CommandSpace: Modeling the Relationships between Tasks, Descriptions and Features

Eytan Adar
University of Michigan
Ann Arbor, MI
eadar@umich.edu

Mira Dontcheva
Adobe Research
San Francisco, CA
mirad@adobe.com

Gierad Laput
Carnegie Mellon University
Pittsburgh, PA
gierad.laput@cs.cmu.edu

ABSTRACT

Users often describe what they want to accomplish with an application in a language that is very different from the application's domain language. To address this gap between system and human language, we propose modeling an application's domain language by mining a large corpus of Web documents about the application using deep learning techniques. A high dimensional vector space representation can model the relationships between user tasks, system commands, and natural language descriptions and supports mapping operations, such as identifying likely system commands given natural language queries and identifying user tasks given a trace of user operations. We demonstrate the feasibility of this approach with a system, COMMANDSPACE, for the popular photo editing application Adobe Photoshop. We build and evaluate several applications enabled by our model showing the power and flexibility of this approach.

Author Keywords

Deep-learning, application language domain, natural language interfaces

ACM Classification Keywords

H.5.2 [Information interfaces and presentation]: User Interfaces. - Natural language

INTRODUCTION

There is a fundamental, and well-recognized, gap between the language of an application and the language people use to describe what they want to accomplish [22]. In large and complex applications, such as Adobe Photoshop and GIMP, new features are constantly added, and the number of ways to put these features together into new workflows grows even more rapidly. End users—even experts—find it increasingly difficult to identify the right set of application commands to fulfill a task, find alternative workflows, or learn how to retarget a familiar workflow to an unfamiliar problem.

To accomplish a task, an end user will often respond in the same way: by searching for Web pages—be they manuals, tutorials, forums, or question answering sites. Unfortunately, leveraging this collective knowledge is hampered by restrictive tools. Conventional search engines allow an end-user to express a query, refine that query when the results are not suitable, dig through the results to find the right information, and repeat the process again—an inefficient and limited solution to the problem. Query-Feature Graphs (QF-Graphs) [7] attempt to solve this problem by directly tying popular user queries to system features, but they still rely on the underlying bipartite structure of the Web where natural language and system features are held distinct from each other.

In this work, we approach the problem from the perspective of jointly modeling system features and natural language in the same continuous vector space (i.e., each "token"—a word, phrase, or system feature—is represented by an n-dimensional vector). Specifically, we apply a "deep-learning" approach to embed natural language words and phrases (e.g., "orange" or "comic books") as well as system features (e.g., "Filter $>$ Render $>$ Clouds" or "Edit $>$ Transform Warp") in the same vector space. As we demonstrate, this approach supports a number of use cases directly. The simplest, and most natural, calculates the distance from "queries" to commands (e.g., finding all commands related to creating a comic book effect). Similarly, using a command as a query can obtain related commands or even a ranked list of tasks that often use the command.

The appeal of this approach, however, extends beyond these simple mappings. As both syntactic *and* semantic relationships are preserved in this vector space, many other possible applications emerge. For example, a workflow can be retargeted by similarity (e.g., identifying that functions useful for scar removal in a portrait are also applicable to removing a scratch on a car), or even by analogy (e.g., barrel distortion *is to* filter$>$lens correction *as* mustache distortion *is to* ?). Because these spaces model relationships at a low-level, answers to questions can be provided even if no exact match was ever described on a specific Web page (e.g., while a specific page on removing scratches from cars may not exist, the model will have learned that scars on portraits and scratches on cars are analogous). All of these applications are supported through simple vector algebra.

In this paper we describe COMMANDSPACE, a system to create and query this joint model as well as a set of demonstration applications that make use of the model. We apply COMMANDSPACE to the domain of the image editing software Adobe Photoshop by collecting and processing a corpus of

UIST '14, October 05 - 08 2014, Honolulu, HI, USA

http://dx.doi.org/10.1145/2642918.2647395

Web pages and PDF documents. Unlike prior work, this approach does not require query logs, thesauri or supervise the training in any way. Additionally, by utilizing a deep-learning model, COMMANDSPACE offers a number of desirable features: (a) it works efficiently on extremely large text datasets (we train on nearly 200 million words in a few hours), (b) it is unsupervised and requires no ground truth, (c) it supports a large vocabulary, (d) it is robust to syntactic and semantic transformations, and (e) it offers a flexible mapping between many different constituents (e.g., system features, tasks, common verbs) of an application's domain that can be easily embedded in a variety of applications.

Our contribution in this work is a mechanism for modeling an application's domain language using a deep-learning approach. We demonstrate how system features and other tasks can all be mapped into the same vector space, which preserves syntactic and semantic relations. We identify key properties of this space, demonstrating how to achieve complex tasks using simple vector operations. Finally, we build and evaluate a number of applications using the vector space. These include the standard task-to-tool mapping, which achieves a high level of precision, as well as applications to map commands to tasks, and a mechanism for identifying alternative workflows. Because the system relies on little more than a collection of text related to a domain, we believe that COMMANDSPACE can be readily applied to a variety of applications and APIs.

RELATED WORK

Tasks to commands

To bridge the language gap between user task and system features many modern applications build extensive search features directly into the applications. These can be viewed as a form of "search-driven interactions" where natural language can be used to identify relevant commands. Some companies (e.g., Adobe) couple their help systems with a broader corpus of pages created by people outside the company and offer an in-application shortcut to a Web-search engine that more rapidly ties the query to possible solutions. Others (e.g., Microsoft Office) have sought to enhance their help offering by supporting "show-me" features. Examples include highlighting matching commands in the menu, or controlling the mouse and demonstrating the action.

There are a number of natural-language interfaces that seek to simply bridge the language gap by using natural language as the system language. Commercial systems such as Apple's Siri, Android's Google Now, and Microsoft's Cortana all attempt to directly interpret a user's speech to execute system commands. Others have tried to provide natural language interfaces to applications such as databases [2], image editing tools [14], and the operating system [4]. Such systems are rapidly improving and are becoming more flexible and more powerful. Nonetheless, they are carefully engineered to support mapping human language to system language.

