Министерство науки и высшего образования

Российской Федерации

Федеральное государственное бюджетное

образовательное учреждение высшего образования

«Новосибирский государственный технический университет»

**Кафедра теоретической и прикладной информатики**

**Практическая работа**

по дисциплине «Программные системы статистического анализа»

**Тема: “** **Сравнение метрик, реализованных для вычисления расстояния между строками в пакете stringdist.”**

Факультет: ПМИ

Группа: ПМИМ-31

Студент: Гончаренко С.К.

Преподаватель: Тимофеева А.Ю.

Новосибирск, 2023

**СОДЕРЖАНИЕ**

[Реализация алгоритмов на языке C# 3](#_Toc156207744)

[Реализация алгоритмов на языке R 9](#_Toc156207745)

[Моделирование 11](#_Toc156207746)

[Практический пример 16](#_Toc156207747)

[Выводы, интерпретация: 19](#_Toc156207748)

## Реализация алгоритмов на языке C#

**Расстояние Левенштейна**

Расстояние Левенштейна, также известное как редакционное расстояние или расстояние редактирования, представляет собой метрику, измеряющую минимальное количество редакционных операций, необходимых для преобразования одной строки в другую. Эти операции включают в себя вставку одного символа, удаление одного символа и замену одного символа на другой.

Рассмотрим пример. Пусть у нас есть две строки: "кот" и "кит". Расстояние Левенштейна между ними будет 1, потому что для превращения "кот" в "кит" нужно всего лишь заменить "о" на "и".

Алгоритм вычисления расстояния Левенштейна обычно реализуется с использованием динамического программирования. Он строит матрицу, где элемент (i, j) представляет собой расстояние между первыми i символами одной строки и первыми j символами другой строки. Значение в правом нижнем углу матрицы представляет собой искомое расстояние Левенштейна.

Пример работы алгоритма на словах "кот" и "кит"(рис 1.):

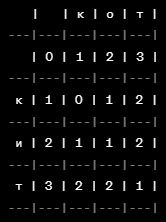


Рисунок 1 - Пример работы алгоритма

В этой матрице каждый элемент (i, j) представляет собой расстояние Левенштейна между первыми i символами "кот" и первыми j символами "кит". Значение в правом нижнем углу (3, 3) равно 1, что и является расстоянием Левенштейна между словами "кот" и "кит".

Этот алгоритм широко используется в области компьютерной лингвистики, поиска по тексту, автокоррекции и других приложениях, где необходимо измерять схожесть или расстояние между строками.

using System;

using System.Diagnostics;

class Program

{

// Метод для вычисления расстояния Левенштейна

static int LevenshteinDistance(string str1, string str2)

{

int len1 = str1.Length;

int len2 = str2.Length;

int[,] dp = new int[len1 + 1, len2 + 1];

for (int i = 0; i <= len1; i++)

{

for (int j = 0; j <= len2; j++)

{

if (i == 0)

dp[i, j] = j;

else if (j == 0)

dp[i, j] = i;

else

dp[i, j] = Math.Min(Math.Min(dp[i - 1, j] + 1, dp[i, j - 1] + 1), dp[i - 1, j - 1] + (str1[i - 1] == str2[j - 1] ? 0 : 1));

}

}

return dp[len1, len2];

}

// Метод для тестирования производительности

static void Benchmark()

{

// Генерация тестовых данных

string str1 = "кот";

string str2 = "кит";

// Инициализация объекта для измерения времени выполнения

Stopwatch stopwatch = new Stopwatch();

// Запуск измерения времени

stopwatch.Start();

// Выполнение операции (в данном случае, вычисление расстояния Левенштейна)

int result = LevenshteinDistance(str1, str2);

// Остановка измерения времени

stopwatch.Stop();

// Вывод результатов

Console.WriteLine($"Levenshtein distance between '{str1}' and '{str2}': {result}");

Console.WriteLine($"Time elapsed: {stopwatch.Elapsed}");

}

// Точка входа в программу

static void Main()

{

// Запуск теста производительности

Benchmark();

