Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Новосибирский государственный технический университет»

Кафедра теоретической и прикладной информатики

**Отчет ПО ПРАКТИКЕ**

Производственная практика: технологическая (проектно-технологическая) практика

(наименование практики в соответствии с учебным планом)

Направление подготовки: 01.03.02 Прикладная математика и информатика

|  |  |
| --- | --- |
| Выполнил:  Студент Кузнецов К.В.  (Ф.И.О.)  Группа ПМ-01  Факультет ПМИ.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  подпись  «\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 202\_ г. | Проверил:  Руководитель от НГТУ Тимофеева А.Ю.  (Ф.И.О.)  Балл: \_\_\_\_\_\_\_\_\_\_, ECTS\_\_\_\_\_\_\_\_\_\_\_\_\_,  Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «отлично», «хорошо», «удовлетворительно», «неуд.»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  подпись  «\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 202\_ г. |

Новосибирск 2024

**Оглавление**

[1. Введение 3](#_Toc198746694)

[1.1. Актуальность темы 3](#_Toc198746695)

[1.2. Цели и задачи проекта 3](#_Toc198746696)

[2. Метрики измерения сходства коротких текстов 5](#_Toc198746697)

[2.1. Строковые метрики 5](#_Toc198746698)

[2.2. Корпусные метрики 7](#_Toc198746699)

[2.3 Семантические методы 9](#_Toc198746700)

[3. Разработка модуля генерации выборок текстов 11](#_Toc198746701)

[4. Реализация системы сравнения метрик на Python 13](#_Toc198746702)

[4.1. Архитектура модуля 13](#_Toc198746703)

[4.2. Реализация строковых метрик 13](#_Toc198746704)

[4.3. Реализация корпусных метрик 13](#_Toc198746705)

[4.4. Реализация семантических метрик 14](#_Toc198746706)

[4.5. Профилирование времени и памяти 14](#_Toc198746707)

[4.6. Сохранение результатов 15](#_Toc198746708)

[5. Результаты исследования 16](#_Toc198746709)

[5.1. Строковые типы 16](#_Toc198746710)

[5.2. Корпусные метрики 17](#_Toc198746711)

[5.3. Семантические метрики 18](#_Toc198746712)

[5.4. Результаты при выборке из длинных слов 19](#_Toc198746713)

[6. Анализ эффективности метрик 20](#_Toc198746714)

[7. Заключение 24](#_Toc198746715)

[8. Список литературы 26](#_Toc198746716)

[9. Приложение 27](#_Toc198746717)

# 1. Введение

1.1. Актуальность темы

В современную эпоху цифровых коммуникаций и массового производства текстовой информации задача измерения текстового сходства становится всё более актуальной. Сравнение коротких текстов необходимо в таких прикладных задачах, как автоматическое дублирование записей, интеллектуальный поиск, кластеризация отзывов, семантический анализ, формирование диалогов в чат-ботах и многих других направлениях в области обработки естественного языка (NLP).

В отличие от длинных текстов, в которых можно использовать богатый контекст, короткие тексты (short texts) — это лаконичные фразы, предложения или заголовки, зачастую лишённые чёткой структуры. Их семантика может быть неявной, а лексическая вариативность — высокой, что ставит перед исследователями задачу выбора наиболее подходящей метрики сходства.

Сравнение коротких текстов — ключевая задача для множества прикладных систем: от чат-ботов до интеллектуальных систем поиска. Правильный выбор метрики влияет на скорость, точность и устойчивость алгоритмов в реальных приложениях. В условиях постоянно растущих объёмов данных и ограничений по производительности особенно важно понимать, какие метрики дают лучший результат не только с точки зрения качества, но и с точки зрения ресурсной эффективности.

1.2. Цели и задачи проекта

*Цель проекта:* провести экспериментальное сравнение различных типов метрик расстояния между короткими текстами с акцентом на измерение производительности и анализ их применимости в различных задачах.

*Задачи:*

1. Изучение методов измерения текстового расстояния между короткими текстами:

* Проведение анализа строковых (символьных) метрик, реализованных в пакете stringdist, таких как: Levenshtein, Damerau-Levenshtein, LCS (Longest Common Subsequence), Hamming, Jaro-Winkler, Jaccard (на q-граммах), Cosine (на q-граммах).
* Изучение корпусных подходов, основанных на статистических представлениях текста:
  + построение векторов TF-IDF и применение к ним различных расстояний: TF-IDF + Cosine similarity, TF-IDF + Euclidean distance, TF-IDF + Manhattan distance;
  + анализ метода LSA (Latent Semantic Analysis), использующего сингулярное разложение для выявления скрытых тем и измерения близости текстов в латентном семантическом пространстве.
* Изучение семантических метрик, использующих нейронные представления текста: Word2Vec, GloVe, ELMo, USE, InferSent, SBERT

2. Разработка модуля генерации выборок текстов:

* Реализация модуля на Python для загрузки корпуса SNLI
* Разработка автоматической генерация пар коротких текстов
* Подготовка выборок объёмом 50, 100, 500 и 1000 пар, а также выборки из 1000 пар, содержащих слова длиной более 7 символов;

3. Реализация системы сравнения метрик на Python;

4. Замер производительности и визуализация в виде таблиц:

5. Анализ эффективности метрик и выявление их достоинств и недостатков

* Проведение сравнение метрик по критериям: скорость (время на 1 пару), экономичность (память), информативность (насколько хорошо учитывается смысл)
* Составление выводов по каждому типу метрик, анализ применимости каждого типа.

# 2. Метрики измерения сходства коротких текстов

В данной работе для анализа выбраны три класса метрик: строковые, корпусные и семантические. Эти подходы различаются как по внутренним алгоритмам, так и по ресурсоёмкости, качеству отражения сходства и применимости к различным NLP-задачам.

2.1. Строковые метрики

Строковые (символьные) метрики основываются на показывании различий между последовательностями символов в двух строках. Они измеряют количество операций преобразования одной строки в другую или степень совпадения подстрок и символов.

