Designing Experiments using Bloom Filters

C. Ian Philpot

Microsoft Corp.

Author Note

Reviewed by: James Taylor - Brenau Univ., Danny Garber - Microsoft

Table of Contents

Abstract

With the proliferation of computing resources, machine learning models, and tooling many today do not spend time understanding the domains and factors that make up the underlying data used to create a modern artificial intelligence. The cloud has not only abstracted away many of the fundamental technologies but is also removing the need for prioritizing the factors or variables that we experiment with. A lot of trial and error work goes into creating models used to predict, decide, or classify entities in our problem domains. While there will always be some of this guess work to boot strap experimentation, a better understanding of the factors that make up our problem space, their importance, and interactions can help us to train models or just better solve problems faster and at a lower cost.

Keywords: Experimentation, Machine Learning, Artificial Intelligence, Bloom Filters

Designing Experiments using Bloom Filters

In today's computing landscape utility computing and layers of abstraction reduce the complexity of creating scientific, advanced decision, or prediction software. This leads to a lack of understanding how these work in modern computing. In this repo our focus is on the underlying data and factors that go into creating machine learning models. Most of the work creating an ML model is spent scrubbing the data and getting it into a shape that it can be processed. This data is then used to train a model. Due to the utility and ubiquity of computing resources today, there is not a lot of thought going into how the data is best used while training a model.

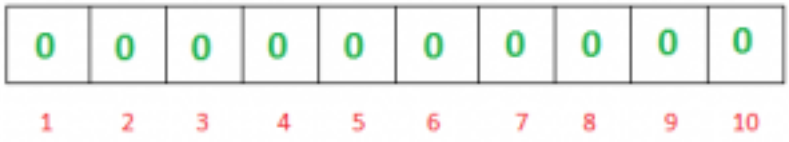
Through a simple, statistically based approach to experimentation we can better understand the factors that make up our data. This will give better understanding to how and why are models are either working or failing. The advanced nature that makes up modern algorithms and computing infrastructure it takes to understand much of the work being done today, we will focus our work on Bloom Filters. This probabilistic data structure is very well understood and gives us a known target to demonstrate how to create experiments that can be applied to many of the problems that face us today. These experiments are only to explore the process of experimentation and not necessarily to learn anything new about this data structure.

Bloom Filters

Bloom filters are a probabilistic data structure created in 1970 by Burton Bloom. It is used to test if an entity is found within a set. False Positive matches can and do occur, but false negatives do not. This means for the user that an item is possibly in the set, or it is not. Because of this, it has very specific use cases, but can drastically increase the speed or scale of a system (Wikipedia).

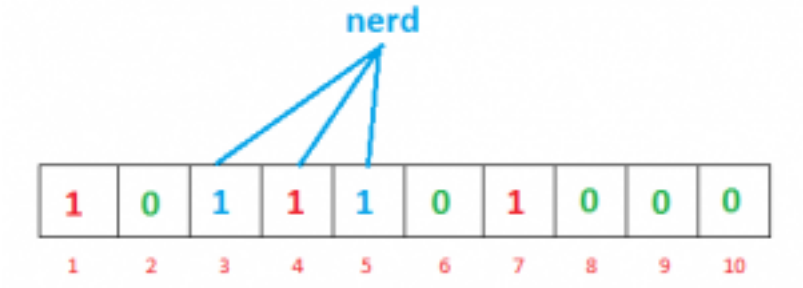
Bloom Filter Internals

An empty bloom filter is a bit array of size “m” all set to zero.

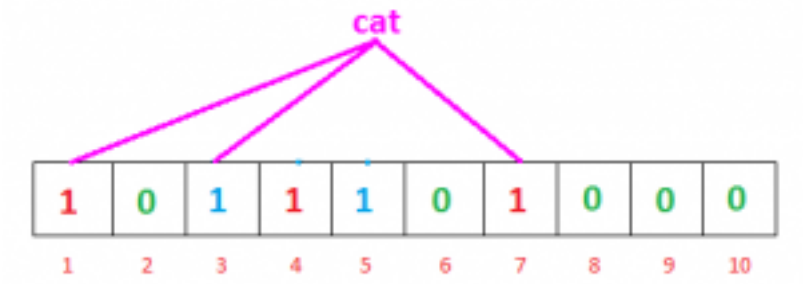


When an item is added it is hashed “k” number of times. In this example k = 3.

