UNIVERSITY OF WATERLOO

SE 499

Vim Awesome: Data mining Vim plugins

Prepared by
David Hu

April 23, 2014

Abstract

Vim is a popular text editor on Unix environments that supports third-party extensions, known as plugins. Existing resources for discovering and browsing Vim plugins do not sufficiently serve the needs of a new user. We examine and highlight shortcomings of the official Vim website, vim.org. We then discuss the main project done in this course, Vim Awesome, a new community resource for Vim plugins. It is a website that presents a comprehensive, up to date, and accurate directory of Vim plugins organized to be useful for exploring and searching for Vim plugins. We discuss challenges involved in building this project, and choose to focus on *data mining* for the remainder of the report. In our context, data mining refers to automatically discovering, extracting details, consolidating, and organizing a directory of Vim plugins. We compare the results of this process—the plugin directory produced—with the information available on vim.org. We tie these results back to initial criteria laid out for the data mining process. Finally, we discuss future work for improvements to the data mining process, as well as remaining work before Vim Awesome will be launched to the world.

Table of Contents

List of Figures

List of Tables

1 Introduction

Vim is a text editor released in 1991 [1]. It is popular among programmers, especially on Unix and Linux. Vim is free and open source software. Originally based on Vi, Vim originally meant "*Vi IMitation*", but now stands for "*Vi IMproved*" [2].

Vim has many features but is also highly extensible through third-party plugins. These plugins can be written in Vimscript, a language specifically designed for extending Vim, but also general-purpose programming languages such as Python, Ruby, Perl, and Lua. These plugins allow Vim to emulate many features of *Integrated Development Environments* (IDEs), including specialized language support, semantic code completion, project management, buffer explorers, file explorers, and new user interface elements [2].

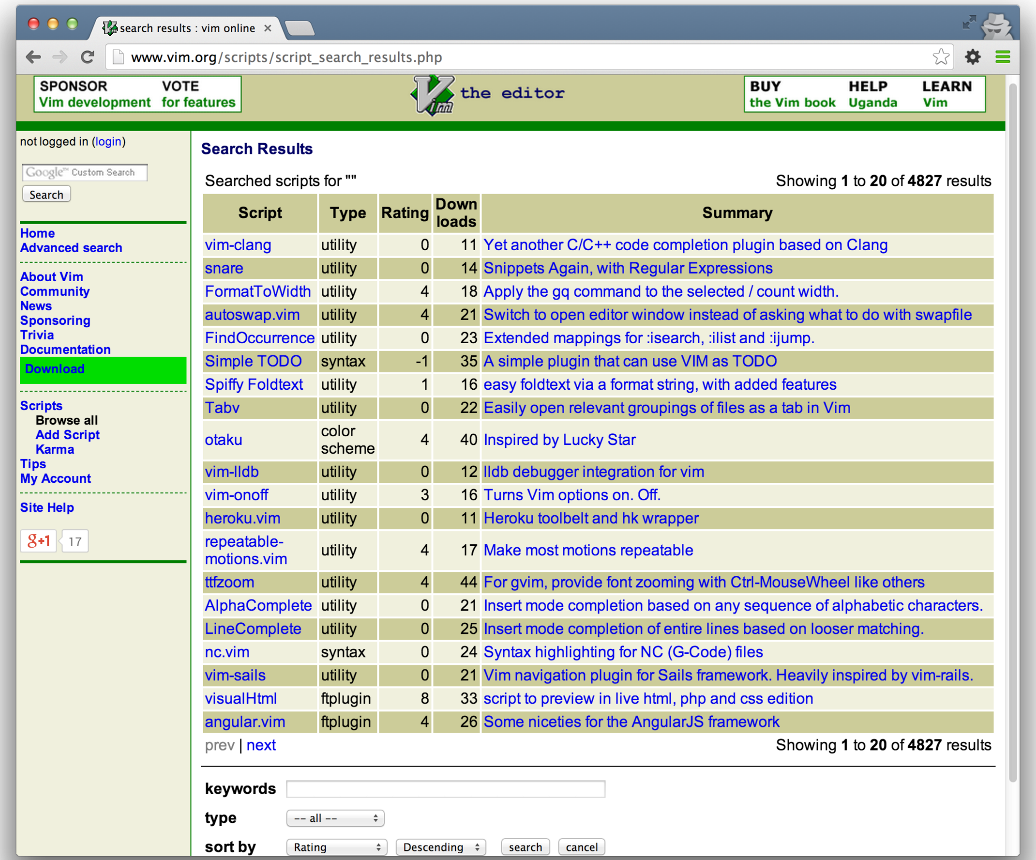
Suppose that one uses Vim as a general-purpose text editor, and one is interested in finding plugins to improve productivity, effectiveness, and user experience. The first place that one might conceivably look is the official website. Alternatively, one might conduct an online search for the keywords "vim plugins", for which the top result will almost always be the official website. It is thus reasonable that one will start at the official website when looking for Vim plugins to try out.

Vim's official website, located at `vim.org`, is the official host of user-contributed plugins. Plugin authors can submit new plugins or update existing plugins, and plugin users can search, sort, and filter the directory of plugins. The interface of this directory is shown in Figure 1-1.

Due to the extensibility and maturity of Vim, there are almost five thousand plugins hosted on vim.org. This proliferation makes it difficult to discover the most useful. It is too time-consuming to download, install, learn, and try each of these plugins. Reading the full descriptions is also time-consuming, and does not provide the tactile experience of how well the plugin works in practice that comes from hands-on use. Thus, one may wish to use the sort and filter options provided by the web UI to reduce the set of plugins to a manageable shortlist, from which one can explore each in greater depth.

Vim.org provides the following functionality to refine search results:

- Search by keywords
- Filter by type: one of *color scheme*, *ftplugin*, *game*, *indent*, *syntax*, *utility*, or *patch*


search results : vim online

www.vim.org/scripts/script_search_results.php

SPONSOR Vim development | VOTE for features | the editor | BUY the Vim book | HELP Uganda | LEARN Vim

not logged in (login)

Search

Home
Advanced search

About Vim
Community
News
Sponsoring
Trivia
Documentation
Download

Scripts
Browse all
Add Script
Karma
Tips
My Account

Site Help

Search Results

Searched scripts for "" — Showing **1** to **20** of **4827** results

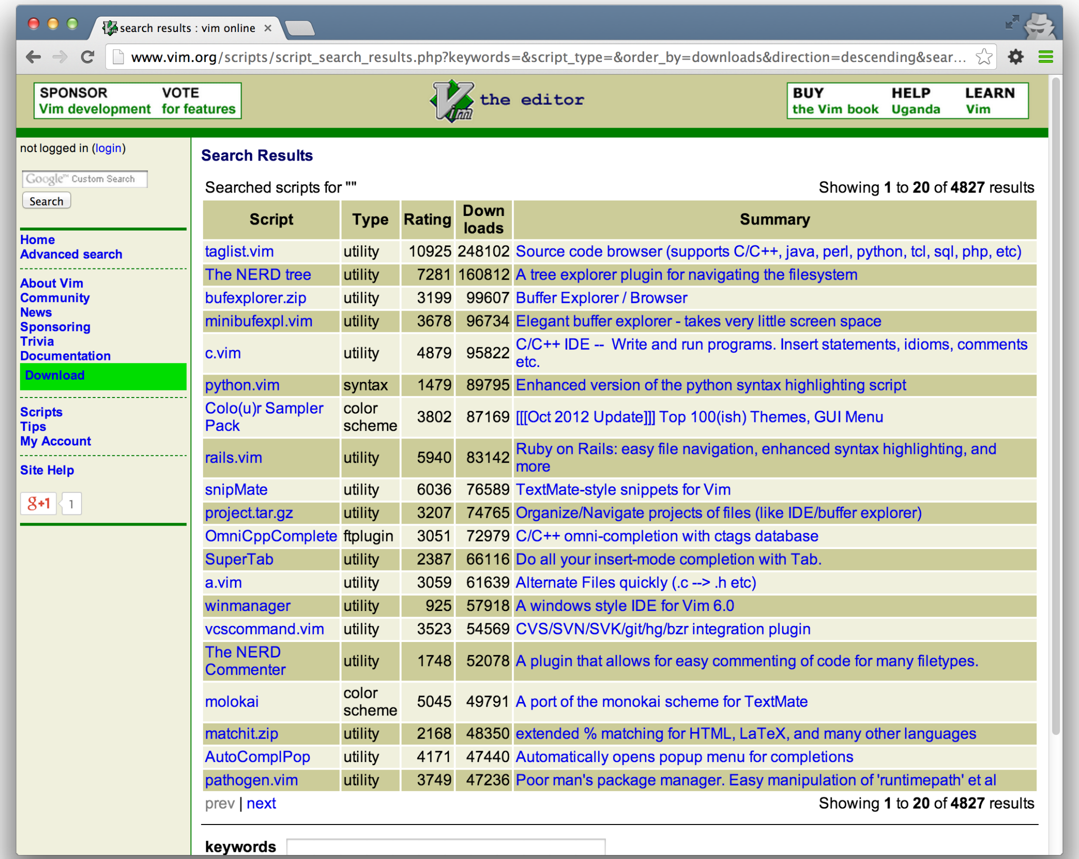
Script	Type	Rating	Down loads	Summary
vim-clang	utility	0	11	Yet another C/C++ code completion plugin based on Clang
snare	utility	0	14	Snippets Again, with Regular Expressions
FormatToWidth	utility	4	18	Apply the gq command to the selected / count width.
autoswap.vim	utility	4	21	Switch to open editor window instead of asking what to do with swapfile
FindOccurrence	utility	0	23	Extended mappings for :isearch, :ilist and :ijump.
Simple TODO	syntax	-1	35	A simple plugin that can use VIM as TODO
Spiffy Foldtext	utility	1	16	easy foldtext via a format string, with added features
Tabv	utility	0	22	Easily open relevant groupings of files as a tab in Vim
otaku	color scheme	4	40	Inspired by Lucky Star
vim-lldb	utility	0	12	lldb debugger integration for vim
vim-onoff	utility	3	16	Turns Vim options on. Off.
heroku.vim	utility	0	11	Heroku toolbelt and hk wrapper
repeatable-motions.vim	utility	4	17	Make most motions repeatable
ttfzoom	utility	4	44	For gvim, provide font zooming with Ctrl-MouseWheel like others
AlphaComplete	utility	0	21	Insert mode completion based on any sequence of alphabetic characters.
LineComplete	utility	0	25	Insert mode completion of entire lines based on looser matching.
nc.vim	syntax	0	24	Syntax highlighting for NC (G-Code) files
vim-sails	utility	0	21	Vim navigation plugin for Sails framework. Heavily inspired by vim-rails.
visualHtml	ftplugin	8	33	script to preview in live html, php and css edition
angular.vim	ftplugin	4	26	Some niceties for the AngularJS framework

prev | next — Showing **1** to **20** of **4827** results

keywords	
type	-- all --
sort by	Rating, Descending

search cancel

Figure 1-1: The default view of vim.org's plugin directory (accessible from "Scripts" > "Browse all") [3]. Note that plugins are actually sorted in descending order of creation date but the form fields incorrectly shows "Rating" as the sort mode.



Search Results

Searched scripts for "" Showing **1** to **20** of **4827** results

Script	Type	Rating	Down loads	Summary
taglist.vim	utility	10925	248102	Source code browser (supports C/C++, java, perl, python, tcl, sql, php, etc)
The NERD tree	utility	7281	160812	A tree explorer plugin for navigating the filesystem
bufexplorer.zip	utility	3199	99607	Buffer Explorer / Browser
minibufexpl.vim	utility	3678	96734	Elegant buffer explorer - takes very little screen space
c.vim	utility	4879	95822	C/C++ IDE -- Write and run programs. Insert statements, idioms, comments etc.
python.vim	syntax	1479	89795	Enhanced version of the python syntax highlighting script
Colo(u)r Sampler Pack	color scheme	3802	87169	[[[Oct 2012 Update]]] Top 100(ish) Themes, GUI Menu
rails.vim	utility	5940	83142	Ruby on Rails: easy file navigation, enhanced syntax highlighting, and more
snipMate	utility	6036	76589	TextMate-style snippets for Vim
project.tar.gz	utility	3207	74765	Organize/Navigate projects of files (like IDE/buffer explorer)
OmniCppComplete	ftplugin	3051	72979	C/C++ omni-completion with ctags database
SuperTab	utility	2387	66116	Do all your insert-mode completion with Tab.
a.vim	utility	3059	61639	Alternate Files quickly (.c --> .h etc)
winmanager	utility	925	57918	A windows style IDE for Vim 6.0
vcscommand.vim	utility	3523	54569	CVS/SVN/SVK/git/hg/bzr integration plugin
The NERD Commenter	utility	1748	52078	A plugin that allows for easy commenting of code for many filetypes.
molokai	color scheme	5045	49791	A port of the monokai scheme for TextMate
matchit.zip	utility	2168	48350	extended % matching for HTML, LaTeX, and many other languages
AutoComplPop	utility	4171	47440	Automatically opens popup menu for completions
pathogen.vim	utility	3749	47236	Poor man's package manager. Easy manipulation of 'runtimepath' et al

prev | next Showing **1** to **20** of **4827** results

keywords

Figure 1-2: Browsing Vim plugins on vim.org, sorting by most downloaded [4].