}

}

Приведенный код на C# реализует вычисление расстояния Левенштейна между двумя строками. Давайте разберем его пошагово:

1. **Метод LevenshteinDistance**:
   * **int LevenshteinDistance(string str1, string str2)**: Этот метод принимает две строки (**str1** и **str2**) и возвращает расстояние Левенштейна между ними.
   * **int len1 = str1.Length;** и **int len2 = str2.Length;**: Определяют длины строк.
   * **int[,] dp = new int[len1 + 1, len2 + 1];**: Создает двумерный массив **dp**, который будет использоваться для хранения промежуточных результатов вычислений.
   * Вложенные циклы **for** итерируются от 0 до длины каждой строки:
     1. Внутри цикла проверяется, находится ли текущая итерация в начале строки (**i == 0** или **j == 0**). Если да, то устанавливается начальное значение в массиве **dp**.
     2. В противном случае, используется рекурсивное соотношение для вычисления расстояния Левенштейна: **dp[i, j] = Math.Min(Math.Min(dp[i - 1, j] + 1, dp[i, j - 1] + 1), dp[i - 1, j - 1] + (str1[i - 1] == str2[j - 1] ? 0 : 1));**. Здесь **Math.Min** используется для нахождения минимального значения среди трех возможных операций: вставка, удаление и замена.
   * В конце метод возвращает значение в правом нижнем углу массива **dp**, которое представляет собой расстояние Левенштейна между двумя строками.
2. **Метод Benchmark**:
   * Этот метод предназначен для тестирования производительности алгоритма. Он использует строки "кот" и "кит" в качестве тестовых данных.
   * Создается объект **Stopwatch** для измерения времени выполнения операции.
   * Запускается измерение времени, вызывается метод **LevenshteinDistance** для вычисления расстояния Левенштейна, и затем останавливается измерение времени.
   * Результат и время выполнения выводятся на консоль.
3. **Метод Main**:
   * Точка входа в программу. Вызывает метод **Benchmark** для запуска теста производительности.

При выполнении кода вы увидите, что расстояние Левенштейна между словами "кот" и "кит" равно 1 (замена 'о' на 'и'), и время выполнения будет отображено. Этот код полезен, когда необходимо измерить схожесть или различия между двумя строками в контексте редакционных операций.

Результат представлен на рисунке 2.

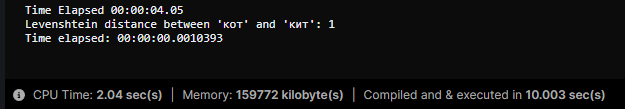


Рисунок 2 – Результат расстояния Левенштейна на языке C#

**Расстояние Хэмминга**

Расстояние Хэмминга - это метрика, используемая для измерения различий между двумя строками одинаковой длины. Она представляет собой сумму позиций, на которых соответствующие символы в двух строках различаются. Расстояние Хэмминга особенно полезно, когда строки представляют собой последовательности битов, и вы хотите определить, насколько они различаются.

using System;

class Program

{

static int HammingDistance(string str1, string str2)

{

if (str1.Length != str2.Length)

{

throw new ArgumentException("Строки должны быть одинаковой длины");

}

int distance = 0;

for (int i = 0; i < str1.Length; i++)

{

if (str1[i] != str2[i])

{

distance++;

}

}

return distance;

}

static void Main()

{

string str1 = "1010101";

string str2 = "1001001";

try

{

int hammingDistance = HammingDistance(str1, str2);

Console.WriteLine($"Расстояние Хэмминга между {str1} и {str2}: {hammingDistance}");

}

catch (ArgumentException ex)

{

Console.WriteLine(ex.Message);

}

}

}

Результат на рисунке 3.

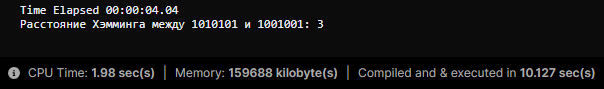


Рисунок 3 - Результат расстояния Хэмминга на языке C#

**Определение функции HammingDistance:**

Эта функция принимает две строки **str1** и **str2** в качестве аргументов и возвращает целочисленное значение - расстояние Хэмминга между ними.

