CS 529: Advanced Data Structures & Algorithms
Assignment 7: Johnson-Lindenstrauss and Monotone Boolean Functions

Nathan Chapman, Hunter Lawrence, Andrew Struthers

April 26, 2024

Summary of Johnson-Lindenstrauss Lemma

The Johnson-Lindenstrauss lemma is a fundamental result in mathematics and computer science that deals with dimensionality reduction. Named after William B. Johnson and Joram Lindenstrauss, who independently proved it in the 1980s, the lemma addresses the problem of preserving the pairwise distances between points in high-dimensional Euclidean space while projecting them into lower-dimensional spaces.

The lemma states that for any set of points in a high-dimensional Euclidean space, there exists a low-dimensional subspace onto which the points can be efficiently and linearly mapped, such that the pairwise distances between the points are approximately preserved. Specifically, for any $\epsilon > 0$ and any integer n, given a set of m points in high-dimensional space (originally in $\mathbb{R}^d$), there exists a mapping to a lower-dimensional space ($\mathbb{R}^k$, where k is significantly smaller than d) such that the distances between any pair of points are preserved up to a factor of $(1 \pm \epsilon)$.

1. **Efficiency**: The lemma assures us that such a mapping can be constructed efficiently, meaning that it doesn't require excessive computational resources even for large datasets.
2. **Applications in Machine Learning and Data Mining**: The Johnson-Lindenstrauss lemma has numerous applications in machine learning and data mining. It is particularly useful in dimensionality reduction techniques like Principal Component Analysis (PCA), t-distributed Stochastic Neighbor Embedding (t-SNE), and Locally Linear Embedding (LLE). By reducing the dimensionality of data, these techniques help in visualizing and understanding high-dimensional datasets, while still preserving the essential geometric structure and relationships between data points.
3. **Sparse Signal Processing**: In signal processing, especially in sparse signal processing, where signals are often represented in high-dimensional spaces, the lemma finds applications in compressive sensing. It enables the reconstruction of sparse signals from a small number of linear measurements by projecting them onto a lower-dimensional space while preserving their structure.
4. **Nearest Neighbor Search**: Another significant application is in nearest neighbor search algorithms. By reducing the dimensionality of the feature space, the computational complexity of searching for nearest neighbors is greatly reduced while maintaining the accuracy of the search.
5. **Big Data Analysis**: With the rise of big data, the Johnson-Lindenstrauss lemma provides a powerful tool for dealing with high-dimensional datasets efficiently. It allows for the analysis of large-scale datasets without sacrificing the quality of results.

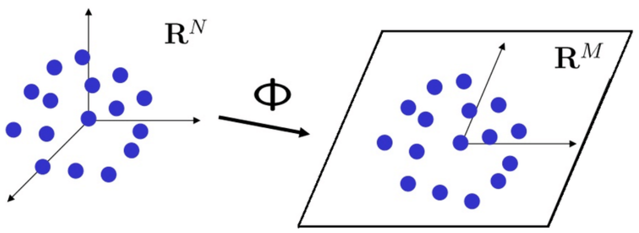

Figure 1: Visualization of Dimensionality Reduction

The Johnson-Lindenstrauss lemma is important in machine learning, data mining, signal processing, and big data analysis by providing a way to reduce the dimensionality of data while preserving its inherent structure and relationships.

Minimal Dimension Reduction

Estimate the minimal dimsension k to preserve n-D distances of 20 arbitrary n-D points within $\pm 5\%$ and $\pm 50\%$ accuracy.

Solution

The Johnson-Lindenstrauss lemma can be stated as

> For any $\epsilon \in (0, 1)$ and any integer $n \geq 15(\log m)/\epsilon^2$, there exists a linear function $f : \mathbb{R}^N \to \mathbb{R}^n$ such that the restriction $f|_X$ is $(1 + \epsilon)$-bi-Lipschitz.

We can equate this notation to our notation as

- The reduced dimension: $n = k$
- The size of the collection of points: $m = 20$
- The percent accuracy: $\epsilon = 5$ or $\epsilon = 50$.

Additionally, the minimal dimension that preserves such accuracy is when $n = \lceil 15(\log m)/\epsilon^2 \rceil$. Therefore, the minimal dimension k that preserves distance with in $\pm\epsilon\%$ for a collection of 20 points is

$$k_5 = \left\lceil \frac{15(\log 20)}{5^2} \right\rceil = 9$$

$$k_{50} = \left\lceil \frac{15(\log 20)}{50^2} \right\rceil = 1$$

Summary of Restoration of Monotone Boolean Functions

Optimal restoration of monotone Boolean functions is a significant problem in computer science, particularly in the field of Boolean function analysis and optimization. Monotone Boolean functions are functions that only increase or stay the same as their inputs increase. The optimal restoration problem involves finding the minimal set of variables to restore in a given faulty Boolean function to make it monotone again. This problem has implications in circuit design, fault tolerance, reliability analysis, and error correction in digital systems.

