A Very Fast Method for Clustering Big Text Datasets

Frank Lin and **William W. Cohen** [1]

Abstract.
Large-scale text datasets have long eluded a family of particularly elegant and effective clustering methods that exploits the power of pair-wise similarities between data points due to the prohibitive cost, time- and space-wise, in operating on a similarity matrix, where the state-of-the-art is at best quadratic in time and in space.

We present an extremely fast and simple method also using the power of all pair-wise similarity between data points, and show through experiments that it does as well as previous methods in clustering accuracy, and it does so with in linear time and space, without sampling data points or sparsifying the similarity matrix.

1 Introduction

Clustering methods based on pair-wise similarity of data points, such as spectral clustering methods, are elegant algorithmically and have been shown to be effective on a variety of tasks [1, 14, 15, 17]. However, there are two great obstacles when applying these methods to large-scale text datasets: (1) these methods require finding eigenvectors of the similarity matrix, a very slow operation, and (2) the similarity matrix itself, for large text datasets, is dense and therefore prohibitively expensive in both storage space and algorithm runtime.

Prior work have tried to address these issues along three directions: (a) sample the data points and do computation on a much smaller matrix [6, 20], (b) sparsify the matrix by a k-nearest neighbor technique and do computation on a much sparser matrix [1, 4, 19], and (c) do computation on lots of machines at the same time [1].

Sampling and sparsifying methods gain speed and storage efficiency at the cost of not using all the pair-wise similarity available in the data. Distributed computing methods only speed up the computation linearly (if that) while the computation and storage requirements increase quadratically with the size of the data. These methods have one thing in common—they still use the same core algorithm. At the end of the day, a similarity matrix is computed and stored, and an expensive operation like eigendecomposition is performed.

So we ask: *what is a fast clustering method that uses the full pair-wise similarity of a large text dataset without incurring the cost of constructing, storing, and operating on such a matrix?*

This work is a solution to this problem. This solution is in two parts and yields three advantages. As the first part of the solution we present a clustering method (in Section 2) that finds cluster results similar to that of spectral clustering from a similarity matrix without eigencomputation—this results in the advantage of it being **fast**. For the second part of the solution we show (in Section 3) how the clustering method can easily be modified to incorporate all pair-wise similarities *without* having to construct a similarity matrix—this results in the advantage of it being **space-efficient**. Lastly, this solution has the advantage of being **simple** in that it is easy to describe and understand and also easy to implement and parallelize.

We test this solution (in Section 4) on a well-known text dataset to show its effectiveness in practice and to demonstrate its scalability. In particular, its runtime display an asymptoticly *linear* behavior with respect to input size. After a brief survey of related work (Section 5) we conclude with notes on issues and future directions (Section 6).

2 Power Iteration Clustering

Given a dataset $X = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$, a *similarity function* $s(\mathbf{x}_i, \mathbf{x}_j)$ is a function where $s(\mathbf{x}_i, \mathbf{x}_j) = s(\mathbf{x}_j, \mathbf{x}_i)$ and $s \geq 0$ if $i \neq j$. It is mathematically convenient to define $s = 0$ if $i = j$ [17]. An *similarity matrix* $S \in \mathcal{R}^{n \times n}$ is defined by $S_{ij} = s(\mathbf{x}_i, \mathbf{x}_j)$. The *degree matrix* D associated with A is a diagonal matrix with $d_{ii} = \sum_j Sij$. A *normalized similarity matrix* W is defined as $D^{-1}S$. Below we will view W interchangeably as a matrix, and an undirected graph with nodes X and the edge from x_i to x_j weighted by $s(\mathbf{x}_i, \mathbf{x}_j)$.

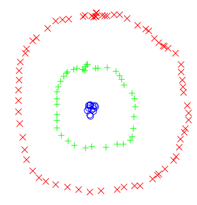
W is closely related to the *random-walk Laplacian* matrix L of Meilă and Shi [13], defined as $L = I - D^{-1}S$. L has a number of useful properties: most importantly to this work, the second-smallest eigenvector of L (one with the second-smallest eigenvalue) defines a partition of the graph W that approximately maximizes the *Normalized Cut* criteria. More generally, the k smallest eigenvectors define a subspace where the clusters are often well-separated. Thus the second-smallest, third-smallest, …, k^{th} smallest eigenvectors of L are often well-suited for clustering the graph W into k components.

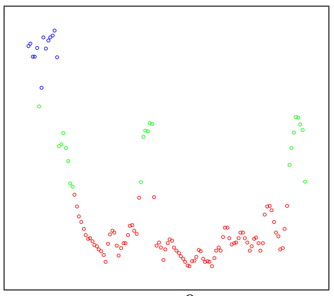
The k *smallest* eigenvectors of L are also the k *largest* eigenvectors of W. One simple method for computing the largest eigenvector of a matrix is *power iteration* (PI), also called the *power method*. PI is an iterative method, which starts with an arbitrary vector $\mathbf{v}^0 \neq \mathbf{0}$ and repeatedly performs the update $\mathbf{v}^{t+1} = cW\mathbf{v}^t$ where c is a normalizing constant to keep $\mathbf{v}^t$ from getting too large (here $c = 1/\|W\mathbf{v}^t\|_1$).

