When The Levee Breaks: a practical guide to sketching algorithms for processing the flood of genomic data

Will PM Rowe (will.rowe@stfc.ac.uk)

Scientific Computing Department, The Hartree Centre, STFC Daresbury Laboratory, UK

Abstract

Considerable advances in genomics over the past decade have resulted in vast amounts of data being generated and deposited in global archives. The growth of these archives exceeds our ability to process their content, leading to significant analysis bottlenecks.

To address this flood of genomic data, sketching algorithms have great utility in accurately summarising genomic data; whilst using minimal resources and requiring only a single pass of the data.

I review the current state of the field, focussing on how the algorithms work and how genomicists can utilise them effectively. I include interactive workbooks for explaining concepts and demonstrating workflows.

https://github.com/will-rowe/genome-sketching (MIT License)

1. Introduction

To gain biological insight from genomic data, a genomicist must design experiments, run bioinformatic software, and evaluate the results. This process is repeated, refined or augmented as new insight is gained. Typically, given the size of genomic data, this process is performed using High Performance Computing (HPC) in the form of local compute clusters, high-memory servers, or cloud services. HPC offers fast disk-access, a large number of processors and high-memory. But HPC resources are in demand, with researchers queuing to use them, or having limited funds to access them. Worse yet, what happens when an experiment has too much data to be realistically stored or analysed using these resources? Similarly, given the advent of real-time sequencing technologies, what if researchers want to ask questions of data as it is being generated or cannot wait for HPC to become available?

As genomics continues to thrive, from basic research through to personalised genome services, data continues to flood into genome archives and databases. One of the many consequences of this has been that genomicists now have a wealth of data to choose from when they design their experiments. This requires sampling considerations to be made, such as the quantity, quality and distribution of data. In an ideal world, most genomicists would elect to include all available data but this is growing harder to achieve as the amount of data drives up runtimes and costs.

In response to being unable to analyse *all the things*, genomicists are turning to analytics solutions from the wider data science field in order to process data quickly and efficiently [1–4]. In particular, the model of streaming data processing is proving incredibly effective in minimising the resource usage of genomic analysis. Rather than capturing, sorting and indexing every piece of data, streaming data processing instead quickly looks at each piece of data as it is received and uses this information to summarise the current state (Figure 1). Once a piece of data has been processed it is no longer accessible; only the overall summary is kept [5]. This summary is termed a **sketch** and serves as an approximation of the data that was processed (Table 1).

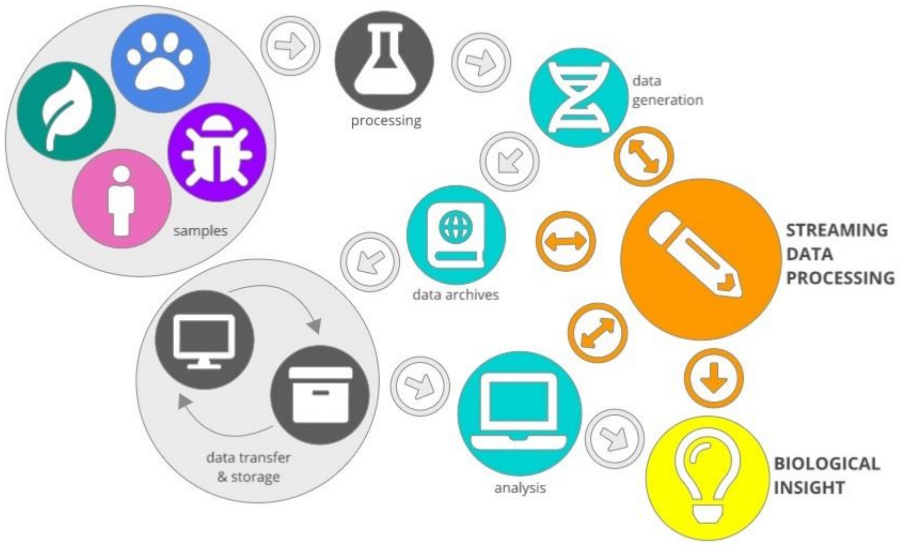

Figure 1. *Streaming data processing (orange arrows) bypasses some traditional genomics workflows (grey arrows) and enables real-time analysis, which can feedback on data generation, transfer and analysis, based on current biological insight.*

TERM	DEFINITION
bit vector	an array data structure that holds bits
canonical k-mer	lexicographically smaller of a k-mer and its reverse complement
hash function	a function that takes input data of arbitrary size and maps it to a bit string, that is of fixed size and typically smaller than the input
Jaccard similarity	a similarity measure defined as the intersection of sets, divided by their union
sketch	a fixed-size, compact data structure that approximates a data set
weighted set	a set that allows for multiple instances of each of its elements

Table 1. *Glossary of terms.*

Sketch data structures are relatively small so fit entirely in memory, they only need only a single pass of the data and you can use a sketch before the underlying data stream has terminated [6]. This makes sketching faster and more efficient than high latency alternatives; you don't have to store an entire data stream and you can analyse data in real-time [4,7].

In the next section, I outline some properties of sketches and how they can be used to approximate the underlying data. In subsequent sections, I describe the core sketching algorithms; detailing their uses, advantages, variants and current implementations for genomics. I provide interactive workbooks to demonstrate key concepts and workflows that utilise sketching to tackle real-world genomics problems (https://github.com/will-rowe/genome-sketching - MIT License).

2. What is sketching

The concept of data sketching has been around for several decades, originating with probabilistic counting algorithms that can estimate the number of distinct elements within a dataset on disk [8]. Sketching has more recently been used to summarise data streams; first applications provided an ephemeral overview of data and more recently have offered persistent summaries of data streams [9].

Put simply, sketching is the process of generating an approximate, compact summary of data. A sketch supports a set of predetermined query and update operations; which are used to approximate the original data. Compared to non-probabilistic algorithms, sketching requires less memory and has constant query time [5].

To be considered a sketching algorithm, there are several requirements that must be satisfied. *Cormode et al*. state that sketch updates must be consistent, irrespective of sketch history or the current sketch state. The sketching process results in a probabilistic data structure (the sketch) that is a linear transform of the input data [6]. Sketches typically implement four key methods: create, update, merge and query. The ability to update and merge means parallelisation is often achievable.

