

Министерство науки и высшего образования РФ
федеральное государственное автономное образовательное учреждение высшего
образования

«Омский государственный технический университет»

Факультет информационных технологий и компьютерных систем

Кафедра «Прикладная математика и фундаментальная информатика»

ЛАБОРАТОРНАЯ РАБОТА №1
«Вероятностные алгоритмы»

по дисциплине «Практикум по программированию»

Вариант 14

Студента	<u>Рау Алексея Евгеньевича</u> фамилия, имя, отчество полностью		
Курс	<u>2</u>	Группа	<u>ФИТ-231</u>
Направление	<u>02.03.02 Фундаментальная информатика и информационные технологии</u> код, наименование		
Проверил	<u>ассистент</u> должность <u>Плескунов Д. А.</u> фамилия, инициалы		
Выполнил	<u></u> дата, подпись студента <u></u> дата, подпись преподавателя		

г. Омск

2025 г.

Задание №1 «Фильтр Блума»

Задача №1 «Реализация фильтра Блума»

Разработать фильтр Блума с использованием стандартной библиотеки *Python*, включая собственные хеш-функции.

Решение

Реализовать фильтр Блума с использованием стандартной библиотеки *Python* (при этом реализовать собственные хеш-функции:

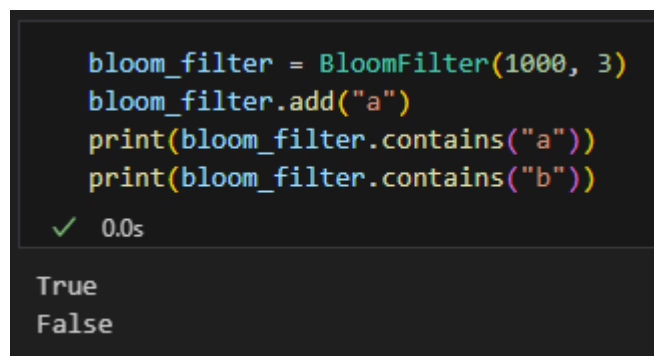
```
class BloomFilter:
    def __init__(self, m, k):
        self.m = m
        self.k = k
        self.bit_array = np.zeros(m, dtype=bool)
        self.hash_functions = [self._create_hash_function(i) for i in
                                range(k)]

    def _create_hash_function(self, seed):
        def hash_func(item):
            hasher = hashlib.sha256()
            hasher.update(f"{seed}".encode('utf-8'))
            hasher.update(str(item).encode('utf-8'))
            return int(hasher.hexdigest(), 16) % self.m
        return hash_func

    def add(self, item):
        for hf in self.hash_functions:
            self.bit_array[hf(item)] = True

    def contains(self, item):
        return all(self.bit_array[hf(item)] for hf in self.hash_functions)
```

Пример работы программы представлен на рисунке 1.



```
bloom_filter = BloomFilter(1000, 3)
bloom_filter.add("a")
print(bloom_filter.contains("a"))
print(bloom_filter.contains("b"))
```

✓ 0.0s

True
False

Рисунок 1 – работа программы для задачи №1

Данная программа позволяет добавлять элементы в фильтр Блума и проверять их наличие.

Задача №2 «Оценка ложноположительных срабатываний»

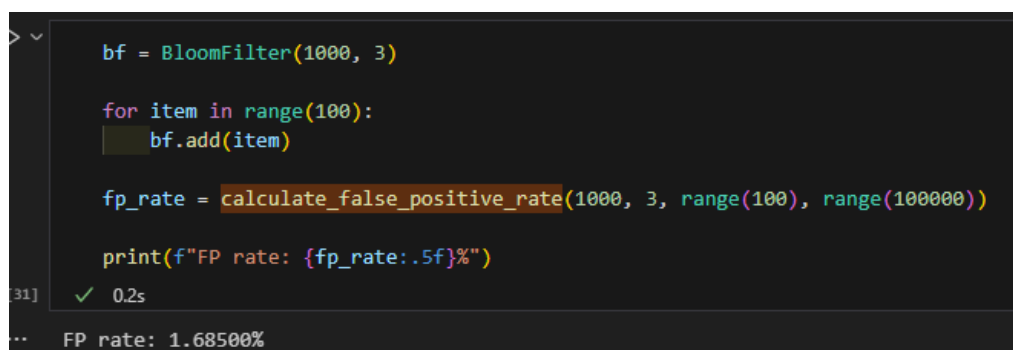
Определить процент ложноположительных срабатываний конкретной реализации.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
def calculate_false_positive_rate(m, k, added_elements, test_elements):  
    bf = BloomFilter(m, k)  
    for elem in added_elements:  
        bf.add(elem)  
    return (sum(1 for elem in test_elements if bf.contains(elem)) /  
            len(test_elements)) * 100
```

Пример работы функции представлен на рисунке 2.



```
> ✓ bf = BloomFilter(1000, 3)  
for item in range(100):  
    bf.add(item)  
fp_rate = calculate_false_positive_rate(1000, 3, range(100), range(100000))  
print(f"FP rate: {fp_rate:.5f}%")  
[31] ✓ 0.2s  
... FP rate: 1.68500%
```

Рисунок 2 – работа функции для задачи №2

Данная функция принимает тестовые данные и проверяет их наличие в фильтре Блума. Если элемент не был добавлен в фильтр, но проверка возвращает положительный результат, это считается ложноположительным срабатыванием. Программа подсчитывает процент таких срабатываний.

Задача №3 «Анализ зависимости от гиперпараметров»

Оценить зависимость ложноположительных срабатываний относительно размерности массива *m* и числа хеш-функций *k* (таблица и графики зависимостей).

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
n = 1000
test_size = 5000
m_values = [1000, 2000, 4000, 8000, 16000]
k_values = [1, 2, 3, 4, 5, 6, 7, 8]

added_elements = generate_random_strings(n)
test_elements = generate_random_strings(test_size)

assert len(set(added_elements) & set(test_elements)) == 0, "Тестовые
элементы не должны пересекаться с добавленными"

results = []
for m in m_values:
    for k in k_values:
        rate = calculate_false_positive_rate(m, k, added_elements,
test_elements)
        results.append({'m': m, 'k': k, 'false_positive_rate': rate})

df = pd.DataFrame(results)

plt.figure(figsize=(10, 6))
for m in m_values:
    subset = df[df['m'] == m]
    plt.plot(subset['k'], subset['false_positive_rate'], marker='o',
label=f'm={m}')
plt.xlabel('k')
plt.ylabel('False Positive Rate (%)')
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 6))
for k in k_values:
    subset = df[df['k'] == k]
    plt.plot(subset['m'], subset['false_positive_rate'], marker='o',
label=f'k={k}')
plt.xlabel('m')
plt.ylabel('False Positive Rate (%)')
plt.legend()
plt.grid(True)
plt.show()

print(df.pivot(index='m', columns='k', values='false_positive_rate'))

bf1 = BloomFilter(1000, 3)
bf1.add("a")
bf2 = BloomFilter(1000, 3)
bf2.add("b")

bf_union = bf1.union(bf2)
```

```

print("Объединение содержит 'a'", bf_union.contains("a"))
print("Объединение содержит 'b'", bf_union.contains("b"))
bf_intersection = bf1.intersect(bf2)
print("Пересечение содержит 'a':", bf_intersection.contains("a"))