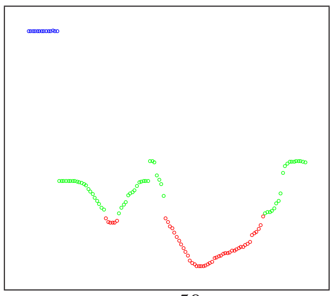
The largest eigenvector of W is not very interesting—in fact, it is a constant vector: since the sum of each row of W is 1, a constant vector transformed by W will never change in direction or magnitude, and is hence a constant eigenvector of W with eigenvalue $\lambda_1 = 1$. However, the intermediate vectors obtained by PI during the convergence process are very interesting. This is best illustrated by example. Figure 1(a) shows a synthetic two-dimensional dataset [2] and Figures 1(b), 1(c) and 1(d) show $\mathbf{v}^t$ at a different t, each illustrated by plotting $\mathbf{v}^t(i)$ for each $\mathbf{x}_i$. For purposes of visualization, the instances $\mathbf{x}$ in the "bullseye" are ordered first, followed by instances in the central ring, then by those in the outer ring. We have also re-scaled the plots to span the same vertical distance. Qualitatively, PI first converges *locally* within a cluster: by 1(d) the points from each

[1] Carnegie Mellon Unversity, USA, email: {frank,wcohen}@cs.cmu.edu

[2] Each $\mathbf{x}_i$ is a point in $\mathcal{R}^2$ space, with $s(\mathbf{x}_i, \mathbf{x}_j)$ defined as $exp((-\|\mathbf{x}_i - \mathbf{x}_j\|^2)/(2\sigma^2))$


(a) 3Circles dataset


(b) t=0


(c) t=50

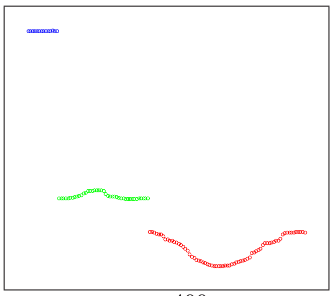

(d) t=400

Figure 1. Clustering result and the embedding provided by $\mathbf{v}^t$ for the 3Circles dataset. (a) shows the dataset and its clusters, each cluster indicated using a different color and point style. In (b), (c) and (d) the value of each component of $\mathbf{v}^t$ is plotted against its index. For visualization, indices are ordered according to cluster and values are scaled so the largest value is always at the top and the minimum value at the bottom.

cluster have approximately the same value in $\mathbf{v}^t$, leading to three disjoint line segments in the visualization.

These observations suggest that an effective clustering algorithm might run PI for some small number of iterations t, stopping *after* it has converged within clusters but *before* final convergence, leading to an approximately piecewise constant vector, where the elements that are in the same cluster have similar values. Specifically, define the *velocity at t* to be the vector $\boldsymbol{\delta}^t = \mathbf{v}^t - \mathbf{v}^{t-1}$ and define the *acceleration at t* to be the vector $\boldsymbol{\epsilon}^t = \boldsymbol{\delta}^t - \boldsymbol{\delta}^{t-1}$. We pick a small threshold $\hat{\epsilon}$ and stop PI when $\|\boldsymbol{\epsilon}^t\|_\infty \leq \hat{\epsilon}$. The stopping criterion is based on the assumption that while the clusters are "locally converging", the rate of convergence changes rapidly; whereas during the final global convergence, the converge rate appears more stable. This assumption turns out to be well-justified. Note that

$$\begin{aligned}
\mathbf{v}^t &= W\mathbf{v}^{t-1} = W^2\mathbf{v}^{t-2} = ... = W^t\mathbf{v}^0 \\
&= c_1 W^t \mathbf{e}_1 + c_2 W^t \mathbf{e}_2 + ... + c_n W^t \mathbf{e}_n \\
&= c_1 \lambda_1^t \mathbf{e}_1 + c_2 \lambda_2^t \mathbf{e}_2 + ... + c_n \lambda_n^t \mathbf{e}_n \\
\frac{\mathbf{v}^t}{c_1 \lambda_1^t} &= \mathbf{e}_1 + \frac{c_2}{c_1}\left(\frac{\lambda_2}{\lambda_1}\right)^t \mathbf{e}_2 + ... + \frac{c_n}{c_1}\left(\frac{\lambda_n}{\lambda_1}\right)^t \mathbf{e}_n
\end{aligned}$$

