Национальный исследовательский университет "Высшая школа экономики"
Московский институт электроники и математики
Департамент прикладной математики

Классификация задач в предметной области на основе статей из Web of Science

Мирошник В.
miroshnik.valerij.98@gmail.com
Научный руководитель: Гуськова М.
maria.guskova@rambler.ru

Аннотация

В данной работе реализована программа, которая получает информацию об интересующих пользователя статьях с сайта Web of Science. На основе этой информации строится граф цитирования, затем производится кластеризация статей методами K-Means и latent Dirichlet allocation (LDA), и их анализ с помощью multidimensional scaling (MDS) и hierarchical document clustering (HDC).

Введение

Web of Science (WoS) — поисковая платформа, объединяющая базы данных публикаций в научных журналах и патентов. WoS охватывает материалы по естественным, техническим, общественным, гуманитарным наукам и искусству. Платформа обладает встроенными возможностями поиска, анализа и управления библиографической информацией.
Задача, решаемая в рамках данной работы, заключается в том, чтобы научиться по запросу пользователя скачивать необходимые данные с WoS (названия статей, имена авторов, аннотации, цитируемые статьи), строить на основе этих данных граф цитирования, классифицировать статьи, решаемые в заданной пользователем области. Для классификации используются метод k-средних в совокупности с методом "локтя"(elbow method), латентное размещение Дирихле (LDA). Для анализа данных применены метод многомерного масштабирования (MDS) и иерархическая кластеризация.
Саму программу можно найти в данном репозитории на GitHub. Также по данной ссылке располагаются инструкции по запуску на платформах Windows и Linux.

Структура статьи:
Глава I: Методы решения поставленной задачи
Кратко обозреваются библиотеки, использованные для сбора данных с WoS и построения графа цитирования, рассматриваются методы анализа и кластеризации текста
Глава II: Описание программы
Подробное описание проделанной работы. Для наглядности данная глава поделена на 3 части:
1) Сбор данных и приведение их к виду, пригодному для анализа
2) Проведение идентификации статей и построение графа цитирования
3) Классификация статей методами k-means и LDA
Глава III: Анализ результатов
Анализ полученных результатов с помощью MDS и иерархической кластеризации, их обсуждение.
Глава IV: Заключение

1 Методы решения поставленной задачи

Для того, чтобы решить данную задачу, во-первых, нужно научиться собирать данные с WoS. Основная проблема состоит в том, что все ссылки зашифрованы, и совершенно не ясно, по какому принципу они формируются. Ее решение заключается в использовании Web Scraping. Суть метода в том, чтобы не просто качать страницы по ссылкам, а действительно открывать браузер, указывать программе нужные html-формы, кликать по нужным кнопкам, по пути собирая данные для анализа. Для того, чтобы это реализовать, в данной программе используется библиотека selenium-инструмент для автоматизации действий веб-браузера.

Второе - построение графа. Для данной задачи в программе при помощи NetworkX строится граф на основе уже полученных данных, затем он импортируется в формат gexf. Графы, хранящиеся в данном формате, можно легко визуализировать с помощью программы Gephi. При наведении на вершину (в данном случае вершина равно статья) показываются все статьи, которые на нее ссылаются. Размер вершины зависит от количетсва входящих в нее ребер: чем их больше, тем больше и размер.

1.1 Векторная модель

Наконец, основная задача: кластеризация и анализ статей. Для того, чтобы каждую статью отнести к тому или иному кластеру, нужно научиться сравнивать их аннотации. Делать это можно, построив векторную модель: представить коллекцию аннотаций векторами из одного общего для всей коллекции векторного пространства.
Документ в векторной модели представляется как множество термов. Термами называют слова, которые употреблены в тексте, а также такие элементы текста, как, например, '2018' или 'Аполлон-13'. Важно научиться определять "вес"терма: то, насколько он "важен"для идентификации данной аннотации. Теперь, если представить каждую аннотацию как вектор d_i, а "веса"всех термов коллекции упорядочить и обозначить как $w_{ij}, j \in [n]$, где n - количество термов в коллекции, то каждый из текстов можно записать в виде выражения

$$d_i = (w_{i1}, ..., w_{in})$$

Теперь, когда каждая аннотация представлена в виде вектора, можно решать задачу подобия: если обозначить каждый текст за точку, то чем ближе точки, тем более похожи текста.
Каким же образом задавать веса для термов? Для этого существует несколько стандартных методов:
1) Булевский вес — равен 1, если терм встречается в документе и 0 в противном случае;
2) TF (term frequency) — вес зависит только от частоты употребления терма в документе;
3) TF-IDF (term frequency — inverse document frequency) [?] — вес зависит не только от частоты употребления слова в данном документе, но и от частоты его употребления в остальных документах коллекции
Ясно, что первые 2 метода более грубые по сравнению с третьим, поэтому для решения задачи был выбран именно он.
TF зависит только от частоты употребления слова в данном документе и рассчитывается по формуле

$$TF(t, d) = \frac{n_t}{\sum_k n_k}$$

где n_t - количество употреблений терма t в документе d, в знаменателе общее количество термов в d
IDF - инверсия частоты, с которой слово встречается в документах коллекции. Чем более уникально слово, тем меньше у него IDF. И наоборот, чем слово более употребимо, тем больше у него IDF. Вычисляется данная величина по формуле

$$IDF(t, D) = \log \frac{|D|}{|\{d_i \in D | t \in d_i\}|}$$

где $|D|$ - количество документов в коллекции D, d_i - i-ый документ коллекции, t - терм
Наконец, "вес"терма с помощью метода TF-IDF вычисляется как

$$TF\text{-}IDF(t, d, D) = TF(t, d) \cdot IDF(t, D)$$

Теперь ясно, как представлять каждую из статей в виде вектора. Однако теперь нужно вернуться немного назад и понять, как для каждой статьи задать набор корректных термов. В данной программе для этого используется модуль nltk (natural language toolkit). Считывая из базы данных аннотацию, из нее сразу же отбрасываются так называемые стоп-слова: предлоги, частицы и т.д. Далее производится стемминг — процесс нахождения основы слова для заданного исходного слова. По полученным таким образом термам строится матрица $TF\text{-}IDF$

1.2 Кластеризация

Кластеризация - задача упорядочивания множества объектов в сравнительно однородные группы Существует множество методов кластеризации текстовых документов [**?**]. Для решения поставленной задачи были выбраны два из них: метод k-средних и латентное размещение Дирихле.

Метод k-средних

Данный метод разбивает множество документов на заранее известное число кластеров k. На каждой итерации алгоритма перерасчитываются центры кластеров так, чтобы минимизовать квадраты расстояний между центрами кластеров и элементами, которые соответствуют этим центрам. [**?**] Алгоритм завершается, как только на какой-то итерации перестает изменяться внутрикластерное расстояние. Это происходит за конечное число шагов, т.к. количество разбиений конечного множества конечно, а внутрикластерное расстояние на каждом шаге уменьшается (рис.1).
Более формально, нужно найти

$$\underset{S}{\operatorname{argmin}} = \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2$$

Где S - множество кластеров, S_i - i-ый кластер, μ_i - центр i-ого кластера.

