MiniProject Final Report

Pagliarini, Giovanni
A0209803H
e0454254@u.nus.edu

Dumery, Corentin
A0198927E
e0404017@u.nus.edu

November 17, 2019

Abstract

Given the problems derived from having to deal with high-dimensional data, methods for dimensionality reduction are commonly used. Two main problems arise when reducing dimensionality: one is to limit the loss of data and still mantain the important information; the second is to avoid expensive solutions. In this work we: (1) introduce dimensionality reduction; (2)analyse a specific reduction method called Fast Johnson-Lindenstrauss Transform (FJLT); (3) experimentally compare its performances with other techniques, ultimately giving some practical recommendations.

Introduction

Nowadays, in the era of “Big Data”, large availability of both computational power and storage allows us to keep and process larger datasets. This is good news, because it increasingly allows for more knowledge to be discovered; however, the size of a datasets is still likely to be an impediment in many ways, and not solely due to a higher cpu workload and amount of memory required [1].

Consider a common problem in Data Science: clustering n datapoints in a space $\mathbb{R}^d$. How can this problem escalate? The number of datapoints can becomes very large, and in this course we studied sampling as a helpful tool for handling datasets with many entries. But what should we do when, instead of the number of entries, it is the size of each entry, namely the *dimensionality* of the space, that becomes inconcievably large (e.g $d = 10^{100}$)?

High-dimensional data might not be treatable, for instance when the task to carry out relies on d being a small constant, i.e algorithms exponential in d. A second major problem is then given by the fact that higher dimensional spaces lead to unexpected and counterintuitive behaviors of the geometric landscape itself, a phenomenon known as “curse of dimensionality”. To name one of the side effects of data being distributed across more dimensions, it is known, for instance, that the euclidean distance between any pair of points of a distribution tends to approach the same value (the points essentially become uniformly distant from each other) [2]. In general, when data distribution of "same patterns" (e.g a think of a Gaussian distribution) is distributed on higher dimensions, the same intuitions we have for the 2D and 3D cases cannot be adopted anymore to make simple assumptions (see [3] for more).

Because of this, research has given birth to different methods for shrinking the dimensionality of datasets, prior to using them. Of course, dimensionality reduction leads to an unavoidable loss of information, so the real quest of dim. reduction techniques is to limit this effect. Luckily, when performing a task such as clustering high-dimensional data, it is expected for most of the information to somehow be redundant, and that there is a way of achieving compression while still maintaining what matters for the task.

At this point, some benefits for doing so are straightforward: less resources needed for computation (for some algorithms this may result in exponential terms being removed from the study of complexity) and alleviating the curse of dimensionality. Moreover, dimensionality reduction can also actively remove redundancies and noise, allowing for actual higher accuracies: think of, for example, removing a feature that does not contribute to solving the task and, instead, was preventing other important features to "speak up".

A final high-level note on dim. reduction: a closer look to what dim. reduction actually is will reveal that the final result achieved on data is quite similar to that of clustering/classification methods, namely summarising the information. However, clustering and classification represent generic concepts of analysing and using data for prediction, whereas dimensionality reduction, at least in this context, comes to aid at an earlier stage. So really, the scope of dim. red. is to find an intermediate representation that (still) allows an algorithm to clearly identify the important information [1].

In the following section we are going to list some common approaches to dimensionality reduction and introduce/analyse the Fast Johnson-Lindenstrauss Transform (or FJLT). Finally, we are going to describe our implementation and experiment design in comparing the performances of this method with others, as well as explain our results.

Background

Formally, a good dim. reduction can be seen as a function $f : \mathbb{R}^d \to \mathbb{R}^k, k \ll d$, (namely a function that maps datapoints to a smaller space) such that f doesn't disrupt the inner structure of the data. There are two main approaches for dim. reduction. The first consists in considering a subset of the original dimensions. Starting from the bottom, for example, one may want to drop a variable with low variance. In fact, considering the extreme case of a variable which is constant for any data entry (null variance): the variable doesn't really have any predictive power, it doesn't tell apart in any way the data entries. Similarly, when two or more variables are codependent with each other, then one of them may be dropped, since, for every datapoint, it could be inferred from the others. More sophisticated approaches involve deriving a new set of variables, rather than dropping some of the originals: for example, methods like Factor Analysis or Principal Components Analysis try to derive new dimensions of maximum variance and eliminating correlated variables.

Linear Projections While these methods consider statistical information of the dataset on its whole, which leads to each datapoint influencing how other datapoints are transformed, projections from a d-space to a k-space can also be computed by applying simple linear functions. Within this context, it is convenient to represent data in the form of a $n \times d$ matrix A where each row represents a point in $\mathbb{R}^d$. The reduced datasets is then obtained by post-multiplying A with a $d \times k$ matrix Φ, embodying the way the original dimensions relates to the new ones. This is equivalent to projecting each datapoint onto a linear k-hyperplane in the d-space.

