

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

Кафедра математической кибернетики и компьютерных наук

**ПОСТРОЕНИЕ И СРАВНЕНИЕ ТЕЗАУРУСОВ НА ОСНОВЕ АНАЛИЗА
ТЕКСТОВЫХ ДАННЫХ НОВОСТНЫХ ИСТОЧНИКОВ**

КУРСОВАЯ РАБОТА

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Янченко Вадима Александровича

Научный руководитель
доцент, к. ф.-м. н. _____ С. В. Папшев

Заведующий кафедрой
доцент, к. ф.-м. н. _____ С. В. Миронов

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1 Понятие тезауруса	4
2 Построение тезауруса	6
2.1 Word2Vec	7
2.1.1 Архитектура модели Skip-Gram	7
2.1.2 Negative sampling	9
2.2 GloVe	10
3 Практическая часть.....	12
3.1 Предварительная обработка данных	12
3.2 Построение тезаурусов	13
3.2.1 Word2Vec.....	13
3.2.2 GloVe.....	15
3.3 Сравнение тезаурусов	16
ЗАКЛЮЧЕНИЕ	19
Приложение А Программный код, выполняющий обработку текста	19
Приложение Б Получение случайных 20 слов	23
Приложение В Построение графа	25

ВВЕДЕНИЕ

В задачах обработки естественного языка (NLP) и информационного поиска (IR) важную роль играет использование различных видов знаний: лексических связей между словами, значений слов, специализированных понятий в предметных областях и знаний. Одним из традиционных способов представления такого рода знаний в системах NLP являются тезаурусы [?].

В контексте компьютерной обработки текста тезаурус представляет собой формализованный ресурс, в котором описаны семантические связи между словами или терминами — например, синонимы, гипонимы и другие виды лексических отношений. Благодаря такой структуре, тезаурусы могут использоваться в автоматизированных системах для анализа, поиска и интерпретации текстовой информации.

Использование готовых тезаурусов, обученных на огромном количестве произведений, не всегда даёт лучший результат из-за особенностей предметной области. Такие ресурсы, как правило, отражают общую лексику языка, но не учитывают контекстуальные особенности и терминологию конкретных тематик.

В связи с этим всё более актуальным становится автоматическое построение тезаурусов на основе анализа конкретных текстов. Такой подход позволяет выявить семантические связи, характерные именно для выбранного корпуса, что способствует более точному и релевантному представлению знаний.

Цель данной курсовой работы — построить и сравнить тезаурусы, созданные с помощью методов PMI, Word2Vec, GloVe и BERT, на основе корпуса новостных текстов. Это позволит оценить, насколько различаются семантические представления в зависимости от выбранной модели и насколько точно каждый подход отражает смысловые связи в реальных текстах.

Для достижения поставленной цели необходимо решить следующие задачи:

- изучить методы PMI, Word2Vec, GloVe и BERT в построении тезаурусов;
- построить тезаурусы с использованием этих методов;
- провести сравнительный анализ полученных тезаурусов

1 Понятие тезауруса

В информационном поиске часто прибегают к механизму расширения поискового запроса. Заключается он в добавлении связанных терминов или понятий к исходному запросу, по сути, переписывая его. Это помогает найти документы, которые могут не совпадать с исходными ключевыми словами, но при этом удовлетворять информационные потребности пользователя.

Для поиска синонимов или близких по смыслу слов прибегают к использованию тезауруса. Тезаурус представляет собой структурированный лексический ресурс, в котором фиксируются семантические отношения между словами и терминами. В отличие от обычных словарей, тезаурусы явно отображают связи между ними — такие как синонимия, антонимия, родо-видовые отношения, ассоциативные связи и другие.

Современные тезаурусы могут быть как ручными (созданными экспертами в конкретной предметной области), так и автоматически построенными на основе анализа текстов с применением статистических и нейросетевых моделей. Автоматические тезаурусы особенно актуальны в условиях быстро меняющейся лексики, характерной, например, для новостных источников.

Одним из наиболее известных лексических ресурсов в сфере компьютерной лингвистики и автоматической обработки текстов является компьютерный тезаурус WordNet. WordNet версии 3.0 включает приблизительно 155 тысяч различных лексем и словосочетаний, организованных в 117 тысяч понятий, или совокупностей синонимов (synset), общее число пар лексема – значение составляет более 200 тысяч.

Основным отношением в WordNet является отношение синонимии. Наборы синонимов — синсеты — являются основными структурными элементами WordNet. Понятие синонимии, используемое разработчиками WordNet, базируется на критерии, что два выражения являются синонимичными, если замена одного из них на другое в предложении не меняет значения истинности этого высказывания.

При этом не требуется заменяемости синонимов во всех контекстах – по такому критерию в естественном языке было бы слишком мало синонимов. Используется значительно более слабое утверждение, что синонимы WordNet должны быть взаимозаменимы хотя бы в некотором множестве контекстов. Например, замена *plank* (доска, планка) для слова *board* (доска) редко меняет

значение истинности в контексте плотницкого дела, но существуют контексты, где такая замена не может считаться приемлемой.

2 Построение тезауруса

Современные методы автоматического построения тезаурусов базируются на анализе больших текстовых корпусов и извлечении статистических или контекстуальных связей между словами. Основная идея таких подходов заключается в том, что слова, встречающиеся в схожих контекстах, как правило, имеют близкие значения. Возникает проблема отражения смысла слова.