It should be stressed that sketching is not sampling. Although both allow for data to be summarised, sampling does not allow certain questions to be asked of the data, e.g. set membership. Sketching can yield better estimates than sampling. Standard error of a sample of size *s* is 1 / √*s* , whereas sketches of size *s* can guarantee error that is proportional to 1 / *s* [5].

Sketching effectively compresses data, resulting in low memory requirements, queries in linear-time, and reduced bandwidth requirements in distributed systems. Consequently sketching has applications in data analytics, signal processing and dimensionality reduction. So if you can accept an approximate answer and need it quickly, sketching algorithms fit the bill. Which particular algorithm to use depends on the nature of the question you want to ask.

3. Sketching algorithms and implementations

3.1 Set similarity with MinHash

3.1.1. Set similarity

Say we wish to compare two music collections, each containing 100 records. Each collection is a set and we can use **Jaccard similarity**, defined as the intersection of two sets divided by their union, to measure their similarity. If our two collections have 60 records in common, then the Jaccard similarity is 60 divided by the number of unique records 140, giving 0.429.

Jaccard similarity is regularly used in data science, for tasks such as document aggregation and duplicate detection [10]. Document similarity can be based simply on the number of shared words, but to take into account document structure, it may be better to represent the document as a set of overlapping groups of words. For instance, the following sentence “*We come from the land of the ice and snow*” can be broken into five 6-word groups: “*We come from the land of*”, “*come from the land of the*”, “*from the land of the ice*”, “*the land of the ice and*”, “*land of the ice and snow*”. Now, compare that with another sentence: “*The hammer of the gods will drive our ships to new land*”. Both of these sentences share the words “*of*”, “*the*” and “*land*” but not in the same 6-word context, so they do not have any similarity when you account for document structure. The Jaccard similarity is 0.176 if structure is not important, but 0 if it is.

Looking at this example, you can tell it is a bit impractical to use the groups of words as they are; a 10-word sentence turns into five 6-word groups. Instead, we can **hash** these groups. Hash functions take large input data of arbitrary size and map it to a much smaller bit string of fixed size (called a hash value). So, each of these groups of words would be mapped to a hash value, and documents are compared by calculating the Jaccard similarity between the sets of hash values.

Even when using sets of hash values, you can see that Jaccard similarity calculations will get harder to compute as the sets get bigger, or the number of sets increases. In fact, in a worse-case scenario, pairwise similarity calculations scale quadratically in terms of time and space complexity. So, if we want to perform set similarity queries efficiently on large datasets, we could accept approximate answers and look to sketching algorithms.

3.1.2. MinHash algorithm

The MinHash algorithm generates a sketch that is designed to allow scalable approximation of the Jaccard similarity between sets. It was originally developed for detection of near-duplicate web pages and images [11].

MinHash, as with other sketching algorithms, relies on hash functions. The hash functions used in sketching algorithms should be uniform and deterministic, i.e. input values should map evenly across an output range and a given input should always produce the same hash value. By applying a hash function to every element in a set and sorting the set by the resulting hash values, a pseudo-random permutation of the set is achieved. Taking this idea a step further, if the same hash function is applied to two sets, the chance of both permutations having the same lowest hash value is going to be equal to the ratio of the number of common elements to the size of the union, i.e. the Jaccard similarity. This is the core concept of the MinHash algorithm [12].

The MinHash sketch data structure is just a set of hash values, plus some extra information describing how the sketch was made (e.g. which hash function). There are several different ‘flavours’ of the MinHash algorithm, such as the K Minimum Values (KMV) sketch, k-partition sketch and the bottom-k sketch [12–15].

For now, let’s focus on MinHash KMV sketches. To generate a sketch of size *S*, a total of *S* different hash functions are required to generate *S* random permutations of the set. For each permutation, the minimum hash value is added to the sketch and all other values are ignored (Algorithm 1). Rather than creating *S* unique hash functions, which can be demanding, we can instead use one or two hash functions and apply simple operations to these base functions for every permutation required.

```
$D$ ← input set
$X$ ← MinHash sketch, initialised with maximum values in each position

for all elements $e$ in $D$ do                  // iterate over each element in the set
    $hv_1$ ← hashFunction1($e$)                 // create base hashes of the input element
    $hv_2$ ← hashFunction2($e$)
    for all positions $i$ in 1 .. $X$ do
        $curMin$ ← X[i]                         // current minimum in this sketch position
        $hv$ ← $hv_1$ + (i * $hv_2$)            // generate hash value for element
        if $hv < curMin$ then                   // evaluate against the current minimum
            X[i] ← $hv$                         // a new minimum value replaces the old
```

***Algorithm 1**. Pseudocode for the MinHash algorithm to generate a MinHash KMV sketch.*

To estimate the Jaccard similarity of two sets using their MinHash sketches, we compare the values in each position of the sketches and increment a counter if they match. The counter is then divided by sketch length, yielding the Jaccard similarity estimate. As this is probabilistic, we can increase the accuracy by increasing the number of random permutations sampled; the longer the sketch, the better the Jaccard similarity estimate. It’s important to remember that when comparing MinHash sketches, they must have been constructed using the same hash functions so that shared elements will yield the same hash values.

To generate a bottom-k sketch instead of a KMV, the MinHash algorithm only creates one permutation of the set and then retains one or more minimum values.

This means that a bottom-k sketch only needs to be updated once a new hash value is encountered that is smaller than the maximum value currently in the sketch. Again, the longer the sketch, the more points sampled and the higher the accuracy of Jaccard similarity estimates.

3.1.3. MinHash implementations for genomics

To apply MinHash to a genomic data stream, we rely on the bioinformatic workhorse of **k-mer decomposition**. This involves breaking down a sequence into a set of overlapping subsequences of length *k*, termed k-mers, equivalent to the word groups in our earlier example (Figure 2a). We don’t need to differentiate between k-mers and their reverse complement, so keep only the lexicographically smaller of the two (the **canonical k-mer**).