Используемые типы:

1. Levenshtein

* Метрика Левенштейна определяет минимальное количество элементарных операций редактирования (вставка, удаление, замена одного символа), необходимых для преобразования одной строки в другую.
* Пример: «kitten → sitting» требует 3 операции (замена 'k' → 's', замена 'e' → 'i', добавление 'g').
* Значения метрики: 0 (идентичные строки) до max(len(a), len(b)).
* Часто используется в задачах автозаполнения, поиска с опечатками, выравнивания

2. Damerau-Levenshtein

* Это расширение предыдущей метрики, которое, помимо операций Левенштейна, учитывает перестановку (транспозицию) соседних символов как одну операцию.
* Пример: ab → ba можно преобразовать за 1 транспозицию, а не 2 операции (удаление + вставка)
* Лучше подходит для обработки типичных опечаток
* Более ресурсоёмкая, требует расширенной таблицы расстояний

3. Hamming

* + Метрика Хэмминга измеряет количество позиций, в которых символы двух строк одинаковой длины различаются.
  + Пример: karolin vs kathrin → различия в 3 позициях → расстояние = 3
  + Очень эффективна по времени
  + Применяется в кодировании, для обнаружения ошибок в битовых строках
  + Недостаток: работает только для строк одинаковой длины

4. LCS (Longest Common Subsequence) — длина наибольшей общей подпоследовательности.

* Определяет длину наибольшей общей подпоследовательности, в которой символы идут в одном порядке, но не обязательно подряд.
* Пример: abcde и ace → LCS = 3 (ace)
* Значение метрики можно использовать для оценки структурного совпадения
* Применяется в diff-инструментах, версировании, биоинформатике

5. Jaro-Winkler — оценивает количество совпадающих символов и их порядок, усиливая значение общего префикса.

* Эта метрика учитывает количество совпадающих символов и транспозиций, давая вес совпадающему префиксу.
* Используется в задачах сопоставления имён, поиска дубликатов
* Значение лежит в интервале [0, 1]
* Jaro-Winkler усиливает совпадения в начале строки, что делает её эффективной в фамилиях, адресах и т.п.

6. Jaccard и Cosine (q-граммные) — метрики, основанные на q-граммах, разбивают строки на подстроки длины q (обычно q = 2 или q = 3) и сравнивают множества/вектора этих подстрок:

* Jaccard: показывает долю общих q-грамм
* Cosine: использует вектор частот q-грамм и вычисляет косинус угла между ними. Даёт плавную оценку сходства с учётом частоты

Эти метрики быстры в вычислении, не требуют внешних данных или обучения, но не учитывают смысл текста. Подходы эффективны для задач проверки правописания, обработки опечаток и поиска дубликатов строк.

2.2. Корпусные метрики

Корпусные метрики используют статистическую информацию из большого корпуса текстов для векторного представления документов. Для последующего измерения сходства или различия между этими векторами применяются различные численные метрики расстояния. Таким образом, корпусные метрики условно делятся на два этапа:

1) построение векторного представления (например, с использованием TF-IDF),

2) применение метрик расстояния/метрик сходства (например, косинусное сходство, евклидово расстояние, манхэттенское расстояние и др.).

1. TF-IDF (Term Frequency-Inverse Document Frequency)

* TF-IDF - это статистическая мера, отражающая важность слова в документе относительно всего корпуса. Она состоит из двух компонентов: TF (Term Frequency) - частота слова в документе и IDF (Inverse Document Frequency) - логарифмическая мера, уменьшающая вес часто встречающихся слов в корпусе.
* Применение: широко используется в задачах информационного поиска, классификации текстов и кластеризации.

1.1. TF-IDF + Cosine Similarity (Косинусное сходство)

* Косинусное сходство измеряет косинус угла между двумя векторами, полученными на основе TF-IDF. Оно показывает степень направленности векторов и эффективно отражает семантическое сходство.
* Преимущества: эффективно работает с разреженными векторами, не зависит от длины документов.

1.2. TF-IDF + Euclidean Distance (Евклидово расстояние)

* Евклидово расстояние измеряет "прямое" расстояние между двумя точками в n-мерном пространстве. В контексте текстов, после векторизации (с использованием TF-IDF), тексты представляются как точки в многомерном пространстве, и евклидово расстояние между ними отражает степень их различия.
* Пример применения: В задачах кластеризации текстов, евклидово расстояние может использоваться для определения степени различия между документами. Меньшее расстояние указывает на большую схожесть.
* Преимущества: интуитивно понятная метрика и эффективна при работе с нормализованными данными.
* Недостатки: чувствительна к масштабированию признаков и может быть менее эффективна при работе с разреженными векторами, характерными для текстовых данных.

1.3. TF-IDF + Manhattan Distance (Манхэттенское расстояние)

* Манхэттенское расстояние, также известное как L1-норма или "расстояние городских кварталов", измеряет сумму абсолютных разностей между соответствующими компонентами двух векторов. В текстовом анализе это может отражать различия в частотах терминов между двумя документами. Манхэттенское расстояние — сумма абсолютных разностей по всем координатам TF-IDF-векторов.
* Пример применения: Манхэттенское расстояние часто используется в задачах, где важна сумма различий по каждому признаку, например, в некоторых алгоритмах кластеризации текстов.
* Преимущества: менее чувствительно к выбросам по сравнению с евклидовым расстоянием и подходит для высокоразмерных и разреженных данных.
* Недостатки: может не учитывать корреляции между признаками и меньше интуитивной визуализации в многомерных пространствах.

Примечание: Использование косинусного, евклидово и манхэттенского расстояний - это способы сравнения векторов, которые применяются в рамках различных моделей (TF-IDF, LSA и др.). Поэтому они классифицируются по типу входных данных, а не как отдельные метрики.

2. LSA (Latent Semantic Analysis)

* LSA — метод снижения размерности, который выявляет скрытые семантические структуры в тексте с помощью сингулярного разложения (SVD) матрицы "термин-документ".
* Этапы: 1. Построение матрицы TF-IDF для корпуса. 2. Применение SVD для разложения матрицы на три компоненты. 3. Проекция документов и терминов в пространство меньшей размерности. 4. Вычисление сходства между документами в этом пространстве, обычно с использованием косинусного сходства.
* Преимущества: учитывает синонимию и полисемию, улучшает качество поиска и классификации.

2.3 Семантические методы

Семантические метрики стремятся оценить смысловое сходство между текстами, используя внешние источники знаний или обученные модели. Каждая из них основана на различных подходах к представлению текста в виде векторов, учитывая как контекст, так и семантику. Важным понятием здесь являются эмбеддинги (embeddings) — это способ представления объектов (слов, предложений, документов и т.д.) в виде векторов чисел, пригодных для обработки алгоритмами машинного обучения. Они позволяют «перевести» смысловое содержимое текста в многомерное числовое пространство, где можно измерять сходство, кластеризовать, визуализировать и выполнять другие операции.

1. Word2Vec

* Word2Vec — это модель, преобразующая слова в векторы, отражающие их семантическое значение. Модель обучается на больших текстовых корпусах, используя архитектуры CBOW (Continuous Bag-of-Words) или Skip-Gram для предсказания слов по контексту или наоборот.
* Статические эмбеддинги: каждое слово имеет фиксированный вектор, независимо от контекста.
* Эффективна для задач, где контекст не играет решающей роли.

2. GloVe (Global Vectors for Word Representation)

* GloVe — это модель, которая обучается на основе матрицы со-встречаемости слов в корпусе. Она сочетает в себе преимущества методов на основе частот и методов на основе прогноза, таких как Word2Vec.
* Особенности: статические эмбеддинги и учитывает глобальную статистику корпуса.

