**ЛАБОРАТОРНАЯ РАБОТА №1**

**«Вероятностные алгоритмы»**

по дисциплине «Практикум по программированию»

Вариант 14

Студента Рау Алексея Евгеньевича

фамилия, имя, отчество полностью

Курс 2 Группа ФИТ-231

Направление 02.03.02 Фундаментальная информатика и информационные технологии

код, наименование

Проверил ассистент

должность

Плескунов Д. А.

фамилия, инициалы

Выполнил

дата, подпись студента

дата, подпись преподавателя

**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.

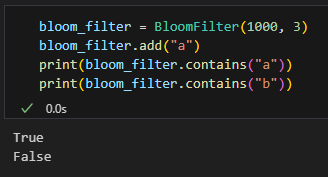


Рисунок 1 – работа программы для задачи №1

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.

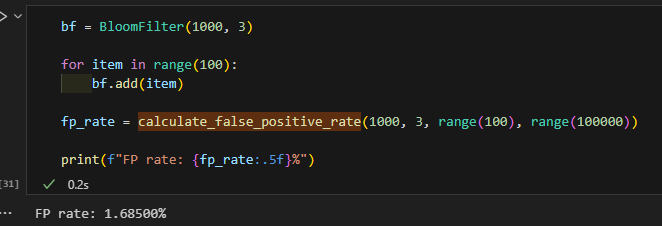


Рисунок 2 – работа функции для задачи №2

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.

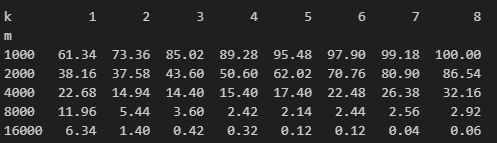


Рисунок 3 – статистическая таблица для задачи №3

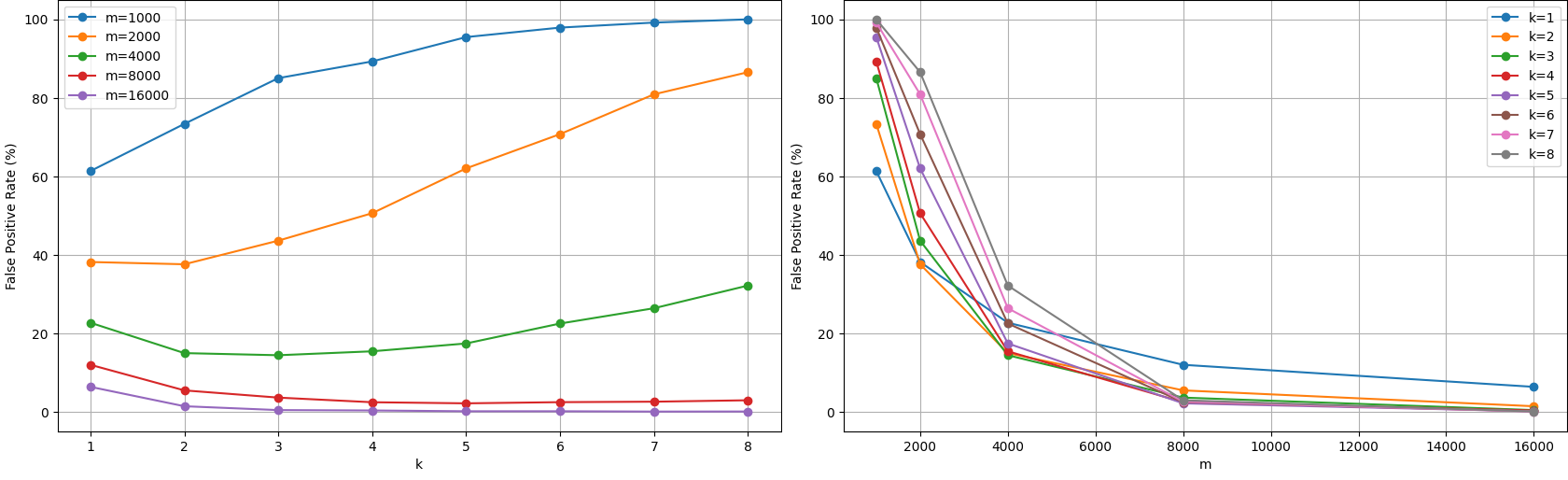


Рисунок 4 – статистические графики для задачи №3

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.