Начнём с рассмотрения слова (возьмём, например, слово *рама*). Слово *рама* является леммой, то есть находится в начальной форме слова. Форма инфинитива используется в качестве леммы для глагола. Конкретные слова *рамы* или *мыли* являются словоформами.

Каждая лемма может иметь несколько значений. Тот факт, что леммы обладают полисемией (многозначностью) может затруднить интерпретацию.

Кроме того, возможны случаи, когда одно слово имеет смысл, значение которого идентично смыслу другого слова или почти идентично. Такие слова называют синонимами. Примеры синонимов: *кавалерия* — *конница*, *смелый* — *храбрый*.

Более формальное определение синонимии следующее: два слова являются синонимами, если они заменяют друг друга в любом предложении без изменения условий истинности предложения, то есть ситуаций, в которых предложение будет истинным.

На практике же два слова вероятно не обладают абсолютно идентичным смыслом. Слова могут различаться оттенками в значениях (*мокрый* — *влажный*) или употребляться в разных сферах (*жена* (общеупотр.) — *супруга* (офиц.)).

Хотя у слов не так много синонимов, у большинства слов есть много похожих слов. Кошка не является синонимом собаки, но кошки и собаки — это, безусловно, похожие слова. При переходе от синонимии к сходству будет полезно перейти от разговора об отношениях между смыслами слов (как при синонимии) к отношениям между словами (как при сходстве). Работа со словами позволяет избежать необходимости придерживаться определенного представления смыслов слов, что, как выясняется, упрощает нашу задачу.

Понятие сходства слов очень полезно в больших семантических задачах. Знание того, насколько похожи два слова, может помочь в вычислении того, насколько схожи значения двух фраз или предложений.

2.1 Word2Vec

Ключевая идея состоит в представлении слова как вектора в многомерном пространстве. Самый простой способ перехода от слова к вектору состоит в следующем. Пусть V — множество всех слов. Слово $w_i \in V$ представим как вектор размера $n = |V|$, в котором все координаты равны 0 кроме i -ой координаты, которая равна 1.

Рассмотрим другой способ представление слова как вектора — эмбеддинг. Эмбеддинги имеет количество размерностей d от 50 до 1000, что существенно меньше количества слов $|V|$ во всех документах. Кроме того, эти векторы плотные: вместо того, чтобы большинство координат были равны 0, значения будут вещественными числами, которые могут быть и отрицательными.

Эмбеддинги Word2Vec являются статическими, это означает, что каждому слову соответствует ровно один эмбеддинг. Такие модели как BERT позволяют создавать контекстуальные эмбеддинги, которые сопоставляют разный вектор к слову в зависимости от контекста.

Word2Vec реализуется в двух основных архитектурах: **CBOW** и **Skip-Gram**.

2.1.1 Архитектура модели Skip-Gram

Модель Skip-Gram является одной из двух основных архитектур Word2Vec и направлена на предсказание контекстных слов по центральному слову. То есть, имея слово w_t , модель обучается предсказывать слова, находящиеся в некотором окне вокруг него: $w_{t-c}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+c}$, где c — размер окна.

Модель Skip-Gram представляет собой простую нейросеть с одним скрытым слоем. Она состоит из следующих компонентов:

1. **Входной слой.** Каждое слово из словаря V кодируется как one-hot вектор $\mathbf{x} \in \mathbb{R}^{|V|}$, в котором все элементы равны нулю, кроме одного, соответствующего индексу слова w .
2. **Скрытый слой.** Представлен матрицей весов $W \in \mathbb{R}^{|V| \times d}$, где d — размерность эмбеддингового пространства. Преобразование входа:

$$\mathbf{h} = \mathbf{x}^\top W$$

Вектор $\mathbf{h} \in \mathbb{R}^d$ и является искомым эмбеддингом слова w .

3. **Выходной слой.** Представлен второй матрицей весов $W' \in \mathbb{R}^{d \times |V|}$. Результат выходного слоя:

$$\mathbf{u} = W'^\top \mathbf{h}$$

где $\mathbf{u} \in \mathbb{R}^{|V|}$ — логиты для каждого слова в словаре.

4. **Softmax.** На выходе применяется softmax-функция, преобразующая логиты в вероятности:

$$P(w_O \mid w_I) = \frac{\exp(\mathbf{v}'_{w_O} \cdot \mathbf{v}_{w_I})}{\sum_{w \in V} \exp(\mathbf{v}'_w \cdot \mathbf{v}_{w_I})}$$

Здесь:

- w_I — входное (центральное) слово;
- w_O — слово из контекста;
- \mathbf{v}_{w_I} — вектор из матрицы W (входной эмбеддинг);
- \mathbf{v}'_{w_O} — вектор из матрицы W' (выходной эмбеддинг).

Рассмотрим функцию потерь. Для одного примера (w_I, w_O) используется кросс-энтропия:

$$\mathcal{L} = -\log P(w_O \mid w_I)$$

Общая функция потерь по корпусу:

$$\mathcal{L}_{\text{total}} = - \sum_{t=1}^T \sum_{\substack{-c \leq j \leq c \\ j \neq 0}} \log P(w_{t+j} \mid w_t)$$

где T — длина корпуса.

Эмбеддингом является как матрица (W), так и матрица(W') — часто используется только W .

