User Modeling for a Personal Assistant

Ramanathan Guha* Vineet Gupta Vivek Raghunathan Ramakrishnan Srikant

Google
Mountain View, CA, USA
{guha,vineet,vraghunathan,srikant}@google.com

ABSTRACT

We present a user modeling system that serves as the foundation of a personal assistant. The system ingests web search history for signed-in users, and identifies coherent contexts that correspond to tasks, interests, and habits. Unlike past work which focused on either in-session tasks or tasks over a few days, we look at several months of history in order to identify not just short-term tasks, but also long-term interests and habits. The features we use for identifying coherent contexts yield substantially higher precision and recall than past work. We also present an algorithm for identifying contexts that is 8 to 30 times faster than previous algorithms. The user modeling system has been deployed in production. It runs over hundreds of millions of users, and updates the models with a 10-minute latency. The contexts identified by the system serve as the foundation for generating recommendations in Google Now.

1. INTRODUCTION

During the last two decades, web search has become the most widely used method for answering information needs. Over this period, search engines have become increasingly sophisticated at finding and ranking search results. While this model for answering information needs has been extremely successful, there are some limitations:

- Mobile device usage has exploded in the last five years. Typing a search query on a smartphone is very slow compared to a desktop. A system that automatically shows the right information at the right time would provide a dramatically better user experience on the phone.
- For topics the user is interested in, the user needs to repeatedly query the search engine to determine whether there is interesting fresh content. If such content appears only occasionally, the user is destined to either be disappointed with most searches not returning anything new, or miss interesting fresh content with high probability.
- Search engines are only mildly personalized. Interactions with search engines are often reminiscent of the movie "50 First Dates", in that the search engine appears to have forgotten all past interactions.

The last three years have seen the introduction of a new category of mobile personal assistants, including Google Now [1] and Cortana [2]. These assistants share the goal of showing the user the right information at the right time, without the user querying for information. The primary motivation of this paper is to enable such personal assistants to address the problems discussed above:

- **Contextual assistance**: The personal assistant should be able to detect a change in the user context, and make useful recommendations for that context. For example, if a user is in a city away from home, the assistant might show a personalized list of nearby restaurants and their reviews, without the user ever typing a query.
- **Interest updates**: Users are often interested in certain topics, like a particular sports team, a celebrity, or a favorite TV show. Personal assistants save users the trouble of finding new information by automatically alerting the user to any new piece of information about their favorite topics.
- **Fully personalized**: The personal assistant utilizes the user model to fully personalize the experience. For example, when suggesting restaurants, in addition to the spatio-temporal context, the assistant personalizes the suggestions based on the user's cuisine preferences and price sensitivity.

We emphasize that such personal assistants (or discovery engines) complement traditional search engines, and while they may substantially reduce the need to use search engines on a smartphone, are not replacements for a search engine.

The foundation for such a personal assistant is a user modeling system that, given a sequence of user actions, identifies coherent **contexts**. A context is a set of user actions corresponding to a single information need. Contexts include:

- **Tasks**: Tasks typically have a beginning and an end, e.g., buying a camera or planning a trip to Paris. Tasks can span multiple sessions, and some long-running tasks, such as planning a wedding, may span weeks or months.

*Authors are listed alphabetically.

WSDM'15, February 2–6, 2015, Shanghai, China.

http://dx.doi.org/10.1145/2684822.2685309.

- **Interests**: Interests may span months or years, and often do not have an end. Examples include following sports teams, celebrities, or TV shows.
- **Habits**: These are actions that users take on a regular basis, e.g., reading a favorite blog or news site, checking stock prices, or checking traffic in the daily commute.

These contexts are annotated with context types, entities, and names.

We have built a system, called **Taba**, comprised of two parts:

1. An user modeling system that builds the user model described above. The system scales to hundreds of millions of users, and updates the user model within 10 minutes of a new user action.
2. A content recommendation system that does collaborative filtering over contexts and users, and also predicts how interested the user will be in recommendations for the context.

In this paper, we focus on the user modeling system, and motivate the user model by describing how the recommendation system uses the contexts. Both these systems have been deployed as part of Google Now, serving tens of millions of recommendations daily.

Paper Outline.

We discuss related work in Section 2. In Section 3, we describe the features and algorithms for identifying tasks, interests and habits from a user's history. We evaluate our user model in Section 4. In Section 5, we describe how the user model serves as the foundation for prediction and recommendation systems in Google Now. We conclude in Section 6.

2. RELATED WORK

Early work on grouping search queries (e.g., [21]) focused on sessions, defined purely by the temporal gap between queries. Spink et al. [22] found that a large majority of multiple-query sessions included multiple topics. Jones et al. [12] showed that using query reformulation and query words to identify topics yielded substantially higher accuracy than only using timeouts.

Subsequent work in this area can be split into two groups: papers that focus on in-session tasks, and those that identify cross-session tasks.

2.1 In-Session Tasks

Lucchese et al. [17] proposed techniques for identifying "task-based sessions": sets of possibly non-contiguous queries within a session that correspond to a user task. They recognized that standard clustering algorithms are too expensive, and proposed a new clustering algorithm QC_{HTC}, which avoids computing the full similarity graph by just considering a cluster to be represented by the first and the last queries in the session.

Li et al. [16] also consider in-session tasks. They use query words, query co-occurrence, and the temporal sequence of queries as their main signals. Their learning algorithm is quadratic, but they have a linear approximation which works well in practice. Hua et al. [10] showed the importance of semantic features for in-session task identification.

User behavior within a session (typically 30 minutes) is quite different from user behavior across months of history. In-session tasks tend to be uniform and small, while long-term tasks and interests tend to be diverse and large. Hence intuitions that work well for in-session tasks do not always scale to identifying coherent contexts across months of history. For example, representing contexts by the first and last queries is reasonable for in-session tasks, but cannot capture a context with hundreds of queries.

