding_model=self.bmodel,
                                  nr_topics=self.num_topics)
        else:
            self.model = BERTopic(language='chinese (traditional)',
                                  calculate_probabilities=True,
                                  nr_topics=self.num_topics)

        _, _ = self.model.fit_transform(j_pro_docs)


    def show_topics(self):
        """Shows the topics with their keywords from the built BERTopic model.
        """

        print("\nTopics from BERTopic Model:")
        for topic in self.model.get_topic_freq().Topic:
            if topic == -1: continue
            twords = [word for (word, _) in self.model.get_topic(topic)]
            print(f"Topic {topic}: {' | '.join(twords)}")


    def pre_evaluate(self):
        """Prepare the original documents per built topic for model evaluation.
        """

        doc_df = pd.DataFrame({"Document": self.docs,
                       "ID": range(len(self.docs)),
                       "Topic": self.model.topics_})
        documents_per_topic = doc_df.groupby(['Topic'], \
                              as_index=False).agg({'Document': ' '.join})
        self.cleaned_docs = self.model._preprocess_text(\
                             documents_per_topic.Document.values)

        # Extract vectorizer and analyzer from BERTopic
        self.bt_vectorizer = self.model.vectorizer_model
        self.bt_analyzer = self.bt_vectorizer.build_analyzer()

    def evaluate(self):
        """Computes and outputs the coherence score.
        """

        try:
            self.pre_evaluate()

            # Extract features for Topic Coherence evaluation
            # words = self.bt_vectorizer.get_feature_names_out()
            tokens = [self.bt_analyzer(doc) for doc in self.cleaned_docs]

            self.dictionary = corpora.Dictionary(tokens)
            self.corpus = [self.dictionary.doc2bow(doc) for doc in tokens]

            topic_words = [[words for words, _ in self.model.get_topic(topic)]
                           for topic in range(len(set(self.model.topics_))-1)]

            coherence = CoherenceModel(topics=topic_words, texts=tokens, corpus=self.corpus,
                                                           dictionary=self.dictionary, coherence='c_v')\
                        .get_coherence()

            if math.isnan(coherence):
                print("** No coherence score computed!")
            else:
                print(f"  Coherence: {coherence}")
        except:
            print("** No coherence score computed!")

    def viz(self):
        """Visualize the built BERTopic model through Intertopic Distance Map,
        Topic Word Score Charts, and Topic Similarity Matrix.
        """

        print("\nBERTopic Model Visualization:")

        # Intertopic Distance Map
        try:
            self.model.visualize_topics().show()
        except:
            print("** No Intertopic Distance Map shown for your text!")

        # Visualize Terms (Topic Word Scores)
        try:
            self.model.visualize_barchart().show()
        except:
            print("** No chart of Topic Word Scores shown for your text!")
```

```
            # Visualize Topic Similarity
            try:
               self.model.visualize_heatmap().show()
            except:
                print("** No heatmap of Topic Similarity shown for your text!")

            print("  If no visualization is shown,")
            print("    you may execute the following commands one-by-one:")
            print("      btm.model.visualize_topics()")
            print("      btm.model.visualize_barchart()")
            print("      btm.model.visualize_heatmap()")
            print()

    class LDA:
        """The LDA object for Latent Dirichlet Allocation (LDA) modeling.

        :cvar num_topics: The number of topics to be built from the modeling,
            default to 10.
        :vartype num_topics: int
        :ivar textfile: The filename of the text file to be processed
        :vartype textfile: str
        :ivar chi: The flag indicating whether the processed text is in Chinese or not,
            True stands for Traditional Chinese or False for English
        :vartype chi: bool
        :ivar num_topics: The number of topics set for the topic model
        :vartype num_topics: int
        :ivar docs: The collection of the original documents to be processed
        :vartype docs: pandas.DataFrame or list
        :ivar pro_docs: The collection of documents, in form of list of lists of words
            after text preprocessing
        :vartype pro_docs: list
        :ivar dictionary: The dictionary of word ids with their tokenized words
            from preprocessed documents ('pro_docs')
        :vartype dictionary: gensim.corpora.Dictionary
        :ivar corpus: The list of documents, where each document is a list of tuples
            (word id, word frequency in the particular document)
        :vartype corpus: list
        :ivar model: The LDA model object
        :vartype model: gensim.models.LdaModel
        :ivar vis_data: The LDA model's prepared data for visualization
        :vartype vis_data: pyLDAvis.PreparedData
        """

        def __init__(self, textfile, chi=False, num_topics=15):
            """Constructor method.
            """

            self.textfile = textfile
            self.chi = chi
            self.num_topics = num_topics
            self.docs = None
            self.pro_docs = None
            self.dictionary = None
            self.corpus = None
            self.model = None
            self.vis_data = None


        def preprocess(self):
            """Process the original English documents (cwordtm.tm.LDA.docs)
            by invoking cwordtm.tm.process_text, and build a dictionary and
            a corpus from the preprocessed documents for the LDA model.
            """

            self.pro_docs = [process_text(doc) for doc in self.docs]

            # Create a dictionary and corpus for the LDA model
            self.dictionary = corpora.Dictionary(self.pro_docs)
            self.corpus = [self.dictionary.doc2bow(doc) for doc in self.pro_docs]

        def preprocess_chi(self):
            """Process the original Chinese documents (cwordtm.tm.LDA.docs)
            by tokenizing text, removing stopwords, and building a dictionary
            and a corpus from the preprocessed documents for the LDA model.
            """

            # Build stop words
            stop_file = files('cwordtm.data').joinpath("tc_stopwords_2.txt")
            stopwords = [k[:-1] for k in open(stop_file, encoding='utf-8')\
                         .readlines() if k != '']

            # Tokenize"the text using Jieba
            dict_file = files('cwordtm.data').joinpath("user_dict_4.txt")
            jieba.load_userdict(str(dict_file))
            docs = [jieba.cut(doc) for doc in self.docs]
```

```
            # Replace special characters
            docs = [[word.replace('\u3000', ' ') for word in doc] \
                                        for doc in docs]

            # Remove stop words
            self.pro_docs = [' '.join([word for word in doc if word not in stopwords]) \
                                          for doc in docs]

            self.pro_docs = [doc.split() for doc in self.pro_docs]

            # Create a dictionary and corpus
            self.dictionary = corpora.Dictionary(self.pro_docs)
            self.corpus = [self.dictionary.doc2bow(doc) for doc in self.pro_docs]


        def fit(self):
            """Build the LDA model with the created corpus and dictionary.
            """

            self.model = models.LdaModel(self.corpus,
                                num_topics=self.num_topics,
                                id2word=self.dictionary,
                                passes=10)

        def viz(self):
            """Shows the Intertopic Distance Map for the built LDA model.
            """

            self.vis_data = gensimvis.prepare(self.model, self.corpus, self.dictionary)
            pyLDAvis.enable_notebook()
            pyLDAvis.display(self.vis_data)
            print("If no visualization is shown,")
            print("  you may execute the following commands to show the visualization:")
            print("    > import pyLDAvis")
            print("    > pyLDAvis.display(lda.vis_data)")

        def show_topics(self):
            """Shows the topics with their keywords from the built LDA model.
            """

            print("\nTopics from LDA Model:")
            pprint(self.model.print_topics())

        def evaluate(self):
            """Computes and outputs the coherence score, perplexity, topic diversity,
                and topic size distribution.
            """

            # Compute coherence score
            coherence_model = CoherenceModel(model=self.model,
                                             texts=self.pro_docs,
                                             dictionary=self.dictionary,
                                             coherence='c_v')
            print(f"  Coherence: {coherence_model.get_coherence()}")

            # Compute perplexity
            perplexity = self.model.log_perplexity(self.corpus)
            print(f"  Perplexity: {perplexity}")

            # Compute topic diversity
            topic_sizes = [len(self.model[self.corpus[i]]) for i in range(len(self.corpus))]
            total_docs = sum(topic_sizes)
            topic_diversity = sum([(size/total_docs)**2 for size in topic_sizes])
            print(f"  Topic diversity: {topic_diversity}")

            # Compute topic size distribution
            # topic_sizes = [len(self.model[self.corpus[i]]) for i in range(len(self.corpus))]
            topic_size_distribution = max(topic_sizes) / sum(topic_sizes)
            print(f"  Topic size distribution: {topic_size_distribution}\n")

    class NMF:
        """The NMF object for Non-negative Matrix Factorization (NMF) modeling.