Что стоит отметить: хотя в модель не заложено явно никакой семантики, а только статистические свойства корпусов текстов, оказывается, что натренированная модель word2vec может улавливать некоторые семантические свойства слов. Классический пример: слово «мужчина» относится к слову «женщина» так же, как слово «дядя» к слову «тётя», что для нас совершенно естественно и понятно, но в других моделях добиться такого же соотношения векторов можно только с помощью специальных ухищрений. Здесь же — это происходит естественно из самого корпуса текстов.

2.1.2 Negative sampling

Полноценный расчёт softmax-функции в модели Skip-Gram требует вычисления скалярного произведения с каждым словом в словаре, что приводит к высокой вычислительной сложности: $\mathcal{O}(|V|)$ на каждый пример.

Для решения этой проблемы в модели Word2Vec используется приближённый метод оптимизации — negative sampling.

Вместо того чтобы обучать модель различать целевое слово w_O от всех остальных слов в словаре V , negative sampling предлагает:

1. обучать модель различать настоящие пары (центральное слово и слово из его контекста),
2. и несколько случайных пар (центральное слово и случайные, нерелевантные слова), которые считаются негативными примерами.

Пусть:

- w_I — центральное слово (input word),
- w_O — контекстное слово (output word, положительный пример),
- $w_1^-, w_2^-, \dots, w_K^-$ — отрицательные примеры, выбранные случайнм образом из словаря по заданному распределению.

Тогда оптимизируется следующая функция потерь:

$$\mathcal{L} = \log \sigma(\mathbf{v}'_{w_O} \cdot \mathbf{v}_{w_I}) + \sum_{k=1}^K \mathbb{E}_{w_k^- \sim P_n(w)} [\log \sigma(-\mathbf{v}'_{w_k^-} \cdot \mathbf{v}_{w_I})]$$

где:

- $\sigma(x) = \frac{1}{1+e^{-x}}$ — сигмоида;
- \mathbf{v}_{w_I} — входной эмбеддинг слова w_I ;
- \mathbf{v}'_{w_O} — выходной эмбеддинг контекстного слова w_O ;
- $P_n(w)$ — распределение, по которому выбираются отрицательные примеры;
- K — число отрицательных примеров.

Для выбора отрицательных слов используется неравномерное распределение. По умолчанию (в оригинальной статье Word2Vec) это распределение частот слов в степени 3/4:

$$P_n(w) = \frac{f(w)^{3/4}}{\sum_{w' \in V} f(w')^{3/4}}$$

где $f(w)$ — частота слова w в корпусе.

2.2 GloVe

Модель GloVe (Global Vectors) предназначена для построения векторных представлений слов на основе глобальной статистики совместных появлений слов в тексте. В отличие от моделей Word2Vec, которые обучаются на локальных контекстных окнах, GloVe использует информацию о числе совместных появлений слов во всём корпусе, агрегируя её в специальную матрицу.

На первом этапе строится матрица совместных появлений $X \in \mathbb{R}^{|V| \times |V|}$, где X_{ij} — количество раз, когда слово j встречается в контексте слова i . Контекст обычно ограничивается окном фиксированного размера (например, 5 слов влево и вправо), при этом можно учитывать взвешивание по расстоянию до центрального слова.

GloVe обучает два векторных представления: $w_i \in \mathbb{R}^d$ — для центрального слова i и $\tilde{w}_j \in \mathbb{R}^d$ — для контекстного слова j , а также два скалярных смещения b_i и \tilde{b}_j . Обучение направлено на аппроксимацию следующего равенства:

$$w_i^\top \tilde{w}_j + b_i + \tilde{b}_j \approx \log X_{ij}$$

Целевая функция, которую минимизирует модель, записывается в следующем виде:

$$J = \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} f(X_{ij}) \left(w_i^\top \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2$$

где $f(X_{ij})$ — весовая функция, ограничивающая влияние часто встречающихся пар слов, например:

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^\alpha & \text{если } x < x_{\max} \\ 1 & \text{иначе} \end{cases}$$

На практике используются значения $\alpha = 0,75$ и $x_{\max} = 100$.

После завершения обучения итоговое представление слова i получается как сумма обученных векторов:

$$v_i = w_i + \tilde{w}_i$$

Таким образом, каждая лексема кодируется плотным вектором фиксированной размерности, отражающим её глобальные статистические связи с другими словами корпуса.

3 Практическая часть

В рамках практической части осуществляется построение тезауруса на основе новостей сайтов СГУ и СГТУ. В процессе построения используются описанные выше модели Word2Vec и GloVe.

Весь программный код написан на языке программирования python версии 3.12.6. В программном коде используются следующие библиотеки:

- pandas для удобного управления данными;
- gensim для работы с моделями

3.1 Предварительная обработка данных

Рассмотрим данные, которые представляют из себя новости, взятые из официального сайта СГУ. Каждый документ характеризуется следующими параметрами:

1. дата написания новости;
2. заголовок новости;
3. необработанный текст новости;
4. обработанный текст новости;
5. количество слов;
6. количество токенов.

Обработка текста заключается в удалении стоп-слов и лемматизации слов.

Стоп-слова — это слова, которые встречаются в языке чрезвычайно часто, но, как правило, не несут самостоятельной смысловой нагрузки в контексте информационного поиска или анализа текста. К таким словам относятся, например, союзы, предлоги, частицы, местоимения: «и», «в», «на», «что», «как», «это», и т.п. Их исключение позволяет снизить «шум» в текстах, повысить эффективность алгоритмов и сосредоточиться на лексемах, действительно значимых для понимания содержания документа.