Johnson-Lindenstrauss Transforms For limiting the level of data disruption that a reduction f causes, a commonly-considered desired property requires the pairwise L2 distances in the original space to not be perturbed too much during the reduction:

$$\exists \epsilon \in [0,1], \forall u, v \in X \subset \mathbb{R}^d, (1-\epsilon)\|u-v\|^2 \leq \|f(u)-f(v)\|^2 \leq (1+\epsilon)\|u-v\|^2$$

A well-known result which goes by the name of "Johnson-Lindenstrauss lemma" states that, for k large enough, for any input set of datapoints, a function with this property exists [4]. Different authors have tried to find efficient ways to compute such projections, and many of the works in this fields use randomized routines. In *Database-friendly random projections: Johnson-Lindenstrauss with binary coins*, D. Achlioptas describes routines where the elements of the matrix are randomly and independently sampled, for example, in $-1, 1$ according to a naive distributions[1] [5].

However, since computing the linear matrix multiplication can be too expensive for large datasets, research has been done in trying to achieve a complexity smaller than $\mathcal{O}(ndk)$ leveraging the fact that, if Φ is sparse enough, then more efficient implementation for matrix multiplication can be used. Enter the *Fast Johnson-Lindenstrauss Transform* (FJLT), introduced by Ailon N. and Chazelle B. in [6]. In the next section, we will focus on explaining how FJLT works; we will describe how it is computed, prove it is a satisfying JL transform, and conclude by showing how much faster it is exactly. For more see [7, 8].

FJLT

Computing the FJLT is rather simple. We define Φ as the product of three matrices P, H, D described below:

- P is a $k \times d$ random matrix actually downsizing the dimensions.

[1] But note that proof given in the paper holds for a high-enough k only.

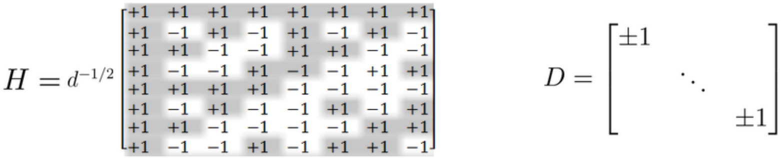
  - Reasoning: A simple way to introduce new dimensions is to make linear combinations of the old ones. For example, we could define our first new dimension as: for each entry x, define $x'_1 = x_2 - x_7 - x_9 + \dots$ where the sign in front of each component is random (but, of course, the same for every entry). Note that P alone is enough for reducing the dimensionality (indeed, a similar approach is used in [5] mentioned above) , but we will see how we can do better by introducing H and D.

    Also, we would like P to be sparse so that every new dimension is a *different aggregation* of only a few dimensions, rather than many.
  - How: Define a sparsity constant $q = \frac{\log n}{\epsilon d}$, where n is the number of entries. With probability $1 - q$, set each $P_{i,j}$ to 0; otherwise (probability q) pick a random value from the distribution $\mathcal{N}(0, q^{-1})$.

$$\begin{bmatrix} 0 & 1.0 & 0 & 0 \\ 1.0 & 0 & -0.3 & 1.1 \\ 0 & -0.3 & 0 & 0 \\ 2.5 & 0 & 0 & 6.0 \end{bmatrix}$$

Figure 1: Visualization of a *potential* P (P is randomly generated)

- D and H (shape: both $d \times d$)
  - Reasoning: Our current transform P might not be good enough; Consider the datapoint $x := (1, 0, \dots, 0)$: our matrix P has only a q-fraction of its elements that are non-zero, therefore $f(x) = Px$ will have a high chance of having only zeroes, having lost all of the input information. Similarly, if some values are significantly larger than the others, these will likely dominate during the aggregation. We want to solve these issues by "smoothing" x before projecting it. First, a matrix D randomly switches the signs of the original dimensions, so that, in the aggregation step, large values will have a higher chance to cancel out, ultimately preventing bigger values and leading to a more homogeneous vector. Secondly, a matrix H is used to redistribute the initial values more evenly, so as to avoid sparse vectors.
  - How: D is a random diagonal matrix with only ± 1 (with equiprobability) on the diagonal; $H_{i,j} = d^{-1/2}(-1)^{\langle i-1, j-1 \rangle}$, where $\langle \, , \rangle$ denotes the dot product of the binary decompositions.



$$H = d^{-1/2}\begin{bmatrix} +1 & +1 & +1 & +1 & +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 & +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 & -1 & -1 & +1 & +1 \\ +1 & +1 & +1 & +1 & -1 & -1 & -1 & -1 \\ +1 & -1 & +1 & -1 & -1 & +1 & -1 & +1 \\ +1 & +1 & -1 & -1 & -1 & -1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 & +1 & +1 & -1 \end{bmatrix} \qquad D = \begin{bmatrix} \pm 1 & & \\ & \ddots & \\ & & \pm 1 \end{bmatrix}$$

Figure 2: Visualization of the H and D matrices for $d = 8$

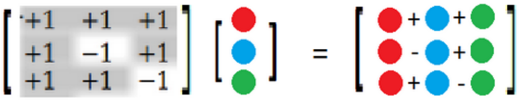


Figure 3: Visualization of how H spreads out the vector

Demonstration

Let x be a vector in $\mathbb{R}^d$. Without loss of generality, assume $\|x\|_2 = 1$. We will first give an overview of the proof.

