Dataset creation from Tabular data

Namit Shetty NAMITS@ANDREW.CMU.EDU
Namit Katariya NKATARIY@ANDREW.CMU.EDU

Abstract

Tables are the standard way of describing relational data and a lot of such tables appear on the web. However, from a machine learning perspective, this data is not in a good format for analysis and learning tasks. One of the ways of representing this data is to assign a unique id to each column in each table and list the entities in that column for each such id. Are there other, possibly better, ways of leveraging tabular data by considering other representations of the same? We worked on generating some such representations of the tables in the ClueWeb09 dataset.

1. Introduction

1.1. Original dataset

The dataset we used consisted of tables from the ClueWeb09 dataset. The dataset generation pipeline considered only those table cells to contain entities (or noun-phrases) [1] which have length in between 2 to 100 characters. Each column in any table is given a unique id which identifies its table as well as the specific column in that table. The dataset is represented as `id <tab> noun-phrase-1 <delim> count <tab> noun-phrase-2 <delim> count <tab> ...`

Below are some of the relevant dataset statistics

Table 1. Dataset statistics

# Unique noun-phrases	3.8 million
# Unique table columns	8.9 million
# Unique tables	1.4 million
Size of dataset	24 GB

[1] We use the terms entity and noun-phrase interchangeably. Both refer to the content of a table cell.

Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

1.2. Motivation

Are there other representations of the same tabular data that could prove useful? Suppose our task is to classify the given noun-phrases into some fixed set of categories given such tabular data and also, some seed training data that specifies the categories of some of the noun phrases. For example, given Pittsburgh, we might want to know whether it's a city or a state or a sports team and so on. What if we want to classify an ambiguous entity such as "Michael Jordan"? Note that in a table, if Michael Jordan appears along with say Andrew Ng or Tom Mitchell, then we can claim with confidence that Michael Jordan here is the ML researcher. However, if it appears along with O'Neil or Kobe Bryant, then he is more likely the basketball player.

This hints towards the fact that bigrams (co-occurrences, in general) might be more useful while making inferences about the properties of the noun-phrases in a table.

2. Representations

Although inverted index is a useful representation, we primarily consider the bigram representation and its variation in this project. However, for sake of completeness, we mention the inverted index idea here.

- **Inverted Index** : This is the same idea that is used in information retrieval where given a set of documents, an index is built where each word is mapped to the documents in which it appears. Analogously, the idea here is to map each entity to the column ids in which it appears. The column ids, thus, are the features here.
- **Bigram representation** : As mentioned in 1.2, we want to capture co-occurrences of entities. Therefore, for an entity in a column, its features are the bigrams that appear in that column. Note that if a certain pair of entities appears in two different columns, in the feature representation we increment the count of that specific bigram. A good analogy is to consider each entity to be a

document and the bigrams to be words. In context of this analogy, we are effectively using a bag-of-words model to represent an entity.

We looked at quite a few variations of this idea. To illustrate these, consider the following column of a table

Pittsburgh
New York
Seattle

All-pairs: Here we map each entity to all the $\binom{n}{2}$ bigrams in that column (n is the number of entities in the column) Thus, `Pittsburgh`, `New York` and `Seattle` will all get mapped to `[Pittsburgh-New York, New York-Seattle, Pittsburgh-Seattle]`

Consecutive : Primarily due to the large size of the data that results from running all-pairs bigram mapping and secondly, for the sake of simplicity, we also considered mapping each entity in a column to only the consecutive bigrams in the column. For example, `Pittsburgh, New York and Seattle` all get mapped to `[Pittsburgh-New York, New York-Seattle]` Since taking consecutive bigrams is slightly arbitrary, one can sort each column before forming the bigram – this can act as some kind of normalization for this procedure.

PMI : Another interesting option is to consider for each column, only the top-k bigrams by their pointwise mutual information scores. A variation of this approach can be to consider those bigrams that have a PMI score above a certain pre-decided threshold.

Pointwise mutual information of a bigram x–y is defined as $log(\frac{\Pr(x,y)}{\Pr(x).\Pr(y)})$ where Pr(x, y) is the probability of x and y occurring together while Pr(x) is the probability of occurrence of x. It can be estimated by approximating the probability with the individual counts. Since we were only interested in the relative order of PMIs, we did not consider logarithms and treated the fraction to be the PMI value for simplicity.

