# Thesis

## 1 OVERVIEW

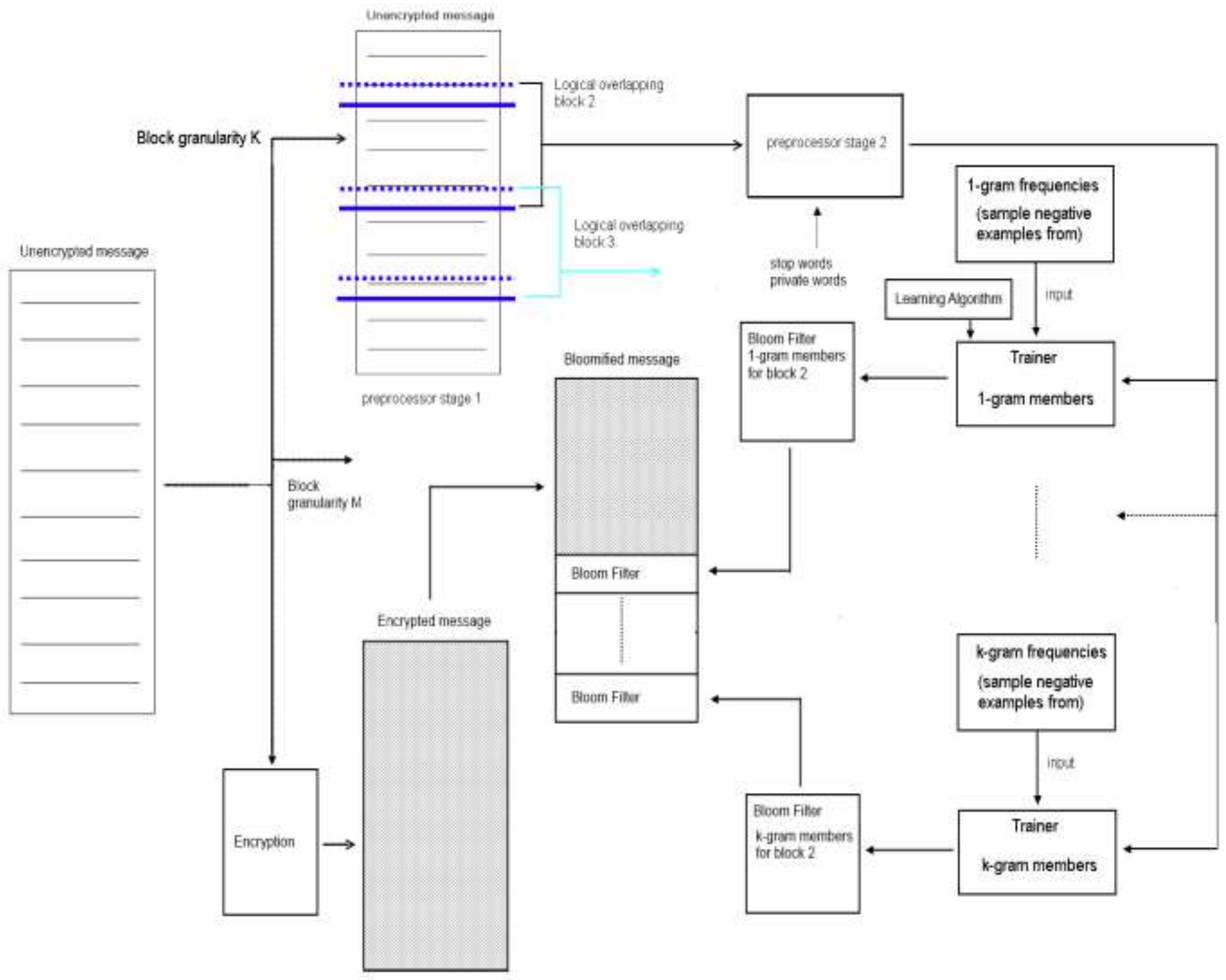


*Figure 1 Flow of Bloomified Message Construction*

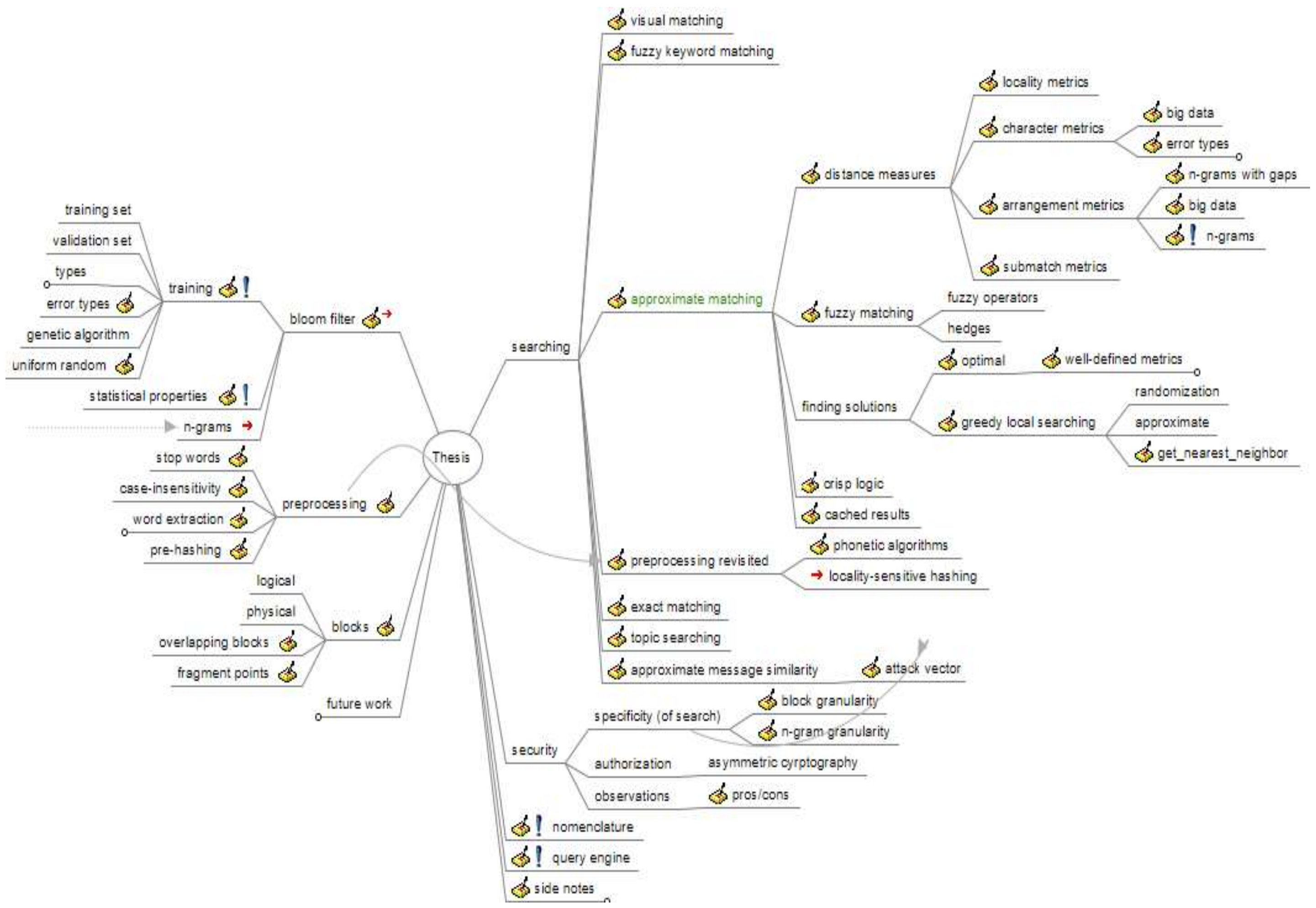
Figure 1 shows how the flow of data goes through the various modules to construct a bloomified message, which stores an encrypted [and compressed?] document along with a set bloom filters (which may consist of multiple hierarchical levels, both in the form of blocks and n-grams). These bloom filters serve as an approximate representation of the document.

The primary reason for using a hash to represent the document: the document in question is encrypted. We do not want to reveal its contents. A hash only maps words to indices; from this, you can only determine, at best, which words (or n-grams) are contained in the document. You cannot determine where they are located in the document, nor the relative ordering of words.

Bloom filters are used, rather than regular hash tables, for the following: The number of members that can be added to the set is enormous, especially if we are also including 1-grams, 2-grams, …, n-grams (for more precise query matching). Bloom filters have the advantageous property that they can be relatively small compared to the size of the space they represent. This is especially true if we are able to train them effectively to have a very small probability of returning a false positive by minimizing the conditional probability of a false positive given a probable search query. Formally, we want to minimize P[false positive | probable n-grams], because that is what most effectively minimizes P[false positive] = P[false positive | negative example]P[negative example]. More on this later.

## 2 MIND MAP

Here's an overview of all of the modules.



## 3 PREPROCESSING

In general, this is a grammar parser (e.g., context-free grammar or regular expression grammar), which determines how the input message is transformed before being fed into the bloom filter set. But, for the sake of convenience, this can be broken down into a few modules.

By default, it is defined as: *preprocess(in) = extract_words(remove_stop_words(remove_private_words(lower_case(in))))*. But it can be anything, including the identity function: *preprocess(in) = in*.

Note that each bloom filter can, in theory, go through a different preprocessing function. See: preprocessing revisited. Whatever grammar is used by the preprocessing engine, it should be saved and associated with the message so that the query engine can apply the same transformations to search queries on the message.

### 3.1 STOP WORDS

Stop words, or patterns, are words like "the" and "and"; they carry no meaningful information so they are removed. Note that this can also be an n-gram, e.g., remove “hello world” but not “hello” or “world” by themselves.

### 3.2 PRESERVE WORDS

Words, or patterns, in this list should not be transformed.

### 3.3 PRIVATE WORDS

Technically, this can be addressed by the “stop words” functionality. Since they will likely have a common database of stop words, the private words functionality can be reserved to be more tailored to the nuances of the particular message being bloomified. For instance, perhaps you do not want them to be able to search for queries like “nuclear bomb codes”.

Also, this feature is more likely to be a regular expression pattern, such that it will remove words meeting things like the following criteria, “remove words which have the pattern ‘password x’, where x is a sequence of non-whitespace characters.  After all, we do not wish to repeat the literal pattern anyone in plaintext space, so pattern matching like this makes more sense. Also, when the message is modified by a legitimate user, the private words must be, again, not included in the bloom filter when re-bloomifying it.

## 3.4 CASE-INSENSITIVITY

In general, we do not care about case information. "Alex" -> "alex". This can be turned on or off.