Лемматизация, в свою очередь, представляет собой процесс приведения слова к его начальной, словарной форме — лемме. Это позволяет уменьшить количество уникальных словоформ в тексте, объединив различные грамматические формы одного и того же слова (например, «пошёл», «идёт», «шли» — к лемме «идти»). Благодаря лемматизации повышается точность анализа, поскольку слова с одинаковым значением, но разными формами, будут обрабатываться как единое понятие.

В новостях часто встречаются имена людей, городов, стран, наименования различных компаний, которые не привносят смысл в предложение, поэтому от этого также стоит избавиться.

Программный код, выполняющий обработку текста, можно рассмотреть в приложении А.

Ознакомиться со статистикой полученных данных можно в табл. 3.1.

Статистика	Значение
Количество новостей	31077
Общее количество предложений	423175
Среднее количество предложений на новость	13.62
Общее количество токенов	4293732
Среднее количество токенов на новость	138.16
Максимальное количество токенов на новость	166
Минимальное количество токенов на новость	1
Количество уникальных токенов	45204

Таблица 3.1 – Статистика по полученным данным

3.2 Построение тезаурусов

3.2.1 Word2Vec

Обучим модель Word2Vec на полученных данных. Для этого напишем следующую программу:

```
1 import dask.dataframe as dd
2 import dask.bag as db
3 from gensim.models import Word2Vec
4 docs = dd.read_parquet("../output.pq/")
5 texts = docs['News_Tokens'].compute()
6 bag = db.from_sequence(texts)
7 sentences = bag.flatten().map(lambda x: x.split()).compute()
8 model = Word2Vec(sentences=sentences, vector_size=250, window=7,min_count=5,
   ↴ sg=0)
9 model.save('word2vec_sent_5_sgu.model')
```

Получив модель, построим тезаурус:

```
1 import pandas as pd
2 import dask.dataframe as dd
3 import dask.bag as db
```

```

4  from gensim.models import Word2Vec
5  docs = dd.read_parquet("../output.pq/")
6  model = Word2Vec.load("word2vec_sent_5_sgu.model")
7  texts = docs['News_Tokens'].compute()
8  bag = db.from_sequence(texts)
9  sentences = bag.flatten()
10 words = sentences.map(lambda x: x.split()).flatten().distinct().compute()
11 results = []
12 threshold = 0.7
13 for word in words:
14     try:
15         # Получаем 10 ближайших слов
16         similar_words = model.wv.most_similar(word, topn=10)
17         filtered_words = [(w, sim) for w, sim in similar_words if sim >=
18             threshold]
19         results.append([word, filtered_words])
20     except KeyError:
21         # Если слово не в модели, пропускаем его
22         print(f"Слово '{word}' не найдено в модели.")
23 df_results = pd.DataFrame(results, columns=["Word", "Most_Similar_Word"])
24
25 df_results.to_csv("similar_words_sent_5.csv", index=False)

```

Первые 10 строчек полученного файла:

Word	Most_Similar_Word
май	
состояться	
акция	[]
бессмертный	
полк	
который	[]
принять	
участие	[]
студент	[]

Таблица 3.2 – Первые 10 строчек тезауруса, построенного моделью Word2Vec

3.2.2 GloVe

Для создания модели GloVe склонируем их официальный репозиторий¹.

Далее выполним следующие команды:

```
1 build/vocab_count -min-count 5 -verbose 2 < corpus.txt > vocab.txt
2 build/cooccur -memory 4.0 -vocab-file vocab.txt -verbose 2 -window-size 7 <
   ↵ corpus.txt >cooccurrence.bin
3 build/shuffle -memory 4.0 -verbose 2 < cooccurrence.bin >
   ↵ cooccurrence.shuf.bin
4 build/glove -save-file vectors -threads 12 -input-file cooccurrence.shuf.bin
   ↵ -x-max 10 -iter 15 -vector-size 250 -binary 2 -vocab-file vocab.txt
   ↵ -verbose 2
```

По полученной модели построим тезаурус:

```
1 import pandas as pd
2 from gensim import models
3 import dask.dataframe as dd
4 import dask.bag as db
5
6 model = models.KeyedVectors.load_word2vec_format('vectors.txt', binary=False,
   ↵ no_header=True)
7 docs = dd.read_parquet("../output.pq/")
8 texts = docs['News_Tokens'].compute()
9 bag = db.from_sequence(texts)
10 sentences = bag.flatten()
11 words = sentences.map(lambda x: x.split()).flatten().distinct().compute()
12 results = []
13 threshold = 0.7
14 skipped=0
15 for word in words:
16     try:
17         # Получаем 10 ближайших слов
18         similar_words = model.most_similar(word, topn=10)
19         filtered_words = [(w, sim) for w, sim in similar_words if sim >=
   ↵ threshold]
20         results.append([word, filtered_words])
21     except KeyError:
22         # Если слово не в модели, пропускаем его
23         print(f"Слово '{word}' не найдено в модели.")
24         skipped += 1
```

¹<https://github.com/stanfordnlp/GloVe>

```

25     continue
26
27 df_results = pd.DataFrame(results, columns=[ "Word" , "Most_Similar_Word" ])
28 print(f "Пропустили {skipped} слов")
29 df_results.to_csv("sim-words_sent_5_glove.csv" , index=False)

```

3.3 Сравнение тезаурусов

Возьмём случайные 10 слов и найдём самые близкие слова по мнению каждой модели. Ознакомиться с программным кодом, который выполняет это можно в приложении Б. Результат

Полученный результат отражен на табл. 3.3.