3. Bigram representation

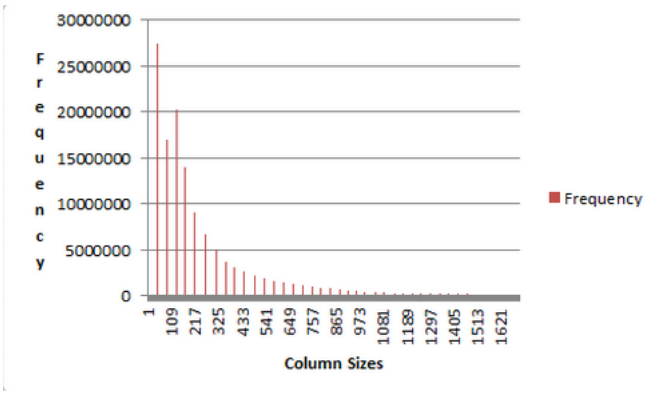
For simplicity, we will consider the consecutive case here. The pipeline to generate the requisite representation is fairly simple. It is given below as algorithm 1

Algorithm 1 Bigrams MapReduce pipeline

Mapper :
Read a line `id <tab> np1 <delim> count ...`
For key np1, emit `np1-np2, np2-np3, ...`
Reducer :
For a key, sum counts of corresponding bigrams
Emit `key <tab> bigram1 <delim> count ...`

Although the pipeline is simple and straight-forward, there are a lot of practical problems here. Firstly, let us see if it is feasible to run such a pipeline on the given data by estimating the size of the generated data.

Note that if a column has n entities, each entity is associated with $n-1$ consecutive bigrams so that the size of the output for this column is $O(n^2)$. To get a preliminary idea of what we are dealing with, we also plotted a histogram of the column sizes and their frequency which is produced below.



So the size of the generated data is going to be $\sum_n O(n^2)$ We calculated this for our data and this quantity turns out to be of the order of 90 billion. Even with a relatively small estimate of 20 bytes per bigram, the size of the data comes out to be 1.8 TB. This is the size of the final data and the intermediate data size can be even bigger. The HDFS size on Opencloud, the cluster on which we were working, is itself 1 TB. Clearly, we need to change something.

One can use a few adhoc techniques to reduce this size

- **Discard infrequent bigrams :** Although this can be done, infrequency need not mean irrelevant. We did not find this option sound for our purposes hence we went with column thresholding described next.
- **Column thresholding :** We could consider only those columns to process in which the number of

entities is above l and below u for choices of l and u that make the resulting data tractable.

- **Strict definition of entity :** As of now, there are a lot of entities in the dataset which are spam. For example, one of the entity in the dataset was - - - - - i.e just 5 hyphens. We could discard such entries by enforcing a string definition of an entity by considering only those that are recognized in the NELL corpus.

4. Bigrams with high PMI

We next worked on trying to map each entity in a column to the top-k bigrams in that column. The idea is that this might discard some spammy bigrams and at the same time, retain only the more informative ones. Also, if a bigram is a top-k bigram for more than one column, then that results in its count being incremented in the feature representation of the entities which map to it. So for example, if there is a column containing `Pittsburgh, New York, Seattle` and another column containing `Pittsburgh, New York, San Francisco` and if `Pittsburgh-New York` has sufficiently high PMI, then `Pittsburgh` (as well as `New York)` would be mapped to `Pittsburgh-New York` with a count of 2.

Rather than keep a fixed k, another approach is to simply threshold the bigrams by their PMI values i.e consider only those bigrams who PMI is above a certain value t. t can be chosen by looking at the minimum, maximum and variation of the PMI values and selecting a suitable value.

4.1. Failed attempt 1

The first idea that we considered was as follows: Given a line in the input file `id <tab> np1 <delim> count ...`, for each entity $\texttt{np}_i$, we output all the entities in the column with their respective counts and also the consecutive bigrams with count 1. For example, if the line was `xyz Pittsburgh-1 NY-2 Seattle-1`, then I would output [Pittsburgh, 1], [NY, 2] and [Seattle, 1] as well as [Pittsburgh-NY, 1] and [NY-Seattle 1] with each of Pittsburgh, NY and Seattle.