## 3.5 WORD EXTRACTION

Only include words (minus stop words), e.g., a message like "Hello, Alex. Who is $##$bill?" will map to "hello alex who is bill". This aspect can be handled by a regular expression engine or more complicated language recognizer (e.g., context-free grammar parser), which can be customized for whatever your company's needs are.

### 3.5.1 DEFINITION OF A WORD

Operational definition: only alphanumeric character sequences qualify as words. Example: “Hello, world!” -> “Hello world”.

## 3.6 PRE-HASHING

By pre-hashing, we transform the original tokens in such a way that they, statistically, occupy a reduced sample space but have more variance and are more uniformly distributed than the original sample space. It is this property that makes the transformation potentially useful. Given the statistical properties of the hashed tokens, it is hoped that it may be easier to generate effective bloom filters of smaller size and fewer and simpler hash functions for a given probability of false positives.

Another advantage of pre-hashing: even if the hash function is very expensive, it must only be performed for each word (non-stop word) once. Subsequently, every other hash function can re-use this hash function's output and hash that value instead.

# 4 BLOCKS

Fragment message into N blocks. Assign a bloom filter to each block (auto-tune it for its particular block). Note: We do not have to create "physical" blocks -- they can be logical instead: start of block, size of block.

## 4.1 PHYSICAL

Each block will contain a separate encoding of the data found in the block's range. For multi-level hierarchies, this means that a block will be stored and encrypted, independently, multiple times. This complicates a number of issues, e.g., if someone accessing a different hierarchy modifies a block in a higher layer, then all of the blocks in the lower layer will need to be updated accordingly.

I do not see any advantages to this approach.

## 4.2 LOGICAL

Instead of storing each block independently, they are just block offsets. The message is only encrypted at the smallest block-size level such that tiered access is still possible, e.g., if a user at a high block granularity wants access to a certain block, this may require giving him private keys to multiple low-level blocks.

## 4.3 OVERLAPPING BLOCKS

Suppose we have a message that reads, "hello word this is alex towell is anyone out there".

If we fragment this message into two blocks like so:

block 1: "hello world this is alex"

block 2: "towell is anyone out there"

Suppose we assign a 2-grams bloom filter to each block. Then, bloom filter 1 has members {"hello world", "world this", "this is", "is alex"} and

bloom filter 2 has members {"towell is", "is anyone", "anyone out", "out there"}.

Now, suppose a search query for "alex towell" is submitted. Clearly, this exact 2-gram is not a member of either bloom filter.

If we had additional bloom filters for each block's 1-grams, and approximate matching was desired, then the most that can be said about the relationship between "alex towell" and the message is that "alex" is in block 1 and "towell" is in block 2, which means these two words are a minimum distance of 0 words apart and a maximum distance of 8 words apart.

If it is desirable to allow exact matches up to 2-grams, then this is not an ideal situation. The reason we could not get an exact match has to do with the fact that the n-grams were generated separately for each block. The 2-gram, "alex towell", is an extract 2-gram in the message, but not in the 2-grams in each block independently.

A solution to this problem will only increase the complexity slightly: use overlapping blocks. For instance, the previous message is instead fragmented like so:

block 1: "hello world this is alex towell"

block 2: "towell is anyone out there"

Now, bloom filter 1 has members {"hello world", "world this", "this is", "is alex", "alex towell"} and

bloom filter 2 has members "towell is", "is anyone", "anyone out", "out there".

A solution, then, is if we are fragmenting a message into blocks, and we want to exactly match against n-gram search queries, then for adjacent blocks i and j, i < j, make the last (n-1) words common to both.

Note that for overlapping blocks, the offset + size of block i will be greater than the offset of block j, i < j.

### 4.4 FRAGMENTATION POINTS

Prefer to fragment (make blocks of) message, in order of priority, at:

1. not inside a quote.
2. paragraph endings.
3. sentence endings.

# 5 SEARCHING

## 5.1 VISUAL MATCHING

Visual matching visualizes the degree of matching of the message.

### 5.1.1 PROCEDURE 1

This is the "simpler" approach in that it doesn't try to find the best match in the document (according to some distance measures, see: approximate matching). This approach uses the approach outlined in fuzzy keyword matching. However, it doesn't just output a single (or interval) result.

(1) Extract words [minus stop words and non-alphanumeric tokens] from search query.

(2) Assign weight to each term -- this weight is, by default, equal to 1 / num(keywords), but a more interesting automatically generated metric is 1 / P[word], such that uncommonly used words (derived from a word frequency database) are given more weight than more common words.

    (a) However, a word can be manually assigned a weight coefficient by user

(3) Perform a single word search for each word in the query.

(4) Score relevance of each block:

    (a) How many of the (unique) words in the query are in the block.

    (b) An estimate of how "distant" (according to some distance measure) words are from each other.

(5) Score relevance of entire message:

    (a) This is just the fuzzy keyword match result. See: fuzzy keyword matching.

### 5.1.2 PROCEDURE 2 -- MORE COMPLICATED

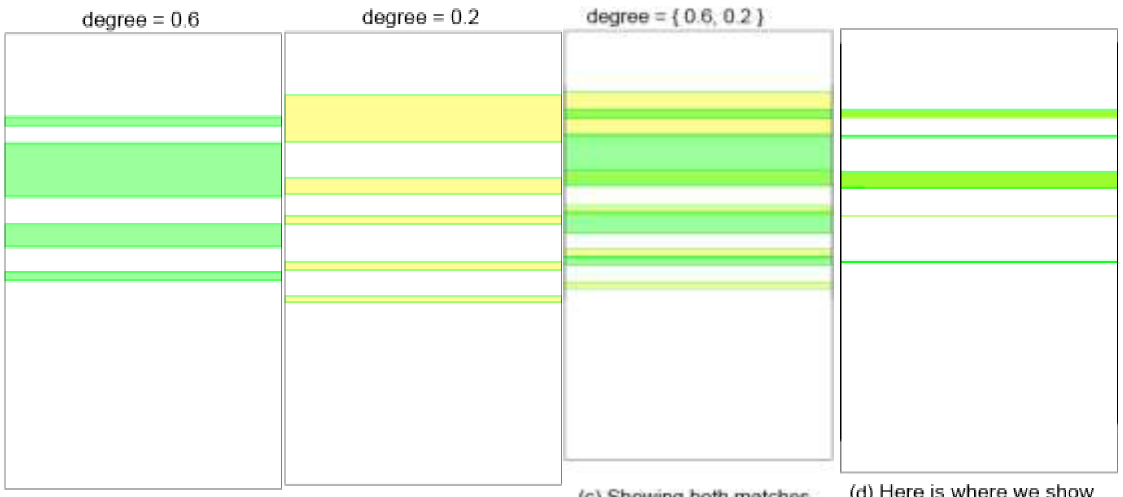
Use fuzzy matching (see: fuzzy matching, finding solutions) to find, approximately, the blocks for which the degree of membership of the search query is at a (local) maximum. This block set will have a degree of membership between 0 and 1. Color the blocks involved a color that depends on this degree of membership.

During the search to find the best match, it will score *many* matches. Remember the top N, and among those N, include in the visualization those matches for which the degree of membership is above a * {degree of membership of best score}, where 0 < a <= 1.

Color different matches (block sets) that match this the same color, but with different intensity. Whenever two matches overlap, the intersection will have a more intense color. In this way, the part that is the most "matching", the part for which there are the most intersections, will have the most intense color.

#### *5.1.2.1 VISUAL DEPICTION*

Shown are four depictions of the visualization. Figure (a) shows the very best fuzzy match. It has a (fuzzy) degree of membership of 0.6. The maximum is a degree is 1, which would have occurred if the user performed a 1-gram (word) search query in which the 1-gram was in the message. (This behavior is different than the “weighted keyword search” behavior, which also factors in an approximation of how often the keyword occurs. Their respective behaviors may unify in the future.) Figure (b) shows a less impressive match of the same exact query on the same exact message, but this is a different set of blocks for which a good match was found. During the process of finding as good of a match as the local search (e.g., gradient descent) method could within a short amount of time, it happened upon many other matches. This is, presumably, the second best match. Figure (c) shows a union of them, with overlapping areas darkened. Figure (d) shows their intersections, which may be especially relevant areas.



(a) A pretty good match, showing only one match result.

(b) A fairly weak match, also showing only one match result.

(c) Showing both matches, on top of each other. The darkest regions are where they overlap.

(d) Here is where we show an intersection of them, colored accordingly (intensified)

## 5.2 FUZZY KEYWORD MATCHING

Suppose the user is interested in a keyword search query, where he is only interested in the presence, or lack thereof, of certain keywords.

Consider that the user is interested in the following keywords: "alex hello world". One thing we can do is just do a simple count: does it have 1, 2, or 3 of the keywords? A slightly more advanced thing we can do is to do an approximate total count of said keywords. Since this count can only exist at the granularity of the smallest block size, if we have N blocks, the highest possible count is 3*N, all three keywords are in all N blocks.

```
KEYWORD_MATCHING_COUNT(KEYWORDS, MESSAGE) ->
    FREQ = 0
    FOR KEYWORD IN KEYWORDS:
        FREQ += GET_BLOCK_NUM_CONTAINING(MESSAGE, KEYWORD)
    RETURN FREQ
```

However, this doesn't take into account how much importance should be given to a keyword. After all, searching for a common keyword, like "car", probably shouldn't be given as much weight as searching for a less common keyword, like "utilitarian". So, let the user provide a weight (of importance) for each keyword if desired. Then, we have:

```
KEYWORD_MATCHING_WEIGHTED(KEYWORDS, MSG, WEIGHT) ->
    WEIGHTED_FREQ = 0
    FOR KEYWORD IN KEYWORDS:
        FREQ += WEIGHT(KEYWORD) * GET_BLOCK_NUM_CONTAINING(MSG, KEYWORD)
    RETURN WEIGHTED_FREQ
```

By default, weight(keyword) could be 1. However, this is rather uninformed default behavior. Why not, instead, weigh a keyword by a measure of how "rare" (reciprocal of probability) it is? The reasoning is that rare words used in a search query are more "relevant". After all, the probability of seeing a rare word is less than the probability of seeing a less rare word, so if we see it in a query that denotes something important about the word's relevance to the user. So, let weight(x) = 1 / probability(x)

Probabilities for words can be derived from relevant word frequency data sets, or even n-gram data sets (this will be discussed later). Let's normalize the output from keyword_matching_weighted by dividing by the summation of weight(keyword) for each keyword in keywords.

```
KEYWORD_MATCHING_WEIGHT_NORMALIZED(KEYWORDS, MSG, WEIGHT) ->
    NORMALIZATION = 0
    FOR KEYWORD IN KEYWORDS:
        NORMALIZATION += WEIGHT(KEYWORD)
    RETURN KEYWORD_MATCHING_WEIGHTED(KEYWORDS, MSG, WEIGHT) / NORMALIZATION
```

If weight just assigns weight(keyword) = 1, then it is the same as keyword_matching_count. keyword_matching_count_normalized has a minimum of 0, and a maximum of num(keywords) * num(blocks in msg). I'd like for it to instead fall between 0 and 1, to facilitate fuzzy degree of memberships.

