Universidad de Concepción
Facultad de ingeniería

# Tópicos en manejo de grandes volúmenes de datos

## Sketches para estimación de similitud entre genomas

Asignatura: Tópicos en manejo de grandes volúmenes de datos

Integrantes: Javier Leiva Ulloa

Sebastian Rosas

Docente: Cecilia Hernandez

Fecha: 27/09/2024

# Introducción

Para la realización de este trabajo se desarrolló e implementó el algoritmo Hyperloglog para resolver problemas relacionados con la estimación eficiente de cardinalidades en conjuntos grandes, como se vió en la teoría del curso. Para este estudio nos enfocamos en llevar el estudio a la práctica en un análisis de datos de comparación de secuencias genómicas utilizando k-mers y minimizers. El objetivo de este trabajo es analizar la precisión de HyperLogLog en la estimación de esta métrica, comparando los resultados con los valores exactos obtenidos mediante el cálculo tradicional de la similitud de Jaccard. Además, se calcularon métricas de error como el Error Relativo Medio (ERM) y el Error Absoluto Medio (EAM), que proporcionan una medida cuantitativa de la desviación entre las estimaciones probabilísticas y los valores exactos.

**Enlace al repositorio de GitHub:**
**https://github.com/sebargith/Hyperloglog-Jaccard-para-estimar-similitud-de-genomas**

## Elección del dataset

Se eligió un conjunto de datos compuesto por secuencias genómicas del archivo: "GCF_001969825.1_ASM196982v1_genomic.fna". En este archivo encontramos varias secuencias ordenadas una tras de la otra, por lo que decidimos que sería una buena fuente para analizar y comparar.

En nuestro código se comparan las primeras 5 secuencias unas con otras, cada secuencia está separada por un encabezado con su descripción más precisa, de ahí seleccionamos el comienzo y el fin de cada una.

## Implementación de los Hyperloglog

La implementación del algoritmo Hyperloglog se basó en dividir el espacio de datos en varios registros llamados "buckets", donde en ellos se cuenta cuántos ceros consecutivos aparecen en la representación binaria de los valores hash de los datos. A partir de estos valores, se realiza una estimación probabilística de la cardinalidad del conjunto.

Cada instancia de Hyperloglog contiene un vector de registros, que inicialmente están llenos de ceros:

```
// Constructor inicializando los registros con ceros
HyperLogLog::HyperLogLog() : registers(m, 0) {}
```

El número de registros se define por el parámetro p, que representa el número de bits utilizados para indexar los registros. En nuestra implementación, se utiliza un valor p=18, lo que significa que tenemos 2^18 registros (para evitar problemas en caso de secuencias muy grandes).

En este caso, se eligió **SpookyHash** por varias razones:

- Velocidad y eficiencia: SpookyHash es una función de hash diseñada para grandes volúmenes de datos
- Distribución uniforme: SpookyHash garantiza una buena distribución de los valores hash en el espacio de salida
- Flexibilidad: La versión SpookyHash32, utilizada en esta implementación, genera valores de 32 bits a partir de cualquier entrada.

## Procesando los elementos

Cada vez que se agrega un elemento (en este caso, un k-mer de una secuencia genómica), se realiza el siguiente proceso:

1. Se genera el valor hash del elemento utilizando SpookyHash.

   Para la implementación se usó la versión SpookyHash32, que genera un valor de 32 bits a partir de cualquier secuencia de entrada, como lo es un k-mer. Esto asegura una buena distribución para el algoritmo y suficiente espacio para dividir los valores en múltiples registros.

2. Se extraen los primeros ‘p’ bits del valor hash para determinar el índice del registro.

   Los primeros ‘p’ bits del valor hash generado se utilizan para determinar el índice del registro en el que se almacenará la información de este k-mer. En esta implementación, p = 14.

```
// Añadir un elemento al HyperLogLog
void HyperLogLog::add(const std::string &data) {
    uint32_t hashValue = hash(data);

    // Extraer el índice del registro (los primeros p bits del hash)
    int registerIndex = hashValue >> (32 - p);
```

Esto permite dividir los datos de manera uniforme entre los registros del Hyperloglog

## Estimación de la cardinalidad

La estimación de la cardinalidad se basa en la suma armónica de los valores almacenados en los registros. El algoritmo Hyperloglog utiliza una constante de corrección (αMM) que ajusta la estimación en función del número de registros utilizados. La fórmula de estimación es la siguiente:

$$\mathrm{r} = \{R0, R1, ..., Rm - 1\}$$

$$\text{Final Estimate} = \text{constant} \cdot m^2 \cdot \left( \sum_{i=0}^{m-1} \left(2^{-Ri}\right) \right)^{-1}$$

Harmonic mean of the individual estimates

Donde `m` es el número de registros, `R_i` son los valores almacenados en los registros, y αMM es la constante en nuestro código:

```
double HyperLogLog::estimate() {
    double alphaMM = 0.7213 / (1 + 1.079 / m) * m * m;
    double harmonicSum = 0.0;
```

*Se aplican algunas correcciones para estimaciones muy bajas o extremadamente altas.