- Sort by one of *rating*, *downloads*, *script name*, or *creation date*

If one is looking for the most popular or potentially useful plugins without a specific use case in mind, one might begin by sorting by most highly rated or downloaded. If we choose to sort by descending order of downloads, we get the list shown in Figure 1-2.

With the exception of *Colo(u)r Sampler Pack* [5], all of these plugins in Figure 1-2 are more than five years old. Why is this noteworthy? In the last five years, many new plugins have emerged that are superior, more modernized, or have novel functionality that subsumes many of these older plugins. These newer plugins tend to be open-sourced and actively maintained on *GitHub* [6], a social code-sharing website that hosts open-source projects. On the other hand, many of the plugins in Figure 1-2 are no longer maintained. Table 1-1 highlights some prominent examples.

Table 1-1: Plugins in vim.org's top 20 most downloaded that have been superseded by more modern alternatives.

Vim.org plugin	Newer alternatives
Taglist [7] – Source code browser sidebar	*Tagbar* [8] provides all the functionality of Taglist and more. Actively developed.
The NERD tree [9] – File explorer sidebar	*Ctrlp* [10] makes it very easy and quick to open files, the primary use case of NERD Tree
OmniCppComplete [11] – Semantic autocomplete for C/C++	*YouCompleteMe* [12] provides semantic code completion for C/C++/Obj-C, Python, and other languages. Actively developed.
SuperTab [13] – General-purpose autocomplete	*YouCompleteMe* provides semantic autocompletion for some languages but falls back to general keyword-matching autocomplete.
AutoComplPop [14] – General-purpose autocomplete	*YouCompleteMe* provides semantic autocompletion for some languages but falls back to general keyword-matching autocomplete.
Molokai – Dark color scheme	*Solarized* [15] has become a much more popular color scheme and is highly recognizable. It has both a dark and light theme.

Sorting by rating instead of downloads does not fare much better – it also shows many old, obsoleted plugins [16].

Thus, just using vim.org to discover Vim plugins is often inadequate. Many of the most useful plugins were created recently, and do not rank highly or sometimes are not even submitted to vim.org. Many of these plugins were publicized only on social news aggregators such as Hacker News [17] or specialized forums, and are hosted only on code repositories such as GitHub [6]. Further, plugins hosted on GitHub tend to be updated more frequently than their corresponding submissions to vim.org (this will be demonstrated in §6).

For example, Table 1-2 lists some popular plugins created in the last five years that are on many recommended lists now [18], but none of these appear in the first few pages of the most downloaded plugins on vim.org.

The user interface also leaves much to be desired. The default "scripts" page is just the release notes of the ten most recently updated plugins—not a useful view for seeking useful plugins to try out. The default view for the "Browse all" page sorts by creation date, but this is not reflected anywhere in the UI. In fact, the UI actually incorrectly reports the sort mode to be "rating", as can be seen in Figure 1-1. The provided filter options are also

Table 1-2: Popular plugins created in the last 5 years and their positions on vim.org when sorted by descending order of downloads, as of 2014-04-19.

Plugin	**Date created**	**Downloads on vim.org**	**Position on vim.org**
Tagbar	2011-02-23	10204	140th (page 6)
Fugitive	2010-02-15	9434	159th (page 7)
Syntastic	2009-08-11	8225	177th (page 8)
Solarized	2011-03-24	6713	219th (page 10)
Ctrlp	2011-10-03	4437	337th (page 16)
Powerline	2012-01-14	2885	517th (page 25)
GitGutter	2013-02-20	54	4748th (page 237)
YouCompleteMe	2012-04-15	N/A (not submitted)	N/A (not submitted)

arguably too coarse to be useful, with the vast majority of plugins resigned to the catch-all "utility" category. These filter options are also not intuitively named or described. For example, "ftplugin" is not an obvious shorthand for "file type plugin", which means a plugin for a specific file type or language.

There are alternatives to browsing for Vim plugins on vim.org. One is to read blog posts, tutorials, and articles that recommend plugins. However, these static resources are also prone to being outdated and are definitely not comprehensive. One could also ask friends and colleagues for recommendations, but again, there are no guarantees of comprehensiveness, objectiveness, or being up to date.

This project aims to make available another alternative for exploring Vim plugins for all levels of familiarity with Vim plugins.

2 Vim Awesome

The main work done for this SE 499 course is the creation of *Vim Awesome*, a project that aims to address the shortcomings with existing alternatives described above (§1). More precisely, Vim Awesome intends to be a directory of easily browsable Vim plugins that facilitates finding the best plugins. The target audience and primary use case is that of Vim users who want to quickly find the plugins that are most likely to be useful for them.

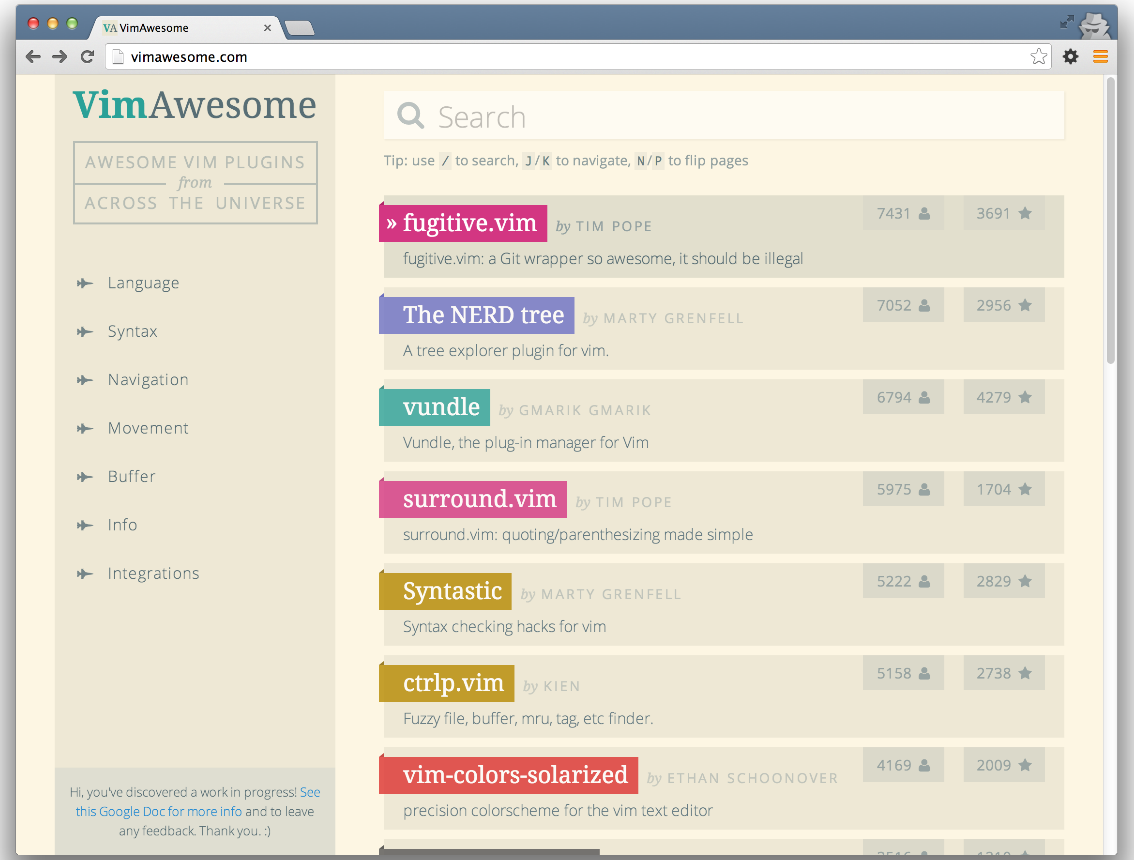
2.1 Features

This project takes the form of a website located at vimawesome.com. The homepage, shown in Figure 2-1, lists the 20 most commonly used Vim plugins, starting with the most common. This optimizes for the intended and common use case of exploring useful plugins. The homepage also features the website logo, slogan, and a navigation sidebar for quickly jumping to popular categories of plugins. Note that many elements on the homepage are still a work in progress, as described further in §7.

Each plugin entry on the homepage summarizes the following key information:

- The plugin's name, inset in a colour-coded banner. The banner's colour is chosen based on the plugin's category.
- The plugin author's name
- A short description of the plugin
- An estimate of the lower bound of number of users, as described in §3. This usage metric is used to sort the plugins on the homepage in descending order.
- If the plugin's code is hosted on GitHub, the number of times the plugin's repository has been *starred*. The number of stars on a GitHub repository roughly gauges the amount of interest and publicity from other GitHub users [19].

The homepage also features a search box. This is used to filter plugins on given keywords. A search using the keyword "mercurial" is shown in Figure 2-2. Search results are updated in real-time as the user types, similar to Google's real-time search. The URL in the address



Hi, you've discovered a work in progress! See this Google Doc for more info and to leave any feedback. Thank you. :)

Tip: use / to search, J / K to navigate, N / P to flip pages

» fugitive.vim *by* TIM POPE 7431 3691
fugitive.vim: a Git wrapper so awesome, it should be illegal

The NERD tree *by* MARTY GRENFELL 7052 2956
A tree explorer plugin for vim.

vundle *by* GMARIK GMARIK 6794 4279
Vundle, the plug-in manager for Vim

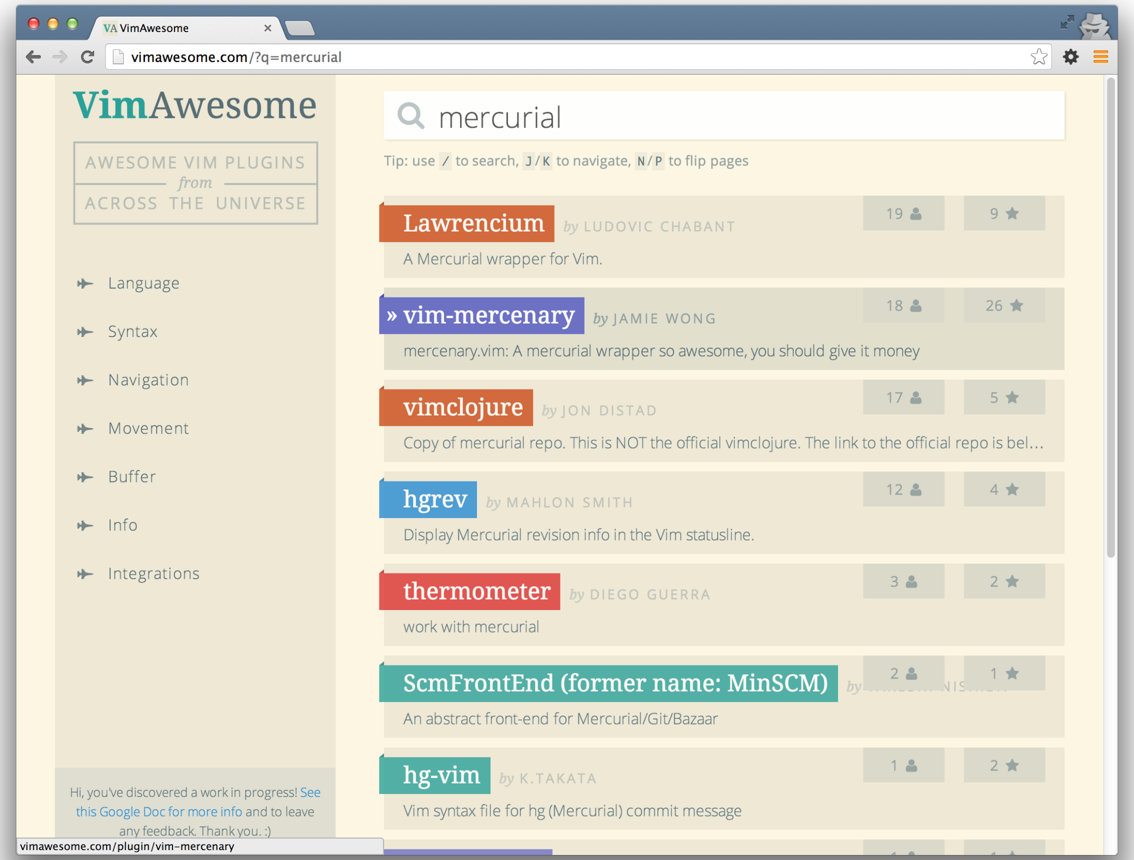
surround.vim *by* TIM POPE 5975 1704
surround.vim: quoting/parenthesizing made simple

Syntastic *by* MARTY GRENFELL 5222 2829
Syntax checking hacks for vim

ctrlp.vim *by* KIEN 5158 2738
Fuzzy file, buffer, mru, tag, etc finder.

vim-colors-solarized *by* ETHAN SCHOONOVER 4169 2009
precision colorscheme for the vim text editor

Figure 2-1: Homepage of Vim Awesome: Lists the most commonly used plugins.