2.2 Cross-Session Tasks

Several papers have recently addressed the considerably more challenging problem of identifying longer range tasks across sessions. Kotov et al. [14] formulate the problem of identifying cross-session tasks as follows: given a query, identify all related queries from previous sessions that are part of the same task. Their main focus is trying to predict if the task will be continued later, so they restrict themselves to primarily syntactic features such as number of words the queries have in common, edit distance between queries and the time between queries. The authors continued their research on task continuation in Agichtein et al. [3], and added semantic features such as ODP categories, and demonstrated that automated systems can significantly outperform human raters on predicting task continuation.

Lucchese et al. [18, 19] have extended their work above to cluster their already computed in-session tasks by using clustering algorithms which produce a specified number of clusters. They use average pairwise similarity to determine the similarity between tasks. Their primary focus is clustering in-session tasks across users, but they do use their algorithm to produce cross-session task clusters for individual users.

Wang et al. [23] propose an algorithm for identifying cross-session search tasks using a variety of signals. They built a training set by completely annotating 5 days of query logs for 1436 users, and learned a latent structural SVM classifier to compute the similarity between pairs of queries. Then they use a bestlink clustering algorithm [11] to cluster queries — each query is clustered with the most similar query that occurred before it, as long as their similarity exceeded a threshold.

2.3 Our Contributions

There are three main differences between our work and prior work on in-session and cross-session tasks.

- We focus on user history over many months, often with thousands of queries and clicks. This lets us identify long-term interests and habits in addition to identifying short-term tasks — identifying such contexts over many months (versus short time spans) is critical for the user model to be useful in a personal assistant (see Section 5).
- We present a set of features that yield better precision/recall than past work [23] while simultaneously being much faster to compute.
- Working over longer time spans means that $O(n^2)$ algorithms do not scale. We present a near-linear segmentation algorithm that is much faster than prior work.

Finally, this is a production system running continuously over hundreds of millions of users with a 10-minute turnaround

time for incorporating new user actions into the model. This has implications for the design of features and algorithms, as we will see in the rest of the paper.

3. USER MODELING

3.1 Problem Statement

The input to the user modeling system is a **sequence of observations** from a single user. In this paper, an *observation* is a query together with its associated web results and clicks. Conceptually, an observation could also be a video watch, or a URL visited in a browser.

The output is a **set of contexts**, where a context is a sequence of observations that constitutes a single information need.

In general, the definition of a single information need is ambiguous, and in fact it can differ among users as well. For a researcher in information retrieval, the queries "graph clustering algorithms" and "synonyms in web search ranking" may be part of different contexts about different papers. For a beginning undergraduate, they would be part of the same context, "Information Retrieval Course".

To determine whether two observations are part of the same information need, we asked human raters: "Are these part of the same task or interest?" We pick a random sample of pairs of observations from a random sample of users, and the ratings on this sample serve as the ground truth. The goal of our algorithm is to partition the sequence of observations into a set of contexts, and we evaluate the precision and recall of the algorithm against the sampled ratings.

We break the context inference problem into two subproblems:

1. Given two candidate contexts, should we merge them into a single context? We formulated this as supervised classification, with the additional requirement that the score returned by the classifier should also serve as a similarity score (Section 3.2).
2. Using the classifier to make decisions, we wish to segment the set of input observations into coherent contexts (Section 3.5).

This approach is similar to supervised clustering, e.g., [9].

Once we have identified contexts, we add semantic annotations like context types and attribute entities (Section 3.6).

3.2 Classification

Given two contexts C_1 and C_2, we need a similarity function that lets us decide whether these two contexts should be merged into a single context. In addition, we would like the function to return a score that reflects the degree of similarity between the contexts.

Compared to prior work, there are three differences in our similarity computation:

1. Given two contexts with m and n observations respectively, we compute the similarity score in $O(m+n)$ time, i.e., linear in the size of the contexts. A straightforward pair-wise similarity computation would need $O(mn)$ time, which would be too slow for our application.
2. Instead of weighting features by inverse document frequency or inverse query frequency, we weight based on the probability that two observations with that feature are merged into the same context (Section 3.3).
3. We incorporate the degree of confidence in a feature into the similarity calculation (Section 3.4).

3.2.1 Classifier

We use the term **feature dimension** to denote a set of homogeneous features, e.g., the set of query words is a feature dimension, or the set of Freebase entities is a feature dimension. Each word or entity will then be called a **feature**. Similarity metrics such as cosine similarity make sense within a feature dimension, but not across feature dimensions, as features are not comparable across dimensions.

We use a linear classifier for our similarity function:

$$\text{Similarity}(C_1, C_2) = w_0 + \sum_{i=1}^{n} w_i v_i(C_1, C_2) \qquad (1)$$

where w_i is the weight of the i^{th} feature dimension, and $v_i(C_1, C_2) \in [-1, 1]$ is the score (value of the similarity metric) of the i^{th} feature dimension for the contexts C_1 and C_2. We consider two contexts to be similar if their similarity score is greater than zero, hence w_0 serves as an (optional) offset. Given a training dataset consisting of pairs of contexts, with the scores of each feature dimension, we use an off-the-shelf SVM classifier to learn the weights w_i.

3.2.2 Feature Dimensions

Figure 1 summarizes the feature dimensions that we used, along with their similarity metrics. Note that all of these are applicable to pairs of contexts, not just pairs of observations.