More items go through the same process.



Finally, when a look up is done, some items can cause false positives due to collisions in hash functions.



```python
class BloomFilter(object):
    def __init__(self, size, hash_passes):
        self.size = size
        self.bit_array = bitarray(self.size)
        self.bit_array.setall(False)
        self.hash_passes = hash_passes

    def __len__(self):
        return self.size

    def __iter__(self):
        return iter(self.bit_array)

    def add(self, item):
        item = item.strip()
        for i in range(self.hash_passes):
            h = mmh3.hash(item, i) % self.size
            self.bit_array[h] = True

    def __iadd__(self, item):
        self.add(item)
        return self

    def count(self):
        return self.bit_array.count()

    def check(self, item):
        digest=[]
        item = item.strip()
        for i in range(self.hash_passes):
            h = mmh3.hash(item, i) % self.size
            present = self.bit_array[h]
            if not present:
                return False
            else:
                digest.append(present)
                if len(digest) == self.hash_passes:
                    return True

    def __contains__(self, item):
        return self.check(item)
```

Figure 1: Bloom Filter Class

The code in Figure 1 shows both how to insert items (add method) and how to query for items (check method). These are referenced by the "__iadd__" and "__contains__" or special methods in python. This is to follow pythonic convention and enable code reuse.

Bloom Filter Factors

There are three main factors we can use to efficiently and effectively store and test for items in a set using Bloom Filters.

- Probability of false positives
    - Let false positives be "p"

$$P = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k$$

```
def false_positive_probability(self, m, k, n):
    '''
    input:
        m = bit array size
        k = number of hash passes
        n = number of items
    output:
        p = false positive probability
    '''

    p = (1-(1-1/m)**(k*n))**k
    return float('{0:.4f}'.format(p))
```

Figure 2: False Positive Probability

- Size of the bit array
    - Let bit array size be “m”

$$m = -\frac{n \ln P}{(ln2)^2}$$

```
def bit_array_size(self, n, p):      You, 13
    '''
    input:
        n = number of items
        p = false positive probability
    output:
        m = bit array size
    '''

    m = -(n * math.log(p)) / (math.log(2)**2)
    return int(m)
```

Figure 3: Bit array size

- Number of hash functions or passes
    - Let hash function pass count be “k”

$$k = \frac{m}{n} ln2$$

```
def hash_pass_count(self, m, n):
    '''
    input:
        m = bit array size
        n = number of items
    output:
        k = number of hash passes
    '''

    k = (m/n) * math.log(2)
    return int(k)
```

Figure 4: Hash Pass Count

These equations are used to understand the settings needed to configure a bloom filter to store "x" number of items (GeeksforGeeks).

Another factor that we are able to control is the type of hash function used. Our experiments hold this constant using the mmh3 pip package to provide this functionality.

Description of Variables in Experiments

For our experiment we will be inserting 10,000 items (usernames) into our bloom filter. Using the above equations, we can right size our bloom filter with the following factors:

1. 10,000 Items inserted
2. 1,000 Absent items to test with
3. Bit array size: 62352
4. Hash pass count: 4
5. Target false positive rate: 0.05

```
# Here we configure the bloom filter using optimal settings for 10,000 items.
# We then test using a list of absent users and display the results. Since we
# have 1,000 absent users and an expected .05 false positive probability, then
# we should get 50 false positives. This simply proves that our calculations and
# bloom filter implementation is accurate.
def main():
    # Present file contains 10,000 generated usernames that are added to the bloom filter.
    present_users_file = './src/resources/present.txt'

    # Absent file contains 1,000 generated usersnames that are not in the bloom filter.
    absent_users_file = './src/resources/absent.txt'

    # Read files into models
    present_users = PresentUsers(present_users_file)
    absent_users = AbsentUsers(absent_users_file)

    # Create bloom filter based on calculations for right-sizing to
    # 10,000 items and a 0.05 false positive rate.
    bloom_filter = BloomFilter(62352, 4)      You, 13 days ago • DOE experiments started

    # Add present users to the bloom filter.
    for i in range(len(present_users)):
        bloom_filter += present_users[i]

    # Test for absent users and count the false positives.
    false_positive_count = 0
    for user in absent_users:
        if user in bloom_filter:
            false_positive_count += 1

    print('There are {} false positives for {} absent users, or {} false positive probability'
        .format(false_positive_count, len(absent_users), false_positive_count/len(absent_users)))

if __name__ == '__main__':
    main()
```

