Appstract: On-The-Fly App Content Semantics With Better Privacy

Earlence Fernandes
University of Michigan

Oriana Riva
Microsoft Research

Suman Nath
Microsoft Research

ABSTRACT

Services like Google Now on Tap and Bing Snapp enable new user experiences by understanding the semantics of contents that users consume in their apps. These systems send contents of currently displayed app pages to the cloud to identify relevant *entities* (e.g., a movie) appearing in the current page and show information related to such entities (e.g., local theaters playing the movie). These new experiences come with privacy concerns as they can send sensitive on-screen data (bank details, medical data, etc.) to the cloud. We propose a novel approach that efficiently extracts app content semantics *on the device*, without exfiltrating user data. Our solution consists of two phases: an offline, user-agnostic, in-cloud phase that automatically annotates apps' UI elements with stable semantics, and a lightweight on-device phase that assigns semantics to captured app contents on the fly, by matching the annotations. With this automatic approach we annotated 100+ food, dining, and music apps, with accuracy over 80%. Our system implementation for Android and Windows Phone—*Appstract*—incurs minimal runtime overhead. We built eight use cases on the Appstract framework.

CCS Concepts

•**Human-centered computing** → **Ubiquitous and mobile computing systems and tools;** •**Information systems** → *Specialized information retrieval;*

Keywords: Mobile applications; Entity extraction; Entity templates.

1. INTRODUCTION

Understanding the semantics of contents users consume in their apps enables new user experiences. This is evidenced by two recent services: Google Now on Tap [16], and Bing Snapp [40]. For instance, a user listening to a song in Spotify can invoke Now on Tap to experience features not provided by Spotify itself—e.g., an augmented "card" with *the song*'s lyrics and options to purchase *the song*. These services work by sending current screen content to their cloud backend, which uses information retrieval techniques to extract key *entities* (e.g., a specific song) and *entity types* (e.g., song title), with the goal of understanding user context and intent. In response, the cloud backend can then send information and actions to the client which are relevant to current in-app activity.

MobiCom'16, October 03-07, 2016, New York City, NY, USA

DOI: http://dx.doi.org/10.1145/2973750.2973770

We expect to see many such experiences that leverage the semantics of contents consumed by users inside apps. Some examples are: (1) *Contextual menus:* A user highlights some text in an app, say "12 140th Ave NE, Portland", and the OS, understanding that the highlighted text is *an address*, brings up a menu to view it in a map; (2) *Personalization:* In Yelp, the user often selects restaurants in the "Italian" category, and the app store, by leveraging user's historical interest in *Italian food*, can suggest Italian food/recipe apps; (3) *Content mash-up:* A user books a hotel and views restaurant pages using Expedia and Open Table, respectively. The user's app automatically creates a trip itinerary by mashing up *hotel*-type and *restaurant*-type information that the user viewed and interacted with, through clicks on "Book" and "Like" buttons.

At the core of these experiences is the task of identifying entities along with their types, which appear in app pages during user interaction. One implementation option is cloud-based, as in Now on Tap. However, the cloud backend can see sensitive user data such as financial or health information appearing in a finance or in a health app, making it "big-brother freaky" [43]. As more services like Now on Tap emerge, provided by both first- and third-parties, we expect increasing privacy concerns. Now on Tap tries to address these issues in two ways: (1) It requires the user to manually invoke it when additional information or suggestions related to the current activity are desirable; however, such an approach precludes the useful scenarios that require *continuous* profiling of a user. (2) It allows app developers to disable Now on Tap for the entire app; but this approach excludes also non-sensitive parts of the app. Besides the loss in functionality, these point-solutions still do not adequately address privacy concerns. Since it is easy to invoke Now on Tap by long tapping the home button, users can mistakenly invoke it while sensitive information is displayed on screen. App developers often do not explicitly disable Now on Tap even if the app contains sensitive data. Out of 1000 top Android apps we analyzed, only 45 explicitly disabled Now on Tap (§2.1). Only 2 of 100 medical apps disabled Now on Tap, thus allowing it to acquire possibly sensitive data like medical conditions or medications.

We explore techniques for extracting semantics (e.g., entity types) of on-screen app contents *without sending the contents to the cloud*. We develop an *on-device* technique that is less privacy invasive than a cloud-based one, as no user data leaves the device. Moreover, it is more efficient. Our measurements on 33 top Android apps show that each invocation of Now on Tap sends 6.8 kB on average (up to 37.2 kB maximum) to the cloud, a nontrivial overhead that our on-device approach can avoid. Moreover, these privacy and efficiency benefits make our solution suitable for sce-

narios requiring *continuous* monitoring of in-app activities.

A key challenge that we must address is how to *efficiently*, *continuously*, and *instantaneously* associate semantics to screen contents (e.g., that some text appearing in an item in a List in the UI is a *RestaurantName*), *without* exfiltrating user data to the cloud, and *without* developer effort—app developers could manually annotate apps to provide semantic information, but this would require them to annotate hundreds of UI elements for each page in the app, making it impractical.

Our key insight lies in the unique structural properties of mobile apps. We observe that for many apps, pages are usually instantiated from a *small* number of page classes, whose UI layout remains the *same* across user interactions (e.g., in a restaurant app, the page "RestaurantDetails" might display information for different restaurant entities but always with the same UI layout). Moreover, many UI elements in those UI layouts contain single *entities* (e.g., a TextBlock contains the name of *one* restaurant). This makes it possible to annotate the semantics and relationships between the UI elements of an app page *offline* once—such annotations remain valid at runtime. These observations lead us to a two-phase solution: (1) In the offline, user-agnostic (in-cloud) phase, we use novel techniques to analyze a large corpus of app contents to automatically identify *entity types* and *entity relationships* of various textual UI elements of an app. The entity types and relationships are then encoded into *entity templates* (i.e., mappings between UI elements and entity types/relationships for each page class of the app). (2) In the lightweight, online (on-device) phase, we assign semantics to captured app contents by simply matching the contents of an app page with the corresponding entity template.

This approach achieves better privacy in two ways. *(i)* The offline phase is user-agnostic, so assuming that the semantics provided by entity templates fulfill the scenarios' needs, no user data leaves the device. *(ii)* If the scenario requires remote processing, the type and amount of data sent out can be selectively controlled through templates. Assume a user configures her device to not share finance-type data (e.g., bank account number). While chatting with her bank assistant in the bank app, she may invoke Now on Tap to retrieve the definition of a legal term, and Now on Tap may use the template to filter out texts of non-permitted entity types.

Although our on-device solution is not as universal as the cloud-based approach of Now on Tap, it covers many scenarios (including some of Now on Tap) with better privacy guarantees. Overall, this paper makes the following contributions. First, while the idea of a hybrid architecture using app models has been used before [10], a key contribution of this work is the *automatic* generation of entity templates (§4), which allows us to scale to many apps from different categories. Note that existing techniques for automatic extraction of web entities do not work with mobile apps (see §2.2). Second, we support our design with an app analysis revealing important structural properties of modern apps (§3). Third, we provide a complete system, called *Appstract*, implemented both on Android and Windows Phone, including a declarative API for subscribing to app events and retrieving user history (§5.1). Finally, we implement eight use cases including new OS features (e.g., contextual notifications), extensions to existing first- (e.g., Cortana) and third-party apps, and new types of apps like information mash-ups (§5.2).

2. MOTIVATION

We motivate the need for an approach for extracting semantics of in-app content which provides better privacy. We use Now on Tap as an example of cloud-based service existing today, which relies on in-app semantics extraction. However, many such services provided by both first- and third-parties are likely to emerge in the near future. In fact, Bing Snapp running on Android phones can already be considered a third party service of such kind. As this ecosystem of semantics-based services grows and the supported user experiences diversify, we expect increasing privacy concerns.

App category	# apps tested	# apps with Now on Tap disabled
Social	100	3
Travel/Local	100	3
Finance	100	17
Medical	100	2
Health/Fitness	100	4
Shopping	100	4
Education	50	1
Entertainment	50	3
Music/Audio	50	0
Lifestyle	50	1
Business	50	5
Personalization	50	0
Tools	50	1
Books/Reference	50	1
Total	1000	45

Table 1. Android apps with Now on Tap disabled.

2.1 Analysis of Google Now on Tap

Now on Tap (as well as Bing Snapp) sends on-screen content to the cloud backend. As mentioned in §1, it addresses privacy concerns by (i) relying on user's permissions, and (ii) relying on app developers. We now discuss these options.

Is explicit user consent enough? Android allows disabling Now on Tap for the entire device, but several user studies have shown that user-based permissions are ineffective, even with additional text warnings [20, 14, 9]. Now on Tap is activated only upon explicit user invocation (a long tap), with the hope that a user invokes it only when the screen does not have any sensitive information. However, it is relatively easy to mistakenly invoke it (e.g., by tapping the home button longer than intended, thus resulting in a long tap). A user may not be aware of which data is actually transmitted (e.g., if lots of content is present on the screen). A long tap action can be associated with other services as well (in system settings), so a user may not even realize that a long tap will start Now on Tap. Finally, an explicit consent model does not fit scenarios requiring continuous app data profiling.

Are app developers protecting user privacy? Developers of apps running on OSes with services similar to Now on Tap have the option of disabling screen content capture for app pages they deem to be sensitive. While this form of privacy control does prevent the user from enjoying the benefits of services like Now on Tap, privacy is preserved. We conducted measurements to determine the prevalence of this form of privacy control. We selected 1000 top Android apps, including 400 top apps from the Google Play top 8 categories [5], and the top 100 apps from privacy sensitive categories (Medical, Finance, Health & Fitness, Travel & Local, Social, and Shopping), and counted the number of apps that explicitly disabled Now on Tap on some of their pages (by setting FLAG_SECURE on the window). As shown in Table 1, across all 14 categories, only 45/1000 apps have disabled Now on Tap. Only 17% of financial apps, and 2% of medical apps have Now on Tap disabled on some pages. We conclude that while some developers make the privacy-conscious decision of opting out of services like Now on Tap, a large majority rely on the user being constantly able to protect the privacy of information such as bank details, medical conditions, and contacts.

