Probabilistic Database Project Final Report

Yanzhao Wang, Michael Xiong, Zhoutian Yuan

December 14, 2018

1 Group Members

Yanzhao Wang, UID: 405229892
Email: wyz8175@gmail.com

Michael Xiong, UID: 404463570
Email: michaelx.2007@gmail.com

Zhoutian Yuan, UID: 605231224
Email: yjohnyuan@gmail.com

2 Code and Dependencies

Our probabilistic database can be found at `https://github.com/yjohnyuan/CS267A_Probabilistic_Database`, which includes the implementation of our algorithm, example query files, example table files, test cases and extensions.

The only libraries in addition to pure Python (Python 2 and Python 3 both work) we used were NumPy 1.11.3 and Pandas 0.23.4. While the code may still work with older version of this software, we tested and implemented our algorithm on Pandas version 0.23.4, and found that older versions of Pandas could cause problems since there's a groupby() function missing in older versions.

3 Implementation

The implementation of our probabilistic database was done in Python, because libraries such as NumPy and Pandas are readily available and provide easy means of representing and manipulating tables and list of queries in our database.

3.1 Database Representation

The base case of lifted inference involves simply reading probabilities of an event from the database. Similar to queries, we implemented a parser to convert from the text representation of a database to its Python equivalent, which we chose to be a dictionary of Pandas dataframes. Pandas provided a natural representation of tables and easy manipulation of data in cases such as computing negations.

3.2 Query Representation

In order to perform query evaluation, our algorithm had to conduct multiple rounds of query rewriting. Therefore, we first implemented a query class to easily and uniquely represent a query, and a parser to convert from a query's string representation to its object representation in Python. This class is comprised of two lists, representing the predicates and variables in the query respectively. For example, a query of the form: $\exists x, y R(x, y), Q(x)$ would be represented by lists: (predicates:$[[R, Q]]$ and variables:$[[[x, y], [x]]]$). The predicates list is simply a list of clauses, where each clause contains the predicate name in the clause, and the variables lists is a list of clauses, where each clause contains lists of variables in each predicate. Because we assume existential quantification and UCQ form, these two lists are able to fully represent a query.

In addition to the two lists representing our query, we implemented a third list to serve as a mapping from the predicate names in our query to the dataframe column names in our database. For example, the query $Q(x, y), R(x, y)$ is clearly differ-

ent from $Q(x, y), R(y, x)$, but our database would only read the predicate R with columns "Var1" and "Var2", and would have no way of distinguishing the variables referring to each column. This third list existed as an implementation-specific detail, allowing us to properly evaluate these two different queries.

3.3 Lifted Inference Details

We built our inference algorithm iteratively, starting with the simplest cases such as $\exists x R(x)$, which involved simple negation and summation of a Pandas dataframe. From the basic case, we were able to also solve slightly more complex queries such as $\exists x, y R(x, y) \wedge Q(x)$. Ultimately, we were able to build up to solving complex queries such as $\exists x_1, y_1, x_2, y_2 (Q(x_1, y_1) \wedge R(x_1)) \vee (Q(x_2, y_2) \wedge S(x_2))$.

4 Algorithm Details and Testing

In our iterative approach to building our lifted inference algorithm, we tested with progressively more complex queries that we built our algorithm to handle. Test tables are stored in t1.txt through t5.txt, and the queries tested are in query1.txt through query10.txt.

4.1 Base Case

The base case of lifted inference involves a single ground atom, resulting in a look up in the database. However, because our database representation uses Pandas dataframes, it is trivial to compute $Pr(\exists x Q(x)) = 1 - \Pi_x (1 - Pr(Q(x)))$ with Pandas operations, so we simply make this our base case instead. This test case can be run with the command:

```
$ python qeval.py
-q query1.txt -t t2.txt
```

4.2 Conjunction with Separator

We built upon our base case with a slightly more complex query of the form: $\exists x, y R(x, y) \wedge Q(x)$. First we perform a panda groupby operation on the separator variable x to generate a dataframe with only the separator variable. For example: R(x) instead of R(x,y). The probabilities in R(x) correspond to at least one R(x,y) combination is true for a given x. The benefit of this design is that for all values of x, we only need to loop through y values once. Then the two dataframes P(x) and R(x) are then joined on x to get a dataframe that includes the probability of x in predicate R and predicate Q. This procedure is an implementation of equation $Pr_{total} = (1 - \Pi_x [1 - Pr(Q(x)) * (1 - \Pi_y (1 - Pr(R(x, y))))$ with the probability of single ground atom computed. This test case can be run with the command:

```
$ python qeval.py -q query2.txt
-t t2.txt -t t3.txt
```

4.3 Conjunction of Independent Clauses

The natural next step from solving conjunctions with separators was to solve conjunctions of independent queries, where each query was independent from others and had a separator. An example query is: $\exists x_1, y_1, x_2, y_2 R(x_1, y_1) \wedge Q(x_1) \wedge S(x_2, y_2) \wedge P(x_2)$.