The Location and Temporal dimensions require custom similarity functions (discussed below). The other feature dimensions all use one of the following similarity metrics:

- *Cosine* is the standard cosine similarity, used when the feature dimension is a weighted vector of features.
- *ScaledCosine* is a variant of cosine similarity that takes into account missing features (e.g., some queries or clicks may not map to any entities), or low confidence in inferred features. ScaledCosine is discussed in Section 3.4.
- *MaxFraction* computes for each context the fraction of observations that satisfy some property (e.g., matching an observation in the other context). It then takes the maximum of these two fractions as the similarity.
- *NormIntersection* is a symmetric version of Jaccard containment [4]. NormIntersection is defined for a pair of sets A, B as

$$\text{NormIntersection}(A, B) = \frac{|A \cap B|}{\min(|A|, |B|)}$$

  This function does not satisfy $\text{NormIntersection}(A, B) = 1 \Rightarrow A = B$, but we prefer it to Jaccard similarity as $A \subseteq B \Rightarrow \text{NormIntersection}(A, B) = 1$, so we would merge A, B as expected.

We typically use ScaledCosine for feature dimensions where we expect missing features or variance in the confidence in features. For feature dimensions where each feature is approximately equally predictive of similarity (like query or URL), we treat the features as a set and use NormIntersection or MaxFraction for similarity.

Feature Dimension	Similarity Function	Inputs to the similarity function
QueryWords	Cosine	Stemmed query words in each context, weighted by frequency and PVFW
QueryRefinements	ScaledCosine	Stemmed words in the refinements of queries in each context
SameWebResults	NormIntersection	Set of top 10 web results for the queries in each context
SameQuery	NormIntersection	Set of queries in each context
Spelling	MaxFraction	Fraction of queries in C_1 that are mis-spellings of queries in C_2, and vice versa
SubQuery	MaxFraction	Fraction of queries in C_1 that are sub-queries of queries in C_2, and vice versa
ClickWords	Cosine	Stemmed words in the titles of clicked URL in each context
QueryClickWords	Cosine	Stemmed query words of C_1 with stemmed click title words C_2, and vice versa
SameClick	NormIntersection	Set of clicked URLs in each context
URLDomain	ScaledCosine	URL domains (for clicks and web results), weighted by frequency and PVFW
ContextType	ScaledCosine	Context types (see Section 3.6.1) with weights
Category	ScaledCosine	Categories for each context (using an off-the-shelf classifier), with weights
Forums	ScaledCosine	Forum results for queries in each context, weighted by frequency
Entity	ScaledCosine.	Weighted vector of Freebase entities extracted from queries and clicks
Location	See Section 3.2.2	Locations referenced in queries and clicks
Session	MaxFraction	Fraction of queries in C_1 in the same session as a query in C_2, and vice versa
Temporal	See Section 3.2.2	Temporal distance between contexts

Figure 1: Features

Query refinements are other queries that frequently co-occur with the given query in search sessions of many users. The other feature dimensions based on queries and clicks are self-explanatory.

The next two feature dimensions, *ContextType* and *Category*, map queries and clicks into taxonomies. *Category* uses an off-the-shelf classifier with around 1500 categories. However, there are many cases where observations that map to different categories would be considered part of the same context, while observations that map to the same category would be considered part of different contexts. *ContextType* is described in more detail in Section 3.6.1, and has the property that if two observations map to different context types, they will not be considered part of the same context. The third taxonomy-like feature is *Forums*, which uses the set of forums that we get when search results for the query are restricted to forums. Forums are especially helpful for tail queries, where they capture fine-grained user interests.

For the *Entity* feature dimension, we identify and score freebase entities for each query and URL. We start with off-the-shelf query-to-entity and document-to-entity extractors. We then do additional disambiguation by using click entities (and if present, knowledge panel entities) to disambiguate query entities, and query entities to disambiguate click entities.

For each observation, we normalize the weight of the features within a feature dimension to be at most 1 + the number of clicks. Intuitively, we give the query and each click equal weight, and an observation that has more clicks has more weight. We aggregate the features from observations to contexts using either set union or by adding the feature vectors. For each feature dimension, we can compute similarity of two contexts with m and n observations respectively in $O(m+n)$ time.

For *Session* and *Temporal*, we compute, for each observation, the time gap to the closest observation in the other context in $O(m \log n + n \log m)$ time. For *Session*, we then use the fraction of observations that occur in the same session as an observation in the other context. For *Temporal*, we compute the average, over all observations, of the number of days to the closest observation in the other context.

Finally, *Location* is the only feature where we do use $O(mn)$ time. We compute the fraction of locations in the context that are close to some location in the other context. (Computing the average lat-long is not very meaningful, *e.g.*, consider a context whose locations include San Francisco and California.) Fortunately the number of locations is much smaller than other features, hence the overall execution time continues to be linear in the size of contexts.

3.3 Predictive Value Feature Weighting (PVFW)

A standard approach in information retrieval is to weight features by inverse document frequency or inverse query frequency. We found that such weighting was useful for very popular words, entities or domains, but not useful beyond the head. For example, consider the queries "Ecco shoes" and "Rockport shoes". "Shoes" is a common word with a small IDF, while "Ecco" and "Rockport" have large IDFs, so the cosine similarity of the query words weighted by IDF is small. However, the word "shoes" is a good indicator that these queries are part of the same context, so we would like "shoes" to have the higher weight, not the brands.

In order to accomplish this, we directly measure the predictive value of a feature. Let O_i, O_j be a random pair of observations for a random user. Then we define the predictive value weight w_f for a feature f as:

$$\begin{aligned} w_f &= \Pr(O_i, O_j \text{ in same context} \mid O_i, O_j \text{ have feature } f) \\ &= \frac{\sum_{u \in U} |P_{fc}(u)|}{\sum_{u \in U} |P_f(u)|} \end{aligned}$$

where U is the set of users, and for a user u, $P_f(u)$ is the set of all pairs of observations that have the feature f, and $P_{fc}(u)$ the set of all pairs of observations that both have the feature f and are in the same context at the end of segmentation.