```

Пример работы программы представлен на рисунках 3 и 4.

k	1	2	3	4	5	6	7	8
m								
1000	61.34	73.36	85.02	89.28	95.48	97.90	99.18	100.00
2000	38.16	37.58	43.60	50.60	62.02	70.76	80.90	86.54
4000	22.68	14.94	14.40	15.40	17.40	22.48	26.38	32.16
8000	11.96	5.44	3.60	2.42	2.14	2.44	2.56	2.92
16000	6.34	1.40	0.42	0.32	0.12	0.12	0.04	0.06

Рисунок 3 – статистическая таблица для задачи №3

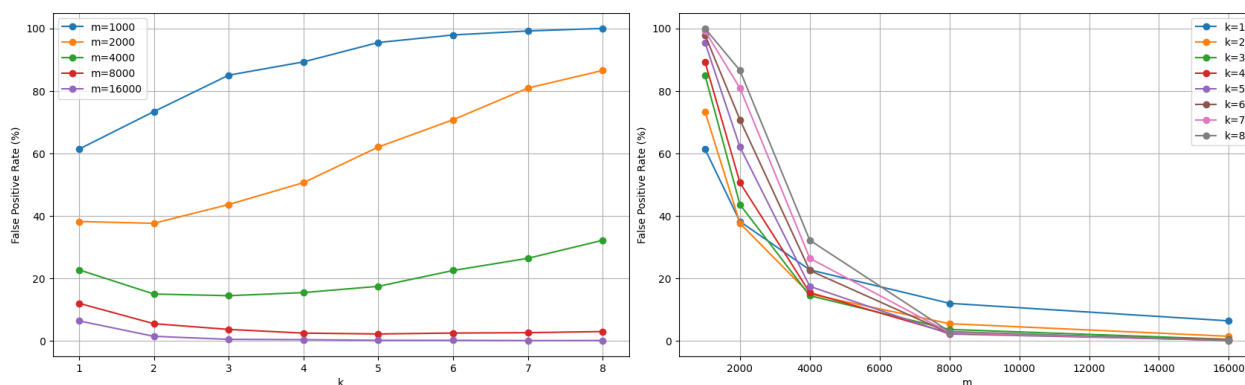


Рисунок 4 – статистические графики для задачи №3

Данная программа тестирует фильтр Блума с разными параметрами (размер массива (m) и число хеш-функций (k)), вычисляя процент ложных срабатываний. Результаты выводятся в виде таблицы и графиков для анализа оптимальных параметров.

Задача №4 «Операции с фильтрами Блума»

Реализовать возможность пересечения и объединения фильтров Блума.

Решение

Проверка возможности выполнения операций на языке *Python* представлена ниже:

```

def union(self, other):
    if self.m != other.m or self.k != other.k:
        raise ValueError("Фильтры должны иметь одинаковые параметры m и k")
    new_bf = BloomFilter(self.m, self.k)

```

```


        new_bf.bit_array = np.logical_or(self.bit_array, other.bit_array)
        return new_bf

    def intersect(self, other):
        if self.m != other.m or self.k != other.k:
            raise ValueError("Фильтры должны иметь одинаковые параметры m и k")

        new_bf = BloomFilter(self.m, self.k)
        new_bf.bit_array = np.logical_and(self.bit_array, other.bit_array)
        return new_bf

```

Пример работы программы представлен на рисунке 5.



```

bf1 = BloomFilter(1000, 3)
bf1.add("a")
bf1.add("c")
bf2 = BloomFilter(1000, 3)
bf2.add("b")
bf2.add("c")

bf_union = bf1.union(bf2)
print("Объединение содержит 'a'", bf_union.contains("a"))
print("Объединение содержит 'b'", bf_union.contains("b"))
print("Объединение содержит 'c'", bf_union.contains("c"))
print("")
bf_intersection = bf1.intersect(bf2)
print("Пересечение содержит 'a':", bf_intersection.contains("a"))
print("Пересечение содержит 'b':", bf_intersection.contains("b"))
print("Пересечение содержит 'c':", bf_intersection.contains("c"))

```

✓ 0.0s

```

Объединение содержит 'a' True
Объединение содержит 'b' True
Объединение содержит 'c' True

Пересечение содержит 'a': False
Пересечение содержит 'b': False
Пересечение содержит 'c': True

```

Рисунок 5 – проверка операций для задачи №4

Данная программа реализует проверку операций объединения и пересечения для фильтров Блума.

Задание №2 «Фильтр Блума со счетчиком»

Задача №1 «Реализация фильтра Блума со счетчиком»

Разработать фильтр Блума на основе счётчиков для поддержки операции удаления элементов.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
class CountingBloomFilter:
    def __init__(self, m, k):
        self.m = m
        self.k = k
        self.counters = np.zeros(m, dtype=int)
        self.hash_functions = [self._create_hash_function(seed) for seed in
range(k)]

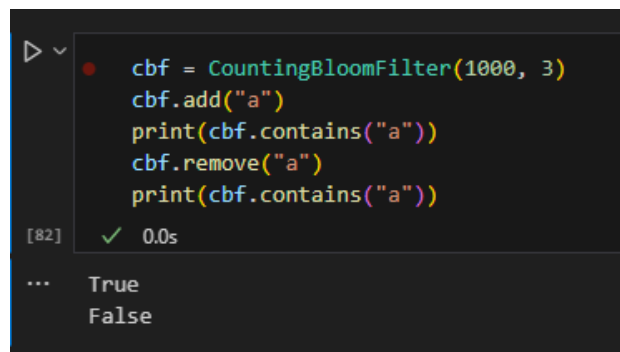
    def _create_hash_function(self, seed):
        def hash_func(item):
            hasher = hashlib.sha256()
            hasher.update(f"{seed}".encode('utf-8'))
            hasher.update(str(item).encode('utf-8'))
            return int(hasher.hexdigest(), 16) % self.m
        return hash_func

    def add(self, item):
        for hf in self.hash_functions:
            index = hf(item)
            self.counters[index] += 1

    def remove(self, item):
        for hf in self.hash_functions:
            index = hf(item)
            if self.counters[index] > 0:
                self.counters[index] -= 1

    def contains(self, item):
        return all(self.counters[hf(item)] > 0 for hf in
self.hash_functions)
    def __del__(self):
        return new_filter
```

Пример работы программы представлен на рисунке 6.



