N-gram based separation of Undemarcated Clickstreams into User Sequences for Frequent Pattern Mining[*]

[Project Report]

Aravind Sankar
CS11B033
aravindsankar28@gmail.com

Adit Krishnan
CS11B063
adit101293@gmail.com

ABSTRACT

This report addresses a precursor problem to the well addressed problem of frequent pattern mining of User clickstream patterns. It is assumed in most pattern mining literature that individual demarcated user wise sequences are readily available for usage. However with the advent of tracker blocking software and simplistic frontend devices(such as information kiosks connected to a single server), the only available data may be just chronological list of requests received at the backend. To the best of our knowledge, there is little or no related work in precisely this problem, Our goal is different from prediction based approaches primarily because we are trying to fit what we are seeing rather than trying to predict future actions based on what is a known user sequence.

It may not always be possible to determine where the requests originated from and hence this prevents them from being stitched in sequences. Our goal is to split the input stream into likely user sequences based on n-gram probabilities and put together these incoming requests into discretized sequences.

General Terms

Frequent Patterns

Keywords

Three click rule, n-gram models, Trie

1. INTRODUCTION

Since we are primarily attempting to stitch clicks, the web design concept that is of importance to us is the Three Click Rule. Most web designers attempt to design their webpages and portals in a manner that allows a user to reach a desired objective/ content page within three clicks after deciding that is what he wants. Even though there has been some criticism concerning this rule, in practice it has been noticed that for most users, clicks separated by more than 3 other clicks are less likely to be correlated. So we will be modelling the click occurences using trigrams.

[*]As part of project, CS6740, Jul - Nov 2014

N-gram based models are primarily used in Language modelling under the assumption that long distance dependencies do not exist in the sentences analyzed. This condition holds quite strongly for user interactions on the web as observed by many independent studies on browsing histories. Furthermore, there is a notion of context drift in web browsing which states that users jump from one topic to another frequently. We will be using this idea to build a trigram based model. There is also the problem of sparsity for higher n-grams. In practice, trigrams work out best for many problems. However, It is easy to extend the idea for quadgrams or pentagrams if the same is deemed necessary.

We are proposing a simple solution with an innovative data structure based on Tries, assuming however the availability of some past user data to train the system. The only purpose of training data is to find the trigrams that occur most frequently in user clickstreams. Though there is scope for a semi supervised approach where we learn as we go(using techniques such as bootstrapping), there is a risk of a few poor initial sequences leading to a bias being propagated. So we are sticking to the prior trigram behaviour probabilities learnt from training data alone.

2. DATA ANALYSIS

Dataset Used : MSNBC refined news portal usage sequences

Number of Sequences : 31000 (after removal of short and repetitive sequences)

Sequence contents : 17 distinct types of clicks. These could be individual links on their site or clusters of links leading to similar results.

Average Sequence Length : 13.336 clicks

Standard Deviation : 7.06 clicks

Distribution of lenghts : The minimum sequence length in the database is 9. The large chunk of sequences lie between lengths 9-15. This can be observed from the following plot.

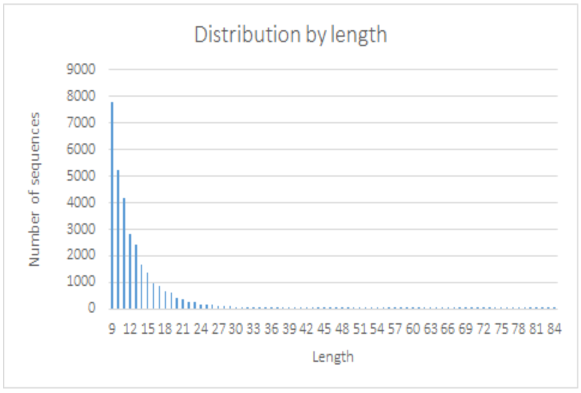

Figure 1: Length dist.

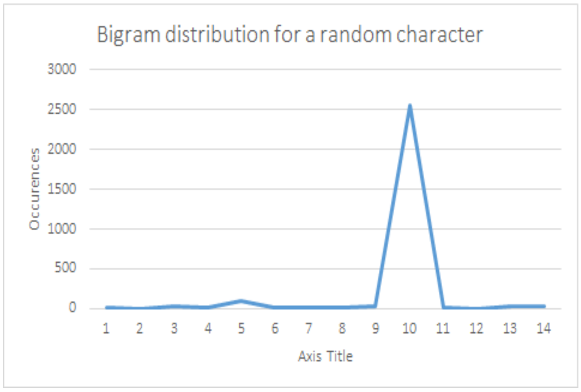

Figure 2: Bigram dist.

