A Scalable Approach to Multi-dimensional Data Analysis

Yong Shi
Department of Computer Science and Information Systems
Kennesaw State University
1000 Chastain Road
Kennesaw, GA 30144Author(s) Name(s)
yshi5@kennesaw.edu

Abstract

Similarity search is one of the most studied research fields in data mining. Given a query data point Q, how to find its closest neighbors efficiently and effectively has always been a challenging research topic. In this paper, we discuss continuous research on data analysis based on our previous work on similarity search problems, and present an approach to improving the scalability of the PanKNN algorithm [13]. This proposed approach can assist to improve the performance of existing data analysis technologies, such as data mining approaches in Bioinformatics.

Keywords: *Nearest neighboring search, scalability, segment mergence.*

1. Introduction

Data mining is an important tool to extract information from various data types. It is currently used in a wide range of applications. For example, given an informational advantage, data mining can help find valuable patterns from data and transform them into business intelligence. As one of the most studied research fields in data mining, the similarity search problem has been studied in the last decade, and many algorithms have been proposed to solve the K nearest neighbor search [10, 12, 2, 9, 8]. PanKNN [13] is a novel technique which explores the meanings of K nearest neighbors from a new perspective, redefines the distances between data points and a given query point Q, and efficiently and effectively selects data points which are closest to Q. In this paper, we propose an approach to improving the scalability of PanKNN, and demonstrate the experimental results.

2. Related work

Nearest neighbor search is an optimization problem for finding nearest data points in a given data space. Different kinds of solutions to the nearest neighbor search problem have been proposed. The similarity between two data points used to be based on a similarity function such as Euclidean distance which aggregates the difference between each dimension of these two data points. Traditional approaches [1, 6, 14] solve the nearest neighbor problems based on the distance between the query point and the data point over a full data space, thus they suffer from the "cure of dimensionality". In a high dimensional space the data are usually sparse, and widely used distance metric such as Euclidean distance may not work well as dimensionality goes higher. Recent research [7] shows that in high dimensions nearest neighbor queries become unstable: the difference of the distances of farthest and nearest points to some query point does not increase as fast as the minimum of the two, thus the distance between two data points in high dimensionality is less meaningful. Although there are approaches [11, 4, 3] targeting partial similarities, they require fixed subset of dimensions or fixed number of dimensions as the input parameter for the algorithms.

3. Solving similarity problems

We first briefly introduce our previous work on PanKNN [13], in which we analyze the nearest neighbor problems from a new perspective. We define the new meanings for the K nearest neighbors problem, and design algorithms accordingly. The similarity between a data point and a query point is not based on the difference aggregation on all the dimensions. Instead, we propose self-adaptive strategies to dynamically select dimensions based on the different situation of the comparison.

For a given data point X_i, and a given query point Q, we call the distance between X_i and Q as Pan-distance $PD(X_i,Q)$. $PD(X_i,Q)$ does not calculate the aggregated differences between X_i and Q on all dimensions. Instead, it only takes into account those dimensions on which X_i is close enough to Q, and sums them up. This strategy not only avoids the negative impacts from those dimensions on which X_i is far to Q, but also eliminate the curse of dimensionality caused by similarity functions such as Euclidean distance which calculates the square root of the sum of squares of distances on each dimensions.

For two data points X_i and X_j, we judge which data point is closer to Q based on how many dimensions on which they are close enough (within dimension-wise K nearest neighbors) to Q, as well as their average distances to Q on such dimensions.

The PanKNN is designed as follows. Given a data set DS, we first calculate the difference δ_{il} of each data point X_i to the query point Q on each dimension D_l. Then we sort the ids on each dimension D_l based on δ_{il}, and select the first K ids on each dimension D_l and put them into KS_l. We define set GS to contain the ids in KS_l on all dimensions, and calculate the $PD(X_i, Q)$ for each data point if its id is in GS. Finally, we sort the ids based on the Pan-distance and select the first K ids in the sorted list as the ids of K nearest neighbors of Q. We do not need to calculate the difference using different number of dimensions. The number of dimensions and the subset of dimensions associated with data point X_i are both dynamically decided depending on the values of X_i and their rankings on different dimensions.