So, to do this, what do we need? We need a *maximum* value for any message of M words. The maximum is when a message contains nothing but the keywords, and we can determine that this is true. For example, if we had a block size where each block was known to only contain a single word, and then we ran keyword_matching_weighted on it, then it would return precisely M. So, why not just divide a weighted result by M? This means we must know the number of words in the message, however. This can be meta-information.

If we are worried about the number of words being a revealing statistic, we can instead give an interval range for M, e.g., min_count <= M <= max_count.

```
KEYWORD_MATCHING_FUZZY(KEYWORDS, MSG, WEIGHT) ->
    K = KEYWORD_MATCHING_WEIGHT(KEYWORDS, MSG, WEIGHT)
    MIN_DEGREE = K / MIN_WORDS(MSG)
    MAX_DEGREE = K / MAX_WORDS(MSG)
    RETURN MIN_DEGREE, MAX_DEGREE
```

The fuzzy return value can be compared with the fuzzy return values of the same query on other messages such that the user can compare their respective degrees of membership.

## 5.3 APPROXIMATE MATCHING

If approximate search query matching is requested then it is mostly the task of the query engine to submit variations of the search query to find a minimal transformation of the query (using primitive operations), if one exists, which has a positive match. Bloom filters do not deal directly with these operations, and so the two components are independent modules.

If approximate search query matching is requested, then it is mostly [1] the task of the query engine to submit variations of the search query to find a minimal transformation of the query (using primitive operations), if one exists, which has a positive match. Bloom filters do not deal directly with these operations, and so the two components are independent modules.

Types of atomic query transformations are:

- permutations (when searching for an n-gram, there are n! permutations)
- character errors (e.g., mispellings)
- submatches

[1] With the one exception in which bloom filters handle matching an ordered n-word search query with (n+k)-word search queries with k gaps. In this special case, locally sensitive search queries may be handled.

### 5.3.1 DISTANCE MEASURES

When performing search queries, the query will either match the message exactly, or it will only match approximately.

Given the nature of the bloom filter's design, the distance measures will have to cope with the fact that it only has very limited knowledge about the distance between the search query string and a hypothetical string represented in the bloom filter. This is handled by the distance metrics for locality, character, arrangement, and submatch.

#### *5.3.1.1 LOCALITY METRICS*

The locality metric is a measure of how concentrated the terms of the search query are. It is a metric the solution finder uses to evaluate this aspect of the message.

On an exact match on an n-query n-gram, the locality measure is simple: 0 (unless we're using n-grams with gaps; a match upon that will have a locality dependent on the number of gaps it has). However, many query searches (except exact matches on n-grams) will not have exact matches; in these cases, the locality measure has to work with the granularity of the block sizes. The more fine-grained the blocks (e.g., a user has more privileged access to the message and so has private keys to bloom filters assigned to more fine-grained block sizes), the less penalty to the locality metric. Locality measure, at the block granularity level, is measured as follows.

If a bloomified message only has, at most, bloom filters assigned to blocks with k-grams as members, then find all partition sets of an n-query where a partition's largest member is a k-query, e.g., if query is a 3-query, "hello planet earth", and bloomified message only has up to 2-grams as members, then partitions are:

{{hello, planet, earth}, {hello planet, earth}, {hello earth, planet}, {planet earth, hello}}

There are four partitions. Note that this number grows quite rapidly, which is why for large queries we will use approximate local (greedy) methods.

Now, for a given partition, we're going to find out its locality measure against a bloomified message. If some of them have no match, we'll deal with that too (this will incur a submatch penalty, as defined by the submatching approximation error metric, although first we may elect to match the partition against character transformations of its terms, which incurs a character mismatch error as defined by the character transformation error metric, e.g., the "edit-distance" measure).

Find all that do match, and then enumerate all of the possible block assignments for each element in the partition. Now, when N members (1-grams, 2-grams, ..., k-grams) fall into the same block, that has a cost of the block size. So, if a particular partition has M partitions and they fall into a total of K blocks, then just add up each of the K block sizes.

This is called the displacement cost.

```
displacement cost = {sum from i = 1 to K get_block_cost(i, x-grams in block[i]) *
block[i].size}
```

As you can see, we also apply a coefficient to each block that depends on a function of the average size of the x-grams matched. Specifically:

```
GET_BLOCK_COST(BLOCK, X-GRAMS IN BLOCK): WEIGHT
    SUM = 0
    FOR EACH X-GRAM IN X-GRAMS
        SUM += X-GRAM.SIZE
    RETURN X-GRAMS.SIZE / SUM
```

So, larger x-gram matches returns a smaller number, making it less costly.

Finally, we consider the dispersion. What is the dispersion of blocks used in a match? This is a simple calculation. The higher the dispersion, the worse the match. Let B = set of blocks in the match. Then, central measure of blocks in B is defined as:

```
CENTRAL_MEASURE(B):
    SUM = 0
    FOR EACH B IN B:
        SUM += B.INDEX
    RETURN SUM / SIZE(B)
```

```
DISPERSION_MEASURE(B):
    SUM = 0
    CENTER = CENTRAL_MEASURE(B)
    FOR EACH B IN B:
        SUM += (B.INDEX - CENTER)^2
    RETURN SQRT(SUM / SIZE(B))
```

So, add this dispersion measure to the total:

```
locality measure = a * displacement measure + b * dispersion measure, where a and b are
weight coefficients.
```

Let's consider an example. Suppose we have a message with 3 blocks.

```
+------------------------------+ 0
|      "hello"                 |
|             "planet"         | block 0
+------------------------------+ 20
|        "earth"               |
|                              | block 1
+------------------------------+ 40
| "hello"                      |
|              "hello planet"  | block 2
+------------------------------+ 60
```

If query = "hello planet earth", there are 3 possible matches (note: word_n, n indicates block it is in):

```
hello_0 planet_0 earth_1

    locality measure = a*(20/1+20/1) + b*sqrt((0.5-0)^2+(0.5-1)^2)=40*a + 0.5*b

hello_2 planet_0 earth_1

    locality measure = a*(20/1 + 20/1 + 20/1) + b*sqrt((1-0)^2+(1-1)^2+(1-2)^2) = 60*a +
    sqrt(2)*b

"hello planet"_2 earth_1

    locality measure = a*(20 + 20/2) + b*sqrt((1.5-1)^2+(1.5-2)^2)= 30*a + 0.5*b
```

Instead of block indexes in the dispersion calculation, it makes more sense to use actual offsets of blocks and block sizes, but this was easier to calculate.

The results: the match with the "hello planet" 2-gram was the best match, the match of 1-grams in 2 different blocks was second best, and the match of 1-grams in 3 different blocks was the worst. In this case, any choice of weight coefficients 'a' and 'b' would rank them the same. But in general, the result depends on both the dispersion and the displacement measures. After all, it's probably better to match a bunch of 1-grams in 1 block, than fewer 2-grams in 2 blocks. But, these parameters can be tuned to the particular needs of the query.

Note: the inclusion of being able to match k-grams, from k = 1 to k = n, and other factors blows up (a combinatorial explosion!) the search space for reasonably small n. We will use randomized approximate local methods nearly everywhere to quickly get good answers, hopefully even on arbitrarily large queries with 50+ terms in the query.

#### *5.3.1.2 CHARACTER METRICS*

character transformations (per word)

Given a search query and a number k, retrieve occurrences of words which can be transformed into the query with k errors, where an error is an insertion, deletion, or replacement of a character. For example:

searching for occurrences of "key" with 0 edit errors:

```
key
```

searching for occurrences of "key" with 1 edit error:

```
insert:
    key[a-z]
    ke[a-z]y
    k[a-z]ey
    [a-z]key
replace:
    ke[a-z]
    k[a-z]y
    [a-z]ey
delete:
    ke
    ky
    ey
```

analysis

If S = character sequence size and Z is size of alphabet, then, number of variations is N = Z(2S + 1). For example, if S = 2 and Z = 2, there are2(2*2+1) = 10 variations.

Alternatively, let S = 3 and Z = 27, then N = 27(2*3+1) = 27*7 = 189 variations. N ~ ZS, so the larger the alphabet or the larger the positions (size of word), the larger N is. For a fixed Z, N ~ S. This is only for 1-edit errors. For 2-edit errors, things grow much much quicker. It's simple enough to make a program traverse through all possible k-edit errors, but its slow and we have no way of telling how "close" an edit is to the bloomified message so we would have to enumerate and submit them individually. This is too much work for the system, so let's focus only on 1-edit errors.

#### 5.3.1.2.1 big data

We could instead use bloom filters that have as members edit errors of n-grams we insert into it. So, big data to the rescue? Let me explain

Why not learn, from very large word n-gram data sets, probable typos/misspellings/variations of it, and insert those into a bloom filter also? This would be done during the preprocessing stage. Now, we don't enumerate spelling variations because the bloom filter already has the most probable spelling variations for a given n-gram.

If only a certain bloom filter set includes the spelling variations in this manner, then we could even include a character transformation penalty for any query that exactly matches the spelling variation. No need to lose that information about character approximation error.

#### 5.3.1.2.2 error types

When partial matches on words are desirable, e.g., misspellings of a word, then the query engine can handle this by automatically creating and submitting spelling variations of the word(s) in the search query.

The degree of error is defined as the minimal number of changes needed to make the 1-gram test positive for membership in the bloom filter.

##### *5.3.1.2.2.1 number of errors*

How many errors in a word are allowed? How many errors are allowed in all k words in the k-query? For instance, allowing three errors in total for a 5-query, but only one error per word, seems like a reasonable constraint.

Note that the number of queries generated grows combinatorially with the number of errors allowed and the size of the size of the word.

###### *5.3.1.2.2.2 insertions*

This is an error in which a character is added to a word, either at the beginning, somewhere in the middle between adjacent characters, or the end. As previously mentioned, insertions are not allowed to create new words (e.g., it may not transform a k-query into a (k+1)-query).

###### *5.3.1.2.2.3 deletions*

This is an error in which a character is removed from a word.

###### *5.3.1.2.2.4 Replacements*

This is an error in which a character in the word is replaced with another alphabetical character.

#### *5.3.1.3 ARRANGEMENT METRICS*

Arrangement measure is a measure of how distant a particular match in the bloomified message is to a particular partition of the n-query. That is, let's say we have a 3-query = “a b c”, which is a query consisting of 3 words (a 3-gram). The partitions are: {a b c}, {a b, c}, {a c, b}, {b c, a}, {a, b, c}.