One approach to mapping user tasks to system commands is to use a standard search engine (e.g., search for task on a search engine and extract matching commands from retrieved documents). However, such an approach requires an extensive thesaurus to work effectively as many task variants will never be observed in the corpus. Even approaches with extensive thesauri [25] still can not easily account for the domain specific terms or other syntactic or semantic relationships. Similarly, traditional techniques that use query-log mining [27] to find associations between queries (e.g., identifying instances of refinement or expansion) are not feasible as they require data that is difficult to obtain.

The most similar approach to our own is the work on QF-Graphs [7]. QF-Graphs are built by identifying common tasks by search log mining and connecting those to system features. A bi-partite graph with common tasks on one side and features on the other is constructed by linking tasks with commands through text mining. Fourney et al. suggest a number of applications for this model including search driven interactions (e.g., search for a task and get back a list of commands), dynamic tooltips (mapping back from feature to task to display in a tooltip), and app-to-app analogy (mapping a feature in one application to a feature in another). While powerful, this technique is limited in that it does not attempt to jointly model features and tasks. Each "translation" step potentially accumulates errors. For example, to find similar commands using the QF abstraction, one needs to map commands to tasks and back to commands (with no indication about the type of relationship between the two commands). In COMMANDSPACE, commands exist in the same space and their relationship (angle and magnitude in vector space) reflect specific relationships (e.g., "alternative" or "inverse"). Furthermore, QF-graphs are focused on terms that are present in the search query logs. With our approach, we can use semantic relationships to offer suggestions for tasks that were never explicitly described in a document. For example, there may not be a tutorial for removing silverware out of a scene, but there may be tutorials showing how to mask silverware and tutorials how to remove objects. COMMANDSPACE models the relationship between these tasks using system features and can make appropriate recommendations for a task where it does not have a repository of content. Finally, QF-Graphs rely on search-engine APIs, which offer little information as to how the result list is generated or how synonyms are used.

Commands to commands

The CommunityCommands system [17] builds models of commands by utilizing ideas from collaborative filtering. Namely, the space is modeled by using a user-item representation where each end-user has a vector representation of command usage. This allows the system to identify similar users (those that utilize similar commands) and to propose non-overlapping commands to each other user. This approach effectively provides users with suggestions to unfamiliar tools. However, CommunityCommands is not designed to offer alternatives to a particular tool because these relationships are not explicitly modeled.

Commands to tasks

The Lumiere Project [10], the basis of Microsoft's Clippy, was the first system to model user behavior and make in-app suggestions. However, this approach was not intended

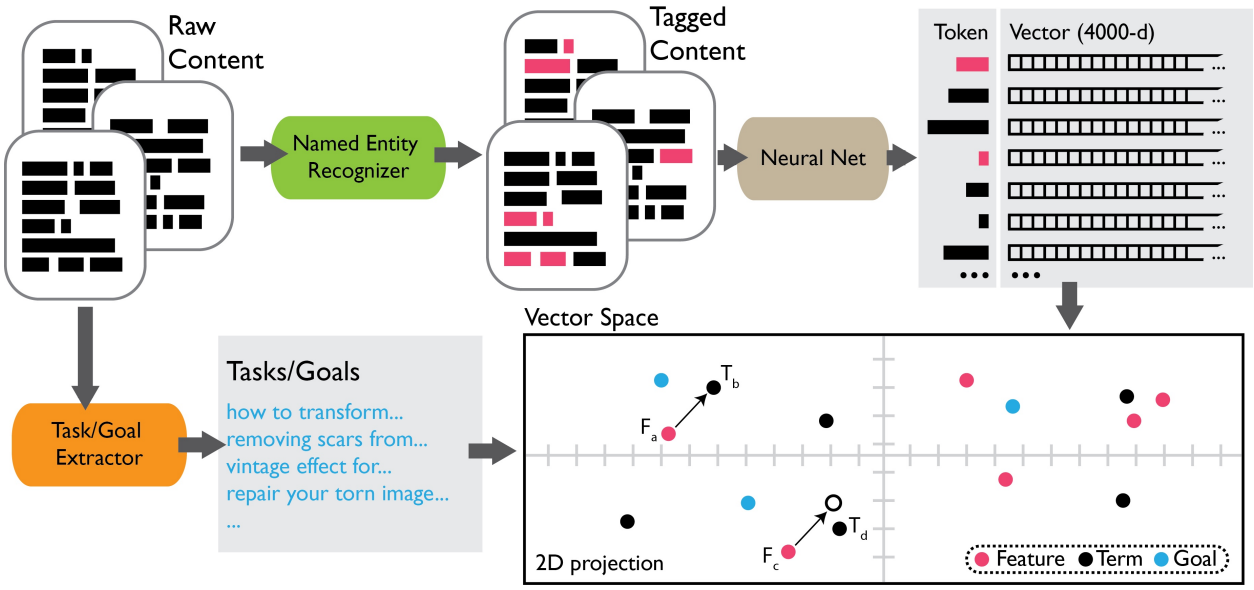

Figure 1. Here is the general architecture for COMMANDSPACE. Raw content crawled from the Web is analyzed by a CRF-based Named-Entity Recognizer (NER) and tagged. The Neural Network transforms all terms (including identified system features) into a n-dimensional vector space representation (n=4000). Simultaneously a task/goal extractor component pulls possible task descriptions from the raw content. A linear vector space allows us to compare generic terms (T), goals (G), and system features (F) in the same space, thus supporting our mapping function.

to scale to the whole of the application's language (or the end user's) but focused on narrow task prediction. While there has been some effort to identify common repeated subsequences (extensively reviewed in [8]), or to predict next commands (e.g., [6, 15]), these are fundamentally different tasks. They often rely on collected logs or traces to identify patterns and produce predictions. Our goal is to be able to support related (and next) commands by mining "found" text.

In the software engineering community a few solutions have tried to bridge API elements and error language (e.g., compiler errors) to explanatory material (e.g., [12, 9]). This material may be produced by crowd-workers or simply be highly ranked pages in a search result. This task is fundamentally different in that it is intended to support the end user in their developer role by finding relevant documentation (rather than automatically "explaining" what they are doing). That said, errors are part of the application's language domain and may lead to interesting connections between features, tasks, and errors. While our dataset includes information about errors, we did not explicitly tag errors in COMMANDSPACE. This remains an interesting direction for future work.

