search

May 9, 2024

1 Search

In this assessment, you'll implement a basic search engine by defining your own Python classes. A **search engine** is an algorithm that takes a query and retrieves the most relevant documents for that query. In order to identify the most relevant documents, our search engine will use **term frequency–inverse document frequency** (tf–idf), an information statistic for determining the relevance of a term to each document from a corpus consisting of many documents.

The **tf–idf statistic** is a product of two values: term frequency and inverse document frequency. **Term frequency** computes the number of times that a term appears in a **document** (such as a single Wikipedia page). If we were to use only the term frequency in determining the relevance of a term to each document, then our search result might not be helpful since most documents contain many common words such as "the" or "a". In order to downweight these common terms, the **document frequency** computes the number of times that a term appears across the **corpus** of all documents.

[2]:
```
!pip install -q nb_mypy pytest ipytest
%reload_ext nb_mypy
%nb_mypy mypy-options --strict
```

```
Version 1.0.5
```

[3]:
```
import os
import math
import re

import pytest
import ipytest
ipytest.autoconfig()
import operator


def clean(token: str, pattern: re.Pattern[str] = re.compile(r"\W+")) -> str:
    """
    Returns all the characters in the token lowercased and without matches to␣
 ↪the given pattern.

    >>> clean("Hello!")
    'hello'
```

```
    """
    return pattern.sub("", token.lower())
```

1.1 Collaboration and Conduct

Students are expected to follow Washington state law on the Student Conduct Code for the University of Washington. In this course, students must:

- Indicate on your submission any assistance received, including materials distributed in this course.
- Not receive, generate, or otherwise acquire any substantial portion or walkthrough to an assessment.
- Not aid, assist, attempt, or tolerate prohibited academic conduct in others.

Update the following code cell to include your name and list your sources. If you used any kind of computer technology to help prepare your assessment submission, include the queries and/or prompts. Submitted work that is not consistent with sources may be subject to the student conduct process.

[9]:
```
your_name = "Cassie Hoang"
sources = [
    "objects.ipynb",
    "asymptopic-analysis.ipynb",
    "file-processing.ipynb"
]

assert your_name != "", "your_name cannot be empty"
assert ... not in sources, "sources should not include the placeholder ellipsis"
assert len(sources) >= 2, "must include at least 2 sources, inclusive of␣
 ↪lectures and sections"
```

1.2 Task: `Document`

Write and test a `Document` class in the code cell below that can be used to represent the text in a web page and includes methods to compute term frequency. (But not document frequency since that would require access to all the documents in the corpus.)

The `Document` class should include:

1. An initializer `__init__` takes a `str` path and initializes the document data based on the text in the specified file. Assume that the file exists, but that it could be empty. In order to implement `term_frequency` later, we'll need to precompute and save the term frequency for each term in the document in the initializer as a field by constructing a word_dictionary that maps each `str` term to its `float` term frequency. Term frequency is defined as *the count of the given term* divided by *the total number of words in the text*. > Consider the term frequencies for this document containing 4 total words: "the cutest cutest dog". > > - "the" appears 1 time out of 4 total words, so its term frequency is 0.25. > - "cutest" appears 2 times out of 4 total words, so its term frequency is 0.5. > - "dog" appears 1 time out of 4 total words, so its term frequency is 0.25.

When constructing this word_dictionary, call the `clean` function to lowercasing letters and ignore non-letter characters so that "corgi", "CoRgi", and "corgi!!" are all considered the same string to the search algorithm.

2. A method `term_frequency` that takes a given `str` term and returns its term frequency by looking it up in the precomputed word_dictionary. Remember to normalize the term before looking it up to find the corresponding match. If the term does not occur, return 0.
3. A method `get_path` that returns the `str` path of the file that this document represents.
4. A method `get_words` that returns a `set` of the unique, cleaned words in this document.
5. A method `__repr__` that returns a string representation of this document in the format `Document('{path}')` (with literal single quotes in the output) where `{path}` is the path to the document from the initializer. The `__repr__` method is called when Jupyter Notebook needs to display a `Document` as output, so we should be able to copy the string contents into a new code cell and immediately run it to create a new `Document`.