Figure 5: Right Sized Bloom Filter Test

One Factor at a Time Experiments

A common strategy of experimentation is One Factor at a Time (OFAT) type of experiments. The experimenter selects a starting point, or baseline, for one of the controllable factors then varying it while holding all other factors constant. After all the experiments are completed a set of graphs are made to show how the response variable changes. “The major disadvantage of the OFAT strategy is that it fails to consider any possible interaction between the factors.” (Montgomery, D. C.).

An interaction causes the response variable to differ when testing a controllable variable at varied levels of the other controllable variables. Understanding this and the effect that this interaction has on our experiments, will help us focus on the factors that are most important when training machine learning models or solving complex problems using other data analysis methods.

```
# Loop over a specified range of ints to adjust how many hash passes the bloom filter
# performs for each item added.
cnt_passes = []
cnt_fp = []
for hash_count in range(1, 10):

    # Bloom filter right sized to bit array size for 10,000 items, but adjusted hash
    # passes
    bloom_filter = BloomFilter(62352, hash_count)

    # Add present users to the bloom filter.
    for i in range(len(present_users)):
        bloom_filter += present_users[i]

    # Test for absent users and count the false positives.
    false_positive_count = 0
    for user in absent_users:
        if user in bloom_filter:
            false_positive_count += 1

    # Add result for graph
    cnt_passes.append(hash_count)
    cnt_fp.append(false_positive_count)

    print('There are {} false positives when hash pass count is {}'
        .format(false_positive_count, hash_count))

# Create and show graph
plt.plot(cnt_passes, cnt_fp)
plt.show()
```

Figure 6: OFAT Hash Pass Count

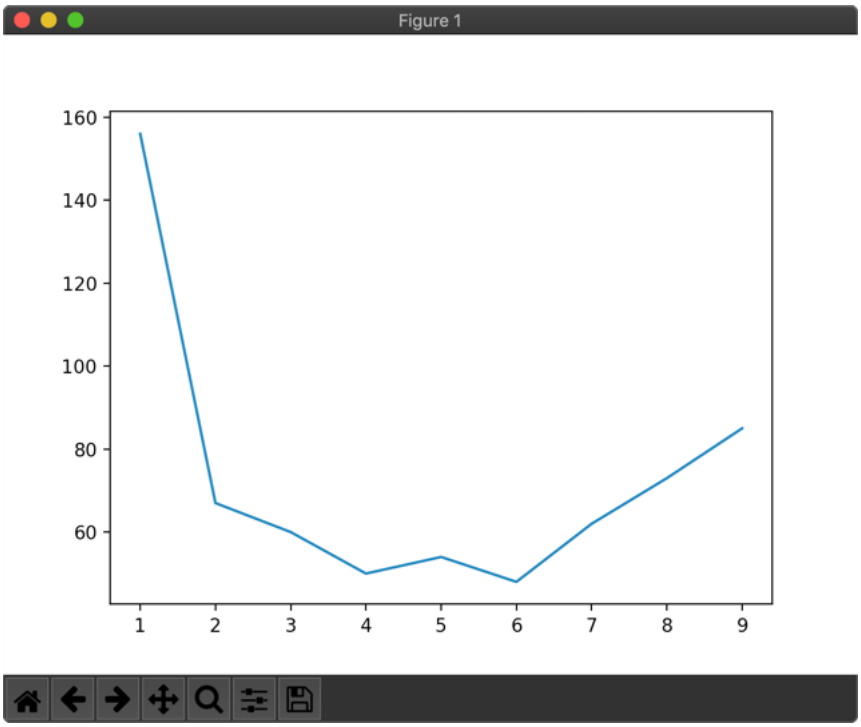


Figure 7: OFAT Hash Pass Count Effect

Figure 1 shows a code snippet where we are conducting a OFAT experiment varying only the count of hash passes used to insert items into the bloom filter's bit array. Figure 2 is a graph of the effect this has on the response variable: false positives. From this we see that when all other variables are fixed, false positives are minimized at 4 and 6 hash passes. This is suspect because when other factors are changed, we may get different results and interactions are not detectable.