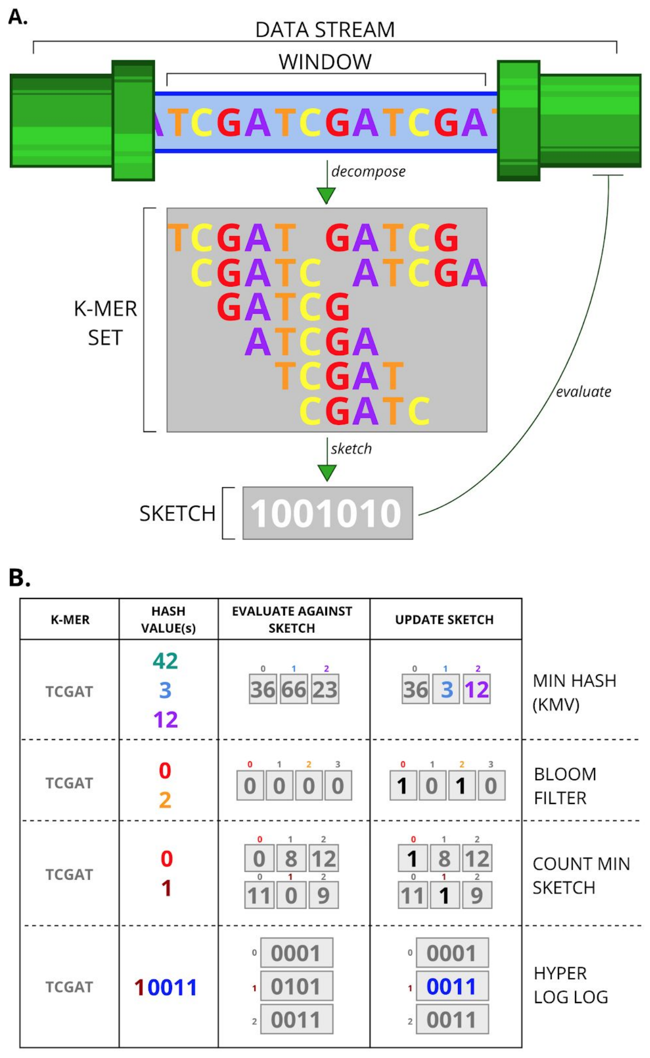

Figure 2a. **Sketching applied to a genomic data stream**. *The genomic data stream is viewed via a window; the window size may be equivalent to the length of a sequence read, a genome, or some other arbitrary length. The sequence within the window is decomposed into a set of constituent k-mers; each k-mer can be evaluated against its reverse complement to keep only the canonical k-mer. As k-mers are generated, they are sketched and the sketch data structure may be updated. The sketch can be evaluated and allow feedback to the data stream process.* **2b**. **Common sketching algorithms applied to k-mer sets, using example parameters**. *MinHash KMV: each k-mer is hashed by 3 functions; each value is evaluated against the corresponding position in a sketch of length 3 and the sketch updates with any new minimum. Bloom filter: each k-mer is hashed by 2 functions; the output of each function is in the range 0-3, corresponding to a sketch position, which is then set to 1. CountMin sketch: each k-mer is hashed by 2 functions; the output of each is in the range 0-2, corresponding to the sketch row length. The hash function and value give a position in the sketch matrix, which is incremented by 1. HyperLogLog: each k-mer is hashed; the output is split into a prefix and suffix. The prefix looks up a register and the suffix is added if it features a larger bit-pattern observable.*

The first implementation of MinHash for genomic data was MASH, which demonstrated that bottom-k MinHash sketches facilitate fast, approximate sequence similarity searches and provide an efficient method of sequence compression [1]. The authors extended MinHash to incorporate a pairwise mutation distance and P-value significance test. MASH is able to accurately cluster genome and metagenome data, as well as perform real-time sequence database search. Since MASH, other multipurpose and well-documented MinHash libraries for genomics have been developed, such as sourmash and Finch [16,17].

As well as these libraries, there are several bioinformatic programs that utilise the MinHash algorithm to provide fast and efficient answers for common genomics workflows (Table 2). These include GROOT which uses KMV sketching for variant detection in metagenome samples [18], mashtree which uses bottom-k sketching for phylogenetic tree construction [19], and MashMap which uses bottom-k sketching for long read alignment [20,21]. MashMap now also uses **minimizers**, which is a concept closely related to MinHash. By sliding a window across a sequence and decomposing windows to k-mers, the smallest hashed k-mer is the minimizer for that window. The minimizer from each window make up the sketch. Minimizers were proposed by *Roberts et al.* as a sequence compression method [22], and have been popularised by the MiniMap read aligner [23,24].

SOFTWARE	**PURPOSE**	**SKETCHING ALGORITHM**
GROOT [18]	variant detection in metagenomes	MinHash (KMV)
mashtree [19]	phylogenetic tree construction	MinHash (bottom-k)

MashMap [20]	long read alignment	MinHash (bottom-k)
MASH [1]	sequence analysis	MinHash (bottom-k)
sourmash [16]	sequence analysis	MinHash (bottom-k)
finch [17]	sequence analysis	MinHash (bottom-k)
MiniMap2 [24]	read alignment	Minimizer
ABySS [25]	genome assembly	Bloom filter
Lighter [26]	sequencing error correction	Bloom filter
BIGSI [27]	sequence index and search	Bloom filter
khmer [28]	nucleotide sequence analysis	Count-Min sketch
dashing [2]	sequence analysis	HyperLogLog
krakenUniq [29]	metagenome classification	HyperLogLog
HULK [4]	metagenome comparison and classification	Histosketch

Table 2. Examples of *bioinformatic software utilising sketching algorithms. For an up to date list, please refer to: (github.com/will-rowe/genome-sketching/blob/master/references.md).*

3.1.4. Considerations and variations

MinHash is an efficient way to estimate Jaccard similarity between sets. I described earlier that Jaccard similarity is based on the union and intersection of sets, and that in genomics we consider k-mers as set elements in order to account for sequence structure. However, Jaccard similarity does not take into account element frequency within a set (referred to as a **weighted set**). In genomics, we may want to include k-mer frequency in our similarity estimates, particularly when dealing with metagenomes. Several MinHash implementations have provision for weighted sets. Of note are sourmash (--*track-abundance*), which uses **cosine similarity** for weighted similarity estimates [16], and the *over-sketching* method of Finch, which creates a parent sketch with tracked k-mer abundance before populating a smaller child sketch with dynamic filtering [17].

Another consideration is how to handle sets of different size; MinHash performance degrades with increasing difference in set size [30]. For instance, using MinHash to find a genome within a metagenome. One solution is to combine MinHash with other data structures and utilise a containment index; set intersection is divided by the size of one of the sets to normalise for imbalance. This approach is now offered by sourmash, Finch and MASH [16,17,31]. Sourmash also features several interesting alternatives, such as k-mer binning and greedy partitioning (see *lca* and *gather* sub commands) [16].