4. A Scalable Approach to Finding the Nearest Neighbors

PanKNN can efficiently and effectively find nearest neighbors when the size of the data set is small. However, the performance time increases dramatically with the increment of the data size. One of the reasons is that in PanKNN, given a query point, on each dimension, we need to sort the whole data set according their distances to the query point. It is well known that the average time complexity of most sorting algorithms is O(nlogn). In this section we propose an algorithm which solves the scalability problem of PanKNN.

4.1. Segment Mergence

Let n denote the total number of data points and d be the dimensionality of the data space. Let D_l be the lth dimension, where l = 1, 2, ..., d. Let the input d-dimensional data set be X

$$\mathbf{X} = \{X_1, X_2, ..., X_n\}$$

which is normalized to be within the hypercube $[0, 1]^d \subset R^d$. Each data point X_i is d-dimensional vector:

$$X_i = [x_{i1}, x_{i2},..., x_{id}] \tag{1}$$

Data point X_i has the *id* number *i*. Let Q be the query point: $Q = [q_1, q_2, ..., q_d]$.

In PanKNN, we used $\Delta_i = [\delta_{i1}, \delta_{i2}, ..., \delta_{id}]$ as the array of differences between the data point X_i and the query point Q on all dimensions. Given a data set DS of n data points $X = \{X_1, X_2, ..., X_n\}$ with d dimensions $D_1, D_2, ..., D_d$, and a query point Q in the same data space, we first sort the data points on each dimension D_l, l = 1, 2, ..., d, based on δ_{il} which is the difference between data point X_i and Q on dimension D_l. With the increment of the data size, this process can be very time consuming.

The purpose of sorting all data points in the data set on each dimension based on their distances to Q is to find a group of data points on each dimension which are closest to Q. Here we design an approach to finding such groups without having to sort the whole data set on any dimension.

Given K as the number of data points we need to find that are closest to Q, on each dimension D_l, l = 1, 2, ..., d, suppose V_{lmax} and V_{lmin} are the largest value and small value of data points in DS on D_l, respectively. Since we first normalize the data set, V_{lmin} and V_{lmax} will be 0 and 1, respectively. We divide $[V_{lmin}, V_{lmax}]$ evenly by $\lfloor n/k \rfloor$, resulting in segments $S_{l1}, S_{l2}, \ldots, S_{l\lfloor n/k \rfloor}$. Each segment will contain $\lceil K \rceil$ data points in average. However, some segments contain more than $\lceil K \rceil$ data points, and some contain less, due to the uneven distribution of data points on each dimension. It is easy to scan through the whole data set to calculate how many data points each segment S_{lj} contains (denoted as $|S_{lj}|$) and record the group of data point ids in S_{lj}, for j=1, 2, ..., $\lfloor n/k \rfloor$.

We next locate the segment containing q_l which is the value of Q on D_l. Suppose segment S_{lj} contains the value q_l, j=1, 2, ..., $\lfloor n/k \rfloor$. We design a process called *segment mergence* which is described in figure 1.

```
Process: segment mergence
Begin
1) In $S_{l_j}$, calculate the number of data points which are less than $q_l$, and denote it as $N_l$;
2) In $S_{l_j}$, calculate the number of data points which are larger than $q_l$, and denote it as $N_r$;
3) While $N_l <$ K and there is a segment $S_{left}$ on the left which is not merged
        Begin
            $S_{l_j} = S_{l_j} \bigcup S_{left}$;
            $N_l = N_l + |S_{left}|$;
        End;
4) While $N_r <$ K and there is a segment $S_{right}$ on the right which is not merged
        Begin
            $S_{l_j} = S_{l_j} \bigcup S_{right}$;
            $N_r = N_r + |S_{right}|$;
        End;
5) return $S_{l_j}$
End.
```