Factorial Experiment

The goal of a factorial experiment is to "turn the knobs" of all the factors being tested and understand the effect and interactions each have on one another. The major advantage is that using these techniques we can utilize all of the data gathered to determine how best to work with the factors when building machine learning models or solving problems considered complex.

For our Bloom Filter, we are designing a 2-factor factorial experiment. Our controllable variables will consist of the Bit Array size and Hash pass count. All other variables will be held constant using the right-size configuration for our bloom filter. I would like to point out that bloom filters and their factors are a known quantity. As mentioned previously, these experiments are only to explore the process of experimentation and not necessarily to learn anything new about this data structure.

```python
#!/usr/bin/env python

from model.bloom_filter import BloomFilter
from model.users import PresentUsers, AbsentUsers
import matplotlib.pyplot as plt
import numpy as np
import math

# This is the last experiment, a 2 factorial designed experiment. The goal is to understand
# the mass effect of each variable, but also the interaction between the to variables we control
# and how they affect the respons variable (false positives).
def main():

    # Present file contains 10,000 generated usernames that are added to the bloom filter.
    present_users_file = './src/resources/present.txt'

    # Absent file contains 1,000 generated usersnames that are not in the bloom filter.
    absent_users_file = './src/resources/absent.txt'

    # Read files into models
    present_users = PresentUsers(present_users_file)
    absent_users = AbsentUsers(absent_users_file)

    # Loop over a specified range of ints to adjust both the bit array size
    # and the hash pass count for the bloom filter. M Range is 50,000 to 70,000 with
    # a step of 10,000. This should surround the right sized value of 62352. k range is 3 to
    # 5 and also should surround the right sized value of 4.
    # TODO: O(n^2) - refactor to more be efficient using a memoization pattern.
    cnt_size = []
    cnt_passes = []
    cnt_fp = []
    for hash_count in range(3, 5):
        for bit_arr_size in range(50000, 70000, 10000):

            # Bloom filter with varying values for both hash passes and bit array sizes
            # for 10,000 items
            bloom_filter = BloomFilter(bit_arr_size, hash_count)

            # Add present users to the bloom filter.
            for i in range(len(present_users)):
                bloom_filter += present_users[i]

            # Test for absent users and count the false positives.
            false_positive_count = 0
            for user in absent_users:
                if user in bloom_filter:
                    false_positive_count += 1

            cnt_fp.append(false_positive_count)
            cnt_passes.append(hash_count)
            cnt_size.append(bit_arr_size)

            print('There are {} false positives when bit array size is {} and hash count is {}'
                .format(false_positive_count, bit_arr_size, hash_count))
```

Figure 8: 2-Factor Factorial

The code found in Figure 8 is in the file 05_doe_fact_all.py and will drive the experiment. Since we already have the answers to what the settings are for right-sized bloom filter, we establish the factor experimental levels so as to gain the greatest amount of information

with the fewest number of experimental runs. For hash pass counts we use 3 and 4. For bit array size we use 50,000 and 60,000. In this experiment we will only do one pass. There is no variation in the response variable between runs, so we know the output is statistically significant. Otherwise we would employ additional techniques to assess if the results are statistically significant.

Figure 9: Cube plot showing response variable

The cube plot in Figure 9 shows a 2-factor experiment. The plot has each factor at different levels and each combination shown. This creates a 2-d square where a 3-factor experiment would create a cube. The different combinations for further analysis will be derived from this plot.