Data: $\{\vec{x_1}, ..., \vec{x_N}\}, K$ ($\vec{x_i}$ - i-ый документ, K - количество кластеров, N - количество документов)
Result: $(\vec{\mu_1}, ..., \vec{\mu_K})$(оптимальные кластеры)
$(\vec{s_1}, ..., \vec{s_K}) \leftarrow SelectRandomSeeds(\{\vec{x_1}, ..., \vec{x_N}\}, K)$;
for $k \leftarrow 1$ **to** K **do**
  $\vec{\mu_k} \leftarrow \vec{s_k}$;
end
while $CentroidsChange()$ **do**
  for $k \leftarrow 1$ **to** K **do**
    $\omega_k \leftarrow \{\}$ (инициализируем кластеры);
  end
  for $n \leftarrow 1$ **to** N **do**
    $j \leftarrow \underset{j'}{\operatorname{argmin}} \|\vec{\mu_{j'}} - \vec{x_n}\|$ (поиск номера кластера, расстояние до которого от n-ого элемента наименьшее);
    $\omega_j \leftarrow \omega_j \cup \{\vec{x_n}\}$ (присоединение элемента к оптимальному кластеру);
  end
  for $k \leftarrow 1$ **to** K **do**
    $\vec{\mu_k} \leftarrow \frac{1}{|\omega_k|} \sum_{\vec{x} \in \omega_k} \vec{x}$ (перераспределение кластеров) ;
  end
end

Algorithm 1: Method K-Means

Для того, что определить оптимальное число кластеров k, в программе используется **"метод локтя"(elbow method)**. Суть метода заключается в том, чтобы построить график зависимости количества кластеров от суммы квадратов расстояний от центров кластеров до элементов, которые им соответствуют (рис. 2). Наиболее оптимальное количество кластеров - точка, в которой сумма квадратов начинает падать более плавно.

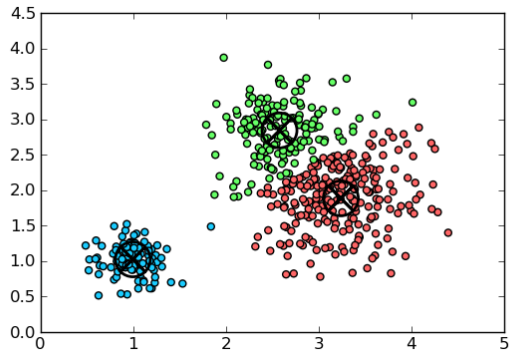

Рис. 1: K-Means

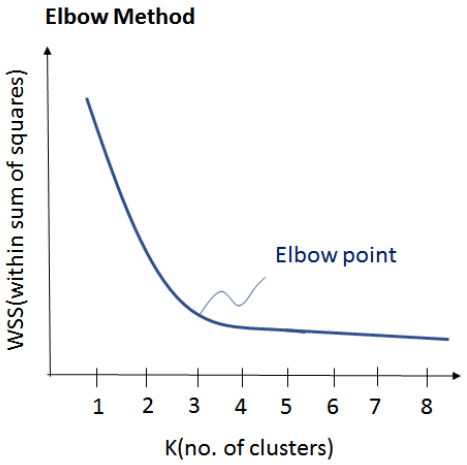

Рис. 2: Elbow method

Латентное размещение Дирихле

Latent Dirichlet allocation (LDA) - порождающая статистическая модель, позволяющая узнавать сходства между объектами и распределять их по группам. [?] [?]
Для того, чтобы не углубляться в математику, здесь приведено только краткое и интуитивное описание алгоритма:

1. для каждой темы t
   (a) выбрать вектор φ_t (распределение слов в теме) по распределению $\phi_t \sim Dir(\beta)$;
2. для каждого документа **w**:
   (a) выбрать $N \sim Poisson(\xi)$
   (b) выбрать вектор $\theta_d \sim Dir(\alpha)$ - вектор "степени выраженности"каждой темы в этом документе;
   (c) для каждого из N слов w_n:
      i. выбрать тему z_n по распределению $Multinomial(\theta)$;
      ii. выбрать слово w_n из $p(w_n|z_n, \beta)$ с вероятностями, заданными в β;

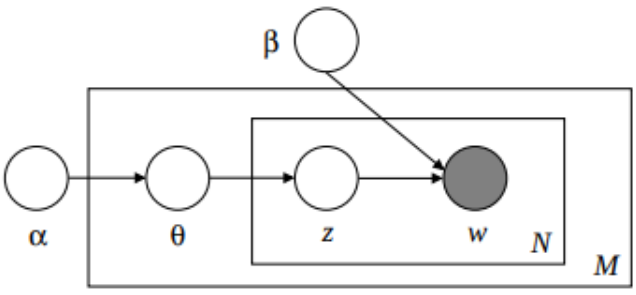

Рис. 3: Графическое представление LDA. Прямоугольник M представляет собой документы, в то время как прямоугольник N - повторяющийся выбор тем и слов в документе

1.3 Анализ результатов

После построения кластеров нужно попробовать понять, насколько хорошо они отображают реальное положение дел. Для этого в рамках данной задачи применены два метода, дающие графическую интерпретацию построенным данным.

MDS

Высокоразмерные данные, т.е. данные, имеющие размерность более трех, может быть трудно интерпретировать. Один из подходов к упрощению состоит в том, чтобы предположить, что данные, представляющие интерес, лежат во вложенном многообразии в пространстве более высокого порядка. Если это многообразие имеет малую размерность, его можно визуализировать в низкоразмерном (размерности 2 или 3) пространсте(рис. 4).
Одним из алгоритмов, реализующих данную идею, является **Multidimensional scaling (MDS)**. Если δ_{ij} - расстояние между точками i и j в пространстве $\mathbb{R}^n$, то MDS пытается найти такую функцию $f : \mathbb{R}^n \to \mathbb{R}^d$ (d - размерность пространства, представление в котором ищется), чтобы расстояния $\delta_{ij} = f(d_{ij})$ между точками i и j в пространстве $\mathbb{R}^d$ максимально коррелировали с d_{ij}. [**?**]
Итоговый результат MDS - матрица с координатами n точек в пространстве $\mathbb{R}^d$

$$\begin{pmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ x_{21} & x_{22} & \dots & x_{2d} \\ \dots & \dots & \dots & \dots \\ x_{n1} & x_{n2} & \dots & x_{nd} \end{pmatrix}$$