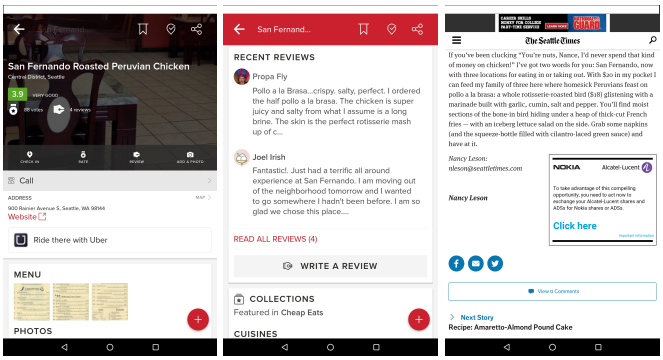


Figure 1. Three app screens with the same "San Fernando" restaurant entity. In-app content varies from structured (app view on the left) to unstructured (app view on the right).

Is data exposure needed from a system point of view? Services like Now on Tap and Bing Snapp work as follows.

1. *Screen content capture:* They capture app contents by taking a screenshot or by capturing textual contents on the screen via accessibility APIs.

2. *Semantics inference:* They send the logged raw data to the cloud backend for semantics inference. For instance, in Figure 1, semantics inference means associating the strings "San Fernando" and "900 Rainier Ave..." to their *entity types* (RestaurantName and Address, respectively), and grouping them based on their *entity relationships*—"San Fernando" and "900 Rainier Ave..." are related to the same entity. Clicks on "Call" or "Menu" buttons are *actions* that must be recognized as belonging to the same entity group.

3. *Entity linking:* Finally, they use the extracted data and semantics to look up an *entity* stored in their knowledge graph.

Now on Tap overlays informative cards that use both entity types and entity linking. For example, based on the entity type of "San Fernando", the card can show an option to make a reservation, and based on entity linking, it can show Google reviews of the restaurant. Entity linking requires a large knowledge base, hence cannot be on-device.

However, in many scenarios (see §1 and §5.2), entity linking is not required. Services like Now on Tap too, may only require entity linking to assemble a card with a restaurant profile or with a song's lyrics, but do not require entity linking to overlay a card offering driving directions to a restaurant's address appearing on the screen—recognizing that the text is of type Address is sufficient. Motivated by this observation, we argue that *if* the second step of semantics inference can be performed on the device, no user data needs to be exposed. A complete service may still send user data to the cloud for entity linking purposes, possibly compromising user privacy. However, this is out of scope for Appstract.

2.2 On-device Semantics Extraction

We aim to design a system capable of continuously capturing semantically-meaningful information about a user's in-app activities, with the following properties: *(i) On-device*: Local processing avoids any possible privacy leaks due to sending sensitive data to the cloud. *(ii) Ease of use and deployment*: The solution should require minimal developer and user effort. It should be compatible with existing mobile platforms and apps. *(iii) Scalability*: It should work with a large number of apps. *(iv) Efficiency*: Processing, memory, and network overhead on mobile devices should be minimal.

#Entities/#Words	Entity type	%Prec.	%Recall	%F1 score
37,018 / 370,022	RestaurantName	56.4	21.5	25.2
	Cuisine	74.5	15.4	25
	Address	81.8	64.7	62.8
53,594 / 281,526	RecipeName	55.7	22.9	26.3
	DishType	59.3	19.6	29.3
	Cocktail	99.85	59.1	70.6
90,438 / 156,632	MusicGenre	50	46	47.9
	SongTitle	37.6	79.6	46.1
	ArtistName	73.8	27.3	36.2
181,050 / 808,180	Average	65.4	39.6	41

Table 2. Precision, recall and F1 score of StanfordNER. Training and testing used k-fold cross validation with dictionaries (gazette lists).

A possible solution for locally extracting in-app content semantics is to require the developer to annotate their apps (more precisely, UI elements) with semantics. However, this compromises the minimal developer effort goal.

Another option is to locally run *named entity recognition* algorithms [39, 22, 34]. These algorithms identify mentions of named entities, say a hotel name, in given input text, typically a web document. They fall under two general categories: techniques which require a knowledge base at runtime (supervised [7, 21], semi-supervised [31, 3, 25], and unsupervised [11, 19, 41]), and techniques which do *not* require a knowledge base at runtime, such as NLP rule-based techniques [36]. As required knowledge bases (e.g., dictionaries of known entities) are typically very large, techniques of the first type are not suitable to run on a mobile device.

Rule-based techniques, on the other hand, can be efficiently run on a device. We tested the well-known Stanford CoreNLP Named Entity Recognizer (StanfordNER) [36]. This tool requires supervised training. Using Monkey [18, 23], a UI automation tool for crawling mobile apps, we collected text from 14 WP apps in the dining, food and music categories. We trained the tool to recognize entity types relevant to such apps, using dataset sizes comparable to those used in other NER use cases [38]. For improved accuracy (as described in [37]), during training, we additionally provided the tool with dictionaries consisting of 855877, 267057, and 35396 entities in the music, restaurant, and recipe categories, respectively. Table 2 reports precision, recall, and F1 scores for each entity type, obtained using k-fold cross validation (k = number of apps).

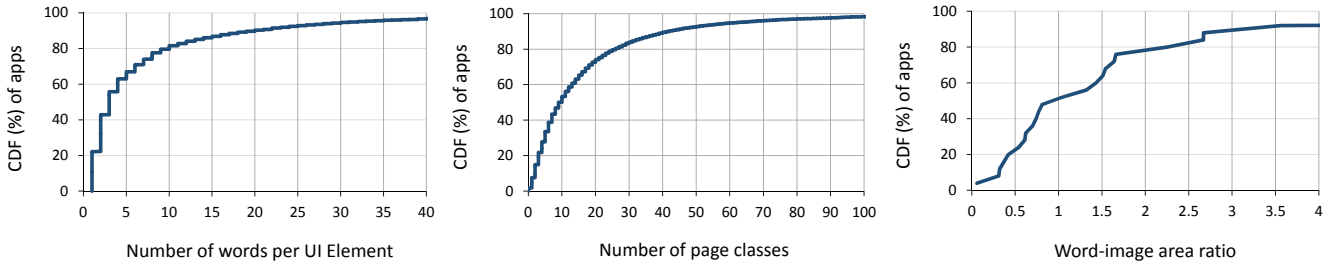
The performance is poor with an average F1 score of 41%. NER recognizes few entities (40% recall) with an average precision of 65%. Out of nine entity types, only addresses and cocktails have F1 scores higher than 60%[1], but this is still insufficient. In the web, entity extraction algorithms usually target at least 80% precision and recall [41, 28].

In summary, none of the above client-based techniques, including state-of-the-art web extraction techniques such as StanfordNER, achieve all our goals. Instead, we propose a 2-phase, template-based approach.

3. OVERVIEW OF OUR SOLUTION

We balance the trade-off of on-device and zero developer effort using a hybrid device-cloud approach (see the overall architecture in

[1]These two entities are better recognized for different reasons. Addresses always appear in a standard format (number, street name, city) which NER learns to recognize. Cocktails are more easily recognized because our cocktail apps contain only cocktail entities and no closely-related entities, thus reducing ambiguity.



(a) avg # words in UI elements (b) # app pages classes (c) avg word-image area ratio per page class

Figure 2. CDFs of (a) the number of words per UI element, (b) the number of page classes per app, and (c) the word-image ratio per page class. Results based on top 25 (a and c) and top 14k (b) Android apps.

Figure 3): apps are annotated with semantics that remain valid at runtime in an offline and user-agnostic phase, and these annotations are used on-device to efficiently and continuously extract app semantics, at runtime. We show that this design is feasible through some key observations on how modern apps display content.

App UI structure. Mobile apps display content organized as a set of pages that a user can interact with and navigate between. An app has a small number of *page classes*, which can be instantiated as many *page instances* (or pages, in short) with different contents (e.g., the details for Nara and Izumi restaurants are shown in two instances of the same page class). A page displays content according to the UI layout defined in its class. An app page contains *UI elements* (buttons, textboxes, lists), which are organized in a *UI tree* (like a DOM tree), where nodes are UI elements and a child element is contained within its parent element. Some UI elements are interactable and have *event handlers*. With the above terminology, we observe the following.

Singleton entities. In an app, many UI elements contain single entities, e.g., the title TextBlock contains one type of information, say the restaurant name. Developers tend to put various types of information (restaurant name, address, and phone number) in distinct UI elements so that they can be independently customized. To confirm this observation we study 25 Android apps in the top categories (news, weather, shopping, etc.). Using Monkey, we crawl the contents of all textual UI elements in the apps for several Monkey interactions. We then compute the CDF of the average number of words per each unique UI element across all page classes of all apps (Figure 2(a)). 70% of UI elements contain 6 or fewer words, and 50% have 3 or fewer.[2] The highest average word counts are with news apps (Flipboard has 17 and CNN has 10) and the lowest with video and shopping apps (YouTubeKids, Vine, Ebay have 3 or less). We repeat the test with 17 Windows Phone (WP) apps in the food, dining and music categories, and observe an average of 2.4 words per UI element. Overall, such small word counts suggest that *a single entity per UI element is relatively common*, so precise semantics can be associated to each UI element.