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)        return new\_filter

Пример работы программы представлен на рисунке 6.

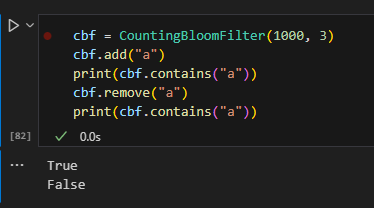


Рисунок 6 – работа программы для задачи №1

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.

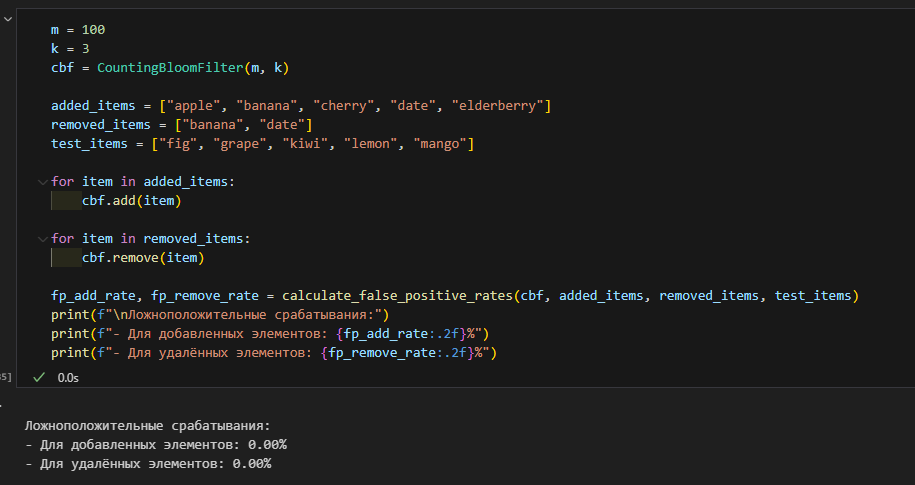


Рисунок 7 – работа функции для задачи №2

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.

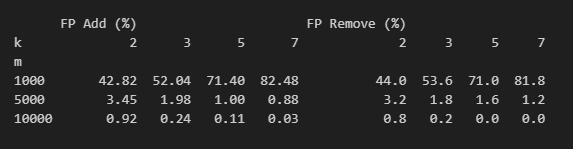


Рисунок 8 – статистическая таблица для задачи №3

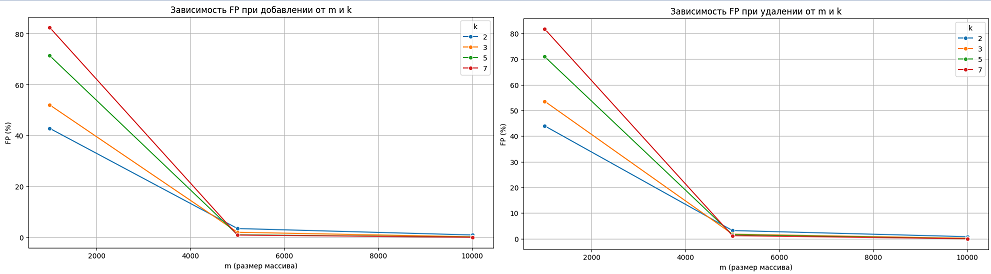


Рисунок 9 – статистические графики для задачи №3

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.

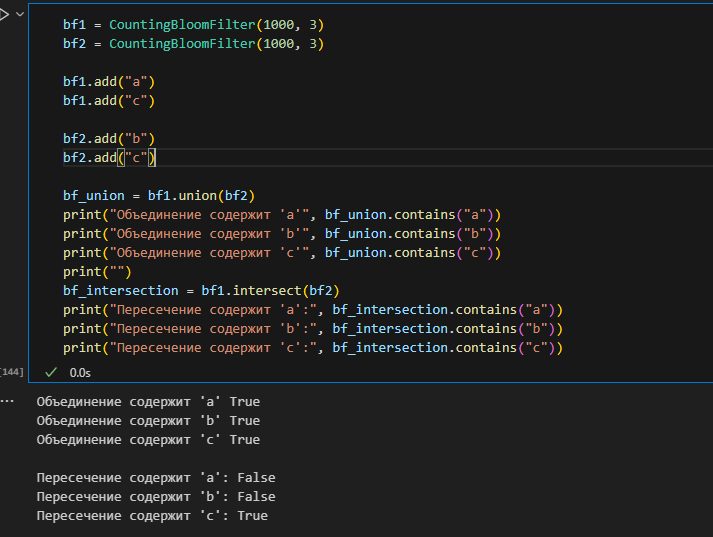


Рисунок 10 – проверка операций для задачи №4

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.