- *Claim 1:* After multiplying x by HD, values cannot be arbitrarily big; with high probability:
$$max\|HDx\|_\infty = \mathcal{O}(d^{-1/2}\sqrt{\log n})$$
- *Claim 2:* $(1-\epsilon)\sqrt{q} \le \mathrm{E}[\sqrt{Z}] \le \sqrt{q}$ (where $Z = \sum_{i=1}^{d} b_i u_i^2$)
- *Claim 3:* The expectation of $\|y\|_1$ is an ϵ-factor approximation of $\|x\|_1$, multiplied by a constant factor α:
$$(1-\epsilon)\alpha\|x\|_1 \le \mathrm{E}[\|y\|_1] \le (1+\epsilon)\alpha\|x\|_1$$
- *Claim 4:* $\|y\|_1$ is sharply concentrated around its mean:
$$\Pr[(1-\epsilon)\mathrm{E}[\|y\|_1] \le \|y\|_1 \le (1+\epsilon)\mathrm{E}[\|y\|_1]] \ge 1 - 1/20$$

Overview: Claim 3 and *Claim 4* are the properties that we want our transform to have. However, because of how complex our pre-processing is, we will first need to show the two other claims. *Claim 1* allows us to assume there exists an s such that $|u_1| \le s$, which is crucial to ensure the convergence of the series we will use for *Claim 3.* It is also used in *Claim 4* to deal with orders : it tells us $|u|$ can be neglected because it is bounded. *Claim 2* gives us a property over the pre-processed data u that will be useful when we'll need a similar property over the output y in *Claim 3.*

Claim 1: After multiplying x by HD, values cannot be arbitrarily big; with high probability:

$$max\|HDx\|_\infty = \mathcal{O}(d^{-1/2}\sqrt{\log n})$$

Proof: Let $u := HDx$. It follows from the definitions of H and D that $u_1 = \sum_{i=1}^{d} a_i x_i$, where $a_i = \pm d^{-1/2}$. Thanks to D, the sign of a_i is positive or negative each with a $1/2$ chance. This is what D is designed for and it will allow us to rewrite the expectancy and easily find an upper bound. We will use the moment generating function. For any $s > 0$, we have by symmetry:

$$\Pr[|u_1| \geq s) = 2\Pr[e^{sdu_1} \geq e^{s^2 d}]$$

We would like to apply Markov's inequality, but need to bound $\mathrm{E}[e^{sdu_1}]$ first. Let's use the fact that $\mathrm{E}[e^{sda_i x_i}] = (\frac{1}{2}e^{s\sqrt{d}x_i} + \frac{1}{2}e^{-s\sqrt{d}x_i}) = cosh(s\sqrt{d}x_i)$:

$$\begin{aligned}
\mathrm{E}[e^{sdu_1}] &= \prod_{i=1}^{d} \mathrm{E}[e^{sda_i x_i}] \\
&= \prod_{i=1}^{d} cosh(s\sqrt{d}x_i) \\
&\leq \prod_{i=1}^{d} e^{s^2 d x_i^2/2} && \text{(using } cosh(x) \leq e^{x^2/2}\text{)} \\
&\leq e^{s^2 d\|x\|_2^2/2} && (1)
\end{aligned}$$

We can now apply Markov's inequality with $\alpha = e^{s^2 d}$:

$$\begin{aligned}
\Pr[|u_1| \geq s) &= 2\Pr[e^{sdu_1} \geq e^{s^2 d}] \\
&\leq 2\mathrm{E}[e^{sdu_1}]/e^{s^2 d} \\
& \qquad \text{(Markov's inequality)} \\
&\leq 2e^{s^2 d(\|x\|_2^2/2-1)} && \text{(using (1))} \\
&\leq 2e^{-s^2 d/2} && \text{(assuming } \|x\|_2 = 1\text{)} \\
\Pr[|u_1| \geq d^{-1/2}\sqrt{\log(40n)}] &\leq 2e^{-\log(40n)/2} \\
& \qquad \text{(setting } s = d^{-1/2}\sqrt{\log(40n)}\text{)} \\
&\leq 1/(20nd) \\
\Pr[\|u\|_\infty \geq d^{-1/2}\sqrt{\log(40n)}] &\leq 1/(20n) \\
\Pr[\exists x \in X, \|HDx\|_\infty \geq d^{-1/2}\sqrt{\log(40n)}] &\leq 1/20
\end{aligned}$$

