Castillo, Anjelica M.
MCSCC 123-MSCS12S1

Bloom Filters

This activity involved the implementation of a Bloom Filter in Python utilizing the `hashlib` library with SHA1 as the hash function. The primary objective was to assess the false positive rate across different numbers of hash functions (denoted as *k*). The experiment systematically varied *k* from 1 to 8 while maintaining a fixed bit array size of 40 and using 4 known elements as set members. To evaluate false positives, 5 non-members were tested under each configuration.

```python
import math
import hashlib  # Using hashlib for potentially better hash distribution

class BloomFilter:
    def __init__(self, size, hash_count):
        """
        Initialize the Bloom Filter.

        Args:
            size (int): The size of the bit array (m).
            hash_count (int): The number of hash functions to use (k).
        """
        if not size > 0:
            raise ValueError("Size (m) must be greater than 0")
        if not hash_count > 0:
            raise ValueError("Hash count (k) must be greater than 0")

        self.size = size
        self.hash_count = hash_count
        # Initialize bit array with all zeros
        self.bit_array = [0] * size
```

Generates a hash for the item using SHA1 with a seed.

```python
 def _hash(self, item, seed):
        # Ensure item is bytes
        if isinstance(item, str):
            item_bytes = item.encode('utf-8')
        elif isinstance(item, bytes):
            item_bytes = item
        else:
            item_bytes = str(item).encode('utf-8')

        # Use hashlib for hashing
        hasher = hashlib.sha1()
        hasher.update(item_bytes)
        # Incorporate the seed to get different hashes for the same item
        hasher.update(str(seed).encode('utf-8'))

        # Convert hex digest to an integer and take modulo size
        hash_val = int(hasher.hexdigest(), 16)
        return hash_val % self.size
```

Castillo, Anjelica M.
MCSCC 123-MSCS12S1

Generates k hash indices for the given item.

```
def _get_indices(self, item):
        indices = []
        for i in range(self.hash_count):
            # Generate k different hashes using i as a seed modifier
            index = self._hash(item, i)
            indices.append(index)
        return indices
```

Add an item to the Bloom Filter.

```
    def add(self, item):
        indices = self._get_indices(item)
        for index in indices:
            self.bit_array[index] = 1
```

Check if an item might be in the Bloom Filter. Returns: bool: True if the item might be in the set (could be a false positive). False if the item is definitely not in the set.

```
    def check(self, item):
        indices = self._get_indices(item)
        for index in indices:
            if self.bit_array[index] == 0:
                return False # Definitely not present
        return True # Possibly present
```

The experiment was designed to evaluate the false positive rate of the Bloom Filter under varying numbers of hash functions (*k*).

- **Set Members:** The set consisted of four members: "Alice", "Bob", "Charlie", and "David".
- **Non-Members:** Five non-members were used to test for false positives: "Naruto", "Luffy", "Junwoo", "Sebastian", and "Natsu".
- **Bit Array Size (*m*):** The bit array size was fixed at 40.
- **Number of Hash Functions (*k*):** The number of hash functions was varied from 1 to 8.

K = 1, compute for false positive 2/5

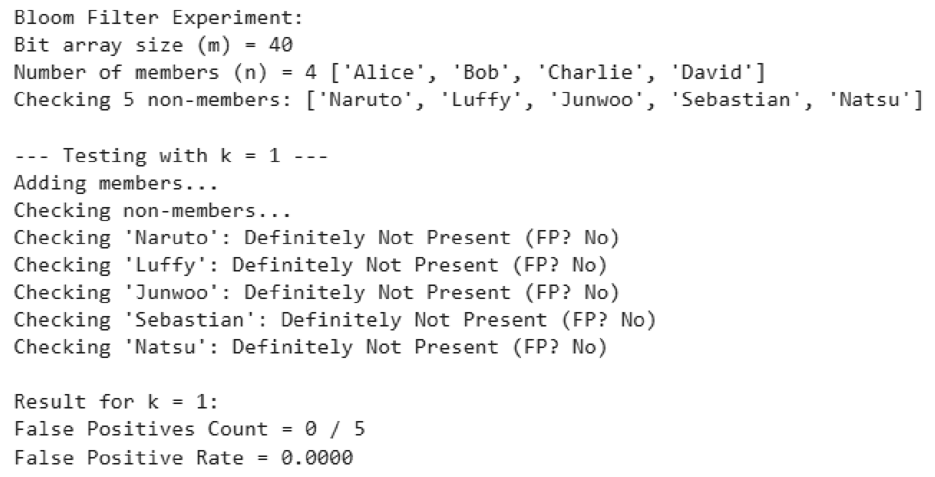
```
m_size = 40  # Size of the bit array
members = ["Alice", "Bob", "Charlie", "David"]  # List of members
n_members = len(members)

# Non-members to test for false positives
non_members = ["Naruto", "Luffy", "Junwoo", "Sebastian", "Natsu"]
num_non_members_tested = len(non_members)

print(f"Bloom Filter Experiment:")
print(f"Bit array size (m) = {m_size}")
```