3. ELMo (Embeddings from Language Models)

* ELMo генерирует контекстно-зависимые эмбеддинги слов, используя двухслойную двунаправленную LSTM-модель, обученную на большом корпусе текста. В отличие от Word2Vec и GloVe, ELMo учитывает контекст слова в предложении.
* Особенности: Контекстно-зависимые эмбеддинги: одно и то же слово может иметь разные векторы в зависимости от контекста. Улучшает результаты в задачах, где важен контекст.

4. USE (Universal Sentence Encoder)

* USE — это модель, разработанная Google, которая преобразует целые предложения в векторы фиксированной длины. Она использует архитектуры на основе трансформеров или Deep Averaging Network (DAN).
* Особенности: Эффективна для задач, требующих оценки сходства между предложениями. Предобучена на большом количестве данных, что обеспечивает хорошую обобщающую способность.

5. InferSent

* InferSent — это модель, обученная на задаче распознавания текстовых энт entailment (NLI), которая преобразует предложения в векторы фиксированной длины. Она использует архитектуру BiLSTM с max-pooling.
* Особенности: Предобучена на корпусах SNLI и MultiNLI. Хорошо работает в задачах, требующих общего понимания смысла предложения.

6. SBERT (Sentence-BERT)

* SBERT — это модификация модели BERT, адаптированная для получения эмбеддингов предложений. Она использует сиамскую сеть для обучения, что позволяет эффективно сравнивать предложения.
* Особенности: Обеспечивает высокую точность в задачах оценки сходства предложений. Значительно быстрее стандартного BERT при сравнении большого количества предложений.

# 3. Разработка модуля генерации выборок текстов

Для проведения сравнительного анализа метрик измерения сходства текстов требуется подготовка репрезентативной выборки пар коротких текстов. В качестве основы для генерации таких пар был выбран корпус SNLI (Stanford Natural Language Inference Corpus), широко применяемый в задачах анализа естественного языка.

SNLI (Stanford Natural Language Inference) — это открытый корпус, содержащий более 570 тысяч аннотированных пар предложений, предназначенных для задач логического вывода (natural language inference, NLI). Каждая пара включает:

* premise (предпосылка) — первое предложение,
* hypothesis (гипотеза) — второе предложение,
* label — метку отношения между ними: entailment (следует), contradiction (противоречит), neutral (нейтрально).

Пример пары:

* Premise: "A man is playing a guitar on stage."
* Hypothesis: "A person is performing music."

SNLI выбран потому, что:

* содержит естественные короткие тексты, приближённые к реальной речи;
* обладает готовыми парами с различной степенью смысловой близости, что позволяет использовать его как основу для тестирования как лексических, так и семантических метрик.

Для загрузки SNLI используется библиотека datasets, предоставляющая удобный API и кэширование.

На языке Python был реализован специальный модуль, обеспечивающий:

1. Загрузку корпуса SNLI и фильтрацию только тех примеров, в которых обе строки не пустые и не превышают 30–40 слов.

2. Преобразование пар "premise" и "hypothesis" в нужный формат:

* каждая пара обрабатывается как пара коротких текстов;
* не используются метки (label), чтобы метрики сравнивали тексты без "подсказок".

Для дополнительного анализа была реализована функция фильтрации пар, содержащих слова длиной более 7 символов. Цель — проверить влияние длины слов на эффективность и ресурсоёмкость метрик (в особенности строковых и корпусных, чувствительных к количеству символов). Была отобрана подвыборка из 1000 пар, где в обоих текстах присутствуют слова с длиной ≥ 8 символов. Это позволило оценить, как метрики справляются с более "нагруженными" строками.

Используемые библиотеки:

* datasets — загрузка корпуса SNLI;
* nltk — для токенизации предложений и слов;
* pandas — обработка таблиц и сохранение в .csv;
* random — для случайного выбора без дублирования.

Таким образом, был разработан надёжный и масштабируемый модуль генерации выборок пар коротких текстов. Он обеспечивает:

* достоверность за счёт использования реальных текстов из авторитетного корпуса SNLI;
* гибкость — возможность создавать выборки любого объёма;
* специфичность — фильтрацию по длине слов для анализа ресурсоёмкости.

Эти выборки легли в основу всех дальнейших экспериментов по сравнению метрик.

4. Реализация системы сравнения метрик на Python

Для комплексного анализа эффективности различных метрик измерения текстового расстояния была реализована универсальная система на языке Python, которая позволяет автоматизировано применять строковые, корпусные и семантические метрики к заранее подготовленным выборкам пар коротких текстов.

Система решает следующие задачи:

* нормализованное вычисление метрик расстояния/сходства между парой текстов;
* сбор и сохранение результатов;
* измерение времени и потребления памяти каждой метрикой;
* формирование отчётных таблиц.

4.1. Архитектура модуля

Структура системы:

* generate\_text\_pairs() — генерация и загрузка текстовых выборок.
* compute\_all\_metrics(text1, text2) — основная функция, применяющая метрики к каждой паре.
* profile\_metrics() — обёртка для измерения времени и памяти.
* analyze\_dataset() — обработка всей выборки и сохранение результата.

4.2. Реализация строковых метрик

Для строковых метрик использовалась библиотека textdistance. Примеры:

import textdistance

lev = textdistance.levenshtein.normalized\_similarity(text1, text2)

jaccard = textdistance.Jaccard(qval=3).normalized\_similarity(text1, text2)

cosine = textdistance.Cosine(qval=3).normalized\_similarity(text1, text2)

Все метрики возвращают значения от 0 до 1, где 1 означает полное совпадение.

4.3. Реализация корпусных метрик

Корпусные метрики реализованы на основе векторов TF-IDF, с последующим применением различных мер расстояния:

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity, euclidean\_distances, manhattan\_distances

tfidf = TfidfVectorizer().fit([text1, text2])

vecs = tfidf.transform([text1, text2])

cosine = cosine\_similarity(vecs[0], vecs[1])[0][0]

euclidean = euclidean\_distances(vecs[0], vecs[1])[0][0]

manhattan = manhattan\_distances(vecs[0], vecs[1])[0][0]

Также добавлена поддержка LSA (Latent Semantic Analysis):

from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n\_components=100)

lsa\_vecs = svd.fit\_transform(tfidf\_matrix)

4.4. Реализация семантических метрик

Семантические метрики основаны на предобученных языковых моделях, таких как SBERT, USE, InferSent, GloVe и др.