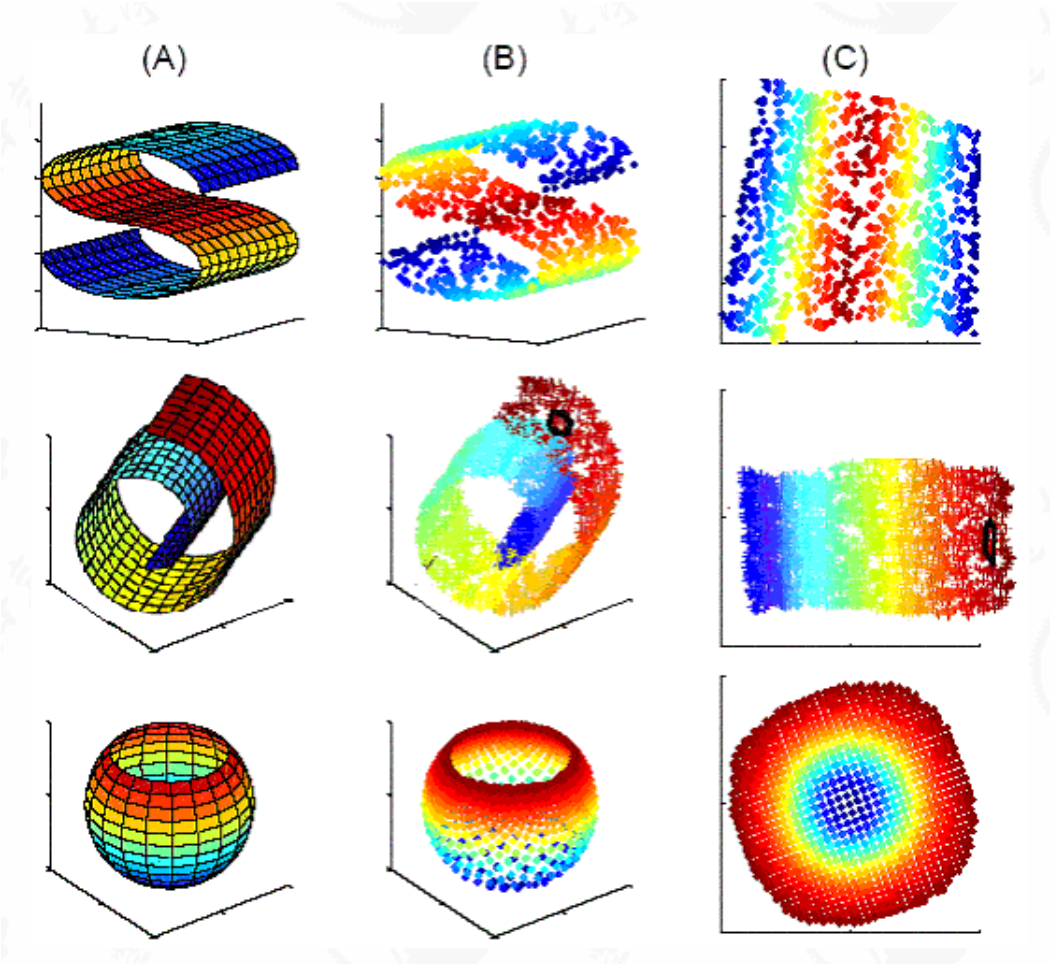

Рис. 4: Графическое представление MDS. Преобразование трехмерного пространства в двумерное

Иерархическая кластеризация (HDC)

Второй метод для анализа данных, использованный в данной программе, заключается в построении дендрограммы(рис. 5). Для ее построения используется алгоритм Уорда (Ward's method), минимизирующий на каждой итерации дисперсию внутри кластеров.
Для реализации алгоритма на каждом шаге находят такую пару кластеров, чтобы увеличение дисперсии после их слияния оказалось минимальным из возможных. [**?**] На начальном этапе все кластеры являются одиночными, т.е. каждому кластеру соответствует ровно один документ. Для применения данного метода начальное расстояние между кластерами должно быть пропорционально квадрату евклидова расстояния.
Поэтому в начальный момент расстояния между документами определяются как квадрат евклидова расстояния:

$$d_{ij} = d(\{X_i\}, \{X_j\}) = \|X_i - X_j\|^2$$

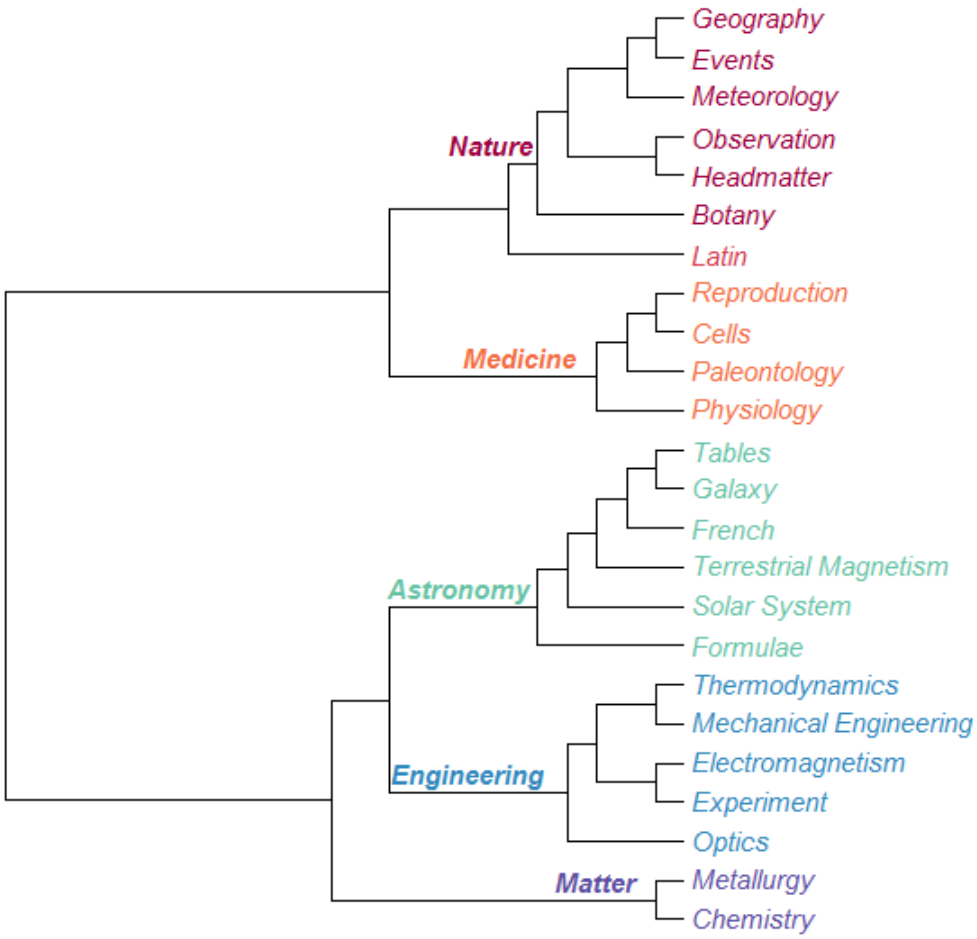

Рис. 5: HDC

2 Описание программы

В данном разделе шаг за шагом описано, как реализована программа. В отчете код часто обрезан с целью большей наглядности. Для того, чтобы увидеть программу целиком, нужно проследовать в репозиторий, ссылка на который указана выше.