The optimal restoration of monotone Boolean functions problem aims to identify the smallest subset of variables in a faulty Boolean function that, when restored (or fixed), render the function monotone. Given a faulty Boolean function that has become non-monotone due to some faults or errors in its variables, the objective is to determine the minimum number of variable fixes needed to restore the monotonicity of the function.

Formally, for a given monotone Boolean function f with inputs $x_1, x_2, \ldots, x_n$, and a set of faulty variables $F \subseteq \{x_1, x_2, \ldots, x_n\}$, the goal is to find the smallest subset $R \subseteq F$ such that the function becomes monotone when the variables in R are restored to their original values.

This problem has been extensively studied in the context of fault-tolerant circuit design, where ensuring monotonicity is crucial for correct circuit behavior, especially in critical applications such as safety-critical systems and aerospace engineering.

1. **Algorithmic Approaches**: Several algorithmic approaches have been proposed to solve the optimal restoration problem efficiently. These approaches often leverage properties of monotone Boolean functions and employ techniques such as dynamic programming, branch-and-bound algorithms, and greedy algorithms to identify the minimal restoration set.
2. **Complexity Analysis**: The optimal restoration problem is known to be NP-hard, as it involves finding the smallest subset of variables that satisfies a specific property (monotonicity) in a Boolean function. As a result, heuristic and approximation algorithms are commonly used to tackle large instances of the problem.
3. **Applications in Circuit Design**: In digital circuit design, ensuring the monotonicity of Boolean functions is crucial for maintaining reliable and predictable behavior. The optimal restoration problem helps in identifying and correcting faults in circuits to prevent non-monotonic behavior, which can lead to incorrect operation or even system failure.
4. **Fault Tolerance and Reliability Analysis**: Monotone Boolean functions are used in modeling various aspects of fault-tolerant systems and reliability analysis. By studying the optimal restoration problem, researchers can develop techniques to enhance the fault tolerance and reliability of digital systems by efficiently identifying and correcting faults.
5. **Error Correction in Digital Systems**: In addition to fault tolerance, the optimal restoration problem has implications for error correction in digital systems. By restoring non-monotone functions to monotone ones, errors introduced during computation or communication can be mitigated, leading to improved system performance and accuracy.