Few, stable UI structures. App pages are usually instantiated from a few page classes. Figure 2(b) shows the CDF of the number of page classes of 14k top Android apps in all categories. 60% of the apps include 13 or fewer page classes, and 80% of the apps include less than 26. A long tail of apps has more page classes (the maximum was 638). The average number of page classes for the 17 WP apps is 8.6. Although exceptions exist, for most apps the number of page classes is tractable. This makes it possible to cover all UI elements in an app by annotating UI elements in that small number of classes. Moreover, the annotations remain stable over

[2]The CDF covers 97% of the UI elements: the remaining 3% contains more than 40 words, and the maximum is 967.

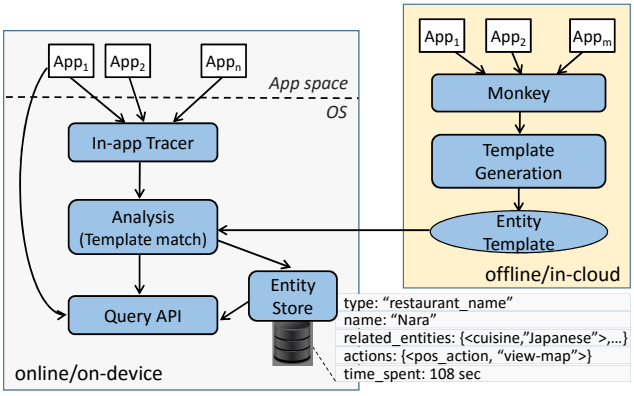


Figure 3. Offline-online hybrid architecture.

time: a UI element annotated as restaurant name can be instantiated with any number of entities, all of which will be restaurant name. This implies that *UI elements can be annotated once offline and be repeatedly used later at runtime*.

Offline-online hybrid architecture. The above observations lead to the two-stage architecture shown in Figure 3. Given an app, offline (in the cloud) and without any user involvement, we execute the app with a *Monkey* [18, 23], a UI automation tool that can automatically crawl pages of an app. The Monkey launches the app, interacts with its UI elements (by tapping on buttons, filling out text boxes, swiping pages), and navigates to various pages. The Monkey is configured to visit all pages of a given app, and to log UI structure (UI tree) and contents of all visited pages. An app is executed several times. This raw data is then processed using several entity extraction techniques (§4) to produce an **entity template** for each page class in the app. An entity template is essentially a UI tree annotated with semantics.

At runtime, the system captures app data (as described in §5.3). This data has a similar format as that generated by Monkey offline, but here the captured content is user-generated, and possibly privacy-sensitive. The system immediately labels the raw data with entity types and groupings using entity templates. For example, it recognizes a restaurant name, address and phone number, and combines them into one object which is stored in the Entity Store.

This design achieves better privacy in two ways. First, entity templates reduce the online step to a simple matching operation which can be executed on the client, so no data exposure occurs unless entity linking is needed. Second, entity templates can reduce the data exposure even for services requiring entity linking. In fact, they provide a systematic way to control which data can leave the device. In a complete system, users may configure permissions regulating which types of data (entities) can be shared, and the system using entity templates may omit data of private entity types.

```
P[0].G[0].B[0].T[0], unknown
P[0].G[0].L[0].C[*], (restaurant, address)
P[0].G[0].T[0], unknown
P[0].G[0].B[0].T[0], unknown

P[0].G[0].L[0].C[*].T[0], group 1
P[0].G[0].L[0].C[*].T[1], group 1
```

Figure 4. Entity template for the UI tree in Figure 5. The format is {node type, entity type} in the first section, and {node type, group identifier} in the second section.

4. AUTOMATICALLY GENERATING ENTITY TEMPLATES

The above design simplifies the online step of semantics extraction, but the offline/cloud stage is nontrivial. Prior work, also adopting a hybrid architecture, employed crowd-sourcing to generate task templates [10]. We advance the state-of-the-art by proposing a framework for *automatically* generating entity templates, with no human involvement. By being automated, it can scale to large numbers of apps.

The entity template of an app is a collection of templates, one per page class. Each template contains two components: (1) mapping between a UI tree node (a UI element) to the entity type of the node's content, and (2) relationship between different UI tree nodes. A template is all-inclusive: a UI tree node for which there is no mapping in the template is assumed to have "unknown" entity type, while the one without relationship information is assumed to be unrelated to all other nodes. Figure 4 shows an example of entity template.

We generate entity templates using a corpus of Monkey-collected app page contents. This corpus is partitioned based on the {app name, page class name} tuple to produce one template for each partition. In the following, we assume one such partition, and describe how we analyze all pages in the partition. As an app page is represented as a UI tree, we use the terms page and UI tree interchangeably.

We use two techniques: (1) using a knowledge base to recognize entity types in UI trees (§4.1), and (2) analyzing UI tree structure to further increase accuracy of entity types and entity relationships (§4.2).

4.1 Extracting Entity Types from Corpus

To generate templates, we need to associate an entity type to the text contained in each leaf node of a page UI tree. We do so by leveraging existing entity extraction techniques. However, note that rather than applying them to the target data (i.e., user data captured at runtime), we apply them to our Monkey-generated corpus. We decide to adopt approaches based on a knowledge base (KB) because compared to NLP rule-based techniques they do not require training data, and, as our evaluation shows (Figure 8 in §6.1), they can provide higher accuracy. Based on the app category, the KB contains names of known entities that are relevant to that category (e.g., song titles and artist names for the music category).

Features for entity type classification. We assign entities to UI tree nodes using a diverse set of clues, called *features*, capturing various properties of the input text. We use the following three features (common also for web documents).

1. Boolean match: If the input text matches the name of a KB entry, then the text is with high probability of the same type as that of the matching KB entry. This approach captures exact matches. However, to deal with shortened words (Cafe vs. Cafeteria) or missing words ("Bellevue Cantina" vs. "Cantina"), we say that two entries match if they have 90% words in common and are in the same relative order.

2. Tf-idf similarity: Some entity domains can be potentially infinite. For example, a recipe can have a creative name, consisting of a combination of key ingredients, which may not appear in the KB of known recipes. We call such domains *high-ambiguity*. For these, exact matches can be rare, particularly when KBs are incomplete and limited in size. Instead, we use a similarity metric that captures the *tf-idf based cosine similarity* [39, 27]. At a high level, the tf-idf value of an input text increases proportionally to the number of times the words composing it appear in the KB (term frequency), but is offset by the frequency of the words in the KB (inverse document frequency), which helps control the fact that some words are generally more common than others. For each input text we construct the normalized tf-idf vector, and then compute the dot product of this vector with the tf-idf vector of each KB entry (cosine similarity). The value of this feature is between 0 and 1, and the more the number of words shared by the input and the KB entry, the higher the value.

3. Text length: This Boolean feature says whether the number of words in the input text is consistent with typical length distribution of entities of the same type, provided by the KB. For instance, restaurant names are usually shorter than addresses, and addresses are usually less than 20 words.

The classifier. We use the following algorithm to compute the weight $w_{x,e}$, $0 \leq w_{x,e} \leq 1$, defined as the likelihood of a given text x being an entity of type e. First, we use the length feature: if x has fewer (or more) words than the lower (or upper) bound of known entities of type e, we produce $w_{x,e} = 0$. For example, in our KB of addresses, entities have a maximum length of 13 words. If an input text consists of 50 words (e.g., a restaurant review) it is unlikely to be an address. Next, for low-ambiguity entities, such as restaurant names and song titles, we consider the exact match feature. If x matches with an entity in the KB which has type e, we produce $w_{x,e} = 1$, otherwise 0. For high-ambiguity entities, such as recipe names, we produce $w_{x,e}$ as the tf-idf similarity score of x and the KB for entity type e.

We repeat the above steps for all possible entity types for the given app (based on its category). Finally, we compute the likelihood of x not being any of the entities under consideration as $w_{x,\phi} = k * \prod_e (1 - w_{x,e})$.[3] Intuitively, $w_{x,\phi}$ is close to zero if x is classified as *at least one* known entity type with high likelihood values, and close to 1 only if it cannot be classified as any entity with high likelihood. Finally, we normalize the weights to probabilities as $p_{x,e} = w_{x,e}/(w_{x,\phi} + \sum w_{x,e})$.

Given a UI tree leaf node n, the above process annotates it with its *uncertain entity type*, defined as the vector $\overline{E}_n = \{p_1, p_2, \ldots, p_k\}$, where the entity type of n is i with probability p_i. Without losing generality, we assume that the vector's first entry represents "unknown", and hence the vector $\{1, 0, \ldots\}$ means that the input text is of type "unknown". A node's entity type is unknown if it does not contain any text or if no entity is recognized based on the KB.

Augmenting the KB-only algorithm. With the algorithm described so far, for each leaf node in the UI tree, the entity type with the largest probability can be deemed the node's deterministic entity type. As our evaluation shows (§6.1), this approach, which we call *KB-only*, may fail to recognize entities or lack confidence

[3]We define $k = n/(n + c)$, where n is the number of possible entities for a given text, and we set c=2 by default, but in general c can be adjusted to control the aggressiveness of the entity assignment.

in its output (small p values in $\overline{E}$), especially when the KB is much smaller than the universe of that entity type. In the web, to augment KB-based techniques, "context" (text surrounding input text) is typically used to disambiguate entities. For example, in the leftmost screen in Figure 1, is "San Fernando" a restaurant or a Saint name? If the name "San Fernando" appears with mentions of other restaurant or food entities (as in the rightmost screen), it may be classified correctly.

Pages in mobile apps typically contain far less text than web documents due to smaller screen sizes, so context is generally poor. Additionally, a common mobile UI design practice is to display images rather than text to organize lots of information (including contextual data like "menu" and "food" that could help disambiguate entities). Using the 25 top Android apps introduced earlier (§3), we estimated how much content in a typical app is textual versus images. For each app page, we computed the ratio between the area of the screen that is covered by words and by images. A small ratio means that the content is image dominated. As the CDF in Figure 2(c) shows, in 92% of the apps the ratio is below 4, and in 52% the ratio is 1 or less. Apps like Pinterest are completely image-dominated (ratio is 0.06).