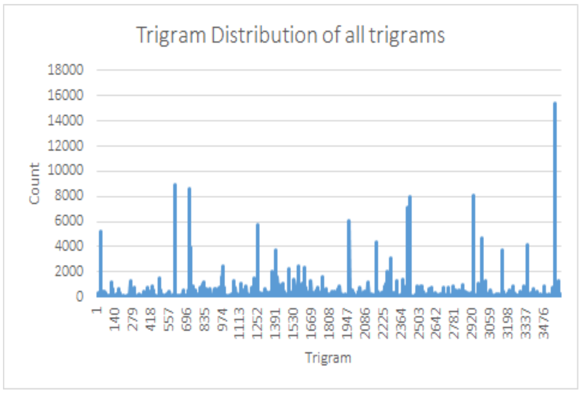

Figure 3: Overall Trigram dist.

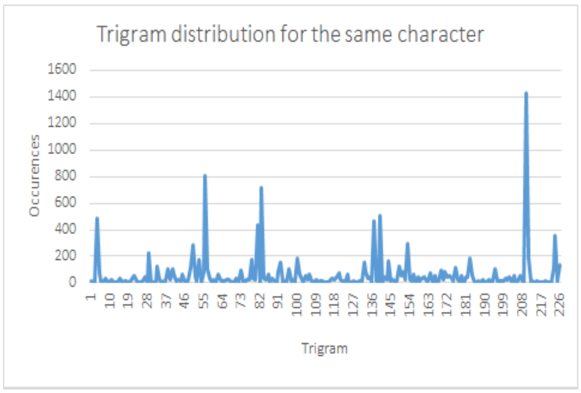

Figure 4: Trigram dist. for a specific character

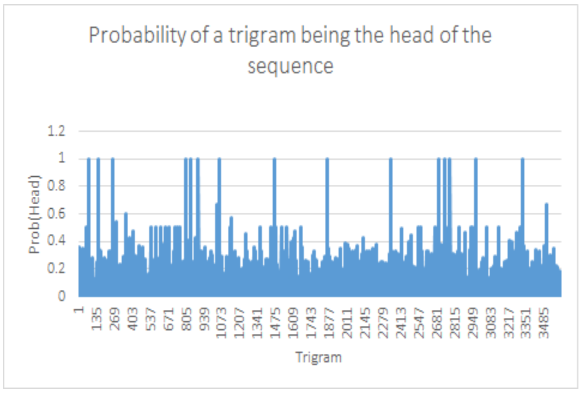

Figure 5: Trigram dist. for being the head of a sequence

Bigram and Trigram properties :

The above is the distribution of bigram counts for different bigrams corresponding to one particular character. This means, for instance if the character is A, all bigrams ending in A, i.e.AA, BA ... are all bigrams corresponding to it as per our convention. Clearly, there is a very strong association of the click to the preceding one. An overwhelming majority can be seen in one bigram among the 14 observed bigrams of this character. The same is also seen from the overall trigram distribution. Clearly, some trigrams have a very strong occurence count compared to others and hence possess good discriminating power.

We also observed if some certain trigrams were strong indicators of the start or end of a user session. The trigram wise probabilities of the same have been plotted for head. Tail gives a very similar plot. Note that these probabilites are calculated as,

$$\text{P(head}|\text{trigram} = \text{T}) = \frac{\texttt{Number of times T was head in training set}}{\texttt{Total occurences of T}}$$

$$\text{P(tail}|\text{trigram} = \text{T}) = \frac{\texttt{Number of times T was tail in training set}}{\texttt{Total occurences of T}}$$

For the same character we also observe trigrams. Again we see a few very clear distinct spikes. These are the highly probably trigrams that end in this click(character). This