VimAwesome

vimawesome.com/?q=mercurial

VimAwesome

AWESOME VIM PLUGINS *from* ACROSS THE UNIVERSE

- Language
- Syntax
- Navigation
- Movement
- Buffer
- Info
- Integrations

Hi, you've discovered a work in progress! See this Google Doc for more info and to leave any feedback. Thank you. :)

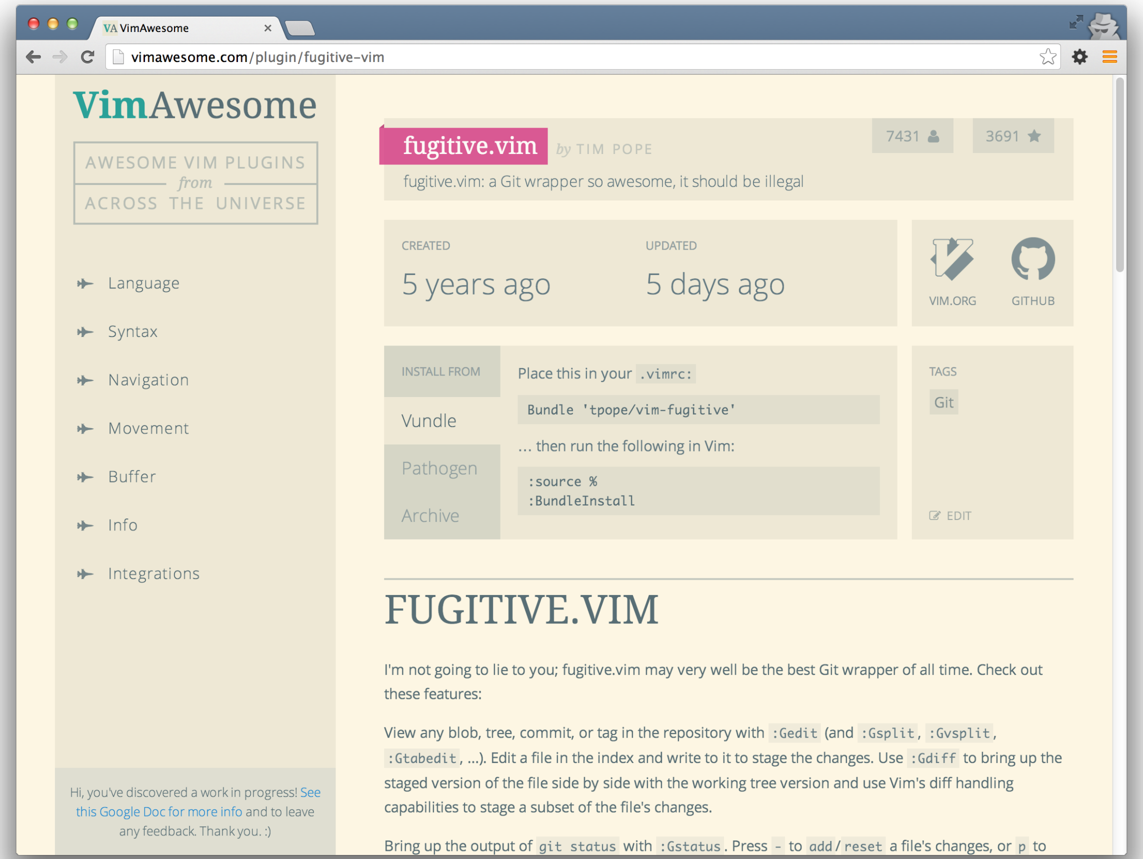
mercurial

Tip: use / to search, J / K to navigate, N / P to flip pages

Lawrencium *by* LUDOVIC CHABANT A Mercurial wrapper for Vim.	19	9 ★
» vim-mercenary *by* JAMIE WONG mercenary.vim: A mercurial wrapper so awesome, you should give it money	18	26 ★
vimclojure *by* JON DISTAD Copy of mercurial repo. This is NOT the official vimclojure. The link to the official repo is bel…	17	5 ★
hgrev *by* MAHLON SMITH Display Mercurial revision info in the Vim statusline.	12	4 ★
thermometer *by* DIEGO GUERRA work with mercurial	3	2 ★
ScmFrontEnd (former name: MinSCM) *by* [illegible] An abstract front-end for Mercurial/Git/Bazaar	2	1 ★
hg-vim *by* K.TAKATA Vim syntax file for hg (Mercurial) commit message	1	2 ★

vimawesome.com/plugin/vim-mercenary

Figure 2-2: A search for the term "mercurial" on Vim Awesome.



Hi, you've discovered a work in progress! See this Google Doc for more info and to leave any feedback. Thank you. :)

fugitive.vim: a Git wrapper so awesome, it should be illegal

Place this in your `.vimrc:`

```
Bundle 'tpope/vim-fugitive'
```

... then run the following in Vim:

```
:source %
:BundleInstall
```

FUGITIVE.VIM

I'm not going to lie to you; fugitive.vim may very well be the best Git wrapper of all time. Check out these features:

View any blob, tree, commit, or tag in the repository with `:Gedit` (and `:Gsplit`, `:Gvsplit`, `:Gtabedit`, ...). Edit a file in the index and write to it to stage the changes. Use `:Gdiff` to bring up the staged version of the file side by side with the working tree version and use Vim's diff handling capabilities to stage a subset of the file's changes.

Bring up the output of `git status` with `:Gstatus`. Press `-` to `add`/`reset` a file's changes, or `p` to

Figure 2-3: The details page for the Fugitive plugin.

bar is updated to reflect the completed search, allowing the user to bookmark or share this search, or to go back to previous searches with the browser's back button.

A user can click on a plugin or press the `enter` key on a focused plugin to go to the plugin's details page (Figure 2-3).

In addition to the key plugin information summarized on the homepage, this page presents the following additional information about a plugin:

- Created date
- Updated date
- A link to the plugin's submission on vim.org, if available
- A link to the plugin's code hosted on GitHub, if available

- Installation instructions for popular methods of installation. For Vundle and Pathogen, commands that can be easily copy-pasted are shown.
- Tags that classify this plugin. Tags are used to aid search, indicate which categories a plugin can be classified as, and can be clicked on to go to a page listing all plugins with that tag.
- A long-form description extracted from either the plugin's submission on Vim.org or its README file from the plugin's GitHub repository if available. This long description will usually describe in detail what the plugin does, describe how the plugin operates, provide screenshots, and provide any special installation instructions if necessary.

It is also possible to use the keyboard to drive much of the navigation on the website. Keyboard shortcuts are provided that aim to be familiar to users of Vim. For example, the `J` and `K` keys scroll up and down on the details page and highlights the next or previous plugin on the homepage. The `/` key can be used to bring keyboard focus to the search box, while pressing `ESC` will unfocus.

As an open-sourced website that leverages modern techniques and technologies, Vim Awesome is intended to be a fast-moving, community-friendly resource around which the community can rally and contribute efforts. The next section describes these technologies.

2.2 Technology

Vim Awesome's color scheme is based on Solarized by Ethan Schoonover [15]. Solarized has become one of the most popular text editor color schemes, and will be recognizable to many Vim users. The Bootstrap CSS library's grid system [20] is used to power the responsive layout.

Vim Awesome is designed as a single-page web application. This means that there is only a single page load when one initially visits vimawesome.com, with all subsequent navigation performed on the client-side with JavaScript, communicating with the server using asynchronous requests as required. The intention is to emulate a native application with immediate feedback on all interactions.

The main technology that facilitates rich client-side interaction is the React JavaScript UI framework released by Facebook [21]. React allows easily writing reusable, composable UI components. It uses a technology called *virtual DOM diff* to ensure UI updates and redraws are quick and optimized.

The HTML5 History API [22] is used in concert with Backbone.js's Router [23] to update URLs on client-side page navigations. This is only enabled on browsers that support this technology, and allows preserving behaviour of the back button across page navigations. Updating the URL to reflect the state of web application also allows bookmarking pages and sharing links.

The backend is written in Python, using Flask as the web server. Data storage is provided by RethinkDB—an open-source, document-oriented, NoSQL database [24]. The Python libraries LXML and Requests are used for data scraping. The entire system is hosted on a Linode SSD Linux server.

Admittedly, many of these technologies were not chosen because they were optimal for the task at hand. Rather, familiarity and the desire to "learn new technologies" had a disproportionate influence on the decision-making.

All of the code for the project is hosted on GitHub [6], a web-based code hosting service that uses the Git revision control system. Currently the code is contained in a private repository, but we intend to open-source this prior to launching.

2.3 Challenges

The following highlights the main challenges involved in this project.

Data mining. This is the challenge of gathering, organizing, and maintaining comprehensive, accurate, and up-to-date data on all Vim plugins. As discussed at length in §1, the key shortcomings of existing alternatives to exploring Vim plugins is the lack of comprehensiveness, objectiveness, and being up-to-date. Equally important is deriving a metric to identify the most popular and useful plugins. This is essential to be able to usefully present data.

Tagging plugins. Tags are the set of labels and categories that are used to classify a plugin. Tags assist with search, help summarize key features, and group together similar plugins. They are also used for navigation from the sidebar. However, with close to 8000 plugins in Vim Awesome's database, it will be difficult to apply accurate tags to each of these plugins.

User interface. Presenting data in a useful and accessible way is another primary motivation for building Vim Awesome. §1 discusses some of the shortcomings in vim.org, which Vim Awesome aims to fix.

Community. Finally, the success of this project depends on being actively used by the community. It requires promotion on relevant channels and possibly forging partnerships to inform Vim users of this website and actually get people to use this. Related is the challenge of building a community of open-source contributors to maintain and extend this project.

This report will discuss in depth the challenge of data mining, while the others are beyond the scope of this report.

3 Data mining challenges

Vim Awesome aims to produce a comprehensive directory of Vim plugins with useful and accurate information about those plugins. The problem of data mining is assembling this data. It is necessary to identify sources of information from which accurate data can be obtained, and then combine that data so that the different sources build up a more accurate and definitive set instead of interfering with each other. It is not enough to just get data, but also aggregate the data in interesting ways to produce new useful information.

However this data is obtained and processed, it is helpful to identify a set of criteria and measures of success:

- The resultant information shall be up to date
- The resultant information shall be accurate
- The resultant information shall be comprehensive and complete
- The resultant information shall be useful
- The resultant information shall be organized such that it can be presented in a useful way
- Data from different sources shall be combined such that:
  - *False negatives*, failing to group together data about the same plugin, are minimized
  - *False positives*, incorrectly grouping together data about different plugins, are minimized
  - Conflicting data shall be reconciled to maximize for accuracy and usefulness in the resultant information.
- The method shall be practical to implement
- The method shall be automated or require minimal human intervention

4 Data sources

This section describes the sources of data that we use to build a comprehensive directory of Vim plugins. We give examples of data that we get from each source, and describe the unique contributions of each data source. Sources of data that were considered but ultimately not used in the data mining pipeline are not described in this report.