The intuition is similar to co-training [5, 7]: we leverage information from all the other features to determine the usefulness of this feature. There is a circularity here, that

the segmenter uses the classifier that in turn uses the segmenter to determine the feature weights. In practice, we built the initial version of the segmenter without feature weights. Once we had contexts for all users, we computed feature weights, re-ran the segmenter, and continue to update the feature weights periodically.

3.4 Incorporating Confidence: ScaledCosine

We motivate ScaledCosine with the following examples:

- When we infer entities from queries or URLs, we often get entities with low confidence. For the query "WSDM 2015", the only entity is "Web Services Distributed Management", with low confidence. Simply computing the cosine similarity between the set of entities in this query and a different query will not capture that this is a low confidence inference.
- For a click on a Wikipedia article, the PVFW weight we compute for the domain `wikipedia.org` is less than 0.1, i.e., the probability that two wikipedia clicks end up in the same context is less than 10%. `insideevs.com` is however a very specialized website, and gets a score of 0.83. But a simple cosine similarity cannot distinguish between a pair of clicks to `wikipedia.org` and a pair of clicks to `insideevs.com`: both of these pairs will get a cosine similarity of 1.

Given a pair of contexts, let

- c be the cosine similarity,
- s_1 (s_2) be the sum of the weights of the features in the first (second) context, and
- m_1 (m_2) be the maximum possible value of s_1 (s_2).

A natural approach would be to try multiplying c by $\frac{s_1}{m_1} \times \frac{s_2}{m_2}$, so that the similarity is low when we have less confidence. However, this conflates two very different cases:

- The similarity value is low because the contexts are very dissimilar.
- The similarity value is low because we do not have much data for this feature dimension.

Hence we first determine a **neutral point** t, so that contexts with cosine similarity $c \ll t$ are considered similar, while contexts with cosine similarity $c \gg t$ are considered dissimilar.[1] We define ScaledCosine for a feature dimension as:

$$\textbf{ScaledCosine} = (c - t) \times \frac{s_1}{m_1} \times \frac{s_2}{m_2}$$

Intuitively, if there is no uncertainty or missing features, $s_1 \approx m_1$, $s_2 \approx m_2$, and the ScaledCosine is simply $c - t$. Since the threshold t is offset in the classifier, the ScaledCosine behaves exactly like the cosine similarity.

If there is very little data for one or both contexts, the ScaledCosine will be close to zero because we scale $c - t$. If the contexts are dissimilar and there is sufficient data, the ScaledCosine will be negative, thus distinguishing the two cases discussed above.

If there is sufficient data and $c \approx t$, the ScaledCosine will be close to zero. However, conflating this case with that of missing data is not a problem because, in both cases, there is no strong signal from this feature dimension as to whether or not the two contexts should be merged.

[1]We can determine t by training a SVM for a feature dimension. Then $t = -w_0$.

1. Initialize each observation as a candidate context.
2. While Similarity(closest pair of candidates) > 0:
   a. Merge the two candidate contexts that are most similar.
   b. Recompute similarity score between the new context and all other candidates.

Figure 2: Hierarchical Agglomerative Clustering

1. Initialize each observation as a candidate context.
2. Within each session, use HAC to merge contexts.
3. For threshold $t_i \in [t_1, t_2, ..., t_k]$:
   a. Find all pairs of candidate contexts that share a feature, together with a light-weight similarity score.
   b. Sort candidate pairs by the light-weight similarity score.
   c. For each candidate pair (c_p, c_q):
      If c_p was already merged into a context c_r, $c_p := c_r$.
      If c_q was already merged into a context c_s, $c_q := c_s$.
      If $c_p = c_q$, skip this pair. Else:
      Compute Similarity(c_p, c_q), and merge if score $> t_i$.
      If the last N candidate pairs had scores $< t_i$, break.

Figure 3: Near-Linear Agglomerative Clustering (NLAC)

3.5 Segmentation

Prior work in user modeling used best-link or average-link, both of which are $O(n^2)$, where n is the number of observations for the user. Conceptually, we would argue that hierarchical agglomerative clustering (HAC) is a good fit, on the grounds that by merging the most similar contexts first, we defer the ambiguous cases to when we have more data. We summarize the HAC algorithm in Figure 2. However, a straightforward implementation of HAC is $O(n^2 \log n)$, which is not scalable when we wish to identify long-term interests that span months.

We now describe a near-linear approximation to HAC that gives us similar clustering quality for our domain. The key insights are to prune the search space, and use lightweight similarity scores to identify candidate pairs with high similarity. Figure 3 sketches our NLAC algorithm.

- In Step 2, we run HAC on the set of observations in each search session. Since sessions are typically small (and we split large sessions), this allows the algorithm to quickly reduce the number of contexts. While useful, this step is not critical to performance.
- In Step 3a, we compute an approximation to the similarity function, similar to canopy clustering [20]. We build an inverted index over the features, and use that to compute, in near linear time, all pairs of contexts that share a feature, as well as a light-weight similarity score for each pair of contexts. (This is similar to using LSH [13], except that we hash each feature.) Figure 4 shows the algorithm to compute cosine similarity for one feature dimension.[2] We repeat this algorithm for

[2]Note that line 6 in Figure 4 can lead to quadratic execution time for features that occur in many contexts. Hence, we sample the set of pairs if the list is long. If two contexts are indeed similar, we expect that some other (more specific) feature will result in them being added to the candidate pairs. In addition, even for high-frequency features, the lists get shorter in later passes as we merge contexts.