Чтобы понять общую структуру программы, полезно посмотреть на main.py.

main.py

```
1  import sys
2  import wos_parser
3  import wos_graph
4  import wos_clasterization
5
6  if __name__ == "__main__":
7      if "help" in sys.argv:
8          print("Parameters:")
9          print("showmds - show multidimensional scaling")
10         print("nodownload - don't download new article, use old")
11         print("showts - show titles for every claster")
12         print("showhdc - show hierarchical document clustering")
13         print("showlda - show latent Dirichlet allocation")
14         sys.exit()
15     print("Add 'help' for showing parameters")
16     print("Input search string")
17     topic_name = input()
18
19     if "nodownload" in sys.argv:
20         print("Input count of articles")
21         cnt_articles = int(input())
22     else:
23         cnt_articles = wos_parser.site_parser(topic_name)
24     articles = []
25     for i in range(1, cnt_articles+1):
26         wos_parser.article_parser(topic_name + str(i), articles)
27     articles = wos_parser.correct_articles(articles)
28
29     articles = wos_graph.build_graph(articles, topic_name)
30
31     showmds = 0
32     showts = 0
33     showhdc = 0
34     showlda = 0
35     if "showmds" in sys.argv:
36         showmds = 1
37     if "showts" in sys.argv:
38         showts = 1
39     if "showhdc" in sys.argv:
40         showhdc = 1
41     if "showlda" in sys.argv:
42         showlda = 1
43
44     wos_clasterization.build_csv(articles, topic_name)
45     wos_clasterization.article_clasterization(topic_name, showmds, showts,
46                                               showhdc, showlda)
```

Помимо main.py, программа состоит еще из трех файлов: wos_parser.py, wos_graph.py и wos_clasterization.py. Эти файлы отвечают соответственно за парсинг WoS, построение графа цитирования и кластеризацию статей. Для удобства пользователя предусмотрен выбор параметров. Без параметров программа скачивает статьи, строит граф и проводит классификацию методом k-средних. Из названий параметров понятно, за что отвечает каждый из них. Возможно, следует пояснить про "nodownload передав данный параметр, пользователь сможет не скачивать статьи с WoS, а воспользоваться теми, которые уже скачаны. При этом пользователь обязан указать их количество.

2.1 Web Scraping

В main.py из файла wos_parser.py используются 3 функции: site_parser, article_parser и correct_articles. Разберем содержание каждой из них.

wos_parser.site_parser() (1)

```
1  def site_parser(topic_name):
2      driver = webdriver.Firefox()
3      driver.get("http://www.webofknowledge.com")
4      element = driver.find_element_by_xpath("//input[@id=" \
5                                             "'value(input1)']")
6      element.send_keys(topic_name)
7      element.submit()
8      cur_articles = driver.find_elements_by_xpath("//a[@class='smallV110']"
           )
9      cur_articles[0].click()
10     page_content = driver.page_source
```

Данная часть кода открывает браузер Firefox, переходит на сайт WoS, вводит в поисковую строку запрос, который пользователь раннее сообщил программе. Иллюстрация того, как выглядит html-форма, которую нужно искать для построения запроса к методу driver.find_element_by_xpath, показана на рис. 6.
После отправки запроса в поисковик программа переходит на первую статью и скачивает содержимое страницы в папку data. Все страницы, скачанные пользователем, не удаляются после завершения программы. Это нужно для того, чтобы, в случае необходимости, была возможность их повторного использования.

Рис. 6: Поиск нужной html-формы

Нахождение количества статей, это число записывается в cnt_pages.

wos_parser.site_parser() (2)

```
1     search_string = 'This table has <b>'
2     ind_in_search = page_content.find(search_string)
3     ind_in_search += 19
4     ind_end = ind_in_search
5     while (page_content[ind_end] != ' '):
6         ind_end += 1
7     cnt_string = ""
8     for l in page_content[ind_in_search:ind_end]:
9         if l != ",":
10            cnt_string += l
11    cnt_pages = int(cnt_string)
```

Для каждой статьи сохраняется html-страница. wos_parser.site_parser() возвращает количество статей, выданных WoS по запросу пользователя.

wos_parser.site_parser() (3)

```
1     ind_page = 0
2     while ind_page < cnt_pages:
3         print("Creating ../data/"+topic_name+str(ind_page+1))
4         write_file = open("../data/"+topic_name+str(ind_page+1), "w")
5         write_file.write(driver.page_source)
6         write_file.close()
7
8         if ind_page != cnt_pages-1:
9             element2 = driver.find_element_by_xpath("//a[@class='
                  paginationNext']")
10            element2.click()
11        ind_page += 1
12    driver.close()
13    return cnt_pages
```

Для дальнейшего парсинга создан абстрактный класс Article. Он служит для структурирования данных. По A.name, A.author, A.abstract сохраняются, соответственно, название статьи, ее автор и аннотация. По A.CitedReference соханяется список статей, на которые ссылается данная.

```
1 class Article:
2     pass
```

Функция wos_parser.article_parser() собирает из html-страницы статьи всю нужную информацию. Одним из параметром является articles: это список статей, который заполняется по ходу выполнения функции. Для примера приведено получение из html-страницы статьи ее аннотации.
Когда в WoS мы проводим поиск по строке, поисковик выделяет слова, которые одинаковы со словами из введенной строки. Эти выделения визуально выглядят как желтые подчеркивания, на практике же важно, что, в наших данных может присутствовать нежелательный html-код. Функция del_highlightings(line) удаляет из line все все такие выделения и возвращает строку без них.

wos_parser.article_parser()

```
def article_parser(file_name, articles):
    A = Article()
    # <...>
    rfile = open("../data/"+file_name, "r")
    prev_abstract = 0
    search_string = '''<div class="title3">Abstract</div>'''
    for line in rfile:
        if prev_abstract == 1:
            line = line[20:-5]
            line = del_highlightings(line)
            A.abstract = line
            break
        else:
            ind_in_search = line.find(search_string)
            if ind_in_search != -1:
                prev_abstract = 1
    rfile.close()
    # <...>
    articles.append(A)
```

Наконец, функция wos_parser.correct_articles принимает массив articles, в котором могут быть пропуски на местах названий статей или имен авторов. Такие статьи не нужны ни для построения графа, ни для проведения классификации, поэтому они удаляются из articles. Возвращает данная функция статьи, откорректированные таким образом.