To compensate for the lack of context, we propose UI tree structural analysis, designed to improve the accuracy of entity classification for app pages with little text (first and second screen from the left in Figure 1). For unstructured pages with lots of text (third screen) web entity extraction techniques using context-derived features may be applied [41].

4.2 Structural Analysis of UI Trees

The previous KB-only algorithm produces many UI trees, one for each page in the corpus, with each leaf node annotated with its uncertain entity type. The goal of structural analysis is twofold: (1) to improve accuracy of entity types, and (2) to discover relationships among entities.

Given a node n in a UI tree U, we define its *node type* T_n to be a sequence of UI element names plus their indices from the root of the UI tree to n. For example, in Figure 5, the first leaf node on the left has type P[0].G[0].B[0].T[0]. We also define the *node type-entity* tuple of n as the tuple $(T_n, \overline{E}_n)$, where $\overline{E}_n$ is the uncertain entity type of n. We say that $T_n \in U$ and $(T_n, \overline{E}_n) \in U$ if U contains a leaf node with type T_n and node type-entity tuple $(T_n, \overline{E}_n)$, respectively.

4.2.1 Exploiting Cross-Page Similarity

Our first optimization is based on the observation that app pages instantiated from the same page class have similar UI structures and content types. For example, in OpenTable, the page displaying restaurant details has the name of the restaurant in a TextBlock at the top of the page. If a page is considered in isolation, the KB-only entity recognition algorithm described above may fail to recognize the name of the restaurant due to, e.g., an incomplete KB. However, knowing that the top TextBlock contains a restaurant name in many other instances of the same page class is a strong indicator that the name at the top of *this* page is a restaurant.

Given a set of UI trees $U_1, U_2, \ldots, U_k$, we consider all their node type-entity tuples. For each unique node type T, we compute $\overline{E}$ as the average of all uncertain entity vectors of T in all input UI trees. We assign $\overline{E}$ to all nodes of node type T as their common uncertain entity types. We represent the set S of all tuples $(T, \overline{E})$ as a merged UI tree that has a leaf node of node type T if and only if T appears in S. Intuitively, we combine entity knowledge across

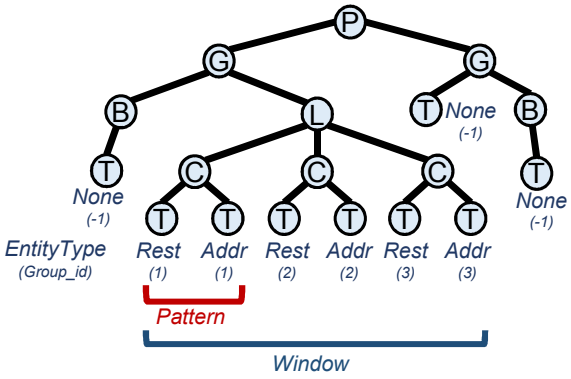


Figure 5. UI tree showing containment relationships of an app page's UI elements. Each node label indicates the UI element type (P=page, G=grid, B=button, L=list, T=text block, C=custom type). Node indexes are not shown. Labels below the leaf nodes indicate true entity types ($Rest$=restaurant name, $Addr$=address) and group ids.

similar pages to improve accuracy and robustness of recognition in pages where it would otherwise fail due to insufficient in-page contents or incomplete KBs.

Example. Consider a UI tree leaf node of node type T with contents "Burger King", "Kabab Palace", "The American" and "Via Tribunali". The KB-based algorithm produces their uncertain entity vectors as (unknown,cuisine,restaurant) = (0,0,1), (0,0.05,0.95), (0,0.5,0.5), (1,0,0), respectively. (The low probability for "The American" is due the name's similarity with the cuisine type; while that for "Via Tribunali" is due to its absence in the KB.) Then, we assign to all nodes the average of their uncertain entity vectors (0.25,0.14,0.61). This enables classifying "The American" and "Via Tribunali" as restaurant names with good confidence (0.61).

4.2.2 Exploiting In-Page Patterns

Our second optimization exploits the observation that semantically similar items in the same app page exhibit consistency in presentation style. Hence, the entity type of one item can be a good hint for that of another item with similar presentation. Consider a List, which is one of the most commonly used app UI structures to enumerate homogeneous items. Suppose the list contains three occurrences of {restaurant name, address} (Figure 5), and our entity recognition algorithm recognizes two of the names as restaurant names with high probabilities; the algorithm fails to recognize the remaining name due to, say, an incomplete KB. However, the fact that the entity pattern {restaurant name, address} repeats within the list suggests that also the other item is a restaurant name.

The example above is rather simplistic. In practice, apps have fairly large and complex UI structures. There are many types of UI elements used for enumerating or displaying collections of items (List, Grid, Hub, FlipView, etc.), and developers can define custom ones. Each item of these collection-like UI elements can contain arbitrary objects, possibly a nested collection-like UI element which recursively contains other collection-like UI elements, resulting in a deep UI tree. Moreover, leaf nodes generally contain heterogeneous entity types. Thus, *repeating patterns of entities* can be long (e.g., <restaurant name, address, phone number, price range, ... >), and simple rule-based techniques may not discover them. The problem is further complicated by the fact that entity types of UI tree leaf nodes are uncertain: discovering repeating patterns in an uncertain sequence is more challenging than in a deterministic sequence.

We introduce some definitions. An *entity pattern* is a sequence of deterministic entity types. An entity sequence of a UI (sub)tree with root n, denoted as $EntitySequence(n)$, is the sequence of uncertain entity types of all its leaf nodes, sorted by their node types.

Def 1. ***Matching Probability****:* The matching probability $MatchPr(\alpha, \mathbb{S})$ of a pattern α and an uncertain sequence $\mathbb{S}$ is $\prod_i \Pr(\mathbb{S}[i] = \alpha[i])$ if $|\alpha| = |\mathbb{S}|$, and 0 otherwise.

Def 2. ***Expected Frequency****:* The expected consecutive, non-overlapping frequency of pattern α in uncertain sequence $\mathbb{S}$, denoted as $ExFreq(\alpha, \mathbb{S})$, is as follows. Let $\mathbb{S} = \mathbb{S}_1 \circ \mathbb{S}_2 \circ \ldots \mathbb{S}_k$, where $\circ$ denotes sequence concatenation, and $\forall i, |\mathbb{S}_i| = |\alpha|$. Then, $ExFreq(\alpha, \mathbb{S}) = \sum_{i=0}^{k} MatchPr(\alpha, \mathbb{S}_i)$. Normalized expected frequency of α in $\mathbb{S}$, denoted as $NormExFreq(\alpha, \mathbb{S})$ is $ExFreq(\alpha, \mathbb{S})/n$, where $n = |\mathbb{S}|/|\alpha|$.

Def 3. ***Maximal Repeating Pattern****:* Consider a UI (sub)tree rooted at node n. Let $\mathbb{S}_i$ be the entity sequence of the subtree rooted at its i'th child. A pattern α maximally repeats in a UI (sub)tree rooted at node n if (1) $\exists i, MatchPr(\alpha, \mathbb{S}_i) \geq \theta$, and (2) $NormExFreq(\alpha, \mathbb{S}_1 \circ \mathbb{S}_2 \circ \ldots)$ is maximum among all α satisfying (1).

Given a (merged) UI tree, produced as described above, we want to identify the maximal repeating entity patterns in various subtrees of the UI tree. The patterns will be used to refine uncertain entity types of all leaf nodes of the UI tree.

We introduce Algorithm 1. It recursively traverses a UI tree top-down, and discovers the patterns bottom-up. It uses the following ideas.

Algorithm 1 MaximalRepeatingPattern(n)

Require: A node n in a UI tree, with leafs annotated with node type-entity tuples
Ensure: The `pattern` property of each nonleaf node has the entity pattern discovered for its descendant subtree

```
1:  n.pattern ← φ
2:  if (n is not leaf) then
3:    for all c ∈ n.children do
4:      MaximalRepeatingPattern(c)
5:    end for
6:    P* ← set of patterns appearing in children of n
7:    P* ← P* ∪ {α | ∃ c ∈ n.children, MatchPr(α, c) ≥ θ1}
8:    if P* is nonempty then
9:      S ← EntitySequence(n)
10:     pattern ← arg max_{α∈P*} NormExFreq(α, S)
11:     if (NormExFreq(α, S) ≥ θ2) then
12:       n.pattern ← pattern
13:       for all c ∈ n.children do
14:         c.pattern ← φ
15:       end for
16:     end if
17:   end if
18: end if
```

(1) Pattern windows: We define a *pattern window* to be the portion of $\mathbb{S}$ within which the pattern is likely to repeat. In Figure 5, a window consists of the six T nodes that have entity type labels below them. Algorithm 1 searches repeated patterns in the entire subtree rooted at a nonleaf node. This is based on the observation that UI elements in a repeated sequence of patterns are contained in a collection-like structure of homogeneous objects. In Figure 5, the elements are contained within a list (type L) of a custom, developer-defined type C. Due to this uniform structure, the sequence of repeated patterns cover *all* nodes under a common nonleaf node. A page may contain multiple (non-overlapping) windows, in which case Algorithm 1 will correctly label the corresponding nonleaf nodes with their patterns.

(2) Determining candidate patterns: For each window, Algorithm 1 generates a list of candidate patterns that *might be* maximally repeating in the window (Lines 6, 7). Theoretically, a pattern can be any substring of the window. However, some patterns are unlikely due to the UI structure. For example, in Figure 5, any pattern of length 3 is unlikely because that would make the C nodes heterogeneous. Therefore, we prune patterns that partially overlap with multiple subtrees. For efficiency reasons, while traversing the tree bottom-up, we only consider patterns that are larger than previously tested patterns (larger patterns subsume smaller ones), and that are smaller than a max pattern size ($maxP$).