```
cbf = CountingBloomFilter(1000, 3)
cbf.add("a")
print(cbf.contains("a"))
cbf.remove("a")
print(cbf.contains("a"))
```

[82] ✓ 0.0s

... True
False

Рисунок 6 – работа программы для задачи №1

Улучшенная программа использует массив счётчиков вместо битов, что позволяет удалять элементы из фильтра. При добавлении элемента она увеличивает счётчики, при удалении — уменьшает. Проверка наличия требует, чтобы все соответствующие счётчики были ненулевыми.

Задача №2 «Оценка ложноположительных срабатываний»

Исследовать процент ложных срабатываний при добавлении и удалении элементов.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
def calculate_false_positive_rates(cbf, added_items, removed_items,
test_items):
    fp_add = sum(1 for item in test_items if cbf.contains(item))
    fp_add_rate = (fp_add / len(test_items)) * 100 if test_items else 0.0

    fp_remove = sum(1 for item in removed_items if cbf.contains(item))
    fp_remove_rate = (fp_remove / len(removed_items)) * 100 if removed_items
    else 0.0

    return fp_add_rate, fp_remove_rate
```

Пример работы функции представлен на рисунке 7.


```
m = 100
k = 3
cbf = CountingBloomFilter(m, k)

added_items = ["apple", "banana", "cherry", "date", "elderberry"]
removed_items = ["banana", "date"]
test_items = ["fig", "grape", "kiwi", "lemon", "mango"]

for item in added_items:
    cbf.add(item)

for item in removed_items:
    cbf.remove(item)

fp_add_rate, fp_remove_rate = calculate_false_positive_rates(cbf, added_items, removed_items, test_items)
print(f"\nЛожноположительные срабатывания:")
print(f"- Для добавленных элементов: {fp_add_rate:.2f}%")
print(f"- Для удалённых элементов: {fp_remove_rate:.2f}%")
```

✓ 0.0s

Ложноположительные срабатывания:
- Для добавленных элементов: 0.00%
- Для удалённых элементов: 0.00%

Рисунок 7 – работа функции для задачи №2

Данная функция сравнивает результаты проверки с эталонным множеством, вычисляя процент ложных срабатываний. Тестируется на смеси реальных и случайных элементов.

Задача №3 «Анализ зависимости от гиперпараметров»

Оценить зависимость ложноположительных срабатываний от гиперпараметров алгоритма.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
m_values = [1000, 5000, 10000]
k_values = [2, 3, 5, 7]
num_elements = 1000
num_removed = 500
num_test_elements = 10000

added_items = list(range(num_elements))
removed_items = added_items[:num_removed]
test_items = list(range(num_elements, num_elements + num_test_elements))

assert not set(added_items).intersection(test_items), "Тестовые элементы не должны пересекаться с добавленными"

results = []
for m in m_values:
    for k in k_values:
```

```

cbf = CountingBloomFilter(m, k)

for item in added_items:
    cbf.add(item)

for item in removed_items:
    cbf.remove(item)

fp_add, fp_remove = calculate_false_positive_rates(cbf, added_items,
removed_items, test_items)
results.append({
    'm': m,
    'k': k,
    'FP Add (%)': fp_add,
    'FP Remove (%)': fp_remove
})

df = pd.DataFrame(results)

plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x='m', y='FP Add (%)', hue='k', marker='o',
palette='tab10')
plt.title('Зависимость FP при добавлении от m и k')
plt.xlabel('m (размер массива)')
plt.ylabel('FP (%)')
plt.grid(True)
plt.show()

plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x='m', y='FP Remove (%)', hue='k', marker='o',
palette='tab10')
plt.title('Зависимость FP при удалении от m и k')
plt.xlabel('m (размер массива)')
plt.ylabel('FP (%)')
plt.grid(True)
plt.show()

print(df.pivot_table(index='m', columns='k', values=['FP Add (%)', 'FP Remove
(%)']))

```

Пример работы программы представлен на рисунке 8 и 9.

k	FP Add (%)				FP Remove (%)			
	2	3	5	7	2	3	5	7
m								
1000	42.82	52.04	71.40	82.48	44.0	53.6	71.0	81.8
5000	3.45	1.98	1.00	0.88	3.2	1.8	1.6	1.2
10000	0.92	0.24	0.11	0.03	0.8	0.2	0.0	0.0

Рисунок 8 – статистическая таблица для задачи №3

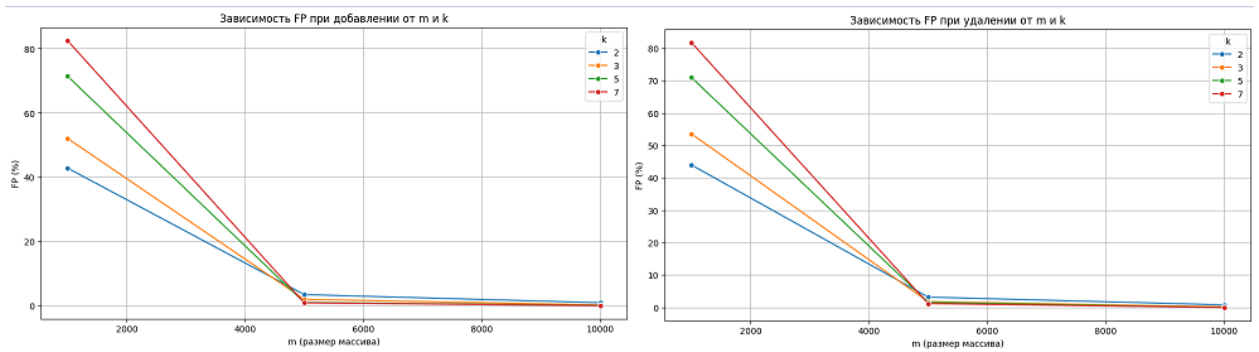


Рисунок 9 – статистические графики для задачи №3

Данная программа анализирует зависимость фильтра Блума со счетчиком от гиперпараметров (размер массива (m) и число хеш-функций (k)). Результаты оптимальных параметров выводятся в виде таблицы и графиков.

Задача №4 «Операции с фильтрами»

Реализовать возможность пересечения и объединения фильтров Блума со счетчиком.

Решение

Проверка возможности выполнения операций на языке *Python* представлена ниже:

```
def union(self, other):
    if self.m != other.m or self.k != other.k:
        raise ValueError("Фильтры должны иметь одинаковые параметры
m и k")
    new_cbf = CountingBloomFilter(self.m, self.k)
    new_cbf.counters = np.maximum(self.counters, other.counters)
    return new_cbf

def intersect(self, other):
    if self.m != other.m or self.k != other.k:
        raise ValueError("Фильтры должны иметь одинаковые параметры
m и k")
    new_cbf = CountingBloomFilter(self.m, self.k)
    new_cbf.counters = np.minimum(self.counters, other.counters)
    return new_cbf
```

Пример работы программы представлен на рисунке 10.