In this query, there are two independent queries $\exists x_1, y_1 R(x_1, y_1) \wedge Q(x_1)$ (note as Q_1) and $\exists x_2, y_2 S(x_2, y_2) \wedge P(x_2)$ (note as Q_2). This is the decomposable and rule in lifted algorithm and the result is $Pr(Q_1) * Pr(Q_2)$, while $Pr(Q_1)$ and $Pr(Q_2)$ are solved in last case. This test case can be run with the command:

```
$ python qeval.py -q query3.txt -t
t1.txt -t t2.txt -t t3.txt -t t4.txt
```

4.4 Disjunction of Independent Clauses

As we are solving existential quantification and UCQ form queris, for unions of CNF clauses, only one CNF clause needs to be true to satisfy the existential quantification. Therefore, the CNF queries are independent with each other if there are no intersection between predicates contained in queries. For example, in the query $\exists x, y R(x, y) \vee Q(x)$, $R(x, y)$ and $Q(x)$ do not share common predicates so they are independent and the query is equal to $\exists x_1, y_1, x_2 R(x_1, y_1) \vee Q(x_2)$. Specifically, we use an outer join when merging two dataframes and let

$Pr(R(x_2)) = 0$ if x_2 is not in R's domain to ensure the two queries are the same.

Then by the decomposable or rule in lifted algorithm, the result is equal to $1 - (1 - P(\exists x_1, y_1 R(x_1, y_1)))(1 - P(\exists x_2 Q(x_2)))$, and the probability of each CNF clause is computed by algorithm stated in previous sections. This test case can be run with the command:

```
$ python qeval.py -q query4.txt
-t t2.txt -t t3.txt
```

4.5 Conjunction of Dependent Clauses

Consider the query $\exists x_1, y_1, x_2, y_2 P(x_1) \wedge R(x_1, y_1) \wedge Q(x_2) \wedge R(x_2, y_2)$. It can be divided into two queries $\exists x_1, y_1 P(x_1) \wedge R(x_1, y_1)$ (note as Q_1) and $\exists x_2, y_2 Q(x_2) \wedge R(x_2, y_2)$ (note as Q_2), but these two queries are not independent because they have a shared predicate R. In this case, we must apply Inclusion/Exclusion Rule in lifted algorithm, that is $P(Q_1 \wedge Q_2) = P(Q_1) + P(Q_2) - P(Q_1 \vee Q_2)$

Here $P(Q_1)$ and $P(Q_2)$ can be computed given formulas above, and how to compute $P(Q_1 \vee Q_2)$ will be described in the following section.

4.6 Disjunction of Dependent Clauses and Rewriting Queries

Still considering the case $\exists x_1, y_1, x_2, y_2 (P(x_1) \wedge R(x_1, y_1)) \vee (Q(x_2) \wedge R(x_2, y_2))$, which is in the form of $P(Q_1 \vee Q_2)$. Here Q_1 and Q_2 are not independent as they share a common predicate R. By the distributive law, we can take out the predicate R and rewrite the query as $\exists x, y R(x, y) \wedge (P(x) \vee Q(x))$.

Here we can generate a dataframe table of $R(x, y)$ and $P(x) \vee Q(x)$, which can be attained by the computation procedure in previous sections. Then join the two dataframes based on the grounding variable (x in this case), we can get the probability of $\exists x, y R(x, y) \wedge (P(x) \vee Q(x))$.

Hence we get the answer of $P(Q_1 \vee Q_2)$ and as a result the Inclusion/Exclusion formula $P(Q_1 \wedge Q_2) = P(Q_1) + P(Q_2) - P(Q_1 \vee Q_2)$ is completed. This test case can be run with the command:

```
$ python qeval.py -q query5.txt -t t1.txt
-t t2.txt -t t3.txt
```

4.7 Unliftable Cases

There are two cases that is unliftable in our experiments.
1. $\exists x, y P(x) \wedge R(x, y) \wedge Q(y)$
2. $\exists x, y (P(x) \wedge R(x, y)) \vee (Q(x) \wedge R(x, y)) \vee (P(x) \wedge Q(y))$
In the first case the variables of all predicates are in one connected component, but there are no common variables that can be grounded on, hence the lifted algorithm fails. And in the second case all CNF clauses are not independent with each other. Our algorithm will stop when detecting these two cases and print an error message.