(3) Maximal repeating pattern: As we traverse the tree bottom-up, we evaluate windows of increasing size (window is given by all leaf nodes of the current nonleaf node). For each window, we find the maximal repeating pattern (Lines 3–5). If the normalized expected frequency of the pattern is above a threshold (ϕ), we accept it as the pattern for the window. The accepted pattern subsumes subtree patterns.

Example. Consider the portion of the UI tree of a dining app in Figure 5. We can assign 4 possible entity types: cuisine, restaurant, address, and unknown. When traversing the tree bottom-up, at node C the MaximalRepeatingPattern algorithm considers patterns of size 2 (patterns of size 1 have been tested at node T, and patterns of size 3 are excluded because they would make the tree heterogeneous). Assuming the first two leaf nodes T in the window have uncertain entity vectors (unknown,cuisine,restaurant,address) which are (0.4,0,0.6,0) and (0.45,0.05,0,0.5), respectively, for a pattern of size 2 the algorithm computes the matching probability of all possible uncertain sequences, including restaurant-cuisine, restaurant-address, restaurant-unknown, etc. (i.e., 0.03,0.3,0.27). restaurant-cuisine has matching probability smaller than θ_1, so it is dropped; restaurant-cuisine and restaurant-unknown have similar probabilities. Once all nodes are processed, a maximal recurring pattern can be found. If so, it helps disambiguate between uncertain entity types with similar probabilities (such as address (0.5) and unknown (0.45) of the second leaf node were in this example).

4.3 Generating Templates

Each template is an annotated UI tree, with each node type mapped to an entity type. The output of the previous algorithm is a UI tree containing leaves annotated with node type-entity tuples, and nonleaf nodes annotated with patterns. To generate templates, we traverse the UI tree top-down. If at any node, we find a pattern, then we assign that pattern to all leaves of the node in question. For example, assuming that in Figure 5 the algorithm is currently processing node L, and assuming that MaximalRepeatingPattern algorithm annotated node L with the pattern restaurant-address, then all leaves of L will be assigned that pattern, as shown in the figure. If there is no pattern, we assign the entity type with the maximum probability (based on the uncertain entity vector) to the leaf node.

4.4 Identifying Relationships

Entities in a page can be related. For instance, in a music app the page showing a song's details will probably show the song's *title*, *artist*, and *album*. Entity templates also encode entity relationships. As shown in the second section of the template in Figure 4, the two node types are related, since both of them have the same group identifier 1.

To identify entity relationships, we observe that related entities in a page exhibit spatial locality in the UI tree. Therefore, we cluster node types according to their pairwise geodesic distance (i.e.,

the number of edges in a shortest path connecting them) in the UI tree, such that related items are in the same cluster. Our clustering algorithm uses two insights. First, *a cluster contains only one node type of each entity type*. For example, a cluster may contain one node type with a song's title and one node type with an artist name, but not two node types with song titles. Second, the *relationship between node types is* symmetric. For example, if a song's title T is related to an artist name A, then A is also related to T.

Given a UI tree U, we first create an undirected graph G with all leaf nodes in U. Initially, this graph has no edges. We add an edge between nodes n_1 and n_2 if the aforementioned constraints are satisfied: i) their entity types are different, and ii) n_1 is n_2's nearest neighbor of entity type $n_1.entityType$ and vice versa. For example, if the entity type of n_1 is RestaurantName, and if the type of n_2 is Address, then an edge is added between these two nodes only if n_1 is the closest RestaurantName among all restaurant names to n_2, and n_2 is the closest Address among all addresses to n_1. Once we have constructed G, we label each connected component of G as one cluster by giving each leaf node the same group identifier. In graph theory, a connected component is a subgraph in which all vertices are connected to each other. This property ensures the symmetricity condition discussed above.

5. APPSTRACT SYSTEM AND USE CASES

Appstract is our implementation of the hybrid system design described in §3, currently running on Windows Phone (WP) and Android. The ability of extracting app content semantics enables two classes of capabilities, broadly classified into *context-driven* and *user-history* capabilities. An example of the former is the system firing notifications upon specific user actions, say booking of a hotel, to allow map and calendar apps to self update. An example of the latter is the system recording types, values, and metadata of contents the user repeatedly consumes (e.g., songs by certain artists) to suggest news or concerts. We describe the Appstract APIs that support such capabilities, and the 8 use cases we built.

5.1 Appstract Abstractions and API

Appstract provides three data abstractions: *entities*, *actions*, and *entity groups*. Entities capture "what" content a user consumes. The entity object includes a type and a name (e.g., <restaurant_name,"Nara">). Actions such as tap events on buttons with labels such as "Like", "View-menu" capture "how" users interacted with the entities. From a system point of view, actions are special types of entities with type=action (e.g., <action,"Book-table">). Entities and actions are extracted through template matching. An entity group consists of one main entity, several related entities and actions (aggregated using entity group IDs), and various metadata:

```
EntityGroup:<id,timestamp,type,name,related_entities,
     related_actions,num_taps,time_spent,UIinfo>
UIinfo:<app_name,app_page,isTapped,ui_coord,ui_trace>
```

Actions and metadata can be used as *indicators of interest*, similar to those used for web pages [44, 42]. For instance, clicks on a "Like" button or on a "Call number" launcher indicate user interest in the corresponding entity. Time spent and frequency of tap events on the page containing the entity can also be mined to infer user interest. UIinfo contains information about where the entity was captured (app, page class, UI element coordinates), whether it was clicked, and the sequence of UI events that led to the page containing it (ui_trace). This trace can be used to automatically replay user interactions, as in the *App-tasker* use case (§5.2).

getCurrentView() → view	get content on screen
getEntities(view) → [entity]	
getActions(view) → [action]	
getName(entity|action) → name	e.g,. "Coldplay"
getType(entity|action) → type	e.g.., "artist_name"
getGroupId(entity|action) → groupId	used to group entities
subscribeToView(listener,filter)	filter is optional
unsubscribeFromView(listener)	
queryEntityGroup(query) → cursor	SQL-like query

Table 3. Summary of Appstract API.

Appstract offers two types of API (Table 3): 1) Online, OS and apps can request the content currently displayed on screen or can request to be notified when specific user actions or types of entities occur. Through this API, it is possible to obtain a kind of snapshot (called View) into the user's current activity. 2) EntityGroups are stored in the Appstract Entity Store that exposes an SQL-like interface. The Entity Store is currently a local database, but it could also be designed as a distributed store where old EntityGroups are offloaded to a store in the cloud and EntityGroups stored on other devices of the same users are synchronized. The following is an example of a query to find out about user cuisine preferences (used in the "Recipes By Restaurants" app, §5.2). It returns cuisine types of entities (i.e., restaurants) that the user added to their Favorites, ordered by occurrence.

```
SELECT Appstract.METADATA.CUISINE, COUNT(ENTRY_ID) AS TOT
FROM Appstract WHERE Appstract.ACTION LIKE 'AddToFav'
GROUP BY CUISINE
ORDER BY TOT;
```

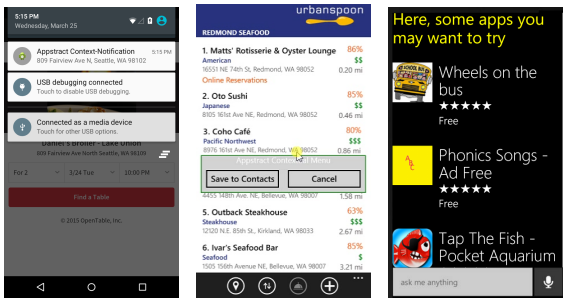
5.2 Implemented Use Cases

We implemented eight use cases which we describe in the following. The consumer of app semantics can be the OS (1-4), a first party app (5-6), or a third party app (7-8).

1. Context Notifications (Android): A reserved restaurant's name (detected by "Book-table" actions) is an example of "relevant" entity. The OS subscribes to the current View with filters on the interested events and/or entities, and fires timely notifications for other apps to consume (see Fig. 6(a)). Google Maps caches the restaurant address, and offers navigation at a later point.

2. Contextual Menu (WP): This is a library that provides context-sensitive pop-up menus in an app with zero developer effort. When a user selects a song name or a restaurant name, the library pops up a menu with options to purchase the song or to call the restaurant, respectively. Appstract notifies the menu service when a UI element is tapped and provides the type of the entity contained, such that the right menu for such content can be displayed. Fig. 6(b) shows a menu built using the SaveContactTask API provided by the WP API. We built these menus by instrumenting app binaries, but the OS would ideally generate them.

3. RankedList and 7. MusicNewsReader (Android): RankedList is a UI Element that allows app developers to quickly build personalization into their apps. From the developer perspective it is as *simple* as creating a List UI Element (no SQL queries, no ranking logic). From a functionality point of view, it is *powerful* because it enables ranking of objects using semantics information extracted from multiple apps of different categories. MusicNewsReader uses this API for ranking music news based on a user's



(a) ContextNotif (b) ContextMenu (c) Cortana-Apps

Figure 6. Some of the apps implemented using Appstract.

"favorite" music artist, defined as "most frequently-viewed artist" (ranklist.orderBy(TYPE_MUSIC_SONG,TYPE_MOST_FREQUENT)).

4. App-tasker (Android): Imagine a user asking to "Play Music." App-tasker queries the Entity Store for the ui_trace associated with TYPE_SONG objects, and infers: 1) an app for the task, 2) the sequence of UI events for the task. Using such a task template, the service can guide the user or complete the task automatically. To replay UI events, we extended the Android Accessibility Service.