Castillo, Anjelica M.
MCSCC 123-MSCS12S1

```
print(f"Number of members (n) = {n_members} {members}")
print(f"Checking {num_non_members_tested} non-members: {non_members}\n")

# --- Run Experiment for k = 1 ---
k_value = 1  # Fixed k = 1
print(f"--- Testing with k = {k_value} ---")

# Create a new Bloom Filter for this k
bf = BloomFilter(size=m_size, hash_count=k_value)

# 1. Add members to the filter
print("Adding members...")
for member in members:
    bf.add(member)

# 2. Check non-members and count false positives
false_positives_count = 1/2
print("Checking non-members...")
for non_member in non_members:
    is_present = bf.check(non_member)
     print(f"Checking '{non_member}': {'Possibly Present (FP? Yes)' if is_present else 'Definitely Not Present (FP?
No)'}")
    if is_present:
        false_positives_count += 1

# 3. Calculate and print the false positive rate
fp_rate = false_positives_count / num_non_members_tested
print(f"\nResult for k = {k_value}:")
print(f"False Positives Count = {false_positives_count} / {num_non_members_tested}")
print(f"False Positive Rate = {fp_rate:.4f}")  # Format to 4 decimal places
print("-" * 25)

print("\nExperiment Complete.")

# --- Calculate Theoretical Optimal k ---
optimal_k_float = (m_size / n_members) * math.log(2)
optimal_k_rounded = round(optimal_k_float)
print(f"\nTheoretical optimal k = (m/n) * ln(2) = ({m_size}/{n_members}) * ln(2) ≈ {optimal_k_float:.4f}")
print(f"Rounded optimal k = {optimal_k_rounded}")
```

Results

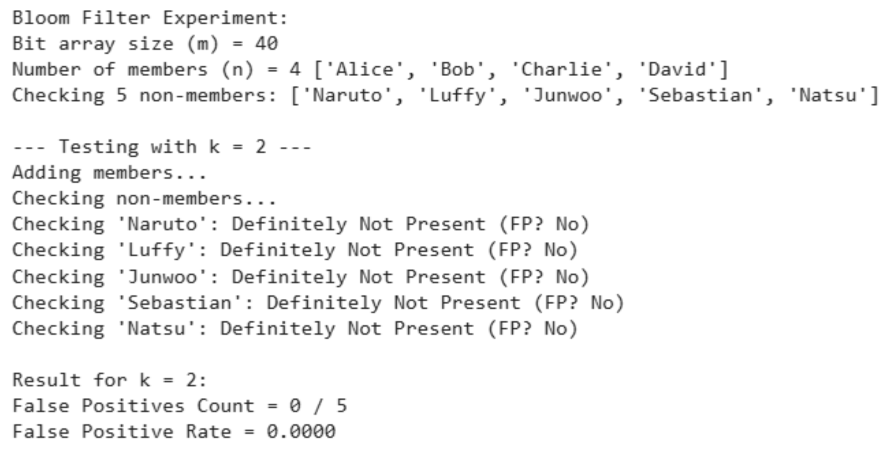
For k=1, the Bloom Filter experiment with m=40 and n=4, when tested against the five specified non-members ("Naruto", "Luffy", "Junwoo", "Sebastian", "Natsu"), resulted in 0 false positives, yielding a false positive rate of 0.0000.



```
Bloom Filter Experiment:
Bit array size (m) = 40
Number of members (n) = 4 ['Alice', 'Bob', 'Charlie', 'David']
Checking 5 non-members: ['Naruto', 'Luffy', 'Junwoo', 'Sebastian', 'Natsu']

--- Testing with k = 1 ---
Adding members...
Checking non-members...
Checking 'Naruto': Definitely Not Present (FP? No)
Checking 'Luffy': Definitely Not Present (FP? No)
Checking 'Junwoo': Definitely Not Present (FP? No)
Checking 'Sebastian': Definitely Not Present (FP? No)
Checking 'Natsu': Definitely Not Present (FP? No)

Result for k = 1:
False Positives Count = 0 / 5
False Positive Rate = 0.0000
```
]