{a b c} represents a 3-gram query. (Does the bloomfied message support 3-gram query matches exactly? Then we can try it for a perfect match.)

{a b, c} represents a 2-gram query "a b" and a 1-gram query "c". Does the bloomified message support 2-gram queries? Then we can try the 2-gram query for a perfect match, and independently the 1-gram query for a perfect match.

{a, b, c} represents three 1-gram queries. The bloomified message can always use this, since every one of them should have at least 1-gram granularity. Note, however, that this will result in a maximum arrangement measure cost.

Let's think about how to measure all of this. At the n-gram level, if an n-query is submitted to a bloomified message which supports exact matches up to n-grams, then the following arrangement distance measure is calculated: how many inversions are there? (That is, must we rearrange the n-query to get an exact match? Then count the inversions.) This is a simple calculation. The minimum number of inversions is 0 and the maximum number of inversions is 2n(n-1), which occurs when the query only exactly matches when it is reversed.

Maybe this is not what we want, but it seems reasonable. If an exact match isn't found, then we try to find the best fit match. Note that I won't discuss, here, the fact that we can also do character distance measures on the individual words to try to find exact matches on those instead. This will impose a character distance measure penalty, however (which is usually relatively small compared to locality distance measure and arrangement distance measure).

So, suppose an n-query is submitted against a bloomified message which has k-gram members up to size k = 3. Best case is if each of these k queries can be matched perfectly to the bloomfied message. If this is so, then we have do not have n-gram members, but 3-gram members, and so we assume the very worst, that they, as 3-gram units, have maximum inversions with respect to the other 3-gram units. Instead of 2n(n-1), though, we have 2(n/3)(n/3-1) inversions. Comparing them, 2n(n-1) / [2(n/3)(n/3-1)] = 9(n-1)/(n-3), which asymptotically converges to 9. So, best-case, the worst match has a rearrangement distance that is a factor of ~9 times more distant than this one when assuming the 3-grams are maximally inverted with respect to each other. And, in general, perfectly matching k-grams is ~k times better.

Let's talk about worst case. Worst case is when we only match n 1-grams, and assume they are maximally inverted with respect to each other. If so, we get back to 2n(n-1), like with the maximally inverted n-gram (but for which was otherwise a perfect match; the n-gram still has locality advantages).

Now, what about other combinations? Maybe we have a partition which consists of six 1-grams, two 2-grams, and one 3-grams. Worst case for this is n = 6+4+3, so 26(12) inversions. That's a big number. Best case is when the bloomified message has 3-gram members, and an exact match is found for the two 2-grams and one 3-grams. So, now our n = 6+2+1=9, so 18(8). Over a factor of 2 improvement. The actual locality measure, which is a separate module, will take care of measuring how far apart a particular partitioning is.

##### 5.3.1.3.1 n-grams with gaps

In locality metrics, we use a worst-case assumption on the locality measure. Say we search for "hello planet" and the message has a block that has in it the 3-gram "hello beautiful planet". The best match for this is simply 1-gram matches on "hello" and "planet" in the same block, so worst-case assumption on distance measure is made.

However, we could make 2-grams of the message, with 1 gap, members of the bloomified message. Then, the respective bloom filter for this block handling 1-gap members would have the member "hello planet", which is an exact 2-gram match for the previous query. Now, we can still add a locality measurement error to this since we know the bloom filter dealing with this has 1-gap 2-gram members. We can do the same for 2-gaps, ..., n gaps.

##### 5.3.1.3.2 big data
Using big data, we could look at all k-grams, and determine for any k-gram in our bloomified message that we wish to perfectly match, a probability distribution over the arrangements of those k terms in the k-gram.

Then, we can make the most probable permutations of said k-gram members and make them members, also, of the bloomified message (and assign a rearrangement error approximation to them, if desired). Then, we wouldn't have to do any permutations on the queries for approximate matching, which could significantly speed things up without blowing up the bloom filters necessarily.

This is the same reasoning we applied to character metrics (edit-distance) big data section.

##### 5.3.1.3.3 n-grams
Matching entire n-grams is dependent upon the bloom filter; the query engine cannot answer this question without support from the bloom filter.

If a message is of size N, and query substrings (exact or permuted matches) up to size M are desired, then this operation has time complexity O(N*M*(M-1)* ... * 2) = O(NM!). Since, in general, M is small, e.g., M = 5 (allowing exact search queries up to 5 words long), the time complexity is O(N), which is linear with respect to N. That is, M can be treated as a constant.

Combined with help from the query engine, this can optionally match any permutation of the M words without complicating the bloom filter itself. This is a form of approximate searching. Other techniques will allow far more sophisticated searching techniques without complicating the actual bloom filter. See: block-level granularity, substrings with gaps, character-level errors, and word-level errors.

#### *5.3.1.4 SUBMATCH METRICS*
If only matching k out of n terms in the n-query, there should be a penalty applied--that is, it's more distant the fewer the terms. Worst-case match (but not non-zero) is only one of the terms out of the n terms matching, that is, only a 1-gram matches out of the n-gram query. Whether a submatch of k terms is less distant than a submatch of k+1 terms depends on the other distance measures.

Do note there are 2^n submatches (0 being the submatch for which no match at all was found), so we cannot explore this exhaustively for large n-queries. Even the space for an k-query is quite large, since we would be exploring all of the partitions of this k-query (which is greater than 2^k). So, clearly, for many reasons, we will not be using exhaustive search methods, except for small queries.

If the user wants a match that has all k terms in the k-query, then the query engine doesn't need to explore any k-query submatches.

### 5.3.2 FUZZY MATCHING
Fuzzy search queries provide a degree of membership value in the range [0, 1]. A degree of membership equal to 1 is an exact match -- it is absolutely a member. A degree of membership equal to 0 is absolutely not a member, which occurs when, given the approximation constraints, no match is found -- not even a single 1-gram search found an approximate match (at that point, the only approximation you can do to a 1-gram is character transformations, e.g., 1-edit errors).

Fuzzy search will use the results of an approximate match, which returns the approximation error. We will normalize this approximation error such that a maximum approximation error will have a value of 1 (we can do this if we define a metric space, see: well-defined metrics). Then, to return a degree of membership, we will simply return 1 - (normalized approximation error).

#### *5.3.2.1 FUZZY OPERATORS*
Once we return degree of membership values in the range [0, 1], we can apply fuzzy operators to the returned values.

Fuzzy logic equivalents for NOT, OR, and AND:

- `NOT(q) = 1 - q`
- `AND(q1, q2) = MIN(q1, q2)`
- `OR(q1, q2) = MAX(q1, q2)`

#### *5.3.2.2 HEDGES*
We can also apply hedges to fuzzy degree of membership values. Hedges modify the degree of membership of a fuzzy value in a way that conforms to our every-day common sense, e.g., something can be "somewhat" true but not "strongly" true. There are many hedges we can use. For example, let very(q) = q^2. Then, very(or(q1, q2)) will only be more than K, K in range [0, 1], if **very(or(q1, q2)) = max(q1^2, q2^2) > K.** For that to be true it must be the case that **q1 > sqrt(K) or q2 > sqrt(K)**.

Another hedge example: somewhat(q) = sqrt(q).

Note: very(q) is quadratic, where q is in range [0, 1]. This means that the rate of change of very(q) is 2q as opposed to identify(q) = q, which has a rate of change of 1. Also, very(q) reaches a maximum at q = 1, and a minimum at q = 0, just like with the linear function, identity(q) = q. However, very(q) is lower everywhere in the range (0, 1) than identity(q), and it is the furthest apart from identify(q) at q = 1/2. If our defuzzification test for whether a fuzzy membership value is true is, say, K = 1/2, then q is true whenever q > 1/2, but very(q) is true only when q > sqrt(1/2) = 1 / sqrt(2) ~ 0.7. This makes since because we are asking if "q is very true" not just "q is true".

### 5.3.3 CRISP LOGIC

In approximate searching, the return value from a search query will not be a Boolean indicating whether a match was found. Rather, the result for each query returns a measure of its minimum approximation error (see: distance measures).

To convert this approximation error into a Boolean, we only need to define a Boolean function that takes as input the approximation error and returns as output a crisp Boolean. For instance:

- TRUE(q) = return q < K

Since we know the maxim approximation error for a given n-query (a query of size n), it makes more sense to define TRUE(q) in terms of the maximum error approximation possible.

- TRUE(q, k) = return q < 0.25 * k

That is, the approximation error, q, must be less than a quarter of the maximum error for it to be considered true.

### 5.3.4 FINDING SOLUTIONS

#### *5.3.4.1 LOCAL SEARCH METHODS FOR AN APPROXIMATE BEST ANSWER*

A search query with n 1-grams, called an n-query, is represented as a set of sets, where the sets are partitions of the n 1-grams, e.g., if the search query is “a b c”, then an example of a set is {{a b}, {c}}. This represents a 2-gram, and a 1-gram, pulled from the 3-gram search query.

Initially, we may represent an n-query as a set with n sets, where each of the n sets has one 1-gram member. On this partition, we can apply the distance measures to it for an estimate of how “distant” (approximate) it is from a perfect n-gram match (a perfect match may – and probability generally is – impossible, e.g., the bloomified message may not even have bloom filters which have n-grams as members, but only k-grams where k < n).

What we have, then, are states represented as partitions of n-gram queries. This is a graph search problem (potentially huge, if the n-query is large). To find better solutions, we may apply various transformations on a state, e.g., union some of the sets together such that some of the sets now represent 2-grams (if previously they were all 1-grams). In this way, we are allowed to look at nearby “neighbors” of a state to see if any of them are an improvement. If so, we will move to a neighbor that lowers (gradient descent – exploring a state space to find a local minima) are distance measure. This is called a local search method, as we are not exhaustively exploring the space (not necessarily), we are only considering local improvements (with the exception of some randomization thrown in to get us out of what are called local minima).

In figure 2, I show an example of how a graph search might unfold. The green arrows depict the nearest neighbor adjacencies. In this graph, the search query was a 5-gram “a b c d e”. So, initially it tries just a bag of 1-grams – it found them all (if it didn’t, then it would definitively remove any missing words since they cannot possibly be in any solution; this would, however, incur a submatch penalty). Then, it tries neighboring nodes, like “a b | c | d | e”. In this node, we see that 1-gram “a” and 1-gram “b” have been unioned into a 2-gram “a b”. If one or more of the bloom filters has “a b” as a member, then maybe this will result in a lower distance score. Or maybe not. It depends on the distance measures and how that particular set of x-grams is distributed throughout the message.

In the far right bloom state on the second level, we see that one of the 1-grams, **c**, has been transformed into **c’**. This represents a 1-edit error approximation. Maybe **c’** results in a better match, taking into account the character distance approximation error. (Note: the number of character transformations for a state are passed down to its children to honor the maximum edit error constraint.)