For each of the 4 methods (excluding the initializer) in the `Document` class, write a testing function that contains at least 3 `pytest`-style assertions based on your own testing corpus. As always, your test cases should expand the domain and range. Documentation strings are optional for testing functions.

We've provided some example corpuses in the `doggos` directory and the `small_wiki` directory. For this task, create your own testing corpus by creating a **New Folder** and adding your own text files to it.

> Be sure to exhaustively test your `Document` class before moving on: bugs in `Document` will make implementing the following `SearchEngine` task much more difficult.

[5]:
```
%%ipytest

class Document:
    """Represents the text in a web page and includes methods to compute term
    frequency."""

    def __init__(self, path: str):
        """This initializer takes a string path that is representative of text␣
 ↪ina specified file. This reads the
        file and saves unique words into a dictionary to keep count of how many␣
 ↪times a word appears in the text."""
        self.path = path
        self.word_dict : dict[str, int] = {}
        self.count = 0

        with open (self.path, "r") as file:
            file_lines = file.readlines()
            for line in file_lines:
                for word in line.split(" "):
                    cleaned = clean(word)
                    if cleaned != "":
```

```
                        self.count += 1
                        if cleaned not in self.word_dict.keys():
                            self.word_dict[cleaned] = 1
                        else:
                            self.word_dict[cleaned] += 1

    def term_frequency(self, word:str) -> float:
        """This method takes a given word/term and returns the term frequency,␣
 ↪which is how many times the word
        appears in the text file. If the term does not exist in the file, a␣
 ↪corresponding 0 is returned."""
        word = clean(word)
        if word not in self.word_dict.keys():
            return 0
        word_freq = self.word_dict[word] / self.count
        return word_freq

    def get_path(self) -> str:
        """This method returns a string of the name of the specified text file.
 ↪"""
        return self.path

    def get_words(self) -> set[str]:
        """This method returns a set of the key strings in the dictionary that␣
 ↪represents unique words in
        the text file."""
        return set(self.word_dict.keys())

    def __repr__(self) -> str:
        """This method returns a string output of the text file in a specified␣
 ↪Document(text file) format to display."""
        return f"Document('{self.path}')"

class TestDocument:
    """This class is used for testing the Document class. This class contains␣
 ↪several test cases using assert statements
    to test the behavior of methods within the Document class."""
    doc1 = Document("doggos/doc1.txt")
    euro = Document("small_wiki/Euro - Wikipedia.html")
    lorem = Document("lorem.txt")
    friends = Document("friends.txt")
    minion = Document("minion.txt")

    def test_term_frequency(self) -> None:
        assert self.doc1.term_frequency("dogs") == pytest.approx(1 / 5)
```

```
        assert self.euro.term_frequency("Euro") == pytest.approx(0.
↪008656632040138506)
        assert self.lorem.term_frequency("lorem") == pytest.approx(6/15)
        assert self.friends.term_frequency("friends") == pytest.approx(5/17)
        assert self.minion.term_frequency("bob") == pytest.approx(4/13)

    def test_get_words(self) -> None:
        assert self.doc1.get_words() == set("dogs are the greatest pets".
↪split())
        assert set(w for w in self.euro.get_words() if len(w) == 1) == set([
            *"0123456789acefghijklmnopqrstuvxyz".lower() # All one-letter words␣
↪in Euro
        ])
        assert self.lorem.get_words() == set("lorem ipsum dolor sit amet".
↪split())
        assert self.friends.get_words() == set("my best friends heidi crystal␣
↪bestie i love marat".split())
        assert self.minion.get_words() == set("my favorite minion bob king␣
↪stuart kevin".split())

    def test_get_path(self) -> None:
        assert self.doc1.get_path() == "doggos/doc1.txt"
        assert self.euro.get_path() == "small_wiki/Euro - Wikipedia.html"
        assert self.lorem.get_path() == "lorem.txt"
        assert self.friends.get_path() == "friends.txt"
        assert self.minion.get_path() == "minion.txt"

    def test_repr(self) -> None:
        assert repr(self.doc1) == "Document('doggos/doc1.txt')"
        assert repr(self.euro) == "Document('small_wiki/Euro - Wikipedia.html')"
        assert repr(self.lorem) == "Document('lorem.txt')"
        assert repr(self.friends) == "Document('friends.txt')"
        assert repr(self.minion) == "Document('minion.txt')"
```

```
....
[100%]
4 passed in 0.03s
```