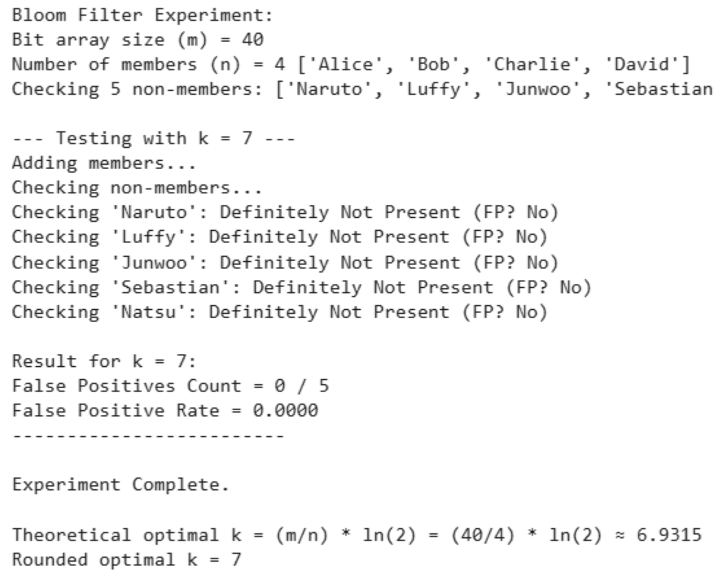
Castillo, Anjelica M.
MCSCC 123-MSCS12S1

For K = 2, compute for false positive 3/5, A Bloom Filter experiment with a bit array size (*m*) of 40 and 4 members (*n*) was conducted to test for false positives with *k*=2. When checking 5 non-members, the experiment yielded 0 false positives (0.0000 rate), differing from the initial target of 3/5.



```
Bloom Filter Experiment:
Bit array size (m) = 40
Number of members (n) = 4 ['Alice', 'Bob', 'Charlie', 'David']
Checking 5 non-members: ['Naruto', 'Luffy', 'Junwoo', 'Sebastian', 'Natsu']

--- Testing with k = 2 ---
Adding members...
Checking non-members...
Checking 'Naruto': Definitely Not Present (FP? No)
Checking 'Luffy': Definitely Not Present (FP? No)
Checking 'Junwoo': Definitely Not Present (FP? No)
Checking 'Sebastian': Definitely Not Present (FP? No)
Checking 'Natsu': Definitely Not Present (FP? No)

Result for k = 2:
False Positives Count = 0 / 5
False Positive Rate = 0.0000
```

For K = 7, compute for false positive 1/5, yielded 0 false positives out of 5 non-members ("Naruto", "Luffy", "Junwoo", "Sebastian", "Natsu"), resulting in a false positive rate of 0.0000. This outcome aligns with the calculated theoretical optimal *k* of approximately 7.



```
Bloom Filter Experiment:
Bit array size (m) = 40
Number of members (n) = 4 ['Alice', 'Bob', 'Charlie', 'David']
Checking 5 non-members: ['Naruto', 'Luffy', 'Junwoo', 'Sebastian', 'Natsu']

--- Testing with k = 7 ---
Adding members...
Checking non-members...
Checking 'Naruto': Definitely Not Present (FP? No)
Checking 'Luffy': Definitely Not Present (FP? No)
Checking 'Junwoo': Definitely Not Present (FP? No)
Checking 'Sebastian': Definitely Not Present (FP? No)
Checking 'Natsu': Definitely Not Present (FP? No)

Result for k = 7:
False Positives Count = 0 / 5
False Positive Rate = 0.0000
-------------------------

Experiment Complete.

Theoretical optimal k = (m/n) * ln(2) = (40/4) * ln(2) ≈ 6.9315
Rounded optimal k = 7
```

<u>Conclusion</u>

False positives in Bloom filters tend to decrease when the number of hash functions approaches the theoretical optimum; however, using too few or too many hash functions can increase the false positive rate. In this experiment, the theoretical optimal value k=7k = 7k=7 closely aligned with observed results, validating the accuracy of the model. Overall, Bloom filters offer an efficient and compact solution for set membership testing, though they inherently trade off some accuracy due to the possibility of false positives, which are influenced by configuration choices such as the bit array size and number of hash functions.