Distributed vector representations

Recent work on "deep-learning" has focused on expanding the techniques to text [20, 19]. Specifically, this research has utilized neural networks to learn continuous language models through large text collections. The techniques offer a mechanism for projecting text into a lower dimensional representation that preserves semantic and syntactic relationships. Conventionally, such models are trained on a text collection where tokens are single words or simple phrases. The models themselves are based on a number of "architectures," including continuous-bags-of-words (CBOW) and skip-grams. CBOW architectures predict a word given the observed words immediately before and immediately after the term (e.g., predict w_n given w_{n-1} and w_{n+1}). Skip-grams (or n-grams in which skipping words is allowable) predict the words immediately before and immediately after the observed term (e.g., predict w_{n-1} and w_{n+1} given w_n). The probabilities associated with each term are learned and represented in a lower-dimensional vector space. In our work we use these techniques to model the space of commands and text. By identifying references to commands in the original text and converting these to special tokens, command "objects" and natural language words and phrases can be simultaneously projected into the same space, thus supporting simultaneous vector transformations.

SYSTEM OVERVIEW

Figure 1 illustrates the COMMANDSPACE architecture. To train the system, we crawl text documents on the Web including step-by-step tutorials, question-and-answer websites, and discussion forums. Such documents capture both the general function of system features (e.g., "*Filter* > *Blur* averages adjacent pixel values to smooth the image") as well as contextualized use (e.g., "*Filter* > *Blur* can be used to achieve a tilt-shift effect"). A Named-Entity Recognizer (NER) based on a trained Conditional Random Field (CRF) extractor identifies system features in the raw text and replaces those with unique identifiers. The labeled data then goes into a neural-network, which models the application's domain language using a distributed vector representation. To do this modeling we leverage the *word2vec* implementation [18]. Finally, we use simple vector transformations to map natural language to system features and vice versa.

COMMANDSPACE also identifies phrases describing "goals" or "tasks" from the raw content using a simple, heuristic-based extractor. The extracted goals can be projected into our vector space (Figure 1, labeled in blue) by summing the

vectors for the constituent terms and normalizing. Thus, we can calculate similarities and find the features that are most similar to specific goals.

COLLECTING THE DATA

To create our collection of Photoshop-related text documents, we issued queries to the Yahoo! search engine through the Yahoo! BOSS API [26] for each of 1339 known "features" (these were semi-automatically extracted from localization strings for Photoshop with some manual cleaning and annotation). Features include tools (e.g., *Line Tool*), menus (e.g., *"Layer > Layer Style > Global Light"'*), and panels (e.g., *Channel Select*). Because some tools can appear in multiple applications, we included the term 'photoshop' in each query to restrict results from the search engine (e.g., *photoshop 'Line Tool'*). We retrieved up to 1000 matching results for each query yielding 187346 unique URLs that mentioned both Photoshop and at least one menu or command. Using the Nutch Web crawler [3], we crawled outward through the links of each webpage to collect 1.4M Web pages. In addition, we introduced the full text of approximately two dozen books related to Photoshop.

Identifying and labeling system features

First, we used Boilerpipe [11] to strip page template information and retain only page text. Next, we tagged tool or menu references in the text using either direct string matching for known strings (the 1339 known features) or a trained Conditional Random Field (CRF) extractor. We trained the CRF on a small set of tutorial pages using the features described in [13]. The CRF learned contextual cues indicating the presence of a string representing a system feature. For example, a button is often preceded by the phrase "click on." The CRF identified the *Clouds Filter* in both sentences: (1) "Click the Filter > Render > Clouds . . . ," and (2) "Click the Clouds option from the Filter > Render . . . ." In the first sentence the tagged phrase is "Filter > Render > Clouds" and in the second the CRF tags "Clouds option from the Filter > Render."

We compared tagged phrases to all known commands using Jaccard distance, which measures text overlap, and selected the one with the highest score. While more sophisticated variants of this are possible [23], we found that this simple approach worked well enough. We replaced each matched phrase with a unique feature ID in the raw text. In the example above, both sentences became: "Click the filter_render_clouds . . . " This unique ID was treated as a single token in the neural network, which creates a vector for each token representing a system feature.

Despite crawling 1.4M webpages, we were only able to find roughly 50% of the 1339 system features in those webpages. Qualitatively, we found that tagging was not nearly as accurate as described in [13] (largely due to the extreme diversity of language in the "wild"). While we believe we are able to capture the more popular system features, we can increase system feature coverage by dynamically creating vectors for any missing features without loss in performance. We describe this in more detail below.

Adding common natural language phrases

In addition to tagging system features, we also tag common natural language phrases. For example, we replace all occurrences of "comic books" with *comic_books*, "album cover" with *album_cover*, and "brightly colored" with *brightly_colored*. To do this tagging, we use the *word2phrase* program (part of the *word2vec* suite [20, 19]) and identify common n-grams that have been observed a significant number of times (in our case, 30).

Final tagged content

In a final step, we filtered the 1.4M set of webpages to English text pages that contained at least one system feature. We define English text pages as pages that contain at least 60% English words. The result set includes 173,567 pages with over 173M words.

MODELING THE SPACE

To perform the vector space modeling we make use of the *word2vec* implementation [18]. We experimented with a number of configurations (see Evaluation) to determine the training parameters. We resolved on a vectors size of 4000, a window of 10 words, an occurrence threshold of .001, using hierarchical softmax, and a minimum number of word occurrence of 30. Once trained, the neural network produces a set of vectors that can support a variety of mapping functions. Because both system features and natural-language text are mapped to the same space–they are both treated as "tokens"–we are able to measure distances between the vector forms of those terms to identify the best match.

Mapping terms, features and goals

Given goal statements, G (e.g., "give this image a comic book effect" or simply "comic book") we would like to identify those system features F (e.g., "halftone effect filter") that can be used to support it: $S(g_i \in G) \rightarrow F$, where S is our search function. Conversely, we would like to identify possible goals given a set of utilized features: $S(f_i \in F) \rightarrow G$. In the former case, the end users are expressing a need or goal in descriptive natural language, and we would like to isolate those system features that could support them in their task. In the latter case, we may have a partial trace of the user actions within the application, and we would like to identify the most likely goals they are pursuing. While this second type of mapping is less conventional, there are a number of compelling applications. For example, a developer could leverage this technique to label collected log traces of user behavior to understand not just which features are being used but *why*.