        :cvar num_topics: The number of topics to be built from the modeling,
            default to 10.
        :vartype num_topics: int
        :ivar textfile: The filename of the text file to be processed
        :vartype textfile: str
        :ivar chi: The flag indicating whether the processed text is in Chinese or not,
            True stands for Traditional Chinese or False for English
        :vartype chi: bool
        :ivar num_topics: The number of topics set for the topic model
        :vartype num_topics: int
        :ivar docs: The collection of the original documents to be processed
        :vartype docs: pandas.DataFrame or list
```

```
:ivar pro_docs: The collection of documents, in form of list of lists of words
    after text preprocessing
:vartype pro_docs: list
:ivar dictionary: The dictionary of word ids with their tokenized words
    from preprocessed documents ('pro_docs')
:vartype dictionary: gensim.corpora.Dictionary
:ivar corpus: The list of documents, where each document is a list of tuples
    (word id, word frequency in the particular document)
:vartype corpus: list
:ivar model: The NMF model object
:vartype model: gensim.models.Nmf
"""

def __init__(self, textfile, chi=False, num_topics=15):
    """Constructor method.
    """

    self.textfile = textfile
    self.chi = chi
    self.num_topics = num_topics
    self.docs = None
    self.pro_docs = None
    self.dictionary = None
    self.corpus = None
    self.model = None


def preprocess(self):
    """Process the original English documents (cwordtm.tm.NMF.docs)
    by invoking cwordtm.tm.process_text, and build a dictionary
    and a corpus from the preprocessed documents for the NMF model.
    """

    self.pro_docs = [process_text(doc) for doc in self.docs]

    # Create a dictionary and corpus for the NMF model
    self.dictionary = corpora.Dictionary(self.pro_docs)
    self.corpus = [self.dictionary.doc2bow(doc) for doc in self.pro_docs]

def preprocess_chi(self):
    """Process the original Chinese documents (cwordtm.tm.NMF.docs)
    by tokenizing text, removing stopwords, and building a dictionary
    and a corpus from the preprocessed documents for the NMF model.
    """

    # Build stop words
    stop_file = files('cwordtm.data').joinpath("tc_stopwords_2.txt")
    stopwords = [k[:-1] for k in open(stop_file, encoding='utf-8')\
                 .readlines() if k != '']

    # Tokenize"the text using Jieba
    dict_file = files('cwordtm.data').joinpath("user_dict_4.txt")
    jieba.load_userdict(str(dict_file))
    docs = [jieba.cut(doc) for doc in self.docs]

    # Replace special characters
    docs = [[word.replace('\u3000', ' ') for word in doc] \
                                  for doc in docs]

    # Remove stop words
    self.pro_docs = [' '.join([word for word in doc if word not in stopwords]) \
                                     for doc in docs]

    self.pro_docs = [doc.split() for doc in self.pro_docs]

    # Create a dictionary and corpus
    self.dictionary = corpora.Dictionary(self.pro_docs)
    self.corpus = [self.dictionary.doc2bow(doc) for doc in self.pro_docs]


def fit(self):
    """Build the NMF model with the created corpus and dictionary.
    """

    self.model = models.Nmf(self.corpus,
                            num_topics=self.num_topics)

def show_topics_words(self):
    """Shows the topics with their keywords from the built NMF model.
    """

    print("\nTopics-Words from NMF Model:")
    for topic_id in range(self.model.num_topics):
        topic_words = self.model.show_topic(topic_id, topn=10)
        print(f"Topic {topic_id+1}:")
        for word_id, prob in topic_words:
```

```
                # word = self.dictionary.id2token[int(word_id)]
                word = self.dictionary[int(word_id)]
                print("%s (%.6f)" %(word, prob))
            print()

    def evaluate(self):
        """Computes and outputs the coherence score, topic diversity,
        and topic size distribution.
        """

        # Compute coherence score
        coherence_model = CoherenceModel(model=self.model,
                                         texts=self.pro_docs,
                                         dictionary=self.dictionary,
                                         coherence='c_v')
        print(f"  Coherence: {coherence_model.get_coherence()}")

        # Compute topic diversity
        topic_sizes = [len(self.model[self.corpus[i]]) for i in range(len(self.corpus))]
        total_docs = sum(topic_sizes)
        topic_diversity = sum([(size/total_docs)**2 for size in topic_sizes])
        print(f"  Topic diversity: {topic_diversity}")

        # Compute topic size distribution
        # topic_sizes = [len(self.model[self.corpus[i]]) for i in range(len(self.corpus))]
        topic_size_distribution = max(topic_sizes) / sum(topic_sizes)
        print(f"  Topic size distribution: {topic_size_distribution}\n")

def btm_process(doc_file, source=0, text_col='text', cat=0, chi=False, group=True, eval=False):
    """Pipelines the BERTopic modeling.

    :param doc_file: The filename of the prescribed text file to be loaded,
        default to None
    :type doc_file: str
    :param source: The source of the prescribed document file ('doc_file'),
        where 0 refers to internal store of the package and 1 to external file,
        default to 0
    :type source: int, optional
    :param text_col: The name of the text column to be extracted, default to 'text'
    :type text_col: str, optional
    :param cat: The category indicating a subset of the Scripture to be loaded, where
        0 stands for the whole Bible, 1 for OT, 2 for NT, or one of the ten categories
        ['tor', 'oth', 'ket', 'map', 'mip', 'gos', 'nth', 'pau', 'epi', 'apo'] (See
        the package's internal file 'data/book_cat.csv'), default to 0
    :type cat: int or str, optional
    :param chi: The flag indicating whether the text is processed as Chinese (True)
        or English (False), default to False
    :type chi: bool, optional
    :param group: The flag indicating whether the loaded text is grouped by chapter,
        default to True
    :type group: bool, optional
    :param eval: The flag indicating whether the model evaluation results will be shown,
        default to False
    :type eval: bool, optional
    :return: The pipelined BTM
    :rtype: cwordtm.tm.BTM object
    """

    btm = BTM(doc_file, chi)
    if source == 0:
        btm.docs = load_bible(btm.textfile, cat=cat, group=group)
    else:
        btm.docs = load_text(btm.textfile, text_col=text_col)

    print("Corpus loaded!")

    if chi:
        btm.preprocess_chi()
        print("Chinese text preprocessed!")
        btm.fit_chi()
    else:
        btm.preprocess()
        print("Text preprocessed!")
        btm.fit()

    print("Text trained!")

    btm.show_topics()

    if eval:
        print("\nModel Evaluation Scores:")
        btm.evaluate()

    btm.viz()

    return btm
```

```
def lda_process(doc_file, source=0, text_col='text', cat=0, chi=False, group=True, eval=False):
    """Pipelines the LDA modeling.

    :param doc_file: The filename of the prescribed text file to be loaded,
        default to None
    :type doc_file: str
    :param source: The source of the prescribed document file ('doc_file'),
        where 0 refers to internal store of the package and 1 to external file,
        default to 0
    :type source: int, optional
    :param text_col: The name of the text column to be extracted, default to 'text'
    :type text_col: str, optional
    :param cat: The category indicating a subset of the Scripture to be loaded, where
        0 stands for the whole Bible, 1 for OT, 2 for NT, or one of the ten categories
        ['tor', 'oth', 'ket', 'map', 'mip', 'gos', 'nth', 'pau', 'epi', 'apo'] (See
        the package's internal file 'data/book_cat.csv'), default to 0
    :type cat: int or str, optional
    :param chi: The flag indicating whether the text is processed as Chinese (True)
        or English (False), default to False
    :type chi: bool, optional
    :param group: The flag indicating whether the loaded text is grouped by chapter,
        default to True
    :type group: bool, optional
    :param eval: The flag indicating whether the model evaluation results will be shown,
        default to False
    :type eval: bool, optional
    :return: The pipelined LDA
    :rtype: cwordtm.tm.LDA object
    """

    lda = LDA(doc_file, chi)
    if source == 0:
        lda.docs = load_bible(lda.textfile, cat=cat, group=group)
    else:
        lda.docs = load_text(lda.textfile, text_col=text_col)

    print("Corpus loaded!")

    if chi:
        lda.preprocess_chi()
    else:
        lda.preprocess()
    print("Text preprocessed!")

    lda.fit()
    print("Text trained!")
    lda.viz()
    print("Visualization prepared!")
    lda.show_topics()

    if eval:
        print("\nModel Evaluation Scores:")
        lda.evaluate()

    return lda

def load_bible(textfile, cat=0, group=True):
    """Loads and returns the Bible Scripture from the prescribed internal
    file ('textfile').