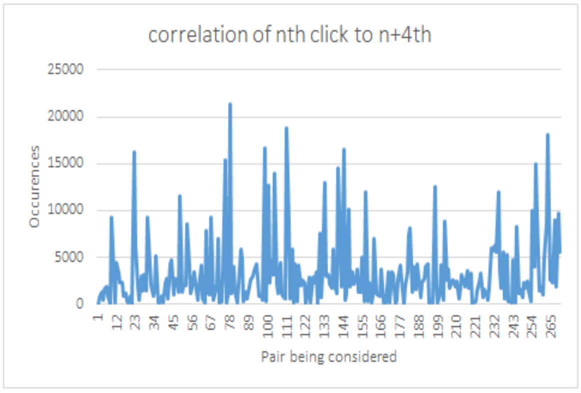

Figure 6: correlation of n-th and $(n+4)$-th clicks

proves that trigrams still possess fairly good discriminating power.

Higher n-grams :

For all pairs of clicks, we counted the number of times, they appeared at a separation of four. If there were some very distinct spikes here, we could say that it is worth using quadgrams, i.e there is some correlation between a click and the 4th click. However, there are not as many peaks and in some sense, the degree of randomness is more. This proves our initial assumption that beyond a separation of 2 or 3, the correlation between clicks rapidly diminishes.

3. SEQUENCE INDEXING STRUCTURE

There were a few factors that needed to be considered. Firstly, at any point in time, there are a large number of incomplete sequences that need to be filled in by characters that are coming in presently through the stream. There is a lot of repetition across sequences. Since each sequence has a lot of common trigrams and as we earlier pointed out, certain trigrams are more likely to fit into the starting and ending positions. Keeping this in mind, we need to avoid redundancy in storage.

The second important factor is that, when a new click arrives on the stream, it must be affixed to one of the existing sequences or it will start a new sequence. Whether it starts a new sequence or not depends on the number of active sequences already present. Clearly, due to the trigram model, the new click can only attach itself to sequences where the last 2 characters are such that when this is attached behind them, the trigram formed is a likely one. Thus we must be able to effortlessly index all sequences by their ending clicks.

With these 2 factors in mind, and drawing inspiration from the lexicographic tree structure(trie modifications) used in stream mining algorithms [1], we came up with a trie based structure holding incomplete sequences in reverse order. The following figure is useful to illustrate the same:

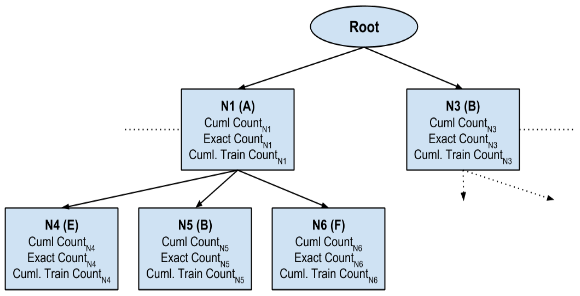

Figure 7: Index structure

As observed in the above figure, if we consider the set of clicks as Σ, the branching factor of this trie is $|\Sigma|$. The root is a dummy node. The children of the root correspond to the ending characters of other sequences. As we go downwards, we go towards the front end of the sequences. This is because the sequences are present in the tree in reversed order. The primary advantage of holding them in reversed order is that, for checking if a new character can fit into some sequence, we just need to see the ending trigram that will be formed by inserting this character and decide accordingly. Since the top nodes correspond to the ending characters, this is very easy to decide.

In addition to the character(or click) that a node corresponds to, it additionally stores 2 counts. The first is called the cumulative count. If we look at the above tree, node $N1$ is the node at which all sequences with the present final character being A are ending. The cumulative count of node $N1$ is the number of sequences that are either just A(i.e starting and ending at $N1$) or some $\sigma_1....\sigma_{t-1}$A, i.e. they end with A.
If we go one level down to $N4$, it has the character E. So all sequences that have their second last character as E and final character as A will pass through this node and are included in its cumulative count. The exact count on the other hand, is the subset of the above sequences that begin at that node. So for instance, the exact count of $N4$, is the number of EA's(the sequence EA).

When we insert a new incoming click to the end of an existing sequence, all we have to do is essentially decrement the counts along the path corresponding to that sequence presently, and then insert it in a similar fashion under the node corresponding to the new click added to the sequence, since that new click is now the ending click in that sequence. We will shortly describe the Growth, Insertion and Retiring of Sequences from this tree.