Figure .1. Proc: Segment Mergence

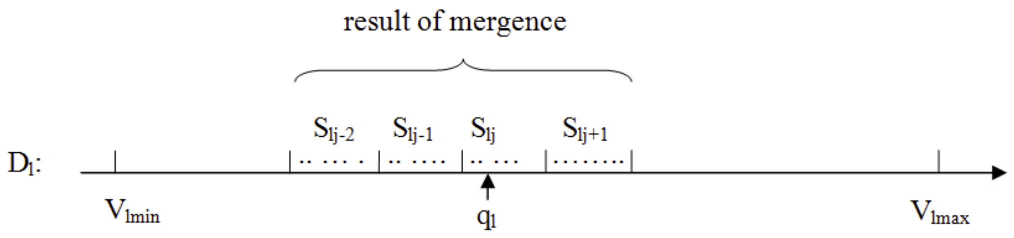

Figure .2. Example 1 of segment mergence on dimension D_l

4.2. Example 1

Figure ure 2 shows an example of the segment mergence. Suppose K is set as 10, we divide the value range on D_l so each segment contains 10 data points in average. S_{lj} contains q_l (the value of Q on D_l) and 5 data points, 2 of which are on the left side of q_l. So the initial value for N_l is 2, and the initial value for N_r is 3. We merge S_{lj} with the segment on its left side (S_{lj-1}), which contains 6 data points. N_l is updated as 8, which is still less than K which is 10. So we continue merging the segment on the left side (S_{lj-2}) which contains 6 data points, and N_l is now updated as 14 which is larger than K. Next we merge the updated segment with the segment on its right side (S_{lj+1}), which contains 8 data points, and N_r is updated as 11 which is larger than K, thus we terminate the mergence process.

From the example we can see, after the segment mergence, the result segment will contain at least K data points on the left side of q_l, and at least K data points on the right side of q_l. The reason is that, in the extreme case, the K nearest neighbors of Q on D_l might all on the same side of Q. By having at least K data points on both sides, we assure that the K nearest data points are within the resulting segment S_{lj} from the process of segment mergence.

By applying the process of segment mergence we do not need to sort the whole data set on each dimension in order to obtain the K nearest neighbors on each dimension.

4.3. Example 2

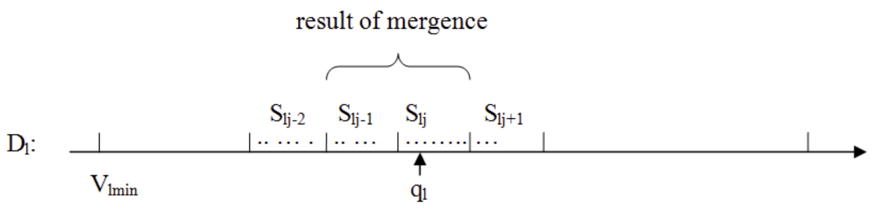

Figure .3. Example 2 of segment mergence on dimension D_l

Figure ure 3 shows another example of the segment mergence. Suppose K is set as 5, we divide the value range on D_l so each segment contains 5 data points in average. S_{lj} contains q_l (the value of Q on D_l) and 8 data points, 2 of which are on the left side of q_l. So the initial value for N_l is 2, and the initial value for N_r is 6. We merge S_{lj} with the segment on its left side (S_{lj-1}), which contains 5 data points. N_l is updated as 7, which is larger than K which is 5. So we stop merging the segment on the left side. Since N_r is 6 which is already larger than K which is 5, we do not merge the updated segment with the segment on its right side at all, and N_r remains as 6. We terminate the mergence process.

In this example, we merge the segment on the left of S_{lj} with S_{lj}, and we do not merge any segments on the right side of S_{lj} since the original value of N_r is already larger than K which is 5. After the segment mergence, the resulting segment will contain at least K data points on the left side of q_l, and at least K data points on the right side of q_l.

4.4. Example 3

Figure ure 4 shows another example of the segment mergence. Suppose K is set as 4, we divide the value range on D_l so each segment contains 4 data points in average. S_{lj} contains q_l (the value of Q on D_l) and 9 data points, 4 of which are on the left side of q_l. So the initial value for N_l is 4, and the initial value for N_r is 5. Since the original value of N_l is 4, which is not less than K which is 4, we do not merge S_{lj} with any segments on the left side. Since the original value of N_r is 5, which is not less than K which is 4, we do not merge S_{lj} with any segments on the right side either.