The convergence rate of PI towards the dominant eigenvector $\mathbf{e}_1$ depends on $(\lambda_i/\lambda_1)^t$ for the significant terms $i = 2, ..., k$, since their eigenvalues are close to 1 if the clusters are well-separated [13], making $(\lambda_i/\lambda_1)^t \simeq 1$. This implies that in the beginning of PI, it converges towards a linear combination of the top k eigenvectors, with terms $k+1, \ldots, n$ diminishing at a rate of $\geq (\lambda_{k+1}/1)^t$. After the noise terms $k+1, \ldots, n$ go away, the convergence rate towards $\mathbf{e}_1$ becomes nearly constant. The complete algorithm, which we call *power iteration clustering* (PIC), is shown in Figure 2.

```
Input: Normalized similarity matrix $W$, number of clusters $k$
Output: Clusters $C_1, C_2, ..., C_k$

1. Pick an initial vector $\mathbf{v}^0$.
2. $\mathbf{v}^{t+1} \leftarrow \frac{W\mathbf{v}^t}{\|W\mathbf{v}^t\|_1}$ and $\boldsymbol{\delta}^{t+1} \leftarrow |\mathbf{v}^{t+1} - \mathbf{v}^t|$.
3. Increment $t$ and repeat above step until $|\boldsymbol{\delta}^t - \boldsymbol{\delta}^{t-1}| \simeq \mathbf{0}$.
4. Use $k$-means on $\mathbf{v}^t$ and return clusters $C_1, C_2, ..., C_k$.
```

Figure 2. The PIC algorithm.

In prior work PIC has shown to be as effective as or even outperform spectral clustering methods such as [17] and [15] on a variety of datasets [10]. Its obvious speed advantage over spectral clustering comes from finding cluster indicators without eigenvectors and early stopping of the already-fast power iteration on sparse matrices. However, this advantage alone does not solve the problem of large text datasets.

3 Bipartite Graph and Similarity Functions

PIC provides us with only one part of the solution: we are able to obtain cluster indicators from a similarity matrix with a fast iterative method. The other issue remains: the input to PIC is still a n-by-n similarity matrix; and constructing, storing, and operating on such a matrix would require computing time and storage *at least* quadratic to the size of the input. Here we present the observation which leads to the complete solution that is the main contribution of this paper.

At the core of PIC is a simple calculation: a matrix-vector multiplication $W\mathbf{v}^t$. If we decompose the matrix W into a series of matrix multiplications, the original PIC matrix-vector multiplication becomes a series of matrix-vector multiplications. This decomposition is not useful if any of these matrices is n-by-n or dense, but if they are all sparse and with size linear to n, then this decomposition is extremely useful. This turns out to be exactly the case.

3.1 Bipartite Graph and "Path Folding"

A bipartite graph is a network with two mutually exclusive groups of nodes where only links between nodes from different groups are allowed and links within the groups are not allowed. A text dataset can be viewed as a bipartite graph or network, where one group the nodes correspond to the documents and the other group the nodes correspond to the words. If a document contains a particular word, a link exists between the document node and the word node. If two documents contain the same word, a path of length two can be traced from one to the other. If two documents are very similar, there would be many such paths between them (since similar documents tend to contains the same words); if two documents are very dissimilar, then there would be very few such paths. The number of paths between two document nodes in such graph then can be viewed as a similarity measure between two documents.

If we are only interested in the similarity between documents, we may "fold" the paths by counting all paths of length two between any two documents and replacing the paths with a direct link between them, weighted by the path count. This "folding" can be expressed concisely with a matrix multiplication:

$$FF^T = S$$

where rows of F represent documents and columns of F represent words; $F(i,j)$ can simply be the binary occurrence of word j in document i, or it could be the word count of j in i, or a weighted word count (e.g., tf-idf term weighting [9]). S is then the "folded" network—each of its nodes is a document and a weighted link between two documents ($S(i,j)$) represent the combined weight of all paths of length two in the original "unfolded" network F.

We now consider the density of these two different representations, the "unfolded" bipartite network F and the "folded" network S, in the context of large text datasets. F will most certainly be a sparse matrix; there are a large number of words in the vocabulary, but only a very small fraction of it will occur in any single document. S is quite the opposite. $S(i,j)$ is zero only if no words are shared between documents i and j; yet the very skewed distribution of word occurrences [12], and in particular that of the most common words, makes $S(i,j)$ highly likely to be non-zero, which subsequently makes S very dense. As the number of documents increases, S, a direct representation of document similarity, becomes very costly in terms of storage and processing time; on the other hand, F is a much more compact, albeit indirect, representation of document similarity. This leads us to our modification of the original PIC algorithm: instead of the similarity matrix we use the decomposition instead, and this decomposition is actually the data in its original form, saving us from having to construct and store a similarity matrix at all, while providing us with the same exact result.