For each context c_i: (1)
  Compute L, the L_2-norm for feature dimension F. (2)
  For each feature term $f_j \in F$ with weight w_j in c_i: (3)
    Add $(c_i, w_j/L)$ to inverted index list for f_j. (4)

For each feature term $f_j \in F$ in the inverted index: (5)
  For each pair of values (c_k, w_k) and (c_m, w_m): (6)
    $W(c_k, c_m) := W(c_k, c_m) + w_k w_m$ (7)

// Now $W(c_k, c_m) = \text{cosine}(c_k, c_m)$ for feature dimension
// F for every pair of contexts c_k, c_m.

Figure 4: Computing Candidate Pairs and Cosine Similarity (for one feature dimension)

each feature dimension, and then, take the weighted sum of the cosines (Equation 1) to get the light-weight similarity score.

- Steps 3b and 3c are critical to performance. Sorting by the light-weight similarity score and using early termination at the end of step 3c allows us to limit the number of full similarity computations.
- In Step 3, we set $t_1 > t_2 > ... > t_k$. The intuition is that while the lightweight scores serve as a reasonable proxy for the full score, they do diverge, especially when c_p and c_q have been merged into other contexts. The descending thresholds t_i ensure that, like HAC, we do the high confidence merges first, and defer the borderline cases. For our dataset, we found that $k = 3$ gave us the same quality as higher values of k.

3.6 Semantic annotations

Once we have contexts, we annotate them with structured metadata. We describe some of these annotations here.

3.6.1 Context Types

Context types are a classification of contexts into popular user information needs at a coarse level, for example, *Travel*, *Soccer* or *Health*. Context types serve several purposes:

- We prune out contexts that map to types like *Health* or other sensitive subjects.
- We identify key attributes based on the context type, e.g., hotels and attractions for *Travel*, or teams and players for *Soccer*.
- Context types let us customize the experience for specific types, e.g., travel.
- Context types are also used by the classifier as a signal.

We bootstrap the context type classifier with a set of hand-built matchers:

- Category nodes from an off-the-shelf classifier with around 1500 categories, for both queries and URLs.
- Types of the freebase entities in the queries and URLs.
- Words in the query or URL title.
- URL patterns.

For popular (head) context types, the above matchers give us sufficient precision and recall. As we expand the set of types, we plan to use matchers to bootstrap the classifier, and use co-training [5, 7] to increase recall.

3.6.2 Attribute Entities

Attributes are freebase types that are especially appropriate for a given context type, e.g., for *Travel*, attributes include location, hotels and restaurants. Attribute entities are the entities in the context whose freebase type matches the attribute.

The primary benefit of using attribute entities rather than all the entities in the context is that attribute entities are usually much higher quality. Hence many of the use cases in Section 5 rely on attribute entities. These entities are also used in context type classification, e.g., in order to label a context with context type Books, we require the identification of an entity of type author or book title.

3.7 Scaling it up

Even with fast clustering algorithms, deploying a user modeling system for a personal assistant in production involves significant scaling challenges. The first observation is that computing features like categories, forums and entities on-the-fly as we process queries and clicks is prohibitively expensive. Hence we pre-compute such expensive features for a few billion frequent clicks and queries, and look these up at runtime. For tail queries and clicks, we fall back to cheaper features such as query words and click titles.

Consider a batch implementation that partitions input data by user and parallelizes computation using Mapreduce [8]. If we wish to run over a hundred million users, with around 5 seconds per user (Section 4.3.4), that works out to 6000 days of CPU time for just segmentation — without counting the load on the back-end for feature lookups. An obvious optimization is to skip over users with no new data since the last run. However, active users tend to have many more observations than inactive users, and hence this optimization (while helpful) does not result in a huge drop in computation time. Clearly, any system that relies on repeated batch runs will result in large latency between new observations coming in and user model updates.

We rely on two solutions to scale our system. First, we have an incremental version of the segmentation algorithm. Given a new observation, the incremental algorithm uses a subset of features to pick the top few candidates from among the existing contexts. The algorithm looks up all the features for those contexts, and for any observations in the same session as the new observation. It then computes the similarity between the current observation and each of the chosen contexts and observations to determine the best match.[3] If the score for the best match is above the threshold, the observation is either merged into an existing context, or can merge with a recent observation to start a new context.

Second, we have built a streaming infrastructure to determine when to run the incremental update versus rerunning segmentation on all the user's data. Given a new observation, the decision is a function of metadata such as the number of new input observations, sessions, and time since the last full model update. This function lets us measure and tune the trade-off between the slight drop in precision and recall for the incremental algorithm (compared to the full version), versus the resource savings.

[3]While Wang et al. [23] can be considered an incremental algorithm, the resource savings in our incremental algorithm come from not having to look up features for all the observations in the user's history.

The streaming infrastructure also enforces privacy requirements that if a user partially or completely deletes their data, there is a hard deadline by which the deletes are reflected in any contexts inferred for that user.

The result of this architecture is a system that has a soft deadline of 10 minutes by when a new observation is incorporated into the user model, and precision and recall very close to the full segmenter, while requiring dramatically fewer resources.

4. USER MODEL EVALUATION

We first describe how we generated the evaluation dataset. We then discuss the evaluation methodology, followed by experimental results.

4.1 Dataset

4.1.1 Can we annotate the complete user history?

In previous work [23, 16], researchers often had human raters completely annotate search histories for a small number of users, and used that as training data. There are two reasons why this was not an option for us.

- Fully annotating the history into contexts is feasible when the number of queries per user is small (e.g., 15 queries per user in [23]), or when identifying in-session tasks (since the number of queries in a session is small and they are more homogeneous). For our sample of users, we had an average of 1425 queries per user. From personal experience trying to do this on our own search history, fully partitioning thousands of queries into contexts is next to impossible, especially since there is often not a single "right" way to partition queries into contexts, e.g., should San Francisco and Palo Alto restaurants be in two separate contexts, or in a single context?
- For privacy reasons, raters are not allowed to see a user's full search history. In our evaluation, each rater saw only a single pair of observations from a user.