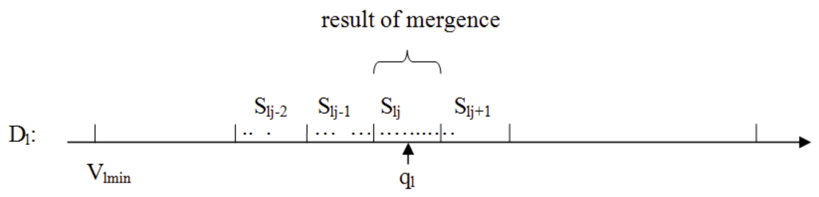

Figure .4. Example 3 of segment mergence on dimension D_l

In the example, we do not merge S_{lj} with any other segments on D_l, because neither N_l (4) nor N_r (5) is less than K which is 4. S_{lj} contains at least K data points on the left side of q_l, and at least K data points on the right side of q_l.

4.5. Finding nearest neighbors with scalability

Given a data set DS of n data points $X = \{X_1, X_2, ..., X_n\}$ with D_l as the dimension l, l=1, 2, ..., d, a query point Q in the same data space, we try to find a set *PKscalable* which consists of k data points from DS so that for any data point $X_i \in$ *PKscalable* and any data point $X_j \in$ DS − *PKscalable*, the PD(X_i, Q) is less than or equal to PD(X_j, Q). The set *PKscalable* is the Pan-K Nearest Neighbor set of Q in DS.

The PanKNN-scalable algorithm is described as follows:

1) For each $X_i \in$ DS, we first calculate $\Delta_i = [\delta_{i1}, \delta_{i2}, ..., \delta_{id}]$ in which $\delta_{il} = |x_{il} - q_l|$;

2) On each dimension D_l, l=1, 2, ..., d, suppose its value range is [V_{lmin}, V_{lmax}], we divide it evenly by K , resulting in segments $S_{l1}, S_{l2}, \ldots, S_{l\lfloor n/k \rfloor}$.

3) Starting from S_{lj} which contains the value q_l, we perform the segment mergence process. The resulting S_{lj} will contain at least 2K data points. We sort the ids of the data points in S_{lj} instead of in DS, based on δ_{il} for X_i. Let G_l be the sorted list on D_l;

4) Let KS_l be the subset of G_l which contains the first K ids in G_l. For each data point X_i, i=1, 2, ... n, we generate $B_i = [b_{i1}, b_{i2}, ..., b_{id}]$ in which $b_{il} = 1$, if $i \in KS_l$; $b_{il} = 0$, if $i \notin KS_l$;

5) Let set GS = {i} in which $i \in KS_l$, l=1, 2, ..., d. For each data point X_i, where $i \in$ GS, we calculate PD(X_i, Q). Next we Sort GS = {i} based on PD(X_i, Q); Let set *PKscalable* contain the first K ids $\in$ GS. We return *PKscalable*.

4.6. Time and space analysis

Suppose the size of the data set is n. Throughout the process, we need to keep track of the information of all points, which collectively occupies O(n) space. For one query point Q, we need to sort the data points in S_{lj} after the segment mergence process on each dimension. The time required is O(dKlogK).

5. Experiments

To assess the accuracy and efficiency of the proposed approach, comprehensive experiments on both synthetic and real data sets were conducted. Our experiments were run on Intel(R) Pentium(R) 4 with CPU of 3.39GHz and Ram of 0.99 GB.

5.1. Experiments on high-dimensional data sets

To test the scalability of our algorithm over dimensionality, data size and K as the number of nearest neighbors required for the query points, we designed a synthetic data generator to produce data sets with normalized distributions. The sizes of the data sets vary from 20,000, 25,000, ... to 40,000, with the gap of 5,000 between each two adjacent data set sizes, and the dimensions of the data sets vary from 20, 25 ... to 40, with the gap of 5 between each two adjacent numbers of dimensions.

Figure ure 5 shows the running time of groups of data sets with dimensions increasing from 20 to 40. Each group has a fixed data size (from 20,000, 15,000, ... to 40,000). And we set K as 10.

Figure ure 6 shows the running time of groups of data sets on one query with sizes increasing from 20,000 to 40,000. Each group has fixed number of dimensions (from 20, 25, ... to 40). And we set K as 10. The two figures indicate that our algorithm is scalable over dimensionality and data size.