Before using the similarity data in its "unfolded" form in a PIC iteration, we need to do one more thing. Recall that W is a normalized form of S where $W = D^{-1}S$; we need to find the diagonal matrix D^{-1} without S. It follows that the values of the diagonal matrix D^{-1} can also be calculated efficiently via a number of sparse matrix-vector multiplications using the same decomposition: calculate a vector $\mathbf{d} = FF^T\mathbf{1}$, where $\mathbf{1}$ is a vector of 1's, and let $D(i,i) = \mathbf{d}(i)$. Now the $W\mathbf{v}^t$ in PIC becomes:

$$D^{-1}(F(F^T\mathbf{v}^t))$$

Note that in the above equation the math is exactly the same without the bracketing, but the order of operations is vital to making this a series of sparse matrix-vector multiplications.

3.2 Similarity Functions

If instead of a bipartite graph we view the rows of F as feature vectors of documents in vector space, then "Path folding" is equivalent to the *inner product similarity* of documents in a vector space model, often used in information retrieval problems [12]. However, this is just one of many similarity functions often used for measuring document similarity; for example, one may want to normalize the document feature vectors by its length.

It turns out that this scalability extension to PIC can be easily swapped with other similarity functions; here we consider one of the most widely used in information retrieval literature [9, 12], the *cosine similarity*: $cos(\mathbf{a},\mathbf{b}) = \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$ where cos(**a**,**b**) is simply the cosine of the angle between vectors $\mathbf{a}$ and $\mathbf{b}$. For the normalizing term $1/(\|\mathbf{a}\|\|\mathbf{b}\|)$, we need to calculate an additional diagonal matrix $N(i,i) = 1/\sqrt{(F(i)F(i)^T)}$ where $F(i)$ is the ith row-vector of F. Then following inner product similarity, the values of the diagonal matrix D can be calculated by $\mathbf{d} = NFF^TN\mathbf{1}$. Then for each iteration of PIC we have:

$$D^{-1}(N(F(F^T(N\mathbf{v}^t)))) \quad (1)$$

Again, all operations in constructing N and D and in calculation PIC are sparse matrix-vector multiplications. As the sharp reader may notice, instead of doing additional multiplications with N we *can* preprocess F to be cosine-normalized and consequently simplify the above to a inner product similarity. However, practically, with extremely large datasets it is very inefficient to store a version of the dataset for every similarity function one might want to apply; calculating similarity functions on-the-fly as in Equation 1 will often prove to be a much more efficient approach.

4 Experiments

Dataset. To test the proposed method, we choose the RCV1 text categorization collection [9]. RCV1 is a well-known benchmark collection of 804,414 newswire stories labeled using three sets of controlled vocabularies. We use the test split of 781,256 documents and category labels from the *industries* vocabulary. To aid clustering evaluation, documents with multiple labels and categories with less than 500 instances were removed, following previous work [1]. We ended up with 193,844 documents and 103 categories.

We generate 100 random category pairs and pool documents from each pair to create 100 two-cluster datasets: first, we randomly draw a category from the 103 categories—this is category A. Then for candidates of category B, we filter out category A itself and any other category that is more than twice the size or less than half the size of category A. Finally, category B is randomly drawn from the remaining categories. This whole process is repeated 100 times. The filtering is done so we do not have datasets that are overly skewed in cluster size ratio, leads to the misinterpretation of clustering accuracy; for example, a dataset with size ratio 1:9 will achieve 90% accuracy with trivial clustering of all the data points in one cluster).

Since the *industries* vocabulary supports many fine distinctions, we end up with 100 datasets of varying difficulty. For example, whereas [PIG FARMING vs GUIDED WEAPONS] should be a relatively "easy" pair to cluster, [SYSTEMS SOFTWARE vs APPLICATIONS SOFTWARE] may be more "difficult" due to similarity in vocabulary. These category pair datasets vary greatly in size—useful in observing how well a method scale up as input data size increases.

Each document is represented as a log-transformed tf-idf (term-frequency . inverse document-frequency) vector, as is typically done in the information retrieval community for comparing similarity between documents [9, 12].

Methods Compared. We compare PIC against two well-known methods—the standard k-means algorithm and Normalized Cuts [17]. k-means is an iterative algorithm with the objective of finding k cluster centers that minimizes the *within-cluster sum of squares* (WCSS), the sum of the Euclidean distances from the cluster centers to the data points within the cluster. In practice, k-means converges fast and gives reasonable results on linearly separable datasets but is sensitive to initial centers and may be trapped in a local minima. In our experiments we run k-means 10 times with random initial centers and use the one with the smallest WCSS as the final result.