In addition to all the above details, we will also hold cumulative counts that were obtained from the training set in addition to the present cumulative counts(called cumulative train count) when initially growing the tree from the training data. This is explained in the following subsection.

3.1 Growth Algorithm(Training Phase)

During the training phase, we build the tree from the sequences that are available in the training set. The primary goal of building the tree using training sequences is to update the Cumulative Train Count parameter in the nodes. This is later used for tiebreaking during insertion when deciding which child of a node to choose(which is equivalent to choice of sequence).

For instace, suppose node $N4$ in **Figure 7** has a Cumulative Train Count of 1500, we know that 1500 sequences in the training set had their second last click as E and final click as A. On the other hand if node $N5$ has CTC of 20, we know it is much less likely to proceed in that direction from the ending character A.

Pseducode for Grow:

```
begin
    $Tree \longleftarrow newTreeNode()$
    $TrigramCounts \longleftarrow \emptyset$
    while !End of Train Set do
        $x \longleftarrow$ Train.getSeq()
        $node \longleftarrow$ Tree.root()
        for char $\in$ x.reverse() do
            TrigramCounts[char.trigram()] += 1
            if char.trigram() = x.start() then
                HeadCounts[char.trigram()] += 1
            end
            if char.trigram() = x.end() then
                TailCounts[char.trigram()] += 1
            end
            -epsilon may be intr.  in trigram if nec.
            if node.children(x) = NULL then
                node.children(x) $\longleftarrow$ new Treenode()
            end

            node $\longleftarrow$ node.children(x)
            node.cumulativeCountTrain += 1

        end
    end
    -Normalizing count for head and tail probabilities
    for trigram in HeadCounts do
        HeadCounts[trigram] = $TrigramCounts[trigram]$

    end
    for trigram in TailCounts do
        TailCounts[trigram] = $TrigramCounts[trigram]$

    end
end
```

Complexity

Since, it is just traversal of the tree using pointers, and the tree is typically small enough to be held in main memory, the time consumed by this phase is low
Operations $\propto$ O(total number of chars in Train Set)

The width of the tree depends on $|\Sigma|$, the number of distinct clicks. In practice, if this should get to be too large, one could always group clicks into logical clusters and perform the analysis. For any modest value, the size of the tree is not too large, since only some of the possible paths will be seen.

3.2 Insertion Algorithm(Live Phase)

During the live phase, clicks are incoming from the user stream. We have to build sequences for users out of these. We have an idea of the number of active users on the site. As and when clicks are received we can either begin a new sequence with these clicks or add them to an existing sequence. We use a greedy approach to decide what to do. For each character, say A, it has a preference among its trigrams. The most common trigrams ending in A are the most preferred ones to which A would like to attach itself. So in a greedy manner we check them in order of decreasing occurence if see if there is a sequence presently in the tree to which this can be added to form the preferred trigram at the end of that sequence.

In this phase, we will use the other two variables in the nodes as seen in **Figure 7**. These are the Cumulative count and Exact Count variables present inside them. The Cumulative Count here has the exact same meaning as the Cumulative Count Train did, except that it is for the presently seen sequences and will not include the training sequences. Exact count on the other hand is a simple variable. Suppose in node $N4$, we have a Cumulative Count of 5 and an Exact Count of 2, this would mean that among the sequences that we are presently constructing, 2 of the incomplete sequences are the sequence EA itself, and there are 3 other sequences that have EA at their end but are not EA, i.e.They are longer sequences ending in EA.

Pseducode for Insert:

```
begin
    while True do
        $C \longleftarrow inputclick()$
        $Trigrams(C) \longleftarrow sortByCount(Trigrams(C))$
        while !End of Trigrams(C)
        do
            Tree.attemptInsert(Trigrams(C).next())
            if (success) then
                break
            end

        end
    end
end
```

Tree Attempt Insert Procedure(trigram is provided as argument, as seen above)

- The inputed trigram is of the form, trigram[0], trigram[1], trigram[2] where trigram[2] is the click that is presently inputed, which we want to add to an existing incomplete sequence.

- So the sequence candidates to which we can add it to form this trigram at the end, are those presently ending in trigram[1]. So we check if the cumulative count of Tree.root.children(trigram[1]) is non zero. If this is met we then check if
Tree.root.children(trigram[1]).children(trigram[0]),
has a non zero cumulative count. If these two things are met, there is atleast one incomplete sequence to which we can add trigram[2].