```
> bf1 = CountingBloomFilter(1000, 3)
bf2 = CountingBloomFilter(1000, 3)

bf1.add("a")
bf1.add("c")

bf2.add("b")
bf2.add("c")

bf_union = bf1.union(bf2)
print("Объединение содержит 'a'", bf_union.contains("a"))
print("Объединение содержит 'b'", bf_union.contains("b"))
print("Объединение содержит 'c'", bf_union.contains("c"))
print("")
bf_intersection = bf1.intersect(bf2)
print("Пересечение содержит 'a':", bf_intersection.contains("a"))
print("Пересечение содержит 'b':", bf_intersection.contains("b"))
print("Пересечение содержит 'c':", bf_intersection.contains("c"))
```

144] ✓ 0.0s

```
... Объединение содержит 'a' True
... Объединение содержит 'b' True
... Объединение содержит 'c' True

... Пересечение содержит 'a': False
... Пересечение содержит 'b': False
... Пересечение содержит 'c': True
```

Рисунок 10 – проверка операций для задачи №4

Данная программа осуществляет проверку объединения фильтров Блума со счетчиком путем суммирования соответствующих счётчиков, а также находит их пересечение через минимальные значения счётчиков, требуя одинаковых параметров m (размер массива) и k (число хеш-функций) у обоих фильтров.

Задание №3 «HyperLogLog»

Задача №1 «Реализация HyperLogLog»

Разработать вероятностную структуру данных для оценки мощности множеств с заданной точностью.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
class HyperLogLog:
    def __init__(self, b=10):
        if b < 4 or b > 16:
            raise ValueError("b должно быть в диапазоне от 4 до 16")
        self.b = b
        self.m = 1 << b
        self.registers = [0] * self.m

    def add(self, element):
        hash_value = mmh3.hash64(str(element).encode('utf-8'),
signed=False)[0]
        index = hash_value >> (64 - self.b)
        remaining = hash_value & ((1 << (64 - self.b)) - 1)
        rho = self._count_leading_zeros(remaining)
        if rho > self.registers[index]:
            self.registers[index] = rho

    def _count_leading_zeros(self, w):
        max_bits = 64 - self.b
        if w == 0:
            return max_bits + 1
        return max_bits - w.bit_length() + 1

    def count(self):
        alpha = self._get_alpha()
        sum_inverse = sum(2.0 ** -r for r in self.registers)
        estimate = alpha * (self.m ** 2) / sum_inverse

        if estimate <= 5 * self.m / 2:
            zeros = self.registers.count(0)
            if zeros != 0:
                estimate = self.m * math.log(self.m / zeros)
        elif estimate > (1 << 32) / 30.0:
            estimate = - (1 << 32) * math.log(1 - estimate / (1 << 32))
        return estimate

    def _get_alpha(self):
        if self.b == 4:
            return 0.673
        elif self.b == 5:
            return 0.697
```

```

elif self.b == 6:
    return 0.709
else:
    return 0.7213 / (1 + 1.079 / self.m)

```

Пример работы программы представлен на рисунке 11.

The screenshot shows a Jupyter Notebook cell with the following Python code:

```

hll = HyperLogLog()

items = ["apple", "banana", "apple", "orange", "banana", "kiwi"]

for item in items:
    hll.add(item)

print("Оценка уникальных элементов:", hll.count())
print("Реальное количество:", len(set(items)))

```

Below the code, the output is displayed:

```

[161] ✓ 0.0s
... Оценка уникальных элементов: 4.007832904843586
    Реальное количество: 4

```

Рисунок 11 – работа программы для задачи №1

Данная программа реализует эффективный подсчёт уникальных элементов в потоке данных с заданной точностью.

Задача №2 «Оценка ложноположительных срабатываний»

Исследовать процент ложных срабатываний при добавлении и удалении элементов.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```

def evaluate_hll_error(b_values, num_elements=100000):
    results = []
    for b in b_values:
        hll = HyperLogLog(b)
        elements = [str(i) for i in range(num_elements)]
        for elem in elements:
            hll.add(elem)
        estimated = hll.count()
        error = abs(estimated - num_elements) / num_elements * 100
        results.append({
            'b': b,
            'm': 1 << b,

```

```

        'Estimated': estimated,
        'Error (%)': error
    })
return pd.DataFrame(results)

```

Пример работы функции представлен на рисунке 12.

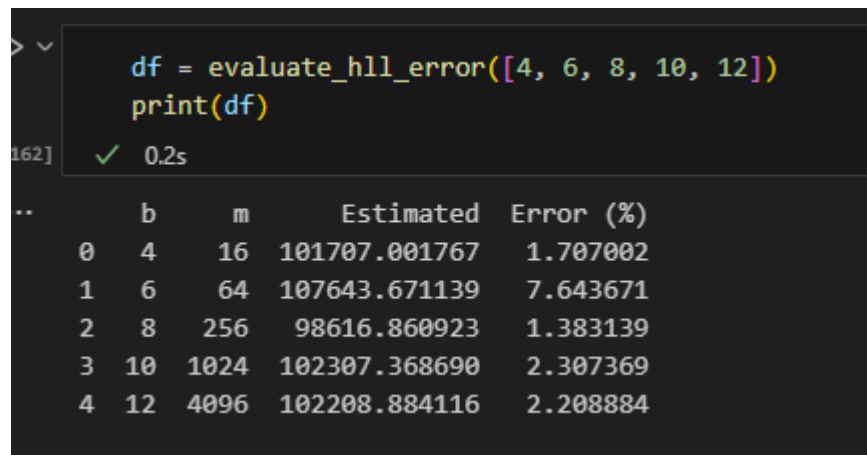


Рисунок 12 – работа функции для задачи №2

Данная функция тестирует точность *HyperLogLog*, сравнивая его оценку кардинальности с реальным количеством уникальных элементов.

Задача №3 «Анализ зависимости от гиперпараметров»

Исследовать влияние точности бит на ложные срабатывания.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```

b_values = [4, 6, 8, 10, 12, 14, 16]
df = evaluate_hll_error(b_values)

plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x='b', y='Error (%)', marker='o')
plt.title('Зависимость ошибки оценки от параметра b')
plt.xlabel('b (количество бит для индекса)')
plt.ylabel('Относительная ошибка (%)')
plt.grid(True)
plt.show()

plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x='m', y='Error (%)', marker='o')
plt.title('Зависимость ошибки оценки от количества регистров (m)')
plt.xlabel('m (количество регистров)')

```

```
plt.ylabel('Относительная ошибка (%)')
plt.grid(True)
plt.show()

print("Таблица зависимости ошибки от гиперпараметров:")
print(df[['b', 'm', 'Error (%)']])
```

Пример работы программы представлен на рисунке 13 и 14.