Normalized Cuts (NCUT) is an elegant spectral clustering method and has shown to be effective in a variety of tasks including network community detection and image segmentation [1,17,19]. The method

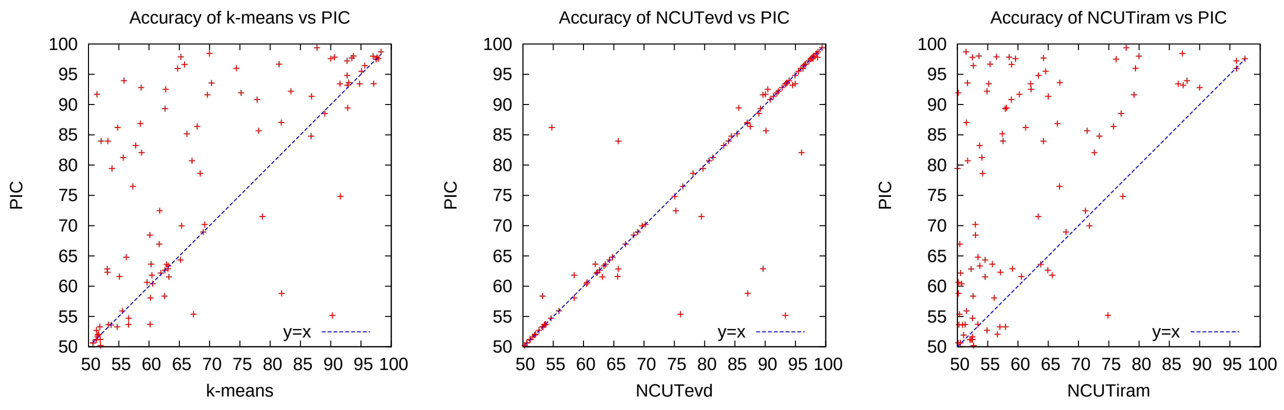


Figure 3. Clustering accuracy correlation plots between PIC and other methods. The diagonal line indicates $y = x$.

first finds the bottom $2 - k$-th eigenvectors of the normalized Laplacian of the similarity matrix $L = I - D^{-1}S$, and the eigenvectors are the embedding of the data points onto a $(k - 1)$-dimensional plane. A k-means algorithm is then used to find the clusters from the embedding. The most computationally expensive part of NCUT is finding the eigenvectors. Finding eigenvectors of a matrix takes $O(n^3)$ time in general, though there are faster methods that provide reasonable approximations. In this experiment we compare results with two versions of the NCUT: **NCUTevd** and **NCUTiram**. NCUTevd uses the slower but more accurate classic eigenvalue decomposition for finding eigenvectors. NCUTiram uses the fast *Implicitly Restarted Arnoldi Method* [8], a more memory-efficient version of the Lanczos algorithm for approximations to the top or bottom eigenvectors of a non-symmetric matrix.

In this experiment we use PIC modified with cosine similarity function as described in Section 3.2 in Equation 1, with $0.00001/n$ as the convergence threshold, where n is the number of documents, and with random initial vectors where components are randomly drawn from [0,1). For both PIC and the NCUT methods, we run k-means 10 times on the embedding and choose the result with the smallest WCSS as the final clustering.

Evaluation Metrics. We evaluate the clustering results according to the *industries* category labels using two metrics: *clustering accuracy* (ACC) and *normalized mutual information* (NMI).

Accuracy in general is defined to be the percentage of correctly labeled instances out of all the labeled instances. Clustering accuracy here is the best accuracy obtainable by a clustering if we are to assign each cluster a unique category label by consider all such possible assignments and then pick one that maximizes the labeling accuracy. To do this with a large number of clusters a dynamic programming approach is needed to avoid searching through all possible label permutations, but here we only need to pick from two possible cluster labeling. NMI is a information-theoretical measure where the mutual information of the true labeling and the clustering are normalized by their entropies. Due to space constraints, we refer readers to [1] for its formal definition.

4.1 Accuracy Results

The experimental results are summarized in Table 1, showing the accuracy and NMI for the methods compared, average over 100 category pair datasets. The "baseline" number for ACC is the average accuracy of a trivial clustering where all the data points are in one cluster and none in the other (i.e., the accuracy of having no clusters). This is provided due to the tendency for clustering accuracy to appear better than it actually is. The differences between numbers in Table 1 are all statistically significant *with the exception* of those between NCUTevd and PIC, where the p-values of one-tailed paired t-tests of ACC and NMI are 0.11 and 0.09 respectively.

Table 1. Summary of clustering results. Higher numbers indicate better clustering. All differences are statistically significant with the exception of those between NCUTevd and PIC, boldface.