1.3 Task: `SearchEngine`

Write and test a `SearchEngine` class in the code cell below that represents a corpus of `Document` objects and includes methods to compute the tf–idf statistic between a given query and every document in the corpus. The `SearchEngine` begins by constructing an **inverted index** that associates each term in the corpus to the list of `Document` objects that contain the term.

To iterate over all the files in a directory, call `os.listdir` to list all the file names and join the directory to the filename with `os.path.join`. The example below will print only the `.txt` files in

the `doggos` directory.

[6]:
```
path = "doggos"
extension = ".txt"
for filename in os.listdir(path):
    if filename.endswith(extension):
        print(os.path.join(path, filename))
```

```
doggos/doc2.txt
doggos/doc3.txt
doggos/doc1.txt
```

The `SearchEngine` class should include:

1. An initializer that takes a `str` path to a directory such as `"small_wiki"` and a `str` file extension and constructs an inverted index from the files in the specified directory matching the given extension. By default, the extension should be `".txt"`. Assume the string represents a valid directory, and that the directory contains only valid files. Do not recreate any behavior that is already done in the `Document` class—call the `get_words()` method! Create at most one `Document` per file.
2. A method `_calculate_idf` that takes a `str` term and returns the inverse document frequency of that term. If the term is not in the corpus, return 0. Inverse document frequency is defined by calling `math.log` on the *the total number of documents in the corpus* divided by *the number of documents containing the given term.*
3. A method `__repr__` that returns a string representation of this search engine in the format `SearchEngine('{path}')` (with literal single quotes in the output) where `{path}` is the path to the document from the initializer. The `__repr__` method is called when Jupyter Notebook needs to display a `SearchEngine` as output, so we should be able to copy the string contents into a new code cell and immediately run it to create a new `SearchEngine`.
4. A method `search` that takes a `str` **query** consisting of one or more terms and returns a `list` of relevant document paths that match at least one of the cleaned terms sorted by descending tf–idf statistic: the product of the term frequency and inverse document frequency. If there are no matching documents, return an empty list.

For each of the 3 methods (excluding the initializer) in the `SearchEngine` class, write a testing function that contains at least 3 `pytest`-style assertions based on your own testing corpus except for `SearchEngine.__repr__`, which may use the given corpuses. Documentation strings are optional for testing functions.