As well as the MinHash libraries discussed already, there have been several recent algorithm variations that deserve mentioning. BinDash is an implementation of binwise densified MinHash, offering improved speed, precision and compression performance over bottom-k sketching implementations [32]. Order MinHash is a variant that considers the relative order of k-mers within sequences, enabling calculation of edit distance between sequences [33]. b-Bit MinHash is a MinHash variant used to compress MinHash sketches, resulting in a tradeoff between accuracy and storage cost [34].

3.2 Set membership with Bloom filters

3.2.1. Set membership

From our earlier example of set similarity, we know that we have similar taste in music. Let’s say that you now want to listen to the records in my collection that you haven’t got in yours. This is a set membership query and, in a worst-case scenario, will require a loop through your collection for each record in my collection.

This worst-case scenario can be improved upon. For example our record collections could (or should!) be sorted alphabetically, meaning if I had “*AC/DC - Powerage*” and went through all the A’s in your collection without finding it, I would not need to continue looking and could play the record. We could also improve our search by remembering the entirety of your collection, bypassing the need to loop through your collection for each record in mine.

However, sorting sets can take time and memorising sets can be difficult or impossible; our example would not scale well if we had millions of records. Fortunately, sketching algorithms allow us to approximate set membership queries and return answers quickly.

3.2.2.Bloom filter algorithm

The Bloom filter algorithm produces a sketch for set membership queries; telling us if an element is possibly in a set, or if it is definitely not in the set (i.e. it allows false positives but no false negatives). Once you add an element to a Bloom filter, a subsequent query using the same element will tell you that you have probably seen it before. Elements cannot be removed from a Bloom filter and the more elements you add, the larger the probability of false positives [35].

The Bloom filter algorithm uses hash functions to populate a **bit vector** (the sketch), which is essentially a row of bits that can be set to 0 or 1 (Algorithm 2). To begin, all bits are set to 0. When adding an element to the sketch, multiple hash functions are used to map the element to several positions in the sketch. At each mapped position, the bit is changed to a 1 if not already set, and cannot be changed back during the life of the sketch.

```
$D \leftarrow$ input set
$S \leftarrow$ Bloom filter sketch, initialised with 0s in each position
$N \leftarrow$ Number of hash functions in Bloom filter
$Z \leftarrow$ Number of bits in Bloom filter

for all elements $e$ in $D$ do                // iterate over each element in the set
    for all function $f$ in 1 .. $N$ do       // iterate over hash functions
        $hv \leftarrow hv_f(e)$               // generate a hash value for the element
        $i \leftarrow hv \% Z$                // get position in the Bloom filter
        if $S_i == 0$ then                    // check bit at position
            $S_i \leftarrow 1$                // set bit to 1
```

Algorithm 2*. Pseudocode for the Bloom filter algorithm.*

To perform a set membership query, the query is hashed using the same functions used to create the Bloom filter. Each of the returned sketch positions are checked; if all bits are set to 1 then the element has probably been seen before. If one or more bits are set to 0, the element has definitely not been seen before. This is thanks to the deterministic property of the hash functions, meaning that an element will always be hashed to the same value.

To calibrate a Bloom filter, the false positive rate is proportional to the sketch length (the number of bits). The longer the sketch, the greater the number of possible hash values and the lower the chance of different elements hashing to the same value (a false positive); this is known as a **hash collision**. The more hash functions a Bloom filter uses the slower it will be, but using too few functions will result in more false positives. To decide how long to make the sketch and how many hash functions to use, optimisation calculations can be performed to parameterise the sketch to give approximations within specified error bounds [36].

3.2.3. Bloom filter implementations for genomics

To apply a Bloom filter to a genomic data stream, we again use k-mer decomposition. The canonical form of each k-mer is passed through a Bloom filter; if the k-mer has not been seen before then it is added to the sketch (Figure 2b). This can have several uses, such as approximating k-mer counts (using an additional hash table to track counts), or excluding unique k-mers from analyses.

Although Bloom filters were first used in bioinformatics around 2007, one of the first genomics applications was BFCounter in 2011 [28]. BFCounter used a Bloom filter to track k-mer counts; it used this sketch to give an approximation of k-mer abundance, or generated exact counts by combining the sketch with a hash table and performing a second pass of the data.

K-mer counting is a very common component in many genomic processes, such as sequence assembly, compression and error correction. Software that utilise Bloom filters for these processes include Minia, ABySS, Xander and dnaasm for assembly [25,37–39], Quip for compression [40], and Musket, BLESS and Lighter for error correction [26,41,42]. Bloom filters are also used in conjunction with other sketching algorithms, such as by MASH to prevent unique k-mers (which often arise from sequencing error) from being added to the MinHash sketch [1].

3.2.4. Considerations and variations

Although Bloom filters offer many performance advantages over non-sketching algorithms for set membership queries, as illustrated by their ubiquity in data science and genomics, they have several shortcomings which must be considered prior to their use. Limitations include the inability to remove elements, dynamically resize the sketch or count the number of occurrences of each item. There are several variants of the Bloom filter algorithm which aim to improve on these shortcomings, including counting, multistage and spectral Bloom filters [3,43], Cuckoo filters [44] and counting quotient filters [45].

In addition to these variants, Bloom filters have been used as building blocks in several algorithms for genomic indexing problems. One example are sequence bloom trees, which are a hierarchy of compressed Bloom filters with each one containing a subset of the items being indexed [46]. Sequence bloom trees have been combined with MinHash to allow disk-based search of sketches [16].

A recent indexing algorithm, the Bit-sliced Genomic Signature Index (BIGSI), utilises a set of indexed Bloom filters for real-time sequence search [27]. BIGSI improves upon other methods, including sequence bloom trees, which suffer from a performance drop when indexing diverse sequences (i.e. a scaling dependance on the total number of k-mers in the union of sequences being indexed). To create a BIGSI index, each sequence (e.g. genome assembly) is decomposed to k-mers, hashed *N* times and sketched using a Bloom filter; each Bloom filter is stored as a column in a matrix (the index). To query the index a k-mer is hashed *N* times to give *N* row indices; the corresponding row (a bit-slice) is returned for each. By performing a bitwise AND operation on the bit-slices, the returned column indices indicate samples containing the query k-mer. Not only is BIGSI exceptionally elegant and simple, it shows that a sketching algorithm that has been around for decades can still be adapted to create novel and high-performance genomics applications.