	ACC-Avg	NMI-Avg
baseline	57.59	-
k-means	69.43	0.2629
NCUTevd	**77.55**	**0.3962**
NCUTiram	61.63	0.0943
PIC	**76.67**	**0.3818**

The ACC results correlate with those of NMI, and NCUTevd is the most accurate algorithm, though not significantly more so than PIC. Both NCUTevd and PIC do much better than k-means, a typical result in most prior work comparing k-means and methods using pair-wise similarity [1, 15, 19]. We are surprised to find NCUTiram doing much worse than all other methods including k-means; the degree to which it failed the task is even more pronounced in NMI, showing the clustering is close to random. In prior work [1, 8] and in our previous experience with other datasets NCUTiram usually do as well or nearly as well as NCUTevd. Perhaps a more advanced tuning of the parameters of IRAM is required for better approximations to eigenvectors, but we are unable to obtain better results from the IRAM implementation at the time of this writing. Regardless, the conclusions we draw from these experiments is no less significant *even if NCUTiram were to perform just as well as NCUTevd*.

Since the datasets are of varying difficulties, we are interested in how well PIC performs compared to other methods in detail. Is PIC always better than k-means? Does it have difficulty clustering the same datasets as other methods? To answer such questions we plot the accuracy of other methods against that of PIC in Figure 3.

Looking at k-means vs PIC accuracy chart, we see that there are clearly some "easy" datasets, with their corresponding points concentrated near the top right, and some "difficult" datasets concentrated near the bottom left. Aside from these, points lie mostly above the center diagonal line, showing that most of the time, PIC do as well or better than k-means. There is not a strong correlation between k-means and PIC accuracy, possibly due to them being very different clustering methods, one using centroid-to-point similarity and one using all point-to-point similarity.

The NCUTevd vs PIC accuracy plot, with the exception of less than 10 datasets, forms a nearly diagonal line through the middle of the chart, showing that most datasets are "equally difficult" to these clustering methods. This may be an indication that the clusters produced by these methods are very similar, possibly due to them both using all point-to-point pair-wise similarity. Although PIC and NCUT yield very similar accuracy results on these datasets, we will see the next section that their ability to scale up to larger datasets are very different. We will not discuss NCUTiram vs PIC accuracy here since NCUTiram seems to have failed completely on this dataset to produce approximate eigenvectors.

4.2 Scalability Results

We plot data size versus runtimes on a log-log chart in Figure 4. Note that these times are the *embedding time* of the methods. Specifically, for NCUT it is the time it took to find the second bottom eigenvector of L and for PIC it is the time it took for the PI loop to converge. We **do not** include the times for constructing the required matrices (S for NCUT and D, N for PIC) and we did not include the times for k-means to run after the embedding. The reasons are: (a) these times are *always* a very small fraction of the embedding time, (b) k-means can be run as many times as desired to avoid being trapped at a local minima, and (c) their inclusion will only favor PIC, since k-means runs take the same amount of time and the matrix construction is $O(n^2)$ for NCUT and $O(n)$ for PIC. All algorithms were implemented in MATLAB and ran on a single Linux machine with two quad-core 2.26Ghz CPUs and 24GB of RAM.

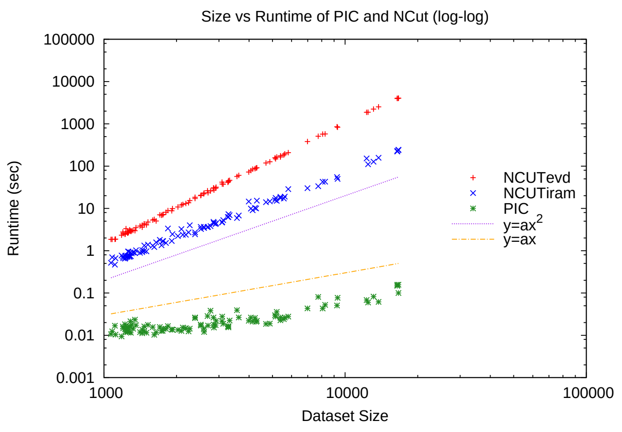


Figure 4. A size versus runtime plot on a log-log scale. The dots show runtime (in seconds) of various methods on datasets of increasing sizes. The lines show the slope of a linear curve and a quadratic curve for comparison purposes and do not correspond directly to any method.

What is immediately outstanding from Figure 4 is that PIC is much faster than either NCUTevd or NCUTiram. On the smallest dataset of 1,070 documents, PIC took only a hundredth of a second, 50 times faster than NCUTiram and 175 times faster than NCUTevd. On the largest dataset of 16,636 documents, PIC took about a tenth of a second, roughly 2,000 times faster than NCUTiram and 30,000 times faster than NCUTevd. In addition, this time is with PIC calculating cosine similarities on-the-fly on each iteration (NCUT is given the pre-calcualted cosine similarity matrix).

What is perhaps less obviously but even more remarkable is the runtime asymptotic behavior. To visualize this in the figure, we include a line with quadratic behavior ($y = ax^2$) and a line with linear behavior ($y = ax$). With these are guidelines, we can see that NCUTiram time is slightly above quadratic and NCUTevd close to cubic. PIC, on the other hand, display a linear behavior. The runtime asymptotic behaviors of NCUTevd and NCUTiram are more or less better understood so these results are no surprise (for a detailed runtime analysis of NCUTiram see [1]). However, one may question the linearity of PIC based solely on a log-log plot.