The vector space representation allows us to use conventional vector-based distance measures (e.g., euclidean, Manhattan distance, and cosine similarity). We have experimented with a few techniques and qualitatively found cosine distance to work well both in providing good results and being highly performant. Thus, our mapping function S is calculated as:

$$S(t_i \in T) = max_k(cosine(t_i, t_j), t_j \in T, t_i \neq t_j)$$

Where T are all tokens, max_k are the top-k most similar tokens to our input token, t_i. The cosine distance is simply the normalized dot product: $(A \cdot B)/(||A|| \times ||B||)$ where A and

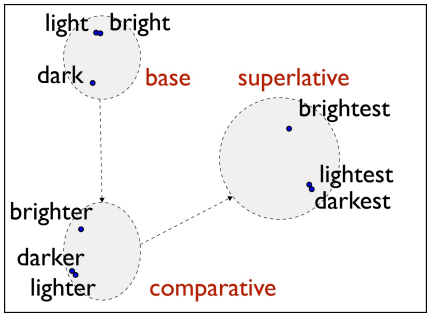

Figure 2. Mappings of term vectors (using PCA) for comparative terms.

B are vectors. If the input to S are multiple tokens we generate a new vector by summing the vectors for all tokens and re-normalizing (e.g., $V(\text{"A B"}) = V(A) + V(B)$).

As a simple demonstration of the function in practical use, we query for the string "shrink nose." (i.e., $S(shrink\ nose) \rightarrow F?$). Internally, we sum the vectors for "shrink" and "nose" to generate a new vector. This vector is compared to the vector of each system feature. The 10 most related features returned are: (1) *filter* > *liquify* (distance = 0.20); (2) *edit* > *transform* > *skew* (0.18); (3) *layer* > *arrange* > *send to back* (0.18); (4) *layer* > *new layer* > *via cut* (0.17); (5) *select* > *transform selection* (0.17); (6) *select* > *grow* (0.16) ; (7) *view* > *zoom in* (0.16) ; (8) *edit* > *transform warp* (0.16); (9) *edit* > *transform* > *distort* (0.15); (10) *view* > *show selection edges* (0.15).

The "shrink nose" example shows the power and downside of this approach. While all the results are reasonable, they confound different tasks. For example, the intent may be to literally shrink the nose. The results returned, however, capture both the idea of shrinking the nose (through layers, transforms, and the liquify filter) but also the idea of zooming in and out of the canvas to "shrink." We consider this challenge later when building our applications.

Syntactic and semantic regularities

As described earlier, vectors generated by the neural network maintain a number of regularities. While it is more evident that synonyms should be near each other, distributed vectors also preserve other relationships that are significantly more complex. The common example in the literature [20, 19] is that the vector for King, $V(King)$, minus the vector for Man, plus the vector for Woman is equal to the vector for Queen ($V(King) - V(Man) + V(Woman) = V(Queen)$). We refer the interested reader to this literature for further examples of semantic relationships.

Figure 2 illustrates one such relationship. The visualization is the 2D Principle-Component Analysis (PCA) projection of the tokens in each of the diagrams. The Figure maps base adjectives (light, dark, bright) with their comparative forms (lighter, darker, brighter) and superlative forms (lightest, darkest, brightest). Note that the forms cluster and that pairs (e.g., base:comparative, base:superlative, comparative:superlative) have roughly the same magnitude and direction of transformation (e.g., getting from light to lighter is the same transform as dark to darker). This regularity may be used, for example, to identify different spectrums of actions and associated commands (e.g., which commands produce *light* → *lighter* → *lightest*). Other relationships that are preserved include pluralization and part-to-whole semantics (e.g., tree to leaf and eye to mouth).

These regularities can be leveraged using the "analogy" transformation. That is, if we select a pair of vectors, $V(F_a)$ and $V(T_b)$, where we know the relationship we want to maintain (e.g., plural, base:comparative, part:whole, etc.) we can find other pairs by picking a new token, F_c and transforming it in the same magnitude/direction. That is, if $F_a : T_b :: F_c : T_d$ then $V(T_d) \approx V(F_a) - V(T_b) + V(F_c)$.

Phrases from strings

While any specific phrase (e.g., "create a grungy album cover") may not occur frequently enough in the corpus for the system to identify it as a token (e.g., *create_a_grungy_album_cover*), we nonetheless want to construct a vector representation for this string so that comparisons are possible (e.g., how close is this goal to a specific system feature?). We find that it is often appropriate to simply sum the constituent tokens into a new vector. In this case, $V(create\_a\_grungy\_album\_cover = V(create) + V(a) + \cdots + V(cover)$. While we would rather have actual instances of the entire phrase, we demonstrate that this works well enough for many applications.

Interestingly, one can weight specific terms by adding extra vectors into the sum. For example, to emphasize "grungy" we can add an extra $V(grungy)$ to the sum. Conversely, one can de-emphasize specific terms tokens by subtracting them out. For example (see Table 1), when identifying commands related to the query "erase," the system returns a combination of eraser tools (e.g., *Eraser Tool* or *Background Eraser Tool*) as well as operations related to masking (e.g., *Layer* > *Layer Mask* > *Apply* and *Layer* > *Layer Mask* > *Delete*). All these commands are highly similar to the *erase* vector (and are in fact different ways to erase content in Photoshop). By adding or subtracting vectors, the set of operations can be focused. For example, $V(erase) + V(mask)$ biases the recommendations to those commands that are close to both by moving the erase vector in the direction of the mask vector. On the other hand, $V(erase) - V(mask)$ will eliminate masking related commands by subtracting out the mask vector from erase. As Table 1 shows, the results can be radically different.

Returning to our previous nose-shrinking example, instead of adding the vector for the command $V(filter\_liquify)$ to the query string, we may simply remove zoom: $V(shrink\ nose) - V(zoom)$. All returned commands now correctly target distortion related operations. We believe that this technique can be used to refine suggestions dynamically based on either behavioral input (the user said 'erase' and then clicked on a mask tool) or explicitly expressed in language (e.g., 'erase without masking').

Growing the vector space

In some situations we would like to "permanently" add novel tokens into our vector space. For example by adding system features that we were unable to detect in the text or creating representative "goal" vectors.