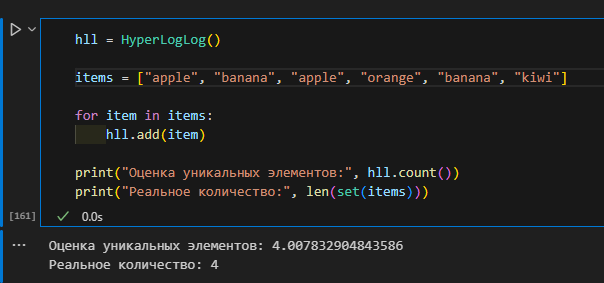


Рисунок 11 – работа программы для задачи №1

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.

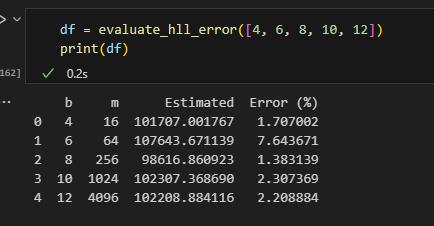


Рисунок 12 – работа функции для задачи №2

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.

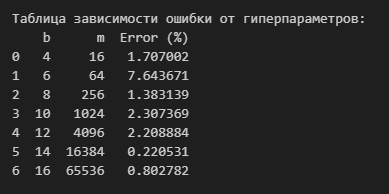


Рисунок 13 – статистическая таблица для задачи №3

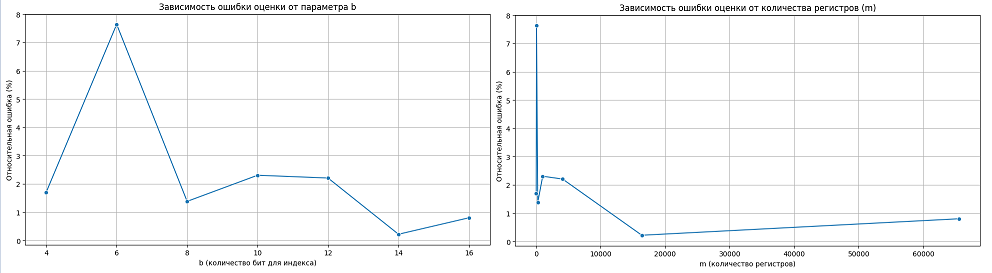


Рисунок 14 – статистические графики для задачи №3

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.

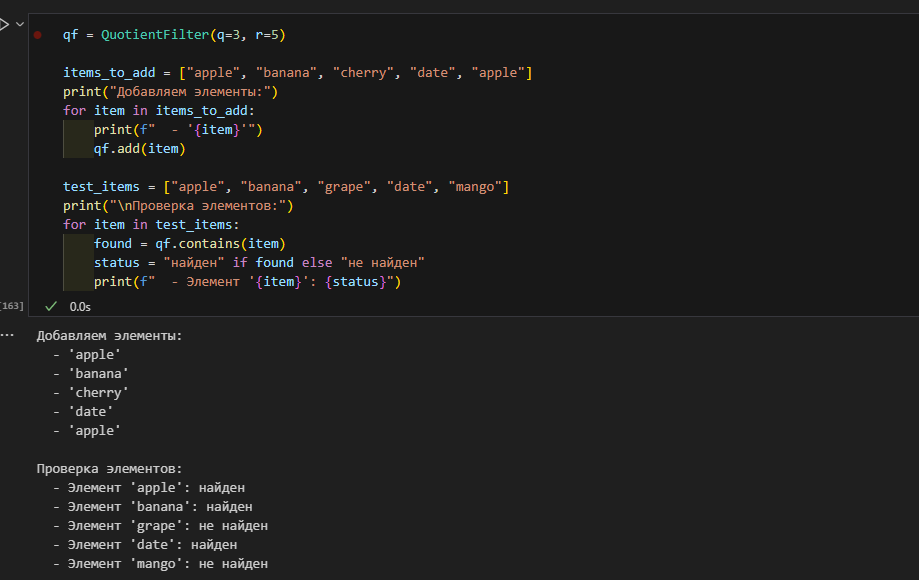


Рисунок 15 – работа программы для задачи №1

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.