Word	GloVe	Word2Vec
недорого	[('развлекаться', 0.77), ('метать', 0.73), ('непогода', 0.72), ('энергично', 0.72), ('географически', 0.71)]	[('инк', 0.86), ('прожект', 0.86), ('хармоны', 0.85), ('траффик', 0.85), ('берингия', 0.85)]
голод	[('холод', 0.72)]	[('фашист', 0.86), ('холод', 0.84), ('умирать', 0.83), ('утрата', 0.83), ('умерший', 0.83)]
орфография	[('пунктуация', 0.86)]	[('пунктуация', 0.72)]
внимательность	[('сосредоточенность', 0.74)]	[('быстрота', 0.84), ('коммуникабельность', 0.78), ('смекалка', 0.77), ('сноровка', 0.77), ('находчивость', 0.76)]
обсценный	[('нецензурный', 0.76), ('лексика', 0.74)]	[('жаргон', 0.80), ('фразеологический', 0.79), ('иврит', 0.78), ('искра', 0.77), ('онтогенетический', 0.77)]
трещина	[('ворона', 0.71)]	[('термометр', 0.90), ('макароны', 0.89), ('укладка', 0.89), ('наледь', 0.89), ('балка', 0.89)]
кадмий	[('селенид', 0.86)]	[('селенид', 0.94), ('силанизировать', 0.87), ('стабилизировать', 0.87), ('альбумин', 0.86), ('бесшовный', 0.86)]
омс	[('полис', 0.70)]	[('полис', 0.91), ('снилс', 0.86), ('дубликат', 0.82), ('предъявление', 0.82), ('пин', 0.81)]
автомат	[('калашников', 0.74)]	[('калашников', 0.75), ('разборка', 0.73), ('сборка', 0.72), ('гранат', 0.71)]
жажда	[('утолять', 0.84)]	[('сдерживать', 0.77), ('запах', 0.77), ('наполнять', 0.77), ('притворство', 0.77), ('аромат', 0.77)]

Таблица 3.3 – Самых близких слов по мнению GloVe и Word2Vec для 10 случайных слов из корпуса

Видим, что в целом модель GloVe даёт более точные ответы. Кроме этого, модель Word2Vec предлагает сильно больше близких слов, большинство из которых ничего общего с рассматриваемым словом не имеет. Заметим, что существуют слова, для которых модель дает совсем неправильные ответы (слово *недорого, трещина*).

Сгенерируем для каждой модели граф. Граф является ориентированным. Вершиной в графе является слово, а ребро — факт того, что слово является близким для рассматриваемого с весом, равным косинусному расстоянию между ними. С программным кодом, генерирующим подобный граф, можно ознакомиться в приложении В.

Характеристики графов отражены в табл. 3.4.

Характеристика	GloVe	Word2Vec
Кол-во вершин	18402	18401
Кол-во ребер	9921	86834
Кол-во неизолированных вершин	3690	12908
Кол-во компонент (без учёта изолированных вершин)	770	337
Кол-во сообществ	800	349
Модулярность	0.453	0.597

Таблица 3.4 – Сравнительная характеристика графов моделей

Визуализируем полученные графы с помощью программы Gephi — бесплатный пакет программ для анализа и визуализации сетей. Ознакомиться с результатом можно на рисунках 3.1, 3.2.