V_1, erase	V_2, erase + mask	V_3, erase - mask
eraser tool	eraser tool	eraser tool
background eraser tool	layer > layer mask > delete	background eraser tool
brush tool	layer > layer mask > reveal all	pencil tool
layer > layer mask > reveal all	layer > layer mask > hide all	dodge tool
layer > layer mask > delete	layer layer mask from transparency	3d > progressive render selection
pencil tool	layer > vector mask > reveal all	view > clear guides
layer> layer mask > from transparency	brush tool	edit > transform warp
layer > layer mask > apply	layer > layer mask > apply	edit > step backward
select inverse	layer > layer mask hide selection	smudge tool
edit clear	background erase tool	select brush

Table 1. Top 10 most similar commands to 3 vectors: $V_1 = V(erase)$; $V_2 = V(erase) + V(mask)$; and $V_3 = V(erase) - V(mask)$

Improving system feature coverage

The limitation of the CRF-driven feature extraction is that some system features are never found. Nearly 500 unique system features (roughly 50%) are not observed in a significant enough quantity to be included in the model. Though these are often less-popular system features, there are applications in which their applications might be useful.

To recover their vector forms, we can use the summation trick to create a new vector. For all commands that we were unable to find a match, we take their human-readable name and calculate a new vector. This allows us to create new "tokens" even when we have never observed them in the raw text. For example, the *filter > sketch > conté crayon* is not found in a significant number of our crawled pages (often because it is mislabeled as "conte crayon"). We generate a vector by summing $V(filter) + V(sketch) + V(conté) + V(crayon)$ and add this new vector into our space of commands.

Goal construction

Once generated, the vector space largely contains tokens representing words and simple phrases (e.g., grungy or "comic book"). However, we often would like to make use of higher-level goal descriptions. The challenge is finding high quality descriptions and then projecting them into the vector space. QF-Graphs [7] address this by finding common search query suggestions. However, while this may identify common tasks the approach does not capture the long-tail of workflows.

Instead, we leverage our crawled datasets to find natural-language task descriptions. As a first step, we simply extract all page titles and section headings from the crawled documents. We filter common strings that are not task related (e.g., "privacy policy" or "responses to . . . ") and remove non-English strings. From each string we also remove text such as "step 1:" that precedes the actual descriptive text (e.g., "step 1: create a new layer"). This process generates over 200k unique strings. These can be further refined by finding phrases that start with common verbs related to Photoshop actions (e.g., create, remove, restore, etc.) or statements that start with "How to..." For the experiment reported in the next section we selected the 20.4k goal statements that included the phrase "how to."

Given the task statements, we can generate vectors for them as before (summing constituent vectors). However, because task descriptions often build on previously identified sub-

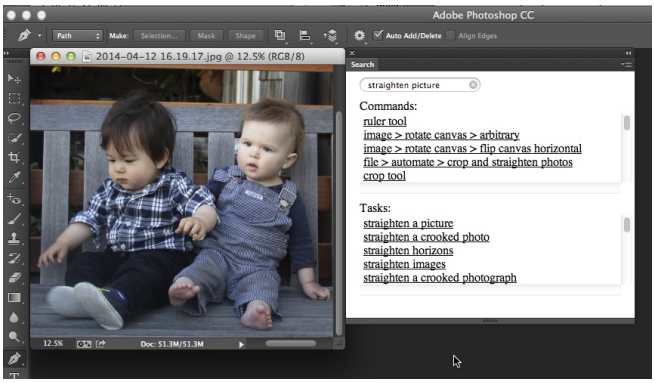

Figure 3. We built a in-app task and command recommender prototype for Adobe Photoshop. In this application of COMMANDSPACE, recommendations are calculated using simple distance metrics (e.g., cosine distance) since tasks and commands are modeled into a single vector representation.

phrases, we pre-process the data to find multi-word tokens. For example, a task description could be "create a graphic novel effect." While we could sum the $V(graphic)$ with $V(novel)$, COMMANDSPACE has already learned a vector for graphic_novel which is preferable (it is more specific). We therefore implement a simple, greedy, phrase matching algorithm that identifies these tokens. Using a suffix-tree we greedily match phrases in the string into their token form (e.g., graphic novel becomes graphic_novel).

APPLICATIONS

We briefly describe a number of applications created using the underlying COMMANDSPACE vector space. We believe that this is but a small sample of what can be constructed.

Keyword search

While users can already use natural language to search the Web for learning materials, COMMANDSPACE enables a task-specific search engine similar to what Brandt et al. built for programmers [5]. Figure 3 shows such an interface inside of Adobe Photoshop. The user can simply type a keyword query to see relevant commands. Additionally, COMMANDSPACE shows tasks that correspond to the query in case the user needs step-by-step guidance. To show relevant commands, COMMANDSPACE maps natural language to Photoshop commands (finding the 5 most similar). To show relevant tasks, COMMANDSPACE maps natural language to the high-level tasks extracted from our corpus. Both commands and tasks

are interactive. Clicking on a command invokes it in the software. Clicking on a task, opens the webpage or document corresponding to that task (but can also be used to query back into COMMANDSPACE as a form of query refinement).

Figure 3 shows the results for the query *straighten picture*. Note that COMMANDSPACE returns not just one tool for straightening but four different ones (*ruler tool*, *image* > *rotate canvas* > *arbitrary*, *crop tool*, *file* > *automate* > *crop and straighten photos*), which can all be used to rotate a photo that may be slightly crooked. If we use a traditional search engine and look for documents that describe how to straighten photos, we find plenty of step-by-step tutorials, but each one focuses on a specific technique rather than giving an overview of all the different tools for straightening. This list of alternatives is not only useful for novices who are still learning about all the available tools, it is also very useful for advanced users who may not know about new features that make their workflows easier. For example, the *image* > *rotate canvas* > *arbitrary* feature predates the ruler tool and is much more difficult to use than the *ruler tool*.

query: *remove blur*	
Commands	*Tasks*
1) filter > blur > shape blur	1) remove gaussian blur
2) blur tool	2) remove blur without disturbing the other area
3) filter > sharpen > sharpen edges	3) remove facial blemishes
4) layer > smart filter > delete filter mask	4) remove freckles
5) filter > blur > blur	5) reduce blur

COMMANDSPACE can implicitly incorporate user feedback for result optimization. For instance, when the user queries for *remove blur*, many of the command suggestions cause blur rather than remove it. The user selects the sharpen edges filter since that one seems most appropriate. COMMANDSPACE keeps track of the user's choice and offers a new list of commands and tasks to reflect the user's selection. In this case, it offers sharpening tools and features. To support this functionality, we create a new mapping function that takes as input the user's goal (remove blur) and a command feature (the sharpen edges filter). In vector space, we simply add the two vectors together before computing similarity to the space of features and tasks.