[7]:
```
%%ipytest

class SearchEngine:
    """This class is representative of a corpus of Document objects and␣
 ↪includes methods used to
    compute tf-idf statistics between a given querey and every document in the␣
 ↪corpus.
    Constructs an inverted index that associates each term in the corpus to the␣
 ↪list of Document
```

```
  objects that contain the term."""

  def __init__(self, path: str, extension: str = '.txt'):
      """This initializer takes a stringi path and optional extension as␣
↪parameters. Takes in
      a path to a directoy and constructs an inverted index from the files in␣
↪the specified
      directory matching the given string extension."""
      self.inverted : dict[str, list[str]] = {}
      self.count = 0
      self.path = f"SearchEngine('{path}', '{extension}')"

      for filename in os.listdir(path):
          if filename.endswith(extension):
              self.count += 1
              file = str(path) + "/" + str(filename)
              doc = Document(file)
              words = doc.get_words()
              for word in words:
                  if word not in self.inverted:
                      self.inverted[word] = []
                  self.inverted[word].append(file)

  def _calculate_idf(self, term: str) -> float:
      """This function takes in a string parameter and calculates the inverse␣
↪document frequency (IDF)
      for the given string term. It returns the IDF for the given term or )␣
↪if the term is not in the corpus."""
      if term in self.inverted:
          return math.log(self.count / len(self.inverted[term]))
      return 0

  def __repr__(self) -> str:
      """This function returns a string representations of the search engine␣
↪in a specific
      'SearchEngine('{path}')' format."""
      return self.path

  def search(self, query: str) -> list[str]:
      """This function takes a string query as a parameter. It searches for␣
↪documents relevant to
      the given query. It returns a list of relevant document paths that␣
↪match at least one of
      the cleaned terms sorted by the tf-idf statistic or an emoty list if␣
↪there are no matches."""
      docs: dict[str, float] = {}
```

```
        for word in query.split():
            cleaned = clean(word)
            cleaned_idf = self._calculate_idf(cleaned)
            if cleaned in self.inverted:
                for path in self.inverted[cleaned]:
                    doc = Document(path)
                    term_freq = doc.term_frequency(cleaned)
                    tf_idf = term_freq * cleaned_idf
                    print(cleaned, term_freq, cleaned_idf, tf_idf)

                    if path not in docs:
                        docs[path] = 0
                    docs[path] += tf_idf

        filtered = sorted(docs.items(),
                          key=operator.itemgetter(1),
                          reverse = True)
        return [path for path, tf_idf in filtered]


class TestSearchEngine:
    """This class is used for testing the SearchEngine class. This class␣
 ↪contains several test
    cases using assert statements to test the behavior of methods within the␣
 ↪SearchEngine class."""
    doggos = SearchEngine('doggos')
    small_wiki = SearchEngine("small_wiki", ".html")
    corpus = SearchEngine('corpus')

    def test_calculate_idf(self) -> None:
        assert self.corpus._calculate_idf('friends') == 1.0986122886681098
        assert self.corpus._calculate_idf('my') == 0.4054651081081644
        assert self.corpus._calculate_idf('favorite') == 1.0986122886681098

    def test_repr(self) -> None:
        assert repr(self.corpus) == "SearchEngine('corpus', '.txt')"
        assert repr(self.small_wiki) == "SearchEngine('small_wiki', '.html')"
        assert repr(self.doggos) == "SearchEngine('doggos', '.txt')"

    def test_search(self) -> None:
        assert self.doggos.search("love") == ["doggos/doc3.txt"]
        assert self.doggos.search("dogs") == ["doggos/doc3.txt", "doggos/doc1.
 ↪txt"]
        assert self.doggos.search("cats") == ["doggos/doc2.txt"]
        assert self.doggos.search("love dogs") == ["doggos/doc3.txt", "doggos/
 ↪doc1.txt"]
```

```
        assert self.small_wiki.search("data")[:10] == [
            "small_wiki/Internet privacy - Wikipedia.html",
            "small_wiki/Machine learning - Wikipedia.html",
            "small_wiki/Bloomberg L.P. - Wikipedia.html",
            "small_wiki/Waze - Wikipedia.html",
            "small_wiki/Digital object identifier - Wikipedia.html",
            "small_wiki/Chief financial officer - Wikipedia.html",
            "small_wiki/UNCF - Wikipedia.html",
            "small_wiki/Jackson 5 Christmas Album - Wikipedia.html",
            "small_wiki/KING-FM - Wikipedia.html",
            "small_wiki/The News-Times - Wikipedia.html",
        ]
        assert self.corpus.search("my") == ['corpus/friends.txt', 'corpus/
↪minion.txt']
        assert self.corpus.search("ipsum") == ['corpus/lorem.txt']
        assert self.corpus.search("love") == ['corpus/friends.txt']
        assert self.corpus.search("nothing") == []
```

```
...
[100%]
3 passed in 1.07s
```

We recommend the following iterative software development approach to implement the `search` method.

1. Write code to handle queries that contain only a single term by collecting all the documents that contain the given term, computing the tf–idf statistic for each document, and returning the list of document paths sorted by descending tf–idf statistic.
2. Write tests to ensure that your program works on single-term queries.
3. Write code to handle queries that contain more than one term by returning all the documents that match any of the terms in the query sorted by descending tf–idf statistic. The tf–idf statistic for a document that matches more than one term is defined as the sum of its constituent single-term tf–idf statistics.
4. Write tests to ensure that your program works on multi-term queries.