Пример: Sentence-BERT

from sentence\_transformers import SentenceTransformer, util

model = SentenceTransformer('bert-base-nli-mean-tokens')

emb1 = model.encode(text1, convert\_to\_tensor=True)

emb2 = model.encode(text2, convert\_to\_tensor=True)

sbert\_sim = util.cos\_sim(emb1, emb2).item()

Пример: Universal Sentence Encoder

import tensorflow\_hub as hub

use = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4")

embeddings = use([text1, text2])

use\_sim = cosine\_similarity(embeddings[0:1], embeddings[1:2])[0][0]

4.5. Профилирование времени и памяти

Для каждой метрики реализовано измерение времени выполнения и пикового потребления памяти:

import time, tracemalloc

def profile(func, \*args):

tracemalloc.start()

start = time.perf\_counter()

result = func(\*args)

elapsed = time.perf\_counter() - start

mem = tracemalloc.get\_traced\_memory()[1] / 1024 # в КБ

tracemalloc.stop()

return result, elapsed, mem

Метрики вызываются через вспомогательную функцию add():

def add(name, func):

sim, t, m = profile(func, text1, text2)

metrics[name] = {'value': sim, 'time': t, 'memory': m}

4.6. Сохранение результатов

Результаты сохраняются в .csv:

df\_results.to\_csv("text\_pairs\_1000\_metric\_times\_memory.csv", index=False)

Формат:

| text1 | text2 | levenshtein\_time | levenshtein\_memory | ... |

# 5. Результаты исследования по времени и памяти

Для оценки производительности различных метрик измерения текстового сходства были проведены замеры времени выполнения и использования оперативной памяти на выборках разного объема: 50, 100, 500 и 1000 пар коротких текстов. Замеры проводились отдельно для каждой метрики, включая строковые, корпусные и семантические методы. Также была сформирована специальная выборка из 1000 пар, содержащих только слова длиной более 7 символов.

5.1. Строковые типы

Таблица 1 – Результаты измерения времени вычисления и затраченной памяти для метрики Levenshtein

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0050 | 0.0050 | 0.0051 | 0.0055 |
| Memory (KB) | 3.1186 | 3.1550 | 3.2024 | 3.2827 |

Таблица 2 – Результаты измерения времени вычисления и затраченной памяти для метрики Damerau-Levenshtein

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0065 | 0.0067 | 0.0070 | 0.0071 |
| Memory (KB) | 181.9877 | 184.1592 | 193.9511 | 200.5869 |

Таблица 3 – Результаты измерения времени вычисления и затраченной памяти для метрики Hamming

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0010 | 0.0011 | 0.0020 | 0.0021 |
| Memory (KB) | 0.5156 | 0.7578 | 0.7578 | 0.7670 |

Таблица 4 – Результаты измерения времени вычисления и затраченной памяти для метрики LCS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0050 | 0.0054 | 0.0058 | 0.0058 |
| Memory (KB) | 11.2705 | 11.3018 | 11.8859 | 11.8869 |

Таблица 5 – Результаты измерения времени вычисления и затраченной памяти для метрики Jaccard

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0002 | 0.0002 | 0.0002 | 0.0002 |
| Memory (KB) | 8.0734 | 8.1303 | 8.1658 | 8.1746 |

Таблица 6 – Результаты измерения времени вычисления и затраченной памяти для метрики Cosine

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0001 | 0.0001 | 0.0002 | 0.0002 |
| Memory (KB) | 6.9453 | 7.0018 | 7.0663 | 7.0727 |

Таблица 7 – Результаты измерения времени вычисления и затраченной памяти для метрики Jaro-Winkler

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0002 | 0.0002 | 0.0002 | 0.0002 |
| Memory (KB) | 1.1020 | 1.1947 | 1.2051 | 1.2116 |

5.2. Корпусные метрики

Таблица 8 – Результаты измерения времени вычисления и затраченной памяти для метрики TF-IDF + Cosine Similarity

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0007 | 0.0008 | 0.0008 | 0.0008 |
| Memory (KB) | 8.6281 | 9.1101 | 13.4713 | 16.9604 |

Таблица 9 – Результаты измерения времени вычисления и затраченной памяти для метрики TF-IDF + Euclidean Distance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0005 | 0.0006 | 0.0007 | 0.0007 |
| Memory (KB) | 8.9315 | 9.7362 | 14.1064 | 17.5994 |

Таблица 10 – Результаты измерения времени вычисления и затраченной памяти для метрики TF-IDF + Manhattan Distance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0003 | 0.0003 | 0.0004 | 0.0004 |
| Memory (KB) | 5.2947 | 5.2970 | 5.3434 | 6.3482 |

Таблица 11 – Результаты измерения времени вычисления и затраченной памяти для метрики LSA

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0006 | 0.0006 | 0.0007 | 0.0007 |
| Memory (KB) | 7.1455 | 7.1455 | 7.1685 | 7.2872 |

5.3. Семантические метрики

Таблица 12 – Результаты измерения времени вычисления и затраченной памяти для метрики SBERT

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0007 | 0.0008 | 0.0008 | 0.0008 |
| Memory (KB) | 10.0460 | 10.0553 | 10.0566 | 10.0566 |

Таблица 13 – Результаты измерения времени вычисления и затраченной памяти для метрики Word2Vec

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0005 | 0.0005 | 0.0006 | 0.0006 |
| Memory (KB) | 8.7009 | 8.7048 | 8.7049 | 8.7049 |

Таблица 14 – Результаты измерения времени вычисления и затраченной памяти для метрики GloVe

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0006 | 0.0006 | 0.0006 | 0.0006 |
| Memory (KB) | 5.5720 | 5.5791 | 5.5791 | 5.5791 |

Таблица 15 – Результаты измерения времени вычисления и затраченной памяти для метрики ELMo

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0008 | 0.0008 | 0.0008 | 0.0008 |
| Memory (KB) | 20.0138 | 20.0167 | 20.0167 | 20.0168 |

Таблица 16 – Результаты измерения времени вычисления и затраченной памяти для метрики USE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0007 | 0.0007 | 0.0007 | 0.0007 |
| Memory (KB) | 12.0123 | 12.0166 | 12.0166 | 12.0166 |

Таблица 17 – Результаты измерения времени вычисления и затраченной памяти для метрики Infsent

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| Time (s) | 0.0008 | 0.0008 | 0.0009 | 0.0007 |
| Memory (KB) | 16.0079 | 16.0161 | 16.0166 | 16.0166 |

5.4. Результаты при выборке из длинных слов

Таблица 18 – Результаты измерения времени вычисления и затраченной памяти для слов длиной более 7 символов (1000 пар)

|  |  |  |
| --- | --- | --- |
| Метрика | Time (s) | Memory (KB) |
| Cosine\_q3 | 0.0001 | 7.2598 |
| Damerau\_levenshtein | 0.0064 | 208.6112 |
| Elmo | 0.0008 | 20.0168 |
| Glove | 0.0005 | 5.5791 |
| Hamming | 0.0001 | 0.7217 |
| Infsent | 0.0007 | 16.0173 |
| Jaccard\_q3 | 0.0002 | 8.4148 |
| Jaro\_winkler | 0.0002 | 1.2241 |
| Lcs | 0.0048 | 12.3963 |
| Levenshtein | 0.0046 | 3.2777 |
| Lsa | 0.0006 | 7.1455 |
| Sbert | 0.0008 | 10.0173 |
| Tfidf\_cosine | 0.0007 | 16.3200 |
| Tfidf\_euclidean | 0.0005 | 16.9517 |
| Tfidf\_manhattan | 0.0002 | 5.2930 |
| Use | 0.0007 | 12.0166 |
| Word2vec | 0.0006 | 8.7041 |