3.3 Element frequency with Count-Min sketch

3.3.1. Element frequency

To continue with our record collection analogy, you are busy enjoying the AC/DC back catalogue but are now wondering how many times they have used that same chord progression. This is an element frequency query and, in a worst-case scenario, requires you to create a list of every chord progression used in all the songs and count the number of times each occurs.

Element frequency queries get harder when you have massive and diverse sets, where the list of counts might become too large to process or hold in memory. For example, keeping count of all k-mers observed in a genome is memory intensive. This is where the Count-Min sketch comes in.

3.3.2. Count-Min sketch algorithm

The Count-Min sketch algorithm produces a sketch that is a 2-dimensional matrix ($d*w$) which is used to approximate element frequency queries [47]. The matrix size is parameterised by two factors, *epsilon* and *delta*, where the error in answering a query is within a factor of *epsilon* with a probability of *delta*.

To add an element to a Count-Min sketch, the element is hashed by *d* pairwise independent hash functions, where each hash function maps an element to a position in the range of 1..*w*. For each position, the counter in the matrix is incremented (Algorithm 3). The requirement of hash functions to exhibit pairwise independence minimises hash collisions. The Count-Min sketch accommodates weighted sets as the counters in the matrix can be incremented by values greater than one.

```
$D \leftarrow$ input set
$X \leftarrow$ Count-Min sketch, initialised with 0s in each position
$d \leftarrow$ Set of hash functions (depth of sketch)
$w \leftarrow$ Number of positions to hash across (width of sketch)

for all elements $e$ in $D$ do                 // iterate over each element in the set
    for all function $f$ in 1 .. $d$ do        // iterate over hash functions
        $hv \leftarrow hv_f(e)$                // generate a hash value for the element
        $i \leftarrow hv \% w$                    // get position for this row in the matrix
        $X_i$ ++                          // increment the counter at this position
```

Algorithm 3*. Pseudocode for the Count-Min sketch algorithm.*

To query a Count-Min sketch and obtain a frequency estimate, an element is hashed as though it is being added to the sketch. Instead of incrementing the counter each matrix position, the counters are evaluated and the minimum value is returned as the estimate.

3.3.3. Count-Min sketch implementations for genomics

Similarly to MinHash and Bloom filters, a Count-Min sketch is implemented for genomics by considering k-mers as set elements. Each k-mer is added to the sketch and the counter incremented by the k-mer frequency (Figure 2b).

Khmer is a multipurpose software library for working with genomic data; at its core is a Count-Min sketch implementation for recording k-mer frequencies [3,28]. Khmer also features a Bloom filter implementation for presence-absence queries. Some of the functionality of Khmer includes: read coverage normalisation, read partitioning, read filtering and trimming. Count-Min Sketching is also utilised by the genome histosketching method, where k-mer spectra are represented by Count-Min sketches and the frequency estimates are utilised to populate a histosketch [4].

3.3.4. Considerations and variations

The Count-Min sketch is a biased estimator of element frequency, due to the possibility of counts being overestimated but not underestimated. Overestimates occur when hash collisions result in the same position in the matrix being incremented by different elements. This is mitigated by increasing the size of the sketch to reduce hash collisions, although this cannot be performed when the sketch is in use (although dynamic sketches are a possibility). Another option is to use Bayesian statistics to characterise uncertainty in the Count-Min sketch frequency approximations [48].

Another variant of the Count-Min sketch involves scaling the counters during the lifetime of the sketch, allowing outdated elements to be gradually forgotten. This is an effective way of handling concept drift, whereby the distribution of the underlying data changes over time [7]. Other variants of the Count-Min sketch exist, mostly aimed at improving the performance of the sketch when it needs to be kept on disk [49].

3.4 Set cardinality with HyperLogLog

3.4.1. Set cardinality

Supposing we want to count how many different songs you have in your record collection. You simply count all the songs by title. If you had multiple copies of a song (e.g. live recordings), you only count them once. This is a set cardinality problem (counting distinct set elements). Set cardinality problems get harder to compute when the set size grows. The classic example is counting unique website views. Counting every unique IP address to visit a website using a hash table or database needs each address to be stored, which for websites with massive traffic requires lots of memory.

3.4.2. HyperLogLog algorithm

HyperLogLog is a sketching algorithm designed to estimate the number of distinct elements in large datasets [50]. Unlike the sketch algorithms looked at so far, which use the ordering of hash values, HyperLogLog is based on **bit-pattern observables**. The bit-pattern observable is the number bits until the leftmost 1 is encountered. For example, 0001010 has three leading 0s before the leftmost 1, so the bit-pattern observable value is 4. We use bit-pattern observable probabilities to estimate set cardinality. The process is akin to flipping a coin; the odds of getting a run of 2 heads before tails is ½ * ½ * ½, which is ⅛ (12.5%). The odds of a run of 3 heads before tails is 6.25%. The odds of getting a run of *N* heads followed by a tails is $1/2^{N+1}$. Rather than a sequence of heads or tails, think of a binary string (e.g. 0001010). The chance of seeing *N* 0s, followed by a 1 is $1/2^{N+1}$.

The key idea behind the HyperLogLog algorithm is that by applying a uniform and deterministic hash function to get a set permutation (à la MinHash), you use bit-pattern observables of the hash values to estimate the number of unique values in the set [8]. If you encounter the hash value 0001010, which has a bit pattern observable of 4, you can estimate you've seen 2^4 distinct values. We use this logic to estimate set cardinality by finding the longest run of 0s. If the longest is *N* 0s and then a 1, you have probably seen around 2^{N+1} elements in your set. However, because this process is random (you might have one element but its hash is 000001, giving an estimated 2^6 elements), we need to average multiple estimates.

To take multiple estimates we use the concept of **registers**. The HyperLogLog sketch is an array of registers; each records a count and is addressable with a unique identifier. By taking multiple estimates and then using **stochastic averaging** to reduce variance, HyperLogLog gives a cardinality estimate within defined error bounds.