4.1 Vim.org

vim.org is the official website of Vim and has a directory where users can upload, browse, and download plugins (Figure 1-1) [1]. Close to 5000 plugins have been submitted to the website. We scrape information about each of these plugins directly from the web search interface, because no API is provided.

Table 4-1 summarizes the information that we extract from vim.org.

Code for extracting plugin data from vim.org is is given in Appendix A.3.

Table 4-1: Plugin data extracted from Vim.org for the fugitive plugin

Field	Value	Description
`vimorg_id`	2975	Unique identifier for the plugin on vim.org
`vimorg_name`	fugitive.vim	Name of the plugin
`vimorg_url`	http://www.vim.org/scripts/script.php?script_id=2975	URL of the permalink plugin page
`vimorg_type`	utility	Category of plugin
`vimorg_rating`	1765	Plugin's rating sum. Each rater can add one of -1, 1, or 4 to the plugin's total rating.
`vimorg_num_raters`	504	Number of people who rated this plugin.
`vimorg_downloads`	9451	Number of times this plugin was downloaded.
`vimorg_short_desc`	A Git wrapper so awesome, it should be illegal	Short description of the plugin
`vimorg_author`	Tim Pope	Plugin author's name
`vimorg_long_desc`	I'm not going to lie to you; fugitive.vim may very well be the best Git wrapper of all time. Check out these features: ...	Full description of plugin
`vimorg_install_details`	Extract in ~/.vim (~\vimfiles on Windows). I highly recommend grabbing directly from http://github.com/tpope/vim-fugitive as this version tends to fall out of date	Special instructions for installing
`updated_at`	2011-04-29	Date the plugin was last updated
`created_at`	2010-02-15	Date the plugin was created

4.2 Vim-scripts organization on GitHub

On the code-sharing website GitHub, the vim-scripts user [25] is an organization that mirrors each of the plugins submitted to vim.org. Each repository hosted by this organization contains the entire code for a plugin. This organization was set up partly because of the creation of various Vim plugin managers that download and manage plugins directly from a Git repository's URL.

We fetch information from this mirror in order to provide a GitHub URL for those plugins that are not hosted on GitHub. Further, for such plugins, we obtain the number of stars of their mirrored repositories, which is one of the metrics we display to gauge relative popularity. We fetch data directly from GitHub's REST API [26].

Table 4-2 summarizes the information that we extract from https://github.com/vim-scripts/taglist.vim. We don't fetch as much information from github.com/vim-scripts because it's a mirror of vim.org and we only care about data that identifies a plugin (can be used as a foreign key to link to a corresponding vim.org plugin) and complementary data unique to github.com/vim-scripts.

Code for extracting plugin data from github.com/vim-scripts is given in Appendix A.2.

Table 4-2: Plugin data extracted from https://github.com/vim-scripts/taglist.vim.

Field	Value	Description
`vimorg_id`	273	Unique identifier for the plugin on vim.org. Used to link this plugin to the corresponding plugin on vim.org.
`github_vim_scripts_repo_name`	taglist.vim	Name of the plugin
`github_vim_scripts_stars`	237	Number of times the repo was starred

4.3 Plugins hosted on GitHub

For plugins where the author uploads the source to GitHub, we can obtain additional useful information that complements the data from vim.org. In particular, the author will sometime include a detailed README file that embeds screenshots and images, often using the Markdown markup language [27]. The number of stars is also a useful metric. GitHub repositories also tend to be more regularly updated than their corresponding submissions on vim.org.

Further, there are many plugins that are just never submitted to vim.org and only exist on GitHub. For those plugins GitHub is the only source of primary information. Thus, unlike with github.com/vim-scripts, we fetch as much information as we can about the plugin from each repository. This additional information is also helpful in associating a GitHub repository with a plugin submission to vim.org.

We fetch information about GitHub plugins directly from GitHub's REST API. Table 4-3 summarizes the information that we extract from https://github.com/tpope/vim-fugitive. Notice that the value of `updated_at` is much more recent than the `updated_at` extracted from vim.org in Table 4-1.

Code for extracting plugin data from GitHub repositories is given in Appendix A.2.

Table 4-3: Plugin data extracted from https://github.com/tpope/vim-fugitive.

Field	Value	Description
`github_owner`	tpope	Username of the repo owner
`github_repo_name`	vim-fugitive	Name of the repo. Together with `github_owner`, this forms a unique identifier for this repo on GitHub.
`github_author`	Tim Pope	Full name of the repo owner
`github_stars`	4104	Number of times this repo has been starred
`github_homepage`	http://www.vim.org/scripts/script.php?script_id=2975	Optional homepage of the repo
`vimorg_id`	2975	If the repo specifies a homepage that is a vim.org plugin page, we extract the ID from the URL and use that to link this plugin to the corresponding plugin on vim.org. This is often not available.
`github_short_desc`	fugitive.vim: a Git wrapper so awesome, it should be illegal	Short description of the repo
`github_readme`	# fugitive.vim I'm not going to lie to you; fugitive.vim may very well be the best Git wrapper of all time. Check out these features: ...	Raw contents of the README file if available.
`created_at`	2009-10-10	Date the plugin was created
`updated_at`	2014-04-14	Date the plugin was last updated

4.4 GitHub dotfiles

There is a trend among Unix users to check in the contents of their Unix configuration files into source control. On GitHub, it has become a convention to submit these repositories with the name *dotfiles*, although many variations exist. These dotfiles repos will usually contain configurations for Unix applications that read configurations from a file, such as

Emacs, Git, Screen, ack, Bash, Zsh, and importantly, Vim. In the cases where the user uses Vundle [28], Pathogen [29], or Neobundle [30] to manage their Vim plugins, we can usually accurately extract the Vim plugins that are used. Since the aforementioned plugin managers heavily rely on Git and GitHub to download plugins, most of the plugin references extracted will be pointers to GitHub repositories.

Looking through dotfiles repositories is one of the key ways in which we discover Vim plugins that are not submitted to vim.org. It is also the primary method of obtaining plugin usage metrics for Vim Awesome. By aggregating the number of appearances of any plugin across all dotfiles repositories on GitHub, we can get an approximate lower bound on the number of actual users of each plugin. This metric is very helpful for ranking plugins on Vim Awesome by how frequently a plugin is used.

Code for extracting Vim plugin usage data from GitHub dotfiles repositories is given in Appendix A.1.

5 Data mining pipeline

All of the sources that we are using to populate our database of plugins are web properties from which data collection can be automated. This property allows us to set up a data pipeline that can be run by a daily cronjob to update Vim Awesome's directory of plugins.

The pipeline consists of the following steps:

1. Discover which plugins exist. Build a table of all known Vim plugins, along with the source from which more data about that plugin can be fetched. For any given source there should not be any duplicated entries about the same plugin, but it is possible at this stage that there are entries from different sources that refer to the same plugin.
2. Fetch information about each plugin from each source.
3. Associate together data from different sources that actually refer to the same plugin to produce de-duplicated rows of the resultant plugins database table.

5.1 Discovery

The first step in the pipeline is to find out which plugins exist. There are two main sources from which we derive this data: vim.org and GitHub dotfiles repos. As discussed in §1, vim.org's plugin collection is incomplete, which motivates using GitHub dotfiles repos to fill in those gaps. Scraping dotfiles repos also produces a list of GitHub repos of Vim plugins, useful for augmenting information on plugins that that have been submitted to vim.org.

Getting data from vim.org is straightforward—we conduct an empty search on vim.org's plugin search page (http://www.vim.org/scripts/script_search_results.php?&show_me=5000) and programatically scrape every result.

For GitHub dotfiles repos, we use GitHub's search API to look for repositories whose name contains one of *vimrc*, *vimfile*, *vim-file*, *vimconf*, *vim-conf*, *dotvim*, *vim-setting*, *myvim*, or *dotfile*. This gives us likely candidates for repositories that will contain Vim configuration files. We then use GitHub's REST API to analyze the contents of the repository in search of indicators of the aforementioned plugin managers. For Vundle and NeoBundle, we look for a file whose name contains one of *vimrc*, *bundle*, *vundle.vim*, *vundles.vim*, *vim.config*, or *plugins.vim*. Within these files we look for lines matching `Bundle`

`<repository_reference>`, where *<repository_reference>* is a shorthand that references a Git repository. For example, `Bundle 'gmarik/vundle'` references https://github.com/gmarik/vundle and `Bundle 'ragtag.vim'` references https://github.com/vim-scripts/ragtag.vim. For Pathogen, we look for a top-level file named `.gitmodules`, a standard git file that references submodules of a particular git repository. We parse the file to look for references to repositories of Vim plugins. Again, these references will usually be URLs to git repositories usually hosted on GitHub.

For each dotfiles repository, we record a row in a database table that uniquely identifies the repository, as well as the list of plugins referenced by that repository. This facilitates incrementally collecting data from dotfiles repositories over multiple runs. This also allows us to decouple the aggregation step from the data fetching. In the aggregation step, we combine the lists of plugins from each dotfiles repository to produce a de-duplicated set of plugins, augmented with the number of times each plugin is referenced.

We limit the dotfiles repositories we analyze to those repositories that have been updated after January 1, 2013. This is done mainly for practical purposes—GitHub's API has a rate limit which restricts the number of searches and requests that can be done per hour, and repositories that have not been updated for a long time tend to be much less likely to contain references to Vim plugins managed by Vundle, Pathogen, or NeoBundle.

From scraping all dotfiles repositories updated after January 1, 2013, we obtain the aggregate statistics summarized in Table 5-1.

Table 5-1: Aggregate statics from scraping GitHub dotfiles repositories in search of references to Vim plugins

Aggregate statistic	**Value**
Total repos scraped	28587
Repos from which we extracted Vim plugins	13962
Repos that use Vundle to manage Vim plugins	6204
Repos that use Pathogen to manage Vim plugins	6172
Repos that use NeoBundle to manage Vim plugins	1586
Total plugins referenced	234926
Total Vundle-managed plugins	120186
Total Pathogen-managed plugins	70463
Total NeoBundle-managed plugins	44277
Total unique plugins discovered	10966

5.2 Extraction

The end result from the previous discovery step (§5.1) is a collection of de-duplicated source-specific identifiers that can then be used to fetch plugin-specific data from. For vim.org, we take the list of vim.org plugin IDs and scrape data from the webpage `http://www.vim.org/scripts/script.php?script_id={vimorg_plugin_id}`. For GitHub plugins, the previous step produces a list of de-duplicated pairs of (GitHub owner name, GitHub repo name) from which we can fetch details about from GitHub's REST API, `https://api.github.com/repos/{owner_name}/{repo_name}`.

5.3 Integration

In this step, we take the disjoint data about plugins fetched from the previous extraction step (§5.2) and combine the data that refer to the same plugin together.

If we just naively scrape all plugins indexed by the sources we have identified above, the result is many duplicated plugins, with information splintered among them. We would like to cluster duplicated plugins together and just have one directory entry to represent each unique plugin. If we do not de-duplicate plugins, we have the following problems:

- Multiple search results representing the same plugin
- Views and user votes for the same plugin are split across multiple entries
- Multiple permalink pages for the same plugin
- Different information about the same plugin

The key idea that we employ to associate plugins is to use a set of identifying fields we obtain from each source to approximately match together identical plugins, as depicted in Table 5-2.