Going down the path, we see the third level down on the far right has what is called a permutation transformation (instead of just a union, now we are changing the permutation of words in a set of sets). The permutation may allow an exact match on “e d c a”, but it also penalizes this permutation, as defined by the rearrangement distance measures.

Finally, on the last level, we see a node that has only four query terms – **a, d, e, b**. Query term **c** has been removed. This is a removal transformation. This effects the submatch distance measure – which, one imagines, has a very strong penalty, as now we are moving terms from the query in order to try to find a more concentrated (less dispersed) match set.

At this point, time's up. The graph search returns the best result it had so far, although it could have also been storing other nodes which scored well (mainly for visualization purposes, e.g., intersecting two of the best matches to see where they overlap).

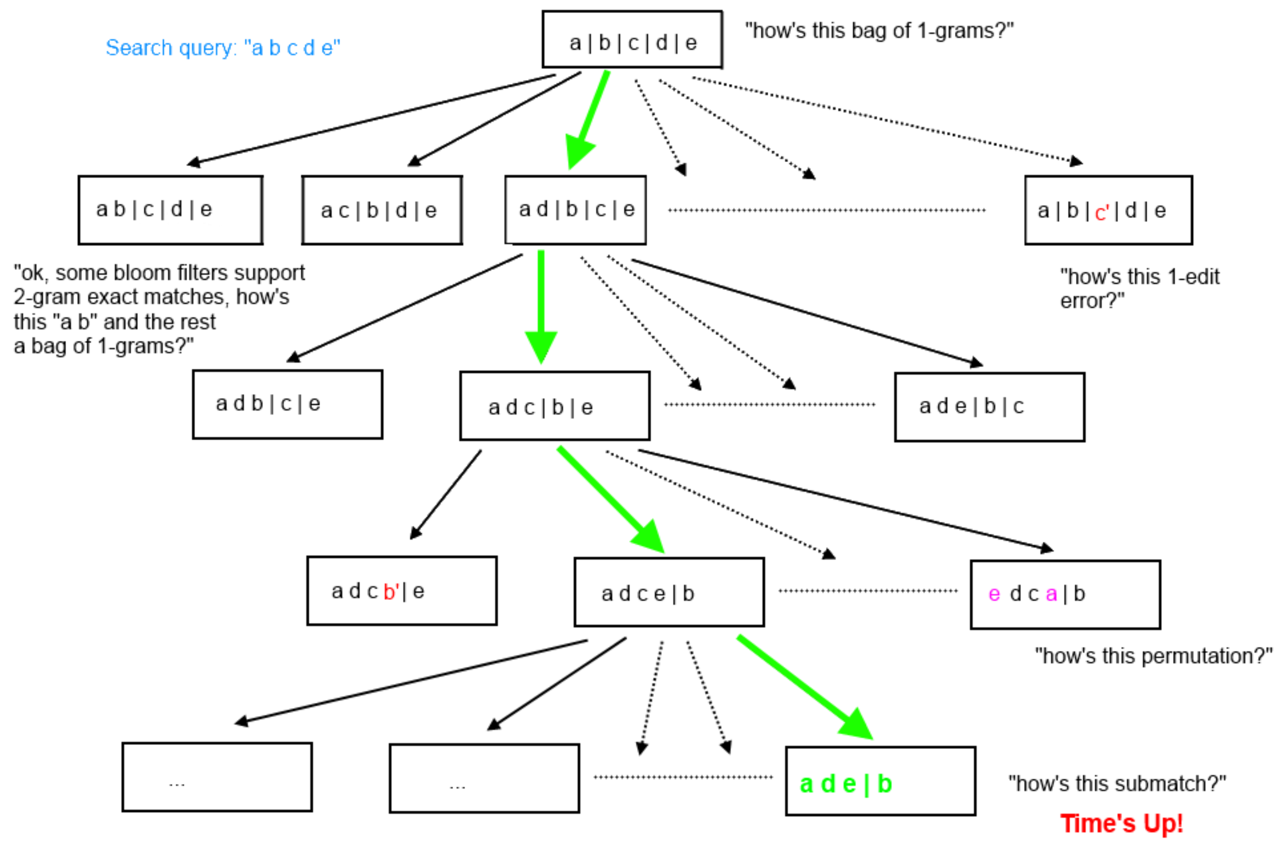

Search query: "a b c d e"
a | b | c | d | e
"how's this bag of 1-grams?"
a b | c | d | e
a c | b | d | e
a d | b | c | e
a | b | c' | d | e
"ok, some bloom filters support 2-gram exact matches, how's this "a b" and the rest a bag of 1-grams?"
"how's this 1-edit error?"
a d b | c | e
a d c | b | e
a d e | b | c
a d c b' | e
a d c e | b
e d c a | b
"how's this permutation?"
...
...
a d e | b
"how's this submatch?"
Time's Up!

*Figure 2 Representing states as nodes in a graph search*

High-level pseudo-code template for local searching is as follows.

```
INPUT:
    PARTITION:
        A STARTING QUERY PARTITION (SET OF SETS), E.G., FOR A K-QUERY, WHICH
        IS A K-GRAM, IT COULD BE A K PARTITIONING WITH A 1-GRAM IN EACH ONE:
            {{ Q1 }, { Q2 }, ..., { QK }}
        OR IT COULD BE ANY OTHER EXHAUSTIVE PARTITIONING, E.G.:
            {{ Q1, Q3}, { Q2, Q4, Q7}, ..., { QK, Q6, Q5, Q10}}
    MEASURES:
        FOUR MEASUREMENT FUNCTIONS, EACH OF WHICH TAKE IN THE PARTITION:
        (1)   ONE FOR LOCALITY METRICS: MEASURE OF HOW DISPLACED THE K
        SETS IN THE PARTITION ARE.
        (2) ONE FOR ARRANGEMENT METRICS: MEASURE OF HOW DISTANT THE QUERY'S
            ARRANGEMENT (SEQUENCE OF 1-GRAMS) IS FROM THE CLOSEST APPROXIMATE
            MATCH IN THE BLOOM FILTER.
        (3) ONE FOR CHARACTER METRICS: A MEASURE OF HOW EDIT DISTANCE, E.G.,
            HOW MANY EDITS HAD TO BE MADE TO IT TO HAVE IT MATCH?
        (4) SUBMATCH PENALTY: IF ONLY MATCHING K OUT OF N, K < N, TERMS,
            APPLY A PENALTY.
    NEIGHBORS:
        A FUNCTION WHICH RETURNS SOME NEIGHBORS OF THE INPUT. A LOT OF
        PRUNING/APPROXIMATION/RANDOMIZATION CAN BE DONE HERE.
    STEPS:
         NUMBER OF STEPS TO TAKE BEFORE RETURNING BEST FOUND VALUE
GREEDY_FIND_MINIMUM(PARTITION, MEASURES, NEIGHBORS, STEPS) -> A SOLUTION
    MEASURES(PARTITION), CONSIDERING NEIGHBORS OF PARTITION."
    DISTANCE = MEASURES(PARTITION)
    NEIGHBORHOOD = NEIGHBORS(PARTITION)
    FOR I = 1 TO STEPS
         NEIGHBOR_PARTITION = EXTRACT A NEIGHBOR FROM THE NEIGHBORHOOD
         DISTANCE2 = MEASURES(PARTITION)
         IF DISTANCE2 <= DISTANCE:
              PARTITION, DISTANCE = NEIGHBOR_PARTITION, DISTANCE2
              NEIGHBORHOOD = GET_NEIGHBORS(PARTITION)
    RETURN {PARTITION, DISTANCE}
```

Many things can be done with this template. For instance, to prevent expending a lot of work on already visited states, use a visited set (this could even be a bloom filter to keep space consumed by it low). We can also do random restarts to get out of local minima, and this is an embarrassingly parallel problem: every call to greedy_find_minimum is independent.

##### 5.3.4.1.1 get_nearest_neighbor function

One way to get a neighboring state from an existing state (to facilitate descending from an already good solution) is to perform union, partitioning sets, removing [sets], and adding [sets] operations on the sets [representing the x-grams of the [sub]query).

If, for a k-query, we [generally] start on a partition of k sets, each with one 1-gram, then we can usually move in a direction of only performing unions; the primary exception to this would be if we use randomized algorithms and many states are not explored. This is actually probably the rule rather than the exception.

Note: if the number of terms in all of the x-grams sums to k, and k < n, and the original query was an n-query, then this represents a submatch k-query, where only k terms of the query are trying to be matched to the bloomified message.

Here's an example using only unions:

partition = {{a},{b},{c},{d}}

neighbors:

{{a,b}, {c},{d}}

{{a,c}, {c},{d}}

{{a,d}, {b},{c}} --> if this is chosen, then its neighbors are: {{a,d,b},{c}}, {{a,d,c},{b}}, {{a,d}{b,c}}

{{b,c}, {a},{d}}

Consider: should this function return variations (e.g., 1-edit errors, permutations)? For instance, {{a,d,b}{c}} has permutations: {(a,d,b),(c)}, {(a,b,d),(c)}, {(d,b,a),(c)}, {(d,a,b),(c)}, {(b,a,d),(c)}, {(b,d,a),(c)}.

##### 5.3.4.1.2 randomization

##### 5.3.4.1.3 approximate

#### *5.3.4.2 OPTIMAL*

We need to define well-defined metrics since for large queries we cannot explore entire state space.

##### 5.3.4.2.1 well-defined metrics

These kind of well-defined metrics will help prune out a lot of the search space on any query matches. For instance, we only need to explore a certain submatch query size, and we only need to explore that submatch query in a certain order, from most optimal to least optimal.