□

Recall that we have defined the sparsity constant $q = \frac{\log n}{\epsilon d}$. We also define $y = Pu = PHDx$. By definition of P, we can write y_1 as $y_1 = \sum_{i=1}^{d} r_i b_i u_i$ where b_i equals 0 with probability q and 1 otherwise, and r_i follows a $\mathcal{N}(0, q^{-1})$ distribution.
We can now define the random variable $Z = \sum_{i=1}^{d} b_i u_i^2$.

Claim 2: $(1-\epsilon)\sqrt{q} \leq \mathrm{E}[\sqrt{Z}] \leq \sqrt{q}$

This proof is left as an exercise to the reader. It is lengthy and uses concepts we have not seen in class. *(Hint: you will need to define a d-dimensional polytope where u^2 lives and identify its extremal cases thanks to its vertices)*
To give you an intuition, this property over Z will be used later to show Claim 3 over y (which is, in the end, what matters here). Z is just a tool that satisfies Claim 2 and thanks to this, once we link that Z with y, we will be able to deduce a corresponding property over y.

Claim 3: The expectation of $\|y\|_1$ is an ϵ-factor approximation of $\|x\|_1$, multiplied by a constant factor α:

$$(1-\epsilon)\alpha\|x\|_1 \leq \mathrm{E}[\|y\|_1] \leq (1+\epsilon)\alpha\|x\|_1$$

Proof: Since $y_1 = \sum_{i=1}^{d} r_i b_i u_i$ and $Z = \sum_{i=1}^{d} b_i u_i^2$, we have $(y_1|Z = z) \sim \mathcal{N}(0, q^{-1}z)$. It is also well known that the expectation of the absolute value of $\mathcal{N}(0,1)$ is $\sqrt{2\pi^{-1}}$. Therefore with conditional expectations we have:

$$\mathrm{E}[|y_1|] = \sqrt{\frac{2}{q\pi}}\mathrm{E}[\sqrt{Z}]$$

Recall that we assume $\|x\|_2 = 1$. Claim 2 gives us the following:

$$\begin{aligned}\sqrt{q}(1-\epsilon)\sqrt{\frac{2}{q\pi}} \leq \mathrm{E}[|y_1|] &\qquad \leq \sqrt{q}\sqrt{\frac{2}{q\pi}}\\ (1-\epsilon)\sqrt{\frac{2}{\pi}}\|x\|_2 \leq \mathrm{E}[|y_1|] &\qquad \leq (1+\epsilon)\sqrt{\frac{2}{\pi}}\|x\|_2\end{aligned}$$

Furthermore, all dimensions are treated equally so $\mathrm{E}[\|y\|_1] = k\mathrm{E}[|y_1|]$. This gives us:

$$\begin{aligned}(1-\epsilon)\sqrt{\frac{2}{\pi}}\|x\|_2 \leq \frac{\mathrm{E}[\|y\|_1]}{k} &\qquad \leq (1+\epsilon)\sqrt{\frac{2}{\pi}}\|x\|_2\\ (1-\epsilon)k\sqrt{\frac{2}{\pi}}\|x\|_2 \leq \mathrm{E}[\|y\|_1] &\qquad \leq (1+\epsilon)k\sqrt{\frac{2}{\pi}}\|x\|_2\end{aligned}$$

All we have to do now to conclude is set $\alpha = k\sqrt{2\pi^{-1}}$. □

Claim 4: $\|y\|_1$ is sharply concentrated around its mean:

$$\Pr[(1-\epsilon)\mathrm{E}[\|y\|_1] \leq \|y\|_1 \leq (1+\epsilon)\mathrm{E}[\|y\|_1]] \geq 1 - 1/20$$

Proof: Without loss of generality, we will focus on the right side of the inequality. A similar proof holds for the other side. Let $\lambda > 0$. We can use the Taylor expansion of the exponential function on the moment generating function of $|y_1|$ to obtain:

$$\mathrm{E}[e^{\lambda|y_1|}] = 1 + \lambda\mathrm{E}[|y_1|] + \sum_{t>1} \frac{\mathrm{E}[|y_1|^t]\lambda^t}{t!}$$