To understand the main effect that the hash pass counts have on our experiment we will take the average of these and plot them. The data from our cube plot would look something like:

Figure 10: Hash Pass Main Effect Cube Data

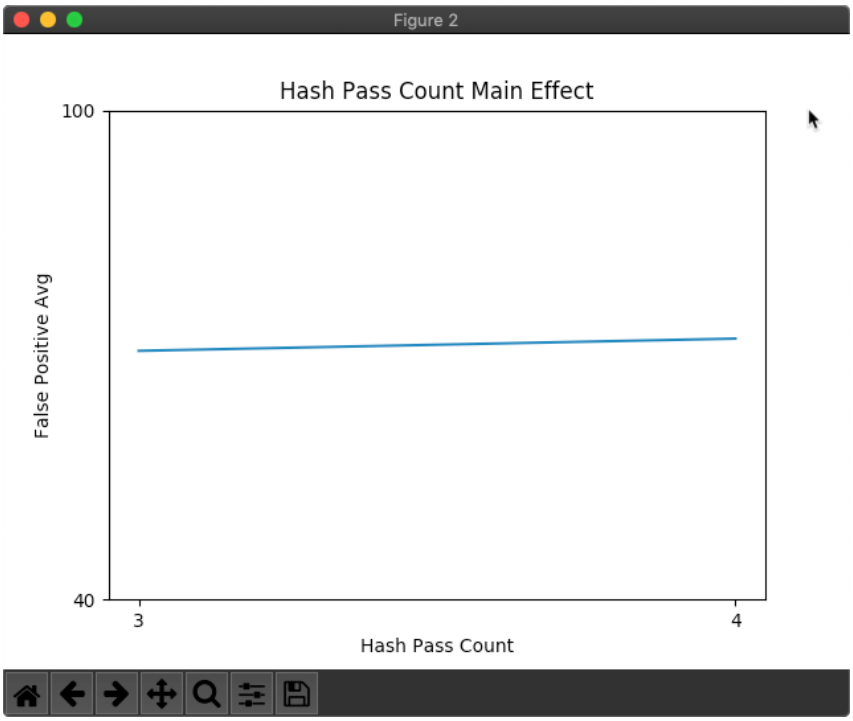


Figure 111: Main Effect Graph of Hash Pass Count

The main effect for hash passes is 1.5

This is found by taking the averages of the two runs and finding the difference:

([83+58]/2)-([95+49]/2) = 1.5

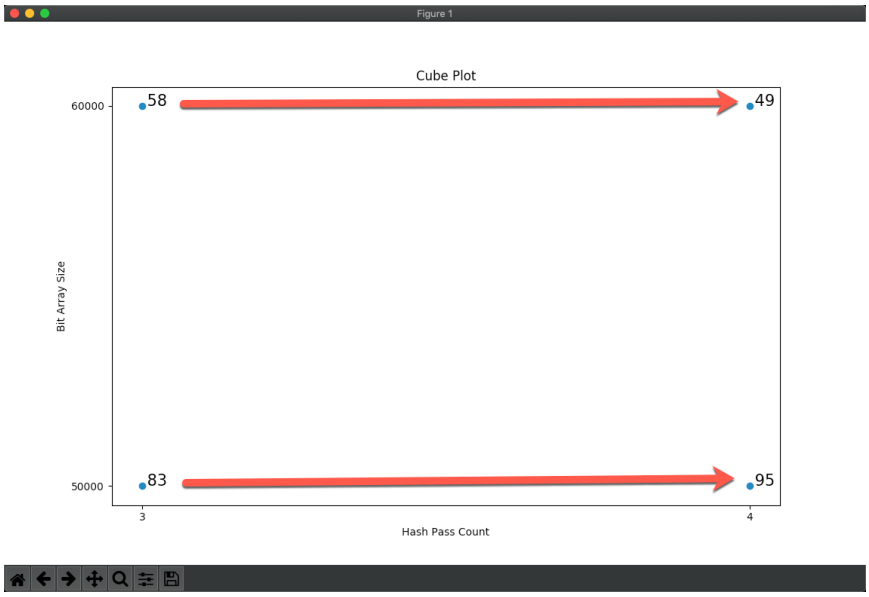


Figure 122: Bit Array Main Effect Cube Data

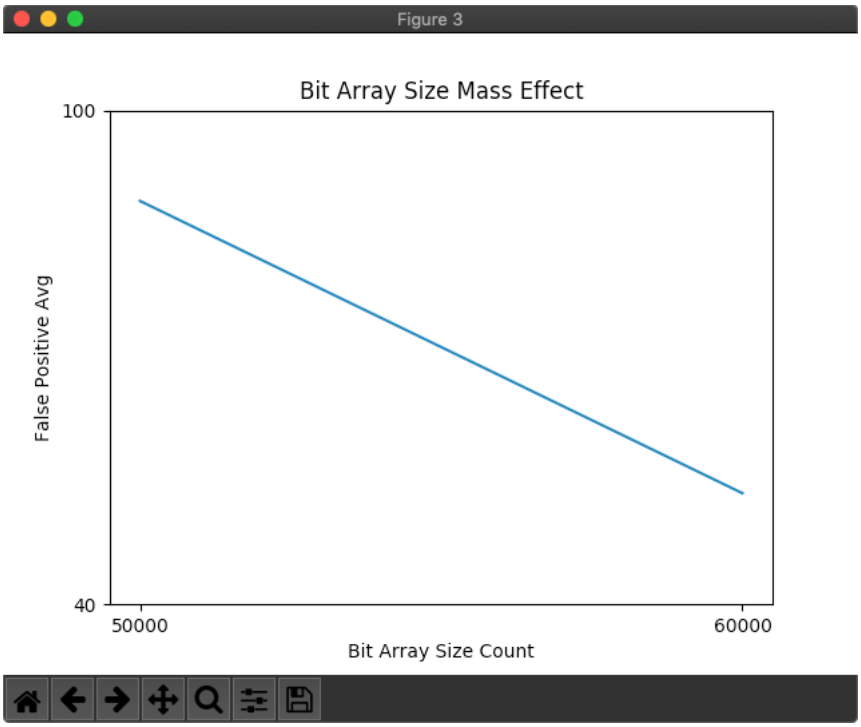


Figure 133: Main Effect Graph of Bit Array Size

The main effect for bit array size is 35.5

Similarly, this is found by taking the averages of the two runs and finding the difference: ([83+95]/2)-([58+49]/2) = 35.5.

From these two graphs we see that both effect the response variable, but that the bit array size has a more significant effect.