Таблица зависимости ошибки от гиперпараметров:			
	b	m	Error (%)
0	4	16	1.707002
1	6	64	7.643671
2	8	256	1.383139
3	10	1024	2.307369
4	12	4096	2.208884
5	14	16384	0.220531
6	16	65536	0.802782

Рисунок 13 – статистическая таблица для задачи №3

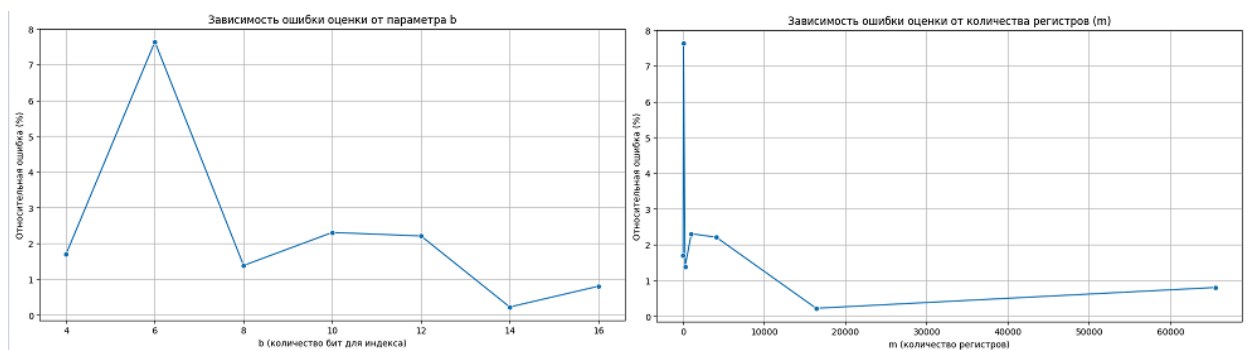


Рисунок 14 – статистические графики для задачи №3

Данная программа анализирует зависимость *HyperLogLog* от гиперпараметров. Результаты оптимальных параметров выводятся в виде таблицы и графиков.

Задание №4 «Фильтр коэффициентов»

Задача №1 «Реализация *Quotient Filter*»

Разработать вероятностную структуру данных для компактного хранения элементов с поддержкой вставки, удаления и проверки принадлежности.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
class QuotientFilter:
    def __init__(self, q, r):
        self.q = q
        self.r = r
        self.size = 1 << q
        self.table = [0] * self.size
        self.occupied = [False] * self.size
        self.continuation = [False] * self.size
        self.run_end = [False] * self.size

    def _hash(self, item):
        hash_bytes = hashlib.sha256(str(item).encode()).digest()
        hash_int = int.from_bytes(hash_bytes, byteorder='big')
        fingerprint = hash_int & ((1 << (self.q + self.r)) - 1)
        return fingerprint >> self.r, fingerprint & ((1 << self.r) - 1)

    def add(self, item):
        q, r = self._hash(item)
        if self.occupied[q]:
            current = self._find_run_start(q)
            while True:
                if self.table[current] == r and not self.continuation[current]:
                    return
                if self.run_end[current]:
                    break
                current = (current + 1) % self.size
            self._shift_and_insert(current, r)
        else:
            self.table[q] = r
            self.occupied[q] = True
            self.run_end[q] = True

    def contains(self, item):
        q, r = self._hash(item)
        if not self.occupied[q]:
            return False
        current = self._find_run_start(q)
        while True:
```

```

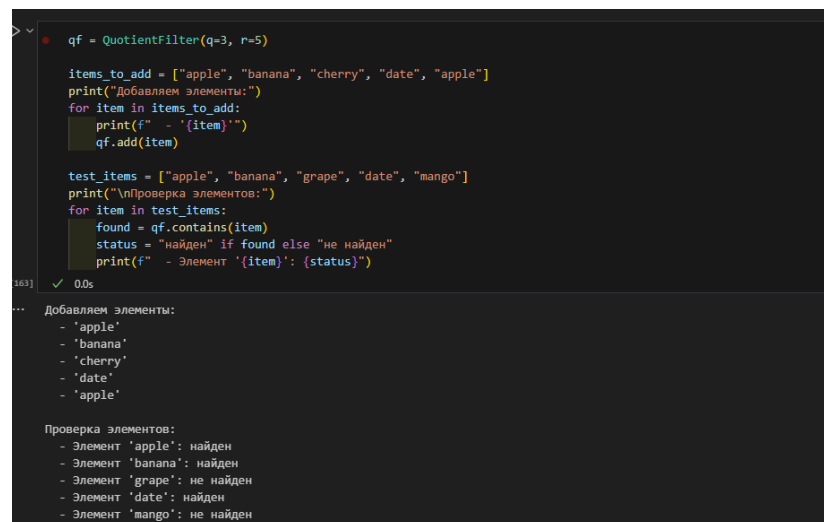
        if self.table[current] == r and not
self.continuation[current]:
            return True
        if self.run_end[current]:
            break
        current = (current + 1) % self.size
    return False

def _find_run_start(self, idx):
    while self.continuation[idx]:
        idx = (idx - 1) % self.size
    return idx

def _shift_and_insert(self, pos, r):
    temp = []
    current = pos
    while not self.run_end[current]:
        temp.append(self.table[current])
        current = (current + 1) % self.size
    temp.append(r)
    current = pos
    for val in temp:
        self.table[current] = val
        self.run_end[current] = False
        current = (current + 1) % self.size
    self.run_end[(current - 1) % self.size] = True

```

Пример работы программы представлен на рисунке 15.



```

qf = QuotientFilter(q=3, r=5)

items_to_add = ["apple", "banana", "cherry", "date", "apple"]
print("Добавляем элементы:")
for item in items_to_add:
    print(f" - '{item}'")
    qf.add(item)