Hence, instead of fully annotating the user history, we pick pairs of observations from a sample of users.

4.1.2 Can we select pairs of observations at random?

We initially tried picking pairs of observations at random to send to raters, but found that only 1% of the pairs were positive, 59% were negative, and the raters did not agree on the remaining 40%. So we instead picked pairs of observations that shared at least one feature. The implication is that the true recall may be slightly lower than the recall on the dataset. However, since a pair of observations with zero common features are very unlikely to be rated part of the same context, we do not expect this difference to be significant.

4.1.3 Can we trust the ratings?

We sent each pair of observations to three raters. Each rater is asked whether

- the pair of observations is part of the same task, interest or habit (*Yes*);
- the observations are related, but may not be part of the same task or interest (*Maybe*); or
- the observations are not related (*No*).

We found that when all three raters agreed, the resulting dataset was very high quality. However, when there was disagreement, either the example was ambiguous, or (in our opinion) some of the raters were wrong — and it was not always the majority that was right. Hence we restricted our dataset to those pairs where all three raters agreed.

4.1.4 Dataset Description

We picked a random sample of 500 users, biased towards users with more queries. Users with more queries are much more likely to see recommendations both because they are more active and because we have more data to generate recommendations. Hence skewing towards active users gives us a more representative sample of our user base.

We picked 50 pairs of observations for each user, over a period of 6 months, for a total of 25,000 pairs of observations. We then used the subset of the data where all the raters unanimously voted either *Yes* or *No*: 48.7% of pairs were in this subset, yielding 12,181 examples. Out of these, 10.5% were *Yes* and the remaining 89.5% were *No*.

4.2 Methodology

The User Modeling algorithm has three key components:

1. The features used to compare pairs of contexts.
2. The classifier that determines whether two contexts should be merged.
3. The segmentation algorithm which uses the classifier to identify contexts from the sequence of observations.

The three components are somewhat independent, but not completely so. In particular, [23] fixed the segmentation algorithm to be BestLink, and used a latent structural SVM to learn the classifier in order to minimize errors in the final segmentation. They found that this yielded slightly better results (3% gain in precision, 2% gain in recall) compared to learning the classifier over pairs of observations (the "AdaptClu" algorithm in [23]).

Since we do not have fully annotated user histories, we did not use a latent structural SVM to train the classifier based on segmentation results. However, we did tune the precision-recall trade-off of the classifier based on segmentation results. We found that such tuning captures some of the gains of training the classifier using segmentation results.

In the rest of this section, we focus on the features and segmentation algorithms. We used an off-the-shelf SVM classification library [6] for the second component. The first and third components are agnostic to the choice of classifier, and using a more sophisticated classifier should help improve the segmentation quality further.

4.3 Experimental Results

Our primary point of comparison is with Wang et al. [23], the most recent work on finding cross-session search tasks. For segmentation algorithms, we also compare against HAC and canopy clustering [20].

All experiments were run using 10-fold cross-validation for the classifier, and 5-fold cross-validation for the segmenter. We measure precision, recall, and F_1 (the harmonic mean of precision and recall).

4.3.1 Predictive Value Feature Weighting and Scaled-Cosine

To our surprise, Predictive Value Feature Weighting (PVFW) and ScaledCosine did not matter for the classifier:

	Precision	Recall	F_1
Base	0.938	0.862	0.898
PVFW	0.960	0.854	0.904
PVFW+ScaledCosine	0.963	0.830	0.892

However, for the segmenter, PVFW + ScaledCosine does better than the base algorithm, and the difference is statistically significant.

	Precision	Recall	F_1
Base	0.807	0.907	0.854
PVFW	0.844	0.899	0.871
PVFW+ScaledCosine	0.894	0.878	0.886

For the classifier, when looking at a single pair of observations, the variance in confidence is only across feature dimensions, e.g., the inferred entities may be low confidence while the categories are high confidence. For the segmenter, when looking at a pair of contexts, there is variance in confidence both across feature dimensions and across observations within a single feature dimension, e.g., within a context the inferred entities may be high confidence for some observations and low confidence for other observations. Hence PVFW+ScaledCosine has much more scope to improve precision and recall in the segmenter than in the classifier.

4.3.2 Taba vs Wang Features

We also compared our features with those used by Wang et al. [23]. We found that our features dramatically outperformed the Wang features. Combining both set of features yielded a small improvement over the Taba features.

Features	Precision	Recall	F_1
Wang	0.949	0.719	0.818
Taba	0.963	0.830	0.892
Taba+Wang	0.968	0.848	0.904

4.3.3 The Top Feature Dimensions

To determine which feature dimensions were most important for the classifier, we used forward selection. We started with no feature dimensions, and picked the feature dimension with the highest F_1 (Query Words). We then added the feature dimension that improved F_1 the most (Category). Repeating this process yielded the results below:

Features	Precision	Recall	F_1
Query Words	0.940	0.682	0.790
+ Category	0.933	0.792	0.857
+ Entity	0.938	0.802	0.865
+ Forum	0.937	0.810	0.869

Interestingly, the model with just two dimensions (Query Words and Category) outperforms Wang, even though Wang has both of these features (and many more). PVFW+ScaledCosine may not matter for classification when we have all the Taba features (due to redundancy between features), but does matter when we have only a few features.