Table 5-2: Fields used to group together plugin data from different sources

Source	**Identifying fields**
vim.org (§4.1)	`vimorg_id`, `vimorg_author`, `vimorg_name`
The vim-scripts user's GitHub repo (§4.2)	`vimorg_id`, `github_vim_scripts_repo_name`
non-vim-scripts GitHub repo (§4.3)	`github_owner`, `github_repo_name`, `vimorg_id` (if extracted from `github_homepage`), `github_author`, URL references from `vimorg_long_desc`

We need to properly associate information scraped about the same plugin from these three sources together. If we do not specify an order of scraping, we need to handle each of the nine possible pairwise orderings from (vim.org, vim-scripts GitHub repo, non-vim-scripts GitHub repo) $\times$ (vim.org, vim-scripts repo, non-vim-scripts GitHub repo):

- Scrape vim.org plugins then one of (vim.org, non-vim-scripts GitHub, vim-scripts GitHub)
- Scrape vim-scripts GitHub repos then one of (vim.org, non-vim-scripts GitHub, vim-scripts GitHub)
- Scrape non-vim-scripts GitHub repos then one of (vim.org, non-vim-scripts GitHub, vim-scripts GitHub)

Now, instead of enumerating each case, we can reduce the number of cases we actually have to handle by making a few assumptions:

- Every vim-scripts GitHub repo has an extractable `vimorg_id` that links to a vim.org plugin, and there will never be two vim-scripts GitHub repos that links to the same vim.org plugin. This is a reasonable assumption because every vim-scripts repo's homepage links to the corresponding vim.org plugin.
- Whenever we scrape a vim-scripts repo, we will have already scraped its corresponding vim.org plugin. This is reasonable because github.com/vim-scripts is a mirror of vim.org updated on a semi-regular basis. This means we will also need to always scrape all of vim.org right before vim-scripts.

- Whenever we scrape a non-vim-scripts GitHub repo, we will have already scraped its corresponding vim.org plugin and vim-scripts repo if they exist. This will be true for the majority of existing plugins, but will not hold if a newly-created plugin is uploaded to GitHub and discovered by scraping dotfiles repositories before a submission is made to vim.org. This is a simplifying assumption that we can make at the expense of a small amount of false negatives in association, if any.

This dictates a scrape ordering of:

1. All of vim.org
2. All vim-scripts GitHub repos
3. All remaining GitHub repos

Combined with the assumptions above, this reduces all cases we have to handle to just one non-trivial case, as shown below.

In the first scraping step—scraping all vim.org plugins—we only need to associate a vim.org plugin with existing vim.org plugins in the database, which can be correctly done using `vimorg_id` as the key. We do not need to consider associating with a vim-scripts GitHub repo or an non-vim-scripts GitHub repo due to assumptions 2 and 3, respectively.

In the second scraping step—scraping all vim-scripts GitHub repos—we can correctly associate a scraped vim-scripts repo with a vim.org plugin by assumption 1 (and transitively associate with any formerly-scraped vim-scripts repos). We do not have to consider associating with a non-vim-scripts GitHub repo by assumption 3.

In the third scraping step—scraping non-vim-scripts GitHub repos—we can associate any GitHub repo with another by using the (owner, repo_name) key.

We are thus left to answer how to associate a non-vim-scripts GitHub repo with a vim.org plugin. First, we define two plugin sources to be in *conflict* if any of the following pairs of fields are both non-empty and differ:

1. `vimorg_id`
2. `(github_owner, github_repo_name)`

3. `normalized_name`: This is generated from the plugin name by replacing accented characters with non-accented ones, stripping all non-alphanumeric characters, lowercasing, and stripping any leading {“vim”, “the”} or trailing {“vim”}.

Two plugin sources that conflict means that we do not want to consolidate them, because they will have conflicting values for key identification fields and will likely not refer to the same plugin.

We can now proceed with the explanation of the heuristic used to associate a non-vim-scripts repo with a vim.org plugin.

1. If the repo’s homepage is a link to a plugin on vim.org, extract the `vimorg_id` from that and use it as the key to associate with that vim.org plugin. Exit.
2. For each vim.org plugin, parse out references to GitHub repositories in the `vimorg_long_desc` field. Produce a map of GitHub repo to a list of vim.org plugin IDs that reference that GitHub repo.
3. Among the set of vim.org plugins that reference this GitHub repo, filter out any of those that conflict. If only one remains, associate with that plugin and exit.
4. If there are multiple vim.org plugins remaining that reference this GitHub repo and do not conflict, filter out any vim.org plugins whose author names are not sufficiently similar to the GitHub repo’s owner’s username or full name. If there is one remaining Vim plugin at this point, associate with that and exit.
5. If are still remaining matching vim.org plugins, log this as an error for further investigation later. This is not expected to be a common occurrence.
6. If all vim.org plugins were filtered out at step 3 or 4, then attempt to find a non-conflicting vim.org plugin with a similar author name among the global set of vim.org plugins. If there is exactly one, associate and exit. If multiple matches, log for investigation. If there are none, insert a new plugin entry.

Admittedly, this method is best-effort, and may result in false positives (incoherency: a single entry about multiple plugins) and false negatives (non-uniqueness: multiple entries about the same plugin).

Avoiding false positives is important, because it is undesirable to have an entry that is supposed to be about a single plugin having inconsistent data constituted from different data for different plugins. For example, the vim.org URL could end up pointing to a different Vim plugin from the GitHub URL. It is more acceptable to have an entry that is missing a vim.org URL due to a false negative (a failure to properly associate) than it is to have the URL be incorrect.

Thus, the heuristic to associate attempts to minimize false positives more than false negatives. A false negative will only result in the following situations:

- In step 1, if a GitHub repo has its homepage set to the wrong vim.org plugin.
- In step 3, if a GitHub repo has a name that is similar to a vim.org plugin's name *and* that vim.org plugin references that GitHub repo in its description *and* they are actually two different plugins.
- In step 4, all of the conditions from the point above *and* author names are similar.
- In step 6, if a GitHub repo has a similar name to a vim.org plugin *and* their author names are similar *and* they are actually two different plugins.

On the other hand, a false positive will result in a number of more common scenarios, such as if a plugin is submitted to vim.org and GitHub with sufficiently different names (such that their `normalized_name`s do not match) or author names.

When consolidating plugins, we keep data we obtain from each source separately, and only combine them when rendering to the UI on Vim Awesome. For field values that are present in multiple sources, we make the choices shown in Table 5-3 to determine which value to show.

Table 5-3: Plugin fields from which we obtain data from multiple sources, and how they are resolved when presented to the user.

Field	**Derived from**
Name	`vimorg_name` if available else `github_repo_name` else `github_vim_scripts_repo_name`
Author	`vimorg_author` if available else `github_author`
Short description	`github_short_desc` if available else `vimorg_short_desc`
GitHub stars	max(`github_stars`, `github_vim_scripts_stars`)
GitHub URL	Construct from `github_owner` and `github_repo_name` if available, else construct from `github_vim_scripts_repo_name`
Created date	The earliest `created_at` from all sources
Updated date	The latest `updated_at` from all sources

Table 6-1: Top 20 most used Vim plugins on Vim Awesome. *F/A* means failure to associate with the corresponding submission on vim.org, while *N/A* means there is no corresponding submission on vim.org.

Plugin name	Users	Created	Updated	Hosted on github.com/...	vim.org ID
fugitive.vim	7431	2009-10-08	2014-04-14	tpope/vim-fugitive	2975
The NERD tree	7054	2006-09-14	2013-05-16	scrooloose/nerdtree	1658
vundle	6794	2010-10-17	2014-04-05	gmarik/vundle	3458
surround.vim	5976	2006-10-27	2013-09-23	tpope/vim-surround	1697
Syntastic	5225	2009-07-11	2014-04-21	scrooloose/syntastic	2736
ctrlp.vim	5159	2011-09-05	2013-07-29	kien/ctrlp.vim	*F/A*
vim-colors-solarized	4169	2011-02-18	2011-05-09	altercation/vim-colors-solarized	*F/A*
vim-coffee-script	3519	2010-01-20	2013-12-18	kchmck/vim-coffee-script	3590
NERD Commenter	3381	2007-04-01	2013-01-10	scrooloose/nerdcommenter	1218
rails.vim	3360	2006-05-30	2014-04-20	tpope/vim-rails	1567
ack.vim	2939	2009-01-20	2014-04-22	mileszs/ack.vim	2572
Tagbar	2800	2011-01-11	2014-04-05	majutsushi/tagbar	3465
SuperTab	2538	2006-08-29	2014-04-09	ervandew/supertab	1643
EasyMotion	2483	2011-03-27	2014-04-20	Lokaltog/vim-easymotion	3526
vim-powerline	2397	2011-11-18	2012-08-17	Lokaltog/vim-powerline	*F/A*
Tabular	2283	2009-03-03	2013-05-16	godlygeek/tabular	3464
vim-markdown	2273	2010-02-24	2014-01-01	tpope/vim-markdown	*N/A*
repeat.vim	2262	2008-01-30	2013-12-24	tpope/vim-repeat	2136
vim-javascript	2213	2009-08-31	2014-03-17	pangloss/vim-javascript	4452
vim-ruby	2151	2008-07-15	2014-04-03	vim-ruby/vim-ruby	*N/A*

6 Results

Recall that the goal of the data mining process is to, in general, produce a much better directory of Vim plugins than what is currently available.

In total, Vim Awesome has 7856 plugins in its database. In comparison, vim.org has 4829 submitted plugins. Sorting by the best metric available to us, the number of users who use a plugin according to dotfiles repos scraped from GitHub, Table 6-1 summarizes the best plugins according to Vim Awesome. Compare with the most downloaded Vim plugins on vim.org, summarized in Table 6-2.

In general, the more commonly used plugins according to Vim Awesome tend to be more modern than the most downloaded plugins on vim.org. The median creation date of Table 6-1 is 2009-08-31 while that of Table 6-2 is 2006-05-31. The median last updated date of Table 6-1 is 2014-03-17 (a little more than one month ago) while that of Table 6-2 is 2011-12-28.

Table 6-2: Top 20 most downloaded plugins on vim.org

Plugin name	**Downloads**	**Rating**	**Created**	**Updated**
taglist.vim	248480	10930	2003-04-25	2013-02-27
The NERD tree	161121	7309	2006-09-15	2011-12-28
bufexplorer.zip	99695	3199	2001-07-25	2013-10-25
minibufexpl.vim	96862	3679	2001-12-01	2004-11-18
c.vim	95973	4880	2002-02-13	2014-04-21
python.vim	89971	1483	2003-10-13	2013-11-18
Colo(u)r Sampler Pack	87229	3802	2012-10-28	2012-10-28
rails.vim	83310	5940	2006-05-31	2014-04-01
snipMate	76694	6037	2009-02-11	2009-07-13
project.tar.gz	74811	3211	2001-10-03	2006-10-13
OmniCppComplete	73081	3051	2006-06-25	2007-09-27
SuperTab	66212	2396	2006-08-30	2014-04-09
a.vim	61758	3061	2001-07-09	2007-06-07
winmanager	58042	925	2002-01-18	2002-04-03
vcscommand.vim	54607	3523	2003-04-18	2013-04-12
NERD Commenter	52160	1748	2007-04-02	2010-12-07
molokai	49903	5053	2008-08-12	2009-01-04
matchit.zip	48411	2168	2001-07-23	2008-01-29
AutoComplPop	47540	4171	2007-05-02	2009-12-12
pathogen.vim	47403	3758	2008-08-07	2014-03-21

Table 6-3: Plugins that appear in both Table 6-2 and Table 6-1.

Plugin name	Last updated on vim.org	Last updated on Vim Awesome
The NERD tree	2011-12-28	2013-05-16
rails.vim	2014-04-01	2014-04-20
SuperTab	2014-04-09	2014-04-09
NERD Commenter	2010-12-07	2013-01-10

In terms of the success of the data integration step described in §5.3, we can see that 15 of the 20 most used plugin entries on Vim Awesome have a correctly associated vim.org ID, 3 of the 20 failed to associate with their corresponding vim.org submission, and 2 of the 20 simply did not have a corresponding submission on vim.org. That is, 3 of the 18 plugins that did have submissions on vim.org resulted in false negatives in data integration. By manually checking each of the top 20 Vim Awesome entries, we verified that all vim.org URLs linked to the correct vim.org submission. That is, there were no false positives among the top 20 plugin entries on Vim Awesome.

It is also interesting to note that only four of the most downloaded plugins on vim.org are among the top 20 most used Vim plugins according to usage data scraped from GitHub dotfiles repos. Table 6-3 summarizes the intersection of Table 6-2 and Table 6-1—plugins that appear in both top-20 lists. Also noteworthy is that, with the exception of SuperTab, Vim Awesome records a more recent last updated date because plugin authors tend to update their GitHub repo more frequently than their vim.org submission.

Changing the sort order on vim.org to be by rating decreases the size of the intersection (with the top 20 most used plugins on Vim Awesome) to three: The NERD Tree, rails.vim, and surround.vim.