Here's a walkthrough of the `search` function from beginning to end. Say we have a corpus in a directory called `"doggos"` containing 3 documents with the following contents:

- `doggos/doc1.txt` with the text `Dogs are the greatest pets.`
- `doggos/doc2.txt` with the text `Cats seem pretty okay`
- `doggos/doc3.txt` with the text `I love dogs!`

The initializer should construct the following inverted index.

```
{"dogs":     [doc1, doc3],
 "are":      [doc1],
 "the":      [doc1],
 "greatest": [doc1],
```

```
 "pets":     [doc1],
 "cats":     [doc2],
 "seem":     [doc2],
 "pretty":   [doc2],
 "okay":     [doc2],
 "i":        [doc3],
 "love":     [doc3]}
```

Searching this corpus for the multi-term query `"love dogs"` should return a list `["doggos/doc3.txt", "doggos/doc1.txt"]` by:

1. Finding all documents that match at least one query term. The word `"love"` is found in `doc3` while the word `"dogs"` is found in `doc1` and `doc3`.
2. Computing the tf–idf statistic for each matching document. For each matching document, the tf–idf statistic for a multi-word query `"love dogs"` is the sum of the tf–idf statistics for `"love"` and `"dogs"` individually.
    1. For `doc1`, the sum of $0 + 0.081 = 0.081$. The tf–idf statistic for `"love"` is 0 because the term is not in `doc1`.
    2. For `doc3`, the sum of $0.366 + 0.135 = 0.501$.
3. Returning the matching document paths sorted by descending tf–idf statistic.

After completing your `SearchEngine`, run the following cell to search our small Wikipedia corpus for the query "data". Try some other search queries too!

[8]:
```
SearchEngine("small_wiki", ".html").search("data")
```

```
data 0.00017115960633290543 0.01438873745209967 2.4627706379289123e-06
data 0.0002578648788035069 7 0.01438873745209967 3.710350039221163e-06
data 4.76258513120922e-05 0.01438873745209967 6.852758704624313e-07
data 5.945303210463734e-05 0.01438873745209967 8.554540696848793e-07
data 0.0002710761724044456 0.01438873745209967 3.900443874247674e-06
data 0.00036773719048786466 0.01438873745209967 5.291273885302649e-06
data 0.003914090726615817 0.01438873745209967 5.631882382897302e-05
data 0.0002636435539151068 0.01438873745209967 3.793497878222956e-06
data 0.0002911208151382824 0.01438873745209967 4.188860975865989e-06
data 3.948199620972836e-05 0.01438873745209967 5.680960775465757e-07
data 0.00014281633818908883 0.01438873745209967 2.054946794073075e-06
data 0.0001180080245456691 0.01438873745209967 1.6979864824285662e-06
data 0.00019409937888198756 0.01438873745209967 2.7928450023485385e-06
data 2.3766517729822226e-05 0.01438873745209967 3.419701837650839e-07
data 0.00014775413711583924 0.01438873745209967 2.125995486421346e-06
data 0.00023030861354214648 0.01438873745209967 3.3138501732150326e-06
data 0.0005279831045406547 0.01438873745209967 7.5970102703799735e-06
data 0.00043645886375209136 0.01438873745209967 6.2800919991705845e-06
data 0.00016409583196586806 0.01438873745209967 2.36113184314074e-06
data 0.00015295197308045274 0.01438873745209967 2.200785783435251e-06
data 0.00014072614691809738 0.01438873745209967 2.0248715806501085e-06
```