- Now we need to decide among the sequences ending this way, which one specifically we want to add trigram[2] into. For this we must traverse further down the tree, and also decide where to stop. Each time we traverse down by one level, we are essentially moving towards the front of the sequence that we want to add to. When traversing down, we must choose which character we would like to add to the sequence(in reverse ofcourse since we are moving from back to front).

- This is done greedily, by using a combination score of 2 things, the probability of the trigram being formed in the sequence by moving in one branch vs another and the Cumulative Count Train (if available) of the branches. At each level we choose the branch which gives the best score [Presently we are using a linear combination of the two with weights tuned from the data].

- We will stop when two conditions are met. First we get a good head trigram for the sequence. Second, there must be some exisitng incomplete sequence which begins at this point, i.e. The Exact Count of the stopping node must be non zero. To a smaller extent we will also consider the length of the sequence as a factor. For this we factor in the cdf of sequences being a particular length, as obtained from the distribution of lenghts, into the stopping criteria.

- Once we have stopped, we will detach the sequence formed from the present branch of the tree, by decrementing Cumulative Count along the path followed and both Cumulative and Exact Count for the final node.

- **Retiring:** Now once we have detached the sequence, we will decide whether to add it back into the tree for potential future expansion, or to retire it as a completed sequence. This is done using a simple thresholding. We take the product of the cdf of the sequence's length and the tail probability of the ending trigram. If this product is above a certain threshold, then most likely the sequence will not grow further. Again we determined this threshold based on the data, by running a simple predictor on the training data and choosing a value for which right sequences were mostly predicted right and randomly generated SubSequences were predicted wrong.

- If we have to add it back to the tree, simply go to the trigram[2] branch from the root and increment cumulative counts moving along the path corresponding to the sequence(add children if they do not already exist).

Complexity

For each insertion, in the worst case, we would need to look through all of its trigrams (O($|\Sigma|^2$). However note that when it fails for a trigram, there is only a single computation done near the root of the tree. When it is successful, it traverses downwards. The maximum depth of the tree is O(L), the maximum length of a sequence. At each level, we need to perform atmost O($|\Sigma|$) comparisions to determine which branch to take. Finally at each level we must also check whether to stop or continue and to reinsert, a max of L traversal steps.

Thus, total computations = O($|\Sigma|^2$ + L.$|\Sigma|$ + L) $\tilde{O}(|\Sigma|^2)$ per insertion.

This is a very small complexity and is independent of the number of active sequences in that Tree or the total size of the Tree itself. Thus the algorithm will scale very will with increases in Number of incoming sequences as opposed to any other data structure which does not exploit the inherent redundancy in sequences.

4. RESULTS

The results obtained were better than that of the baseline algorithm in some respects, However certain truly random subsequences that do not concur with the trigram model, were not captured by our heuristics. On the whole many intricate correlated sequences were found very well by our algorithm as we will illustrate using results.

Evaluation Metric Used : Ratio of Weighted information content of extracted frequent sequences.

For every frequent subsequence that was obtained from the test set in the original data, we try to find the best match that was extracted as a frequent subsequence from the sequences generated by us.

Info content of frequent subsequence originally(say subseq X) = -log(support/total original frequent subseqs)
Info content of our subsequence(say subseq Y) = -log(support/total new frequent subseqs)

We will find the best matching Y for every X so that Y is also part of X. The best matching Y for a given X is that which has the highest product:

Similarity(X,Y) * InfoContent(Y)

For each X, we find these best products among Y's and add them up. Their sum is the measure of performance on our generated sequences. We will compare this with the Info content sum over all X's in the originial set of frequent sequences. These numbers themselves dont mean anything. We are interested in their ratio, which is what we will be