7 Future work

7.1 Data mining

As the results above show (§6), there is still a non-trivial portion of plugins that failed to be associated with their corresponding vim.org plugins. More work can be done to try to reduce that number of these false positives. The set of heuristics currently employed to associated plugins (§5.3) are rather conservative and lean towards avoiding false positives at all costs. Thus, we can experiment with shifting this balance and measuring the effects of using less restrictive heuristics (for example, reducing the threshold of author name similarity required). We can also brainstorm and experiment with additional approximate-matching heuristics. We could investigate applying near-duplicate clustering techniques such as shingling plugin descriptions and computing the Jaccard similarity between shingle sets.

It is also possible that there may be subtle bugs that reduce the effectiveness of the algorithm. Evaluating the results of the data mining is currently performed manually after running the entire pipeline. Though we have unit tests for most of the individual steps of the pipeline, building end-to-end integrated testing infrastructure may allow for quicker iterations on experimenting with changes. For example, write tests that, given a GitHub repo and a vim.org plugin URL, assert that the two sources are associated, or assert that they are not.

7.2 Launch

The project has not yet been launched and announced to the world. There are a few remaining features to be completed before the project is ready to be shipped.

At the moment, small subset of the plugins have been tagged. These tags assist with classification, summarization, search, and navigation purposes. All tags assigned at the moment have been manually tagged. It is desirable to have enough of the plugins tagged such that clicking each category and sub-category option on the sidebar yields at least one plugin. On this point, the sidebar is currently just a mock and not functional. Appropriate categories and sub-categories need to be chosen to make the sidebar a useful way to explore the directory.

A page with a simple form where users can submit a new plugin will be helpful. This will complement the existing data mining pipeline and give plugin authors an alternative to submitting plugins to vim.org (or not submitting at all). Initially, all submissions from this form will require a manual review and approval before the plugin will be added to the directory. Related to the needs of plugin authors and keen users, a view or sort mode that shows the newest or “hottest” new plugins will help bring attention to newly-created plugins.

Before launching, a full suite of analytics instrumentation will be installed to learn about usage patterns. At the moment, Google Analytics has been installed to collect page view and visitor statistics. It may also be helpful to install Mixpanel to log specific custom events to get an idea of which features and buttons are actually being used.

The homepage needs to be readied for launch. This includes a more detailed description of what Vim Awesome is, and maybe explanatory information to guide new users on how to try out and install plugins. The GitHub repository hosting all of Vim Awesome’s code need to be prepared to be open-sourced, which entails updating setup instructions, adding a license, and seeding with some initial issues that open-source contributors can use as initial first tasks to tackle.

Finally, when launching, a blog post will be written about the rationale for Vim Awesome, a summary of its key features, and how the initial directory of plugins was compiled (a brief overview of the data mining process). A link to the Vim Awesome website will be submitted to social news aggregators such as Hacker News and relevant specialty blogs and forums such as *usevim* [31]. We will also contact influencers and prolific plugin authors who may be interested in this new initiative and help promote it.

To ensure continued traffic and referrals after launch, effort will be put towards ensuring the website can be indexed by search engines. Due to the heavy client-side rendering, work must be done to ensure we serve pre-rendered static HTML webpages to search engine crawlers.

8 Conclusion

To conclude, let us assess the results of the data mining and see how well they achieved the initial aims set out in §3.

In terms of the quality of the data, we can see that the data mining pipeline produced a more comprehensive directory that included all vim.org plugins plus close to three thousand additional plugins that were not submitted to vim.org. Further, this data is the combination of multiple sources—vim.org and GitHub repositories, with the latter usually being updated more frequently. Although, there did exist false negatives—plugins that failed to have their corresponding vim.org entry associated. Additional work can be done to address this issue. Despite that, there is almost always enough useful information from just the GitHub repo of the plugin to be able to gauge usefulness and download and install the plugin. The trade-off to allow more false negatives was made to ensure there would not be incorrect information due to false positives, of which none were found in the top 20 Vim Awesome entries.

Accompanying the discussion of results (§6), we determined that the median date that plugins were created and last updated on the top 20 Vim Awesome plugins were much more recent than the top 20 most downloaded or highest rated on vim.org. Since the data mining process is automated and will be run daily, it will be at least as up to date as vim.org. The entire data mining pipeline has been implemented and was last run in its entirety in December 30, 2013. That is the data that is currently surfaced on vimawesome.com as of April 23, 2014.

The small overlap between the top 20 Vim Awesome plugins and the top 20 vim.org plugins is evidence that there is distinct value added by Vim Awesome. From this, we can see the key insight in the data mining process, scraping GitHub dotfiles repositories for relative usage data on Vim plugins, ensures that we can get more comprehensive and up to date data that can be organized in a more useful way. With this automated usage discovery mechanism, we hope that, whether or not plugin authors submit new plugins to Vim Awesome, it will nonetheless provide useful, unique information on Vim plugins.

It remains to be seen how Vim Awesome will fare after launch. That will be the true test of whether Vim Awesome has fulfilled its goals—that of being a new community resource for Vim plugins, spurning and supporting the development of Vim plugins, and serving as a useful guide for exploring Vim plugins for all levels of users.

References

[1] *Welcome : vim online*, http://www.vim.org (current Apr. 2014).

[2] R. Paul, *Two decades of productivity: Vim's 20th anniversary*. Ars Technica, 2011, http://arstechnica.com/information-technology/2011/11/two-decades-of-productivity-vims-20th-anniversary/ (current Apr. 2014).

[3] *Search results : vim online*, http://www.vim.org/scripts/script_search_results.php (current Apr. 2014).

[4] *Search results : vim online*, http://www.vim.org/scripts/script_search_results.php?keywords=&script_type=&order_by=downloads&direction=descending&search=search (current Apr. 2014).

[5] R. Melton, *Colo(u)r Sampler Pack*. vim.org, 2012, http://www.vim.org/scripts/script.php?script_id=625 (current Apr. 2014).

[6] *Github - build software better, together*, https://github.com/ (current Apr. 2014).

[7] Y. Lakshmanan, *taglist.vim*. vim.org, 2003, http://www.vim.org/scripts/script.php?script_id=273 (current Apr. 2014).

[8] J. Larres, *majutsushi/tagbar*. github.com, https://github.com/majutsushi/tagbar (current Apr. 2014).

[9] M. Grenfell, *The NERD tree*. vim.org, 2006, http://www.vim.org/scripts/script.php?script_id=1658 (current Apr. 2014).

[10] kien, *kien/ctrlp.vim*. github.com, https://github.com/kien/ctrlp.vim (current Apr. 2014).

[11] V. Neang, *OmniCppComplete*. vim.org, 2006, http://www.vim.org/scripts/script.php?script_id=1520 (current Apr. 2014).

[12] V. Markovic, *Valloric/YouCompleteMe*. github.com, https://github.com/Valloric/YouCompleteMe (current Apr. 2014).

[13] E. V. Dewoestine, *SuperTab*. vim.org, 2006, http://www.vim.org/scripts/script.php?script_id=1643 (current Apr. 2014).

[14] T. Nishida, *AutoComplPop*. vim.org, 2007, http://www.vim.org/scripts/script.php?script_id=1879 (current Apr. 2014).

[15] E. Schoonover, *altercation/vim-colors-solarized*. github.com, https://github.com/altercation/vim-colors-solarized (current Apr. 2014).

[16] *Search results : vim online*, http://www.vim.org/scripts/script_search_results.php?keywords=&script_type=&order_by=rating&direction=descending&search=search (current Apr. 2014).

[17] *Hacker news*, https://news.ycombinator.com/ (current Apr. 2014).

[18] J. Hooks, *5 Essential VIM Plugins That Greatly Increase my Productivity*. 2013, http://joelhooks.com/blog/2013/04/23/5-essential-vim-plugins/ (current Apr. 2014).

[19] *Notifications & stars*, https://github.com/blog/1204-notifications-stars (current Apr. 2014).

[20] *Bootstrap*, http://getbootstrap.com/ (current Apr. 2014).

[21] *React - a javascript library for building user interfaces*, http://facebook.github.io/react/ (current Apr. 2014).

[22] *Manipulating the browser history*, https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Manipulating_the_browser_history (current Apr. 2014).

[23] *Backbone.router*, http://backbonejs.org/#Router (current Apr. 2014).

[24] *An open-source distributed database built with love - rethinkdb*, http://rethinkdb.com/ (current Apr. 2014).

[25] *Vim-scripts*, https://github.com/vim-scripts (current Apr. 2014).

[26] *Github api v3*, https://developer.github.com/v3/ (current Apr. 2014).

[27] *Daring fireball: markdown*, https://daringfireball.net/projects/markdown/ (current Apr. 2014).

[28] gmarik, *gmarik/Vundle.vim*. github.com, https://github.com/gmarik/Vundle.vim (current Apr. 2014).

[29] T. Pope, *tpope/vim-pathogen*. github.com, https://github.com/tpope/vim-pathogen (current Apr. 2014).

[30] S. Matsu, *Shougo/neobundle.vim*. github.com, https://github.com/Shougo/neobundle.vim (current Apr. 2014).

[31] *Usevim*, http://usevim.com/ (current Apr. 2014).

[32] *Mhyee/uw-wkrpt*, https://github.com/mhyee/uw-wkrpt (current Apr. 2014).

Acknowledgements

I would like to acknowledge my collaborators with whom I started this project. Emily Eisenberg wrote the initial version of the scraper to fetch data from vim.org. Ben Alpert converted the fledgling web client code from AngularJS to Facebook's React library. Although both initial collaborators are no longer actively working on the project, Ben Alpert has been diligently reviewing every commit that has gone into Vim Awesome.

I would also like to thank Simon Law and all subsequent contributors for creating the LaTeX template which I used to write this report [32].

All other parts of the project are my own work. That includes the web server; the web frontend code, including design; database models; data mining pipeline; and this report.

Finally, I would also like to acknowledge Bram Moolenaar for creating the fantastic Vim text editor in 1991, without which this entire project would not exist. This report and all the code for Vim Awesome was written in Vim.

Appendix A   Code listings

All of the code for this project will be open-sourced at https://github.com/divad12/vim-awesome when this project is launched. As the code has not been open-sourced yet, code relevant to various sections of the report are given here.