wos_parser.correct_articles()

```
def correct_articles(articles):
    new_articles = []
    for A in articles:
        if not hasattr(A, "name") or not hasattr(A, "author"):
            continue
        NA = Article()
        NA.name = A.name
        NA.author = correct_authors(A.author)
        if hasattr(A, "abstract"):
            NA.abstract = A.abstract
        NA.CitedReferences = []
        for i in range(len(A.CitedReferences)):
            # <...>
        new_articles.append(NA)
    return new_articles
```

Функция correct_authors() приводит авторов к такому виду, чтобы статью можно было однозначно идентифицировать. Так, на сайте WoS у одной и той же статьи может быть как 10 авторов, так и всего 2, а остальные скрыты за подписью "et al.". Также имена авторов могут быть написаны как в полной форме, так и в сокращении. Для того, чтобы считать такие статьи одинаковыми, эта функция выполняет разбор случаев. А именно, все авторы приводятся к такому виду: фамилия начинается с заглавной букву, остальные строчные; если у автора присутствует имя, то берется только его первая буква, остальные отбрасываются; если есть второе имя, то с ним происходит то же самое. Все остальное, если оно есть, отбрасывается.
Благодаря данной процедуре, не учитываются одни и те же статьи дважды. Как именно отбрасывать ненужные статьи, описано немного ниже.

2.2 Граф цитирования

В main.py для построения графа используется метод wos_graph.build_graph(). Он строит граф и сохраняет его в формате gexf. С помощью программы Gephi можно легко визуализировать построенный граф. Метод возвращает массив статей без повторений.
В данной функции используются еще 2 абстрактных класса, описанные в файле abstract_data.py

```
1 class CitedReference:
2     pass
3
4
5 class AVertex:
6     pass
```

Класс CitedReference - контейнер для хранения ссылки на статью. Он имеет поля name и author.
Класс AVertex используется для представления статьи как вершины графа, он имеет поля name и author и indeg (количество входящих ребер).
Несмотря на то, что по своей сути эти классы одинаковы, они не объединены в один. Мотивацей к этому послужило желание увеличить абстракцию данных и понятность кода.
Рассмотрим код начала функции build_graph().

```
1  def build_graph(articles, topic_name):
2      articles = del_equal_articles(articles)
3      vertexes = []
4      edges = []
5      v_number = {}
6      for i in range(len(articles)):
7          AV = AVertex()
8          AV.name = articles[i].name
9          AV.author = articles[i].author
10         v_number[AV.name] = i
11         AV.indeg = 0
12         vertexes.append(AV)
```

Функция del_equal_articles(articles) принимает массив со статьями articles и возвращает массив без повторений (в смысле, описанном выше). Две статьи считаются одинаковыми, если их названия совпадают, а множество авторов одной статьи является подмножеством множества авторов другой статьи. Авторы считаются одинаковыми, если их фамилии совпадают, а сокращения имен либо полностью совпадают, либо одно из сокращений включает другое. Например , авторы "Abba G H"и "Abba G"считаются одинаковыми, а "Abba G H"и "Abba G M"нет.
v_number - хеш-таблица, отвечающая за то, чтобы при добавлении новой статьи определять, добавлена ли она граф. Например, если **A**, **B**, **C** - статьи, > ссылка одной статьи на другую, то при добавлении "**A** -> **B**"мы добавим в граф как вершину (статью) **A**, так и **B**. Тогда при добавлении "**C** -> **B**"вершина **B** не должна дублироваться.
Цикл for добавляет в массив с вершинами vertexes статьи, которые были скачаны по запросу пользователя.
После добавляются статьи, на которые они ссылаются

```
    ind_v = len(articles)
    for art_i in range(len(articles)):
        for ref_i in range(len(articles[art_i].CitedReferences)):
            if v_number.get(articles[art_i].CitedReferences[ref_i].name):
                if is_equal(vertexes[v_number[articles[art_i].
                                                CitedReferences[ref_i]
                                                .name]].author,
                            articles[art_i].CitedReferences[ref_i].author)
                                :
                    vertexes[v_number[articles[art_i].
                                      CitedReferences[ref_i].
                                      name]].indeg += 1
                    edges.append((v_number[articles[art_i].name],
                                  v_number[articles[art_i].
                                           CitedReferences[ref_i].
                                           name]))
                    continue
            AV = AVertex()
            AV.name = articles[art_i].CitedReferences[ref_i].name
            AV.author = articles[art_i].CitedReferences[ref_i].author
            AV.indeg = 1
            vertexes.append(AV)
            v_number[AV.name] = ind_v
            edges.append((v_number[articles[art_i].name], ind_v))
            ind_v += 1
```

Данный код обрабатывает каждую из статей, проходя по всем статьям, на которые она ссылается. Если такая статья уже есть в графе, то ее степень увеличивается и добавляется соответствующее ребро в массив с ребрами edges. Если же ее нет в графе, вершина добавляется в массив vertexes и в хеш-таблицу v_number, также добавляется ребро и входящая степень инициализируется единицей. Таким образом, после выполнения данного кода строятся массивы с вершинами vertexes и с ребрами edges. Для построения графа используется библиотека NetworkX.

```
    G = nx.DiGraph()
    for i in range(len(vertexes)):
        G.add_node(i)
        vertex_name = vertexes[i].name+" | "
        for a in vertexes[i].author:
            vertex_name += a+"; "
        G.nodes[i]['label'] = vertex_name
        G.nodes[i]['viz'] = {'size': 4 + 3*vertexes[i].indeg}
        if i < len(articles):
            G.nodes[i]['viz']['color'] = {'a' : 0,
                                          'r' : 255,
                                          'g' : 255,
                                          'b' : 255}
        else:
            G.nodes[i]['viz']['color'] = {'a' : 0,
                                          'r' : 67,
                                          'g' : 255,
                                          'b' : 20}
    G.add_edges_from(edges)

    nx.write_gexf(G, "../data/"+topic_name+".gexf")
    print("Graph saved in ", "../data/"+topic_name+".gexf")
    return articles
```

Имя вершины в графе записывается в виде <title> | <authors>. Размер вершины коррелирует с числом входящих ребер indeg.

После создания графа он сохраняется в формате gexf для дальнейшего использования.