## Cálculo de los ceros consecutivos en los bits restantes

Los bits restantes se utilizan para contar cuántos ceros consecutivos hay en el valor hash. Esto se realiza utilizando la función "*__builtin_clz*" que es una operación que cuenta los ceros a la izquierda en un número entero de 32 bits.

```
// Usamos __buildtin_clz__ para contar los ceros a la izquierda
int HyperLogLog::countLeadingZeros(uint32_t hashValue) {
    return __builtin_clz(hashValue);
}
```

Una vez que se ha determinado el índice del registro y el número de ceros consecutivos, se actualiza el registro correspondiente. El valor almacenado en el registro es el máximo entre el valor actual y el número de ceros consecutivos que acabamos de calcular.

# Implementación de la similitud de Jaccard, alternativa 1

Iniciamos creando una función que generará los k-mers, los cuales son subsecuencias de longitud dentro de una secuencia más grande. En particular para este trabajo, un k-mer sería una subsecuencia que se extrae de una secuencia de genomas. Para esto se definió la función ‘generateKMers’.

```cpp
// Función para generar k-mers de una secuencia
std::unordered_set<std::string> generateKMers(const std::string& sequence, int k) {
    std::unordered_set<std::string> kmers;
    for (size_t i = 0; i <= sequence.size() - k; ++i) {
        kmers.insert(sequence.substr(i, k));
    }
    return kmers;
}
```

La función ‘generateKMers’ toma como entrada una secuencia de genomas y un valor k que representa la longitud deseada de las subsecuencias (k-mers). La función recorre la secuencia de ADN y extrae todas las subsecuencias consecutivas de longitud k, almacenándolas en un conjunto para asegurar que cada k-mer sea único. Al final, devuelve el conjunto de k-mers únicos que fueron generados a partir de la secuencia.

```cpp
// Creamos instancias de HyperLogLog para cada genoma
HyperLogLog hllA, hllB;
for (const auto& kmer : kmersA) {
    hllA.add(kmer);
}
for (const auto& kmer : kmersB) {
    hllB.add(kmer);
}
```

Cada conjunto de k-mers generado se inserta en una instancia de Hyperloglog, que almacena la información necesaria para estimar la cardinalidad.

## Cálculo de la Similitud de Jaccard

$$\cdot J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Para implementar esto, se utiliza Hyperloglog para estimar las cardinalidades ∣A∣|A|∣A∣, ∣B∣|B|∣B∣ y ∣A∪B∣|A \cup B|∣A∪B∣. El método 'merge' de Hyperloglog permite combinar los datos de dos instancias para calcular la unión.

```
// Función para calcular la similitud de Jaccard
double jaccardSimilarity(HyperLogLog& hllA, HyperLogLog& hllB) {
    double estimateA = hllA.estimate();
    double estimateB = hllB.estimate();

    // Fusionar ambos HyperLogLog para estimar la unión
    HyperLogLog hllUnion = hllA;  // Copiamos hllA en hllUnion
    hllUnion.merge(hllB);         // Fusionamos hllB en hllUnion
    double estimateUnion = hllUnion.estimate();

    // Calcular la similitud de Jaccard
    return (estimateA + estimateB - estimateUnion) / estimateUnion;
}
```

## Comparación de genomas

El programa compara genomas en pares. Para cada par de genomas, genera los k-mers correspondientes, los inserta en instancias de HyperLogLog, y calcula la similitud de Jaccard entre ellos.

```cpp
// Vamos a comparar los genomas de manera par a par
for (size_t i = 0; i < genomes.size(); ++i) {
    for (size_t j = i + 1; j < genomes.size(); ++j) {
        std::cout << "Comparando genoma " << i + 1 << " con genoma " << j + 1 << std::endl;

        // Generamos los k-mers para ambos genomas
        auto kmersA = generateKMers(genomes[i], k);
        auto kmersB = generateKMers(genomes[j], k);

        // Creamos instancias de HyperLogLog para cada genoma
        HyperLogLog hllA, hllB;
        for (const auto& kmer : kmersA) {
            hllA.add(kmer);
        }
        for (const auto& kmer : kmersB) {
            hllB.add(kmer);
        }

        // Calculamos la similitud de Jaccard entre ambos
        double jaccard = jaccardSimilarity(hllA, hllB);
```