4.8 Conclusion

With all these computation function implemented, our algorithm is able to solve complex queries. For example, $\exists x, y T(x) \vee (Q(y) \wedge R(x, y)) \vee (P(x) \wedge R(x, y) \wedge S(x))$:

1. The query will be divided into two independent UCQ clauses (one is $T(x)$ and the rest are the other one)

2. The second query is simplified into $\exists x, y R(x, y) \wedge (Q(y) \vee (P(x) \wedge S(x))$

3. Solve the second query by solving $Q(y) \vee (P(x) \wedge S(x))$ recursively using decomposable and/or rules, join the result dataframe with $R(x, y)$ and get the probability grounding on x

4. Using decomposable or rule to combine the probability of two UCQs and get the final result.

5 Extension

For our extension we incorporated National Basketball League stats provided by Kaggle into our PDB

algorithm. Since the data was denormalized and dirty, we first loaded our data sets into SQL server 2017 on a windows machine to remove erroneous records, aggregate and filter our results. We loaded 3 csv files containing player info, salary data, and season stats into sql tables. A sample SQL script which we used to compile our probabilities can be found in appendix 1. All of these extension queries can be executed with the readme_nba.txt file included.
Using our PDB algortihm and the loaded table data, we explored the following scenarios:

1. What is the probability that you are making at least 20 million dollars as a NBA player in 2017?
For this we exercise we created a t_salary.txt which gives the probability for each salary range above 20 million dollars. We found that for the 2017 season, if you are a NBA player, there is a 0.0827 probability you are making at least 20 million dollars.

2. What is the probability that an NBA player is being overpaid?
We defined overpaid as making over 20 million dollars and scoring under 1000 points in a season, so our query was $\exists Salary(x) \wedge Overpaid(x, y)$. We calculated the probability to be overpaid is 0.0132.

3. What is the probability that your favorite team has at least one overpaid player?
Since there are 15 players on each NBA roster. We simply create a new database $MyTeam(x)$, where x is the probability of $Salary(x) \wedge Overpaid(x, y)$. with 15 entries in the table. We query $\exists MyTeam(x)$. This gives each player an equivalent chance of being overpaid.

We calculated that there is a 0.1811 probability that there is at least one overpaid player on your favorite team.
What we learned from this example is the PDB algorithm allows us to quickly calculate probabilities from real data sets. These calculations would otherwise take lots of manual work.

4. Finally, we ran historical analysis.
Assuming you're an NBA star, what is the probability that over the last ten years, you've scored at least 1000 points in a season? To do this analysis we need to create a table, thousand, and query $\exists thousand(x)$. Our thousand table has one entry for each year between 2008-2017.

Our results show thousand(x)=0.7637, so if you played in the NBA over the last ten years, there is a 76.37% probability that you've scored at least 1000 points once. This is a pretty high probability and essentially means that three quarters of all NBA players should have scored 1000 points at least once in the last decade, something that does not make intuitive sense.

This example illustrates a big problem in our tuple independence assumption. In reality, the probability that a player scores a set amount of points is not independent from year to year. Factors like player skill, injury, personal motivation alter this probability quite a bit, so the probability that we calculated is most likely higher or lower depending on the situation. Its very difficult to find truly tuple independent databases in real data sets, as there are often underlying dependencies.

Overall, by combining the NBA dataset and our PDB algorithm, we were able to solve statistical questions about the NBA easily. The challenge we came upon was that many of the statistics were not tuple independent so we could not implement many of the queries we wanted to.

6 Project Feedback

We thoroughly enjoyed coding the PDB algorithm once we figured out how to define our query class and how to recursively define our query inputs. We learned a lot about pandas and python through the process. In the extension we learned a lot about tuple independence and some of the limitations of the PDB. The initial ramp up for the project was a bit rough as we were unclear about the desired results and project spec. The guide that the TA's posted greatly helped. It would have been very helpful to have more sample queries to double check our work.

7 References

We downloaded the following datasets
from Kaggle:
[1] https://www.kaggle.com/drgilermo/NBA-players-stats#Seasons_Stats.csv
[2] https://www.kaggle.com/drgilermo/NBA-players-stats#Players.csv
[3] https://www.kaggle.com/koki25ando/salaryNBA_season1718_salary.csv

8 Appendix

8.1 Appendix 1

A sample query used to compile probabilities:

```
select top_10_stat.year,
    stat.tm,
    stat.player,
    cast(salary.salary as int) sal_calc,
    stat.pts
from [basketball].[dbo].[Season_Salary_2017] Salary
left join (select year,pts,player FROM [basketball].[dbo].[Season_Stats]
where year ='2011') Stat
on Salary.Player=Stat.Player
order by cast(stat.pts as int) desc
```