A.1   Python code to extract Vim plugins from GitHub dotfiles repositories

```
###############################################################################
# Code to scrape GitHub dotfiles repos to extract plugins used.
# TODO(david): Write up a blurb on how all of this works.


# The following are names of repos and locations where we search for
# Vundle/Pathogen plugin references. They were found by manually going through
# search results of
# github.com/search?q=scrooloose%2Fsyntastic&ref=searchresults&type=Code

# TODO(david): It would be good to add "vim", "settings", and "config", but
#     there are too many false positives that need to be filtered out.
_DOTFILE_REPO_NAMES = ['vimrc', 'vimfile', 'vim-file', 'vimconf',
        'vim-conf', 'dotvim', 'vim-setting', 'myvim', 'dotfile',
        'config-files']

_VIMRC_FILENAMES = ['vimrc', 'bundle', 'vundle.vim', 'vundles.vim',
        'vim.config', 'plugins.vim']

_VIM_DIRECTORIES = ['vim', 'config', 'home']


# Regexes for extracting plugin references from dotfile repos. See
# github_test.py for examples of what they match and don't.

# Matches eg. "Bundle 'gmarik/vundle'" or "Bundle 'taglist'"
# [^\S\n] means whitespace except newline: stackoverflow.com/a/3469155/392426
_BUNDLE_PLUGIN_REGEX_TEMPLATE = r'^[^\S\n]*%s[^\S\n]*[\'"]([^\'"\n\r]+)[\'"]'
_VUNDLE_PLUGIN_REGEX = re.compile(_BUNDLE_PLUGIN_REGEX_TEMPLATE % 'Bundle',
        re.MULTILINE)
_NEOBUNDLE_PLUGIN_REGEX = re.compile(_BUNDLE_PLUGIN_REGEX_TEMPLATE %
        '(?:NeoBundle|NeoBundleFetch|NeoBundleLazy)', re.MULTILINE)

# Extracts owner and repo name from a bundle spec -- a git repo URI, implicity
# github.com if host not given.
# eg. ('gmarik', 'vundle') or (None, 'taglist')
_BUNDLE_OWNER_REPO_REGEX = re.compile(
        r'(?:([^:\'"/]+)/)?([^\'"\n\r/]+?)(?:\.git|/)?$')

# Matches a .gitmodules section heading that's likely of a Pathogen bundle.
_SUBMODULE_IS_BUNDLE_REGEX = re.compile(r'submodule.+bundles?/.+',
        re.IGNORECASE)


def _extract_bundles_with_regex(file_contents, bundle_plugin_regex):
    """Extracts plugin repos from contents of a file using a given regex.

    Arguments:
        file_contents: A string of the contents of the file to search through.
        bundle_plugin_regex: A regex to use to match all lines referencing
            plugin repos.

    Returns:
        A list of tuples (owner, repo_name) referencing GitHub repos.
    """
    bundles = bundle_plugin_regex.findall(file_contents)
    if not bundles:
        return []

    plugin_repos = []
    for bundle in bundles:
        match = _BUNDLE_OWNER_REPO_REGEX.search(bundle)
        if match and len(match.groups()) == 2:
            owner, repo = match.groups()
            owner = 'vim-scripts' if owner is None else owner
            plugin_repos.append((owner, repo))
        else:
            logging.error(colored(
                'Failed to extract owner/repo from "%s"' % bundle, 'red'))

    return plugin_repos


def _extract_bundle_repos_from_file(file_contents):
    """Extracts Vundle and Neobundle plugins from contents of a vimrc-like
    file.

    Arguments:
        file_contents: A string of the contents of the file to search through.

    Returns:
        A tuple (Vundle repos, NeoBundle repos). Each element is a list of
        tuples of the form (owner, repo_name) referencing a GitHub repo.
    """
    vundle_repos = _extract_bundles_with_regex(file_contents,
            _VUNDLE_PLUGIN_REGEX)
    neobundle_repos = _extract_bundles_with_regex(file_contents,
            _NEOBUNDLE_PLUGIN_REGEX)

    return vundle_repos, neobundle_repos


def _extract_bundle_repos_from_dir(dir_data, depth=0):
    """Extracts vim plugin bundles from a GitHub dotfiles directory.

    Will recursively search through directories likely to contain vim config
    files (lots of people seem to like putting their vim config in a "vim"
    subdirectory).

    Arguments:
        dir_data: API response from GitHub of a directory or repo's contents.
        depth: Current recursion depth (0 = top-level repo).

    Returns:
        A tuple (Vundle repos, NeoBundle repos). Each element is a list of
        tuples of the form (owner, repo_name) referencing a GitHub repo.
    """
    # First, look for top-level files that are likely to contain references to
    # vim plugins.
    files = filter(lambda f: f['type'] == 'file', dir_data)
    for file_data in files:
        filename = file_data['name'].lower()

        if 'gvimrc' in filename:
            continue

        if not any((f in filename) for f in _VIMRC_FILENAMES):
            continue

        # Ok, there could potentially be references to vim plugins here.
        _, file_contents = get_api_page(file_data['url'])
        contents_decoded = base64.b64decode(file_contents.get('content', ''))
        bundles_tuple = _extract_bundle_repos_from_file(contents_decoded)

        if any(bundles_tuple):
            return bundles_tuple

    if depth >= 3:
        return [], []

    # No plugins were found, so look in subdirectories that could potentially
    # have vim config files.
    dirs = filter(lambda f: f['type'] == 'dir', dir_data)
    for dir_data in dirs:
        filename = dir_data['name'].lower()
        if not any((f in filename) for f in _VIM_DIRECTORIES):
            continue

        # Ok, there could potentially be vim config files in here.
        _, subdir_data = get_api_page(dir_data['url'])
        bundles_tuple = _extract_bundle_repos_from_dir(subdir_data, depth + 1)

        if any(bundles_tuple):
            return bundles_tuple

    return [], []


def _extract_pathogen_repos(repo_contents):
    """Extracts Pathogen plugin repos from a GitHub dotfiles repository.

    This currently just extracts plugins if they are checked in as submodules,
    because it's easy to extract repo URLs from the .gitmodules file but
    difficult to determine the repo URL of a plugin that's just cloned in.

    Arguments:
        repo_contents: API response from GitHub of a directory or repo's
            contents.

    Returns:
```

```
    A list of tuples (owner, repo_name) referencing GitHub repos.
"""
gitmodules = filter(lambda f: f['type'] == 'file' and
        f['name'].lower() == '.gitmodules', repo_contents)

if not gitmodules:
    return []

_, file_contents = get_api_page(gitmodules[0]['url'])
contents_decoded = base64.b64decode(file_contents.get('content', ''))
contents_unicode = unicode(contents_decoded, 'utf-8', errors='ignore')

parser = configparser.ConfigParser(interpolation=None)

try:
    parser.read_string(unicode(contents_unicode))
except configparser.Error:
    logging.exception(colored(
            'Could not parse the .gitmodules file of %s.' %
            file_contents['url'], 'red'))
    return []

plugin_repos = []
for section, config in parser.items():
    if not _SUBMODULE_IS_BUNDLE_REGEX.search(section):
        continue

    if not config.get('url'):
        continue

    # The parser sometimes over-parses the value
    url = config['url'].split('\n')[0]
    match = _BUNDLE_OWNER_REPO_REGEX.search(url)
    if match and len(match.groups()) == 2 and match.group(1):
        owner, repo = match.groups()
        plugin_repos.append((owner, repo))
    else:
        logging.error(colored(
                'Failed to extract owner/repo from "%s"' % url, 'red'))

return plugin_repos
```

```
def _get_plugin_repos_from_dotfiles(repo_data, search_keyword):
    """Search for references to vim plugin repos from a dotfiles repository,
    and insert them into DB.

    Arguments:
        repo_data: API response from GitHub of a repository.
        search_keyword: The keyword used that found this repo.
    """
    owner_repo = repo_data['full_name']

    # Print w/o newline.
    print "    scraping %s ..." % owner_repo,
    sys.stdout.flush()

    res, contents_data = get_api_page('repos/%s/contents' % owner_repo)

    if res.status_code == 404 or not isinstance(contents_data, list):
        print "contents not found"
        return

    vundle_repos, neobundle_repos = _extract_bundle_repos_from_dir(
            contents_data)
    pathogen_repos = _extract_pathogen_repos(contents_data)

    owner, repo_name = owner_repo.split('/')
    db_repo = DotfilesGithubRepos.get_with_owner_repo(owner, repo_name)
    pushed_date = dateutil.parser.parse(repo_data['pushed_at'])

    def stringify_repo(owner_repo_tuple):
        return '/'.join(owner_repo_tuple)

    repo = dict(db_repo or {}, **{
        'owner': owner,
        'pushed_at': util.to_timestamp(pushed_date),
        'repo_name': repo_name,
        'search_keyword': search_keyword,
        'vundle_repos': map(stringify_repo, vundle_repos),
        'neobundle_repos': map(stringify_repo, neobundle_repos),
        'pathogen_repos': map(stringify_repo, pathogen_repos),
    })

    DotfilesGithubRepos.log_scrape(repo)
    DotfilesGithubRepos.upsert_with_owner_repo(repo)
    print 'found %s Vundles, %s NeoBundles, %s Pathogens' % (
            len(vundle_repos), len(neobundle_repos), len(pathogen_repos))

    return {
        'vundle_repos_count': len(vundle_repos),
        'neobundle_repos_count': len(neobundle_repos),
        'pathogen_repos_count': len(pathogen_repos),
    }


def scrape_dotfiles_repos(num):
    """Scrape at most num dotfiles repos from GitHub for references to Vim
    plugin repos.

    We perform a search on GitHub repositories that are likely to contain
    Vundle and Pathogen bundles instead of a code search matching
    Vundle/Pathogen commands (which has higher precision and recall), because
    GitHub's API requires code search to be limited to
    a user/repo/organization. :(
    """
    # Earliest allowable updated date to start scraping from (so we won't be
    # scraping repos that were last pushed before this date).
    EARLIEST_PUSHED_DATE = datetime.datetime(2013, 1, 1)

    repos_scraped = 0
    scraped_counter = collections.Counter()

    for repo_name in _DOTFILE_REPO_NAMES:
        latest_repo = DotfilesGithubRepos.get_latest_with_keyword(repo_name)

        if latest_repo and latest_repo.get('pushed_at'):
            last_pushed_date = max(datetime.datetime.utcfromtimestamp(
                    latest_repo['pushed_at']), EARLIEST_PUSHED_DATE)
        else:
            last_pushed_date = EARLIEST_PUSHED_DATE

        # We're going to scrape all repos updated after the latest updated repo
        # in our DB, starting with the least recently updated.  This maintains
        # the invariant that we have scraped all repos pushed before the latest
        # push date (and after EARLIEST_PUSHED_DATE).
        while True:
```

```
        start_date_iso = last_pushed_date.isoformat()
        search_params = {
            'q': '%s in:name pushed:>%s' % (repo_name, start_date_iso),
            'sort': 'updated',
            'order': 'asc',
        }

        per_page = 100
        response, search_data = get_api_page('search/repositories',
                query_params=search_params, page=1, per_page=per_page)

        items = search_data.get('items', [])
        for item in items:
            try:
                stats = _get_plugin_repos_from_dotfiles(item, repo_name)
            except ApiRateLimitExceededError:
                logging.exception('API rate limit exceeded.')
                return repos_scraped, scraped_counter
            except Exception:
                logging.exception('Error scraping dotfiles repo %s' %
                        item['full_name'])
                stats = {}

            scraped_counter.update(stats)

            # If we've scraped the number repos desired, we can quit.
            repos_scraped += 1
            if repos_scraped >= num:
                return repos_scraped, scraped_counter

        # If we're about to exceed the rate limit (20 requests / min),
        # sleep until the limit resets.
        maybe_wait_until_api_limit_resets(response.headers)

        # If we've scraped all repos with this name, move on to the next
        # repo name.
        if len(items) < per_page:
            break
        else:
            last_pushed_date = dateutil.parser.parse(
                    items[-1]['pushed_at'])

return repos_scraped, scraped_counter
```