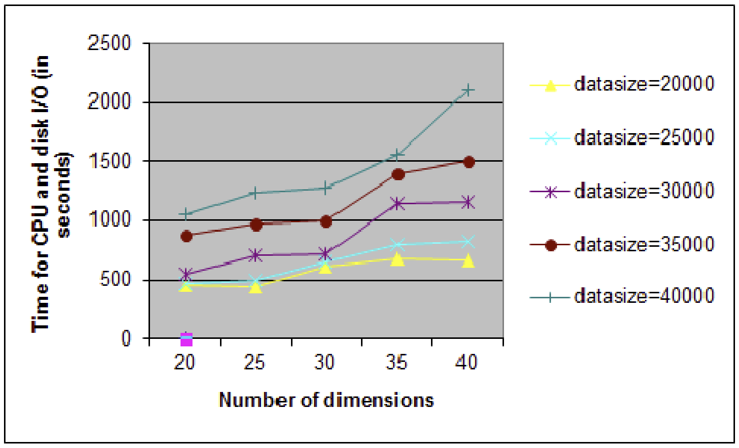

Figure .5. Running time on one query point with increasing dimensions (K = 10)

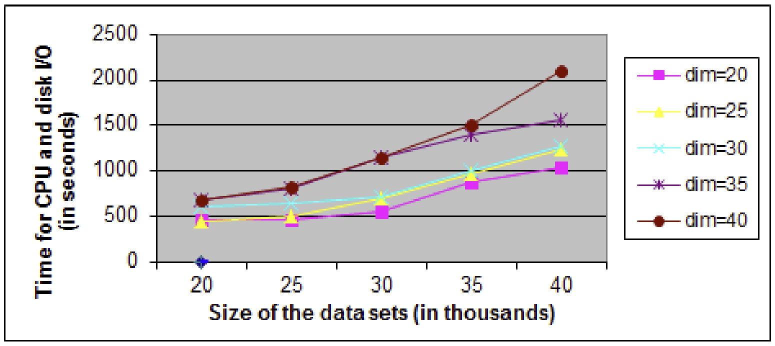

Figure .6. Running time on one query point with increasing data set sizes (K = 10)

Figure ure 7 shows the running time of 3 groups of data sets with the size of 20000, 30000 and 40000 on one query with K increasing from 5,10,... to 30. And we set dimension as 20.

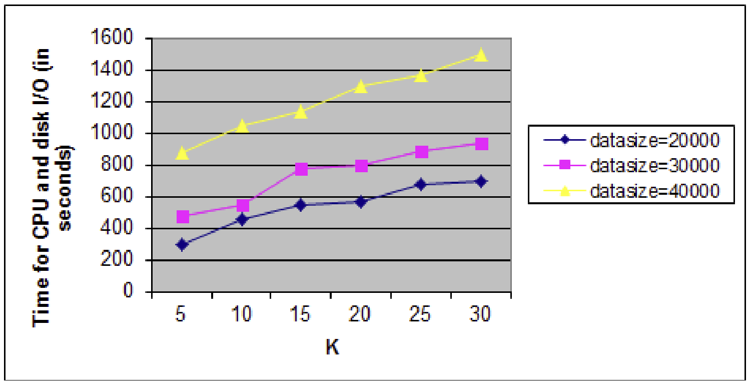

Figure .7. Running time on one query point with increasing K values (dimensionality = 20)

5.1. Experiments of PanKNN-scalable vs. PanKNN

In this section we will demonstrate how PanKNN-scalable improves the performance compared to the original PanKNN.

We first use the synthetic data sets to demonstrate the advantage of PanKNN-scalable. Figure ure 8 shows the running time of PanKNN-scalable vs. PanKNN with sizes increasing from 20,000 to 40,000. We set the dimensionality as 20 and K as 10. From this picture we can see that PanKNN-scalable performs better than PanKNN when the data size increases.