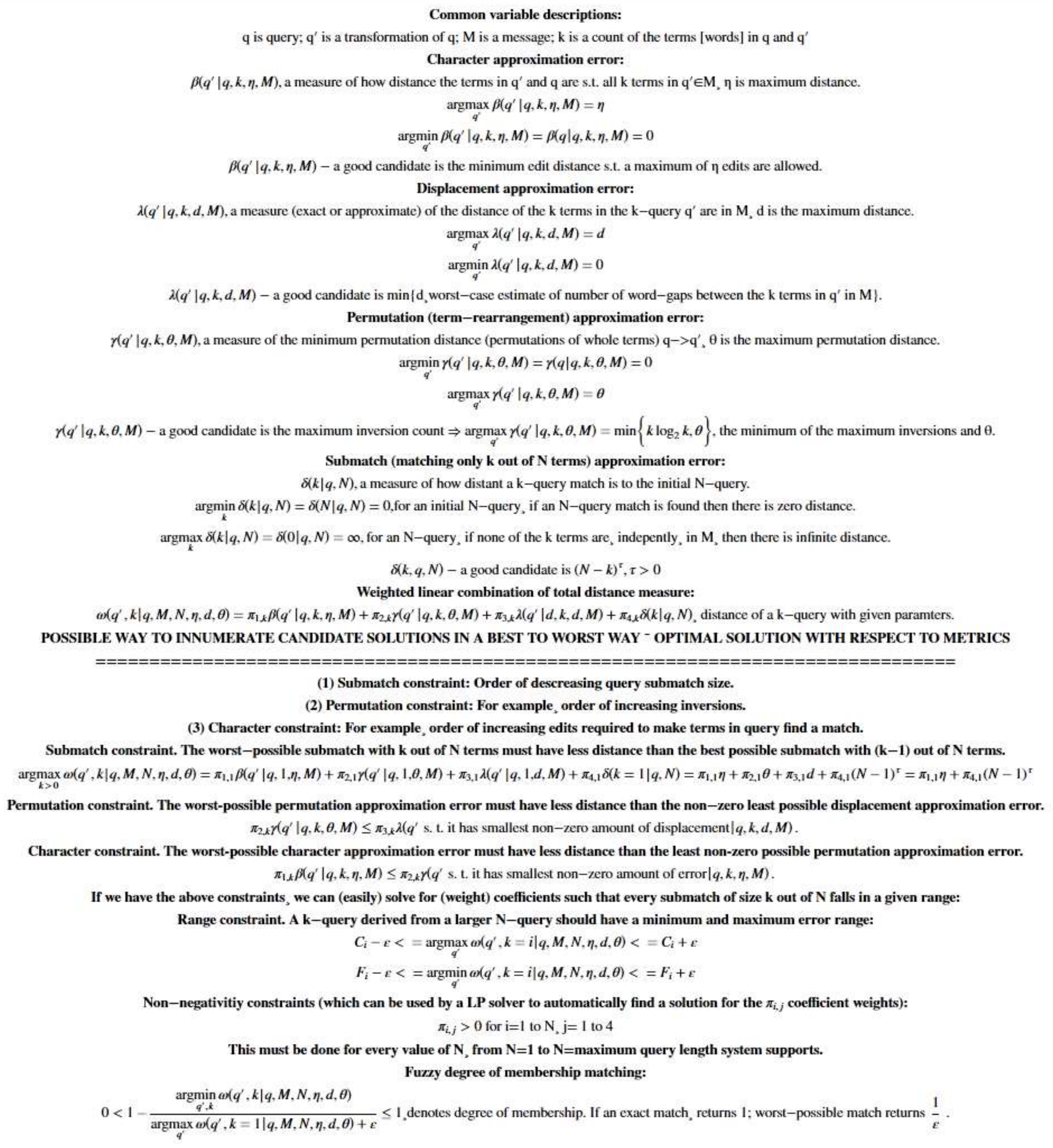
If we have the worst-possible match at that submatch size, that's still a potentially enormous state space to explore, so we can still explore it using local search methods which only guarantee finding a local maximum. I would argue approximate minima are OK since the bloomified message itself is an approximation of the message. Let's find a reasonable solution quickly, without having to use too much memory or time.

Note: The very best match for a k-query search happens when a k-gram granularity bloomified message has a perfect match to it, which has O(# of bloom filters) complexity. It's constant. Always try this one first.

If we don't use the metric constraints, but we do use the distance metrics (e.g., we have the distance functions, but we don't have the constraints on those), then we can still use those metrics to provide a degree of membership estimate, according to those metrics. And, as stated previously, these metrics give us an upper bound on maximum number of approximation errors.

Here are the formulas. They might be hard to see – I lost my MathML file I used to make this, unfortunately, but the image itself can be zoomed.

---



**Common variable descriptions:**

q is query; q′ is a transformation of q; M is a message; k is a count of the terms [words] in q and q′

**Character approximation error:**

$\beta(q'|q,k,\eta,M)$, a measure of how distance the terms in q′ and q are s.t. all k terms in q′∈M, η is maximum distance.

$$\operatorname*{argmax}_{q'} \beta(q'|q,k,\eta,M) = \eta$$

$$\operatorname*{argmin}_{q'} \beta(q'|q,k,\eta,M) = \beta(q|q,k,\eta,M) = 0$$

$\beta(q'|q,k,\eta,M)$ – a good candidate is the minimum edit distance s.t. a maximum of η edits are allowed.

**Displacement approximation error:**

$\lambda(q'|q,k,d,M)$, a measure (exact or approximate) of the distance of the k terms in the k–query q′ are in M, d is the maximum distance.

$$\operatorname*{argmax}_{q'} \lambda(q'|q,k,d,M) = d$$

$$\operatorname*{argmin}_{q'} \lambda(q'|q,k,d,M) = 0$$

$\lambda(q'|q,k,d,M)$ – a good candidate is min{d, worst–case estimate of number of word–gaps between the k terms in q′ in M}.

**Permutation (term–rearrangement) approximation error:**

$\gamma(q'|q,k,\theta,M)$, a measure of the minimum permutation distance (permutations of whole terms) q–>q′, θ is the maximum permutation distance.

$$\operatorname*{argmin}_{q'} \gamma(q'|q,k,\theta,M) = \gamma(q|q,k,\theta,M) = 0$$

$$\operatorname*{argmax}_{q'} \gamma(q'|q,k,\theta,M) = \theta$$

$\gamma(q'|q,k,\theta,M)$ – a good candidate is the maximum inversion count $\Rightarrow \operatorname*{argmax}_{q'} \gamma(q'|q,k,\theta,M) = \min\left\{k\log_2 k, \theta\right\}$, the minimum of the maximum inversions and θ.

**Submatch (matching only k out of N terms) approximation error:**

$\delta(k|q,N)$, a measure of how distant a k–query match is to the initial N–query.

$\operatorname*{argmin}_{k} \delta(k|q,N) = \delta(N|q,N) = 0$, for an initial N–query, if an N–query match is found then there is zero distance.

$\operatorname*{argmax}_{k} \delta(k|q,N) = \delta(0|q,N) = \infty$, for an N–query, if none of the k terms are, indepently, in M, then there is infinite distance.

$\delta(k,q,N)$ – a good candidate is $(N-k)^{\tau}, \tau > 0$

**Weighted linear combination of total distance measure:**

$\omega(q',k|q,M,N,\eta,d,\theta) = \pi_{1,k}\beta(q'|q,k,\eta,M) + \pi_{2,k}\gamma(q'|q,k,\theta,M) + \pi_{3,k}\lambda(q'|d,k,d,M) + \pi_{4,k}\delta(k|q,N)$, distance of a k–query with given paramters.

**POSSIBLE WAY TO INNUMERATE CANDIDATE SOLUTIONS IN A BEST TO WORST WAY ⁻ OPTIMAL SOLUTION WITH RESPECT TO METRICS**

**==============================================================================**

**(1) Submatch constraint: Order of descreasing query submatch size.**

**(2) Permutation constraint: For example, order of increasing inversions.**

**(3) Character constraint: For example, order of increasing edits required to make terms in query find a match.**

**Submatch constraint. The worst–possible submatch with k out of N terms must have less distance than the best possible submatch with (k–1) out of N terms.**

$$\operatorname*{argmax}_{k>0} \omega(q',k|q,M,N,\eta,d,\theta) = \pi_{1,1}\beta(q'|q,1,\eta,M) + \pi_{2,1}\gamma(q'|q,1,\theta,M) + \pi_{3,1}\lambda(q'|q,1,d,M) + \pi_{4,1}\delta(k=1|q,N) = \pi_{1,1}\eta + \pi_{2,1}\theta + \pi_{3,1}d + \pi_{4,1}(N-1)^{\tau} = \pi_{1,1}\eta + \pi_{4,1}(N-1)^{\tau}$$

**Permutation constraint. The worst-possible permutation approximation error must have less distance than the non–zero least possible displacement approximation error.**

$\pi_{2,k}\gamma(q'|q,k,\theta,M) \le \pi_{3,k}\lambda(q'$ s. t. it has smallest non–zero amount of displacement$|q,k,d,M)$.

**Character constraint. The worst-possible character approximation error must have less distance than the least non-zero possible permutation approximation error.**

$\pi_{1,k}\beta(q'|q,k,\eta,M) \le \pi_{2,k}\gamma(q'$ s. t. it has smallest non–zero amount of error$|q,k,\eta,M)$.

**If we have the above constraints, we can (easily) solve for (weight) coefficients such that every submatch of size k out of N falls in a given range:**

**Range constraint. A k–query derived from a larger N–query should have a minimum and maximum error range:**

$$C_i - \varepsilon <= \operatorname*{argmax}_{q'} \omega(q',k=i|q,M,N,\eta,d,\theta) <= C_i + \varepsilon$$

$$F_i - \varepsilon <= \operatorname*{argmin}_{q'} \omega(q',k=i|q,M,N,\eta,d,\theta) <= F_i + \varepsilon$$

**Non–negativitiy constraints (which can be used by a LP solver to automatically find a solution for the $\pi_{i,j}$ coefficient weights):**

$\pi_{i,j} > 0$ for i=1 to N, j= 1 to 4

**This must be done for every value of N, from N=1 to N=maximum query length system supports.**

**Fuzzy degree of membership matching:**

$$0 < 1 - \frac{\operatorname*{argmin}_{q',k} \omega(q',k|q,M,N,\eta,d,\theta)}{\operatorname*{argmax}_{q'} \omega(q',k=1|q,M,N,\eta,d,\theta) + \varepsilon} \le 1$$, denotes degree of membership. If an exact match, returns 1; worst–possible match returns $\frac{1}{\varepsilon}$.

### 5.3.5 CACHED RESULTS

If a query is submitted to the query engine, its results are saved for later use to service a new query. The state is small--it is just a partitioning of the query--so many can be saved.

However, do not cache the final fuzzy membership value. Instead, cache the best state found during the last (approximate) search. This new best state will be a starting point for the gradient descent. Note, however, that this starting point may be a local minimum. So, same principles apply to get out of local minima: random restarts, etc.

Of course, if the state is a perfect match, then there will be no further work.

## 5.4 PREPROCESSING REVISITED

A form of approximate searching can be done in the preprocessing step by training a bloom filter on a transformation of the message. For instance, if we feed a bloom filter the output from preprocess(in) = soundex(extract_words(remove_stop_words(lower_case(in)))), then it will test membership on the basis of how similar a search query sounds to substrings in the original message.

Besides the [potential] benefit of making searching more phonetic, Soundex has the added benefit that multiple words will map to the same exact sequence of characters, reducing the number of members further and thus simplifying any bloom filters that use it.

### 5.4.1 PHONETIC ALGORITHMS

Phonetic algorithms include algorithms like Soundex. Soundex transforms words that approximately sound the same to the same hash.