A.2 Python code to extract plugin details from GitHub repositories

```
###############################################################################
# Routines for scraping Vim plugin repos from GitHub.


_VIMORG_ID_FROM_URL_REGEX = re.compile(
        r'vim.org/scripts/script.php\?script_id=(\d+)')


def _get_vimorg_id_from_url(url):
    """Returns the vim.org script_id from a URL if it's of a vim.org script,
    otherwise, returns None.
    """
    match = _VIMORG_ID_FROM_URL_REGEX.search(url or '')
    if match:
        return match.group(1)

    return None


def fetch_plugin(owner, repo_name, repo_data=None, readme_data=None,
        scrape_fork=False):
    """Fetch info relevant to a plugin from a GitHub repo.

    This should not be used to fetch info from the vim-scripts user's repos.

    Arguments:
        owner: The repo's owner's login, eg. "gmarik"
        repo_name: The repo name, eg. "vundle"
        repo_data: (optional) GitHub API /repos response for this repo
        readme_data: (optional) GitHub API /readme response for this repo
        scrape_fork: Whether to bother scraping this repo if it's a fork

    Returns:
        A tuple (plugin_data, repo_data) where plugin_data is a dict of
        properties that can be inserted as a row in the plugins table, and
        repo_data is the API /repos response for this repo.
    """
    assert owner != 'vim-scripts'

    if not repo_data:
```

```
    res, repo_data = get_api_page('repos/%s/%s' % (owner, repo_name))
    if res.status_code == 404:
        return None, repo_data

if repo_data.get('fork') and not scrape_fork:
    return None, repo_data

if not readme_data:
    _, readme_data = get_api_page('repos/%s/%s/readme' % (
        owner, repo_name))

readme_base64_decoded = base64.b64decode(readme_data.get('content', ''))
readme = unicode(readme_base64_decoded, 'utf-8', errors='ignore')

homepage = repo_data['homepage']
vimorg_id = _get_vimorg_id_from_url(homepage)

repo_created_date = dateutil.parser.parse(repo_data['created_at'])

# Fetch commits so we can get the update/create dates.
_, commits_data = get_api_page('repos/%s/%s/commits' % (owner, repo_name),
        per_page=100)

if commits_data and isinstance(commits_data, list) and len(commits_data):

    # Unfortunately repo_data['updated_at'] and repo_data['pushed_at'] are
    # wildy misrepresentative of the last time someone made a commit to the
    # repo.
    updated_date_text = commits_data[0]['commit']['author']['date']
    updated_date = dateutil.parser.parse(updated_date_text)

    # To get the creation date, we use the heuristic of min(repo creation
    # date, 100th latest commit date). We do this because repo creation
    # date can be later than the date of the first commit, which is
    # particularly pervasive for vim-scripts repos. Fortunately, most
    # vim-scripts repos don't have more than 100 commits, and also we get
    # creation_date for vim-scripts repos when scraping vim.org.
    early_commit_date_text = commits_data[-1]['commit']['author']['date']
    early_commit_date = dateutil.parser.parse(early_commit_date_text)
    created_date = min(repo_created_date, early_commit_date)

else:
    updated_date = dateutil.parser.parse(repo_data['updated_at'])
```

```
        created_date = repo_created_date

    # Fetch owner info to get author name.
    owner_login = repo_data['owner']['login']
    _, owner_data = get_api_page('users/%s' % owner_login)
    author = owner_data.get('name') or owner_data.get('login')

    plugin_data = {
        'created_at': util.to_timestamp(created_date),
        'updated_at': util.to_timestamp(updated_date),
        'vimorg_id': vimorg_id,
        'github_owner': owner,
        'github_repo_name': repo_name,
        'github_author': author,
        'github_stars': repo_data['watchers'],
        'github_homepage': homepage,
        'github_short_desc': repo_data['description'],
        'github_readme': readme,
    }

    return (plugin_data, repo_data)


def scrape_plugin_repos(num):
    """Scrapes the num plugin repos that have been least recently scraped."""
    query = r.table('plugin_github_repos').filter({'is_blacklisted': False})
    query = query.filter({'is_fork': False}, default=True)

    # We scrape vim-scripts separately using the batch /users/:user/repos call
    query = query.filter(r.row['owner'] != 'vim-scripts')

    #query = query.order_by('last_scraped_at').limit(num)
    query = query.order_by(r.desc('plugin_manager_users')).limit(num)
    repos = query.run(r_conn())

    # TODO(david): Print stats at the end: # successfully scraped, # not found,
    #     # redirects, etc.
    for repo in repos:
        repo_name = repo['repo_name']
        repo_owner = repo['owner']

        # Print w/o newline.
        print "    scraping %s/%s ..." % (repo_owner, repo_name),
        sys.stdout.flush()

        # TODO(david): One optimization is to pass in repo['repo_data'] for
        #     vim-scripts repos (since we already save that when discovering
        #     vim-scripts repos in build_github_index.py). But the
        #     implementation here should not be coupled with implemenation
        #     details in build_github_index.py.
        plugin_data, repo_data = fetch_plugin(repo_owner, repo_name)

        repo['repo_data'] = repo_data
        PluginGithubRepos.log_scrape(repo)

        # If this is a fork, note it and ensure we know about original repo.
        if repo_data.get('fork'):
            repo['is_fork'] = True
            PluginGithubRepos.upsert_with_owner_repo({
                'owner': repo_data['parent']['owner']['login'],
                'repo_name': repo_data['parent']['name'],
            })

        r.table('plugin_github_repos').insert(repo, upsert=True).run(r_conn())

        # For most cases we don't care about info from forked repos and just
        # want to scrape the original repo. We can whitelist important forked
        # repos if the need arises, or perhaps we can allow forks w/ a minimum
        # number of stars.
        if repo_data.get('fork'):
            print 'skipping fork of %s' % repo_data['parent']['full_name']
            continue

        if plugin_data:

            # Insert the number of plugin manager users if present.
            if repo.get('plugin_manager_users'):
                plugin_data['github_bundles'] = repo['plugin_manager_users']

            db.plugins.add_scraped_data(plugin_data, repo)
            print 'done'

        else:
            # TODO(david): Insert some metadata in the github repo that this is
            #     not found
            print 'not found.'
```

```
            continue


def scrape_vim_scripts_repos(num):
    """Scrape at least num repos from the vim-scripts GitHub user."""
    _, user_data = get_api_page('users/vim-scripts')

    # Calculate how many pages of repositories there are.
    num_repos = user_data['public_repos']
    num_pages = (num_repos + 99) / 100  # ceil(num_repos / 100.0)

    num_inserted = 0
    num_scraped = 0

    for page in range(1, num_pages + 1):
        if num_scraped >= num:
            break

        _, repos_data = get_api_page('users/vim-scripts/repos', page=page)

        for repo_data in repos_data:

            # Scrape plugin-relevant data. We don't need much info from
            # vim-scripts because it's a mirror of vim.org.

            # vimorg_id is required for associating with the corresponding
            # vim.org-scraped plugin.
            vimorg_id = _get_vimorg_id_from_url(repo_data['homepage'])
            assert vimorg_id

            repo_name = repo_data['name']

            repo = PluginGithubRepos.get_with_owner_repo('vim-scripts',
                    repo_name)
            num_bundles = repo['plugin_manager_users'] if repo else 0

            db.plugins.add_scraped_data({
                'vimorg_id': vimorg_id,
                'github_vim_scripts_repo_name': repo_name,
                'github_vim_scripts_stars': repo_data['watchers'],
                'github_vim_scripts_bundles': num_bundles,
            })
            # Also add to our index of known GitHub plugins.
            inserted = PluginGithubRepos.upsert_with_owner_repo({
                'owner': 'vim-scripts',
                'repo_name': repo_name,
                'repo_data': repo_data,
            })

            num_inserted += int(inserted)
            num_scraped += 1

        print '    scraped %s repos' % num_scraped

    print "\nScraped %s vim-scripts GitHub repos; inserted %s new ones." % (
            num_scraped, num_inserted)
```

A.3 Python code to extract plugin details from vim.org

```
import datetime
import logging
import re
import sys

import requests
import lxml.html
import lxml.html.html5parser

import db
import util


class HTMLParser(lxml.html.html5parser.HTMLParser):
    """An html parser that doesn't add weird xml tags to things"""
    def __init__(self, *args, **kwargs):
        kwargs.setdefault('namespaceHTMLElements', False)
        super(HTMLParser, self).__init__(*args, **kwargs)


PARSER = HTMLParser()


def scrape_scripts(num):
    """Scrapes and upserts the num most recently created scripts on vim.org."""
```

```
res = requests.get(
        'http://www.vim.org/scripts/script_search_results.php?show_me=%d' %
            num)

html = lxml.html.html5parser.document_fromstring(res.text, parser=PARSER)

# Since there are no identifying classes or ids, this is the best way to
# find the correct table
scripts = html.xpath('//table[tbody/tr/th[contains(text(),"Script")]]/*/*')

# the first two rows and the last row aren't scripts
for tr in scripts[2:-1]:
    link = tr[0][0].attrib['href']

    vimorg_id = re.search("script_id=(\d+)", link).group(1)
    name = tr[0][0].text

    # Print w/o newline.
    print "    scraping %s (vimorg_id=%s) ..." % (name, vimorg_id),
    sys.stdout.flush()

    # TODO(david): Somehow also get a count of how many plugins failed to
    #     be scraped in total. Maybe return a tuple with error status.
    # TODO(david): Fix error scraping vimcat (id=4325) (something about
    #     only unicode and ascii allowed, no null bytes or control chars)
    # TODO(david): Fix error scraping
    #     http://www.vim.org/scripts/script.php?script_id=4099 (unicode
    #     chars in plugin name) due to invalid escape sequence \\xe0 in the
    #     Rql regex match in db_upsert._find_by_similar_name.
    try:
        # Merge the data we get here with extra data from the details page.
        plugin = dict({
            "vimorg_url": "http://www.vim.org/scripts/%s" % link,
            "vimorg_id": vimorg_id,
            "vimorg_name": tr[0][0].text,
            "vimorg_type": tr[1].text,
            "vimorg_rating": int(tr[2].text),
            "vimorg_downloads": int(tr[3].text),
            "vimorg_short_desc": tr[4][0].text,
        }, **get_plugin_info(vimorg_id))
        db.plugins.add_scraped_data(plugin)
        print "done"
    except Exception:
            logging.exception(
                    '\nError scraping %s (vimorg_id=%s) from vim.org' %
                    (name, vimorg_id))


def _clean_text_node(node):
    """Cleans a text description node on vim.org.

    More precisely, this removes <br>s (newlines will be kept) and replaces <a>
    tags with the text of the link.
    """
    # TODO(david): Check that all <br>s have been removed.

    # Sometimes there's an <a> at the beginning of the description, and that
    # breaks some code down below because there's no text in the body. This
    # fixes that.
    if len(node) > 0 and not node[0].getparent().text:
        node[0].getparent().text = ""

    # Iterate through the tags of the description
    for elem in node:
        # Remove <br> tags completely
        if elem.tag == 'br':
            # lxml wizardry to remove the tag but keep the text
            if elem.tail:
                if elem.getprevious():
                    elem.getprevious().tail += elem.tail
                else:
                    elem.getparent().text += elem.tail
            elem.getparent().remove(elem)
        # Replace <a> tags with the text of the link
        elif elem.tag == 'a':
            # We've only identified links where the text is the same as the
            # href, or which look like "vimscript #1923"
            if elem.text == elem.attrib["href"] or 'vimscript' in elem.text:
                # lxml wizardry to remove the tag but keep the text
                if elem.getprevious():
                    elem.getprevious().tail += elem.text
                    if elem.tail:
                        elem.getprevious().tail += elem.tail
                else:
                    elem.getparent().text += elem.text
                    if elem.tail:
                        elem.getparent().text += elem.tail
                elif 'vimtip' in elem.text:
                    pass  # Ignore the rare link to www.vim.org/tips/index.php
                else:
                    # Throw an error if it's not one of those types
                    raise Exception("Weird link to %s with text %s" %
                            (elem.attrib["href"], elem.text))
                elem.getparent().remove(elem)


def get_plugin_info(vimorg_id):
    """Gets some more detailed information about a vim.org script

    Scrapes a given vim.org script page, and returns some detailed information
    about the plugin that is not available from the search page, like how many
    people rated a plugin, the author's name, and a long description.
    """
    res = requests.get(
            'http://www.vim.org/scripts/script.php?script_id=%s' % vimorg_id)

    html = lxml.html.html5parser.document_fromstring(res.text, parser=PARSER)

    rating = html.xpath('//td[contains(text(),"Rating")]/b')[0]
    rating_denom = int(re.search("(\d+)/(\d+)", rating.text).group(2))

    body_trs = html.xpath(
            '//table[tbody/tr/td[contains(@class,"prompt")]]/*/*')

    assert body_trs[0][0].text == "created by"
    creator = body_trs[1][0][0].text

    assert body_trs[6][0].text == "description"
    description_node = body_trs[7][0]
    _clean_text_node(description_node)

    assert body_trs[9][0].text == "install details"
    install_node = body_trs[10][0]
    _clean_text_node(install_node)

    download_trs = html.xpath(
            '//table[tbody/tr/th[text()="release notes"]]/*/*')

    # Parse created and updated dates
    assert download_trs[0][2].text == "date"
    updated_date_text = download_trs[1][2][0].text
    created_date_text = download_trs[-1][2][0].text

    date_format = "%Y-%m-%d"
    updated_date = datetime.datetime.strptime(updated_date_text, date_format)
    created_date = datetime.datetime.strptime(created_date_text, date_format)

    return {
        "vimorg_num_raters": rating_denom,
        "vimorg_author": creator,
        "vimorg_long_desc": _get_innerhtml(description_node),
        "vimorg_install_details": _get_innerhtml(install_node),
        "updated_at": util.to_timestamp(updated_date),
        "created_at": util.to_timestamp(created_date),
    }


# Stolen from KA scraping script
def _get_outerhtml(html_node):
    """Get a string representation of an HTML node.

    (lxml doesn't provide an easy way to get the 'innerHTML'.)
    Note: lxml also includes the trailing text for a node when you
          call tostring on it, we need to snip that off too.
    """
    html_string = lxml.html.tostring(html_node)
    return re.sub(r'[^>]*$', '', html_string, count=1)


# Stolen from KA scraping script
def _get_innerhtml(html_node):
    """Get a string representation of the contents of an HTML Node

    This takes the outerhtml and pulls the two tags surrounding it off
    """
    html_string = _get_outerhtml(html_node)
    html_string = re.sub(r'^<[^<>]*?>', '', html_string, count=1)
    return re.sub(r'<[^<>]*?>$', '', html_string, count=1)
```