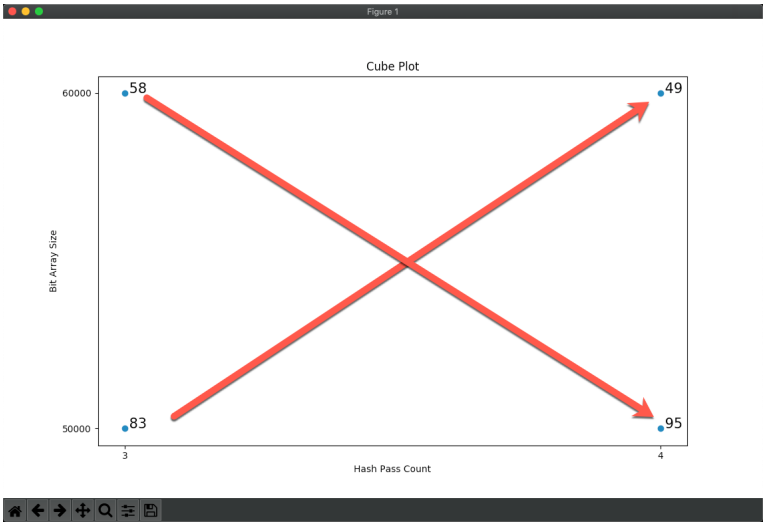


Figure 144: Interaction Effect Cube Plot Data

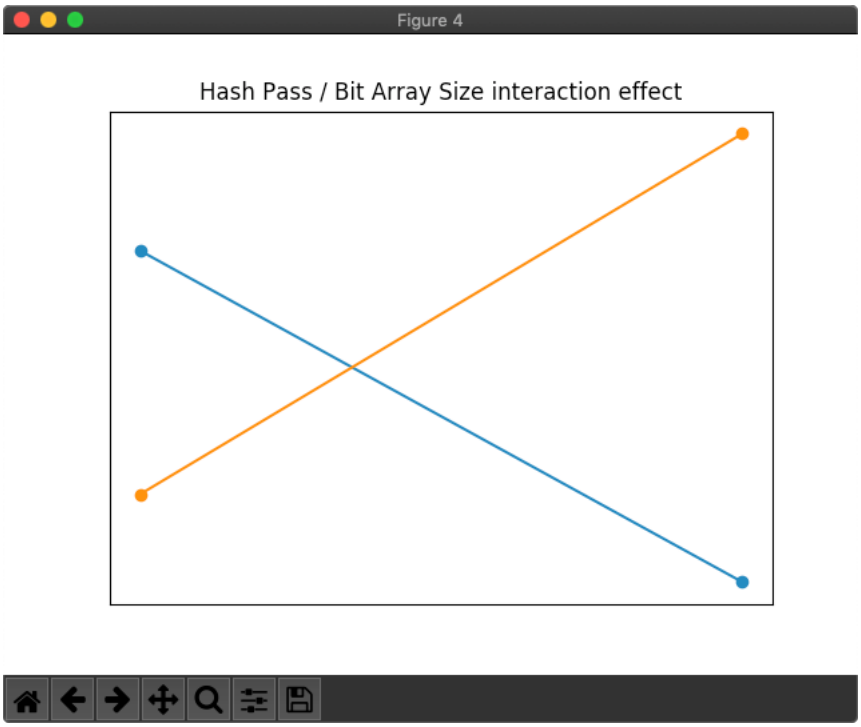


Figure 155: Interaction Effect Graph

The Hash Pass / Bit Array Size interaction effect is 10.5

The hash pass / bit array size interaction effect is obtained by taking the average false positives from left to right diagonal: ([83+49]/2)-([58+95]/2) = 10.5

From this we see that there is an interaction between the two as we'd expect. 10.5 is considered significant and would direct further experimentation to focus on the interaction. Each factor has an effect, but they also depend on the level of other factors. From this we see that when 4 hash passes are deployed, false positives are minimized with a 60000 bit array. Using these values and knowledge, we can create further experiments to find the optimal settings for each factor. Adjusting the "knobs" in relation to the main and interaction effects found.

Conclusion

The idea for this study is not to use Bloom Filters for anything other than simply a data structure to experiment with. We could've used anything in its place, but it turned out to be a great example. And fun to learn about.

The main point is how we modeled the experiment, the results, and the findings they drove. Ultimately bit array size (m) is more important than hash passes (k) and the interaction of both (m + k) is very significant. This supports the known properties of Bloom filters.

Applying this to an ML model or complex problems, we can decide what are the more important factors and how they interact, or if there is no interaction between the factors. This gives us an efficient starting point for training models or writing algorithms. Ultimately, we can hypothesize that we can reduce overall training time and costs using this method.

References

Montgomery, D. C. (2017) *Design and Analysis of Experiments, 9th Edition.* John Wiley & Sons

Footnotes

[1][Add footnotes, if any, on their own page following references. For APA formatting requirements, it's easy to just type your own footnote references and notes. To format a footnote reference, select the number and then apply the Footnote Reference. The body of a footnote, such as this example, uses the Normal text style. *(Note: If you delete this sample footnote, don't forget to delete its in-text reference as well.)*]

Tables

Table 1

[Table Title]

Column Head	Column Head	Column Head	Column Head	Column Head
Row Head	123	123	123	123
Row Head	456	456	456	456
Row Head	789	789	789	789
Row Head	123	123	123	123
Row Head	456	456	456	456
Row Head	789	789	789	789

Note: [Place all tables for your paper in a tables section, following references (and, if applicable, footnotes). Start a new page for each table, include a table number and table title for each, as shown on this page. All explanatory text appears in a table note that follows the table, such as this one. Use the Table/Figure style to get the spacing between table and note. Tables in APA format can use single or 1.5 line spacing. Include a heading for every row and column, even if the content seems obvious. To insert a table, on the Insert tab, tap Table. New tables that you create in this document use APA format by default.]

Figures

Figure 1. [Include all figures in their own section, following references (and footnotes and tables, if applicable). Include a numbered caption for each figure. Use the Table/Figure style for easy spacing between figure and caption.]

For more information about all elements of APA formatting, please consult the *APA Style Manual, 6th Edition*.