Probability, Spring 2020, Final Report

B07902001 徐維謙, B07902064 蔡銘軒, B07902075 林楷恩

Task 1

The best result of this task is generated by the common greedy algorithm. Although the greedy algorithm delivers great performance, it is discussed by many other groups and we have covered most details about it in our presentation. Thus, in this section, we will focus on the other algorithm we have tried.

1.1 Core Idea

The evaluation metric in this task is the *Hellinger distance*, which is given by the formula

$$H(P,Q) = \sqrt{1 - \left(\sum_{i=1}^{n} \sqrt{p_i q_i}\right)}$$

where P and Q are two discrete probability distributions. It is easy to see $H(P,Q)$ is minimized when $p_i = q_i$ for $i = 1, 2 \ldots, n$. This leads us to our goal — we wish to maintain a subset of data such that for every attribute, p_i is as close to q_i as possible for $i = 1, 2, \ldots, n$.

1.2 Notations

Throughout the following paragraphs, we use the notations defined as follows

- S is the set of all instances, and S_i is the *i-th* instance.
- S' is the set of instances we sample from S.
- N is the number of instances we are required to sample from S.
- S_{id} is the ideal set such that $|S_{id}| = N$ and the probability distributions of all attributes are the same as that in S.
- A is the set of all attributes in S, and A_i is the *i-th* attribute.
- $S_i[A_j]$ is the value of the *j-th* attribute of S_i. For example, let A_j be the attribute "age", then $S_i[A_j] = 5$ indicates the instance S_i is of age 5.
- $S'[A_i][j]$ is the number of instances in S' such that the *i-th* attribute is of value j. This notation extends to S_{id}.

Note that S_{id} is not necessarily a subset of S. There may not exist a subset of S that possesses the property of S_{id}.
With the notations defined, we can rephrase our goal as finding S' such that S' is as close to S_{id} as possible.

1.3 Algorithm And Analysis

1.3.1 Algorithm

Following the idea described above, we came up with an algorithm that worked as follows

1. Calculate the probability distribution for all attributes in S and construct S_{id} accordingly
2. Initialize $S' = \emptyset$.
3. Choose an attribute A_i

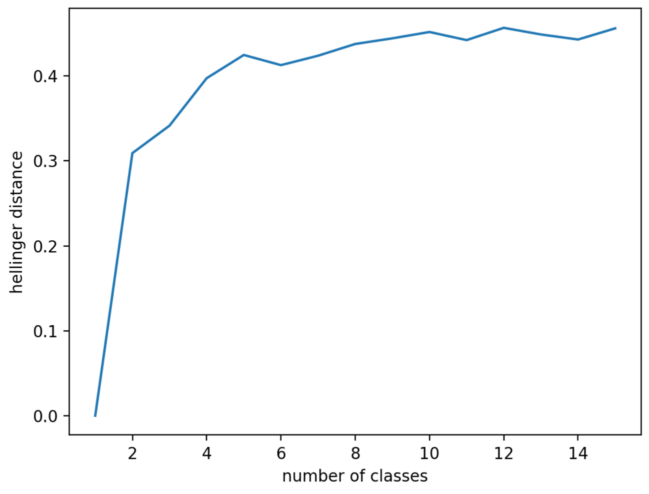
4. Shuffle S and iterate over all instances in S. If $S'[A_i][S_i[A_i]] < S_{id}[A_i][S_i[A_i]]$, add S_i to S'.
5. Iterate over S', score each instance S'_i such that $S'[A_i][S'_i[A_i]] > S_{id}[A_i][S'_i[A_i]]$ according to the the following rule:
    - Iterate over all attribute A_j, increment the score by 1 if $S'[A_j][S'_i[A_j]] > S_{id}[A_j][S'_i[A_j]]$
6. Remove instances with the highest score until $|S'| = N$
7. Go back to step 3

In essence, when we choose an attribute A_i, we make sure the probability distribution of A_i in S' is the same as that in S_{ind}. We can run the algorithm sequentially against all attributes, and then repeat for multiple rounds.