    :param textfile: The package's internal Bible text from which the text is loaded,
        either World English Bible ('web.csv') or Chinese Union Version (Traditional)
        ('cuv.csv'), default to None
    :type textfile: str
    :param cat: The category indicating a subset of the Scripture to be loaded, where
        0 stands for the whole Bible, 1 for OT, 2 for NT, or one of the ten categories
        ['tor', 'oth', 'ket', 'map', 'mip', 'gos', 'nth', 'pau', 'epi', 'apo'] (See
        the package's internal file 'data/book_cat.csv'), default to 0
    :type cat: int or str, optional
    :param group: The flag indicating whether the loaded text is grouped by chapter,
        default to True
    :type group: bool, optional
    :return: The collection of Scripture loaded
    :rtype: pandas.DataFrame
    """

    # textfile = "web.csv"
    scfile = files('cwordtm.data').joinpath(textfile)
    print("Loading Bible '%s' ..." %scfile)
    df = pd.read_csv(scfile)

    cat_list = ['tor', 'oth', 'ket', 'map', 'mip',\
                'gos', 'nth', 'pau', 'epi', 'apo']
    cat = str(cat)
    if cat == '1' or cat == 'ot':
```

```
            df = util.extract(df, testament=0)
        elif cat == '2' or cat == 'nt':
            df = util.extract(df, testament=1)
        elif cat in cat_list:
            df = util.extract(df, category=cat)

        if group:
            # Group verses into chapters
            df = df.groupby(['book_no', 'chapter'])\
                           .agg({'text': lambda x: ' '.join(x)})\
                    .reset_index()

        df.text = df.text.str.replace('  ', '')
        return list(df.text)

    def load_text(textfile, text_col='text'):
        """Loads and returns the list of documents from the prescribed file ('textfile').

        :param textfile: The prescribed text file from which the text is loaded,
            default to None
        :type textfile: str
        :param text_col: The name of the text column to be extracted, default to 'text'
        :type text_col: str, optional
        :return: The list of documents loaded
        :rtype: list
        """

        docs = pd.read_csv(textfile)
        return list(docs[text_col])

    def nmf_process(doc_file, source=0, text_col='text', cat=0, chi=False, group=True, eval=False):
        """Pipelines the NMF modeling.

        :param doc_file: The filename of the prescribed text file to be loaded,
            default to None
        :type doc_file: str
        :param source: The source of the prescribed document file ('doc_file'),
            where 0 refers to internal store of the package and 1 to external file,
            default to 0
        :type source: int, optional
        :param text_col: The name of the text column to be extracted, default to 'text'
        :type text_col: str, optional
        :param cat: The category indicating a subset of the Scripture to be loaded, where
            0 stands for the whole Bible, 1 for OT, 2 for NT, or one of the ten categories
            ['tor', 'oth', 'ket', 'map', 'mip', 'gos', 'nth', 'pau', 'epi', 'apo'] (See
            the package's internal file 'data/book_cat.csv'), default to 0
        :type cat: int or str, optional
        :param chi: The flag indicating whether the text is processed as Chinese (True)
            or English (False), default to False
        :type chi: bool, optional
        :param group: The flag indicating whether the loaded text is grouped by chapter,
            default to True
        :type group: bool, optional
        :param eval: The flag indicating whether the model evaluation results will be shown,
            default to False
        :type eval: bool, optional
        :return: The pipelined NMF
        :rtype: cwordtm.tm.NMF object
        """

        nmf = NMF(doc_file, chi)
        if source == 0:
            nmf.docs = load_bible(nmf.textfile, cat=cat, group=group)
        else:
            nmf.docs = load_text(nmf.textfile, text_col=text_col)

        print("Corpus loaded!")

        if chi:
            nmf.preprocess_chi()
        else:
            nmf.preprocess()
        print("Text preprocessed!")

        nmf.fit()
        print("Text trained!")
        nmf.show_topics_words()

        if eval:
            print("\nModel Evaluation Scores:")
            nmf.evaluate()

        return nmf

    def pprint(object, stream=None, indent=1, width=80, depth=None, *,
               compact=False, sort_dicts=True, underscore_numbers=False):
```

```
        """Pretty-print a Python object to a stream [default is sys.stdout]."""
        printer = PrettyPrinter(
            stream=stream, indent=indent, width=width, depth=depth,
            compact=compact, sort_dicts=sort_dicts,
            underscore_numbers=underscore_numbers)
        printer.pprint(object)

    def process_text(doc):
        """Processes the English text through tokenization, converting to lower case,
        removing all digits, stemming, and removing punctuations and stopwords.

        :param doc: The prescribed text, in form of a string, to be processed,
            default to None
        :type doc: str
        :return: The list of the processed strings
        :rtype: list
        """

        # List of punctuation
        punc = list(set(string.punctuation))

        # List of stop words
        add_stop = []
        stop_words = ENGLISH_STOP_WORDS.union(add_stop)

        doc = TweetTokenizer().tokenize(doc)
        doc = [each.lower() for each in doc]
        doc = [re.sub('[0-9]+', '', each) for each in doc]
        doc = [SnowballStemmer('english').stem(each) for each in doc]
        doc = [w for w in doc if w not in punc]
        doc = [w for w in doc if w not in stop_words]
        return doc

    6. Submodule util:
    def add_chi_vocab():
        """Loads the Chinese Bible vocabulary from the internal file 'bible_vocab.txt',
        and adds to the Jieba word list for future tokenization
        """

        vocab_file = files('cwordtm.data').joinpath('bible_vocab.txt')
        print("Loading Chinese vocabulary '%s' ..." %vocab_file)
        with open(vocab_file, 'r', encoding='utf8') as f:
            vocab_list = f.readlines()
            for vocab in vocab_list:
                jieba.add_word(vocab.replace('\n', ''), freq=1000)

    def chi_sent_terms(text):
        """Returns the list of Chinese words tokenized from the input text.

        :param text: The input Chinese text to be tokenized, default to None
        :type text: str
        :return: The list of Chinese words
        :rtype: list
        """

        text = re.sub("[、．。，：！？『』「」〔〕]", "", text)
        terms = []
        for t in jieba.cut(text, cut_all=False):
            if t not in stops:
                terms.append(t)
        return terms

    def chi_stops():
        """Loads the common Chinese (Traditional) vocabulary to Jieba for
        future tokenization, and the Chinese stopwords for future
        wordcloud plotting.

        :return: The list of stopwords for wordcloud plotting
        :rtype: list
        """

        dict_file = files('cwordtm.dictionary').joinpath('dict.txt.big.txt')
        cloud_file = files('cwordtm.dictionary').joinpath('stopWord_cloudmod.txt')
        jieba.set_dictionary(dict_file)
        with open(cloud_file, 'r', encoding='utf-8-sig') as f:
            return f.read().split('\n')

    def clean_sentences(sentences):
        """Cleans the list of sentences by invoking the function preprocess_text.

        :param sentences: The list of sentences to be cleaned, default to None
        :type sentences: list
        :return: The list of cleaned sentences
        :rtype: list
        """
```

```
        cleaned = []
        for sentence in sentences:
            cleaned_sent = preprocess_text(sentence)
            if len(cleaned_sent) > 0:
                cleaned.append(cleaned_sent)

        return cleaned

    def clean_text(df, text_col='text'):
        """Cleans the text from the Scripture stored in the DataFrame 'df',
        by removing all digits, replacing newline by a space, removing
        English stopwords, converting all characters to lower case, and
        removing all characters except alphanumeric and whitespace.

        :param df: The input DataFrame storing the Scripture, default to None
        :type df: pandas.DataFrame
        :param text_col: The name of the text column to be extracted, default to 'text'
        :type text_col: str, optional
        :return: The cleaned text in a DataFrame
        :rtype: pandas.DataFrame
        """

        df[text_col] = [re.sub(r'\d+', '', str(v).replace('\n', ' ')) for v in df[text_col]]
        for sw in stopwords.words('english'):
            df[text_col] = [v.replace(' ' + sw + ' ', ' ') for v in df[text_col]]

        df[text_col] = df[text_col].apply(lambda v: " ".join(w.lower() for w in v.split()))
        df[text_col] = df[text_col].str.replace('[^\w\s]', '', regex=True)
        return df

    def extract(df, testament=-1, category='', book=0, chapter=0, verse=0):
        """Extracts a subset of the Scripture stored in a DataFrame by testament,
        category, or book/chapter/verse.