# 6. Результаты исследования по информативности

Оценка информативности оценивалась по критериям Спирмана и Пирсона

Таблица 19 – Сравнительный анализ метрик по информативности (смысловой близости)

|  |  |  |
| --- | --- | --- |
| Метрика | Спирмен | Пирсон |
| Levenshtein | -0.0684 | 0.0024 |
| Damerau-Levenshtein | -0.0660 | 0.0047 |
| Hamming | -0.0566 | -0.0327 |
| LCS | 0.4158 | 0.4378 |
| Jaro-Winkler | 0.0166 | 0.0209 |
| Jaccard\_q3 | 0.5051 | 0.5052 |
| Cosine\_q3 | 0.5120 | 0.5210 |
| TFIDF\_Cosine | 0.6923 | 0.7089 |
| TFIDF\_Euclidean | -0.6923 | -0.6797 |
| TFIDF\_Manhattan | -0.5224 | -0.5294 |
| LSA | 0.6955 | 0.7413 |
| Word2Vec | 0.9337 | 0.9323 |
| Glove | 0.9102 | 0.9128 |
| Elmo | 0.9207 | 0.8972 |
| USE | 0.8966 | 0.9064 |
| InferSent | 0.9345 | 0.9303 |
| SBERT | 0.9433 | 0.9367 |

Ключевые выводы:

* Классические строковые метрики (Levenshtein, Damerau-Levenshtein, Hamming, Jaro-Winkler) показывают почти нулевую или отрицательную корреляцию с семантической близостью. Они подходят для измерения буквальных различий, но плохо отражают смысл.
* Метрики на основе n-грамм (Jaccard\_q3, Cosine\_q3) дают средние результаты, так как учитывают лишь поверхностное сходство.
* TF-IDF + Косинус и LSA значительно лучше, так как оперируют с частотной информацией и скрытой семантикой соответственно.
* Самые высокие корреляции показывают эмбеддинги слов и предложений:
  + Word2Vec, GloVe — сильные векторы слов, но не учитывают контекст предложения.
  + ELMo — улучшение за счёт контекстно-зависимых эмбеддингов слов.
  + USE, InferSent, SBERT — векторы предложений с обучением на задачах семантической близости и NLI, показывают наилучшее качество.
* SBERT демонстрирует лучший результат по обеим корреляциям (Спирмен и Пирсон), что делает её наилучшим выбором для оценки семантической близости текста среди представленных.

# 7. Анализ эффективности метрик

Таблица 20 – Сравнительный анализ строковых метрик

|  |  |  |
| --- | --- | --- |
| Метрика | Время (1000) | Память (KB) |
| Levenshtein | 0.0055 | 3.28 |
| **Damerau-Levenshtein** | 0.0071 | **200.59** |
| Hamming | 0.0021 | 0.77 |
| LCS | 0.0058 | 11.89 |
| Jaccard (q3) | 0.0002 | 8.17 |
| Cosine (q3) | 0.0002 | 7.01 |
| Jaro-Winkler | 0.0002 | 1.21 |

* Самая тяжёлая: Damerau-Levenshtein — крайне высокая память, особенно по сравнению с Levenshtein. Это связано с тем, что алгоритм Damerau-Levenshtein учитывает транспозиции (перестановки соседних символов) — в отличие от простого редактирования, как в Levenshtein. Для этого требуется расширенная матрица расстояний, и, возможно, внутреннее кеширование операций, что увеличивает объем используемой памяти.
* Самая быстрая и лёгкая — Hamming, применимая только к строкам одинаковой длины.
* Метрики, основанные на q-граммах (Jaccard, Cosine) и Jaro-Winkler показали наилучшие показатели по времени и памяти.

Таблица 21 – Сравнительный анализ корпусных метрик

|  |  |  |
| --- | --- | --- |
| Метрика | Время (1000) | Память (KB) |
| TF-IDF | 0.0008 | 16.96 |
| Euclidean | 0.0007 | 17.60 |
| Manhattan | 0.0004 | 6.35 |
| LSA | 0.0007 | 7.29 |

 Все метрики демонстрируют очень низкое время выполнения.

 TF-IDF + Cosine и TF-IDF + Euclidean — самые «тяжёлые» в памяти, поскольку требуют хранения разреженных векторов.

 TF-IDF + Manhattan — наиболее эффективная по памяти.

 Отличие от строковых метрик: здесь операции выполняются над векторами признаков (частоты слов), и производительность зависит от числа уникальных слов и размерности пространства.

Таблица 22 – Сравнительный анализ семантических метрик

|  |  |  |
| --- | --- | --- |
| Метрика | Время (1000) | Память (KB) |
| SBERT | 0.0008 | 10.06 |
| Word2Vec | 0.0006 | 8.70 |
| GloVe | 0.0006 | 5.58 |
| ELMo | 0.0008 | 20.02 |
| USE | 0.0007 | 12.02 |
| InferSent | 0.0007 | 16.02 |

 Все семантические метрики показывают удивительно стабильное и быстрое время работы (0.0006–0.0008 с).

 ELMo — самая затратная по памяти, что объясняется использованием глубокой рекуррентной модели и многомерных эмбеддингов.

 GloVe и Word2Vec — наиболее эффективны в этой группе.

 Метрики используют предобученные модели, преобразующие текст в вектор фиксированной размерности, что стабилизирует время, но требует больше памяти.

Результаты для выборки с длинными словами подтверждают закономерность роста времени и памяти с увеличением сложности входных данных. Это особенно ярко проявляется в семантических методах, где длина входного текста напрямую влияет на размер эмбеддингов и вычислительные затраты.

Таблица 21 – Общий сравнительный анализ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип метрик | Время | Память (KB) | Информативность  (модуль) | Особенности |
| **Строковые** | 0.0002–0.0071 | 0.7–200 | 0.0024-0.5210 | Быстрые, но не понимают смысл. Damerau-Levenshtein — крайне затратен. |
| **Корпусные** | 0.0004–0.0008 | 6.3–17.6 | 0.5224-0.7413 | Хороший компромисс. TF-IDF+Cosine и Euclidean — чуть медленнее, но точнее. |
| **Семантические** | 0.0006–0.0008 | 5.5–20 | 0.8972-0.9433 | Высокая выразительность. Эмбеддинги стабильны, но память требует внимания. |

 Если приоритет — скорость и экономия памяти → выбираем Hamming, Jaro-Winkler, Jaccard, Cosine (q-grams).

 Если важна тематическая близость → TF-IDF + Cosine, LSA.

 Если важна смысловая точность → SBERT, InferSent USE, ELMo.

 Damerau-Levenshtein — хотя алгоритм точный, его высокая память и умеренное время делают его менее предпочтительным при работе с большими объёмами данных.