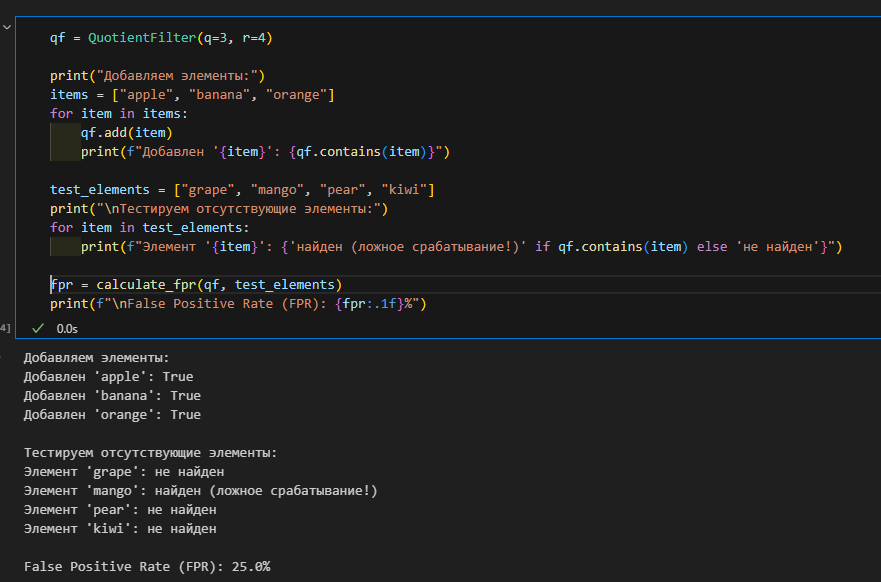


Рисунок 16 – работа функции для задачи №2

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.

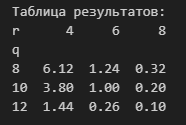


Рисунок 17 – статистическая таблица для задачи №3

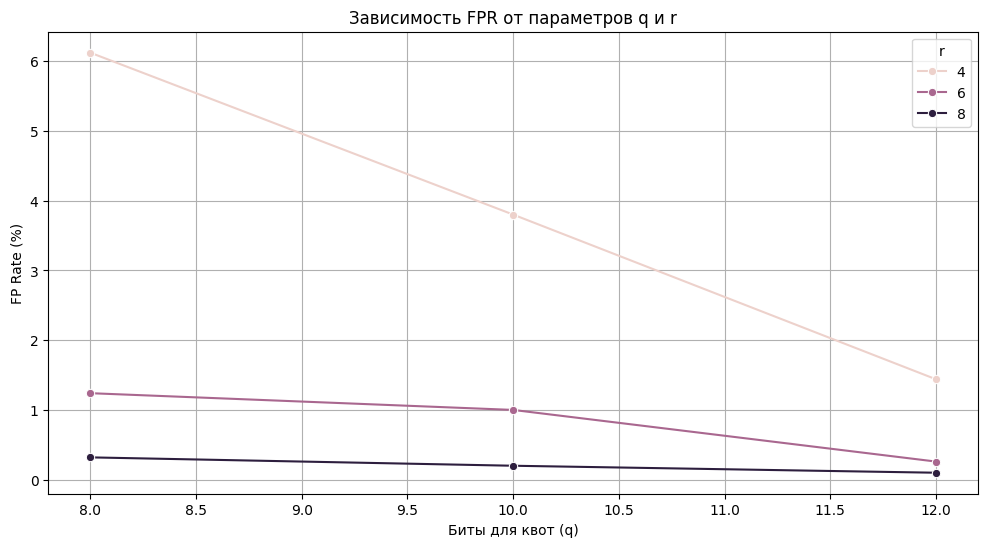


Рисунок 18 – статистический график для задачи №3

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.