### 5.4.2 LOCALITY-SENSITIVE HASHING

http://en.wikipedia.org/wiki/Locality-sensitive_hashing

Words (or n-grams) that are alike in some way match to the same hash. Reduces sample space. A generalization of the ideas expressed in preprocessing and pre-hashing.

## 5.5 EXACT MATCHING

Exact query matching is the simplest case.

For an exact match on a n-query to happen, the bloomified message must support exact match n-grams.

If the n-query does have an exact match in the bloomified message, and the bloomified message consists of N bloom filters at the largest block granularity, then on average N/2 bloom filter queries will be submitted by the query engine. This can even be done in parallel.

If the n-query doesn't have an exact match in the bloomified message, and the bloomified message consists of N bloom filters at the largest block granularity, then N bloom filter queries will be submitted.

If on average, a fraction of P the time the bloomified message has an exact match to user queries, then on average P(N/2) + (1-P)N = PN/2 + N - PN = N(P/2 + 1 - P) = N(1 - P/2).

## 5.6 TOPIC SEARCHING

Using Bayes rule, and an assumption of independence, identically distributed n-grams, we have the following:

```
P[topic | message] ~

    P[n-grams from message | topic] * P[topic] / P[n-grams from message] =

    P[n-gram₁ from message | topic] * ... * P[n-gramₖ from message | topic] * P[topic] /
    (P[n-gram₁ from message] * ... * P[n-gramₖ from message])
```

So, if you generate the n-grams from a repository about medical science, then the n-gram models P[n-gram evidence | message comes from a medical science distribution]. Or, hopefully, P[n-gram evidence | message is about medical science].

- P[message is about medical science | n-gram evidence] = P[n-gram evidence | message is about medical science] * P[message is about medical science] / P[n-gram evidence].
- So, if we have a bunch of n-gram evidence and assume i.d.d. on the n-grams:
- Thus, we can use the bloom filter to quickly see if it contains any n-grams in the evidence provided to answer the question, what is the probability that the document this bloom filter represents belongs to a certain topic or class of documents?

## 5.7 APPROXIMATE MESSAGE SIMILARITY

Another type of search query can be articulated as, "How closely do this message match the encrypted message?" This can be seen as a special variation of "topic searching" where the n-gram evidence comes from the unencrypted message that we wish to compare with the encrypted message. In this case, we would just submit the relevant n-grams from the unencrypted message to the topic searching module. See: topic searching.

Another approach: if the bloomified message has an n-gram granularity at a block granularity that is the size of the entire message (which may be done to facilitate rapid queries) is to bloomify the unencrypted message with the same hash functions as the encrypted message, and then compare their bloom filters.

Then, simple operations on the bloom filter can be used to see how alike they are, e.g., treat the respective bloom filters as vectors, and then all of the tools of linear algebra become available to us to compare them. The simplest operation is to just count how many of the entries match, e.g., 134 out of 155 match may indicate a lot of similarity.

A final solution is to just use the interface provided by the approximate matching module, e.g., construct k-grams derived from the unencrypted message and submit it as an approximate search query. This may give a really interesting picture if combined with the visualization module, but inputting a large message to this interface may not get very good results due to the state space being too large to explore effectively.

### 5.7.1 ATTACK VECTOR

A bloom filter maps every message to a Boolean vector (which represents the member set). If they know the configuration of the bloom filter set of the bloomified message, which they probably can find out since this type of information is revealed, then they could try various candidate messages, input it into their own bloom filter set, and compare their respective Boolean vectors.

If they are equivalent or roughly equivalent, then they may infer that they are probably similar in many respects – or even the same message if an exact match is found. Do note, however, that since we are mapping an arbitrarily large space (of messages) to a finite space (of Boolean vectors of dimension N), an arbitrarily large number of messages will map to the same boolean vector. However, the bloom filter is trained to reduce false positives, so it must at least try to find a way to map members differently than non-members, and the best way to do that is probably to map probable negative examples of n-grams to locations sufficiently different than members.

The vast proportion of messages in the sample space are so unlikely to be messages of interest (unless they are interested in a uniform distribution of characters) that a key attack of this nature may be possible. It is still too vast space to explore, however, without some insight into the actual exact contents of the message. If the attacker already has reason to believe it may be some message, any further damage caused by the attacker verifying the equivalency of the messages will probably be relatively minimal.

# 6 BLOOM FILTER

http://en.wikipedia.org/wiki/Bloom_filter

Binary classifiers answer the following query: is a sample (e.g., example or feature vector) a member of the class or not? A bloom filter is a binary classifier. More precisely, a bloom filter answers the following query: is a particular sample a positive member of its set?

If we make the n-grams (or some subset thereof) of a message the members of a bloom filter, then we can do search query operations on the bloom filter. And, more importantly, we can do this without revealing the large-scale structure or content (only at the n-gram and block levels) of the message. That is to say, we can represent a message (of textual data) as a bloom filter. This representation cannot be read, but it can be queried (within limitations), e.g., "Does this message contain the word 'alex'?".

The most obvious n-gram to use is the 1-gram -- single words. By itself, this only allows us to ask the bloom filter if the message contains particular words. This may be useful, but when we combine the bloom filter with other data structures and algorithms, we can get a lot more mileage out of it, e.g., fuzzy approximate searching.

One approach to expand the set of useful queries is to fragment a message into multiple blocks. Then, for each block, assign a separate bloom filter to it. Now, when we ask the system (of bloom filters) if it contains the query, "hello world", it can not only determine whether the message contains both words, but it can also provide a measure of distance between these two words, e.g., if one word is in $\text{block}_i$ and another is in $\text{block}_j$, by using meta-information about the blocks we can determine an upper and lower bound on how distant they are.

When we combine the block concept (which provides an approximate location of a word) with n-gram members, then we can have the best of both worlds. We can find exact matches up to n-gram search queries, and if an exact n-gram match cannot be found, we can decompose the n-gram search into multiple x-gram searches, where x is any number from 1 to n-1, to try to find the best match (according to various distance measures; see the section in searching on distance measures).

## 6.1 TRAINING

Language is very structured. Seeing a character sequence like "ljojsdf ajh4 aa" or "laugh pencil ocean you" is unlikely. So, given that language samples character sequences from an extremely non-uniform distribution (n-grams may be used to model the actual distribution), why not do a many-to-one transformation, in which these improbable sequences of characters or words map to the same hash as probable sequences. In general, nothing will be lost by this sort of collision because these sort of queries are (1) unlikely, and (2) not, in general, very interesting.

So, the space of tokens is infinite, and map this infinite space to a finite space, e.g., every word maps to n bytes. By the pigeonhole principle, this necessarily means that for every index in the hash, an infinite number of character sequences will map to it. However, if these collisions are between one meaningful token and an infinite number of non-meaningful tokens, then nothing is lost; we may reasonably assume that if the position (n bytes) that a meaningful token maps to is true then it is the meaningful (probable) token that is true rather than the infinite number of linguistically non-meaningful (less probable) tokens.

It is also, however, possible that 2 or more meaningful tokens can map to the same position. The task, then, is to choose hash functions for which this is unlikely to occur. If meaningful collisions can be made sufficiently rare, then our problem is solved. This is our goal in training the bloom filters.

Once we insert the positive examples in the bloom filter, we are done with the positive example data set. Our training stage will not need to use this information. Rather, we only need to look at negative examples, ideally weighted by their probability.

Training is described as optimizing the parameters of the system to minimize the probability of a false positive. To reiterate, we do not wish to reduce the false positive rate on improbable search queries; if it gets those wrong, it does not matter nearly as much. To more effectively reduce the probability of false positives we weigh more heavily the probability of false positives on PROBABLE queries:

1. Look at n-gram data sets and weigh them by their frequency.
2. Sample from this distribution, repetitions allowed, to train the parameters to prefer (due to greater penalization) to avoid false positives on probable n-grams moreso than less probable n-grams. If improbable search queries result in false positives, like random sequences of characters, this is not such a problem.
3. If it is important that some improbable negative examples not have a false positive, include such examples in the negative example training set.

In pseudo-code:

```
P[FALSE POSITIVE] =
    PROBABILITY = 0
    FOR N-GRAM IN NEGATIVE EXAMPLES:
        PROBABILITY += P[POSITIVE | N-GRAM] * P[N-GRAM] }
    RETURN PROBABILITY
```

So, to minimize p[false positive], we should prefer to not misclassify probable negative n-grams. All we need is data, of which there is plenty.

We'll learn the following parameters:

- Number of hash functions for a bloom filter
- Types of hash functions
- Number of bloom filters?
- Size of blocks?

### 6.1.1 TYPES

#### *6.1.1.1 WEIGHTED N-GRAMS*

Use probabilities of n-grams derivable from corpus of n-gram data sets findable on the internet

### 6.1.2 TRAINING SET

During the training phase, we sample from a distribution of negative examples, e.g., negative examples of 1-grams, or 2-grams, …, or n-grams.

A bloom filter could have 1-grams, 2-grams, …, k-grams as members. If this is the case, we would sample negative examples from a distribution of 1-grams, 2-grams, …, k-grams. We can explore the positives and negatives of this.

- A positive of multiple n-gram levels: we simplify the architecture. Instead of a set of k bloom filters in which each bloom filter supports only one n-gram type, e.g., 1-grams, or 2-grams, …, or k-grams, we would only have one bloom filter.
    - E.g., it could include up to 5-grams, but only include probable 5-grams from the document (as determined by a 5-gram corpus).
    - If we want to use locality metrics, then we do need a separate bloom filter for those n-grams which skip words, e.g., if document is "alex says hello world" and we include 2-grams with 1 gap, then the membership set is {"alex hello", "says world"}. This may have a locality distance of just 1, corresponding to the fact that it skips only one word.
        - To use a more granular (more approximate) version of this, we could have only one bloom filter for n-grams with gaps up to M gaps, and since we only know a lower and upper bound on the number of gaps, from 1 to M, we can assume the worst-case and give any matchings from this bloom filter a locality distance of M.
        - We could include only probable gapped n-grams, e.g., before inserting a gapped 5-gram, see how probable this 5-gram is: (P[5-gram | corpus of 5-grams]. It's a conditional probability, so we only include a 5-gram if, given that it's a 5-gram, how probable is it? Anything above a certain probability threshold is added, e.g., THRESHOLD = 1e-20.
            - Only need to pass through document once to construct a bloom filter with n-grams with up to m gaps. Specifically, for each word (1-gram) position i in the document, we have a small inner loop, from j = 1 to n (for up to r-grams of size n), and inside this loop we have another inner loop from k = 1 to m (for up to m gaps in the r-gram), where n and m are must smaller than the number of word positions in the document.

### 6.1.3 ERROR TYPES

True positive (TP): anything inserted into the bloom filter will subsequently always be tested as a true positive.

Formally: P[positive | a member] * P[a member] = (1) * P[a member] = P[a member]

False negative (FN): this is not possible with a bloom filter (Type II error).