        :param df: The collection of the Bible Scripture with columns 'book',
            'book_no', 'chapter', 'verse', 'text', 'testament', 'category',
            'cat', and 'cat_no', default to None
        :type df: pandas.DataFrame
        :param testament: The prescribed testament to be extracted,
            -1 stands for no prescription, 0 for OT, or 1 for NT,
            default to -1
        :type testament: int, optional
        :param category: The prescribed category to be extracted, and
            it should be either a full category name or a short name with
            3 lower-case letters from a list of 10 categories, default to ''
        :type category: str, optional
        :param book: The prescribed Bible book to be extracted, and
            it should be either a 3-letter short book name or a book number
            from 1 to 66, default to 0
        :type book: str, int, optional
        :param chapter: The prescribed chapter or a tuple indicating the range of
            chapters of a Bible book to be extracted, default to 0
        :type chapter: int or tuple, optional
        :param verse: The prescribed verse or a tuple indicating the range of verses
            from a chapter of a Bible book to be extracted, default to 0
        :type verse: int or tuple, optional
        :return: The subset of the input Scripture, if any, otherwise,
            the message 'No scripture is extracted!'
        :rtype: pandas.DataFrame or str
        """

        no_ret = "No scripture is extracted!"
        sub_df = pd.DataFrame()  # Empty DataFrame
        isbook = ischapter = False

        if (testament > -1) & (testament < 2):
            sub_df = df[df.testament==int(testament)]
        elif category != '':
            if category in get_list(df, column='category'):
                sub_df = df[df.category==category]
            elif category in get_list(df, column='cat'):
                sub_df = df[df.cat==category]
        elif book in get_list(df, column='book'):
            sub_df = df[df.book==book]
            isbook = True
        elif isinstance(book, int):
            if book > 0 & book < 67:
                sub_df = df[df.book_no==book]
                isbook = True
        elif isinstance(book, tuple):
            if (book[0] <= book[1]) & (book[0] > 0) & (book[1] < 67):
                sub_df = df[(df.book_no >= book[0]) & (df.book_no <= book[1])]
                isbook = True

        if isbook & (len(sub_df) > 0) & (chapter != 0):
            if isinstance(chapter, int):
```

```
                sub_df = sub_df[sub_df.chapter==chapter]
                ischapter = True
            elif isinstance(chapter, tuple):
                if chapter[0] <= chapter[1]:
                    sub_df = sub_df[(sub_df.chapter >= chapter[0]) & (sub_df.chapter <= chapter[1])]
                    ischapter = True

            if ischapter & (len(sub_df) > 0) & (verse != 0):
                if isinstance(verse, int):
                    sub_df = sub_df[sub_df.verse==verse]
                elif isinstance(verse, tuple):
                    if verse[0] <= verse[1]:
                        sub_df = sub_df[(sub_df.verse >= verse[0]) & (sub_df.verse <= verse[1])]

        if len(sub_df) > 0:
            return sub_df.copy()
        else:
            return no_ret

    def extract2(df, filter=''):
        """Extracts a subset of the Scripture through a specific filter string by
        invoking the function 'util.extract'.

        :param df: The collection of the Bible Scripture, default to None
        :type df: pandas.DataFrame
        :param filter: The prescribed filter string with the format
            '<book> <chapter>:<verse>[-<verse2>]' for extracting a range of verses
            in the Scripture, default to ''
        :type filter: str, optional
        :return: The prescribed range of verses from the input Scripture, or
            the whole Scripture if the filter string is empty
        :rtype: pandas.DataFrame
        """

        chapter = verse = 0

        if filter == '':
            return df
        else:
            parts = filter.split()
            book = parts[0]
            if len(parts) > 1:
                parts = parts[1].split(':')
                if parts[0] == '':
                    chapter = 0
                else:
                    chapter = int(parts[0])

                if (len(parts) > 1):
                    if (parts[1] != ''):
                        parts = parts[1].split('-')
                        if parts[0] == '':
                            verse = 1
                        else:
                            verse = int(parts[0])

                        if (len(parts) > 1):
                            if (parts[1] == ''):
                                verse = (verse, 999)
                            else:
                                verse = (verse, int(parts[1]))

            return extract(df, book=book, chapter=chapter, verse=verse)

    def get_diction(docs):
        """Determines which is the target language, English or Chinese,
        in order to build a dictionary of words with their frequencies.

        :param docs: The collection of documents, default to None
        :type docs: pandas.DataFrame or list
        :return: The dictionary of words with their frequencies
        :rtype: dict
        """

        if glang == 'en':
            return get_diction_en(docs)
        else:
            return get_diction_chi(docs)

    def get_diction_chi(docs):
        """Tokenizes the collection of Chinese documents and builds a dictionary
        of words with their frequencies.

        :param docs: The collection of documents, default to None
        :type docs: pandas.DataFrame or list
        :return: The dictionary of words with their frequencies
```

```
        :rtype: dict
        """

        if isinstance(docs, pd.DataFrame):
            docs = ''.join(list(docs.text))
        elif isinstance(docs, pd.Series):
            docs = ''.join(list(docs))
        elif isinstance(docs, list) or isinstance(docs, np.ndarray):
            docs = ''.join(docs)

        text = docs.replace('\u3000', '')
        text = re.sub("[、 . 。 , ! ? 『』 「」 〔〕]", "", text)

        terms = []
        for t in jieba.cut(text, cut_all=False):
            if t not in stops:
                terms.append(t)

        diction = Counter(terms)
        return diction

    def get_diction_en(docs):
        """Tokenizes the collection of English documents and builds a dictionary
        of words with their frequencies.

        :param docs: The collection of text, default to None
        :type docs: pandas.DataFrame or list
        :return: The dictionary of words with their frequencies
        :rtype: dict
        """

        if isinstance(docs, pd.DataFrame):
            docs = ' '.join(list(docs.text))
        elif isinstance(docs, pd.Series):
            docs = ' '.join(list(docs))
        elif isinstance(docs, list) or isinstance(docs, np.ndarray):
            docs = ' '.join(docs)

        words = word_tokenize(docs)
        stem = PorterStemmer()

        terms = []
        for t in words:
            t = stem.stem(t)
            if t not in stops:
                terms.append(t)

        diction = Counter(terms)
        return diction

    def get_list(df, column='book'):
        """Extracts and returns the prescribed column from the Scripture
        stored in the DataFrame 'df'.

        :param df: The input DataFrame storing the Scripture, default to None
        :type df: pandas.DataFrame
        :param column: The column by which the Scriture is grouped, default to 'book'
        :type column: str, optional
        :return: The grouped Scripture
        :rtype: pandas.DataFrame
        """

        if column in list(df.columns):
            return list(df[column].unique())
        else:
            return "No such column!"

    def get_sent_terms(text):
        """Determines how to tokenize the input text, based on the global language
        setting, either English ('en') or Traditional Chinese ('chi').

        :param text: The input text to be tokenized, default to None
        :type text: str
        :return: The list of tokenized words
        :rtype: list
        """

        if glang == 'en':
            return word_tokenize(text)
        else:
            return chi_sent_terms(text)

    def get_text(df, text_col='text'):
        """Extracts and returns the text from the Scripture
        stored in the DataFrame 'df' after joining the list of text into a string
        and removing all the ideographic spaces ('\u3000') from the text.
```

```
        :param df: The input DataFrame storing the Scripture, default to None
        :type df: pandas.DataFrame
        :param text_col: The name of the text column to be extracted, default to 'text'
        :type text_col: str, optional
        :return: The extracted text
        :rtype: str
        """

        return ' '.join(list(df[text_col])).replace('\u3000', '')

    def get_text_list(df, text_col='text'):
        """Extracts and returns the list of text from the Scripture
        stored in the DataFrame 'df' after removing all the ideographic spaces
        ('\u3000') from the text.

        :param df: The input DataFrame storing the Scripture, default to None
        :type df: pandas.DataFrame
        :param text_col: The name of the text column to be extracted, default to 'text'
        :type text_col: str, optional
        :return: The extracted text
        :rtype: list
        """

        return df[text_col].apply(lambda x: x.replace('\u3000', '')).tolist()

    def group_text(df, column='chapter'):
        """Groups the Bible Scripture in the DataFrame 'df' by the prescribed column, and
        'df' should include columns 'book', 'book_no', 'chapter', 'verse', 'text',
        'testament', 'category', 'cat', and 'cat_no'.

        :param df: The input DataFrame storing the Scripture, default to None
        :type df: pandas.DataFrame
        :param column: The column by which the Scriture is grouped, default to 'chapter'
        :type column: str, optional
        :return: The grouped Scripture
        :rtype: pandas.DataFrame
        """

        gdf = df.groupby(['book_no', column])\
                            .agg({'text': lambda x: ''.join(x)})\
                    .reset_index()
        return gdf

    def is_chi():
        """Checks whether the Chinese language flag is set.

        :return: True if the Chinese language flag (chi_flag) is set,
            False otherwise
        :rtype: bool
        """

        return chi_flag

    def load_text(filepath, nr=0, info=False):
        """Loads and returns the text from the prescribed file path ('filepath').

        :param filepath: The prescribed filepath from which the text is loaded,
            default to None
        :type filepath: str
        :param nr: The number of rows of text to be loaded; 0 represents all rows,
            default to 0
        :type nr: int, optional
        :param info: The flag whether the dataset information is shown,
            default to False
        :type info: bool, optional
        :return: The collection of text with the prescribed number of rows loaded
        :rtype: pandas.DataFrame
        """

        print("Loading file '%s' ..." %filepath)
        df = pd.read_csv(filepath)
        if nr > 0:
           print("Initial Records:")
           print(df.head(int(nr)))
        if info:
            print("\nDataset Information:")
            df.info()
        return df

    def load_word(ver='web.csv', nr=0, info=False):
        """Loads and returns the text from the prescribed internal file ('ver').