* Первым делом, функция проверяет, что длины строк равны. Если нет, выбрасывается исключение **ArgumentException**, так как расстояние Хэмминга требует, чтобы строки были одинаковой длины.
* Затем идет цикл **for**, который проходит по каждому символу в строках. Если символы на соответствующих позициях не совпадают, увеличивается счетчик **distance**.
* Функция возвращает значение **distance**, представляющее расстояние Хэмминга между строками.

**В методе Main:**

* + Создаются две строки **str1** и **str2**.
  + В блоке **try** вызывается функция **HammingDistance** с этими строками, и результат сохраняется в переменной **hammingDistance**.
  + Если строки не имеют одинаковой длины, функция может выбросить исключение **ArgumentException**, которое обрабатывается в блоке **catch**. В этом случае выводится сообщение об ошибке.
  + В противном случае выводится результат - расстояние Хэмминга между **str1** и **str2**.

## Реализация алгоритмов на языке RНачало формы

**Расстояние Левенштейна**

# Функция для вычисления расстояния Левенштейна

levenshtein\_distance <- function(str1, str2) {

len1 <- nchar(str1)

len2 <- nchar(str2)

# Создание матрицы для хранения промежуточных результатов

dp <- matrix(0, nrow = len1 + 1, ncol = len2 + 1)

# Инициализация начальных значений

dp[ ,1] <- 0:len1

dp[1, ] <- 0:len2

# Заполнение матрицы

for (i in 2:(len1 + 1)) {

for (j in 2:(len2 + 1)) {

cost <- ifelse(substr(str1, i - 1, i - 1) == substr(str2, j - 1, j - 1), 0, 1)

dp[i, j] <- min(c(dp[i - 1, j] + 1, dp[i, j - 1] + 1, dp[i - 1, j - 1] + cost))

}

}

return(dp[len1 + 1, len2 + 1])

}

# Пример использования

str1 <- "кот"

str2 <- "кит"

distance <- levenshtein\_distance(str1, str2)

cat("Расстояние Левенштейна между '", str1, "' и '", str2, "':", distance, "\n")

Результат расстояние Левенштайна на языке R представлен на рисунке 4.

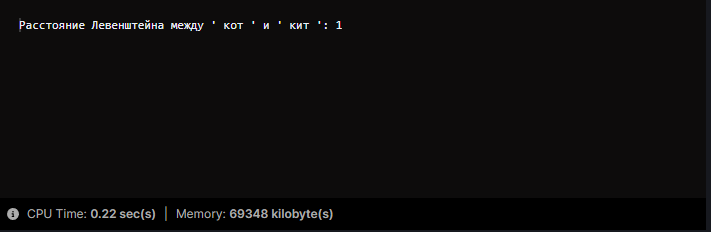


Рисунок 4 - Результат расстояния Левенштейна на языке R

В этом коде **levenshtein\_distance** является функцией, которая принимает две строки (**str1** и **str2**) и возвращает расстояние Левенштейна между ними. Функция использует матрицу **dp** для хранения промежуточных результатов и два вложенных цикла для заполнения этой матрицы в соответствии с алгоритмом Левенштейна.

**Расстояние ХэммингаНачало формы**

# Функция для расчета расстояния Хэмминга

hamming\_distance <- function(str1, str2) {

if (nchar(str1) != nchar(str2)) {

stop("Строки должны быть одинаковой длины")

}

# Преобразование строк в векторы символов

vec1 <- strsplit(str1, NULL)[[1]]

vec2 <- strsplit(str2, NULL)[[1]]

# Сравнение символов и подсчет несовпадений

distance <- sum(vec1 != vec2)

return(distance)

}

# Пример использования

str1 <- "1010101"

str2 <- "1001001"

tryCatch({

distance <- hamming\_distance(str1, str2)

cat("Расстояние Хэмминга между", str1, "и", str2, ":", distance, "\n")

}, error = function(e) {

cat("Ошибка:", e$message, "\n")

})

Результат представлен на рисунке 5.