As shown in Sections 2 and 3, within each PIC iteration the runtime is strictly linear to the size of the input—that is, linear to the number of non-zero elements in the input document vectors. Assuming the vocabulary size is constant, then PIC runtime is:

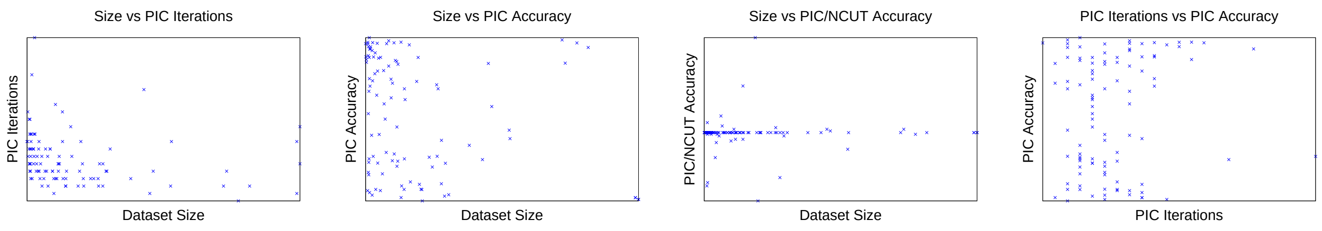
$$O(n) \times (\text{\# of PIC iterations})$$

Generally, it is difficult to analyze the number of steps required for convergence in an iterative algorithm (e.g., k-means), but if we are interested in the asymptotic behavior on certain datasets, we can instead ask a simpler question: *does the number of iterations increase with dataset size?* To observe this experimentally, we plot a correlation chart of the size of the dataset and the number of PIC iterations and calculate the R^2 correlation value, show in Figure 5(a). We find no noticeable correlation between the size of the dataset and the number of PIC iterations. This implies that the number of iterations is independent of dataset size, which means that asymptotically, the number of iterations is constant with respect to dataset size.

The sharp reader may raise further questions regarding this analysis. What if larger datasets are more "difficult" to PIC? It is meaningless to point out an algorithm being linear if it fails to work as dataset size gets bigger. To observe this we calculate R^2 values and plot correlations between dataset size and PIC accuracy in Figure 5(b) and between dataset size and the ratio of PIC accuracy to NCUTevd accuracy in Figure 5(c). Again, with no discernible correlation in these figures, we conclude that PIC accuracy is independent of dataset size (Figure 5(b)) and that PIC is as accurate as NCUT as dataset size increases (Figure 5(c)).

An additional correlation statistic that may be of interest between that of PIC's accuracy and number of iterations. It is not unusual for an iterative algorithm to converge much faster on a "easy" dataset and slower on a more "difficult" dataset. Since the number of iterations is directly related to runtime, we may expect PIC to be slower on more "difficult" datasets. Surprisingly, Figure 5(d) does not show correlation between the two, indicating that PIC work just as fast on "difficult" datasets as on "easy" datasets. This leads us to our conclusion concerning the runtime scalability of PIC—that as far as text datasets are concerned, its runtime is linear with respect to input size.

Perhaps PIC's runtime scalability is only matched by its small memory footprint. In addition to the input dataset, the "bipartite graph" PIC embedding requires *exactly* $4n$ storage ($\mathbf{v}^t$, $\mathbf{v}^{t-1}$ and diagonal matrix $\boldsymbol{\delta}^{t-1}$, D) for inner product similarity and $5n$ (an additional diagonal matrix N) for cosine similarity, *regardless of vocabulary size*. This is much more feasible compared to at least n^2 storage required by methods requiring explicit construction of a similarity matrix.



(a) $R^2 = 0.0424$ (b) $R^2 = 0.0552$ (c) $R^2 = 0.0007$ (d) $R^2 = 0.0134$

Figure 5. Correlation plots and R^2 correlation values. None of these plots or values indicate even a weak correlation, thus providing further evidence of PIC's runtime linearity. Note on average it takes 15 iterations for PIC to converge, with 31 iterations being the maximum.

5 Related Work

The basic PIC algorithm (in Section 2) is related to [18] and [21] in that repeated matrix multiplications reveals cluster structure in a similarity matrix; however these methods do matrix-matrix multiplication instead of matrix-vector multiplication—a major disadvantage when it comes to scalability. PIC is perhaps most related to spectral clustering [5, 15–17, 19]; both find a low-dimensional embedding related to the eigenvectors of the similarity matrix and use k-means to produce the final clusters. The difference most relevant to this work is that PIC creates the embedding without explicitly finding *any* eigenvector [10]. This makes PIC much faster than spectral clustering methods. Methods that attempt to make spectral clustering faster have mostly relied on data pint sampling or matrix sparsifying as described in Section 1.