We will use the following theorem, which is well known in the literature: *if* $U \sim \mathcal{N}(0,1)$*, then* $\mathrm{E}[|U|^t] = \mathcal{O}(t)^{t/2}$.
Our expansion becomes:

$$\begin{aligned}\mathrm{E}[e^{\lambda|y_1|}] &= 1 + \lambda\mathrm{E}[|y_1|] + \sum_{t>1} \frac{\mathcal{O}(t)^t\lambda^t}{t!} \\ &= 1 + \lambda\mathrm{E}[|y_1|] + \mathcal{O}(\lambda^2) \\ &= e^{\lambda\mathrm{E}[|y_1|] + \mathcal{O}(\lambda^2)}\end{aligned}$$

We can now apply this to $\|y\|_1$, since the dimensions are independent:

$$\begin{aligned}\mathrm{E}[e^{\lambda\|y\|_1}] &= \mathrm{E}[e^{\lambda|y_1|}] \\ &= e^{\lambda k\mathrm{E}[\|y\|_1] + \mathcal{O}(\lambda^2 k)}\end{aligned}$$

Finally we are able to apply the results given by the generating function on the initial norm:

$$\begin{aligned}\Pr[\|y\|_1 \geq (1+\epsilon)\mathrm{E}[\|y\|_1]] &= \Pr[e^{\lambda\|y\|_1} \geq e^{\lambda(1+\epsilon)\mathrm{E}[\|y\|_1]}] \\ &\leq \mathrm{E}[e^{\lambda\|y\|_1}]/e^{\lambda(1+\epsilon)\mathrm{E}[\|y\|_1]} \\ &\qquad \text{(using Markov's inequality)} \\ &\leq e^{-\lambda\epsilon\mathrm{E}[\|y\|_1] + \mathcal{O}(\lambda^2 k)} \\ &\leq e^{-\Omega(\epsilon^2 k)} \\ &\qquad \text{(setting } \lambda = \epsilon \text{ and since } \mathrm{E}[\|y\|_1] = \mathcal{O}(k)\text{)}\end{aligned}$$

This shows that is now possible to set k such that this probability is below 1/40. In particular, one would need to choose k at least of the order of ϵ^{-2}. □

To conclude, we have shown that (1) the expectancy of $\|y\|_1$ is unbiased and (2) the result is sharply concentrated around it. Therefore, the FJLT defined in the previous section can be used with $k = \mathcal{O}(\epsilon^{-1/2})$ with a high probability of success.

Time analysis

We have shown that this transform is satisfying, but is it indeed faster than existing work? We have explained earlier than existing methods are computed in $\mathcal{O}(d^2)$. We want to achieve a significantly better time complexity, even though we are now multiplying by PHD.

- P is $q = \frac{\log n}{\epsilon d}$ sparse. It can therefore be computed q times faster than $\mathcal{O}(d^2)$, i.e. it is computed in $\mathcal{O}(\frac{d \log n}{\epsilon})$.
- H is not sparse at all, but the Walsh-Hadamard matrix is known to be computable in $\mathcal{O}(d \log d)$ time thanks to its simple FFT.
- D is a diagonal matrix and only requires $\mathcal{O}(d)$ as a result.

Overall, we can conclude that this product $PHDx$ can be computed in $\mathcal{O}(\frac{d \log n}{\epsilon} + d \log d)$ time. This is a great improvement, as long as d is much greater than $\log n$; otherwise, it may be preferable to consider simpler methods.

In the following sections, we will see what results we get for various approaches. Our FJLT may allow us to gain some time, but it remains to see in practice if the results it gives are comparable to that of the simpler methods. We will compare its performance on different problems. We will also try to answer the questions: which approach works best when we halve the number of dimensions? Similarly, which approach should we choose to reduce the dimension by a factor of 100?

Implementation

We implemented a few dim. reduction methods in Python 3, and quantified their goodness. A few factors are at play. The general flow is as follows: first, the reduction method is used on a dataset, to obtain a reduced dataset; a test problem is performed on the original and reduced datasets. If the reduction method is good enough, the result on the reduced dataset should not be much worse than the one obtained on the original dataset. More generally, we need a metric for comparing the two outputs.

Methods For the implementation of FJLT, we first tried our own implementation, but we couldn't achieve the expected results, therefore we turned to open-source code and found an implementation by a PhD student at the University of Utah [9]. The code is optimised for vectorization and (allegedly) leverages the sparseness of the matrices. In particular, D is instantiated as a 1D `numpy` array of the size of d's next closest power or 2. The Hadamard transform used comes from the open-source world yet again [10]. Finally, P is instantiated as a `sparse.csr_matrix`.