While monitoring actions and refining the list can be implicit feedback, an application could also support explicit feedback. In this case, a user can opt to include or exclude selected tasks or commands (e.g., include "reduce blur" but exclude "remove freckles"), and the new query into the system would be a simple linear combination (i.e., $V(reduce\_blur) - V(remove\_freckles)$). This can be used as a form of classical relevance feedback [24] or active learning [1]. However, in an in-application interface, this may be too complex for an end user to understand the semantics, so we opted not to expose those features in the current prototype.

Tool and task recommendations

Previous work has shown that offering recommendations for application commands can be effective in helping users discover new parts of an application [17]. With COMMANDSPACE, we can build a recommender system that maps commands to other commands using natural language words. Rather than using collaborative filtering to base recommendations on how others use the application, we can make recommendations based on how people talk about specific tools and which tools they discuss with the same semantics. For example, when the user is working on a comic book effect and using the ink outlines filter, COMMANDSPACE can recommend relevant subsequent tools or alternative tasks.

query: *filter_brush_strokes_ink_outlines*	
Commands	*Task*
1) filter > brush strokes > sumi-e	1) use ink outlines filter to get a cool ink sketch look
2) filter > sketch > charcoal	2) use the oil paint filter
3) filter > artistic > fresco	3) create a graduated and filter effect
4) filter > brush strokes	4) sharpen using high pass filter
5) filter > brush strokes > dark strokes	5) filter snakeskin textures

Similarly, when the user removes an object using the content-aware fill tool, COMMANDSPACE suggests other ways to use the fill tool, such as creating background patterns.

query: *edit_fill*	
Commands	*Task*
1) layer > layer style > pattern overlay	1) create seamless background pattern
2) layer > new fill layer > pattern	2) create checkered chess pattern
3) filter > artistic > rough pastels	3) create damask scrapbook paper or pattern repeat background
4) filter > stylize > tiles	4) add gradient fill layer
5) edit > preset manager > patterns > delete pattern	5) create zig zag pattern

As described above, the user can give the system feedback by the tools they use or by giving more information about their intent by selecting a task. COMMANDSPACE can flexibly take as input a trace including any number of commands and task descriptions. We simply sum the respective vectors and calculate new closest commands and tasks.

Connecting commands

COMMANDSPACE can connect commands at the semantic level. Though we can calculate the distances between commands in vector space, it is often unclear why these tools are linked. Are they often used together? Do they act on the same kinds of objects? Do they create similar effects? One of the benefits of our representation is that we can begin to answer these questions by leveraging the human language that is embedded in the space. We achieve this by identifying parts-of-speech tokens (e.g., verbs, nouns, adjectives, etc.) that are similar to the system feature vectors. That is, we identify triads of strongly connected (high similarity) tokens where two of them are commands and the third is a word or phrase.

For example, we see that *File* > *Automate* > *Photomerge* and *Image* > *Adjustments* > *HDR Toning* are connected, but not why. A Photoshop expert might recognize that to achieve a High Dynamic Range effect (HDR), one often imports multiple images that were exposed at different levels, combines them, and then makes adjustments to specify how to integrate this. If we took the noun "HDR" we would find that it is close to both the *file_automate_photomerge* and *image_adjustments_hdr_toning* system features. This acts to "explain" the relationship between the two commands. Or

Tutorial	Commands (subset)
Retro Comic Book Effect	film_grain, filter_pixelate_color_halftone
Tilt Shift Photograph	gradient_tool, image_adjustments_hue_saturation, filter_blur_lens_blur
Straighten Crooked Photos	ruler_tool, image_rotate_canvas_arbitrary, crop_tool
Content Aware Editing	edit_fill, spot_healing_brush_tool, lasso_tool
Black And White High Key Effect	layer_new_adjustment_layer_gradient_map, filter_blur_gaussian_blur
Sepia	layer_layer_style_blending_options, layer_new_adjustment_layer_photo_filter
Enlarging An Image	image_image_size
Vintage Photo Effect	layer_new_adjustment_layer_curves, image_adjustments_curves

Table 2. A subset of the tutorials used in our COMMANDSPACE evaluation

we might note that the *Sponge Tool* is connected to the *Dodge Tool* but not understand why (the expert would know that both tools are used to change lightness/darkness in different ways). However, by looking at verbs that connect the two, we identify the common verb "brighten" and "whiten."

To do this at a large scale, we tagged the 20k task descriptions we mined from the Web using a conventional part-of-speech tagger [21], and built lists based on their labels (e.g., verbs, nouns, adjectives, adverbs, etc.). The top-100 most frequently appearing words in each category were then used as candidate "explanations." For each word we retrieved the most similar commands (cosine similarity > 0.2) and compute similarity between all of the system features. The edge between those features is labeled with the explanation terms. Note that this map does not include all features, just the ones that are relevant for the most common tasks. Diagrams showing these maps are available as supplementary materials.

The labels we generate can be pushed to the interface to explain recommended commands. For example, if the user chooses the *Crop Tool*, the system can suggest (1) the *Recompose Tool* because both are used to "straighten," and (2) the *Image* $>$ *Crop* command because both features are used to "crop." Similarly, *Edit* $>$ *Copy* and the *Magic Wand Tool* are related since both are used to act on a "selection."

EVALUATION

In order to test the various mappings provided by COMMANDSPACE, we created a test set of 40 workflows described in online tutorials. These were selected based on common workflow categories (e.g., turning an image to black and white, dimension adjustments, filters and effects, spot correction). Since some tasks can be completed through many different techniques, we made sure our tutorials covered a range of techniques (e.g., we had three different tutorials describing how to turn an image into black and white). For each tutorial, we performed the steps described and captured all commands used, a subset of which are shown in Table 2. We further categorized all of the commands as task-specific or general-purpose. Task-specific commands are discriminative of a task (e.g., content-aware fill for removing objects), while general purpose commands are task independent (e.g., selecting a layer or moving an object). For the subsequent analysis, we only used task-specific commands since they allow us to better evaluate the effectiveness of COMMANDSPACE.