5. Cortana-AppsForYou (WP): We extended Cortana to recommend apps based on music interests. For example, the user has repeatedly selected the Toddler station in Pandora, so Cortana queries the Entity Store for recently-listened music and infers her interest in kids-related apps (Fig. 6(c)).

6. Dinner Memex (Android): Mashup of info (recipes, music, etc.) collected from various apps about a dinner plan. The developer specifies a mashup object (with entity types), and this is automatically populated with matching entities.

8. Recipes-by-Restaurant (WP): We modified the Food & Drink recipe app (by adding 6 lines of dalvik bytecode) to rank recipes based on preferred restaurant cuisine types inferred from the Yelp app (we used the SQL query in §5.1).

5.3 Implementation

We implemented Appstract for Android 5.0 and Windows Phone (WP) 8.0, and tested it with apps in the food, dining and music categories. Appstract's cloud-side uses Monkey from the VanarSena [29] (WP) and PUMA [18] (Android) systems to crawl app contents. Entity template generation code is the same for WP and Android apps. For entity extraction, we used dictionaries for 9 entity types (Table 4) that were found in the app categories we selected.

Android. We implemented Appstract as a type of accessibility service. Appstract registers for the TYPE_VIEW_SELECTED, TYPE_VIEW_CLICKED and TYPE_WINDOW_CONTENT_CHANGE event types to capture app contents and user actions. The entity store is a SQLite database.

WP. WP does not provide an extensible accessibility service like Android. Therefore, to limit overhead to only apps that use Appstract, we used AppInsight binary instrumentation [30]. We overcame two challenges while implementing on WP—(1) We can only extract text when the page is stable. Therefore, we tracked all asynchronous page additions using ProcessingComplete events of AppInsight; (2) Text extraction occurs on the UI thread and can slow down the entire user experience. We tested several optimizations until we achieved acceptable performance (§6.2).

Apps	Entity types	Dictionaries # Entities	Dictionaries Source
Pandora, Spotify, FreeMP3Music	music genre, song title, artist name	1,636,582	XBoxMusic
TripAdvisor, OpenTable, Yelp, Urbanspoon	address, cuisine, restaurant name	267,051	Yelp
FoodSpotting	address, recipe name, restaurant name	301,594	Yelp, BigOvenAPI
AllRecipes, Epicurious, 111Cocktails, BigOven Allthecooks, PizzaPointer, BettyCrockerCookBook, CocktailGenie	recipe name, cocktail, dish type	35,396	Wikipedia, BigOvenAPI
YumvY	cuisine, recipe name, dish type	35,442	BigOvenAPI

Table 4. Apps, entities and dictionaries used in the evaluation.

6. EXPERIMENTAL EVALUATION

We have already discussed the privacy benefits of our approach in §2. Here we focus on evaluating the accuracy of automatic template generation (which ultimately makes our approach scalable), and the client-side performance and overhead. We use apps in the food, dining and music categories. We expect similar observations to hold for categories such as movies, books, hotels, etc.

6.1 Automated Entity Template Generation

To evaluate template generation, we created a dataset of 104 WP apps based on popularity from the top 200 apps in the food, dining, and music categories. We expect similar results for Android apps because UI structures tend to be similar.

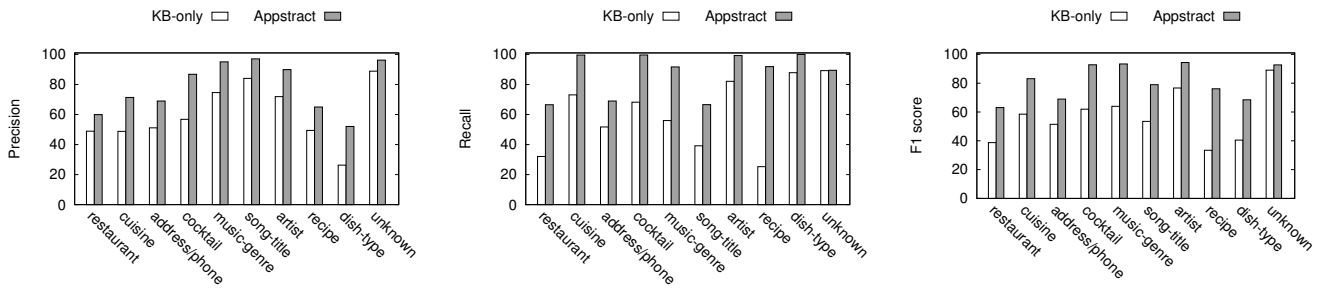
Generation speed. We run each app using Monkey for a maximum of 20 minutes, and then feed the Monkey logs into entity extraction. For each test app, we extract entities of types listed in Table 4 based on the app category in the store.[4] Generating entity templates takes on average 10.3 seconds per app. This low execution time allows us to scale template generation to all apps in the store, and to even refresh templates on a daily-basis (e.g., due to app updates).

Template accuracy: entity types. We consider two algorithms. Our baseline, *KB-only*, uses the knowledge base (similarity and word length features as described in §4.1). *Appstract* is *KB-only* plus the optimizations based on structural analysis of UI trees (§4.2). We use a random subset of 17 WP apps (Table 4) out of the 104 apps above. We run each app using Monkey and generate templates. We compare the entities extracted by the two different algorithms with *ground truth templates*, compiled manually by observing the content of hundreds of UI elements from several Monkey runs. (Hence limiting the apps tested to 17).

For both algorithms, we measure (1) *precision:* the fraction of correctly labeled UI elements across all those labeled with the same entity type, (2) *recall:* the fraction of correctly labeled UI elements across all UI elements of the same true entity type (based on ground truth), and (3) *accuracy* or F_1 *score*: the harmonic mean of precision and recall, i.e., $2 \times \frac{precision \times recall}{precision + recall}$.

Figure 7 reports the results for different entity types, aggregated across all apps. The results show some key points. First, our optimizations significantly improve the performance of entity extraction in mobile apps. *Appstract* outperforms *KB-only* on every single entity type. The average precision, recall, and F_1 score of *KB-only* are 60%, 60%, and 57% respectively, while those for *Appstract* are 78%, 87%, and 81%. *Appstract*'s entity extraction performance is comparable to the state-of-the-art for web documents (e.g., [41]).

[4]Restaurant names, song titles, and artist names are treated as low-ambiguity entities (exact match); the rest as high-ambiguity (tf-idf). Other parameters introduced in §4 are $\theta_1 = 0.01$, $\theta_2 = 0.3$ and $max_P = 8$.



(a) Precision (b) Recall (c) F_1 score

Figure 7. Performance of *KB-only* and *Appstract* template generation algorithms across all 9 entities (17 apps).

Second, the performance of both algorithms varies across entity types. Considering the F_1 score, entity types with a relatively small number of possible values, such as cuisine type, music genre, and artist name perform relatively well. This is because many of the entities can be recognized by directly consulting the KB. However, entities with a large number of potential values (song title, address, restaurant name, and recipe name) are more difficult to recognize, especially if the KB is small.

Third, while *Appstract* provides better accuracy than *KB-only* for both classes of entity types (with a small or large number of values), *Appstract*'s benefit is more prominent for the second class. *Appstract* has 37% better average accuracy than *KB-only* for cuisine type, music genre, and artist name, while it is 68% better for song title, address, restaurant, and recipe name. This is because *Appstract* has more opportunities to disambiguate entities of the second class.

Finally, larger dictionaries can improve *Appstract*'s accuracy, as in the case of music-related entities.

Overall, these results show how our approach provides reasonably accurate templates, and, because it is automated, how it can scale to many types of apps with little effort.

Template accuracy: entity relationships. We evaluate whether our clustering algorithm is accurate. For each of the 17 WP apps, we manually identify entity groups and compute the number of edges in each one. On average we observe 29.2 edges per app. When the input UI tree is the ground truth template annotated only with entity types, our algorithm achieves 100% precision (all output edges are correct) and 91.5% recall (some edges are missed). Most errors occur with FoodSpotting and Spotify, and are due to entity type encoding. While we treat address and phone as the same entity type, FoodSpotting treats them separately such that these entities have separate edges to a restaurant entity. Similarly, while we assume song and album name to be the same entity type, Spotify does not. With the introduction of two additional entity types, the recall is 95.7%.

Comparison with StanfordNER and KB-only. We now evaluate the "final" accuracy of Appstract-generated templates to extract entities from app pages, in the same way we evaluated StanfordNER in Table 2. Figure 8 compares Appstract to StanfordNER (our baseline for web entity extraction), and to the KB-only algorithm (§4.1).[5] Appstract outperforms both, on all entity types. On average, StanfordNER achieves an F1 score of 41%, while Appstract's F1 score is 79%. In many cases, Appstract's F1 score is more than double. A main reason for the performance improvement

[5]We use KB-only and Appstract-generated templates to extract entities from a set of 14 apps with 4657 page instances (the same set used for StanfordNER). Three apps are not used because not enough training data exists for StanfordNER.

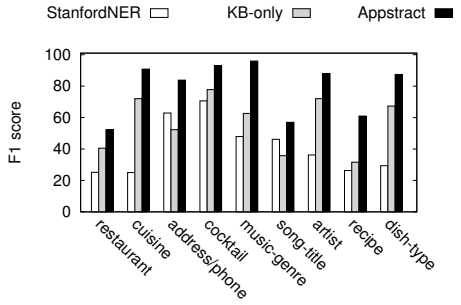


Figure 8. F1 scores when templates generated using StanfordNER, KB-only, or Appstract are used to extract entities from 14 apps.

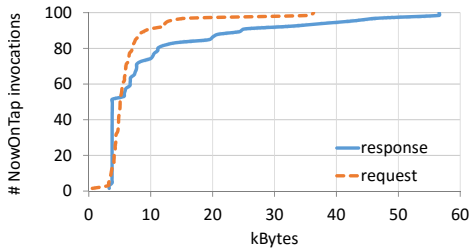


Figure 9. CDF of Now On Tap request (dotted orange line) and response (solid blue line) sizes in kB.