PIC's repeated multiplication of a normalized matrix with a vector can be viewed as a sort of iterative averaging or a backward random walk. This idea has been used in semi-supervised learning method for propagating class labels on network data [3, 11, 22].

This paper extends the prior PIC algorithm to work efficiently with text datasets, where "adjacency" is defined by tf-idf distance. In doing so, we exploit a well-known equivalence between cosine distance and the inner-product computation, and more generally, between multistep random walks and iterated matrix multiplication [2].

6 Conclusion and Future Work

We have shown that, on large text datasets, our proposed method exploits the power of pair-wise similarity to provide clustering accuracy equal to that of Normalized Cuts without constructing or operating on a similarity matrix, yielding remarkable computational efficiency in space and time. It not only runs much faster; according to many observable statistics, it shows a runtime linear to input size, making clustering based on pair-wise similarity feasible and practical for large-scale text data.

Additionally, crucial to its practical use is its simplicity—all core operations are simple matrix-vector multiplications. Not only is it trivial to implement, it is easy parallelize in a large-scale distributed computing environment [7].

In order to ascertain the accuracy and asymptotic behavior of a new method on text data, we have restricted the experiments in this paper to be on two-cluster datasets. We plan to expand experiments to include multi-cluster problem, and we also plan to extend the method to allow for more complicated structures such as hierarchical clusters and mixed-membership clusters.

ACKNOWLEDGEMENTS

This work was funded by NSF under grant IIS-0811562, by NIH under grant R01 GM081293, and by a gift from Google.

REFERENCES

[1] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y. Chang, 'Parallel spectral clustering in distributed systems', *PAMI*, (2010).
[2] Edith Cohen and David D. Lewis, 'Approximating matrix multiplication for pattern recognition tasks', in *SODA*, (1997).
[3] Nick Crawell and Martin Szummer, 'Random walks on the click graph', in *SIGIR*, (2007).
[4] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis, 'Weighted graph cuts without eigenvectors: A multilevel approach', *PAMI*, **29**(11), 1944–1957, (2007).
[5] Miroslav Fiedler, 'Algebraic connectivity of graphs', *Czechoslovak Mathematical Jour.*, **23**(98), 298–305, (1973).
[6] Charless Fowlkes, Serge Belongie, Fan Chung, and Jitendra Malik, 'Spectral grouping using the Nyström Method', in *PAMI*, (2004).
[7] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos, 'Pegasus: A peta-scale graph mining system - implementation and observations', in *ICDM*, (2009).
[8] R.B. Lehoucq and D. C. Sorensen, 'Deflation techniques for an implicitly re-started arnoldi iteration', *SIAM Journal on Matrix Analysis and Applications*, **17**, 789–821, (1996).
[9] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li, 'RCV1: A new benchmark collection for text categorization research', *JMLR*, **5**, 361–397, (2004).
[10] Frank Lin and William W. Cohen, 'Power iteration clustering', in *ICML(to appear)*, (2010).
[11] Sofus A. Macskassy and Foster Provost, 'Classification in networked data: A toolkit and a univariate case study', *JMLR*, **8**, 935–983, (2007).
[12] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze, *Introduction to Information Retrieval*, Cambridge University Press, 2008.
[13] Marina Meilă and Jianbo Shi, 'A random walks view of spectral segmentation', in *AISTAT*, (2001).
[14] M. E. J. Newman, 'Finding community structure in networks using the eigenvectors of matrices', *Physical Review E*, **74**(3).
[15] Andrew Y. Ng, Michael Jordan, and Yair Weiss, 'On spectral clustering: Analysis and an algorithm', in *NIPS*, (2002).
[16] Tom Roxborough and Arunabha Sen, 'Graph clustering using multiway ratio cut', in *Graph Drawing*, (1997).
[17] Jianbo Shi and Jitendra Malik, 'Normalized cuts and image segmentation', *PAMI*, **22**(8), 888–905, (2000).
[18] Naftali Tishby and Noam Slonim, 'Data clustering by markovian relaxation and the information bottleneck method', in *NIPS*, (2000).
[19] Ulrike von Luxburg, 'A tutorial on spectral clustering', *Statistics and Computing*, **17**(4), 395–416, (2007).
[20] Donghui Yan, Ling Huang, and Michael I. Jordan, 'Fast approximate spectral clustering', in *KDD*, (2009).
[21] Hanson Zhou and David Woodruff, 'Clustering via matrix powering', in *PODS*, (2004).
[22] Xiaojin Zhu, Zoubin Ghahramani, and John Lafferty, 'Semi-supervised learning using Gaussian fields and harmonic functions', in *ICML*, (2003).