test_items = ["apple", "banana", "grape", "date", "mango"]
print("\nПроверка элементов:")
for item in test_items:
    found = qf.contains(item)
    status = "найден" if found else "не найден"
    print(f" - Элемент '{item}': {status}")

```

Добавляем элементы:

- 'apple'
- 'banana'
- 'cherry'
- 'date'
- 'apple'

Проверка элементов:

- Элемент 'apple': найден
- Элемент 'banana': найден
- Элемент 'grape': не найден
- Элемент 'date': найден
- Элемент 'mango': не найден

Рисунок 15 – работа программы для задачи №1

Данная программа реализует компактное хранение элементов с поддержкой операций вставки, удаления и проверки принадлежности.

Задача №2 «Оценка ложноположительных срабатываний»

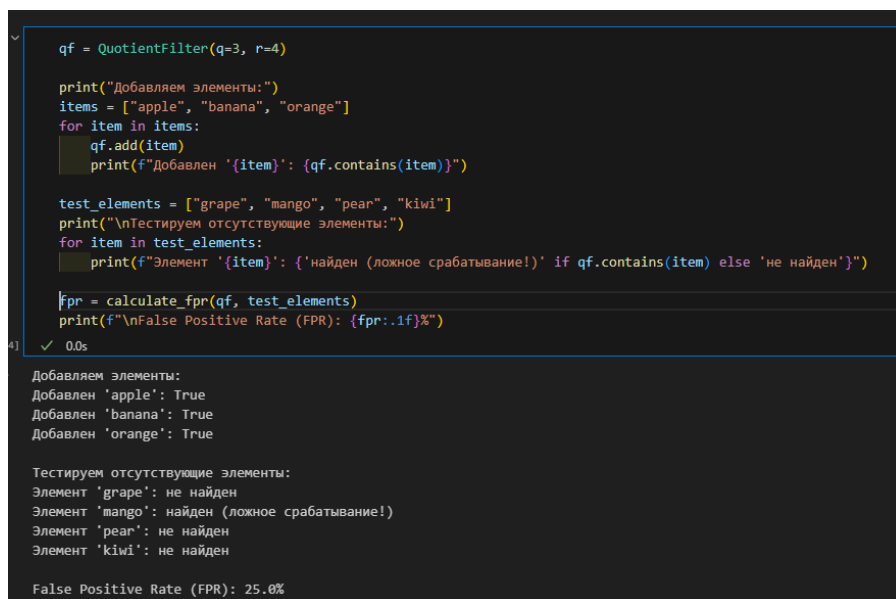
Исследовать процент ложных срабатываний при добавлении элементов.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
def calculate_fpr(qf, test_items):  
    fp = sum(1 for item in test_items if qf.contains(item))  
    return (fp / len(test_items)) * 100
```

Пример работы функции представлен на рисунке 16.



```
qf = QuotientFilter(q=3, r=4)  
  
print("Добавляем элементы:")  
items = ["apple", "banana", "orange"]  
for item in items:  
    qf.add(item)  
    print(f"Добавлен '{item}': {qf.contains(item)}")  
  
test_elements = ["grape", "mango", "pear", "kiwi"]  
print("\nТестируем отсутствующие элементы:")  
for item in test_elements:  
    print(f"Элемент '{item}': {'найден (ложное срабатывание!)' if qf.contains(item) else 'не найден'}")  
  
fpr = calculate_fpr(qf, test_elements)  
print(f"\nFalse Positive Rate (FPR): {fpr:.1f}%")
```

✓ 0.0s

Добавляем элементы:
Добавлен 'apple': True
Добавлен 'banana': True
Добавлен 'orange': True

Тестируем отсутствующие элементы:
Элемент 'grape': не найден
Элемент 'mango': найден (ложное срабатывание!)
Элемент 'pear': не найден
Элемент 'kiwi': не найден

False Positive Rate (FPR): 25.0%

Рисунок 16 – работа функции для задачи №2

Данная функция тестирует *Quotient Filter*, сравнивая его результаты с контрольным множеством для вычисления частоты ложноположительных срабатываний и полноты покрытия добавленных элементов.

Задача №3 «Анализ зависимости от гиперпараметров»

Исследовать влияние частного и остатка на ложные срабатывания.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```

def evaluate_parameters(q_values, r_values, num_elements=1000,
num_tests=5000):
    results = []
    added_items = [f"item_{i}" for i in range(num_elements)]
    test_items = [f"test_{i}" for i in range(num_tests)]

    for q in q_values:
        for r in r_values:
            qf = QuotientFilter(q, r)
            for item in added_items:
                qf.add(item)

            fp_rate = calculate_fpr(qf, test_items)
            results.append({
                'q': q,
                'r': r,
                'FP Rate (%)': fp_rate
            })

    return pd.DataFrame(results)

df = evaluate_parameters(
    q_values=[8, 10, 12],
    r_values=[4, 6, 8]
)

plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x='q', y='FP Rate (%)', hue='r', marker='o')
plt.title("Зависимость FPR от параметров q и r")
plt.xlabel("Биты для квот (q)")
plt.grid(True)
plt.show()

print("Таблица результатов:")
print(df.pivot(index='q', columns='r', values='FP Rate (%)'))

```

Пример работы программы представлен на рисунке 17 и 18.

Таблица результатов:			
r	4	6	8
q			
8	6.12	1.24	0.32
10	3.80	1.00	0.20
12	1.44	0.26	0.10

Рисунок 17 – статистическая таблица для задачи №3

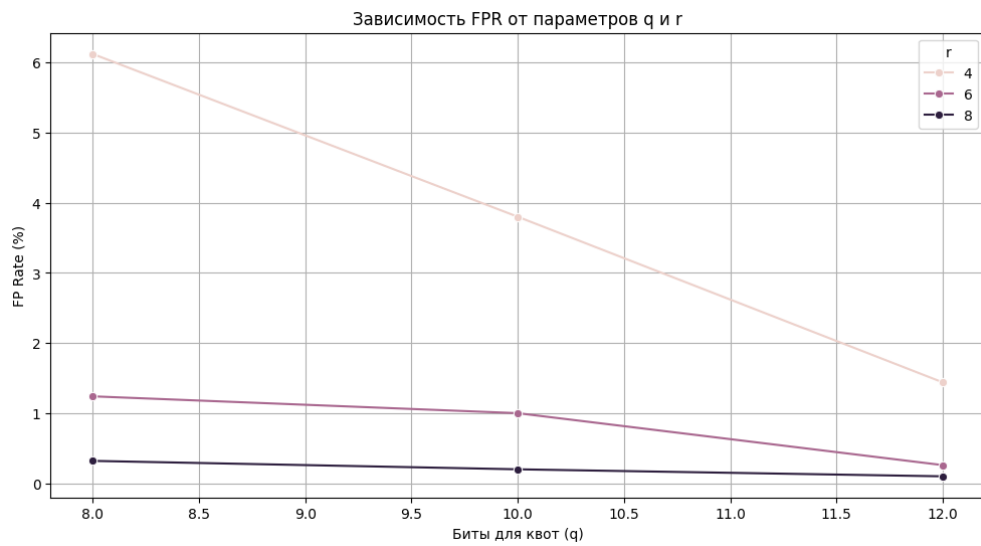


Рисунок 18 – статистический график для задачи №3

Данная программа анализирует зависимость *Quotient Filter* от гиперпараметров. Результаты оптимальных параметров выводятся в виде таблицы и графика.

Задание №5 «Count-Min Sketch»

Задача №1 «Реализация Count-Min Sketch»

Разработать вероятностную структуру данных для оценки частоты элементов в потоке данных с заданной точностью.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
class CountMinSketch:
    def __init__(self, width, depth):
        self.width = width
        self.depth = depth
        self.counters = np.zeros((depth, width), dtype=np.int32)
        self.seeds = [random.randint(0, 2**32 - 1) for _ in range(depth)]

    def _hash(self, item, seed):
        return mmh3.hash(str(item), seed, signed=False) % self.width

    def add(self, item):
        for i in range(self.depth):
            h = self._hash(item, self.seeds[i])
            self.counters[i][h] += 1

    def estimate(self, item):
        return min(
            self.counters[i][self._hash(item, self.seeds[i])]
            for i in range(self.depth)
        )

    def merge(self, other):
        if self.width != other.width or self.depth != other.depth:
            raise ValueError("Count-Min Sketches must have the same dimensions")
        merged = CountMinSketch(self.width, self.depth)
        merged.counters = self.counters + other.counters
        return merged
```