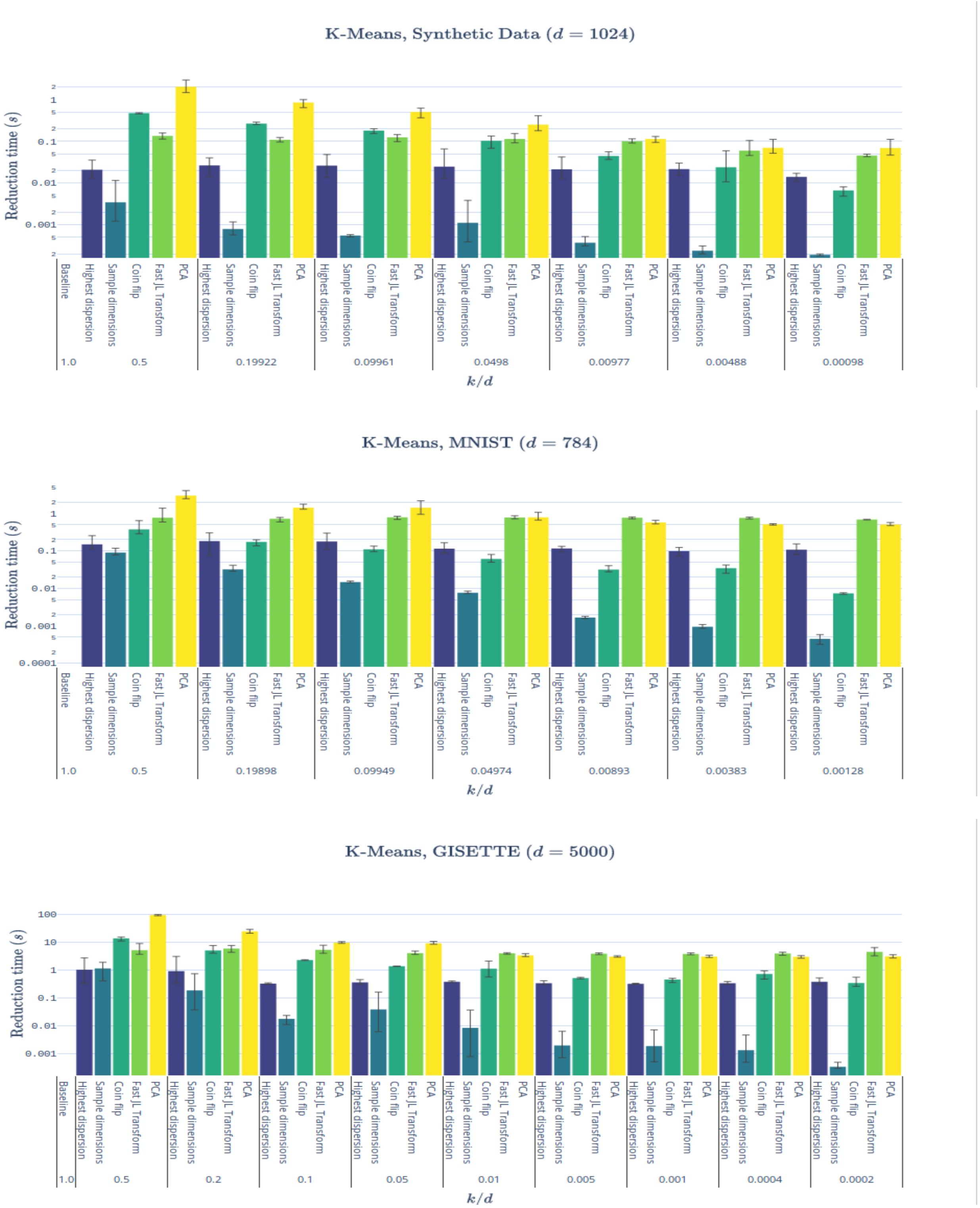
We also implemented simpler methods for reducing dimensionality, such as low-dispersion filter (namely, selecting the dimensions with highest normalized-variance) and dimensions sampling, and tried using PCA, together with other out-of-the-box algorithms available with Python's scikit-learn module. Finally, we added one of the "coin-flips" transforms described by Achlioptas [5]. The idea is simple: construct a matrix R where any $R_{i,j}$ is randomly assigned a value in $\{-1, 0, 1\}$ with respective probabilities $\{\frac{1}{6}, \frac{2}{3}, \frac{1}{6}\}$. The transform is then given by the product $\frac{1}{\sqrt{k}} A \times R$.

Test problem + Metrics We decided to limit this study to clustering, and in particular, with K-means and K-Nearest Neighbor clustering, mainly because the both let us indicate the number of clusters, whereas different clustering routines do not allow this. We used scikit-learn's implementations, and parallelised the by passing `n_jobs=-1` to the instance constructors. The interesting metrics here are the time required by the reduction and the "goodness" of the reduction. As for this last metric, we decided to measure the similarity between clusterings with an entropy-based metric called "validity measure", or v-measure [**?**], consisting in a $[0, 1]$-value.