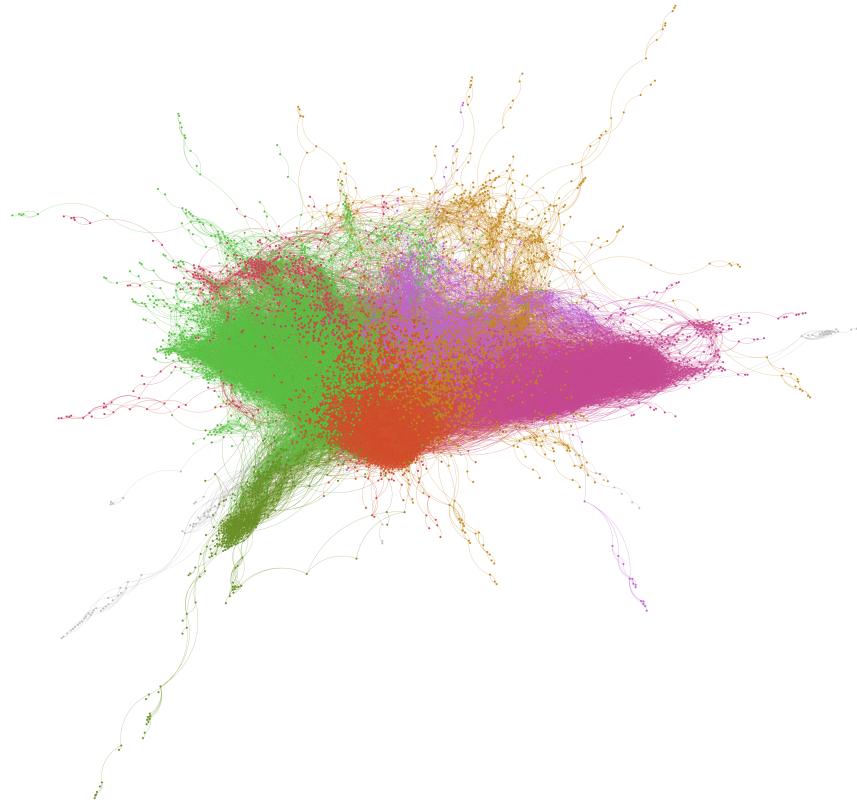


Рисунок 3.1 – Граф, полученный моделью Word2Vec

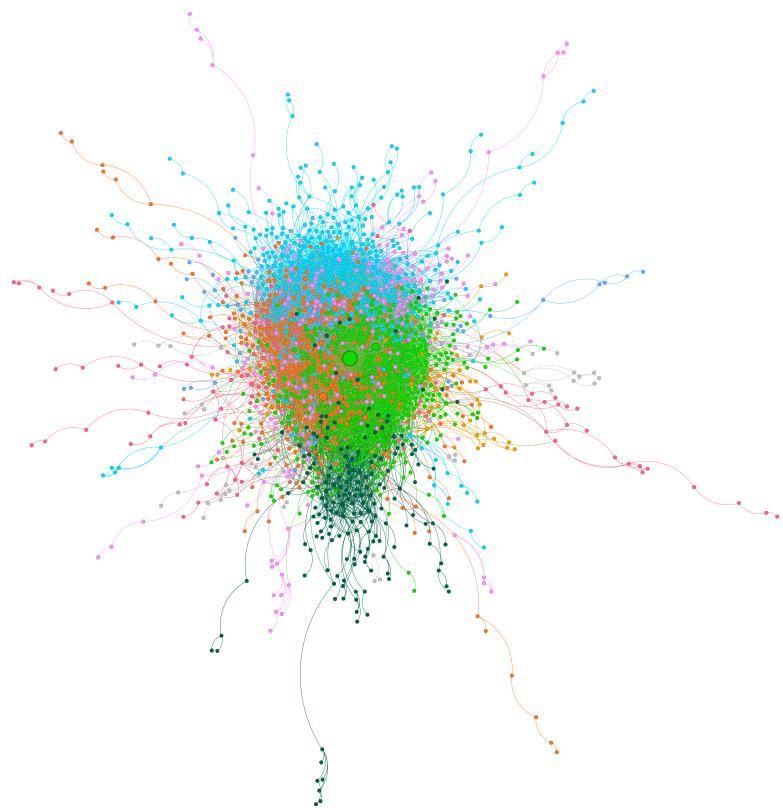


Рисунок 3.2 – Граф, полученный моделью GloVe

ЗАКЛЮЧЕНИЕ

ПРИЛОЖЕНИЕ А

Программный код, выполняющий обработку текста

```
1 import pandas as pd
2 import pymorphy3
3 import re
4 import dask.dataframe as dd
5
6 def string_dev(a_in):
7     x_in = re.sub(r'\n', ' ', a_in)
8     y_in = x_in.lower()
9     b_in = re.sub(r'&w+;', ' ', y_in)
10    d_in = re.sub(r'<[^>]*>', ' ', b_in)
11    f_in = re.sub(r'www|.w+.w{2,3}?', ' ', d_in)
12    a_in = re.sub(r'\xad', ' ', f_in)
13    c_in = re.sub(r'\\xa0-', ' ', a_in)
14    u_in = re.sub(r'\\u200e', ' ', c_in)
15    w_in = re.sub(r'\d+', ' ', u_in)
16    y_in = re.sub(r'\-', ' ', w_in)
17    yy_in = re.sub(r'_+', ' ', y_in)
18    yyy_in = re.sub(r'[---/-()]\[ \\\\ ,\-\:;<>@#%\\|\\|+|*``&~\$|^<>i-]+',
19                     ' ', yy_in)
20    q_in = re.sub(r'[a-z]*', ' ', yyy_in)
21    sss_in = re.sub(r'\b\w{,2}\b', ' ', q_in)
22    qms_in = re.sub(r'\s{2,}', ' ', sss_in)
23    nms_in = qms_in.strip()
24
25    return nms_in
26
27 def lematization(f_input_list):
28     morph = pymorphy3.MorphAnalyzer()
29     lnorm = []
30     for word in f_input_list:
31         p = morph.parse(word)[0]
32         lnorm.append(p.normal_form)
33
34     return lnorm
35
36 def del_my_stop_words(word_tokens):
37     stop_words_nltk = {'который', 'кому', 'имя', 'сегодня', 'вчера',
38                       'завтра', 'также', 'в', 'во', 'свой',
```

```

35      'это', 'часто', 'зачастую', 'мочь', 'смочь', 'а',
        ↳ 'без', 'более', 'больше', 'будет',
36      'будто', 'бы', 'был', 'была', 'были', 'было',
        ↳ 'быть', 'в', 'вам', 'vas', 'вдруг',
37      'ведь', 'во', 'вот', 'впрочем', 'все', 'всегда',
        ↳ 'всего', 'всех', 'всю', 'вы', 'где',
38      'да', 'даже', 'два', 'для', 'до', 'другой', 'его',
        ↳ 'ее', 'ей', 'ему', 'если', 'есть',
39      'еще', 'ж', 'же', 'за', 'зачем', 'здесь', 'и', 'из',
        ↳ 'или', 'им', 'иногда', 'их', 'к',
40      'как', 'какая', 'какой', 'когда', 'конечно', 'кто',
        ↳ 'куда', 'ли', 'лучше', 'между',
41      'меня', 'мне', 'много', 'может', 'можно', 'мой',
        ↳ 'моя', 'мы', 'на', 'над', 'надо',
42      'наконец', 'нас', 'не', 'него', 'нее', 'ней',
        ↳ 'нельзя', 'нет', 'ни', 'нибудь', 'никогда',
43      'ним', 'них', 'ничего', 'но', 'ну', 'о', 'об',
        ↳ 'один', 'он', 'она', 'они', 'опять', 'ом',
44      'перед', 'по', 'под', 'после', 'потом', 'потому',
        ↳ 'почти', 'при', 'про', 'раз', 'разве',
45      'с', 'со', 'сам', 'свою', 'себе', 'себя', 'сейчас',
        ↳ 'с', 'со', 'совсем', 'так', 'такой',
46      'там', 'тебя', 'тем', 'теперь', 'то', 'тогда',
        ↳ 'того', 'тоже', 'только', 'том', 'том',
47      'три', 'тут', 'ты', 'у', 'уж', 'уже', 'хорошо',
        ↳ 'хоть', 'что', 'чего', 'чем', 'через',
48      'что', 'чтоб', 'чтобы', 'чуть', 'эти', 'этого',
        ↳ 'этой', 'этом', 'этот', 'эту', 'я',
49      'сказал', 'человек', 'жизнь', 'говорил', 'кажется',
        ↳ 'сказать', 'сегодня', 'сказала',
50      'сказал'}

51

52      my_stop_words = {'сгт', 'свой', 'стать', 'кроме', 'разный', 'около',
        ↳ 'затем', 'помимо', 'ваш', 'вам',
53          'некоторый', 'лишь', 'каждый', 'самый', 'также',
        ↳ 'неоднократно', 'ещё', 'сразу', 'среди',
54          'однако', 'вновь', 'иной', 'ныне', 'пока', 'хотя',