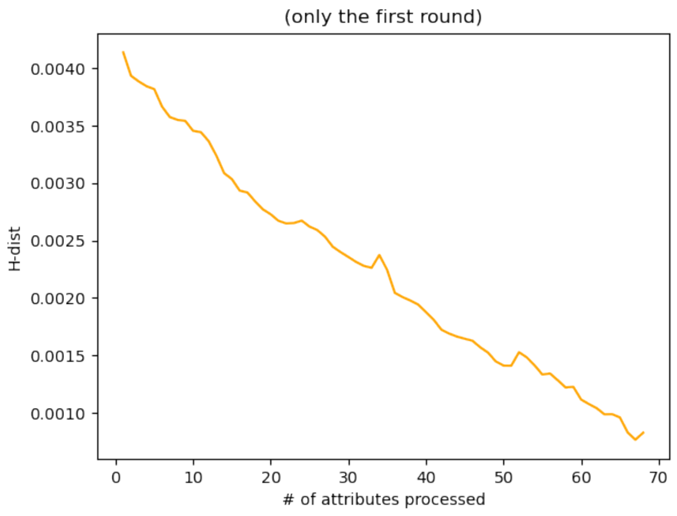
1.3.2 Analysis

For the following discussions, the algorithm is run against the development data set. That is, $|S| = 2 \cdot 10^6$ and $N = 2 \cdot 10^4$. And for each round, we run the algorithm against all 68 attributes sequentially.

- Time Complexity: From the description of the algorithm, we can see that for each round, the algorithm runs in $O(|S| \cdot |A|)$. If we run the algorithm for T rounds, the overall complexity is $O(T \cdot |S| \cdot |A|)$. We believe this time complexity is efficient compared to other possible solutions for this task, as merely examining all the instances once takes $O(|S| \cdot |A|)$.
- Order of Attribute: In step 3 of the algorithm, choosing the appropriate A_i for each iteration is crucial to the performance. In order to determine the order of selecting the attributes, we conducted an experiment to see how the number of classes within an attribute affects the *Hellinger distance*.
  In the experiment, we randomly generate $2 \cdot 10^6$ instances, and random select $2 \cdot 10^4$ instances from them, and calculate the distance between the two sets. We repeated the experiment for 100 times and took the mean as the result.

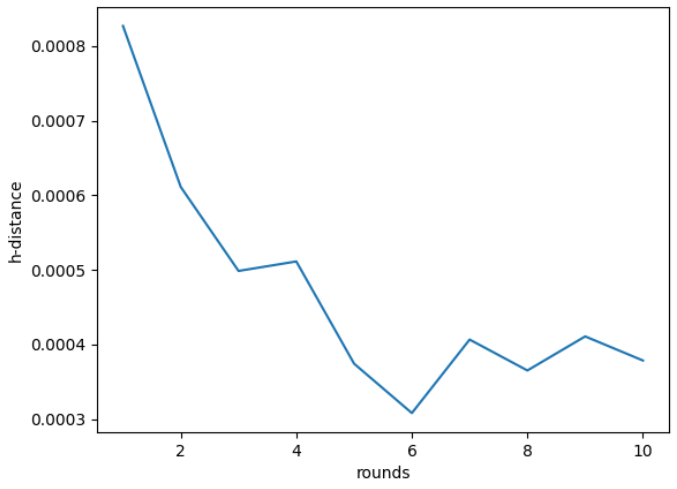


  In the figure, we see that as the number of classes within a attribute increases, a randomly selected subset tends to have a larger distance from the set of all instances.
  From the result of the experiment, we believe that in order to achieve a better result, we should place more importance on the attributes that have more classes within them. So we set the order of the attributes according the number of classes, with the latter having more classes.
- Performance of a single round



Overall, as we iterate through each attribute, the distance between S' and S decreases.

- Performance of multiple rounds



Although in the early stage, the performance improved as more rounds were conducted, the improvement did not last in the latter stage.

- Drawbacks: The main shortcoming of this algorithm is that, when we are focusing on a attribute, we disregard all other attributes. So it is possible that after processing A_i, it gets disturbed when we move on to A_{i+1}. We could resolve this issue by paying attention to all attributes that we have processed, trying not to disturb the probability distribution. However, this improvement requires more efforts on programming and would incur much more time complexity, so there is a tradeoff between performance and efficiency.

1.4 Evaluation Metrics

We can see that the result of *KL-Divergence* agrees with that of *Hellinger distance*. We think there are some reasons to use *Hellinger distance* instead of *KL-Divergence*.

- *KL-Divergence* does not deal with the case where some probabilities are 0 ($log 0$ is not defined). To remedy this, we assign $\Delta = 10^{-10}$ to replace 0.