Given a dataset of labeled entries we perform, say, K-Means in the original space, obtaining a clustering which approximates the ground-truth clustering (the one induced by the dataset's classes). We call the result of K-means on the original data our "baseline clustering". Then, apply dim. reduction, perform K-means in the reduced space, obtaining a "reduced" clustering. Finally, we compare the v-measure of the baseline against the ground-truth clustering, with the v-measure of the reduced clustering with the ground-truth clustering. Note that this allows us to potentially see some cases where the reduction actively helps the clustering routines in deriving a better solution.

Datasets We chose to work with labeled data. We first experimented with a subset of the famous MNIST dataset of handwritten digits [11], with 10 clusters, as much as 10000 entries distributed across 784 dimensions. Next, we used a dataset Gisette [12] as downloaded from [12]. Derived from two MNIST's clusters ("fours" and "nines") but augmented by adding probes variables, it has a dimensionality of 5000 and 7000 datapoints. Finally, we chose to compare with some synthetic data, and found datasets tailor-made for clustering, with 16 non-intersecting clusters and dimensionalities up to 1024 [13, 14].

Results

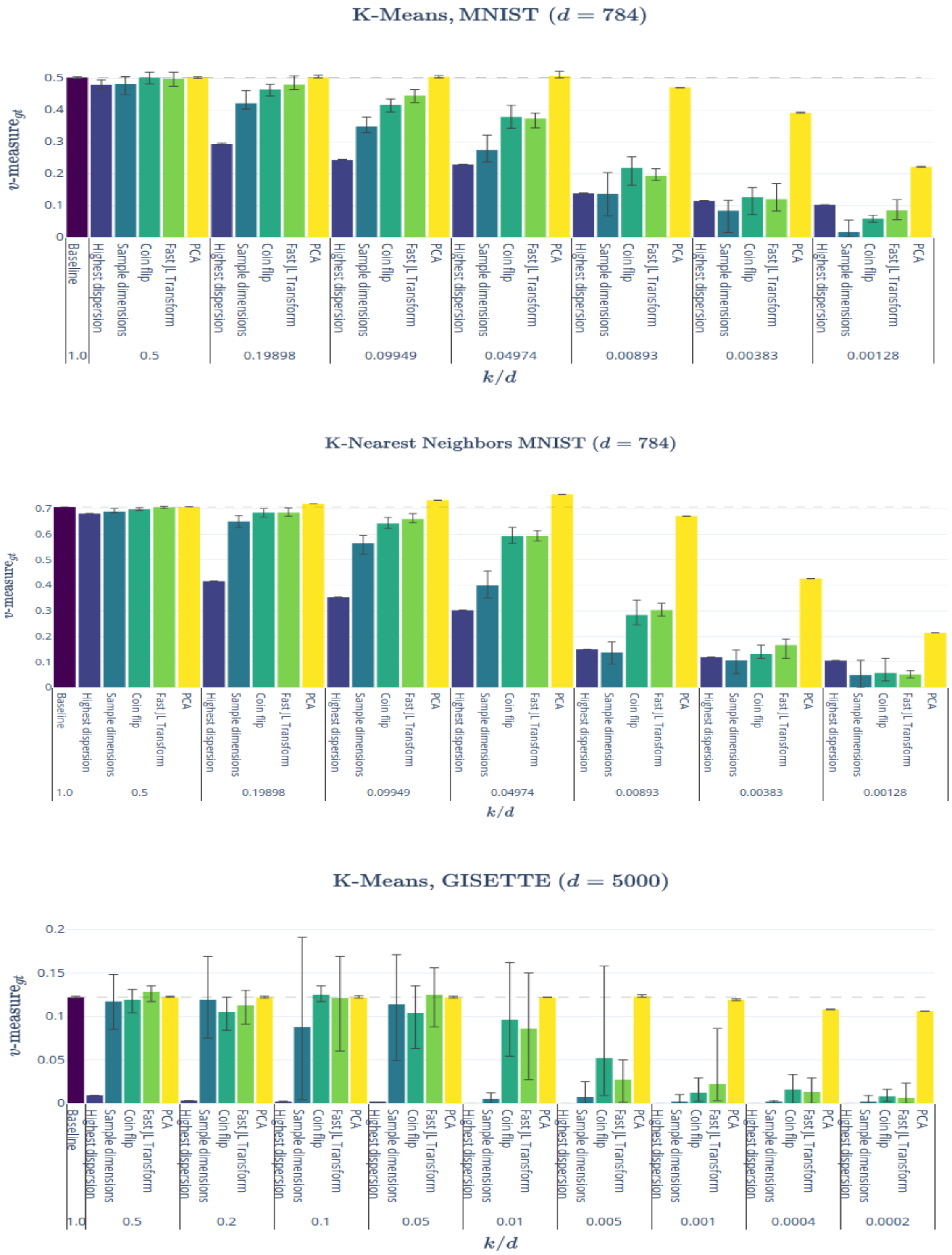


K-Means, Synthetic Data ($d = 1024$)

K-Means, MNIST ($d = 784$)