        ↳ 'либо', 'немного', 'гораздо', 'ничто',
55          'нередко', 'наоборот', 'впереди', 'таковой', 'мимо',
        ↳ 'тесно', 'вряд', 'нечто', 'почём',
56          'почему', 'любой', 'обратно', 'оттуда', 'очень',
        ↳ 'понапрасну', 'поскольку', 'поэтому',

```

```

57             'прежде', 'причём', 'прочий', 'пусть', 'наш',
58             ↳ 'несколько', 'никак', 'твои', 'подробный',
59             'информация' }

60 stop_words = stop_words_nltk | my_stop_words
61 filtered_sentence = [w for w in word_tokens if w not in stop_words]
62 return filtered_sentence

63

64 from natasha import Segmenter, MorphVocab, NewsEmbedding, NewsMorphTagger,
65     ↳ NewsNERTagger, Doc

66 segmenter = Segmenter()
67 morph_vocab = MorphVocab()
68 emb = NewsEmbedding()
69 morph_tagger = NewsMorphTagger(emb)
70 ner_tagger = NewsNERTagger(emb)

71

72 def remove_proper_nouns(text):
73     if not isinstance(text, str) or not text.strip():
74         print("text имеет неожидаемый тип")
75         return "error type"
76     try:
77         doc = Doc(text)
78         doc.segment(segmenter)
79         doc.tag_morph(morph_tagger)
80         doc.tag_ner(ner_tagger)
81         if not doc.spans:
82             return text
83         spans_to_remove = [span for span in doc.spans if span.type in ['PER',
84             ↳ 'LOC', 'ORG']]]
85         text_cleaned = text
86         for span in sorted(spans_to_remove, key=lambda x: x.start,
87             ↳ reverse=True):
88             text_cleaned = text_cleaned[:span.start] +
89             ↳ text_cleaned[span.stop:]
90         print(text_cleaned)
91         return text_cleaned.strip()
92     except Exception as e:
93         print(f"Ошибка при обработке текста: {text[:50]}... Ошибка:
94             ↳ {str(e)}")
95         return "error"

```

```

92
93 def process_str_of_the_news(string):
94     string_2 = string_dev(string)
95     if len(string_2) != 0:
96         list_of_sentences = re.split(r '[. !?]', string_2)
97         list_of_sentences = [x.split(' ') for x in list_of_sentences if x]
98         filtered_sentences = []
99         for sentence in list_of_sentences:
100             no_stopwords_1 = del_my_stop_words(sentence)
101             lemmatized = lematization(no_stopwords_1)
102             text = ' '.join(lemmatized)
103             text = re.sub(r '\b\w{1,2}\b', ' ', text)
104             text = re.sub(r '\s{2,}', ' ', text)
105             text = text.strip()
106             if text:
107                 filtered_sentences.append(text)
108
109
110 import pyarrow as pa
111 def apply_tokenization():
112     docs = dd.read_parquet("raw-data.pq").repartition(npartitions=8).loc[:1]
113     print("Read finished")
114     docs[ 'News_Tokens' ] = docs[ 'News_Text' ].map_partitions(
115         lambda s: s.apply(remove_proper_nouns),
116         meta=( "News_Tokens" , object)
117     )
118     docs[ 'News_Tokens' ] = docs[ "News_Tokens" ].map_partitions(
119         lambda s: s.apply(process_str_of_the_news),
120         meta=( "News_Tokens" , object)
121     )
122     print("Returning")
123     schema = {
124         "News_Text": pa.string(),
125         "News_Tokens": pa.list_(pa.string()),
126         "News_Title": pa.string()
127     }
128     return docs.to_parquet("output.pq", schema=schema,
129                           write_metadata_file=True)
130
apply_tokenization()