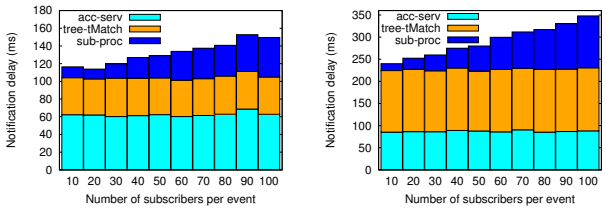
is Appstract's structural analysis of UI page structures that reduces ambiguity in closely-related entity types. Although KB-only performs better than StanfordNER, Appstract still outperforms it due to structural analysis.

6.2 Client Performance and Overhead

We compare Appstract to a cloud approach in terms of network bandwidth, latency, energy consumption, and storage.

Network bandwidth. Appstract uses the network only to download app templates—roughly 400 kB (without compression) per app. It does *not* require network communication at runtime. This is a significant overhead reduction compared to cloud-based systems which consume bandwidth to upload app data for semantics analysis. To get a rough estimate, we use Now on Tap. We pick 33 top Android apps, and measure the bandwidth consumed when invoking Now on Tap on two unique pages (see Figure 9). Now on Tap sends 6.8 kB on average to the cloud (the request packet size is always larger than 3 kB, and up to 37.2 kB maximum).[6]

[6]The size of the response depends on the specific application, the cards of Now on Tap in this case, so it is not comparable to the Appstract scenarios.



(a) Pandora (b) Urbanspoon

Figure 10. Appstract notification delay.

Latency. To evaluate latency, we measure the Appstract delay between a UI event (e.g., click on a button) and the notification with the extracted entity types firing. We run Appstract on a LG Nexus 4 running Android 5.0.[7] We use Pandora (music) and Urbanspoon (restaurants). We create from 10 to 100 subscriptions for the actions "book-table" (Urbanspoon) and "thumb-up" (Pandora), and we use *monkeyrunner* [4] to automatically generate 100 UI events of these types, every 6 seconds. A 100 subscriptions for the *same type* of event (i.e., 100 services interested in the same event) is a realistic upper bound. Figure 10 reports the average notification delay. The breakdown of the total delay includes: (1) *acc-serv*, the time the accessibility service takes for processing the UI event and invoking callbacks, (2) *tree-tMatch*, the time for parsing the captured UI tree and matching the entity template, and (3) *sub-proc*, the time for matching subscriptions and firing notifications. The overall delay in Pandora increases linearly from 126 ms (10 subscriptions) to 150 ms (100). The increase, due to *sub-proc*, is proportional to the number of subscriptions. In Urbanspoon, the delay is roughly 2 times higher (242–350 ms) because the loaded Urbanspoon page has about twice the UI elements of the Pandora page (191 vs. 88). 100–200 UI elements is the typical range of UI elements we have observed, so given that users may generate at most one UI event per second, Appstract processes UI events in a timely manner.

The latency of a cloud approach is much higher because of the network transfer. Using the same 33 apps above, we measure the RTT of Now on Tap over WiFi to be on average 1.7 seconds (SD = 2.4). We expect the cellular network RTT to be larger—Sommers et al. show that cellular latency is often a factor of two greater than WiFi latency [35]. However, a cloud-based approach provides a super set of the Appstract functionality.

Energy and computation overhead. Appstract consumes energy due to the local processing. We evaluate the processing overhead using both our phone prototypes. For *Android*, we use the CPU-intensive NEON MP MFLOPS benchmark [33], and measure its slowdown due to Appstract. We generate a workload of UI events by interacting intensively with Urbanspoon for 3 minutes, and recording the UI events using RERAN [15]. Then, always using RERAN, we replay the UI events in three test configurations. As Table 5 shows, when there are no subscriptions, Appstract processes UI events, extracts entities/actions, and computes entity groups with an overhead of only 1.7%. Overhead increases as online subscriptions grow.

WP does not expose an accessibility service for tracking UI events, so we use binary instrumentation to capture page contents. The instrumentation affects the UI thread, so we measure the overhead as the additional time, compared to an uninstrumented app, a UI takes to stabilize since the user interaction that triggers the UI change. We use the original and instrumented versions of Urbanspoon and Pandora on a Lumia 820 running WP 8.0, for 10 minutes.

[7] On WP, due to the siloed nature of the apps, communications occur through the cloud, so we cannot realistically measure this delay.

Case	MFLOPS	% Slowdown
Baseline	4655.3 (31.9)	0.0
Appstract	4576.0 (163.8)	1.7
Appstract+100sub	4435.0 (111.5)	4.7

Table 5. Appstract overhead on Android 5.0. Average, variance and % slowdown of a CPU-intensive benchmark.

The average overhead, measured as the percentage slow down (in loading or updating a page, depending on the UI event) is 1.8%. The overhead increases with the complexity of the UI layout of the app pages, with the worst-case overhead we observed being 3.5%.

To fully compare Appstract to a cloud-based approach in terms of energy consumed, we ran the following experiment. We select 5 apps (Zomato, Pandora, TripAdvisor, MyBar, MSN Food and Drink), and interact with them for 2 hours. Using the record-and-replay HiroMacro [26] tool, we record a trace of the UI events triggered during interaction including UI tree contents. We later replay the UI event trace in two conditions: 1) *Appstract*, where our service running in the background processes page UI trees at each new UI event, and 2) *Cloud-based*, where a background service makes a network request at each UI event to a cloud server which responds with a packet containing the size of bytes received. To minimize network usage, the service only sends the delta between two subsequent UI trees. This cloud-based approach does not exactly simulate Now-on-Tap since our cloud service does not do entity extraction, but we expect client's energy consumption to be due mostly to network activity so it gives a good estimate. In both conditions, we use 3G with a Nexus 4 charged to 100%, and sample the battery level at regular intervals. The Appstract condition ends with 75% battery remaining, whereas, the cloud-based condition ends with 74% battery remaining. The cloud-based approach does not consume significantly more energy than Appstract because packets are transmitted while apps are running. Therefore, the radio is active most of the time (i.e., no 3G tail effect), and any data transferred due to semantics analysis can be piggy-backed onto ongoing app transfers. Appstract's energy consumption is comparable (or less) because, as shown above, its processing overhead is negligible (and Appstract does not use the network at runtime).

Storage overhead. Finally, while the cloud-based approach has no storage overhead, Appstract stores extracted entities locally. To get a rough estimate, we use data collected through the deployment of the WP prototype to 11 users for 8 weeks, where we observed entity groups of 1.5 kB (without compression). We consider two of our WP users: one is a regular app consumer who produced an average of 19.6 entities per week, per app; the other is an occasional user who generated only 1.2 entities per week, per app. We estimate that the monthly storage overhead for 10 Appstract-monitored apps is 1.2 MB and 72 kB, respectively. Projecting these numbers to 50–100 apps over a year, the upper bound is 780 MB–1.6 GB. Data compression can reduce the sizes, and further reduction is possible by periodically summarizing the data to inferred user interests (e.g., user likes Chinese food), and discarding (at least locally) the raw data. So we conclude that these storage requirements are reasonable.

7. DISCUSSION AND LIMITATIONS

App scope. We have focused on food, dining and music app categories, however, Appstract is applicable to communication and social network apps (people, organizations, places), travel apps (flights, hotels), and shopping and entertainment apps (movies, books, purchased items). Games, tools, productivity, and weather apps are less interesting from an entity extraction point of view.

Free-form text-based apps, such as email and office apps, fall in the category of unstructured content apps so they require deeper text analysis (possible with existing web techniques through a cloud-based approach).

Entities in images. Appstract offers a more privacy-respecting approach for extracting semantics of in-app structured text, but it does not recognize entities in images. Image-based entity recognition could be supported if a suitably trained model were available. Such a trained model could be created in the cloud with user-agnostic data in much the same way Appstract uses Monkey-generated data to create templates, and then it could be sent to the device for semantics extraction, ensuring that the images a user is viewing do not leave the device. However, the challenge here is to obtain a large enough labeled training dataset to achieve a well-trained model. Furthermore, if images were organized in a UI tree (like a grid view of photos, for example), Appstract's techniques could be used to boost accuracy by leveraging patterns in UI trees.

Incomplete app text crawling: Appstract's automated process for generating entity templates relies on a corpus of data collected using Monkey (or similar UI automation tools). Such tools do not provide 100% coverage of an app's pages so the accuracy of our approach might be affected. Ravindranath et al. conducted a study of Monkey coverage of unique page classes, and found that in a set of 35 randomly selected apps Monkey has a page class coverage of 100% for 26 apps, with 90% of apps being explored fully under 365 seconds [29]. However, Appstract requires good coverage of page instances as well. On the other hand, as long as Monkey achieves good coverage of unique page classes, the Appstract in-page pattern analysis can increase accuracy of entity extraction, especially in cases where few instances of the same page class are available.

Privacy and access control. The trusted OS must regulate access to semantic data extracted from the apps. An all-or-nothing model, where an app has access either to the entire Entity Store or to nothing, is unlikely to work. Extracted data carries information of varying sensitivity, so finer-grained control is preferable. An all-or-nothing model would also not be compatible with privacy frameworks like Contextual Integrity [24, 8]. Ultimately, extracted entities can be treated as system resources, and protected by the same model current mobile OSes use to control access to resources like sensors and network. Users could grant permissions to apps, separately, per entity category. For instance, Pandora and Spotify both produce entities of category SongTitle, and an app with permission granted to access entities of category SongTitle might access all such entities produced by *any* app on the device. While permissions are essential, research shows that in mobile apps they are not sufficient for protecting user data [14, 20]. Orthogonal techniques are applicable here [13, 6].