System commands to user tasks

To assess how well COMMANDSPACE can map a list of system commands to a high-level user task, we performed two analyses—first, at the command level, and then at the workflow level. In the first analysis, we looked at how well COMMANDSPACE predicts the natural language descriptions that correspond to each command. In the workflow-level analysis, we evaluate whether using all commands in a workflow would allow us to predict the title of the tutorial. While ideally our system would be able to identify the correct tutorial as a "top hit" (for direct end-user applications), it is still valuable for us to have a probability distribution over all tutorials.

Command-level descriptions

For each of the 40 tutorials, we generated a vector representation based on the tutorial name. We took the 152 commands that were used in these tutorials and issued them as queries to find the top ranked tutorial (i.e., which tutorial vector was most similar to the command). In all, we found the mean rank to be 16.3 (median: 15, stdev: 12). While this would appear discouraging, it is worth noting that many of the commands are used broadly across many tasks (e.g., *Lasso Tool* or *Gradient Tool*). Thus, the discriminative power of most commands is fairly low. On the other hand, if we took the command that was most discriminative for each tutorial (i.e., the one that resulted in the highest ranking), we found a mean rank of 5.5 (median: 3.5; std: 5.8). Note that a random ordering would deliver an expected rank of 20 for each tool.

In a user application, one would likely want to eliminate these non-discriminative system features. One could identify these non-discriminative features by polling for highly ranked goals or task vectors that are returned for each tool. If this number is low, the tool is not particularly discriminative and may be removed from consideration.

Task-level descriptions

In addition to the test above, we were curious if the union of all features used in a tutorial would be a good signal for what the tutorial was about. We created a summative vector for all the commands in each tutorial, and used this new vector to query into the tutorial space. Again, we identified the rank of the matched result. Here we found a mean of 12.6 (Median: 9, and stdev: 11.4). While the union appears to be more discriminative than each system feature individually, the performance may still be too low for a user-facing application. Again, by weighting those features that are discriminative, this number can likely be improved significantly.

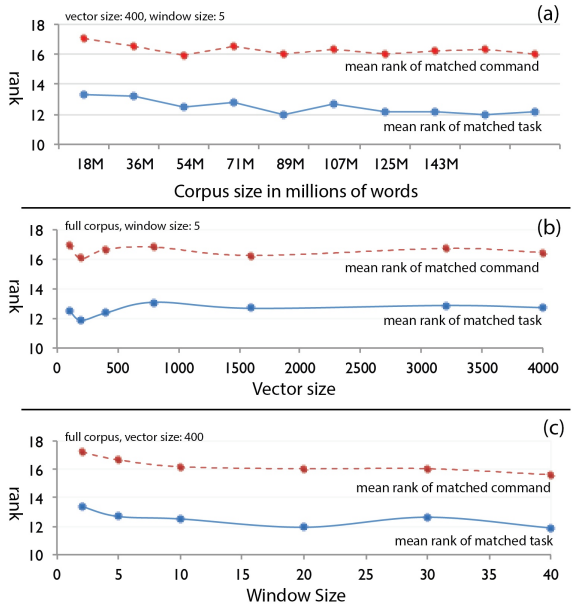

Figure 4. Model performance on command and task evaluations with varying word2vec configurations: (a) varying corpus size; (b) varying vector length; (c) varying window size.

User tasks to system commands

To evaluate whether a natural language query returns the correct set of commands, we query COMMANDSPACE with the titles of our test tutorials. For each command that COMMANDSPACE returns, we match it with all the commands in a given tutorial. We find that on average, COMMANDSPACE returns 23.9% of the commands required by the tutorial in the top 20 results. Qualitatively, we find that the commands that are returned constitute the "key" pieces to a task.

While these results may not make our deep-learning approach useful for all tasks, we believe that our numbers reflect a conservative baseline that can be significantly improved even for conventional search tasks. A great deal of optimization can be achieved by leveraging conventional information retrieval techniques to weight queries and rank responses (e.g., weighting query terms, including models of likelihood of tool use, etc.). Google, as an example, includes 100s of features in page ranking (ranging from document features to click behaviors). Finally, "deep learning" in the context of text is rapidly developing. For example, a recent paper [16] demonstrated vector representations on sentences and paragraphs rather than just words. Given enough data, incorporating such an approach could also improve performance.

Parameter sensitivity

To determine if our results were sensitive to corpus size and training parameter, we constructed a number of different models using *word2vec*. Figure 4 illustrates the effect of using different corpus sizes (a), vector sizes (b), and windows (c). Performance increased (rank of results goes down) with increasing amounts of data. However, even a modest corpus of 17M words (about 35K pages) produced good results. Performance on the specific tasks above appeared to be best with a vector length of 400. Nonetheless, we qualitatively found that longer vectors encoded more semantic relationships. Additional work would be necessary to confirm the optimal size given specific tasks. As the model file size and memory requirements are dramatically reduced with fewer dimensions, tasks that are less demanding in modeling relationships may prefer to use smaller vectors. Finally, we observed some performance gain with increasing window size, but given increased computational costs, the benefit may not justify the time costs.

Comparison to query-feature graphs

To evaluate the Web-mined dataset in isolation, we utilized the set of 20 queries used to test Query-Feature Graphs [7]. These commands were used to test the accuracy of queries related to the GIMP photo-editing software, and were largely (though not entirely) applicable to Photoshop. We modified the commands to their general form and eliminated one which did not apply to Photoshop because they manipulated the UI. Similar to Fourney et al., we used mean precision at 1, 5, and 10 to measure the proportion of test queries whose top-ranking result is judged to be relevant. Table 3 shows our results. Note that we do not view this as an entirely fair comparison, as we crawled a larger corpus. Also, it is possible that Photoshop-related pages are of different and possibly better quality than GIMP pages. Nonetheless, we report these results for completeness.