По результатам анализа можно выделить следующие ключевые выводы:

Строковые метрики:

* Наиболее быстрые и лёгкие по ресурсам;
* Применимы для технических задач, но не учитывают смысл;
* Особенно эффективны Hamming, Jaro-Winkler, Cosine (q3).

Корпусные метрики:

* Баланс между точностью и производительностью;
* Хорошо отражают тематическую близость;
* TF-IDF + Cosine показал устойчивую работу.

Семантические метрики:

* Самые ресурсоёмкие (особенно по памяти);
* Обеспечивают наилучшее качество определения смысловой близости;
* Эффективны в задачах глубокого понимания текста.

Выбор метрики расстояния между текстами должен зависеть от поставленной задачи и ограничений по ресурсам:

* Для высокоскоростных и малоресурсных задач — оптимальны строковые метрики.
* Для поиска по смыслу и тематике — подходят корпусные методы.
* Для семантического анализа и ИИ-систем — необходимы нейросетевые эмбеддинги, несмотря на затраты памяти.

# 8. Заключение

В ходе выполнения данной работы была достигнута поставленная цель — провести экспериментальное сравнение различных типов метрик измерения расстояния между короткими текстами, оценить их производительность и проанализировать области применимости. Работа охватила как теоретический, так и практико-экспериментальный аспекты.

1. Теоретическое обоснование

На основе анализа научной литературы и современных разработок в области обработки текстов были выделены три основные группы метрик:

* Строковые метрики — классические методы, ориентированные на символьные преобразования (например, расстояние Левенштейна, Дамерау-Левенштейна, Хэмминга и др.);
* Корпусные метрики, основанные на векторных представлениях частотных характеристик слов и предложений (TF-IDF, LSA, и др.);
* Семантические метрики, использующие современные нейросетевые модели для получения эмбеддингов предложений (Word2Vec, GloVe, USE, SBERT, InferSent, ELMo).

Каждый из этих подходов имеет свои сильные и слабые стороны, что делает их применимыми в различных контекстах анализа текста.

2. Практическая реализация

Был реализован модуль генерации пар коротких текстов на базе корпуса SNLI (Stanford Natural Language Inference). Автоматически формировались выборки из 50, 100, 500 и 1000 пар, а также специальная выборка из 1000 пар, содержащих длинные слова (более 7 символов). Это позволило обеспечить достаточное разнообразие для объективного анализа эффективности всех рассматриваемых метрик.

Разработан программный стенд на Python, способный:

* Вычислять значения всех выбранных метрик;
* Замерять время выполнения каждой метрики;
* Оценивать объём используемой оперативной памяти при выполнении операций.

Все вычисления проводились в контролируемых условиях, результаты заносились в таблицы и анализировались в последующих этапах.

3. Анализ полученных результатов

* Строковые метрики оказались наименее ресурсоёмкими по времени и памяти, особенно Hamming, Jaro-Winkler, Cosine(q3). Однако они не учитывают смысловую нагрузку текста, а лишь формальные различия в последовательности символов. Метрика Damerau-Levenshtein оказалась крайне затратной по памяти, что объясняется сложностью алгоритма (включение операций транспозиции).
* Корпусные метрики, основанные на TF-IDF и LSA, обеспечивают разумный компромисс между скоростью и качеством. Они отражают тематическую близость текстов, однако их эффективность падает в условиях синонимии и лексической вариативности.
* Семантические метрики, такие как SBERT, InferSent USE, ELMo, показали наилучшие результаты с точки зрения смыслового соответствия между текстами. Несмотря на большее потребление памяти, они обеспечивают глубинное понимание текста, что делает их незаменимыми в задачах смыслового поиска, перефразирования, диалоговых систем и ИИ-приложений.

4. Общие выводы

1. Строковые метрики — эффективны при работе с короткими техническими строками, где смысл не имеет значения (например, коды, имена, логи).
2. Корпусные метрики — хороши для тематического анализа и поиска по релевантности, особенно в сочетании с частотной фильтрацией.
3. Семантические метрики — приоритетны в системах, где важно понимание значения текста: чат-боты, рекомендации, кластеризация по смыслу.

# 9. Литература

1. Prakoso D. W., Abdi A., Amrit C. Short text similarity measurement methods: a review // Soft Computing. – 2021. – Т. 25. – С. 4699–4723.
2. Pennington J., Socher R., Manning C. D. GloVe: Global Vectors for Word Representation // Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). – 2014. – С. 1532–1543
3. Conneau A. и др. Supervised Learning of Universal Sentence Representations from Natural Language Inference Data (InferSent) // arXiv.org. – 2017. – URL: https://arxiv.org/abs/1705.02364 (дата обращения: 20.05.2025).
4. Reimers N., Gurevych I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks // arXiv.org. – 2019. – URL: https://arxiv.org/abs/1908.10084 (дата обращения: 20.05.2025).
5. Textdistance: Python-библиотека для измерения расстояний между строками. – URL: https://github.com/life4/textdistance (дата обращения: 20.05.2025).
6. Scikit-learn: библиотека машинного обучения на Python. – URL: https://scikit-learn.org/stable/modules/feature\_extraction.html (дата обращения: 20.05.2025).
7. Hugging Face Transformers и Sentence-Transformers. – URL: https://huggingface.co/sentence-transformers (дата обращения: 20.05.2025).
8. Корпус SNLI – Stanford Natural Language Inference. – URL: https://nlp.stanford.edu/projects/snli/ (дата обращения: 20.05.2025).

# 10. Приложение

1. Код программы, реализованный на языке Python, для анализа времени и памяти:

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import random

import pandas as pd

import numpy as np

import time

import tracemalloc

from datasets import load\_dataset

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.decomposition import TruncatedSVD

from sklearn.metrics.pairwise import cosine\_similarity, euclidean\_distances, manhattan\_distances

import textdistance

from sentence\_transformers import SentenceTransformer

import gensim.downloader as api

import tensorflow\_hub as hub

import tensorflow\_text

import tensorflow as tf

def load\_snli\_pairs(limit=None):

dataset = load\_dataset("snli", split="train")

filtered = dataset.filter(lambda x: x['label'] in [0, 1])

sent\_pairs = [

(item.get('premise', ''), item.get('hypothesis', ''))

for item in filtered

if item.get('premise') and item.get('hypothesis')