        :param ver: The package's internal Bible text from which the text is loaded,
            either World English Bible ('web.csv') or Chinese Union Version
            (Traditional)('cuv.csv'), default to 'web.csv'
```

```
        :type ver: str, optional
        :param nr: The number of rows of Scripture to be loaded; 0 represents all rows,
            default to 0
        :type nr: int, optional
        :param info: The flag whether the dataset information is shown,
            default to False
        :type info: bool, optional
        :return: The collection of Scripture with the prescribed number of rows loaded
        :rtype: pandas.DataFrame
        """

        scfile = files('cwordtm.data').joinpath(ver)
        print("Loading file '%s' ..." %scfile)
        df = pd.read_csv(scfile)
        if nr > 0:
           print("Initial Records:")
           df.head(int(nr))
        if info:
            print("\nDataset Information:")
            df.info()
        return df

    def preprocess_text(text):
        """Preprocesses English text by converting text to lower case, removing
        special characters and digits, removing punctuations, removing stopwords,
        removing short words, and Lemmatize text.

        :param text: The text to be preprocessed, default to None
        :type text: str
        :return: The preprocessed text
        :rtype: str
        """

        if isinstance(text, list) or isinstance(text, np.ndarray):
            text = ' '.join(text)

        # print("Preprocessing text ...")

        # Convert text to lowercase
        text = text.lower()

        # Remove special characters and digits
        text = re.sub(r'[^a-zA-Z\s]', '', text)

        # Remove punctuation
        text = text.translate(str.maketrans('', '', string.punctuation))

        # Remove stopwords
        text = " ".join([word for word in nltk.word_tokenize(text) \
                        if word.lower() not in stopwords.words('english')])

        # Remove short words (length < 3)
        text = " ".join([word for word in nltk.word_tokenize(text) if len(word) >= 3])

        # Lemmatization
        lemmatizer = WordNetLemmatizer()
        text = " ".join([lemmatizer.lemmatize(word) for word in nltk.word_tokenize(text)])

        return text

    def set_lang(lang='en'):
        """Sets the prescribed language (English or Chinese (Traditional))
        for further text processing.

        :param lang: The prescribed language for text processing, where
            'en' stands for English or 'chi' for Traditonal Chinese,
            default to 'en'
        :type lang: str, optional
        """

        global glang, stops
        glang = lang
        if glang == 'en':  # English
            stops = set(stopwords.words("english"))
        else:  # Chinese (Traditional)
            add_chi_vocab()
            stops = chi_stops()
            chi_flag = True

    def word_tokenize(text, language="english", preserve_line=False):
        """
        Return a tokenized copy of *text*,
        using NLTK's recommended word tokenizer
        (currently an improved :class:`.TreebankWordTokenizer`
        along with :class:`.PunktSentenceTokenizer`
        for the specified language).
```

```
        :param text: text to split into words
        :type text: str
        :param language: the model name in the Punkt corpus
        :type language: str
        :param preserve_line: A flag to decide whether to sentence tokenize the text or not.
        :type preserve_line: bool
        """
        sentences = [text] if preserve_line else sent_tokenize(text, language)
        return [
            token for sent in sentences for token in _treebank_word_tokenizer.tokenize(sent)
        ]

    7. Submodule version:
    8. Submodule viz:
    def chi_wordcloud(docs, figsize=(15, 10), bg='white', image=0):
        """Prepare and show a Chinese wordcloud

        :param docs: The collection of Chinese documents for preparing a wordcloud,
            default to None
        :type docs: pandas.DataFrame
        :param figsize: Size (width, height) of word cloud, default to (15, 10)
        :type figsize: tuple
        :param bg: The background color (name) of the wordcloud, default to 'white'
        :type bg: str, optional
        :param image: The filename of the image as the mask of the wordcloud,
            default to 0 (No image mask)
        :type image: int or str, optional
        """

        util.set_lang('chi')
        diction = util.get_diction(docs)

        if image == 0:
            mask = None
        elif image == 1:  # Internal image file ('heart.jpg')
            img_file = files('cwordtm.images').joinpath('heart.jpg')
            mask = np.array(Image.open(img_file))
        elif isinstance(image, str) and len(image) > 0:
            mask = np.array(Image.open(image))
        else:
            mask = None

        font_file = files('cwordtm.data').joinpath('msyh.ttc')
        wordcloud = WordCloud(background_color=bg, colormap='Set2',
                              mask=mask, font_path=str(font_file)) \
                        .generate_from_frequencies(frequencies=diction)

        plot_cloud(wordcloud, figsize=figsize)

    def plot_cloud(wordcloud, figsize):
        """Plot the prepared 'wordcloud'
        :param wordcloud: The WordCloud object for plotting, default to None
        :type wordcloud: WordCloud object
        :param figsize: Size (width, height) of word cloud, default to None
        :type figsize: tuple
        """

        plt.figure(figsize=figsize)
        plt.imshow(wordcloud)
        plt.axis("off");

    def show_wordcloud(docs, clean=False, figsize=(12, 8), bg='white', image=0):
        """Prepare and show a wordcloud

        :param docs: The collection of documents for preparing a wordcloud,
            default to None
        :type docs: pandas.DataFrame
        :param clean: The flag whether text preprocessing is needed,
            default to False
        :type clean: bool, optional
        :param figsize: Size (width, height) of word cloud, default to (12, 8)
        :type figsize: tuple
        :param bg: The background color (name) of the wordcloud, default to 'white'
        :type bg: str, optional
        :param image: The filename of the image as the mask of the wordcloud,
            default to 0 (No image mask)
        :type image: int or str, optional
        """

        if image == 0:
            mask = None
        elif image == 1:  # Internal image file ('heart.jpg')
            img_file = files('cwordtm.images').joinpath('heart.jpg')
            mask = np.array(Image.open(img_file))
        elif isinstance(image, str) and len(image) > 0:
```

```
    mask = np.array(Image.open(image))
else:
    mask = None

if isinstance(docs, pd.DataFrame):
    docs = ' '.join(list(docs.text))
elif isinstance(docs, pd.Series):
    docs = ' '.join(list(docs))
elif isinstance(docs, list) or isinstance(docs, np.ndarray):
    docs = ' '.join(docs)

if clean:
    docs = util.preprocess_text(docs)

wordcloud = WordCloud(background_color=bg, colormap='Set2', mask=mask) \
                .generate(docs)

plot_cloud(wordcloud, figsize=figsize)
```

In [4]:
```
# Show execution time
df = util.load_text("BBC/BBC News Train.csv", timing=True)
```

```
Loading file 'BBC/BBC News Train.csv' ...
Finished 'load_text' in 0.0805 secs
```

In [5]:
```
# Execute and show code
df = util.load_text("BBC/BBC News Train.csv", code=1)
```

```
Loading file 'BBC/BBC News Train.csv' ...

def load_text(filepath, nr=0, info=False):
    """Loads and returns the text from the prescribed file path ('filepath').

    :param filepath: The prescribed filepath from which the text is loaded,
        default to None
    :type filepath: str
    :param nr: The number of rows of text to be loaded; 0 represents all rows,
        default to 0
    :type nr: int, optional
    :param info: The flag whether the dataset information is shown,
        default to False
    :type info: bool, optional
    :return: The collection of text with the prescribed number of rows loaded
    :rtype: pandas.DataFrame
    """

    print("Loading file '%s' ..." %filepath)
    df = pd.read_csv(filepath)
    if nr > 0:
       print("Initial Records:")
       print(df.head(int(nr)))
    if info:
        print("\nDataset Information:")
        df.info()
    return df
```

In [6]:
```
# Show code without execution
df = util.load_text("BBC/BBC News Train.csv", code=2)
```

```
def load_text(filepath, nr=0, info=False):
    """Loads and returns the text from the prescribed file path ('filepath').

    :param filepath: The prescribed filepath from which the text is loaded,
        default to None
    :type filepath: str
    :param nr: The number of rows of text to be loaded; 0 represents all rows,
        default to 0
    :type nr: int, optional
    :param info: The flag whether the dataset information is shown,
        default to False
    :type info: bool, optional
    :return: The collection of text with the prescribed number of rows loaded
    :rtype: pandas.DataFrame
    """

    print("Loading file '%s' ..." %filepath)
    df = pd.read_csv(filepath)
    if nr > 0:
       print("Initial Records:")
       print(df.head(int(nr)))
    if info:
        print("\nDataset Information:")
        df.info()
    return df
```

In [7]:
```
# Add timing and code reveal features to some other function
from importlib_resources import files
files = meta.addin(files)
files(code=2)
```

```
@package_to_anchor
def files(anchor: Optional[Anchor] = None) -> Traversable:
    """
    Get a Traversable resource for an anchor.
    """
    return from_package(resolve(anchor))
```

2. Utility Features

Load BBC News

In [8]:
```
bbc_file = "BBC/BBC News Train.csv"
df = util.load_text(bbc_file, info=True)
```

```
Loading file 'BBC/BBC News Train.csv' ...

Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1490 entries, 0 to 1489
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   ArticleId  1490 non-null   int64
 1   Text       1490 non-null   object
 2   Category   1490 non-null   object
dtypes: int64(1), object(2)
memory usage: 35.0+ KB
```

Preprocessing Text

In [9]:
```
text_list = util.get_text_list(df.iloc[:500], text_col='Text')
text = util.preprocess_text(text_list)
```

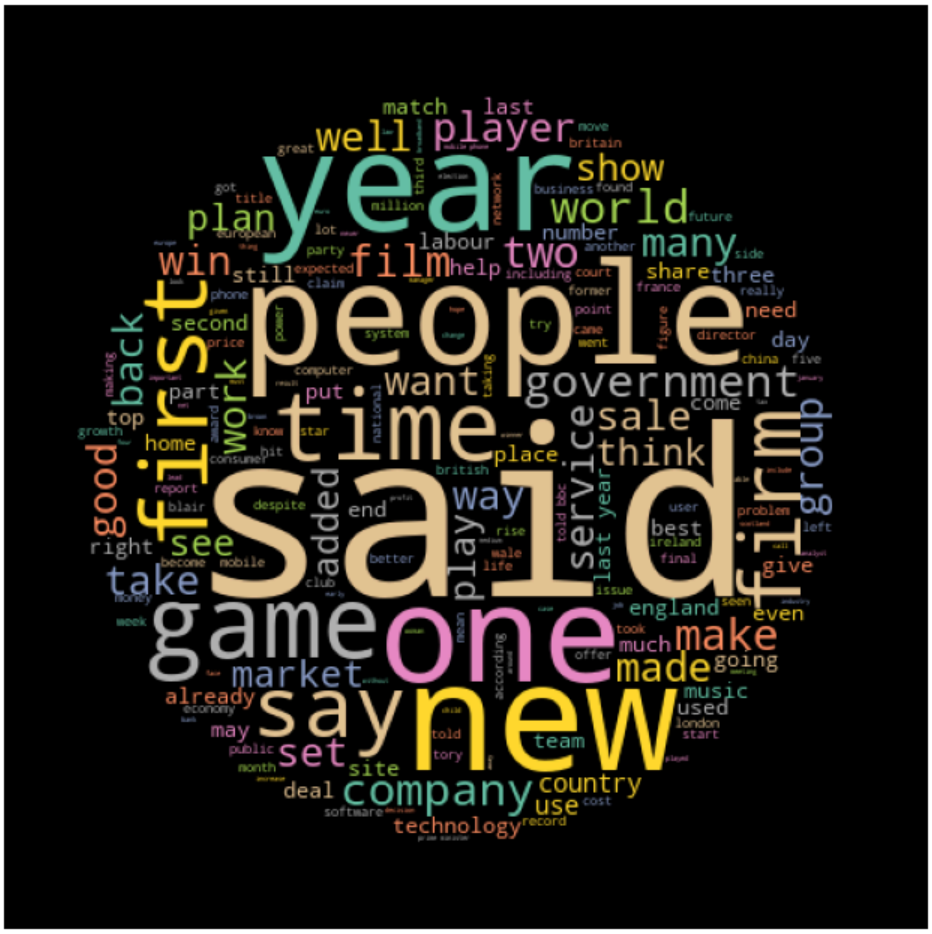
3. Text Visualization - Word Cloud

In [10]:
```
# White background with no image mask
viz.show_wordcloud(text)
```

```
D:\Dev\Anaconda3\lib\site-packages\wordcloud\wordcloud.py:106: MatplotlibDeprecationWarning: The get_cmap function w
as deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or
``matplotlib.colormaps.get_cmap(obj)`` instead.
  self.colormap = plt.cm.get_cmap(colormap)
```

In [11]:
```
# Black background with the prescribed image as the mask
viz.show_wordcloud(text, bg='black', image='images/disc.png')
```

```
D:\Dev\Anaconda3\lib\site-packages\wordcloud\wordcloud.py:106: MatplotlibDeprecationWarning: The get_cmap function w
as deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or
``matplotlib.colormaps.get_cmap(obj)`` instead.
  self.colormap = plt.cm.get_cmap(colormap)
```



4. Text Summarization

In [12]:
```
news = df.iloc[:5]['Text']  # "df" stores previously loaded text
ta.summary_en(news, sent_len=5)
```

Out[12]:
```
['but ms cooper  who now runs her own consulting business  told a jury in new york on wednesday that external auditors arthur andersen had approved worldcom s accounting in early 2001 and 2002. she said andersen had given a  green light  to the procedures and practices used by worldcom.',
 'cynthia cooper  worldcom s ex-head of internal accounting  alerted directors to irregular accounting practices at the us telecoms giant in 2002. her warnings led to the collapse of the firm following the discovery of an $11bn (£5.7bn) accounting fraud.',
 'prosecution lawyers have argued that mr ebbers orchestrated a series of accounting tricks at worldcom  ordering employees to hide expenses and inflate revenues to meet wall street earnings estimates.',
 'people around the world are saying:  i m ok  but the world isn t .  there may be a perception that war  terrorism and religious and political divisions are making the world a worse place  even though that has not so far been reflected in global economic performance  says the bbc s elizabeth blunt.',
 'the university of california said the trial in the case is scheduled to begin in october 2006. it joined the lawsuit in december 2001alleging  massive insider trading  and fraud  claiming it had lost $145m on its investments in the company.']
```

5. Topic Modeling

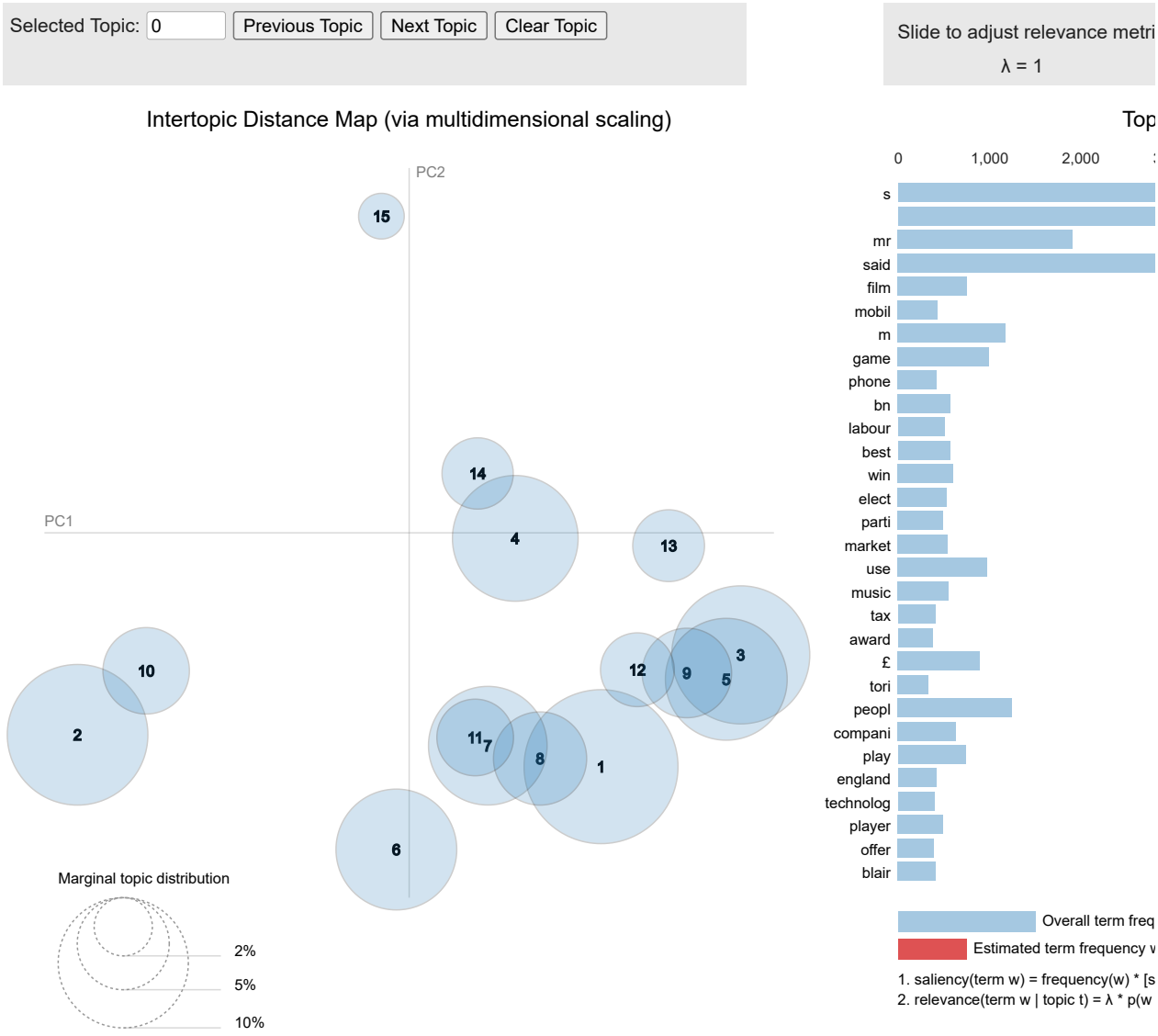
LDA Model

In [13]:
```
doc_file = "BBC/BBC News Train.csv"
lda = tm.lda_process(doc_file, source=1, text_col='Text', eval=True)
```

```
Corpus loaded!
Text preprocessed!
Text trained!
If no visualization is shown,
  you may execute the following commands to show the visualization:
    > import pyLDAvis
    > pyLDAvis.display(lda.vis_data)
Visualization prepared!