Formally: P[negative | a member] * P[a member] = 1 - P[positive | a member] = 0

This means that the sensitivity, which is TP / (TP + FN), is 1. That is, the probability of a positive test, given that the message is a positive example, is certainly 1.

Then, the only things we can improve are:

True negative (TN): correctly tested a negative example as false

False positive (FP): incorrectly tested a negative example as positive (Type I error)

The metric we wish to improve is specificity:

TN / (TN + FP) = TN / [total number of negative samples].

Note: P[negative] = FN + TN = TN; P[positive] = FP + TP = FP + P[a member].

### 6.1.4 GENETIC ALGORITHM

### 6.1.5 UNIFORM RANDOM

Already implemented this. The basic outline is:

```
LET K = NUMBER OF HASH FUNCTIONS = SAMPLE FROM A UNIFORM DISCRETE DISTRIBUTION, GET A
NUMBER BETWEEN 1 ... N.
GENERATE K HASH FUNCTIONS BY CALLING A HASH FUNCTION GENERATOR K TIMES
LET T = SAMPLE FROM A UNIFORM DISCRETE DISTRIBUTION, GET A NUMBER IN 1 … M
```

```
4  CALL A HASH GENERATOR FACTORY WITH PARAMETER T TO INFORM IT WHAT TYPE OF HASH FUNCTION
   YOU WOULD LIKE
5  THIS FACTORY  MAKES A HASH FUNCTION OF TYPE T; THIS MAY ITSELF CREATE A RANDOM OR
   DETERMINISTIC HASH FUNCTION
6  ADD HASH FUNCTION TO A LIST OF HASH FUNCTIONS.
7  INSERT POSITIVE EXAMPLES INTO BLOOM FILTER (1-GRAMS, ..., N-GRAMS OF THE MESSAGE).
8  SEE HOW BLOOM FILTER DOES ON A SET OF EXAMPLES; IT WILL CORRECTLY MATCH ANY POSITIVE
   EXAMPLES FOUND, BUT HOW WILL IT DO ON NEGATIVE EXAMPLES?
9  COUNT FALSE POSITIVES MADE BY BLOOM FILTER ON NEGATIVE EXAMPLES. DO THIS BY MAKING A
   (LARGE) SET OF NEGATIVE EXAMPLES, AND ASKING BOTH THE BLOOM FILTER AND A DETERMINISTIC
   SET DATA STRUCTURE IF EACH EXAMPLE IS A MEMBER OR NOT.
10 IF FALSE POSITIVES LESS THAN MINIMUM FALSE POSITIVES SEEN BY ANY OTHER BLOOM FILTER TO
   DATE ON THE TRAINING DATA, MAKE IT THE BEST CANDIDATE SO FAR.
11 GO BACK TO STEP 1 UNLESS OUT OF TIME
12 RETURN BEST BLOOM FILTER FOUND.
```

## 6.2 STATISTICAL PROPERTIES

The bloom filter can be seen as a statistical structure -- it can be used as a classifier.

## 6.3 N-GRAMS

http://en.wikipedia.org/wiki/N-gram

# 7 SECURITY

## 7.1 SPECIFICITY (OF SEARCH)

### 7.1.1 BLOCK GRANULARITY

The larger the block size, the larger the bounds on the minimum and maximum distance between any pairing of words that test positively as a member of the bloomified message.

This makes it more difficult to determine the structure and content of a message since you can only locate n-gram proximity to some upper and lower bound defined by the block granularity. This upper and lower bound defined by the block granularity level can be used to derive a distance measure. See: locality distance measures.

### 7.1.2 N-GRAM GRANULARITY

Exact matches can only happen for n-gram word sequences, where n depends on the configuration of the bloom filter. If a bloomified message supports k-gram exact matches, but an n-query search is submitted, n > k, then the system will do its best to find the best approximate match given the amount of information that can be derived from the n-gram granularity plus block granularity.

#### *7.1.2.1 ADVANTAGES AND DISADVANTAGES*

Advantages of n-gram granularity over block granularity have to do with the fact that exact matches can be made up to n-grams (max n-gram supported by the bloomified message). We insert the n-grams from the message into the bloom filter, where said n-grams can be quite large, e.g., n > 5, for exact matches on very large search queries.

And despite that, it wouldn't give too much away about the content or structure of the message since it cannot be localized at all; other exact matches on this bloom filter wouldn't tell you anything about its relationship to other exact matches, e.g., they could be overlapping n-grams or they could be at opposite ends of the message.

This hampers any effort to reconstruct the message from search queries. And, what’s more, certain sensitive phrases can be added to the ignore list so that they are not made into members.

Disadvantages, however, include the fact that only limited locality measures are available to us, e.g., if two n-grams are found in a message, we have no way of knowing how close they are. It would be beneficial to be able to approximately tell how close they are to each other. More closely spaced n-grams are, by the principle of locality, more relevant towards one another.

So, a lot of the magic comes from combining both approaches. That is what we set out to do.

## 7.2 AUTHORIZATION

### 7.2.1 ASYMMETRIC CYRPTOGRAPHY

# 8 NOMENCLATURE

**term**: a legal word that can be searched for, e.g., alphanumeric sequence of characters

**k-query**: a search query with k terms, where each term is separated by whitespace

**bloomified** message: a message (document) which is being represented by a set of bloom filters.

**bloom filter**: ...

**approximate matching**: ...

**fuzzy matching**: ...

## 8.1 QUERY ENGINE

The query engine is responsible for transforming a user's query into a form digestible by the other modules.

The query engine, at least initially, will only support atomic queries. Atomic queries consist of a single search term, e.g., "hello world". They do not have disjunctions, conjunctions, or negations, e.g., "NOT('hello world') OR 'alex'". See future work::complex queries for ideas about how all of this can work as a propositional fuzzy logic search language.

# 9 FUTURE WORK

## 9.1 OTHER CLASSIFIERS

### 9.1.1 NAIVE BAYES

### 9.1.2 NEURAL NETWORK

## 9.2 MITIGATING RECONSTRUCTION COSTS

If a message is modified, its bloom filters may need to be reconstructed at some point.

- A degree of staleness in the bloom filters may be acceptable though.

However, if we use block-granularity bloom filters, then in theory only the affected blocks need to be reconstructed.

### 9.2.1 BLOCK GRANULARITY

### 9.2.2 DETECTING NON-MEANINGFUL EDITS

If a user modifies the structure of the message by:

- Adding/removing stop words
- Adding/removing whitespace / non-alphanumeric characters
- Changing the casing of any characters (lower case or upper case)

Then the modification is not meaningful from the perspective of the bloom filter. That is, such changes do not cause the bloom filter to change if you reconstruct them using as input the modified message. Thus, these modifications do not necessarily call for a reconstruction task. However, do note that if we are using (logical or physical) blocks, then such changes may call for updating block boundary meta-information.

A relevant question, then, is will it cost more to reconstruct the bloom filter unconditionally or detect such unmeaningful changes to avoid the reconstruction cost? That is, how frequently are edits of this kind? For instance, it is my suspicion that whitespace is frequently the only target of a modification; however, I have no hard data to back this up.

stop words

non-alphanumeric characters

changing character cases

## 9.3 BLOOM FILTER NETWORKS

### 9.3.1 MULTINOMIAL CLASSIFICATION

Construct a bloom filter, or a set of bloom filters, for each type of desired classification. This is, essentially, a single-layer bloom filter network.

### 9.3.2 MULTI-LAYER HASH (OR BLOOM) NETWORKS

Instead of a single layer of hash functions, as in bloom filters, use multiple layers.

When a bloom filter in a lower-layer processes input, it will output a Boolean value indicating membership.

## 9.4 COMPLEX QUERIES

A complex search query allows for logical operations like conjunctions, disjunctions, negations on the terms of the search query. See: BNF.

So, search queries like "(NOT(term_1) OR term_2) AND term_3" can be performed. This complex query must then be decomposed, by the query engine, into multiple atomic queries. For instance, the prior complex query can be decomposed into the following sequence of atomic queries, along with instructions that correctly execute the query:

```
Q1 = RESULT OF SEARCHING FOR TERM1
Q2 = RESULT OF SEARCHING FOR TERM2
Q3 = RESULT OF SEARCHING FOR TERM3
Q4 = NOT(Q1)
Q5 = OR(Q2, Q4)
Q6 = AND(Q3, Q5)
RETURN Q6
```

### 9.4.1 PROPOSITIONAL FUZZY LOGIC

To reformulate the complex queries example, mentioned in the complex queries node, all we need is the following:

```
Q1 = DEGREE OF MEMBERSHIP OF SEARCH QUERY "TERM_1"
Q2 = DEGREE OF MEMBERSHIP OF SEARCH QUERY "TERM_2"
Q3 = DEGREE OF MEMBERSHIP OF SEARCH QUERY "TERM_3"
Q4 = NOT(VERY(Q1))
Q5 = OR(SOMEWHAT(Q2), EXTREMELY(Q4))
Q6 = AND(Q3, Q5)
RETURN Q6
```

### 9.4.2 BNF GRAMMAR

A full grammar (expressed in BNF notation) for propositional compound queries would look something like:

```
<QUERY>          -> <ATOMIC QUERY> | <COMPLEX QUERY>
         -> <ATOMIC QUERY> <BINARY OP> <QUERY>
         -> <COMPLEX QUERY> <BINARY OP> <QUERY>
<ATOMIC QUERY>   -> "<STRING>"
<COMPLEX QUERY>  -> <UNARY OP>(<QUERY>)
         -> (<QUERY> <BINARY OP> <QUERY>)
<STRING>         -> <ALPHANUMERIC> <ALPHANUMERIC>
<UNARY OP>       -> NOT | ...
<BINARY OP>      -> AND | OR | ...
<ALPHANUMERIC>   -> [A-Z0-9]+
```

### 9.4.3 APPROXIMATE MATCHING

In crisp logic, I show how to convert an approximation error into crisp Boolean logic. All you need is a TRUE(q) function.

So, to reformulate the complex queries example, mentioned in the complex queries node, all we need is the following:

```
Q1 = APPROXIMATE ERROR OF SEARCH QUERY FOR TERM1
Q2 = APPROXIMATE ERROR OF SEARCH QUERY FOR TERM2
Q3 = APPROXIMATE ERROR OF SEARCH QUERY FOR TERM3

// Q1, Q2, AND Q3 ARE APPROXIMATION ERRORS
// THEREFORE, NEED TO CONVERT IT TO CRISP
// LOGIC BY DEFINING THE TRUE OPERATOR OVER IT
Q4 = NOT(TRUE(Q1))Z
Q5 = OR(TRUE(Q2), TRUE(Q4))
Q6 = AND(TRUE(Q3), TRUE(Q5))

RETURN TRUE(Q6)
```