4.3.4 NLAC is very fast, and just as accurate.

We compared HAC, NLAC and BestLink, using the Taba features:

Segmenter	Precision	Recall	F_1	Time/User (s)
NLAC	0.894	0.878	0.886	4.26
Canopy	0.896	0.867	0.881	34.97
BestLink	0.901	0.812	0.855	53.58
HAC	0.892	0.879	0.885	122.96

NLAC was 30 times faster than HAC, and around 8 to 12 times faster than BestLink and Canopy clustering [20]. While Canopy and NLAC are motivated by similar intuitions, the details matter. Canopy runs HAC within each canopy. For our dataset, we often have large canopies, and that slows down the Canopy algorithm. In contrast, NLAC only re-computes similarity when deciding whether to merge two candidates, and is thus substantially faster. By making several passes, starting with a high threshold, NLAC is able to maintain accuracy while requiring substantially fewer full similarity computations than Canopy.

With respect to quality, NLAC, HAC and Canopy were very similar. Since both NLAC and Canopy are approximations to HAC, this says that both NLAC does not sacrifice quality for the performance gains. (While NLAC has the highest F_1, the difference from HAC or Canopy is not statistically significant.) BestLink did worse on F_1 than the other three algorithms, and the difference was statistically significant.[4]

We wished to experiment with Wang+BestLink, but found that it was more than 1000 times slower than Taba+NLAC. Computing the Wang features for a pair of observations was more than 100 times slower than computing the Taba features, and BestLink computes similarity for many more pairs than NLAC.[5] Hence we were unable to compare the Wang and Taba features for segmentation.

4.3.5 Context Distributions

Finally, we looked at the distribution of time spans of the contexts. While this sample is biased towards more active users, it is still noteworthy that half the queries were in contexts that spanned more than a month.

	Contexts per User	Queries per Context	% of Queries
Multi-query	171	7.5	90
Multi-session	91	11.3	72
Multi-day	75	12.7	67
Span > 1 week	59	14.7	61
Span > 1 month	36	19.7	50

[4]In our initial set of experiments, where we trained on 60 users and tested on 40 users, the gap between BestLink and the other algorithms was much larger. This suggests that BestLink is more sensitive to the amount of training data and the quality of the classifier.

[5]The slowest Wang feature is edit distance — for every pair of observations, we need to compute the minimum and average edit distance between their sets of web results. For a user with 2000 queries and 10 web results per query, the Wang+BestLink algorithm would need to compute the edit distance between 200 million pairs of URLs.

5. USE CASES IN A PERSONAL ASSISTANT

The primary goal for a personal assistant system is to determine the most relevant information for a user at a given point in time. We now describe, by walking through some real world (deployed) examples, how the user model defined in Section 3 serves as the foundation for building a personal assistant. We can group these examples into two classes, based on the source of the content:

- The content is a feed of public data extrinsic to the user contexts, and the primary problem is to match contexts against the content.
- The observations and contexts are the source of the information, and collaborative filtering is used to generate recommendations.

5.1 Matching public data feeds against the user model

This class includes many use cases that are highly rated by users. Examples include:

- Sports score updates for teams that the user follows. The content is a licensed third party feed of sports scores, where the sports teams have been reconciled against Freebase. The attributes (Section 3.6.2) from contexts with the appropriate context type tell the personal assistant which teams the user follows.
- Entity reminders personalized to a user. The entities are those identified as attributes of the appropriate type. Examples include reminders when
  - an artist a user is interested in is performing nearby,
  - a band a user likes releases an album,
  - an author a user follows releases a new book.

At first glance, it may appear that we do not need contexts for these examples: couldn't we simply count entities referenced in the user's query stream? While counting entities will work reasonably well, using contexts can yield significantly better results. For example, merging queries that are about different players in a football team together with queries directly referencing the football team gives a more accurate estimate of the level of interest in a football team.

The second takeaway is that most of these use cases will not work if we have only a few days of data. For example, there is typically a long gap between when the user searches for an artist and when that artist is performing nearby. Similarly, recall for sports teams would be very poor with just one week of data. Being able to scale context identification to many months of data is critical for these use cases.

5.2 Collaborative filtering

Matching public data feeds against the user model works well when such feeds are available. For the huge number of use cases where there are no public data feeds, the observations and contexts serve as the source of recommendations.

5.2.1 *Interest Updates*

We show personalized content recommendations for the user's long term interests, using collaborative filtering. Our system consists of the following components:

- **Fresh Aggregates**: For each entity (or query) that occurs in a sufficiently large number of contexts, we get the set of users who have that entity (query) in any of their contexts. We then aggregate what those users have read recently.[6] Thus for a given entity, we have aggregate statistics on what fresh content would be interesting for users with that entity.
- **Context Relevance Score**: We use a number of signals computed from the context to decide whether the user will be interested in updates for this context. The signals include a "fresh intent" score (a combination of how often the user looked at fresh content in this context, and how often other users looked at fresh content in similar contexts), the recency of the context, and the temporal distribution of activity in the context. This is similar in spirit to the work by Agichtein et. al. [3] on predicting search task continuation. Computing these signals over the context is more accurate than looking at an entity or query in isolation. For example, a user may not have looked at a player recently, but if she has recently looked at articles about the player's team, that is indirect evidence of continued interest in the player.
- **Recommendation Ranking**: Given a context with a high context relevance score, we look up the fresh aggregates for each entity and query in the context. Each aggregate gives us the articles users interested in that entity or query have read recently, and combining these lists gives us a candidate set of recommendations. These recommendations are filtered for quality, and then scored for timeliness (to get new content) and relevance to the context.