Topics from LDA Model:
[(0,
  '0.015*"s" + 0.011*"" + 0.010*"lse" + 0.008*"said" + 0.007*"deutsch" + '
  '0.006*"boers" + 0.006*"london" + 0.005*"bid" + 0.005*"offer" + '
  '0.004*"price"'),
 (1,
  '0.025*"" + 0.019*"s" + 0.012*"game" + 0.009*"said" + 0.007*"film" + '
  '0.005*"year" + 0.005*"new" + 0.005*"time" + 0.005*"music" + 0.005*"use"'),
 (2,
  '0.024*"s" + 0.020*"" + 0.009*"play" + 0.009*"game" + 0.009*"england" + '
  '0.009*"win" + 0.009*"said" + 0.006*"player" + 0.006*"match" + 0.006*"year"'),
 (3,
  '0.020*"s" + 0.014*"said" + 0.011*"" + 0.006*"peopl" + 0.006*"game" + '
  '0.005*"film" + 0.005*"time" + 0.004*"mr" + 0.004*"work" + 0.004*"say"'),
 (4,
  '0.025*"s" + 0.018*"mr" + 0.016*"said" + 0.016*"" + 0.014*"film" + '
  '0.011*"best" + 0.008*"award" + 0.007*"labour" + 0.006*"parti" + '
  '0.006*"actor"'),
 (5,
  '0.027*"s" + 0.019*"" + 0.011*"said" + 0.006*"new" + 0.006*"t" + 0.005*"bn" '
  '+ 0.005*"offer" + 0.005*"deal" + 0.005*"m" + 0.004*"firm"'),
 (6,
  '0.037*"" + 0.018*"said" + 0.016*"s" + 0.009*"year" + 0.007*"rate" + '
  '0.006*"peopl" + 0.006*"growth" + 0.006*"economi" + 0.005*"govern" + '
  '0.005*"month"'),
 (7,
  '0.021*"" + 0.020*"s" + 0.008*"said" + 0.008*"chelsea" + 0.006*"win" + '
  '0.006*"game" + 0.005*"liverpool" + 0.005*"play" + 0.005*"leagu" + '
  '0.005*"year"'),
 (8,
  '0.026*"" + 0.014*"said" + 0.012*"use" + 0.012*"mobil" + 0.012*"phone" + '
  '0.012*"s" + 0.010*"peopl" + 0.007*"technolog" + 0.006*"year" + 0.006*"mr"'),
 (9,
  '0.023*"" + 0.022*"said" + 0.016*"s" + 0.016*"mr" + 0.007*"say" + '
  '0.007*"govern" + 0.007*"tax" + 0.007*"elect" + 0.006*"labour" + '
  '0.006*"parti"'),
 (10,
  '0.023*"" + 0.018*"said" + 0.011*"s" + 0.006*"use" + 0.005*"court" + '
  '0.005*"year" + 0.005*"olymp" + 0.004*"lord" + 0.004*"attack" + '
  '0.004*"govern"'),
 (11,
  '0.034*"" + 0.025*"s" + 0.019*"m" + 0.011*"said" + 0.010*"£" + 0.009*"year" '
  '+ 0.006*"bn" + 0.006*"music" + 0.006*"sale" + 0.005*"new"'),
 (12,
  '0.032*"" + 0.017*"s" + 0.016*"said" + 0.008*"india" + 0.008*"hunt" + '
  '0.007*"year" + 0.006*"market" + 0.006*"servic" + 0.005*"new" + '
  '0.005*"deficit"'),
 (13,
  '0.016*"said" + 0.013*"" + 0.012*"mr" + 0.012*"s" + 0.008*"compani" + '
  '0.007*"yuko" + 0.007*"blog" + 0.006*"peopl" + 0.006*"site" + 0.006*"firm"'),
 (14,
  '0.026*"" + 0.021*"said" + 0.014*"s" + 0.006*"peopl" + 0.006*"uk" + '
  '0.006*"year" + 0.005*"tv" + 0.005*"consum" + 0.004*"broadband" + '
  '0.004*"new"')]

Model Evaluation Scores:
  Coherence: 0.3667641353303661
  Perplexity: -7.883481066035845
  Topic diversity: 0.0008048220614871675
  Topic size distribution: 0.0019006244909041542
```

In [14]:
```
# LDA Model Visualization
import pyLDAvis
pyLDAvis.display(lda.vis_data)
```

Out[14]:



BERTopic Model

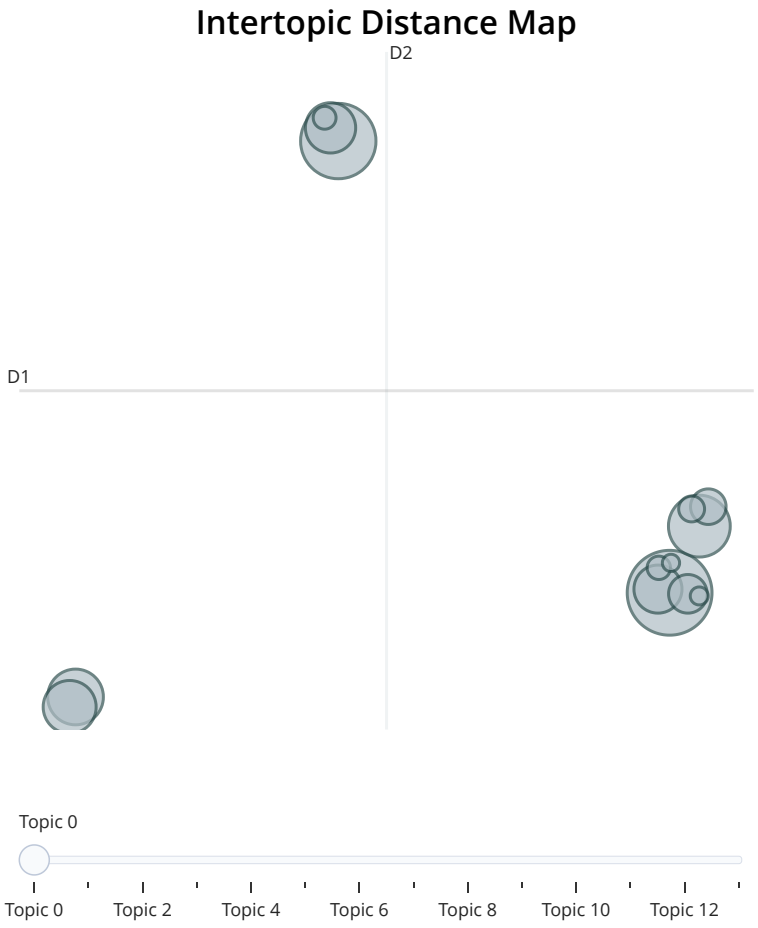
In [15]:

```
btm = tm.btm_process(doc_file, source=1, text_col='Text', eval=True)
```

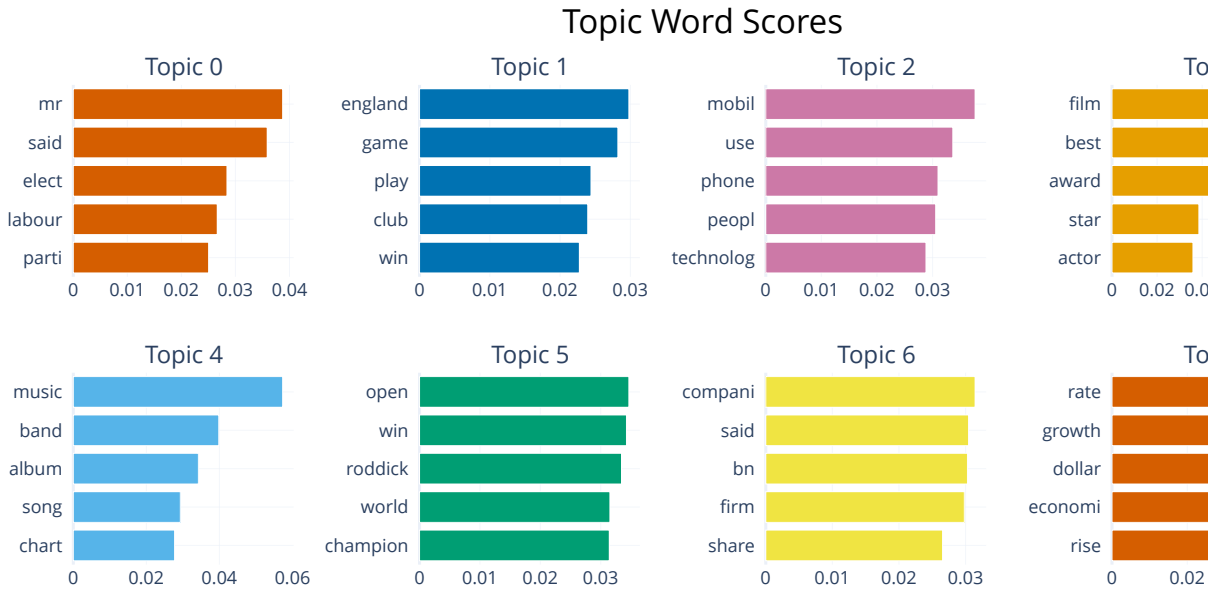
```
Corpus loaded!
Text preprocessed!
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertModel: ['cls.predictio
ns.decoder.weight', 'cls.predictions.transform.LayerNorm.bias', 'cls.seq_relationship.bias', 'cls.seq_relationship.w
eight', 'cls.predictions.transform.dense.bias', 'cls.predictions.transform.dense.weight', 'cls.predictions.bias', 'c
ls.predictions.transform.LayerNorm.weight']
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with
another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactl
y identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
D:\Dev\Anaconda3\lib\site-packages\hdbscan\hdbscan_.py:1170: DeprecationWarning: `alltrue` is deprecated as of NumPy
1.25.0, and will be removed in NumPy 2.0. Please use `all` instead.
  self._all_finite = is_finite(X)
```

```
Text trained!

Topics from BERTopic Model:
Topic 0: mr | said | elect | labour | parti | govern | blair | say | minist | tori
Topic 1: england | game | play | club | win | player | ireland | wale | half | chelsea
Topic 2: mobil | use | phone | peopl | technolog | said | servic | digit | gadget | user
Topic 3: film | best | award | star | actor | oscar | nomin | director | actress | year
Topic 4: music | band | album | song | chart | record | singl | singer | year | best
Topic 5: open | win | roddick | world | champion | year | olymp | seed | match | final
Topic 6: compani | said | bn | firm | share | year | market | car | sale | mr
Topic 7: rate | growth | dollar | economi | rise | said | year | econom | deficit | figur
Topic 8: virus | mail | spam | site | secur | user | program | attack | use | softwar
Topic 9: game | consol | nintendo | gamer | xbox | titl | soni | halo | ds | develop
Topic 10: yuko | russian | russia | gazprom | tax | oil | khodorkovski | compani | auction | court
Topic 11: test | kenteri | iaaf | cont | greek | olymp | drug | thanou | athlet | ban
Topic 12: sri | disast | lanka | indonesia | countri | econom | tsunami | damag | thailand | said
Topic 13: oil | price | crude | barrel | gas | suppli | opec | oecd | bp | deal

Model Evaluation Scores:
  Coherence: 0.6341205792622266

BERTopic Model Visualization:
```

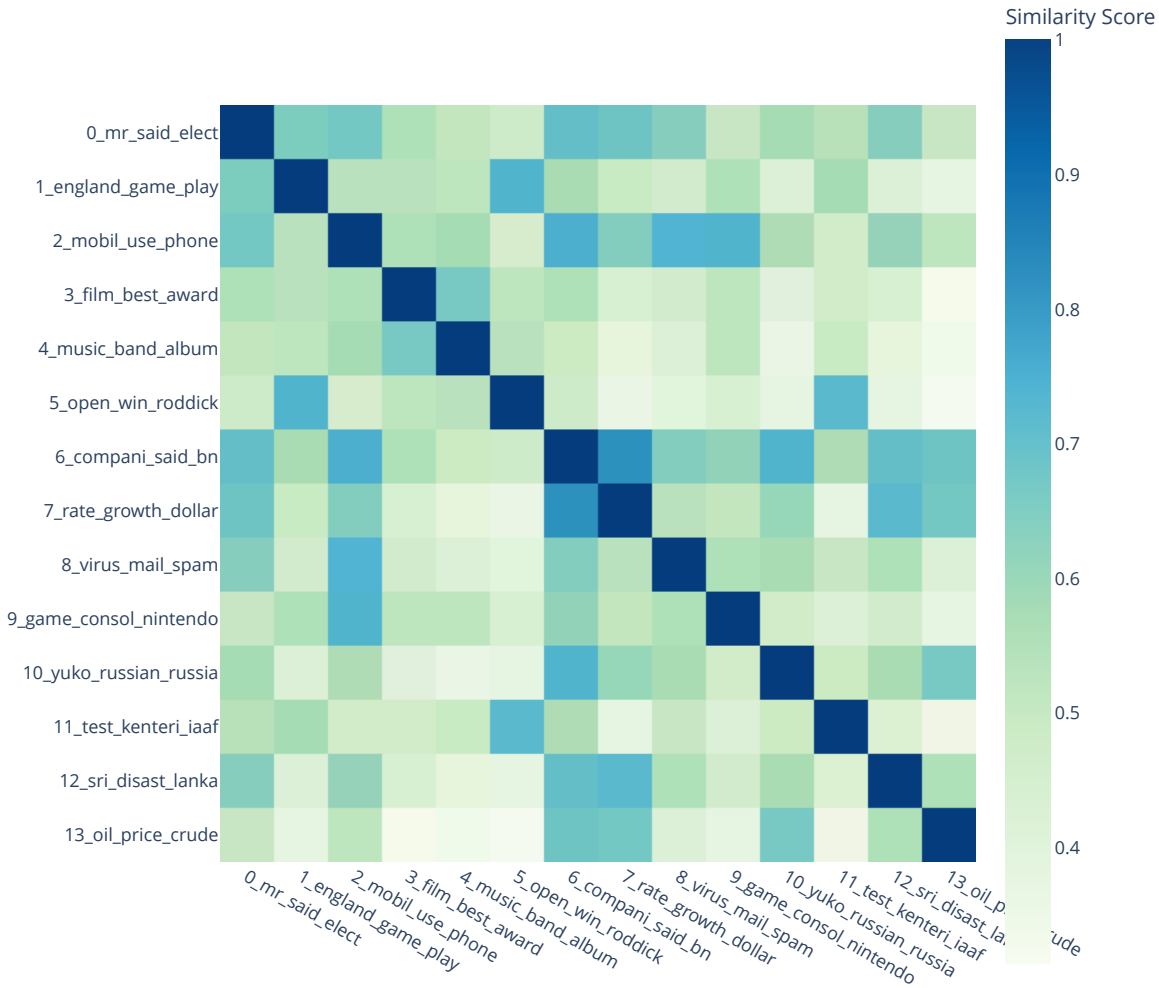
```
D:\Dev\Anaconda3\lib\site-packages\plotly\io\_renderers.py:395: DeprecationWarning:

distutils Version classes are deprecated. Use packaging.version instead.

D:\Dev\Anaconda3\lib\site-packages\plotly\io\_renderers.py:395: DeprecationWarning:

distutils Version classes are deprecated. Use packaging.version instead.
```




```
D:\Dev\Anaconda3\lib\site-packages\plotly\io\_renderers.py:395: DeprecationWarning:

distutils Version classes are deprecated. Use packaging.version instead.

D:\Dev\Anaconda3\lib\site-packages\plotly\io\_renderers.py:395: DeprecationWarning:

distutils Version classes are deprecated. Use packaging.version instead.
```




```
D:\Dev\Anaconda3\lib\site-packages\plotly\io\_renderers.py:395: DeprecationWarning:

distutils Version classes are deprecated. Use packaging.version instead.

D:\Dev\Anaconda3\lib\site-packages\plotly\io\_renderers.py:395: DeprecationWarning:

distutils Version classes are deprecated. Use packaging.version instead.
```




```
If no visualization is shown,
  you may execute the following commands one-by-one:
    btm.model.visualize_topics()
    btm.model.visualize_barchart()
    btm.model.visualize_heatmap()
```