Data ownership. Who owns app usage data? If a developer owns all usage data in her app and if there is little incentive to share this data with other apps, a developer might not want the OS to extract it. On the other hand, if the user owns her data, she may want the OS to use it for on-device services or even across apps for her own benefit. While we can address some concerns of app developers (e.g. app opt-out, access control), further discussion is required that we hope this paper will initiate.

8. RELATED WORK

Techniques for web entity extraction and for task extraction, as well as general analytics frameworks are related to our work.

Named entity recognition in the web. There are four broad categories [22, 34]: rule-based, supervised [7, 21], semi-supervised [31, 3, 25], and unsupervised [11, 19, 41]. Our approach falls in the last category. We exploit cross- and in-page structural similarity in mobile apps to overcome the challenge of insufficient context in apps. Google Now on Tap and Bing Snapp [16, 40] use their backend for semantic analysis. In contrast, Appstract enables *continuous* and *efficient* entity type extraction of app contents, using on-device entity templates. Unlike Google and Bing, Appstract's focus is not on extracting "entities," but on extracting entity types and relationships. Google and Bing could use Appstract templates to improve the privacy, and possibly the accuracy and the efficiency of their entity extraction pipeline.

Entity recognition in the web for tweets is closely related to our work due to the lack of context in both, tweets and mobile apps. Ritter et al. show how existing entity recognition tools like StanfordNER do not perform well in tweets, and develop a distantly supervised approach which uses a dictionary of entities and LabeledLDA models [32]. They observe an F1 score of 67% (an increase of 52% over StanfordNER) by leveraging a custom-built NLP pipeline (part-of-speech tagging, shallow parsing, capitalization), which is designed to take advantage of the specific structure of tweets. Similarly, Appstract is designed to take advantage of the structure of mobile apps to achieve an accuracy of over 80%.

Task extraction. Insider [10] builds app task models for task completion. Insider's templates are derived from tasks such as "search for a song by artist in music-genre" which are defined manually. Moreover, templates are generated using crowdsourcing. In contrast, 1) Appstract generates templates *automatically*, and 2) captures entity types, entity relationships and user actions. Insider focuses on extracting user context, whereas, Appstract also builds a history of user behavioral data. Finally, Appstract is a general framework that enables a range of new experiences (§5.2).

App and web analytics. AppInsight [30], Localytics [2], Flurry [1] and Google Analytics [17] report app performance and usage statistics. These systems do not report semantics of app usage and hence are not comparable to Appstract. For instance, Flurry can log a click event, but cannot report that it is on a call button associated with a restaurant. Google Analytics [17] is a widely used on-site analytics service for web personalization [12]. It is implemented with "page tags," JS code that site owners *manually* add to their webpages. Unlike Appstract, it provides aggregated analytics across users, but not semantic information of a single user's activity.

9. CONCLUSIONS

We have proposed an architecture and a set of techniques for continuously extracting semantically-meaningful content from mobile apps efficiently and on-device. Our approach does not send private usage data to the cloud, so it can provide better privacy than a cloud-based approach adopted by existing systems. Key to our approach is the ability to automatically generate accurate entity templates offline. We tested apps in the food, dining, and music categories and achieved accuracy over 80%. However, Appstract is applicable to many other categories with structured content: communication, travel, shopping, entertainment, etc.

Acknowledgements

We thank the anonymous reviewers and our anonymous shepherd for their insightful feedback on our work. We thank Kaushik Chakrabarti and Atul Prakash for discussions that improved this paper. We also thank Lenin Ravindranath for his help with AppInsight. Earlence Fernandes was partly supported by the Microsoft Research internship program during the course of this work.

References

[1] Flurry. http://www.flurry.com.

[2] Localytics. http://www.localytics.com/.

[3] E. Agichtein and L. Gravano. Snowball: Extracting Relations from Large Plain-text Collections. In *Proc. of the 5th ACM conference on Digital libraries (DL '00)*, pages 85–94, 2000.

[4] Android Developers. monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[5] AppBrain. Top 10 Google Play categories. http://www.appbrain.com/stats/android-market-app-categories.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of PLDI '14*, pages 259–269. ACM, 2014.

[7] M. Asahara and Y. Matsumoto. Japanese named entity extraction with redundant morphological analysis. In *Proc. NAACL*, pages 8–15, 2003.

[8] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and Contextual Integrity: Framework and Applications. *2013 IEEE Symposium on Security and Privacy*, 0:184–198, 2006.

[9] K. Benton, L. J. Camp, and V. Garg. Studying the effectiveness of android application permissions requests. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, pages 291–296, March 2013.

[10] V. Chandramouli, A. Chakraborty, V. Navda, S. Guha, V. Padmanabhan, and R. Ramjee. Insider: Towards breaking down mobile app silos. In *TRIOS Workshop held in conjunction with the SIGOPS SOSP 2015*, September 2015.

[11] M. Collins and Y. Singer. Unsupervised models for named entity classification. In *Proc. of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 100–110, 1999.

[12] M. Eirinaki and M. Vazirgiannis. Web mining for web personalization. *ACM Trans. Internet Tech.*, 3(1):1–27, 2003.

[13] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of OSDI '10*, pages 393–407, October 2010.

[14] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Proc. of SOUPS '12*, pages 3:1–3:14. ACM, 2012.

[15] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and Touch-sensitive Record and Replay for Android. In *Proc. of ICSE '13*, pages 72–81, 2013.

[16] Google. Now on Tap. https://support.google.com/websearch/answer/6304517?hl=en.

[17] Google Analytics. http://www.google.com/analytics/.

[18] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps. In *Proc. of MobiSys*, pages 204–217. ACM, June 2014.

[19] J. Hoffart, M. A. Yosef, I. Bordino, H. Fürstenau, M. Pinkal, M. Spaniol, B. Taneva, S. Thater, and G. Weikum. Robust disambiguation of named entities in text. In *Proc. of EMNLP*, pages 782–792, 2011.

[20] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In *Proc. of USEC '12*, pages 68–79, 2012.

[21] A. McCallum and W. Li. Early Results for Named Entity Recognition with Conditional Random Fields, Feature Induction and Web-enhanced Lexicons. In *Proc. of CONLL*, pages 188–191, 2003.

[22] D. Nadeau and S. Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, Jan. 2007.

[23] S. Nath, F. X. Lin, L. Ravindranath, and J. Padhye. SmartAds: bringing contextual ads to mobile apps. In *Proc. of MobiSys*, pages 111–124, 2013.

[24] H. Nissenbaum. A Contextual Approach to Privacy Online. *Daedalus 140*, (4):32–48, Fall 2011.

[25] M. Pasca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Organizing and Searching the World Wide Web of Facts - Step One: The One-Million Fact Extraction Challenge. In *Proc. of AAAI*, pages 1400–1405, 2006.

[26] ProHiro. HiroMacro Auto-Touch Macro App. https://play.google.com/store/apps/details?id=com.prohiro.macro&hl=en, 2016.

[27] Pyevolve. Machine Learning - Text feature extraction (tf-idf). http://pyevolve.sourceforge.net/wordpress/?p=1589, 2014.

[28] L. Ratinov and D. Roth. Design challenges and misconceptions in named entity recognition. In *Proc. of the 13th Conference on Computational Natural Language Learning*, CoNLL '09, pages 147–155, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[29] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proc. of MobiSys*, pages 190–203, 2014.

[30] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proc. of OSDI*, pages 107–120, 2012.

[31] E. Riloff and R. Jones. Learning Dictionaries for Information Extraction by Multi-level Bootstrapping. In *Proc. of AAAI*, pages 474–479, 1999.

[32] A. Ritter, S. Clark, Mausam, and O. Etzioni. Named entity recognition in tweets: An experimental study. In *Proc. of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 1524–1534, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[33] Roy Longbottom. Android NEON MP MFLOPS Benchmark. http://www.roylongbottom.org.uk/android\%20neon\%20benchmarks.htm.

[34] S. Sarawagi. Information extraction. *Found. Trends databases*, 1(3):261–377, 2008.

[35] J. Sommers and P. Barford. Cell vs. WiFi: On the Performance of Metro Area Mobile Connections. In *Proc. of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 301–314, New York, NY, USA, 2012. ACM.

[36] Stanford University. CoreNLP: Named Entity Recognizer. http://nlp.stanford.edu/software/CRF-NER.shtml.

[37] Stanford University. How do you use gazettes with Stanford NER? http://nlp.stanford.edu/software/crf-faq.shtml#gazette.

[38] Stanford University. NLP Named Entity Recognition Results on CoNLL tasks. http://nlp.stanford.edu/projects/project-ner.shtml.

[39] S. Tata and J. M. Patel. Estimating the Selectivity of Tf-idf Based Cosine Similarity Predicates. *SIGMOD Rec.*, 36(4):75–80, Dec. 2007.

[40] VB. Microsoft beats Google to the punch: Bing for Android update does what Now on Tap will do. http://venturebeat.com/2015/08/20/microsoft-beats-google-to-the-punch-bing-for-android-update-does-what-now-on-tap-will-do, 2015.

[41] C. Wang, K. Chakrabarti, T. Cheng, and S. Chaudhuri. Targeted disambiguation of ad-hoc, homogeneous sets of named entities. In *Proc. of WWW*, pages 719–728, 2012.

[42] R. White. *Implicit Feedback for Interactive Information Retrieval*. PhD thesis, 2004. University of Glasgow.

[43] WSJ.D. Why You Should Care About Google Now on Tap. http://blogs.wsj.com/personal-technology/2015/05/28/why-you-should-care-about-google-now-on-tap/.

[44] N. Zemirli. WebCap: Inferring the user's interests based on a real-time implicit feedback. In *Proc. of ICDIM*, pages 62–67, Aug 2012.