5.2.2 *Task Recommendations*

The goal is to assist users with longer tasks, e.g., trip planning or buying a camera, by leveraging the experience of users with similar tasks. The approach is similar to the system for interest updates, though the details are quite different:

- We use **context aggregates** instead of fresh aggregates. For each URL, entity or query that occurs in a sufficiently large number of contexts, we compute aggregate statistics over the set of contexts in which that URL (or entity or query) appears. For example, given an URL, the aggregates tell us what other URLs, entities and queries users looked at later in that context. We also compute meta-data, such as the fraction of clicks in those contexts that are on news or videos.
- The context relevance score function is tuned differently, e.g., with much tighter bounds on how recently the last activity in that context occurred. For long-term interests, people are usually interested in fresh content even without recent activity. However, recommendations for tasks are only useful when the user is actively involved in the task.
- The scoring functions for recommendations are different, e.g., diversity matters more, and freshness matters less.

[6]For privacy reasons, the algorithm only retains articles that have been read by a sufficiently large number of users.

6. CONCLUSION

We presented a user modeling system that, given web search history for opted-in users, identifies coherent contexts not just within sessions or over a few days, but across months of data. Looking at these long time spans allows the system to identify long-term tasks, interests and habits, in addition to identifying short-term tasks.

Our work includes three main technical contributions. First, we showed that intuitions and algorithms that work well over small time periods often do not scale to large time spans — this opens up a very interesting (and high impact) area for future research. Second, we have identified a set of features that yield much higher precision/recall than prior work, and whose computation requires only a small fraction of the CPU time (compared to prior work). Finally, we presented a new segmentation algorithm, NLAC, that uses indexing and lightweight scoring to provide similar precision and recall to HAC while being 30 times faster.

The last two contributions were critical, since our user modeling system is deployed in production for hundreds of millions of users, as part of Google Now. The contexts identified by the system are used both for recommending content from public data feeds, as well as serving as input to a collaborative filtering system for generating interest updates and task recommendations.

There are many promising directions for future work. We highlight three areas. First, while our set of features outperform prior work, we think there is plenty of scope for doing even better. In particular, it would be interesting to see if embedding query words (as well as other feature dimensions) in a vector space (e.g., [15]) and computing similarity between words in that vector space will yield better results than the bag of words approach. Second, modeling relationships between contexts (sub-contexts, related contexts) will yield a much richer user model. Finally, the incremental version of our algorithm has not yet been able to match the quality (precision/recall) of the non-incremental version. Developing an incremental algorithm that yields similar quality, or even just being able to predict when an incremental algorithm would do poorly, are challenging problems.

Acknowledgments

We thank Carolyn Au, Aparna Chennapragada, Andrew Dai, Elena Erbiceanu, Surabhi Gupta, Mahesh Keralapura, Karthik Lakshminarayanan, Carl Lischeske, David Martin, Kiran Panesar, Kelvin So, Shen Wang, Chenjun Wu, and Matt Wytock, who worked with us to design, build and launch the Taba system. We thank Andrew Kirmse, Baris Gultekin, Pablo Bellver, German Cheung, Jan-Willem Maarse, Anand Pillai, Jie Shao, Phil Verghese, Randy Wilson and Benyu Zhang for their help in launching Taba as part of Google Now. We thank Anurag Vyas for help in analyzing the evaluation data.

7. REFERENCES

[1] Google Now, `http://www.google.com/landing/now/`.

[2] Microsoft Cortana, `http://www.windowsphone.com/en-us/how-to/wp8/cortana/meet-cortana`.

[3] E. Agichtein, R. W. White, S. T. Dumais, and P. N. Bennet. Search, interrupted: Understanding and predicting search task continuation. In *SIGIR*, 2012.

[4] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD*, 2010.

[5] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proc. of the Conf. on Computational Learning Theory*, 1998.

[6] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[7] M. Collins and Y. Singer. Unsupervised models for named entity classification. In *Joint SIGDAT Conf. on Empirical Methods in NLP and Very Large Corpora*, 1999.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *CACM*, 51(1), Jan. 2008.

[9] T. Finley and T. Joachims. Supervised clustering with support vector machines. In *ICML*, 2005.

[10] W. Hua, Y. Song, H. Wang, and X. Zhou. Identifying users' topical tasks in web search. In *WSDM*, 2013.

[11] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, Sept. 1999.

[12] R. Jones and K. L. Klinkner. Beyond the session timeout: Automatic hierarchical segmentation of search topics in query logs. In *CIKM*, 2008.

[13] H. Koga, T. Ishibashi, and T. Watanabe. Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowl. Inf. Syst.*, 12(1):25–53, May 2007.

[14] A. Kotov, P. N. Bennett, R. W. White, S. T. Dumais, and J. Teevan. Modeling and analysis of cross-session search tasks. In *SIGIR*, 2011.

[15] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In *ICML*, 2014.

[16] L. Li, H. Deng, A. Dong, Y. Chang, and H. Zha. Identifying and labeling search tasks via query-based Hawkes processes. In *SIGKDD*, 2014.

[17] C. Lucchese, S. Orlando, R. Perego, F. Silvestri, and G. Tolomei. Identifying task-based sessions in search engine query logs. In *WSDM*, 2011.

[18] C. Lucchese, S. Orlando, R. Perego, F. Silvestri, and G. Tolomei. Discovering tasks from search engine query logs. *ACM Trans. Inf. Syst.*, 31(3), Aug. 2013.

[19] C. Lucchese, S. Orlando, R. Perego, F. Silvestri, and G. Tolomei. Modeling and predicting the task-by-task behavior of search engine users. In *OAIR*, 2013.

[20] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *SIGKDD*, 2000.

[21] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33, 1999.

[22] A. Spink, M. Park, B. J. Jansen, and J. Pedersen. Multitasking during web search sessions. *Inf. Process. Manage.*, 42(1), Jan. 2006.

[23] H. Wang, Y. Song, M.-W. Chang, X. He, R. W. White, and W. Chu. Learning to extract cross-session search tasks. In *WWW*, 2013.