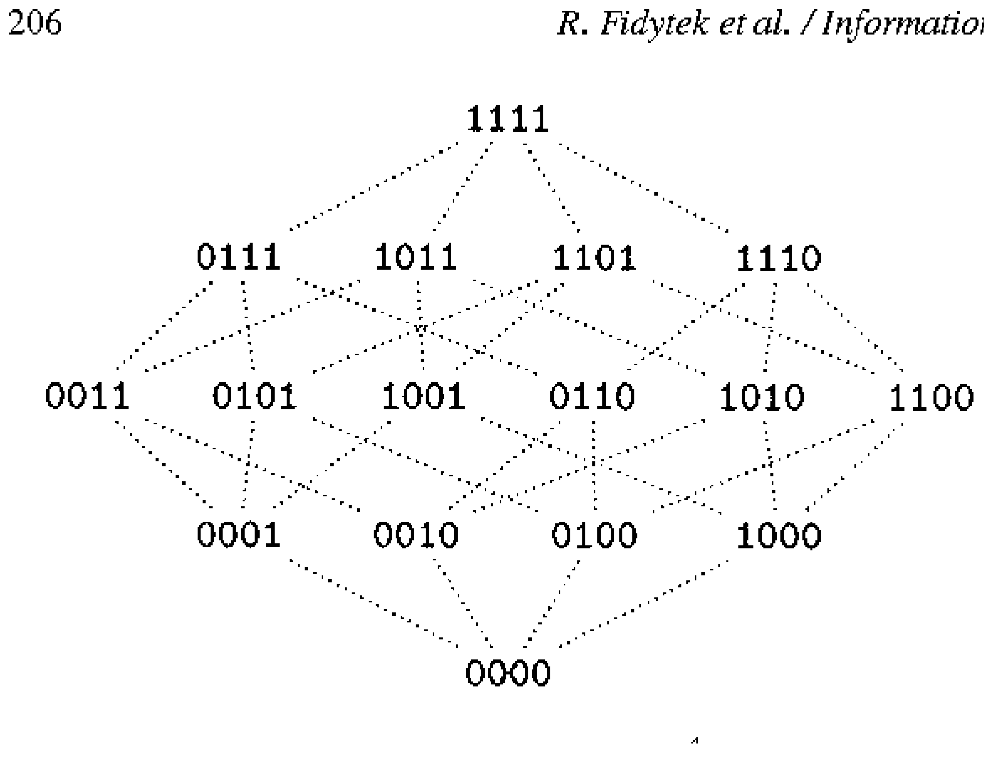

Figure 2: Visualization of a Boolean function

Constructing Hansel Chains for Monotone Boolean Functions

The first step required to restore a function is to construct a sorted list of Hansel Chains using the functions $createHanselChains()$. After this is completed, it is necessary to initialize $restoredFunctionValue$ to infinity so that even large values of the boolean function $f(x)$ will be caught when checking the chains for the function value.

From here, we reconstruct the function by checking for link heads that do not produce a positive result for $f(x)$, when one is found, it is iterated through until the smallest value in the chain that produces a positive value for $f(x)$ is found. This value is then converted to decimal and, if the vector's decimal value is less than $restoredFunctionValue$, is saved as the $restoredFunctionValue$. When this loop is completed, and all chain heads are checked, the value of $restoredFunctionValue$ will represent the function value for $f(x)$.

```
Algorithm 1: $generateBooleanFunction(f(x))$
Input: A function to re-construct a given monotone boolean function $f(x)$ and the number of dimensions for the
       solution vectors $dimensions$
Output: The function value to restore (assuming the boolean function returns true if above this value to restore)
Assumes: The presence of some function $getDecimalValue()$ which returns the decimal value of a binary vector.
$hanselChains \leftarrow createHanselChains(dimensions)$
$restoredFunctionValue \leftarrow \infty$
$numChains \leftarrow hanselChains.length()$
$i \leftarrow 0$
while $i < numChains$ do
    // for each chain head that does not produce a positive value
    if $!f(hanselChains[i][0])$ then
        // find the first (lowest) index that will produce a 1
        $chainLength \leftarrow hanselChains[i].length()$
        $j \leftarrow 0$
        while $j < hanselChains[i].length()$ do
            if $f(hanselChains[i][j])$ then
                // if the current vector is less than the best so far, save it as $restoredFunctionValue$
                if $hanselChains[i][j].getDecimalValue() < restoredFunctionValue$ then
                    $restoredFunctionValue = hanselChains[i][j].getDecimalValue()$
            $j++$
    $i++$
return $restoredFunctionValue$
```

.

Algorithm 2: *createHanselChains*(n)

```
Input: An integer $n$ representing the length of each boolean vector
Output: An unsorted two dimensional list of Hansel Chains and the vectors in each
Assumes: The availability of some function $sort()$, which will sort chains based on the total value of their first
  vector in order from highest to lowest.
$hanselChains \leftarrow <0, 1>$
$i \leftarrow 1$
while $i < n$ do
    $numChains \leftarrow 2^(i-1)$
    // clone and grow all current chains
    $cloneAndGrowChains(hanselChains, numChains)$
    // cut and add to newly changed chains
    $cutAndAddChains(hanselChains, numChains)$
    $i{+}{+}$
$hanselChains.sort()$
return $hanselChains$
```

Algorithm 3: *cloneAndGrowChains*($chains, numChains$)

```
Input: A three dimensional array of chains $chains$, and a number of currently available chains $numChains$
Result: Clones and grows present chains
Assumes: The presence of some function $clone()$ that copies an array, some function $insert(index, value)$ that
  places a $value$ at a provided index $index$ and shifts all other values right, and some function $add(value)$ that places
  a $value$ at the last index of an array.
$i \leftarrow 0$
while $i < numChains$ do
    // clone each chain
    $newChain \leftarrow chains[i].clone()$
    // grow each chain
    $newChain.insert(0, 0)$
    $chains[i].insert(0, 1)$
    $chains.add(newChain)$
    $i{+}{+}$
```

Algorithm 4: *cutAndAddChains*($chains, numChains$)

```
Input: A three dimensional array of chains $chains$, and a number of currently available chains $numChains$
Result: Cuts and grows present chains
Assumes: The presence of some function $add(value)$ that places a $value$ at the last index of an array, some
  function $getLastIndex()$ which gets the last index of an array, some function $deleteLastIndex()$ which removes the
  last index of an array.
$i \leftarrow 0$
while $i < numChains$ do
    // move the last vector from the clone chain to original chain
    $chains[i].add(chains[numChains + i].getLastIndex())$
    $chains[numChains + i].deleteLastIndex()$
    $i{+}{+}$
```