2.3 Кластеризация

Наконец, рассмотрим часть программы, ответственную за кластеризацию.
Метод wos_clasterization.build_csv() используется для построения csv файла. С файлом в таком формате работать намного удобнее, чем с массивом, в частности, появляется возможность использовать библиотеку pandas.

wos_clasterization.build_csv()

```
1  def build_csv(articles, topic_name):
2      new_articles = del_empty_abstracts(articles)
3      for A in new_articles:
4          A.abstract = A.abstract +" "+A.name
5      csvf = open("../data/"+topic_name+".csv", "w")
6      csvf.write("name\0author\0abstract\n")
7      for A in new_articles:
8          csvf.write(A.name+"\0")
9          for i in range(len(A.author)):
10             if i != len(A.author)-1:
11                 csvf.write(A.author[i]+", ")
12             else:
13                 csvf.write(A.author[i]+"\0")
14         csvf.write(A.abstract+"\n")
15     csvf.close()
16     print("Database from articles saved in ../data/"+topic_name+".csv")
```

Функция del_empty_abstracts() удаляет из массива статьи с пустым описанием. Для того, чтобы улучшить результат кластеризации, в конец каждой аннотации добавляется название статьи.

wos_clasterization.article_clasterization() (1)

```
1  def article_clasterization(topic_name, showtfidf, showmds,
2                             showts, showhdc, showlda):
3      df = pd.read_csv("../data/"+topic_name+".csv", sep="\0")
4      with redirect_stdout(open(os.devnull, "w")):
5          nltk.download('punkt')
6          nltk.download('stopwords')
7      stopwords = nltk.corpus.stopwords.words('english')
8      stemmer = SnowballStemmer("english")
9
10     def local_stop_words(word):
11         not_stopword = 1
12         if stemmer.stem(word) == "use":
13             not_stopword = 0
14         return not_stopword
```

Создается датафрейм на основе подготовленного раннее csv-файла. Используя модуль nltk, подгружаются пунктуация и стоп-слова. Также инициализируется стеммер. Функция local_stop_words() проверяет, является ли слово искусственным стоп-словом. Иногда в кластеры попадают слова, которые явно не имеют оригинальности. Для того, чтобы избавиться от таких слов, и существует данная функция.

wos_clasterization.article_clasterization() (2)

```
1     def tokenize_and_stem(text):
2         tokens = [word for sent in nltk.sent_tokenize(text) for word in
3                   nltk.word_tokenize(sent)]
4         tokens = [word for word in tokens if local_stop_words(word)]
5         filtered_tokens = []
6         for token in tokens:
7             if (re.search('[a-zA-Z]', token) and
8                     not re.search("'", token)):
9                 filtered_tokens.append(token)
10        stems = [stemmer.stem(t) for t in filtered_tokens]
11        return stems
12
13    def tokenize(text):
14        tokens = [word.lower() for sent in nltk.sent_tokenize(text) for
              word in
15                  nltk.word_tokenize(sent)]
16        tokens = [word for word in tokens if local_stop_words(word)]
17        filtered_tokens = []
18        for token in tokens:
19            if (re.search('[a-zA-Z]', token) and
20                    not re.search("'", token)):
21                filtered_tokens.append(token)
22        return filtered_tokens
```

Функция tokenize_and_stem() разделяется слова на термы, удаляет из них как стоп-слова из модуля nltk, так и локальные стоп-слова. Также из множества термов удаляются те, в которых есть апостроф или в которых нет латинских букв. В конце функции термы подвергаются стеммингу. Функция tokenize() делает все то же самое, кроме стемминга.

wos_clasterization.article_clasterization() (3)

```
1     totalvocab_stemmed = []
2     totalvocab_tokenized = []
3
4     df = df.dropna(axis=0, how='any')
5     for abstract in df['abstract'].tolist():
6         allwords_stemmed = tokenize_and_stem(abstract)
7         totalvocab_stemmed.extend(allwords_stemmed)
8
9         allwords_tokenized = tokenize(abstract)
10        totalvocab_tokenized.extend(allwords_tokenized)
11
12    vocab_frame = pd.DataFrame({'words': totalvocab_tokenized},
13                               index = totalvocab_stemmed)
```

totalvocab_stemmed - массив с термами из аннотаций, которые были переданы в tokenize_and_stem().
totalvocab_tokenized - массив с термами из аннотаций, которые были переданы в tokenize().
vocab_frame - датафрейм с одной колонкой: термами из массива totalvocab_tokenized. Индексами в этом датафрейме служат слова, соответствующие словам в колонках, но подвергнутым стеммингу.

wos_clasterization.article_clasterization() (4)

```
    tfidf_vectorizer = TfidfVectorizer(max_df=0.8,
                                       max_features=200000,
                                       min_df=0.001,
                                       stop_words='english',
                                       use_idf=True,
                                       tokenizer=tokenize_and_stem,
                                       ngram_range=(1,8))

    tfidf_matrix = tfidf_vectorizer.fit_transform(df['abstract'].tolist())
    terms = tfidf_vectorizer.get_feature_names()

    dist = 1 - cosine_similarity(tfidf_matrix)
```

TfidfVectorizer - метод из библиотеки sklearn. Он реализует построение модели *TF-IDF*. Параметры:
max_df - максимальное значение *TF-IDF*
min_df - минимальное значение *TF-IDF*
stop_words - стоп-слова
use_idf - использовать ли значение *IDF*
tokenizer - функция для разделения документов на термы
ngram_range - создание N-грамм: сочетаний из термов
С помощью метода fit_transform создается *TF-IDF* матрица на основе аннотаций.
cosine_similarity - расстояние между каждым из термов
dist - то, насколько термы схожи друг с другом

wos_clasterization.article_clasterization() (5)

```
    distortions = {}
    for num_clusters in range(1, min(11, len(df['abstract'].tolist())+1)):
        print("K Means with "+str(num_clusters)+" clusters")
        km = KMeans(n_clusters = num_clusters)
        km.fit(tfidf_matrix)
        distortions[num_clusters] = km.inertia_

    plt.figure()
    plt.plot(list(distortions.keys()), list(distortions.values()))
    plt.xlabel("Number of cluster")
    plt.ylabel("Distortions")
    plt.show()
```

С помощью метода KMeans из sklearn производится кластеризацию статей. k выбирается от 1 до 10 и для каждого k в distortions записываются суммы квадратов расстояний от элементов кластеров до их цетров. Затем применяется метод "локтя"и строим график. Так, для запроса Dijkstra's algorithm график схлж с рис. 7 (иногда KMeans может попадать в локальные минимумы, поэтому графики могут быть различны).

wos_clasterization.article_clasterization() (6)

```
    print("Input count of clusters:")
    num_clusters = int(input())
    km = KMeans(n_clusters = num_clusters)
    km.fit(tfidf_matrix)

    print("Kmeans saved in ../data/"+topic_name+".pkl")
    joblib.dump(km, "../data/"+topic_name+".pkl")
    km = joblib.load("../data/"+topic_name+".pkl")
```

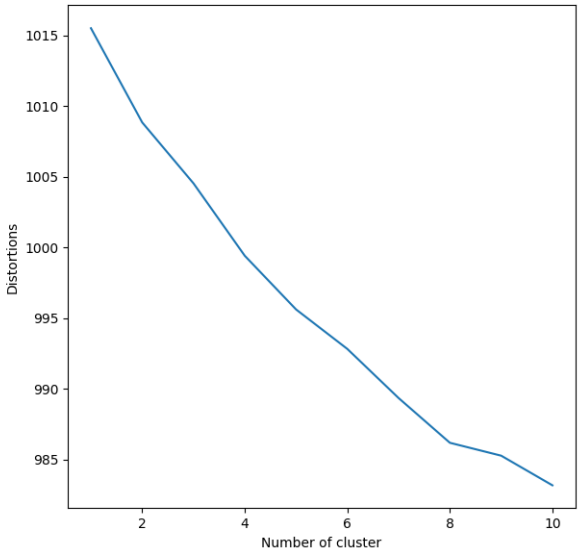

Рис. 7: Elbow method для 'Dijkstra's algorithm'