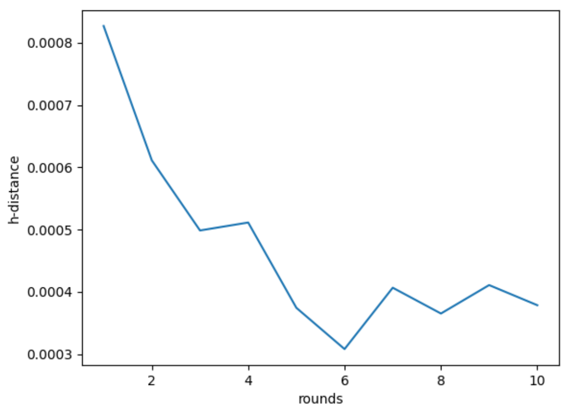


Figure 1: Hellinger Distance

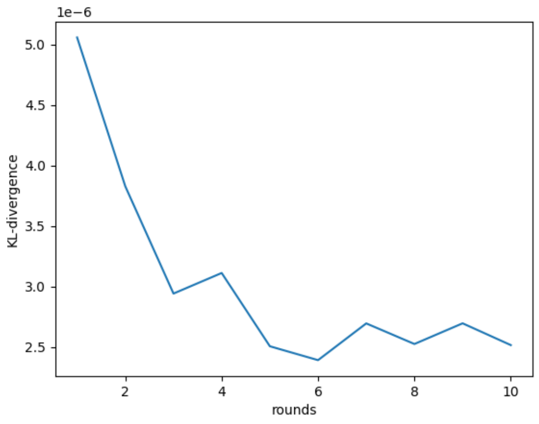


Figure 2: KL Divergence

- *KL-Divergence* is asymmetric. To remedy this, we calculate the *KL-Divergence* as $(KL(P||Q) + KL(Q||P))/2$.

Task 2

Because we did not get the chance to present task 2 in the class, all contents in this section are not included in our presentation.

2.1 Maintain the Ground Truth of Seen Data

In this section, we describe how we maintain the necessary information of the data stream. Since the data instances come sequentially, we will never get the real statistic until processing through the last one. However, according to *The Law of Large Number*, we can expect that our statistic approaches the real one after we see enough number of instances. In this task, we want to know the ranking of each rule based on the provided greedy algorithm, with constraints on both space complexity and time complexity. In the following, we will first describe our original approach that gives a more accurate statistic under the memory limit, then the compromised approach that is better in efficiency.

2.1.1 Notations

- *Rule-Generating Algorithm*: The greedy algorithm provided in the evaluation code
- D: the full data set
- A: the set of attributes
- R: the set of possible rules, where each rule $r_i \in R$ is in the form of $(attribute, value)$
- S: the sample set

2.1.2 The Original Approach

To run the *Rule-Generating Algorithm*, we should know the positive ratio of each rule **given any set of rules unsatisfied**. That is, the dependency between rules should be considered. To achieve this, we maintain a $|R| \times |R|$ table as follow:

Table 1: example when $|R| = 4$

	$\mathbf{r_1}$	$\mathbf{r_2}$	$\mathbf{r_3}$	$\mathbf{r_4}$
$\mathbf{r_1}$	■	$cnt(r_1 \cap r_2)$	$cnt(r_1 \cap r_3)$	$cnt(r_1 \cap r_4)$
$\mathbf{r_2}$	$cnt(r_1 \cap r_2 \cap +)$	■	$cnt(r_2 \cap r_3)$	$cnt(r_2 \cap r_4)$
$\mathbf{r_3}$	$cnt(r_1 \cap r_3 \cap +)$	$cnt(r_2 \cap r_3 \cap +)$	■	$cnt(r_3 \cap r_4)$
$\mathbf{r_4}$	$cnt(r_1 \cap r_4 \cap +)$	$cnt(r_2 \cap r_4 \cap +)$	$cnt(r_3 \cap r_4 \cap +)$	■

* $cnt(r)$ is the number of data instances that satisfy condition r
* + denotes the positive label

With this table, we approximate the positive ratio of any $r \in R$ given a set of rules S unsatisfied by this formula. Note that it is still not accurate because it does not consider the overlapping of rules (e.g. a instance that satisfies r_1, r_2 and r_3 simultaneously). However, we believe that it is more accurate than the compromised approach since it considers the dependency between rules.

$$\Pr\Big(\text{ label} = + \mid (r \text{ is satisfied}) \cap (r' \text{ is unsatisfied } \forall r' \in S)\ \Big)$$
$$\approx \left(\sum_{r' \notin S} cnt(r \cap r' \cap +)\right) \div \left(\sum_{r' \notin S} cnt(r \cap r')\right)$$

Then, we can simulate the *Rule-Generating algorithm*. The procedure is described as follow:

1. $S \leftarrow$ empty list
2. Select the rule r^* with the highest positive ratio given all rules in S are unsatisfied.

3. Append r^* to S.
4. If there is no positive case left, break; otherwise, go to stage 2.

However, this approach has a significant drawback, which is its time complexity. For every incoming data instance, we need to update the counting table, which takes $|A|^2$ operations. In this project, $|A| = 67$ in development set while $|A| = 200$ in the test set. In our experiments, it is estimated to take over 2 days to process through the development set with this approach. Hence, we think it is only viable when $|A|$ is reasonably small, so we deprecate it in this project and turn to the compromised one.