Пример работы программы представлен на рисунке 19.

```
cms = CountMinSketch(width=1000, depth=5)

items = ["apple", "banana", "apple", "orange", "banana", "apple"]

for item in items:
    cms.add(item)

print("\nЧастота элементов:")
test_items = ["apple", "banana", "orange", "grape"]
for item in test_items:
    count = cms.estimate(item)
    print(f"{item}: {count}")
```

✓ 0.0s

Частота элементов:
apple: 3
banana: 2
orange: 1
grape: 0

Рисунок 19 – работа программы для задачи №1

Данная программа реализует компактную вероятностную структуру данных для оценки частоты элементов в потоке данных.

Задача №2 «Оценка ложноположительных срабатываний»

Исследовать процент ложных срабатываний при добавлении элементов.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
def calculate_false_positive_rate(cms, added_set, test_items):
    fp = 0
    for item in test_items:
        if cms.estimate(item) > 0 and item not in added_set:
            fp += 1
    return (fp / len(test_items)) * 100
```

Пример работы функции представлен на рисунке 20.

```
items = ["яблоко", "банан", "яблоко", "вишня"]
true_counts = {"яблоко": 2, "банан": 1, "вишня": 1}
test_items = ["яблоко", "банан", "вишня", "груша", "апельсин"]

width_values = [2, 10]
depth_values = [2]

print("Результаты тестирования ложноположительных срабатываний:")
for width in width_values:
    for depth in depth_values:
        cms = CountMinSketch(width, depth)
        for item in items:
            cms.add(item)

        fp_rate = calculate_false_positive_rate(cms, set(items), test_items)
        print(f"Ширина={width}, Глубина={depth}: Ложные срабатывания={fp_rate:.1f}%")
```

✓ 0.0s

Результаты тестирования ложноположительных срабатываний:
Ширина=2, Глубина=2: Ложные срабатывания=20.0%
Ширина=10, Глубина=2: Ложные срабатывания=0.0%

Рисунок 20 – работа функции для задачи №2

Данная функция тестирует *Count-Min Sketch*, сравнивая оценки частот элементов с эталонными значениями для вычисления точности и частоты ложных срабатываний.

Задача №3 «Анализ зависимости от гиперпараметров»

Исследовать влияние ширины (*width*) и глубины (*depth*) на ложные срабатывания.

Решение

Реализация кода для выполнения данной задачи на языке программирования Python представлена ниже:

```
num_elements = 1000
num_test_elements = 10000
width_values = [100, 500, 1000, 2000]
depth_values = [2, 3, 4, 5]

added_items = list(range(num_elements))
added_set = set(added_items)
test_items = list(range(num_elements, num_elements + num_test_elements))

assert set(test_items).isdisjoint(added_set), "Тестовые элементы не должны быть в добавленных"

results = []
for width in width_values:
    for depth in depth_values:
        cms = CountMinSketch(width, depth)
        for item in added_items:
            cms.add(item)
        fp_rate = calculate_false_positive_rate(cms, added_set, test_items)
        results.append({
            'Width': width,
            'Depth': depth,
            'FP Rate (%)': fp_rate
        })

df = pd.DataFrame(results)

plt.figure(figsize=(12, 8))
sns.lineplot(data=df, x='Width', y='FP Rate (%)', hue='Depth', marker='o')
plt.title('Зависимость ложноположительных срабатываний от ширины и глубины')
plt.xlabel('Ширина (Width)')
plt.ylabel('Процент ложных срабатываний (%)')
plt.grid(True)
plt.show()

pivot_table = df.pivot(index='Width', columns='Depth', values='FP Rate (%)')
print(pivot_table)
```


Пример работы программы представлен на рисунке 21 и 22.

Depth	2	3	4	5
Width				
100	100.00	100.00	100.00	100.00
500	73.05	62.55	57.47	47.85
1000	37.62	26.33	15.44	9.58
2000	15.22	6.25	2.29	1.07

Рисунок 21 – статистическая таблица для задачи №3

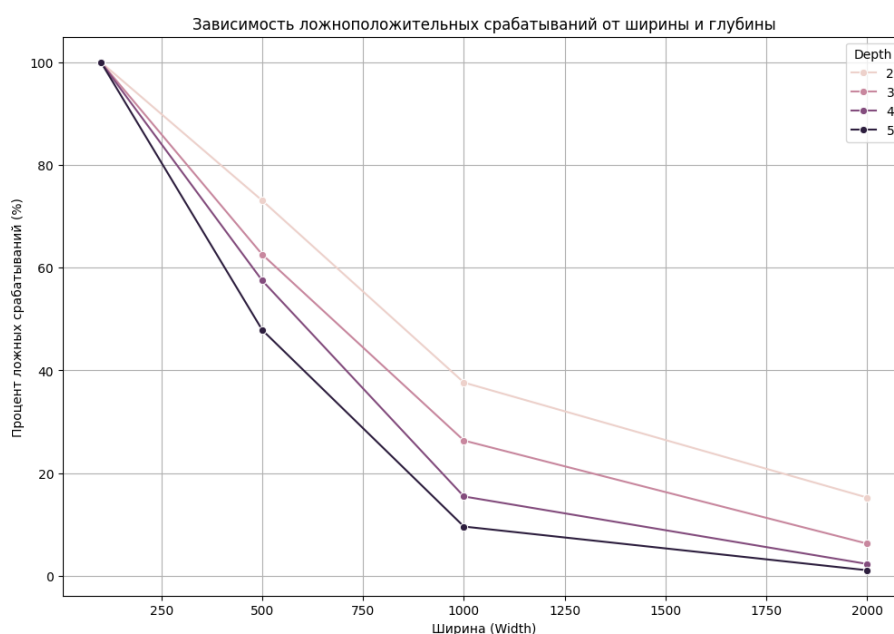


Рисунок 22 – статистический график для задачи №3

Данная программа анализирует зависимость *Count-Min Sketch* от гиперпараметров (ширины (*width*) и глубины (*depth*)). Результаты оптимальных параметров выводятся в виде таблицы и графика.

Задание №6 «MinHash»

Задача №1 «Реализация MinHash»

Разработать вероятностную структуру данных для оценки сходства между множествами.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```
class MinHash:
    def __init__(self, num_hashes):
        self.num_hashes = num_hashes
        self.seeds = np.random.randint(0, 1000000, num_hashes)

    def _hash(self, item, seed):
        hash_value = int(hashlib.md5((str(item) +
str(seed)).encode()).hexdigest(), 16)
        return hash_value

    def compute_signature(self, items):
        signature = np.full(self.num_hashes, np.inf)
        for item in items:
            for i in range(self.num_hashes):
                hash_value = self._hash(item, self.seeds[i])
                if hash_value < signature[i]:
                    signature[i] = hash_value
        return signature

    def similarity(self, set1, set2):
        sig1 = self.compute_signature(set1)
        sig2 = self.compute_signature(set2)
        return np.mean(sig1 == sig2)
```