It's easy to see that the reducer, for a particular entity (key), will therefore receive all the relevant entity counts as well as bigram counts. Hence, the reducer can compute the PMI of each bigram and hence choose the top-k or thresholded bigrams for that particular entity to get its feature representation.

However, the size of the intermediate data in this case will be huge because if the column size is n, then the $O(n)$ entity counts and the $O(n)$ bigram counts are repeated for each of the n entities in the column. Moreover, this will happen for every column. This approach, therefore, although simple and correct, is practically infeasible. The approach is outlined in algorithm 2

Algorithm 2 Bigrams-PMI Naive pipeline

Mapper :
Read a line `id <tab> np1 <delim> count ...`
$\forall$ $\texttt{np}_i$, emit [$\texttt{np}_j$ `count`] $\forall j$ and [$\texttt{np}_i$-$\texttt{np}_{i+1}$ `1`] $\forall i$

Reducer :
For a key, sum counts of corresponding entities and bigrams
Calculate the PMI of each bigram and choose top-k / thresholded bigrams

4.2. Failed attempt 2

The idea here was to get the PMI values of all bigrams and then go through the input file again to get the top-k or thresholded bigrams. For this, we realized that since one needs to query for the PMI of each bigram while going through the input file line-by-line, one would need a "table" which supports efficient querying. Also, considering the large size of the data and the potential to parallelize the queries (each line can be processed independently), a table that supports distributed querying would be handy. This is when we learnt about Apache Accumulo. Accumulo is a distributed key-value store that provides cell-based access control and customizable server-side processing.

Thus, using accumulo, we were planning on creating the required dataset using the pipeline described in algorithm 3

Algorithm 3 Bigrams-PMI using Accumulo

Phase 1
Mapper :
$\forall$ entity x in column, emit [x, 1] to Accumulo
$\forall$ bigram (x-y), emit [(x-y), 1] to Accumulo
Reducer :
Use Accumulo iterators to aggregate counts
Phase 2
Mapper :
$\forall$ (x-y), compute PMI by querying Accumulo
Emit top-k / thresholded bigrams as features
Reducer :
$\forall$ entity, merge features and aggregate counts

We implemented the above pipeline and got all the counts aggregated in accumulo. However, while trying

to learn querying of accumulo, we noted that query answering time of accumulo was of the order of 10k random queries per second. However, by the same logic as given previously, there will be $O(n)$ queries per column and thus a total of $O(n^2)$ queries in the worst case. Given the query time of accumulo and that there were only 4 accumulo servers, this approach turned out to be impractical.

4.3. Alternative approach

Since querying accumulo was infeasible, we needed a different approach. Hence we came up with the following extensive, five-stage pipeline described in algorithm 4

4.3.1. Stage 1

This is a simple MapReduce pipeline that counts the occurrences of entities as well as bigrams.

Mapper : For every column `col-id` $\texttt{np}_1$ $\texttt{np}_2$ $\texttt{np}_3$ ... in the input, it outputs [`(`$\texttt{np}_i$`,*)` 1] for every entity and [`(`$\texttt{np}_i$`,`$\texttt{np}_j$`)` 1] for every consecutive bigram. The '`*`' is for the next stage to know whether it is a entity count or a bigram count.

Reducer : The reducer simply aggregates the counts for each key thus generating the occurrence counts for each entity and bigram.

4.3.2. Stage 2

This stage has two parts, PMI calculation and Inverted index.

- PMI calculation : The input for this stage is the output of stage 1 i.e the entity and bigram counts.

  There are a bunch of things to be noted here. Firstly, we assume that every mapper knows the number of reducers that will be deployed in the reduce phase. Secondly, the character & does not occur anywhere in the dataset and so we use it to design our partitioner for this stage. We wrote a MapReduce pipeline to find out characters that do not appear anywhere in the original data so that they can be used as delimiters.