To add an element to the sketch, it is hashed and the prefix (first *A* bits) of this value is removed and used to lookup a register in the sketch. The remainder of the hash value (the suffix) is used for the bit-pattern observable; the register is updated if the new bit-pattern observable is greater than the current one (Algorithm 4). To obtain a cardinality estimate from this sketch, the harmonic mean is calculated across all sketch registers and this is the approximate number of distinct elements in the set.

```
$D$ ← input set
$X$ ← HyperLogLog sketch, initialised with 0s in each register
$h$ ← Hash function
A ← Prefix size

for all elements $e$ in $D$ do              // iterate over each element in the set
        $hv$ ← $hv_f(e)$                     // generate a hash value for the element
        p, s ← hv[0:A], hv[A:]           // split hash value into prefix + suffix
        B ← countBits(s)                 // bit-pattern observable for element
        curB ← X[p]                      // current bit-pattern observable in register
        if $B > curB$ then                    // compare bit-pattern observables
                X[p] $\leftarrow B$            // a new maximum value replaces the old
```

Algorithm 4*. Pseudocode for the HyperLogLog algorithm.*

3.4.3. HyperLogLog implementations for genomics

HyperLogLog can estimate the number of distinct k-mers in a set (Figure 2b). One of the earliest implementations of HyperLogLog for genomics was in khmer [28]. HyperLogLog has recently been implemented in the Dashing software for estimation of genomic distances [2]. Similarly to MinHash methods, Dashing uses sketches to summarise genomes and calculates pairwise distances based on k-mer set cardinality. HyperLogLog generally results in faster sketching and greater accuracy compared to MinHash-based methods [2]. In addition, HyperLogLog does not suffer from the same performance degradation as MinHash when dealing with varying set size. However, for distance estimations HyperLogLog can be slower (BinDash). Another limitation is that Dashing cannot report intersections (k-mers common between sets).

HyperLogLog has also been used by krakenUniq for metagenome classification, specifically to approximate how many unique k-mers are covered by reads [25]. This improves upon the original classifier by enabling distinction between false-positive and true-positive database matches. The Meraculous assembler is another example of bioinformatic software that has been optimised using

HyperLogLog. In this case, estimating k-mer cardinality for informing Bloom filter size [51].

3.4.4. Considerations and variations

As already mentioned, the main limitation of HyperLogLog is that it cannot accurately perform set intersection or difference operations. These operations are better suited to algorithms such as MinHash.

HyperLogLog currently has only a few implementations in genomics, with no variants that I am aware of. In the wider data science field there are variants such as HyperLogLog++, which has an updated bias correction scheme [52], and the sliding HyperLogLog, which is designed to operate on data streams [53].

3.5 Other algorithms

I've now covered several common set queries and the sketching algorithms designed to approximate them. There are many more algorithms not covered which are already used in genomic research. For instance, histogram similarity using histoksetch can classify microbiomes, based on incomplete data streams [4]. Another histogram based sketch is ntCard, which uses a multiplicity table of hashed k-mers for estimating k-mer coverage frequencies [54].

The Counting Quotient Filter is a sketch for approximate membership and counting queries, with space usage is as good or better than CountMin sketch [45]. The Counting Quotient Filter is used by the Mantis software to index and search sequence collections; the index being smaller and faster than a sequence bloom tree [55]. It can also be used for constructing weighted de Bruijn graphs [56].

The field is constantly being augmented with new algorithms and implementations. In an effort to keep a current list of sketching algorithms for genomics, please refer to the accompanying repository and file a pull request if tools are missing (https://github.com/will-rowe/genome-sketching/blob/master/references.md).

4. Workflows for genomics

I have included several workflows for genomics that utilise sketching algorithms. These workflows tackle the various stages of an outbreak investigation from a paper by *Reuter et al*. [57], which compared whole genome sequence analysis of bacterial isolates to standard clinical microbiology practice. They demonstrated that genomics enhanced diagnostic capabilities in the context of investigating nosocomial outbreaks caused by multidrug-resistant bacteria. However, the authors relied on intensive analyses such as genome assembly and phylogenetic tree construction. Here, I show that sketching algorithms can replicate their analyses in several minutes, using just 1GB of memory and not needing to store data on disk.

Workflow 1 demonstrates the use of Count-Min sketches and Bloom filters for checking and improving sequence read quality [26,28], as well as showing how MinHash sketches can be used to classify sequence data [16]. **Workflow 2** performs resistome profiling of bacterial isolates using MinHash and Minimizers whilst sequence data is read from online repositories [18,24]. **Workflow 3** replicates the outbreak surveillance of *Reuter et al*., using MinHash distances to build a Newick tree that shared the same topology as the phylogeny from the paper [1,19]. **Workflow 4** augments the analysis from the original paper by using the Bloom filter-based BIGSI to identify additional isolates matching the resistome profile of the outbreak bacteria [27].

5. Conclusions and future directions

Sketching clearly offers many benefits for genomic research. I've shown how sketches can compress, index and search data, using a fraction of the resources needed by other classes of algorithms. A key property of sketches that I've not covered is that they are mergeable, meaning that sketching can be easily parallelised. This makes sketching not only attractive for laptops etc., but also for HPC. Sketching also has applications for data privacy, whereby sketching effectively anonymises confidential data and enables remote analytics. For example, Balaur is a read aligner that sketches data to preserve privacy before outsourcing alignments to cloud compute [58].

There are many exciting genomics applications for sketching that are beginning to be realised. As sketches are relatively stable, a genome sketch is unlikely to change if an assembly is updated, so they are excellent candidates for database indexing and compression. This functionality is already being used by projects such as sourmash, who are able to provide indexed collections of reference sequences which are ready to be interrogated using user-defined sketches [16]. This could allow you to download any genome collection, sketching the download in real-time and using this information to inform downstream analysis, e.g. what genomes to write to disk or to analyse on HPC. This real-time usability of sketches lends them to machine learning applications. We recently showed their utility as succinct representations of microbiome data streams that can be used to predict information about the samples [4,59]. Sketching has clear potential in real-time analytics, such as for monitoring sequencing progress.

In response to the recent adoption of sketching algorithms for genomics, I set out to cover how these algorithms can be used to address some of the challenges we are encountering as genomic data sources continue to grow. Hopefully this has provided an understanding of how sketching algorithms work, their benefits and limitations, and how sketching can be applied to existing genomic workflows.