Пример работы программы представлен на рисунке 23.

```

set1 = {"python", "java", "c++"}
set2 = {"python", "java", "rust"}

minhash = MinHash(3)

similarity = minhash.jaccard_similarity(set1, set2)
print(f"Оценка схожести: {similarity:.2f}")

real_sim = len(set1 & set2) / len(set1 | set2)
print(f"Реальная схожесть: {real_sim:.2f}")

test_pairs = [
    ({"python", "java"}, {"python", "rust"}),
    ({"python", "java"}, {"c++", "rust"}),
    ({"python", "java"}, {"python", "java"}),
]

true_similarities = [0.33, 0.0, 1.0]

fp_rate = minhash.false_positive_rate(test_pairs, true_similarities, threshold=0.5)
✓ 0.0s

```

Рисунок 23 – работа программы для задачи №1

Данная программа реализует компактную вероятностную структуру данных для оценки сходства между множествами.

Задача №2 «Оценка ложноположительных срабатываний»

Исследовать процент ложных срабатываний при добавлении элементов.

Решение

Реализация кода для выполнения данной задачи на языке программирования *Python* представлена ниже:

```

def jaccard_similarity(set1, set2):
    intersection = len(set1.intersection(set2))
    union = len(set1.union(set2))
    return intersection / union if union != 0 else 0

```

Пример работы функции представлен на рисунке 24.

```

set1 = {'яблоко', 'банан', 'апельсин'}
set2 = {'яблоко', 'банан', 'апельсин'}
print(jaccard_similarity(set1, set2))

set3 = {'кошка', 'собака', 'попугай'}
set4 = {'собака', 'попугай', 'хомяк'}
print(jaccard_similarity(set3, set4))

set5 = {'красный', 'зеленый', 'синий'}
set6 = {'круг', 'квадрат', 'треугольник'}
print(jaccard_similarity(set5, set6))

set7 = {'Python', 'Java', 'C++'}
set8 = set()
print(jaccard_similarity(set7, set8))
✓ 0.0s

1.0
0.5
0.0
0.0

```

Рисунок 24 – работа функции для задачи №2

Данная функция тестирует *MinHash*, оценивая частоту ложноположительных срабатываний при определении сходства множеств.

Задача №3 «Анализ зависимости от гиперпараметров»

Исследовать влияние количества хэш-функций (*num_hashes*) на ложные срабатывания.

Решение

Реализация кода для выполнения данной задачи на языке программирования Python представлена ниже:

```
num_hashes_values = [10, 30, 50, 80, 100]
num_elements = 100
num_trials = 20

set1 = set(range(num_elements))
set2 = set(range(num_elements // 2, num_elements + num_elements // 2))

true_jaccard = jaccard_similarity(set1, set2)

results = []

for num_hashes in num_hashes_values:
    errors = []
    for _ in range(num_trials):
        minhash = MinHash(num_hashes)
        estimated_jaccard = minhash.similarity(set1, set2)
        error = abs(estimated_jaccard - true_jaccard)
        errors.append(error)
    avg_error = np.mean(errors)
    results.append((num_hashes, avg_error))

df = pd.DataFrame(results, columns=['Number of Hashes', 'Average Error'])

plt.figure(figsize=(12, 8))
plt.plot(df['Number of Hashes'], df['Average Error'], marker='o')
plt.title('Зависимость ошибки оценки сходства от количества хеш-функций')
plt.xlabel('Количество хеш-функций')
plt.ylabel('Средняя ошибка')
plt.grid(True)
plt.show()

print(df)
```

Пример работы программы представлен на рисунке 25 и 26.

	Number of Hashes	Average Error
0	10	0.133333
1	30	0.058333
2	50	0.059333
3	80	0.035000
4	100	0.047500

Рисунок 25 – статистическая таблица для задачи №3

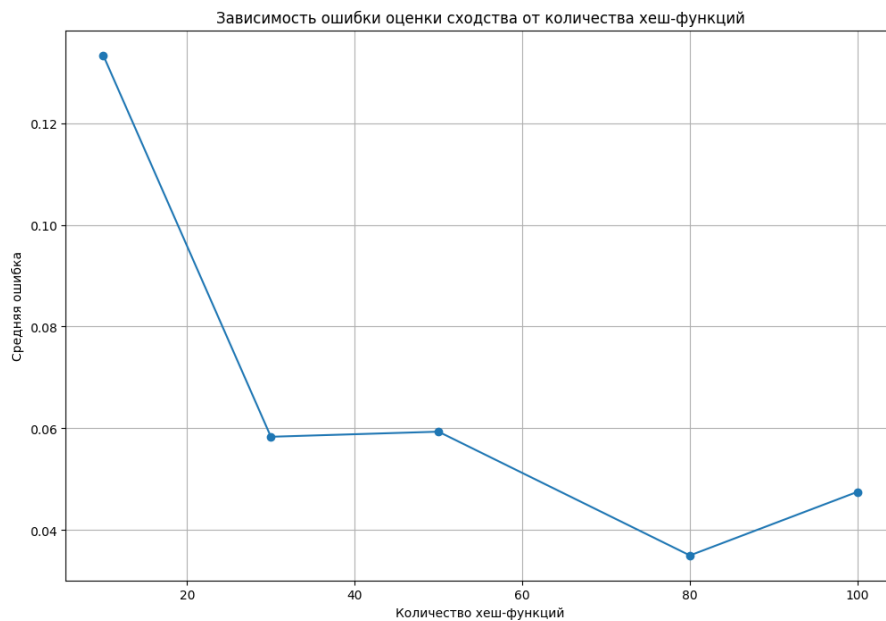


Рисунок 26 – статистический график для задачи №3

Данная программа анализирует зависимость *MinHash* от гиперпараметров. Результаты оптимальных параметров выводятся в виде таблицы и графика.