	P@1	P@5	P@10
COMMANDSPACE	100%	96.6%	85.0%
Query Feature Graphs	80%	66%	43%

Table 3. Comparison between QF-Graphs and COMMANDSPACE

Performance

CRF-Tagging was by far the most costly operation in our chain (approximately 20 pages per minute per core on an Intel Quad Core Q9550 @2.38GhZ with 8Gb of RAM). We believe that this can be significantly optimized by only running the CRF on sentences that are likely to contain matches (as identified by a significantly "cheaper" classifier).

Training the neural network took under 16 hours on 11 cores of a 12 core server (2 Six-Core AMD Opteron(tm) Processor 2431 2.4GHz with 48Gb of RAM). The resulting model could be loaded into approximately 1Gb of memory and held approximately 65k tokens and their corresponding vectors (words, phrases, commands, etc.). Calculating cosine similarities against *all* tokens took under 4 seconds using a naive Python implementation (approximately 60 microseconds per calculation). However, as most applications only compare the "query" against a smaller set, many achieve a real-time feel ($< 100ms$ response times) even in a desktop environment.

CONCLUSIONS

We present COMMANDSPACE, a novel system tying natural language descriptions and low-level application commands. By combining the application's language and the end-user's language together, we are able to create a model that incorporates the syntactic and semantic relationships between terms, goals, and system features. The advantage of the distributed vector representation is that it can be trained in an

unsupervised fashion on fairly large corpora. The output is a model that is robust to many transformations and offers flexible mapping between different elements of an application's domain, ranging from natural language to system commands.

New innovations in deep-learning techniques are opening many avenues for integrating these models with user interfaces. We are eager to apply this approach to other domains and to embed the vector models in new tools and existing applications (e.g., desktop applications such as Excel and PowerPoint). We also think that a similar approach may make programming APIs such as jQuery or D3 more accessible. These libraries have a similar vocabulary gap between user goals and specific APIs. Given their popularity and the wealth of available documentation, we believe we can effectively model these domain spaces and support new interactions.

Additional materials and examples are available at http://cond.org/commandspace.html.

REFERENCES

1. Amershi, S., Fogarty, J., and Weld, D. Regroup: interactive machine learning for on-demand group creation in social networks. In *CHI'12* (2012), 21–30.
2. Androutsopoulos, I., Ritchie, G. D., and Thanisch, P. Natural language interfaces to databases-an introduction. *arXiv preprint cmp-lg/9503016* (1995).
3. Apache. Nutch. `https://nutch.apache.org/`. Accessed: 2014-04-15.
4. Branavan, S., Zettlemoyer, L. S., and Barzilay, R. Reading between the lines: Learning to map high-level instructions to commands. In *ACL'10* (2010), 1268–1277.
5. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. R. Example-centric programming: Integrating web search into the development environment. In *CHI'10* (2010), 513–522.
6. Davison, B. D., and Hirsh, H. Predicting sequences of user actions. In *AAAI/ICML Workshop on Predicting the Future* (1998), 5–12.
7. Fourney, A., Mann, R., and Terry, M. Query-feature graphs: bridging user vocabulary and system functionality. In *UIST'11*, ACM (2011), 207–216.
8. Han, J., Cheng, H., Xin, D., and Yan, X. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery 15*, 1 (2007), 55–86.
9. Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S. R. What would other programmers do: suggesting solutions to error messages. In *CHI'10*, ACM (2010), 1019–1028.
10. Horvitz, E., Breese, J., Heckerman, D., Hovel, D., and Rommelse, K. The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Uncertainty in Artificial Intelligence*, Morgan Kaufmann Publishers Inc. (1998), 256–265.
11. Kohlschütter, C., Fankhauser, P., and Nejdl, W. Boilerplate detection using shallow text features. In *WSDM'10*, ACM (2010), 441–450.
12. Kononenko, O., Dietrich, D., Sharma, R., and Holmes, R. Automatically locating relevant programming help online. In *VL/HCC'12*, IEEE (2012), 127–134.
13. Laput, G., Adar, E., Dontcheva, M., and Li, W. Tutorial-based interfaces for cloud-enabled applications. In *UIST'12*, ACM (2012), 113–122.
14. Laput, G. P., Dontcheva, M., Wilensky, G., Chang, W., Agarwala, A., Linder, J., and Adar, E. Pixeltone: a multimodal interface for image editing. In *CHI'13*, ACM (2013), 2185–2194.
15. Lau, T., Bergman, L., Castelli, V., and Oblinger, D. Sheepdog: learning procedures for technical support. In *IUI'04*, ACM (2004), 109–116.
16. Le, Q. V., and Mikolov, T. Distributed representations of sentences and documents. *CoRR abs/1405.4053* (2014).
17. Li, W., Matejka, J., Grossman, T., Konstan, J. A., and Fitzmaurice, G. Design and evaluation of a command recommendation system for software applications. *TOCHI 18*, 2 (2011), 6.
18. Mikolov, T. word2vec, accessed 2014-04-15. `https://code.google.com/p/word2vec/`.
19. Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
20. Mikolov, T., Yih, W.-t., and Zweig, G. Linguistic regularities in continuous space word representations. In *NAACL-HLT'13* (2013), 746–751.
21. NLTK. Natural language toolkit. `http://www.nltk.org/`. Accessed: 2014-04-15.
22. Norman, D. A. Cognitive engineering. *User centered system design* (1986), 31–61.
23. Ratinov, L., Roth, D., Downey, D., and Anderson, M. Local and global algorithms for disambiguation to wikipedia. In *NAACL-HLT'11*, ACL (2011), 1375–1384.
24. Rocchio, J. J. Relevance feedback in information retrieval. In *The SMART Retrieval System: Experiments in Automatic Document Processing*, G. Salton, Ed. Prentice-Hall, Englewood Cliffs NJ, 1971.
25. Varelas, G., Voutsakis, E., Raftopoulou, P., Petrakis, E. G., and Milios, E. E. Semantic similarity methods in wordnet and their application to information retrieval on the web. In *International Workshop on Web Information and Data Management*, ACM (2005), 10–16.
26. Yahoo! Search BOSS. `https://boss.yahoo.com/`. Accessed: 2014-04-15.
27. Zhang, Z., and Nasraoui, O. Mining search engine query logs for query recommendation. In *WWW'06*, ACM (2006), 1039–1040.