  Mapper : For every line of the form [`(`$\texttt{np}_i$`,*)` count], it outputs [`(&`$\texttt{np}_i$`,*&r)` count] $\forall r = 1 \cdots R$ where R is the number of reducers. Thus each entity count is outputted R times. The idea is that the partitioner will send one of these entries to each of the reducers because the reducer needs the entity counts to calculate the PMIs. For every line of the form [`(`$\texttt{np}_i$`,`$\texttt{np}_j$`)` count], it simply outputs the same line again without any change.

  Partitioner : If the key starts with a '`&`', the key will be of the form [`(&`$\texttt{np}_i$`,*&r)` count]. In this case, it sends the entry to reducer `r`. Otherwise, it performs its default behaviour which is to send it to the reducer obtained by finding the remainder the key's hash code leaves on dividing by the number of reducers.

  Reducer : Note that each reducer will get all the entity counts. The number of entities is tractable and can be kept in memory (see table 1.1). Now, the reducer receives the keys in sorted order. The specific structure of the key (the preceding '`&`' character) exploits this ensuring that all the entity counts appear before any bigram counts thus making PMI calculation possible. The reducer simply puts the count in a hash map if the key starts with a '`&`' else it computes the bigram's PMI and outputs [bigram`&` PMI-value]

- Inverted Index : Here we simply build an inverted index on the bigrams by mapping each bigram to the column ids in which it appears.

  Mapper : For every bigram $\texttt{np}_i$`-`$\texttt{np}_j$, output [`(`$\texttt{np}_i$`,`$\texttt{np}_j$`*)` column-id]

  Reducer : The reducer simply outputs whatever it reads

4.3.3. Stage 3

This stage takes as input the outputs of both the parts of stage 2, the PMIs and the inverted index.

Mapper : Identity i.e outputs [bigram`&` PMI-value] as [bigram`&` PMI-value] and [`(`$\texttt{np}_i$`,`$\texttt{np}_j$`*)` column-id] as [`(`$\texttt{np}_i$`,`$\texttt{np}_j$`*)` column-id]

Partitioner : The partitioner treats as its input the key upto its penultimate character. The idea is that bigram`*` and bigram`&` should both go to the same reducer.

Reducer : The reducer receives its input in sorted order which ensures that bigram`&` i.e the PMI value appears before the column ids. Thus for each column-id that the bigram appears in, the reducer outputs [column-id, (bigram, PMI-value)]

4.3.4. Stage 4

Mapper : Identity mapper

Reducer : The records received here would be of the [column-id `(x,y)`,PMI] The reducer maintains two sets – `column-entities` and `features`. For each record, the reducer adds `x, y` to the set `column-entities` and if the PMI value is above the

pre-decided `threshold`, then it adds the bigram `(x,y)` to the `features` set. When a key has been processed entirely i.e the sets have been built, it then goes through the `column-entities` set and for each entity, emits each bigram in the `features` set.

4.3.5. Stage 5

Mapper : Identity mapper

Reducer : Corresponding to each entity, it aggregates the features i.e bigrams and their counts and emits the feature representation.

4.4. Progress

We have implemented the entire pipeline, as of now. We have produced the datasets corresponding to different thresholding values for PMI. We have summarized the results in 4.5. In particular, refer table 2

4.5. Some PMI statistics

We found that the maximum PMI value is 0.1 while the minimum is of the order 10^{-13}.

We have summarized some of the statistics that we got by running the pipeline with different thresholding values for PMI in table 2. We ran the pipeline and noted the size of the data at the end of stage 4 and also the size of the data at the end of stage 5 and the number of examples we end up with in the final dataset.

At `threshold` = 0.01 which turns out to be the 99th percentile PMI value, stage 4 fails after generating around 500 gigabytes of data. Hence we could not get the statistics for lower thresholds.

After seeing these, we wanted to plot a histogram of the PMIs. However, both GNUPlot and Octave failed with "out of memory" error. Due to lack of time, we did not research it further. This is something that would be good to see. By plain inspection, we did note that most of the low values i.e around 10^{-13} to around 10^{-7} had frequency 1 while the ones closer to 0.1 had higher frequencies. For example, the frequency of 0.1 PMI value is 242.