Acknowledgements

Availability

The source code for the notebooks is available under an MIT License (at https://github.com/will-rowe/genome-sketching) and the live notebooks are available via Binder [60]. The genome data data used in the notebooks is available from the ENA under BioProject PRJEB3353.

Funding

This work was supported in part by the STFC Hartree Centre’s Innovation Return on Research programme, funded by the Department for Business, Energy & Industrial Strategy.

Authors’ contributions

W.P.M.R conceived and wrote the review. W.P.M.R coded and tested the notebooks.

Acknowledgments

I am grateful to Martyn Winn at the STFC for his continued guidance and his feedback on this manuscript.

Conflict of Interest

None declared.

References

1. Ondov BD, Treangen TJ, Melsted P, Mallonee AB, Bergman NH, Koren S, et al. Mash: fast genome and metagenome distance estimation using MinHash. Genome Biol. 2016;17:132.
2. Baker DN, Langmead B. Dashing: Fast and Accurate Genomic Distances with HyperLogLog. bioRxiv. 2019;501726.
3. Zhang Q, Pell J, Canino-Koning R, Howe AC, Brown CT. These Are Not the K-mers You Are Looking For: Efficient Online K-mer Counting Using a Probabilistic Data Structure. PLOS ONE. 2014;9:e101271.
4. Rowe WP, Carrieri AP, Alcon-Giner C, Caim S, Shaw A, Sim K, et al. Streaming histogram sketching for rapid microbiome analytics. Microbiome. 2019;7:40.
5. Cormode G. Data sketching. Commun ACM. 2017;60:48–55.
6. Cormode G. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. Found Trends Databases. 2011;4:1–294.
7. Yang D, Li B, Rettig L, Cudre-Mauroux P. HistoSketch: Fast Similarity-Preserving Sketching of Streaming Histograms with Concept Drift. 2017 IEEE Int Conf Data Min ICDM [Internet]. New Orleans, LA: IEEE; 2017 [cited 2019 Jan 31]. p. 545–54. Available from: http://ieeexplore.ieee.org/document/8215527/
8. Flajolet P, Nigel Martin G. Probabilistic counting algorithms for data base applications. J Comput Syst Sci. 1985;31:182–209.
9. Wei Z, Luo G, Yi K, Du X, Wen J-R. Persistent Data Sketching. Proc 2015 ACM SIGMOD Int Conf Manag Data - SIGMOD 15 [Internet]. Melbourne, Victoria, Australia: ACM Press; 2015 [cited 2019 Feb 9]. p. 795–810. Available from: http://dl.acm.org/citation.cfm?doid=2723372.2749443
10. Gomaa WH, Fahmy AA. A Survey of Text Similarity Approaches. Int J Comput Appl. 2013;68.
11. Broder AZ. On the resemblance and containment of documents. Proc

Compression Complex Seq 1997 Cat No97TB100171. 1997. p. 21–9.
12. Broder AZ. Identifying and Filtering Near-Duplicate Documents. In: Giancarlo R, Sankoff D, editors. Comb Pattern Matching. Springer Berlin Heidelberg; 2000. p. 1–10.
13. Bar-Yossef Z, Jayram TS, Kumar R, Sivakumar D, Trevisan L. Counting Distinct Elements in a Data Stream. In: Rolim JDP, Vadhan S, editors. Randomization Approx Tech Comput Sci. Springer Berlin Heidelberg; 2002. p. 1–10.
14. Cohen E. Size-Estimation Framework with Applications to Transitive Closure and Reachability. J Comput Syst Sci. 1997;55:441–53.
15. Cohen E, Kaplan H. Summarizing data using bottom-k sketches. Proc Twenty-Sixth Annu ACM Symp Princ Distrib Comput - PODC 07 [Internet]. Portland, Oregon, USA: ACM Press; 2007 [cited 2019 Feb 22]. p. 225. Available from: http://dl.acm.org/citation.cfm?doid=1281100.1281133
16. Brown CT, Irber L. sourmash: a library for MinHash sketching of DNA [Internet]. J. Open Source Softw. 2016 [cited 2019 Feb 9]. Available from: http://joss.theoj.org
17. Bovee R, Greenfield N. Finch: a tool adding dynamic abundance filtering to genomic MinHashing [Internet]. J. Open Source Softw. 2018 [cited 2019 Feb 9]. Available from: http://joss.theoj.org
18. Rowe WPM, Winn MD. Indexed variation graphs for efficient and accurate resistome profiling. Bioinformatics. 2018;34:3601–8.
19. Katz LS, Griswold T, Carleton HA. Generating WGS Trees with Mashtree. 2017 [cited 2019 Feb 9]. Available from: https://github.com/lskatz/mashtree
20. Jain C, Dilthey A, Koren S, Aluru S, Phillippy AM. A Fast Approximate Algorithm for Mapping Long Reads to Large Reference Databases. In: Sahinalp SC, editor. Res Comput Mol Biol. Springer International Publishing; 2017. p. 66–81.
21. Jain C, Koren S, Dilthey A, Phillippy AM, Aluru S. A fast adaptive algorithm for computing whole-genome homology maps. Bioinformatics. 2018;34:i748–56.
22. Roberts M, Hayes W, Hunt BR, Mount SM, Yorke JA. Reducing storage requirements for biological sequence comparison. Bioinforma Oxf Engl. 2004;20:3363–9.
23. Li H. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. Bioinformatics. 2016;32:2103–10.
24. Li H. Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics. 2018;34:3094–100.
25. Jackman SD, Vandervalk BP, Mohamadi H, Chu J, Yeo S, Hammond SA, et al. ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter. Genome Res. 2017;27:768–77.
26. Song L, Florea L, Langmead B. Lighter: fast and memory-efficient sequencing error correction without counting. Genome Biol. 2014;15:509.
27. Bradley P, Bakker HC den, Rocha EPC, McVean G, Iqbal Z. Ultrafast search of all deposited bacterial and viral genomic data. Nat Biotechnol. 2019;37:152.
28. Crusoe MR, Alameldin HF, Awad S, Boucher E, Caldwell A, Cartwright R, et al. The khmer software package: enabling efficient nucleotide sequence analysis. F1000Research. 2015;4:900.
29. Breitwieser FP, Baker DN, Salzberg SL. KrakenUniq: confident and fast