2.1.3 The Compromised Approach

To reduce the time complexity, we assume that the dependency between rules is low for most of the cases. With this assumption, we can just store $cnt(r_i)$ and $cnt(r_i \cap +)$ for $r_i \in R$ and the estimated rules and rank are generated by directly sorting R in the descending order of the positive ratio ($= cnt(r_i \cap +) \div cnt(r_i)$). It only requires $2|R|$ integers of memory and $|A|$ operations to update, so the space complexity is $\Theta(|R|)$ while the time complexity is $\Theta(|A|)$, which is a great improvement compared to the ideal approach. In our experiments, it takes only 1 hour to process through the development set and 10 minutes for the test set. Thus, we choose this method as the final solution.

2.2 Sampling Based on Known Rank

No matter which method is used to generate the current known rules and rank, we assume it is the ground truth that the sample set should follow. Our final goal is to make the result of the *Rule-Generating Algorithm* on the sample set similar to the result on the full data. Theoretically, if the distribution of the sample set is exactly the same as the full data, the result of the algorithm will also be the same. However, instead of maintaining the distribution, we choose to maintain the rank because the nDCG score is calculated based on the rank.

2.2.1 Maintain the Rank of the Sample Set

We use a simple heuristic to maintain the rank by manipulating the number of positive instances and negative instances of a rule. For every rule, we define the target number of positive and negative instances before starting sampling, such that the rank of positive ratio is the same as the ground truth. The exact number is shown as follow:

Table 2: target number of positive/negative instances of each ranking

	1	**2**	**3**	...
Positive	POS	POS	POS	...
Negative	0	1	2	...

* POS is a parameter that can be adjusted

That is, let the ranking of r_i be k, then the target numbers of positive and negative instances are POS and $k - 1$ respectively. Additionally, the rules with higher ranking have higher priority to be satisfied because they have larger weights when calculating nDCG score. Since their denominators are the same, the positive ratio depends on the number of negative instances. So if the target is perfectly satisfied, then the rank of rules will be exactly the same as the ground truth, or we can at least expect that it is true for the highest several rankings. However, when we add one instance to our sample, it actually changes the counts of $|A|$ rules. These “noises” can influence the result greatly. To overcome this obstacle, we come up with a noise elimination scheme described in the next subsection.

2.2.2 Noise Elimination

If we just add a data instance if it helps to achieve the target, we will get very bad score because of the noises from the other attributes. To eliminate such noises, we maintain the current rank of the sample data and perform a check when we see a new data instance. The algorithm is described as follow:

1. For i from 0 to $M-1$:
   (a) Let $S[i]$ be the rule ranked i in the sample set
   (b) If $S[i]$ not in the top K of the ground truth rank and the incoming instance is a negative case of $S[i]$, add this instance to S.

That is, we try to reduce a rule's ranking by increasing the number of negative instances satisfying this rule. Both M and K are tunable parameters. In our experiments, the choice of M and K depends on the number of rules. If the number of rules is higher, then M and K should be smaller to release the constraints. A possible explanation is that there are more noises when there are more rules so we should not pay too much attention to them which may distract us from the other rules. Therefore, on the development set, we let $M = K = 5$, while on the test set, we let $M = K = 1$, since the number of rules is much higher on the test set.

2.2.3 Timing to Start Sampling

Because before we saw a sufficient number of data instances, the statistic we obtain should be inaccurate. Actually, if we start sampling in the beginning, the performance will be really bad. Thus, we should decide a good starting point. Since we know the total number of instances in advance, we can decide a proper proportion. After some experiments, We decide to start after we saw 50% of the data, and stop updating the statistic after this. We believe that 50% is great enough to obtain a accurate result, so we do not need to continue the time-consuming updating procedure.

2.3 Comments on Randomness

Our algorithm does not have any randomness given fixed coming order, so the result should be deterministic. However, if the coming order is randomized, the result can vary greatly. We will describe this phenomenon is the next section.

2.4 Mean and Variance of Random Coming Order

2.4.1 Statistic

We do 20 times of random experiments that shuffle the coming order of data on the test set.

- **mean**: 0.423274
- **variance**: 0.003121

2.4.2 Explanation

We find that there is a huge decrease in performance if we random shuffle the original data ($0.565717 \to 0.412912$), which ends up being worse than the random solution (0.45). We think it is because our algorithm is easy to get stuck in a target that is hard or impossible to be satisfied. For example, it may keep finding a positive instance of a rule which may be at the end of the data set or even not exist. If such a thing happens, only the first few rules can be satisfied while the other is totally ignored.

Workload Balance

- B07902001 徐維謙: $\frac{1}{3}$
- B07902064 蔡銘軒: $\frac{1}{3}$
- B07902075 林楷恩: $\frac{1}{3}$