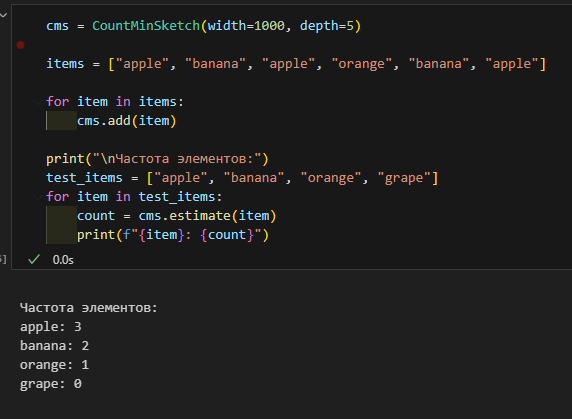
****

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.

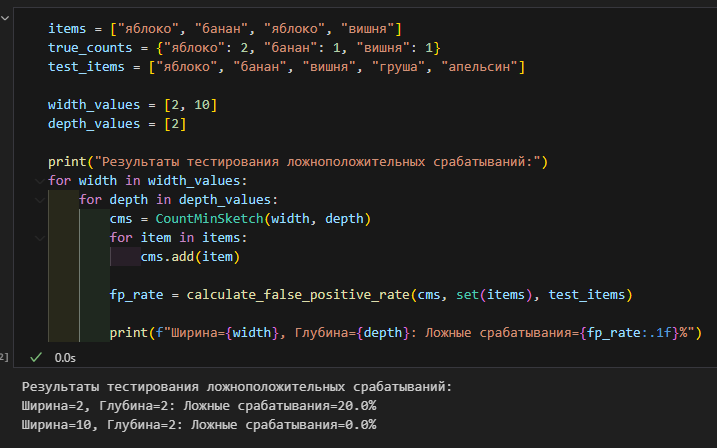


Рисунок 20 – работа функции для задачи №2

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.

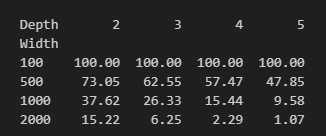


Рисунок 21 – статистическая таблица для задачи №3

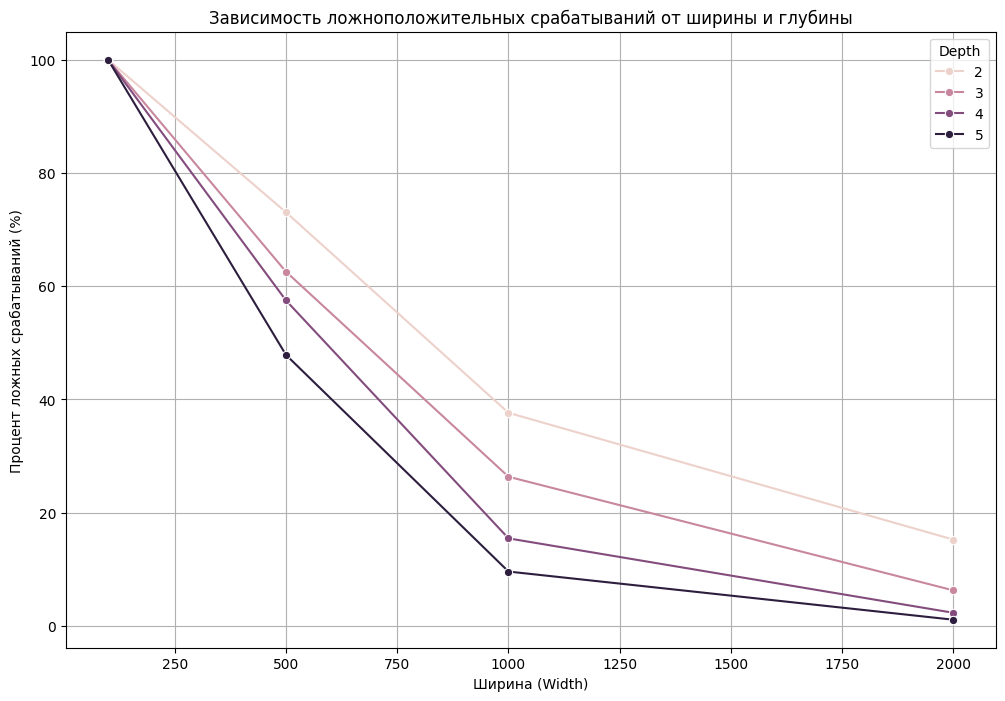


Рисунок 22 – статистический график для задачи №3

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.