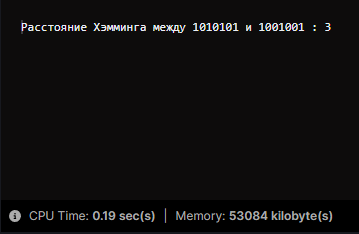


Рисунок 5 - Результат расстояния Хэмминга на языке R

## Моделирование

Вычисление расстояние Левенштейна для двух случайных строк на C#:

using System;

class Program

{

static void Main()

{

// Генерация случайных строк

string str1 = GenerateRandomString();

string str2 = GenerateRandomString();

// Вывод сгенерированных строк

Console.WriteLine($"String 1: {str1}");

Console.WriteLine($"String 2: {str2}");

// Вычисление и вывод расстояния Левенштейна

int distance = LevenshteinDistance(str1, str2);

Console.WriteLine($"Levenshtein distance: {distance}");

}

static int LevenshteinDistance(string str1, string str2)

{

int len1 = str1.Length;

int len2 = str2.Length;

int[,] dp = new int[len1 + 1, len2 + 1];

for (int i = 0; i <= len1; i++)

{

for (int j = 0; j <= len2; j++)

{

if (i == 0)

dp[i, j] = j;

else if (j == 0)

dp[i, j] = i;

else

dp[i, j] = Math.Min(Math.Min(dp[i - 1, j] + 1, dp[i, j - 1] + 1), dp[i - 1, j - 1] + (str1[i - 1] == str2[j - 1] ? 0 : 1));

}

}

return dp[len1, len2];

}

static string GenerateRandomString()

{

Random random = new Random();

int length = random.Next(5, 15); // Генерация случайной длины строки от 5 до 15 символов

const string chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

char[] randomChars = new char[length];

for (int i = 0; i < length; i++)

{

randomChars[i] = chars[random.Next(chars.Length)];

}

return new string(randomChars);

}

}

Этот код на C# представляет собой консольное приложение, которое генерирует две случайные строки и затем вычисляет расстояние Левенштейна между ними. Основные элементы кода:

1. **Метод Main**:
   * Генерирует две случайные строки с помощью метода **GenerateRandomString**.
   * Выводит сгенерированные строки на консоль.
   * Вычисляет и выводит расстояние Левенштейна между этими строками с использованием метода **LevenshteinDistance**.
2. **Метод LevenshteinDistance**:
   * Принимает две строки (**str1** и **str2**) и возвращает расстояние Левенштейна между ними.
   * Использует динамическое программирование для создания матрицы dp, где хранятся промежуточные результаты.
   * Заполняет матрицу согласно алгоритму Левенштейна.
   * Возвращает значение в правом нижнем углу матрицы, представляющее собой расстояние Левенштейна.
3. **Метод GenerateRandomString**:
   * Генерирует случайную строку заданной длины (от 5 до 15 символов) из латинских букв (верхнего и нижнего регистра).

Когда программа выполняется, она выводит сгенерированные строки, а затем выводит расстояние Левенштейна между ними. Код также содержит генерацию случайных строк, что полезно для тестирования алгоритма на различных данных

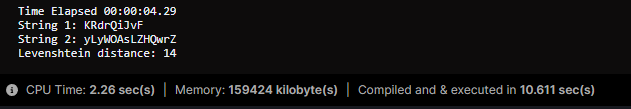
Результат компиляции представлен на рисунке 6.  


Рисунок 6 – Результат компиляции

Расстояние Хэмминга между двумя случайно сгенерированными строками на C#:

using System;

class Program

{

static void Main()

{

// Генерация двух случайных строк битов

string str1 = GenerateRandomBinaryString(10);

string str2 = GenerateRandomBinaryString(10);

Console.WriteLine($"Сгенерированные строки: {str1} и {str2}");

try

{

// Вычисление расстояния Хэмминга

int hammingDistance = HammingDistance(str1, str2);

Console.WriteLine($"Расстояние Хэмминга между строками: {hammingDistance}");

}

catch (ArgumentException ex)

{

Console.WriteLine(ex.Message);

}

}

static int HammingDistance(string str1, string str2)