K-Means, GISETTE ($d = 5000$)

Figure 4: Comparison of the reduction time for different methods



K-Means, MNIST $(d = 784)$

K-Nearest Neighbors MNIST $(d = 784)$

K-Means, GISETTE $(d = 5000)$

Figure 5: Comparison of the goodness for different methods

Considering the time required for the reduction, we see that FJLT can potentially perform better than the simple coin-flip when the reduced space is large enough (between $10 - 50\%$ of d). Besides this nice area, we cannot really see the benefits of this methods. Considering the v-measure, PCA seems to be the best method in almost every case, and also the one that can make the best out of having just a few dimensions (one or two). FJLT and PCA take about the same time when the reduction is important; otherwise, FJLT can be faster of up to little more than an order of magnitude. In some cases, we also observe that a reduction of $\simeq 50\% to 5\%$ using these two methods actually improves the final result. As expected, we do not have reasons to prefer the low-dispersion filter, since computing variance and mean of every variable is too expensive, and the reason behind of this strategy quickly vanishes when the sampling realm, that is d, is very large (e.g GISETTE). The pure sampling of dimensions is clearly the fastest method, and we see one case (KNN on MNIST) where the goodness is comparable with coin-flip and FJLT.

Conclusions

Since our study was focused on the accuracy of the selected methods, we didn't implement FJLT with enough detail to see improvement in computing time. Ultimately, we are not sure it is exploiting the structure of these matrices: in fact, with all of our datasets $d \gg \log n$ is satisfied, but we are not seeing enough time improvement to think that the matrix multiplication by P is using some routine only logarithmic in n, instead of a naive one linear in n. However, the accuracy of FJLT seems quite comparable with the one achieved with the simpler method. We can therefore conclude that if d^2 is an acceptable order of time for a given problem, it is more interesting to use a classic Gaussian JL transform. However, if that's not the case and you are looking for a method around $dlog(d)$, then FJLT provides a rather accurate result that will be computed almost d times faster.

Apart from these result, sampling dimensions might be a good way to go when time is a strict constraint. On the other hand, scikit-learn's implementation of PCA seems to do an overall good job, even when the dimensionality is shrunk to 1 (!); the cost for this is an expected 100x-to-10000x more time compared with sampling. Midway between the two, we could suggest the JL transforms, but only when $n \times d$ is not exceedingly large, e.g 1M, 10M for databases of these sizes.

References

[1] "A beginner's guide to dimensionality reduction in machine learning." https://towardsdatascience.com/dimensionality-reduction-for-machine-learning-80a46c2ebb7e. Accessed: 2019-13-09.

[2] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *Database Theory — ICDT 2001* (J. Van den Bussche and V. Vianu, eds.), (Berlin, Heidelberg), pp. 420–434, Springer Berlin Heidelberg, 2001.

[3] "The curse of dimensionality." https://towardsdatascience.com/the-curse-of-dimensionality-f07c66128fe1. Accessed: 2019-13-09.

[4] W. Johnson and J. Lindenstrauss, "Extensions of lipschitz maps into a hilbert space," *Contemporary Mathematics*, vol. 26, pp. 189–206, 01 1984.

[5] D. Achlioptas, "Database-friendly random projections: Johnson-lindenstrauss with binary coins," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 671 – 687, 2003. Special Issue on PODS 2001.

[6] N. Ailon and B. Chazelle, "The fast johnson–lindenstrauss transform and approximate nearest neighbors," *SIAM Journal on Computing*, vol. 39, no. 1, pp. 302–322, 2009.

[7] john Thickstun, "The fast johnson-lindenstrauss transform." https://homes.cs.washington.edu/~thickstn/docs/fast_jlt.pdf. Accessed: 2019-13-09.

[8] L. Chen, "Johnson-lindenstrauss transformation and random projection." https://www.math.uci.edu/~chenlong/MathPKU/JL.pdf. Accessed: 2019-13-09.

[9] M. Matheny, "A fast implementation of the fast johnson lindenstrauss transform." https://github.com/michaelmathen/FJLT. Accessed: 2019-13-09.

[10] N. Barbey, "Fast hadamard transform." https://github.com/nbarbey/fht/. Accessed: 2019-13-09.

[11] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[12] I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror, "Result analysis of the nips 2003 feature selection challenge," in *Advances in Neural Information Processing Systems 17* (L. K. Saul, Y. Weiss, and L. Bottou, eds.), pp. 545–552, MIT Press, 2005.

[13] P. Fränti, O. Virmajoki, and V. Hautamäki, "Fast agglomerative clustering using a k-nearest neighbor graph," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 28, no. 11, pp. 1875–1881, 2006.

[14] "Clustering basic benchmark." https://cs.uef.fi/sipu/datasets/. Accessed: 2019-13-09.