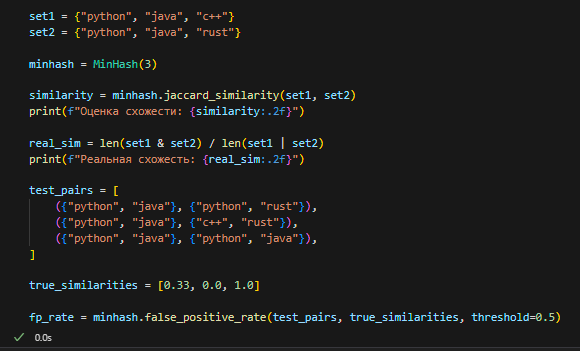
****

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.

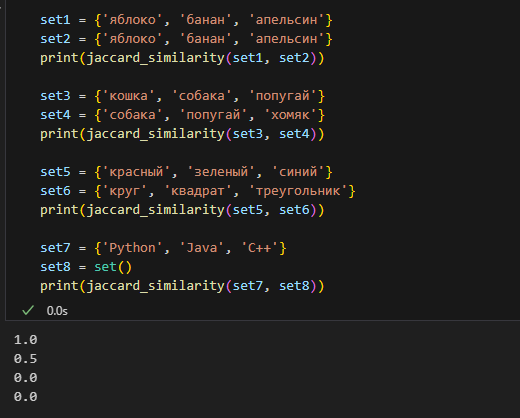


Рисунок 24 – работа функции для задачи №2

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.

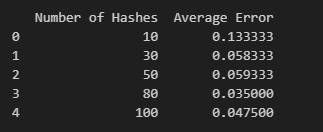


Рисунок 25 – статистическая таблица для задачи №3

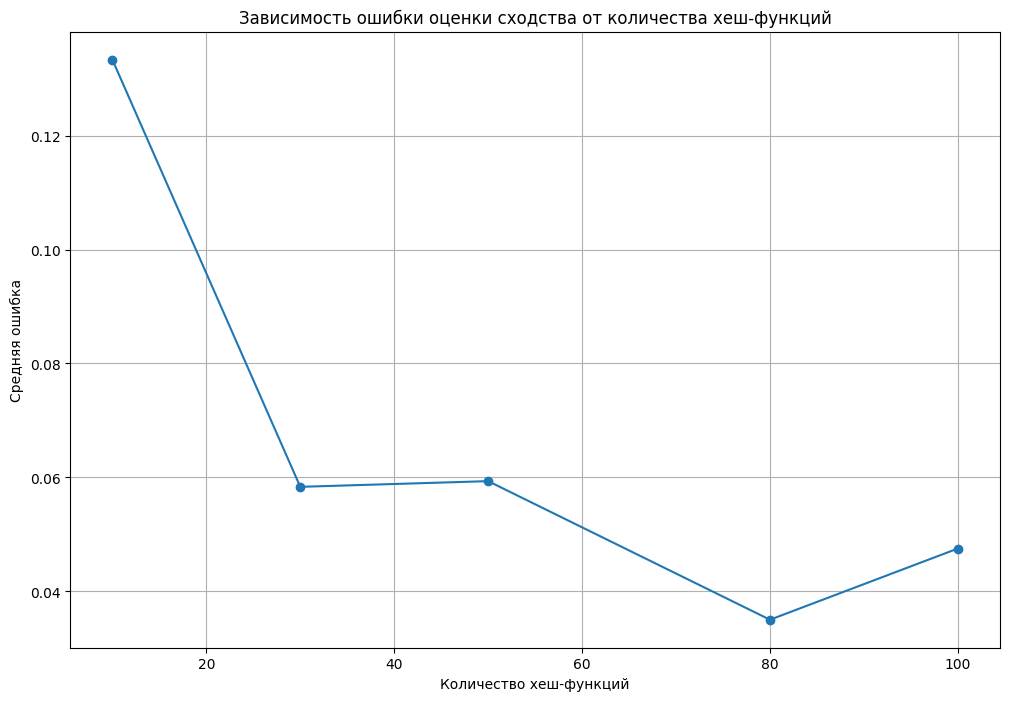


Рисунок 26 – статистический график для задачи №3

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