Пользователь выбирает количество кластеров, которое ему кажется оптимальным. Затем обученная модель сохраняется (для возможного переиспользования).

wos_clasterization.article_clasterization() (7)

```
clusters = km.labels_.tolist()

articles_describe = {'title' : df['name'].tolist(),
          'author' : df['author'].tolist(),
          'abstract' : df['abstract'].tolist(),
          'cluster' : clusters}

cluster_df = pd.DataFrame(articles_describe,
                      index = [clusters],
                      columns = ['title',
                                 'author',
                                 'abstract',
                                 'cluster',])

print("\nTop terms per cluster:\n")

for i in range(num_clusters):
    print("Cluster %d words:" % (i+1), end='')
    # <...>
    if showts:
        print("Cluster %d titles:" % i)
        ind_t = 1
        for title in cluster_df.ix[i]['title'].values.tolist():
            print("Title", str(ind_t)+":", title)
            ind_t += 1
        print("\n")
```

Выводятся термы, которые соответствуют каждому из кластеров. Если пользователь также передал параметр "showts для каждого кластера выводится список статей, которые в него входят.

Пример кластеризации будет в следующей главе.
Реализация алгоритма LDA с помощью библиотеки gensim.
Разделение аннотаций на термы, удаление стоп-слов.

wos_clasterization.article_clasterization() (8)

```
1     if showlda:
2         tokenized_text = [tokenize_and_stem(text) for text
3                           in df['abstract'].tolist()]
4         texts = ([[word for word in text if word not in stopwords]
5                   for text in tokenized_text])
```

В словарь gensim добавляются термы, они фультруются по количеству употреблений: нижнего ограничения нет, верхнее - употреблено не более чем в половине абстрактов.
doc2bow - возвращает кортеж: (id слова, количество употреблений каждого слова в абстрактах)

wos_clasterization.article_clasterization() (9)

```
1         dictionary = corpora.Dictionary(texts)
2         dictionary.filter_extremes(no_below=1, no_above=0.5)
3         corpus = [dictionary.doc2bow(text) for text in texts]
```

Модель LDA на основе аннотаций. Параметры:
corpus - словарь с id слов и количеством их использований
id2word - словарь, ставящий с соответствие каждому id само слово
num_topics - количество кластеров
update_every - количество итераций между обновлениями кластеров
chunksize - размер буффера для каждого обновления
passes - количество прохождений алгоритма по corpus
После создания модели получается матрица с топ-словами для каждого из кластеров, производится их вывод на экран

wos_clasterization.article_clasterization() (10)

```
1         lda = models.LdaModel(corpus, num_topics=num_clusters,
2                               id2word=dictionary,
3                               update_every=5,
4                               chunksize=10000,
5                               passes=100)
6         topics_matrix = lda.show_topics(formatted=False, num_words=10)
7         topics_matrix = np.array(topics_matrix, dtype=object)
8         # <...>
```

Реализация MDS с помощью библиотеки sklearn. Параметры:
n_components - размерность пространства, получаемого на выходе алгоритма
dissimilarity = "precomputed" - массив расстояний уже предподсчитан и подается в функцию mds.fit_transform()
random_state - генератор для инициализации кластеров
xs, ys - координаты вершин по осям X и Y

wos_clasterization.article_clasterization() (11)

```
1     if showmds:
2         mds = MDS(n_components=2, dissimilarity="precomputed",
              random_state=1)
3         pos = mds.fit_transform(dist)
4         xs, ys = pos[:, 0], pos[:, 1]
```

Построение коллекций цветов и имен кластеров.

wos_clasterization.article_clasterization() (12)

```
cluster_colors = {0: '#000000', 1: '#ff0000', 2: '#9a37b4',
                  3: '#308b1b', 4: '#2dc7e7', 5: '#00097e',
                  6: '#9a9a9a', 7: '#b7d41b', 8: '#90aeff',
                  9: '#92002c'}
cluster_names = {}
for i in range(len(res_labels)):
    cluster_names[i] = res_labels[i][:min(3, len(res_labels[i]))]
```

Построение графика на основе этих данных. Вершины, лежащие в одном кластере, окрашены в один цвет.

wos_clasterization.article_clasterization() (13)

```
df_mds = pd.DataFrame(dict(x=xs, y=ys,
                           label=clusters))
groups_mds = df_mds.groupby("label")
for name, group in groups_mds:
    ax.plot(group.x, group.y, marker='o', linestyle='',
            label=cluster_names[name], ms=12,
            color=cluster_colors[name], mec='none')
```

Вывод легенды, сохранение построенного графика.

wos_clasterization.article_clasterization() (14)

```
ax.legend(numpoints=1)
plt.draw()
plt.savefig("../data/mds_"+topic_name+".png", dpi=200)
print("Mds plot saved in ../data/mds_"+topic_name+".png")
plt.close()
```

Построение модели HDC по матрице расстояний dist с помощью метода ward() из библиотеки scypy.
По полученной матрице строится дендрограмма.

wos_clasterization.article_clasterization() (15)

```
linkage_matrix = ward(dist)
ax = dendrogram(linkage_matrix, orientation="right",
            labels=df['name'].tolist())
```

Сохраняем построенный график.

wos_clasterization.article_clasterization() (16)

```
plt.draw()
plt.savefig("../data/hdc_"+topic_name+".png", dpi=200)
print("Hdc plot saved in ../data/hdc_"+topic_name+".png")
plt.close()
```

3 Анализ результатов

В качестве примера работы программы рассмотрим ее вывод при вводе запроса 'Dijkstra's algorithm' и выбора трех кластеров.
Вывод результов работы алгоритмов **K-Means** и **LDA** показан на рисунках 8 и 9.
Первый из кластеров на рисунке 9 выводит наиболее общие данные. Можно предположить, что статьи, соответствующие этому кластеру, описывают сам алгоритм Дейкстры, как-то его анализируют или улучшают.
Второй из кластеров, скорее всего, описывает применение алгоритма Дейкстры в технологиях, так или иначе связанных с Wi-Fi сетями и датчиками.
Третий кластер, по-видимому, связан с траффиком и протоколами.
Как видно, разные алгоритмы по-разному кластеризируют статьи. Для того, чтобы в дальнейшем улучшать результаты кластеризации, на мой взгляд, лучше подходит **LDA** из-за его гибкости.

```
Top terms per cluster:

Cluster 1 words:path,graph,shortest,problem,method,images,computations,se
arch,optimizer,proposes,dijkstra

Cluster 2 words:sensors,wireless,energy,networks,route,nodes,clustering,r
elays,optimizer,consumption,path

Cluster 3 words:route,networks,path,optimizer,time,proposes,traffic,model
ing,vehicle,protocol,simulation
```