data 0.0006207508128879693 0.01438873745209967 8.931820469822438e-06
data 0.0005167958656330749 0.01438873745209967 7.436040026924894e-06
data 0.0004589261128958238 0.01438873745209967 6.6033673483706615e-06
data 0.0001766472354707649 0.01438873745209967 2.541730692828064e-06
data 0.0001662510390689941 8 0.01438873745209967 2.392142552302522e-06
data 0.0013819789939192924 0.01438873745209967 1.9884932907821545e-05
data 0.0001217730150998538 8 0.01438873745209967 1.7521599430223661e-06
data 2.900063801403631e-05 0.01438873745209967 4.172825663273497e-07
data 0.0015214083894805478 0.01438873745209967 2.18911458736574e-05
data 0.0002999400119976004 5 0.01438873745209967 4.315758084013098e-06
data 0.0003297065611605671 0.01438873745209967 4.744061144774042e-06
data 0.0002968239833778569 4 0.01438873745209967 4.27092236631038e-06
data 0.0002915451895043732 0.01438873745209967 4.19496718720107e-06
data 0.0003590664272890485 0.01438873745209967 5.166512550125555e-06
data 0.0009600614439324117 0.01438873745209967 1.381407205462718e-05
data 0.0002937720329024677 0.01438873745209967 4.227008652203193e-06
data 6.272541947624274e-05 0.01438873745209967 9.02539592416476e-07
data 0.0004722550177095632 0.01438873745209967 6.795153460259585e-06
data 0.0002426595486532395 0.01438873745209967 3.4915645358164694e-06
data 0.0028826483169053377 0.01438873745209967 4.147766979868791e-05
data 0.0002738225629791895 0.01438873745209967 3.939960967168584e-06
data 0.0001766659600028266 7 0.01438873745209967 2.5420001152038144e-06
data 0.0002155636990730761 0.01438873745209967 3.101689470165913e-06
data 0.0003578457684737878 0.01438873745209967 5.148948810914178e-06
data 0.0006277463904582549 0.01438873745209967 9.032477998807076e-06
data 6.861534239055853e-05 0.01438873745209967 9.872881468436715e-07
data 0.0006598482349059716 0.01438873745209967 9.494383010293416e-06
data 0.0004121728378099883 0.01438873745209967 5.930646748134782e-06
data 0.0002593360995850622 4 0.01438873745209967 3.731519048781035e-06
data 0.0002563445270443476 0.01438873745209967 3.6884740969237813e-06
data 8.78348704435661e-05 0.01438873745209967 1.263832889951662e-06
data 0.0001078574125006741 2 0.01438873745209967 1.5519319907350128e-06
data 0.0004483299708585519 0.01438873745209967 6.450902242591199e-06
data 0.0003516174402250351 7 0.01438873745209967 5.05933103097738e-06
data 0.0028682882055988987 0.01438873745209967 4.1271045927316636e-05
data 0.0026194144838212635 0.01438873745209967 3.769006728593134e-05
data 4.0246307401295933e-05 0.01438873745209967 5.790935506137429e-07
data 0.0003295978905735003 0.01438873745209967 4.742497512227973e-06
data 0.0005078720162519045 0.01438873745209967 7.3076371011171505e-06
data 5.9623181492964464e-05 0.01438873745209967 8.579023045611538e-07
data 2.132332558585837e-05 0.01438873745209967 3.0681573346055545e-07
data 0.0001864280387770320 5 0.01438873745209967 2.682464103672571e-06
data 0.0001359249694168819 0.01438873745209967 1.9557886981241905e-06
data 0.0002432300956705043 0.01438873745209967 3.499773987051971e-06
data 0.0008169934640522876 0.01438873745209967 1.1755504454329796e-05
data 2.7552763542183283e-05 0.01438873745209967 3.96449480688259e-07
data 0.0007712082262210797 0.01438873745209967 1.1096712687994605e-05
data 0.0002086811352253756 3 0.01438873745209967 3.002658065964038e-06

```
[8]: ['small_wiki/Internet privacy - Wikipedia.html',
      'small_wiki/Machine learning - Wikipedia.html',
      'small_wiki/Bloomberg L.P. - Wikipedia.html',
      'small_wiki/Waze - Wikipedia.html',
      'small_wiki/Digital object identifier - Wikipedia.html',