{

if (str1.Length != str2.Length)

{

throw new ArgumentException("Строки должны быть одинаковой длины");

}

int distance = 0;

for (int i = 0; i < str1.Length; i++)

{

if (str1[i] != str2[i])

{

distance++;

}

}

return distance;

}

static string GenerateRandomBinaryString(int length)

{

Random random = new Random();

char[] binaryString = new char[length];

for (int i = 0; i < length; i++)

{

binaryString[i] = (random.Next(2) == 0) ? '0' : '1';

}

return new string(binaryString);

}

}

1. **GenerateRandomBinaryString функция:**

Эта функция генерирует случайную строку битов указанной длины. Она использует объект Random для генерации случайных чисел и создает массив char для представления строки битов. Затем она заполняет этот массив случайными символами '0' или '1' и возвращает новую строку, созданную из этого массива.

1. **HammingDistance функция:**

Эта функция вычисляет расстояние Хэмминга между двумя строками. Она проверяет, что строки имеют одинаковую длину, иначе выбрасывает исключение. Затем она проходит по каждому символу в строках и увеличивает счетчик distance при обнаружении несовпадения.

1. **Main метод:**

* В методе Main создаются две случайные строки битов (str1 и str2) с использованием функции **GenerateRandomBinaryString**.
* Затем выводятся сгенерированные строки.
* Пробуем вычислить расстояние Хэмминга между этими строками с помощью функции **HammingDistance**. Если строки имеют разную длину, то выбрасывается исключение, и выводится соответствующее сообщение об ошибке. В противном случае выводится расстояние Хэмминга между строками.

Результат компиляции представлен на рисунке 7.

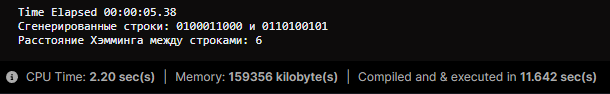


Рисунок 7

## Практический пример

Ниже представлен фрагмент кода с примером вычисления расстояния между строками. Возвращает количество изменений символов (удалений, вставок, замен), которые необходимо выполнить для перехода из строки A в строку B. В этом коде вместо зубчатого массива используется двумерный массив, поскольку требуемое пространство будет иметь только одну ширину и одну высоту. Пример взят с сайта <https://thedeveloperblog.com/c-sharp/levenshtein>.

using System;

/// <summary>

/// Contains approximate string matching

/// </summary>

static class LevenshteinDistance

{

/// <summary>

/// Compute the distance between two strings.

/// </summary>

public static int Compute(string s, string t)

{

int n = s.Length;

int m = t.Length;

int[,] d = new int[n + 1, m + 1];

// Step 1

if (n == 0)

{

return m;

}

if (m == 0)

{

return n;

}

// Step 2

for (int i = 0; i <= n; d[i, 0] = i++)

{

}

for (int j = 0; j <= m; d[0, j] = j++)

{

}

// Step 3

for (int i = 1; i <= n; i++)

{

//Step 4

for (int j = 1; j <= m; j++)

{

// Step 5

int cost = (t[j - 1] == s[i - 1]) ? 0 : 1;

// Step 6

d[i, j] = Math.Min(

Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1),

d[i - 1, j - 1] + cost);

}

}

// Step 7

return d[n, m];

}

}

class Program

{

static void Main()

{

Console.WriteLine(LevenshteinDistance.Compute("aunt", "ant"));

Console.WriteLine(LevenshteinDistance.Compute("Sam", "Samantha"));

Console.WriteLine(LevenshteinDistance.Compute("flomax", "volmax"));

}

}

Результат вычисления расстояния Левенштейна представлен на рисунке 8.