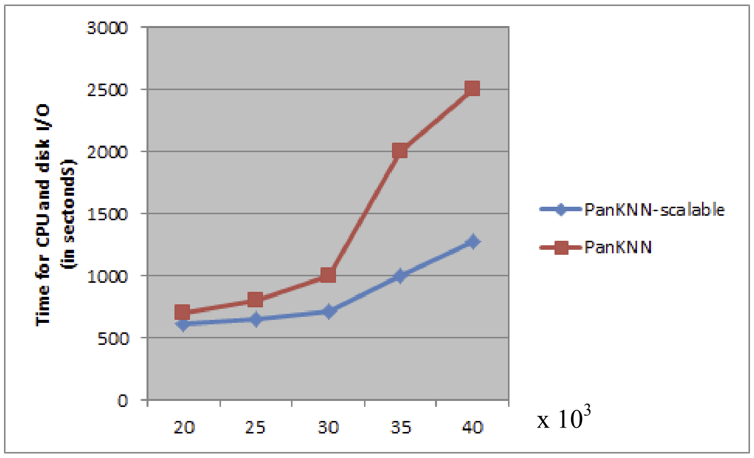

Figure .8. Running time on one query point of PanKNN-scalable vs. PanKNN on increasing data sizes (dimensionality = 20 and K = 10)

We also use real data sets from UCI Machine Learning Repository [5] to demonstrate the performance difference of PanKNN-scalable vs. PanKNN. Here we demonstrate the testing result on one real data set called Wine Recognition data set. It contains the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. There are 178 instances, each of which has 13 features (dimensions), including alcohol, magnesium, color intensity, etc. The data set has three clusters with the sizes of 59, 71 and 48. We perform the algorithms on the Wine data set. The accuracy rate of PanKNN-scalable is 94.1%, which is higher than the accuracy rate of PanKNN (92.9%).

6. Conclusion

In this paper we present our strategy to improve the similarity search approaches. On each dimension we divide the value range into segments with equal size and merge the segment containing the query point with the neighboring segments to acquire the group of data points closest to data point Q on each dimension. This data processing algorithm can be applied in many fields such as bioinformatics, pattern recognition, data clustering and signal processing.

References

1. White D.A. and Jain R. Similarity Indexing with the SS-tree. In Proceedings of the 12th Intl. Conf. on Data Engineering, pages 516–523, New Orleans, Louisiana, February 1996.

2. E. Achtert, C. Bohm, P. Kroger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In SIGMOD '06, pages 515–526, New York, NY, USA, 2006. ACM.

3. C. C. Aggarwal. Towards meaningful high-dimensional nearest neighbor search by human-computer interaction. In ICDE, 2002.

4. C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional space. Lecture Notes in Computer Science, 1973, 2001.

5. S. D. Bay. The UCI KDD Archive [http://kdd.ics.uci.edu]. University of California, Irvine, Department of Information and Computer Science.

6. D. A. Berchtold S., Keim and H.-P. Kriegel. The X-tree : An index structure for high-dimensional data. In VLDB'96, pages 28–39, Bombay, India, 1996.

7. K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In International Conference on Database Theory 99, pages 217–235, Jerusalem, Israel, 1999.

8. B. Cui, H. Shen, J. Shen, and K. Tan. Exploring bit-difference for approximate KNN search in high-dimensional databases. In Australasian Database Conference, 2005., 2005.

9. R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation, 2003.

10. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In The VLDB Journal, pages 518–529, 1999.

11. A. Hinneburg, C. C. Aggarwal, and D. A. Keim. What is the nearest neighbor in high dimensional spaces? In The VLDB Journal, pages 506–515, 2000.

12. T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. SIGMOD Rec., 27(2):154–165, 1998.

13. Y. Shi and L. Zhang. A dimension-wise approach to similarity search problems. In the 4th International Conference on Data Mining (DMIN'08), 2008.

14. R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In Proc. 24th Int. Conf. Very Large Data Bases, VLDB, pages 194–205, 24–27 1998.

Author

Yong Shi received the BS and MS degrees, both in computer science, from the University of Science and Technology of China in 1996 and 1999, respectively. He received Ph.D. in computer science from the State University of New York at Buffalo in 2006. He is currently an assistant professor in the Department of Computer Science and Information Systems in Kennesaw State University. His research interests include data mining, database, machine learning, and information retrieval.