      'small_wiki/Chief financial officer - Wikipedia.html',
      'small_wiki/UNCF - Wikipedia.html',
      'small_wiki/Jackson 5 Christmas Album - Wikipedia.html',
      'small_wiki/KING-FM - Wikipedia.html',
      'small_wiki/The News-Times - Wikipedia.html',
      'small_wiki/Robert Mercer (businessman) - Wikipedia.html',
      'small_wiki/Federal Bureau of Investigation - Wikipedia.html',
      'small_wiki/Viacom - Wikipedia.html',
      'small_wiki/Seattle Municipal Street Railway - Wikipedia.html',
      'small_wiki/Mandalay Bay - Wikipedia.html',
      'small_wiki/Hal Abelson - Wikipedia.html',
      'small_wiki/File_Googleplex-Patio-Aug-2014.JPG - Wikipedia.html',
      'small_wiki/IEEE Computer Society - Wikipedia.html',
      'small_wiki/Nintendo - Wikipedia.html',
      'small_wiki/Mountain View, California - Wikipedia.html',
      'small_wiki/University District, Seattle - Wikipedia.html',
      'small_wiki/Category_Wikipedia articles with ISNI identifiers -
     Wikipedia.html',
      'small_wiki/RealNetworks - Wikipedia.html',
      'small_wiki/Robot (dance) - Wikipedia.html',
      'small_wiki/Category_American Presbyterians - Wikipedia.html',
      'small_wiki/Category_Wikipedia articles with NLA identifiers - Wikipedia.html',
      'small_wiki/Henry Holt and Company - Wikipedia.html',
      'small_wiki/David Swinson Maynard - Wikipedia.html',
      'small_wiki/Hans-Hermann Hoppe - Wikipedia.html',
      'small_wiki/Paul Gottfried - Wikipedia.html',
      'small_wiki/MJ &amp;amp; Friends - Wikipedia.html',
      'small_wiki/Mesoparapylocheles - Wikipedia.html',
      'small_wiki/Ubisoft - Wikipedia.html',
      'small_wiki/Cultural impact of Michael Jackson - Wikipedia.html',
      'small_wiki/AIM (software) - Wikipedia.html',
      'small_wiki/Moonwalk (dance) - Wikipedia.html',
      'small_wiki/Seattle Mardi Gras riot - Wikipedia.html',
      'small_wiki/Black or White - Wikipedia.html',
      'small_wiki/Unity Tour - Wikipedia.html',
      'small_wiki/Baltimore Afro-American - Wikipedia.html',
      'small_wiki/Thriller 25_ Limited Japanese Single Collection - Wikipedia.html',
      'small_wiki/ABC (The Jackson 5 album) - Wikipedia.html',
      'small_wiki/Republican Study Committee - Wikipedia.html',
      'small_wiki/The Rolling Stone Album Guide - Wikipedia.html',
      'small_wiki/Euro - Wikipedia.html',
      'small_wiki/Living with Michael Jackson - Wikipedia.html',
```

```
'small_wiki/Republican Party of Virginia - Wikipedia.html',
'small_wiki/Humberto Gatica - Wikipedia.html',
'small_wiki/1964 Republican National Convention - Wikipedia.html',
'small_wiki/William E. Miller - Wikipedia.html',
'small_wiki/Light rail - Wikipedia.html',
'small_wiki/Fisheries science - Wikipedia.html',
'small_wiki/Steve Forbes - Wikipedia.html',
'small_wiki/In the Closet - Wikipedia.html',
'small_wiki/Andrew Breitbart - Wikipedia.html',
'small_wiki/Tenor - Wikipedia.html',
'small_wiki/Bob Corker - Wikipedia.html',
'small_wiki/Rhinoplasty - Wikipedia.html',
'small_wiki/VH1 - Wikipedia.html',
"small_wiki/Jehovah's Witnesses - Wikipedia.html",
'small_wiki/List of songs recorded by The Jackson 5 - Wikipedia.html',
'small_wiki/Airliner - Wikipedia.html',
'small_wiki/Traditionalist conservatism - Wikipedia.html',
'small_wiki/Orrin Hatch - Wikipedia.html',
'small_wiki/Jeb Bush - Wikipedia.html',
'small_wiki/William McKinley - Wikipedia.html',
'small_wiki/Alcoholic drink - Wikipedia.html',
'small_wiki/Game of Thrones - Wikipedia.html',
'small_wiki/Michael Jackson - Wikipedia.html']
```