```

ПРИЛОЖЕНИЕ Б

Получение случайных 20 слов

```
1 import pandas as pd
2 import os
3
4 # Пути к файлам и соответствующие названия колонок
5 file_paths = {
6     # 'bert': '/bert/sim-words5-bert.csv',
7     'glove': '/glove_python/sim-words_sent_5_glove.csv',
8     'word2vec': '/word2vec/similar_words_sent_5.csv'
9 }
10
11 # Создаем пустой DataFrame с колонкой 'word'
12 data = pd.DataFrame(columns=['word'])
13
14 # Обрабатываем каждый файл
15 for model_name, path in file_paths.items():
16     # Читаем CSV-файл
17     full_path = ".." + path
18     df = pd.read_csv(full_path)
19
20     # Переименовываем колонку Most_Similar_Word в название модели
21     df = df.rename(columns={'Most_Similar_Word': model_name})
22
23     # Если это первый файл, используем его слова как основу
24     if data.empty:
25         data['word'] = df['Word']
26
27     # Добавляем данные в основной DataFrame
28     data[model_name] = df[model_name]
29 import ast
30 # Функция для проверки, является ли значение пустым списком или NaN
31 def is_valid_similar_words(value):
32     if pd.isna(value): # проверяем NaN
33         return False
34     if isinstance(value, str): # если данные хранятся как строки (например,
35     ↪ "[('слово', 0.5), ...]")
36         try:
37             lst = ast.literal_eval(value) # преобразуем строку в список
38             return len(lst) > 0 # True, если список не пустой
39         except (ValueError, SyntaxError):
```

```

39         return False
40     elif isinstance(value, list): # если данные уже в формате списка
41         return len(value) > 0
42     else:
43         return False # на случай других форматов
44
45 # Применяем фильтрацию: оставляем строки, где ВСЕ столбцы с похожими словами
46 #→ не пусты
46 filtered_data = data[
47     data[ 'glove' ].apply(is_valid_similar_words) &
48     data[ 'word2vec' ].apply(is_valid_similar_words)
49 ]
50 n = 25 # кол-во случайных строк
51
52 random_sample = filtered_data.sample(n=n, random_state=42) # random_state для
53 #→ воспроизводимости
54
54 # Сохраняем в новый CSV-файл
55 output_file = 'random_sample_similar_words.csv'
56 random_sample.to_csv(output_file, index=False, encoding='utf-8')

```

ПРИЛОЖЕНИЕ В

Построение графа

```
1 import pandas as pd
2 import os
3
4 # Пути к файлам и соответствующие названия колонок
5 file_paths = {
6     # 'bert': '/bert/sim-words5-bert.csv',
7     'glove': '/glove_python/sim-words_sent_5_glove.csv',
8     'word2vec': '/word2vec/similar_words_sent_5.csv'
9 }
10
11
12 data = {}
13 for model_name, path in file_paths.items():
14     full_path = ".." + path
15     df = pd.read_csv(full_path)
16     data[model_name] = df
17 import networkx as nx
18 from ast import literal_eval
19
20 graphs = []
21 for model_name, df in data.items():
22     G = nx.Graph() # или nx.DiGraph() для направленного графа
23
24     for _, row in df.iterrows():
25         word = row['Word']
26         similar_words = row['Most_Similar_Word']
27
28         # Пропускаем пустые списки
29         if not similar_words or pd.isna(similar_words):
30             continue
31
32         # Преобразуем строку в список кортежей (если данные в формате строки)
33         if isinstance(similar_words, str):
34             try:
35                 similar_words = literal_eval(similar_words)
36             except (ValueError, SyntaxError):
37                 continue
38
39         # Добавляем рёбра в граф
```

```

40         for similar_word, weight in similar_words:
41             G.add_edge(word, similar_word, weight=weight)
42 graphs[model_name] = G
43 # nx.write_gexf(G, f"{model_name}_graph.gexf") # формат GEXF для Gephi
44 print(f "Граф '{model_name}' содержит {len(G.nodes)} узлов и
45       {len(G.edges)} ребер.")
46 from community import community_louvain
47 for model_name, G in graphs.items():
48     print(f "model name: {model_name}")
49     partition = community_louvain.best_partition(G, weight='weight')
50     # Кол-во кластеров
51     num_clusters = max(partition.values()) + 1
52     print(f "Число кластеров (Louvain): {num_clusters}")
53
54     # Размеры кластеров
55     from collections import Counter
56     cluster_sizes = Counter(partition.values())
57     print(f "Размеры кластеров: {cluster_sizes.most_common(5)}" # Top-5
58           кластеров
59
60     # 3. Модулярность (качество кластеризации)
61     modularity = community_louvain.modularity(partition, G, weight='weight')
62     print(f "Модулярность: {modularity:.3f}")

```