metagenomics classification using unique k-mer counts. Genome Biol. 2018;19:198.
30. Koslicki D, Zabeti H. Improving MinHash via the containment index with applications to metagenomic analysis. Appl Math Comput. 2019;354:206–15.
31. Ondov BD, Starrett GJ, Sappington A, Kostic A, Koren S, Buck CB, et al. Mash Screen: High-throughput sequence containment estimation for genome discovery. bioRxiv. 2019;557314.
32. Zhao X. BinDash, software for fast genome distance estimation on a typical personal laptop. Bioinformatics [Internet]. 2018 [cited 2019 Feb 10]; Available from: https://academic.oup.com/bioinformatics/advance-article/doi/10.1093/bioinformatics/bty651/5058094
33. Marcais G, DeBlasio D, Pandey P, Kingsford C. Locality sensitive hashing for the edit distance. bioRxiv. 2019;534446.
34. Li P, König C. b-Bit minwise hashing. Proc 19th Int Conf World Wide Web - WWW 10 [Internet]. Raleigh, North Carolina, USA: ACM Press; 2010 [cited 2019 Mar 7]. p. 671. Available from: http://portal.acm.org/citation.cfm?doid=1772690.1772759
35. Bloom BH. Space/Time Trade-offs in Hash Coding with Allowable Errors. Commun ACM. 1970;13:422–426.
36. Kirsch A, Mitzenmacher M. Less hashing, same performance: Building a better bloom filter. Proc 14th Annu Eur Symp Algorithms ESA 2006. 2006. p. 456–467.
37. Wang Q, Fish JA, Gilman M, Sun Y, Brown CT, Tiedje JM, et al. Xander: employing a novel method for efficient gene-targeted metagenomic assembly. Microbiome. 2015;3:32.
38. Kuśmirek W, Nowak R. De novo assembly of bacterial genomes with repetitive DNA regions by dnaasm application. BMC Bioinformatics. 2018;19:273.
39. Chikhi R, Rizk G. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms Mol Biol. 2013;8:22.
40. Jones DC, Ruzzo WL, Peng X, Katze MG. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. Nucleic Acids Res. 2012;40:e171–e171.
41. Liu Y, Schröder J, Schmidt B. Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data. Bioinformatics. 2013;29:308–15.
42. Heo Y, Wu X-L, Chen D, Ma J, Hwu W-M. BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. Bioinformatics. 2014;30:1354–62.
43. and, Almeida J, Broder AZ. Summary cache: a scalable wide-area Web cache sharing protocol. IEEEACM Trans Netw. 2000;8:281–93.
44. Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD. Cuckoo Filter: Practically Better Than Bloom. Proc 10th ACM Int Conf Emerg Netw Exp Technol [Internet]. New York, NY, USA: ACM; 2014 [cited 2019 Mar 14]. p. 75–88. Available from: http://doi.acm.org/10.1145/2674005.2674994
45. Pandey P, Bender MA, Johnson R, Patro R. A General-Purpose Counting Filter: Making Every Bit Count. Proc 2017 ACM Int Conf Manag Data [Internet]. New York, NY, USA: ACM; 2017 [cited 2019 Mar 14]. p. 775–787. Available from: http://doi.acm.org/10.1145/3035918.3035963
46. Solomon B, Kingsford C. Fast search of thousands of short-read sequencing

experiments. Nat Biotechnol. 2016;34:300–2.
47. Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. J Algorithms. 2005;55:58–75.
48. Cai D, Mitzenmacher M, Adams RP. A Bayesian Nonparametric View on Count-Min Sketch. In: Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R, editors. Adv Neural Inf Process Syst 31 [Internet]. Curran Associates, Inc.; 2018 [cited 2019 Apr 4]. p. 8768–8777. Available from: http://papers.nips.cc/paper/8093-a-bayesian-nonparametric-view-on-count-min-sketch.pdf
49. Eydi E, Medjedovic D, Mekic E, Selmanovic E. Buffered Count-Min Sketch. In: Hadžikadić M, Avdaković S, editors. Adv Technol Syst Appl II. Springer International Publishing; 2018. p. 249–55.
50. Flajolet P, Fusy É, Gandouet O, al et. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. Aofa '07 Proc 2007 Int Conf Anal Algorithms. 2007.
51. Georganas E, Buluç A, Chapman J, Oliker L, Rokhsar D, Yelick K. Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly. Proc Int Conf High Perform Comput Netw Storage Anal [Internet]. Piscataway, NJ, USA: IEEE Press; 2014 [cited 2019 Mar 14]. p. 437–448. Available from: https://doi.org/10.1109/SC.2014.41
52. Heule S, Nunkesser M, Hall A. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. Proc 16th Int Conf Extending Database Technol [Internet]. New York, NY, USA: ACM; 2013 [cited 2019 Mar 7]. p. 683–692. Available from: http://doi.acm.org/10.1145/2452376.2452456
53. Chabchoub Y, Hebrail G. Sliding HyperLogLog: Estimating Cardinality in a Data Stream over a Sliding Window. 2010 IEEE Int Conf Data Min Workshop. 2010. p. 1297–303.
54. Mohamadi H, Khan H, Birol I. ntCard: a streaming algorithm for cardinality estimation in genomics data. Bioinforma Oxf Engl. 2017;33:1324–30.
55. Pandey P, Almodaresi F, Bender MA, Ferdman M, Johnson R, Patro R. Mantis: A Fast, Small, and Exact Large-Scale Sequence-Search Index. Cell Syst. 2018;7:201-207.e4.
56. Pandey P, Bender MA, Johnson R, Patro R. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. Bioinformatics. 2017;33:i133–41.
57. Reuter S, Ellington MJ, Cartwright EJP, Köser CU, Török ME, Gouliouris T, et al. Rapid Bacterial Whole-Genome Sequencing to Enhance Diagnostic and Public Health Microbiology. JAMA Intern Med. 2013;173:1397–404.
58. Popic V, Batzoglou S. A hybrid cloud read aligner based on MinHash and kmer voting that preserves privacy. Nat Commun. 2017;8:15311.
59. Carrieri AP, Rowe WPM, Winn MD, Pyzer-Knapp EO. A Fast Machine Learning Workflow for Rapid Phenotype Prediction from Whole Shotgun Metagenomes. Innov Appl Artif Intell. 2019;
60. Bussonnier M, Freeman J, Granger B, Head T, Holdgraf C, Kelley K, et al. Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. Proc 17th Python Sci Conf. 2018;113–20.