Algorithm 4 Bigram PMIs multi-stage pipeline

Stage 1
- **Mapper :**
  - ∀ entity x in column, emit [`(x,*)` 1]
  - ∀ bigram `(x-y)`, emit [`(x-y)`, 1]
- **Reducer :**
  - Aggregate counts ∀ entity and bigram

Stage 2.1
- **Mapper :**
  - for key `(x,*) count`, emit [`(&x,*&r)` count]
  - for key `(x,y) count`, emit [`(x,y)` count]
- **Partitioner :**
  - **if** (key == `(&x,*&r)`)
    - return `r`
  - **else:**
    - return `key.hashcode() % numReducers`
- **Reducer :**
  - Initialize hash map `entity-count`
  - **if** (key == `(&x,*&r)`)
    - `entity-count.add(x, count)`
  - **else:**
    - compute PMI of bigram
    - emit (`bigram&`, PMI)

Stage 2.2
- **Mapper :**
  - ∀ bigram `(x-y)`, emit [`(x-y)`, column-id]
- **Reducer :**
  - emit [`bigram*`, column-id]

Stage 3
- **Mapper :**
  - Identity mapper
- **Partitioner :**
  - return `key[0:len(key)-1].hashcode() % numReducers`
- **Reducer :**
  - **if** (key == `bigram&`)
    - lastPMI = value
  - **else:**
    - for all column-ids:
      - emit (column-id `bigram`, lastPMI)

Stage 4
- **Mapper :**
  - Identity mapper
- **Reducer :**
  - ∀ [`(x,y)` PMI)]:
    - Add `x, y` to the set `column-entities`
    - **if** (PMI > `threshold`)
      - Add `(x,y)` to the set `features`
    - **for** x in `column-entities`:
      - **for** bigram in `features`:
        - emit [x, bigram]

Stage 5
- **Mapper :**
  - Identity mapper
- **Reducer :**
  - Aggregate features as well as counts for each entity and output

Table 2. PMI statistics

Threshold PMI	Stage 4 end Data size	Stage 5 end Data size	Stage 5 end # ex
0.09	2.8 MB	0.4 MB	2,541
0.08	4.8 MB	0.7 MB	5,262
0.07	10.1 MB	1.5 MB	9,766
0.06	21.6 MB	2.9 MB	16,228
0.05	49.4 MB	6.3 MB	34,499
0.04	107.5 MB	11.8 MB	63,006
0.03	6.1 GB	251.1 MB	268,121
0.02	10.6 GB	392.4 MB	497,325
0.019	55.5 GB	1.3 GB	970,876
0.018	64.6 GB	1.5 GB	1,214,031
0.017	73.4 GB	1.8 GB	1,434,191

5. Difficulties

The major difficulty we faced in this project was the instability of the Opencloud cluster on which we were working. It was very frustrating to be not able to login or to login, write the code but not being able to test it due to HDFS being down or busy with a heavy task.

Apart from that, while designing any pipeline, we had to always keep the size of the intermediate as well as output data in mind due to the prohibitive size of the ClueWeb09 data. Most of the naive single stage pipelines that we came up with turned out to be impractical for this reason and we had to always come up with better ways of accomplishing the same task.

6. Conclusions and Future Work

This turned out to be a true "big-data" project and we faced numerous difficulties dealing with it. For example, an important takeaway was that simply estimating the size of the final data isn't enough but we also need to care about intermediate data and whether it is tractable. Nevertheless, it was a very good learning experience to work in such a setting.

We would have loved to be able to actually run a few simple algorithms such as Naive Bayes for some of the motivating tasks mentioned in 1.2 but the instability of Opencloud did not help our cause. Running some semi-supervised or unsupervised algorithm on the generated dataset is something that can be done in the future. Also, whether there are less complex and shorter pipelines to generate the data is something that can be given a thought.

Acknowledgments

We would like to thank Prof. William and Bhavana who helped us with the project. A special thanks to Lin Xiao who went out of her way to help us out when we were facing problems with the Opencloud cluster and also with learning Accumulo.

References

Dalvi, Bhavana Bharat, Cohen, William W, and Callan, Jamie. Websets: Extracting sets of entities from the web using unsupervised information extraction. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pp. 243–252. ACM, 2012.

Van Durme, Benjamin and Lall, Ashwin. Streaming pointwise mutual information. *Advances in Neural Information Processing Systems*, 22:1892–1900, 2009.