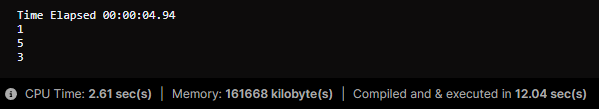


Рисунок 8 – Результат вычисления расстояния Левенштейна

Ниже представлен фрагмент кода с примером алгоритма расстояние Хэмминга. В приведенном ниже примере мы возьмем две строки, и если длина строк не равна, то мы покажем исключение, иначе он вычислит расстояние между двумя строками. Пример взят с сайта: https://www.nookery.ru/hamming-distance-and-lowenstein-distance-example-implementation-in-c/

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Hamming\_Distance

{

public static class StringDistance

{

public static int GetHammingDistance(string s, string t)

{

if (s.Length != t.Length)

{

throw new Exception("Строки должны иметь одинаковую длину");

}

int distance =

s.ToCharArray()

.Zip(t.ToCharArray(), (c1, c2) => new { c1, c2 })

.Count(m => m.c1 != m.c2);

return distance;

}

}

class Program

{

static void Main()

{

Console.WriteLine(StringDistance.GetHammingDistance("Лук", "Люк"));

Console.WriteLine(StringDistance.GetHammingDistance("Трактор", "Самолет"));

Console.WriteLine(StringDistance.GetHammingDistance("Кремень", "Картон"));

Console.ReadKey();

}

}

}

Результат алгоритма расстояние Хэмминга представлен на рисунке 9.

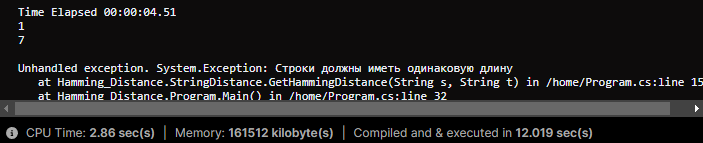


Рисунок 9 – Результат алгоритма расстояние Хэмминга

## Выводы, интерпретация:

Проведем сравнение языков C# и R по затраченному времени и используемой памяти:

Таблица 1 - Время выполнения на разных языках в секундах.

|  |  |  |
| --- | --- | --- |
| Наименование | C# | R |
| Расстояние Левенштейна | 2,04 | 0,22 |
| Расстояние Хэмминга | 1,98 | 0,19 |

Таблица 2 - Используемая память в килобайтах

|  |  |  |
| --- | --- | --- |
| Наименование | C# | R |
| Расстояние Левенштейна | 159772 | 69348 |
| Расстояние Хэмминга | 159688 | 53084 |

Подводя итоги сравнения, язык R имеет преимущества на двух этапах сравнения.

R и C# - это два различных языка программирования, которые часто используются в разных областях. Оба языка могут успешно решить задачи вычисления расстояния Левенштейна и Хэмминга. Выбор между R и C# может зависеть от того, в какой среде вы удобнее работаете, а также от других требований вашего проекта. Если у вас уже есть предпочтение по языку программирования или вы работаете в специфической области, это также может влиять на ваш выбор.

Рассмотрим основные аспекты, по которым R является более подходящим языком для расчета расстояния Левенштейна и расстояния Хэмминга:

1. **Богатая экосистема для анализа данных:**

R создавался с учетом анализа данных и статистики, и его экосистема насчитывает большое количество пакетов, предназначенных для обработки текстовых данных, включая вычисление расстояния Левенштейна.

1. **Готовые пакеты и функции:**

В R существует несколько готовых пакетов, таких как **stringdist**, которые содержат оптимизированные функции для расчета расстояния Левенштейна. Эти пакеты упрощают реализацию и обеспечивают высокую производительность.

1. **Простой и выразительный синтаксис:**

Синтаксис R обычно более простой и выразительный, что делает его удобным для реализации и использования в контексте статистического анализа.

1. **Встроенные операции с текстовыми данными:**

R предоставляет удобные встроенные операции для работы с текстовыми данными, такие как разделение строк, доступ к символам и т. д.

1. **Богатая экосистема:**

R имеет богатую экосистему для анализа данных, что может быть полезно, если вы занимаетесь анализом текстовых данных и вам нужны дополнительные статистические и визуализационные инструменты.

Хотя R имеет свои преимущества в контексте анализа данных и работы с текстовыми данными, важно отметить, что C# также способен эффективно решать эти задачи. Выбор между R и C# может зависеть от контекста вашего проекта, ваших предпочтений и общих требований вашей команды разработки.