## Implementación de los cálculos de error

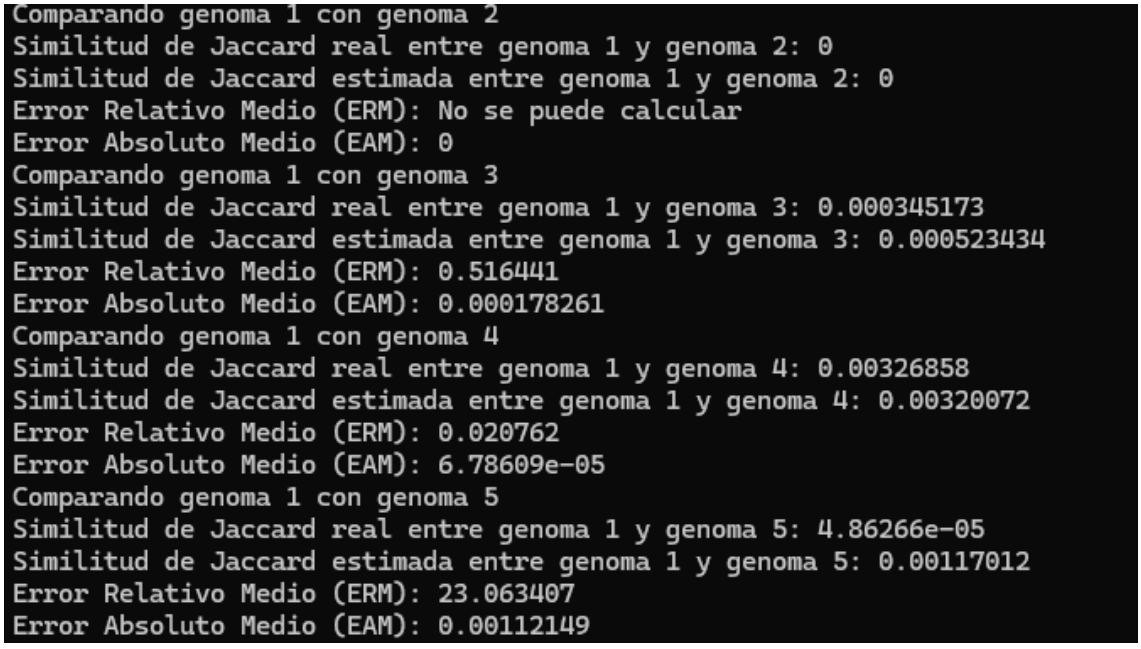
En el código, los cálculos de ERM y EAM se realizan en la función ‘CalculodeErrores’. Esta función toma como entrada las similitudes de Jaccard (real y estimada) y calcula ambos errores. A continuación, imprime los resultados:

```cpp
// Función para calcular ERM y EAM para una comparación específica
void CalculodeErrores(double realJaccard, double estimatedJaccard) {
    double erm = 0.0;
    double eam = 0.0;

    // Evitar dividir por 0 en ERM
    if (realJaccard != 0) {
        erm = fabs(estimatedJaccard - realJaccard) / realJaccard;
    }A

    // Calcular EAM
    eam = fabs(estimatedJaccard - realJaccard);

    std::cout << "Error Relativo Medio (ERM): " << (erm == 0.0 ? "No se puede calcular" : std:
    std::cout << "Error Absoluto Medio (EAM): " << eam << std::endl;
}
```

Durante la ejecución del programa, se comparan genomas de manera par a par. Por cada par de genomas, se calcula la similitud de Jaccard real y la estimada usando Hyperloglog, y luego se calculan los errores ERM y EAM.

# Resultados

En general, cuando la similitud real de Jaccard es distinta de cero, el algoritmo Hyperloglog muestra una buena capacidad para aproximar la similitud real, aunque con cierto margen de error.




Los resultados muestran que para valores de Jaccard reales pequeños, la diferencia entre la estimación y el valor real puede ser significativa, como en el caso de genoma 1 con

genoma 5, donde la similitud real es 0.0000486266 y la estimada es 0.00117012, resultando en un Error Relativo Medio (ERM) elevado (23.06). Esto indica que Hyperloglog tiene dificultades para manejar correctamente similitudes muy pequeñas.

Por otro lado, cuando la similitud de Jaccard real es moderada o alta (por ejemplo, en el caso de genoma 1 con genoma 4, donde la similitud real es 0.00326858), la estimación es mucho más precisa, con un ERM de tan solo 0.0207. Esto indica que el algoritmo es más preciso cuando las similitudes reales son más grandes.

# Conclusión

Si bien a lo largo de la realización de la tarea hubo algunas complicaciones, como al decidir cuál función de hash era conveniente o al estimar la cardinalidad del algoritmo Hyperloglog, se pudieron observar resultados variables a la hora de observar las estimaciones y sus respectivos errores. En los casos donde la similitud es cercana a 0, el algoritmo tiende a generar pequeñas estimaciones no nulas, lo que puede afectar la interpretación de los resultados. Algunas posibles explicaciones a los resultados anómalos podrían ser la elección del tamaño de k, el número de registros o darle un enfoque más preciso los tamaños de datasets varían mucho, como normalizar la muestra.

Como experiencia de esta tarea concluimos que, si bien Hyperloglog es una herramienta eficiente y precisa en la estimación de la cardinalidad y la similitud en grandes conjuntos de datos, es importante considerar sus limitaciones en la estimación de valores muy pequeños, y podría requerirse ajustar sus parámetros para mejorar la precisión en esos casos.