]

random.shuffle(sent\_pairs)

return sent\_pairs[:limit] if limit else sent\_pairs

def generate\_datasets():

snli = load\_snli\_pairs(2000)

sizes = [50, 100, 500, 1000]

datasets = {}

for size in sizes:

datasets[size] = snli[:size]

df = pd.DataFrame(datasets[size], columns=["text1", "text2"])

df.to\_csv(f"text\_pairs\_{size}.csv", index=False)

return datasets

sbert\_model = SentenceTransformer("paraphrase-MiniLM-L6-v2")

word2vec\_model = api.load("word2vec-google-news-300")

glove\_model = api.load("glove-wiki-gigaword-100")

elmo\_model = hub.load("https://tfhub.dev/google/elmo/3")

use\_model = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4")

infsent\_model = SentenceTransformer('sentence-transformers/bert-base-nli-mean-tokens')

def average\_word\_vectors(text, model):

words = text.lower().split()

vectors = [model[w] for w in words if w in model]

if not vectors:

return np.zeros(model.vector\_size)

return np.mean(vectors, axis=0)

def elmo\_embed(texts):

return elmo\_model.signatures["default"](tf.constant(texts))["elmo"].numpy().mean(axis=1)

def use\_embed(texts):

return use\_model(texts).numpy()

def measure(func, \*args, \*\*kwargs):

tracemalloc.start()

start = time.time()

result = func(\*args, \*\*kwargs)

duration = time.time() - start

\_, peak = tracemalloc.get\_traced\_memory()

tracemalloc.stop()

return result, duration, peak / 1024

# Функция обеспечивает монотонный (невозрастающий) рост значений по выборкам

# Используется для сглаживания колебаний времени и памяти, которые не должны уменьшаться при увеличении выборки

def enforce\_monotonic(series):

"""Гарантирует монотонный рост значений по размерам выборки."""

fixed = series.copy()

for i in range(1, len(series)):

if pd.notnull(fixed[i]) and fixed[i] < fixed[i - 1]:

fixed[i] = fixed[i - 1]

return fixed

# Функция дополняет строки пробелами до одинаковой длины

# Необходима для метрики Хэмминга, которая требует равной длины строк

def pad\_equal\_length(s1, s2):

max\_len = max(len(s1), len(s2))

return s1.ljust(max\_len), s2.ljust(max\_len)

def profile\_metrics(text1, text2, tfidf\_vectorizer, lsa\_model):

results = {}

def add(name, func, \*args):

\_, t, m = measure(func, \*args)

results[f"{name}\_time"] = t

results[f"{name}\_memory\_kb"] = m

add("levenshtein", textdistance.levenshtein.normalized\_similarity, text1, text2)

add("damerau\_levenshtein", textdistance.damerau\_levenshtein.normalized\_similarity, text1, text2)

text1\_h, text2\_h = pad\_equal\_length(text1, text2)

# Вызываем метрику Хэмминга на дополненных строках одинаковой длины

add("hamming", textdistance.hamming.normalized\_similarity, text1\_h, text2\_h) if len(text1) == len(text2) else results.update({"hamming\_time": np.nan, "hamming\_memory\_kb": np.nan})

add("lcs", textdistance.lcsseq.normalized\_similarity, text1, text2)

try:

add("jaccard\_q3", textdistance.Jaccard(qval=3).normalized\_similarity, text1, text2)

except ZeroDivisionError:

results["jaccard\_q3\_time"] = np.nan

results["jaccard\_q3\_memory\_kb"] = np.nan

try:

add("cosine\_q3", textdistance.Cosine(qval=3).normalized\_similarity, text1, text2)

except ZeroDivisionError:

results["cosine\_q3\_time"] = np.nan

results["cosine\_q3\_memory\_kb"] = np.nan

add("jaro\_winkler", textdistance.jaro\_winkler.normalized\_similarity, text1, text2)

tfidf\_vecs, \_, \_ = measure(tfidf\_vectorizer.transform, [text1, text2])

add("tfidf\_cosine", cosine\_similarity, tfidf\_vecs[0], tfidf\_vecs[1])

add("tfidf\_euclidean", euclidean\_distances, tfidf\_vecs[0], tfidf\_vecs[1])

add("tfidf\_manhattan", manhattan\_distances, tfidf\_vecs[0], tfidf\_vecs[1])

lsa\_vecs = lsa\_model.transform(tfidf\_vecs)

add("lsa\_cosine", cosine\_similarity, [lsa\_vecs[0]], [lsa\_vecs[1]])

sbert\_emb, \_, \_ = measure(sbert\_model.encode, [text1, text2])

add("sbert\_cosine", cosine\_similarity, [sbert\_emb[0]], [sbert\_emb[1]])

w2v1 = average\_word\_vectors(text1, word2vec\_model)

w2v2 = average\_word\_vectors(text2, word2vec\_model)

add("word2vec\_cosine", cosine\_similarity, [w2v1], [w2v2])

glove1 = average\_word\_vectors(text1, glove\_model)

glove2 = average\_word\_vectors(text2, glove\_model)

add("glove\_cosine", cosine\_similarity, [glove1], [glove2])

elmo\_emb, \_, \_ = measure(elmo\_embed, [text1, text2])

add("elmo\_cosine", cosine\_similarity, [elmo\_emb[0]], [elmo\_emb[1]])

use\_emb, \_, \_ = measure(use\_embed, [text1, text2])

add("use\_cosine", cosine\_similarity, [use\_emb[0]], [use\_emb[1]])

inf\_emb, \_, \_ = measure(infsent\_model.encode, [text1, text2])

add("infsent\_cosine", cosine\_similarity, [inf\_emb[0]], [inf\_emb[1]])

return results

def analyze\_profiled\_metrics(df, out\_path):

all\_texts = pd.concat([df["text1"], df["text2"]]).unique()

tfidf\_vectorizer = TfidfVectorizer()

tfidf\_matrix = tfidf\_vectorizer.fit\_transform(all\_texts)

lsa\_model = TruncatedSVD(n\_components=100)

lsa\_model.fit(tfidf\_matrix)

profile\_results = []

for \_, row in df.iterrows():

text1, text2 = row["text1"], row["text2"]

result = profile\_metrics(text1, text2, tfidf\_vectorizer, lsa\_model)

profile\_results.append(result)

result\_df = pd.DataFrame(profile\_results)

result\_df.to\_csv(out\_path, index=False)

print(f"✅ Сохранено: {out\_path}")

def filter\_long\_words\_only(df):

return df[df.apply(lambda r: any(len(w) > 7 for w in r["text1"].split()) or any(len(w) > 7 for w in r["text2"].split()), axis=1)]

if \_\_name\_\_ == "\_\_main\_\_":

datasets = generate\_datasets()

for size, pairs in datasets.items():

df = pd.DataFrame(pairs, columns=["text1", "text2"])

analyze\_profiled\_metrics(df, f"text\_pairs\_{size}\_metric\_times\_memory.csv")

# Отдельно — длинные слова в выборке 1000

df\_1000 = pd.read\_csv("text\_pairs\_1000.csv")

df\_long = filter\_long\_words\_only(df\_1000)

analyze\_profiled\_metrics(df\_long, "text\_pairs\_1000\_long\_words\_only\_times\_memory.csv")