Рис. 8: K-Means для 'Dijkstra's algoritm'

```
LDA:
Cluster 1: route, networks, nodes, optimizer, system, based, sensors, sim
ulation, energy, wireless,
Cluster 2: problem, networks, shortest, route, time, optimizer, graph, me
thod, computations, based,
Cluster 3: images, method, segmentation, graph, shortest, structure, base
d, approach, data, computations,
```

Рис. 9: LDA для 'Dijkstra's algorithm'

На рисунках 10 и 11 показан граф, построенный на основе статей, выданных по данному запросу. Граф построен с помощью программы Gephi.
При наведении на вершину показывается название статьи ее авторы. Размер вершины зависит от количества входящих ребер, ее цвет от того выдана ли статья, соответствующая вершине, в поиске WoS по запросу пользователя, или же эта статья нашлась только в тех, на которые ссылались. Белые вершины - статьи первого типа, зеленые - второго.
Как видно, есть статья, на которую очень много ссылаются. Автором данной статьи является сам Дейкстра.

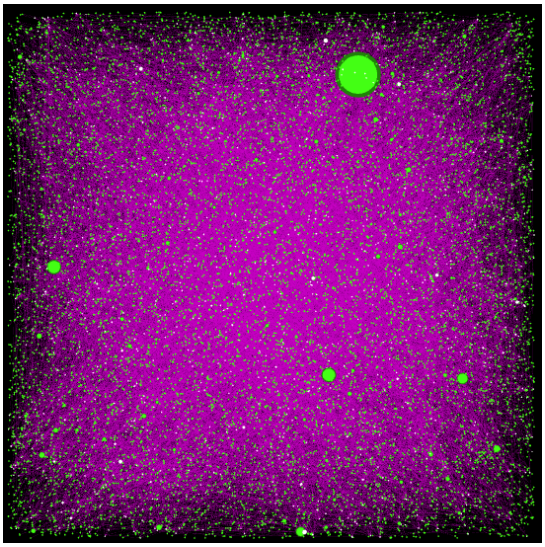

Рис. 10: Граф для 'Dijkstra's algorithm'

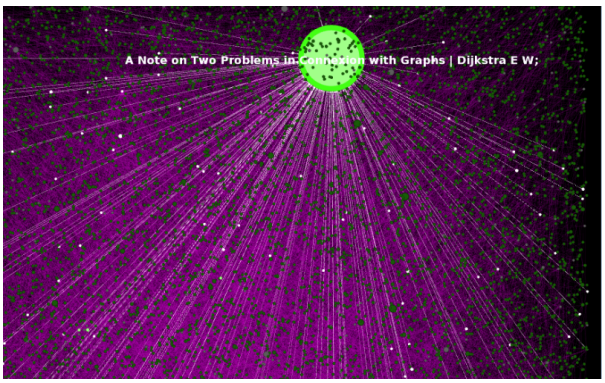

Рис. 11: Статья Дейкстры

MDS для данной выборки показан на рисунке 12.
Как видно, алгоритм показывает неплохой, но не самый лучший результат. Связано это с тем, что, во-первых, размерность исходного пространства - пространства термов - огромна. Во-вторых, на самом деле множество статей не имеет явно выраженных кластеров, все статьи по данному запросу, в общем-то, сильно схожи.
В любом случае, результат есть: один из кластеров (черный), в основном, занимает места, находящиеся около границы построенного эллипса, второй (красный) места в правом нижнем углу, третий (фиолетовый) оставшуюся внутренность эллипса.

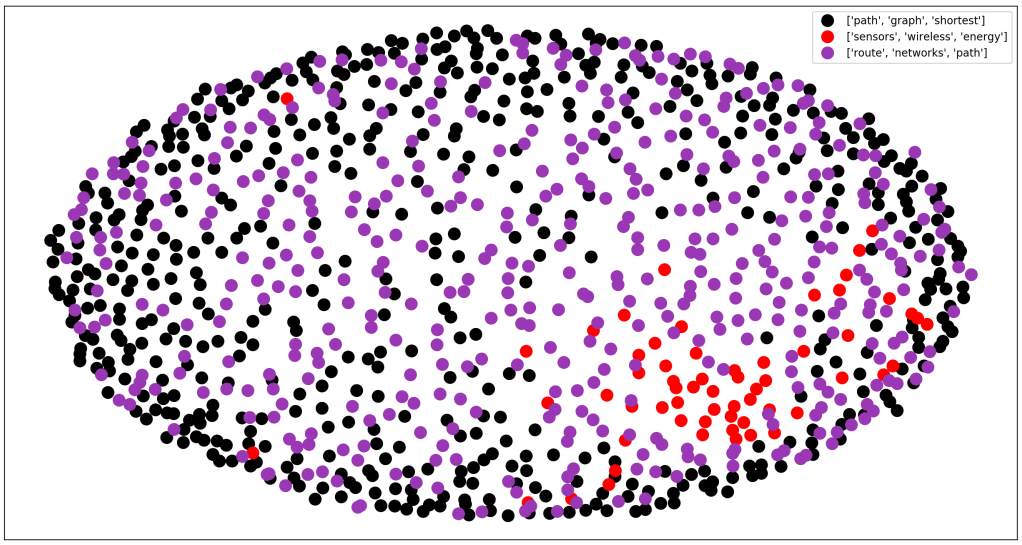

Рис. 12: MDS для 'Dijkstra's algorithm'

4 Заключение

В работе реализована программа, которая получает информацию об интересующих пользователя статьях с сайта Web of Science. На основе этой иноформации программа строит граф цитирований статей и проводит их кластеризацию.
Описаны методы, которые обычно используются для данного класса задач: алгоритмы кластеризации (K-Means, latent Dirichlet allocation) и анализа (multidimensional scaling, hierarchical document clustering).
Приведена инструкция по написанию программы.

Список литературы

[1] Spärk K.J, “A statistical interpretation of term specificity and its application in retrieval”, *Journal of Documentation*, **60**:5 (2004), 493–502.

[2] Andreev A., Berezkin D., Morozov V., Simakov K., “The method of clustering texts collections and clusters annotating”.

[3] MacQUEEN J., “Some methods for classification and analysis of multivariate observations”.

[4] Diane J. Hu, “Latent Dirichlet Allocation for Text, Images, and Music”.

[5] Blei D.M., Ng A.Y., Jordan M.I., “Latent Dirichlet Allocation”, *Journal of Machine Learning Research*, 2003, № 3, 993–1022.

[6] Kruskal J.B., “Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis”, *Psychometrika*, **29**:1 (1964), 1–27.

[7] Fung B.C.M., Wang K., Ester M., “Hierarchical Document Clustering”.

[8] Rose B., “Document Clustering with Python”, [Online] Available: http://brandonrose.org/clustering.