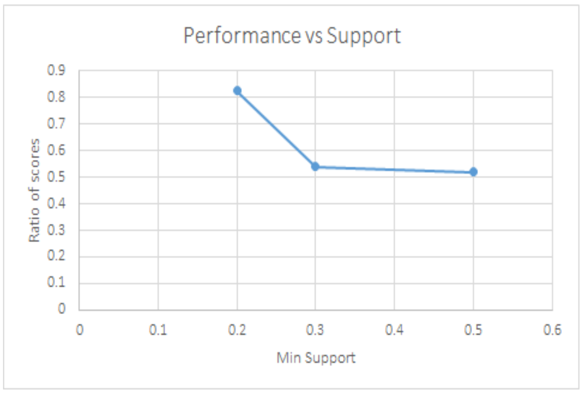

Figure 8: performance vs Min support for VMSP extraction

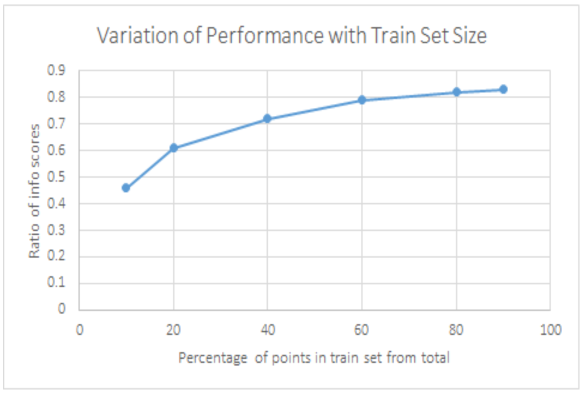

Figure 9: performance vs size of train set

using to get an idea of the performance.

There are 2 approximations that we did here.

- Firstly, we took the Similarity(X,Y) as a simple String similarity ratio which may not be a good measure here.
- Secondly, we use the InfoContent(Y) among the frequent subsequences of our generated sequences, whereas in reality, both X and Y should be evaluated on some common ground with respect to InfoContent. However, we felt this would be good enough to give us a decent idea of performance, atleast for the sake of comparisions.

Results obtained vs Min Support for frequent sequences: For lower Min support, we get lots of intricate sequences, where our algorithm seems to perform quite well. However we did notice sudden spikes in the general smooth trend at a few places due to the appearence of certain sets of subsequences that seem fairly random and uninformative.

- There is a certain sequence 1-1-1 which enters the set of frequent subsequences at threshold around 0.4. A few other such sequences creep in merely due to noise. This particular subsequence appears due to the fact that 1 is the most common click in the data, but also the most random since it does not form good trigrams.
- For one run at threshold 0.1 or so, we found the sequence 4-7-1 successfully detected from our generated sequences. This corresponds to a trigram that has a good probability of occurence. For many other such sequences we found good detection.
- Thus our model will perform poorly when there is noisy data and when the website itself forces certain constraints that cause the trigram rule to be violated (in which case one could always switch to higher grams)

The final test that we carried out was versus the number of training sequences. The most important aspect here is that the trigram probability computation which requires a fairly representative unbiased training set. At around 33% onwards we found the performance to be fairly steady. Which implies that for this dataset, around 11,000 sequences were required to learn the trigram probabilities.

Running time :(core i3 system with 4GB memory)

- The average time taken to insert one character into the tree in the live phase, is less than 0.002 seconds, which gives a throughput of over 500 clicks per second.
- Preprocessing involved in the Growth phase takes around 1.5 seconds for a training set size of 20000 sequences.

5. CONCLUSIONS

Our primary focus here is on the running time which is quite fast inspite of the size of the tree. The efficient storage of sequences and the quick pruning of trigrams using the reversed trigram idea are quite effective here.

On the whole, we feel rather than clicks alone, if we could incorporate additional user information, site information or time stamps, we could produce a much richer set of trigrams. We were restricted in our approach due to the lack of richness of data(since only clicks were avaliable, not even timestamps). In the future, we can also think about looking back at the previously inserted characters in every active sequence and seeing if replacing them with the new character actually leads to a better sequence. This would lead to the correction of errors introduced in the previous insertions using Backtrack, however it may prove costly if not done correctly.

6. REFERENCES

[1] Mendes, Luiz F., Bolin Ding, and Jiawei Han. ”Stream sequential pattern mining with precise error bounds.” Data Mining, 2008. ICDM’08. Eighth IEEE International Conference on. IEEE, 2008.

[2] Fournier-Viger, Philippe, et al. ”VMSP: Efficient Vertical Mining of Maximal Sequential Patterns.” Advances in Artificial Intelligence. Springer International Publishing, 2014. 83-94.

[3] Testing the Three-Click Rule by Joshua Porter - http://www.uie.com/articles/three_click_rule/