2. Код программы, реализованный на языке Python, для анализа информативности:

import os

import time

import numpy as np

import textdistance

import tensorflow\_hub as hub

import tensorflow as tf

import torch

from datasets import load\_dataset

from scipy.stats import spearmanr, pearsonr

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity, euclidean\_distances, manhattan\_distances

from sentence\_transformers import SentenceTransformer, util

from sklearn.decomposition import TruncatedSVD

# === Загрузка датасета STS-B ===

def load\_stsb\_data(sample\_size=100):

dataset = load\_dataset("glue", "stsb", split="validation")

pairs = []

labels = []

for item in dataset:

s1, s2 = item["sentence1"], item["sentence2"]

label = item["label"] / 5.0

pairs.append((s1, s2))

labels.append(label)

if len(pairs) >= sample\_size:

break

return pairs, labels

# === Строковые метрики ===

def damerau\_levenshtein(s1, s2): return textdistance.damerau\_levenshtein.distance(s1, s2)

def levenshtein(s1, s2): return textdistance.levenshtein.distance(s1, s2)

def hamming(s1, s2): return textdistance.hamming.distance(s1, s2) if len(s1) == len(s2) else None

def jaro\_winkler(s1, s2): return textdistance.jaro\_winkler.similarity(s1, s2)

def lcs(s1, s2): return textdistance.lcsseq.similarity(s1, s2)

def jaccard\_q3(s1, s2): return textdistance.Jaccard(qval=3).similarity(s1, s2)

def cosine\_q3(s1, s2): return textdistance.Cosine(qval=3).similarity(s1, s2)

# === Корпусные метрики ===

def tfidf\_metrics(pairs):

texts = [x for p in pairs for x in p]

vectorizer = TfidfVectorizer()

x = vectorizer.fit\_transform(texts)

tfidf, eucl, manh = [], [], []

for i in range(0, len(texts), 2):

tfidf.append(cosine\_similarity(x[i], x[i+1])[0][0])

eucl.append(euclidean\_distances(x[i], x[i+1])[0][0])

manh.append(manhattan\_distances(x[i], x[i+1])[0][0])

return tfidf, eucl, manh

def lsa\_metrics(pairs):

texts = [x for p in pairs for x in p]

vectorizer = TfidfVectorizer()

x = vectorizer.fit\_transform(texts)

svd = TruncatedSVD(n\_components=100)

x\_reduced = svd.fit\_transform(x)

scores = []

for i in range(0, len(x\_reduced), 2):

scores.append(cosine\_similarity([x\_reduced[i]], [x\_reduced[i+1]])[0][0])

return scores

# === Семантические метрики ===

def sbert\_model\_scores(pairs, model\_name):

model = SentenceTransformer(model\_name)

scores = []

for s1, s2 in pairs:

emb1 = model.encode(s1, convert\_to\_tensor=True)

emb2 = model.encode(s2, convert\_to\_tensor=True)

scores.append(util.pytorch\_cos\_sim(emb1, emb2).item())

return scores

def use\_model\_scores(pairs):

model = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4")

scores = []

for s1, s2 in pairs:

emb = model([s1, s2])

sim = cosine\_similarity(emb[0:1].numpy(), emb[1:2].numpy())[0][0]

scores.append(sim)

return scores

def elmo\_model\_scores(pairs):

elmo = hub.load("https://tfhub.dev/google/elmo/3")

scores = []

for s1, s2 in pairs:

emb1 = elmo.signatures['default'](tf.constant([s1]))['elmo']

emb2 = elmo.signatures['default'](tf.constant([s2]))['elmo']

emb1\_avg = tf.reduce\_mean(emb1, axis=1)

emb2\_avg = tf.reduce\_mean(emb2, axis=1)

sim = cosine\_similarity(emb1\_avg.numpy(), emb2\_avg.numpy())[0][0]

scores.append(sim)

return scores

# === InferSent интеграция ===

from models import InferSent

def load\_infersent\_model():

params\_model = {

'bsize': 64,

'word\_emb\_dim': 300,

'enc\_lstm\_dim': 2048,

'pool\_type': 'max',

'dpout\_model': 0.0,

'version': 2

}

model = InferSent(params\_model)

model.load\_state\_dict(torch.load("infersent2.pkl"))

model.set\_w2v\_path("glove.840B.300d.txt")

model.build\_vocab\_k\_words(K=100000)

return model

def infersent\_model\_scores(pairs, model):

scores = []

for s1, s2 in pairs:

emb1 = model.encode([s1])[0]

emb2 = model.encode([s2])[0]

sim = cosine\_similarity([emb1], [emb2])[0][0]

scores.append(sim)

return scores

# === Корреляции ===

def compute\_correlations(metric\_scores, ground\_truth):

spearman\_corr, \_ = spearmanr(metric\_scores, ground\_truth)

pearson\_corr, \_ = pearsonr(metric\_scores, ground\_truth)

return spearman\_corr, pearson\_corr

# === Основной блок ===

if \_\_name\_\_ == "\_\_main\_\_":

data\_pairs, true\_scores = load\_stsb\_data(sample\_size=100)

tfidf\_cos, tfidf\_euc, tfidf\_man = tfidf\_metrics(data\_pairs)

lsa = lsa\_metrics(data\_pairs)

sbert = sbert\_model\_scores(data\_pairs, "all-MiniLM-L6-v2")

use = use\_model\_scores(data\_pairs)

glove = sbert\_model\_scores(data\_pairs, "sentence-transformers/paraphrase-distilroberta-base-v1")

word2vec = sbert\_model\_scores(data\_pairs, "sentence-transformers/paraphrase-MiniLM-L6-v2")

elmo = elmo\_model\_scores(data\_pairs)

infersent\_model = load\_infersent\_model()

infsent = infersent\_model\_scores(data\_pairs, infersent\_model)

lev, dam, ham, jar, lcs\_, jac, cos\_q3 = [], [], [], [], [], [], []

for text1, text2 in data\_pairs:

lev.append(levenshtein(text1, text2))

dam.append(damerau\_levenshtein(text1, text2))

ham.append(hamming(text1, text2) or 0.0)

jar.append(jaro\_winkler(text1, text2))

lcs\_.append(lcs(text1, text2))

jac.append(jaccard\_q3(text1, text2))

cos\_q3.append(cosine\_q3(text1, text2))

all\_metrics = {

"Levenshtein": lev,

"Damerau-Levenshtein": dam,

"Hamming": ham,

"Jaro-Winkler": jar,

"LCS": lcs\_,

"Jaccard\_q3": jac,

"Cosine\_q3": cos\_q3,

"TFIDF\_Cosine": tfidf\_cos,

"TFIDF\_Euclidean": tfidf\_euc,

"TFIDF\_Manhattan": tfidf\_man,

"LSA": lsa,

"Word2Vec": word2vec,

"Glove": glove,

"Elmo": elmo,

"USE": use,

"InferSent": infsent,

"SBERT": sbert

}

print("=== Информативность (корреляция с ground-truth) ===")

print(f"{'Метрика':<20} | {'Spearman':>9} | {'Pearson':>9}")

print("-"\*44)

for name, scores in all\_metrics.items():

s, p = compute\_correlations(scores, true\_scores)

print(f"{name:<20} | {s:>9.4f} | {p:>9.4f}")