Table of Contents

Goals for the Project ........................................................................................................... 1
System Design & Algorithmic choices ............................................................................. 1
  System Overview ............................................................................................................ 1
  Classifier ....................................................................................................................... 2
  Database ....................................................................................................................... 4
Typical Usage Scenarios ................................................................................................. 5
  Scenario 1: Sign up, create a new category, find a feed, and assign the feed to a category ...................................................................................................................... 5
  Scenario 2: Browse articles in a category. Flip through pages. Mark some articles as irrelevant. Read an article ................................................................. 7
  Scenario 3: Edit name and remove feeds in a category .............................................. 8
  Scenario 4: View Queue ............................................................................................. 9
Experiments and Evaluations ......................................................................................... 10
  Classifier Testing ......................................................................................................... 10
  Usability Testing .......................................................................................................... 11
What we learned/what we would have done differently .............................................. 12
Conclusions ..................................................................................................................... 13
Appendices ....................................................................................................................... 14
  Who Did What ............................................................................................................ 14
  External Code .............................................................................................................. 14
  Readme for Read.me ................................................................................................. 15

List of Figures

Figure 1: System Diagram ............................................................................................... 2
Figure 2: Entity Relationship Diagram for Database .................................................... 4
Figure 3: Category Page After Sign Up ......................................................................... 5
Figure 4: Adding a Category ......................................................................................... 6
Figure 5: Finding a feed ................................................................................................. 6
Figure 6: Adding a feed to a category ........................................................................... 7
Figure 7: Browsing articles in a category ....................................................................... 7
Figure 8: Marking articles as irrelevant ....................................................................... 8
Figure 9: Reading the full article .................................................................................. 8
Figure 10: Activating the view for editing a category ................................................ 9
Figure 11: View for editing a category ......................................................................... 9
Figure 12: Viewing the queue ...................................................................................... 10
Figure 13: F-measure vs. Number of Training Examples for Classifier Variants ...... 11
Goals for the Project

We had two major goals:

1. We wanted to simulate the look and feel of a newspaper. Many RSS readers lack a stylized appearance, and we felt that an aesthetically pleasing interface would engage users of our service and help distinguish it from ‘the competition.’

2. We wanted to avoid overloading users with content; showing them articles that they were most interested in with minimal effort on their part.

System Design & Algorithmic choices

System Overview

Users create their own categories - which are meant to resemble the sections of a newspaper. Users can then associate RSS feeds with their categories. When a user selects a category, he/she will see the most recent summaries for all the feeds associated with that category. To avoid showing the user too much content we limited the number of pages served to a maximum of 10 (or 4 if the classifier is filtering articles, as discussed later).

Every feed is updated at least once a hour. Additionally all feeds associated with category are updated when a user selects a category. A crontab job to update feeds has been set up to run on the first minute of every hour. The system keeps track of last modified and html etags to avoid retrieving articles in case the local data is up-to-date. This is a good practice that allows feed publishers to save bandwidth. The standard out from this job is emailed to a group member to ensure successful execution.

Old articles are deleted once an hour. This is also scheduled as a crontab job. The system removes all but the 20 newest articles from every feed. An exception is made for the outdated articles that has been queued by at least one user.

When a user selects an article to read more, the system will load the article website in real-time. Many levels of html parsing must be done before it is displayed to the user. On the server, we remove html tags that could potentially prevent our site from loading it correctly. Another level of parsing is done in javascript so everything except the article content is removed. If the article
content could not be retrieved for one reason or another, the description of the article along with the link to the full story is shown instead.

Each category created by the user has its own classifier. The classifier initially has no training data and is inactive; at this stage, when a user selects a category the system displays articles from all the feeds associated with the category.

To train the classifier, the system makes use of two different kinds of user feedback. Firstly, when a user selects an article to read more, this is treated as implicit positive feedback - the article is added to the classifier as a relevant training example (specifically, the title and description of the RSS item is added, as opposed to the full article story). Secondly, each displayed article includes an ‘x’ which a user may select if he/she dislikes an article, which adds the article to the classifier as an irrelevant training example. When the user refreshes the page, all articles that have been read or marked irrelevant will be removed.

**Classifier**

The classifier is Naïve Bayes with two classes: ‘relevant’ and ‘irrelevant’. To reduce the size of the feature space, stop words are omitted and words are stemmed (this is not done in the classifier itself but in an external utility function).
The classifier is capable of two different types of feature pruning: frequency based and mutual information based. As of the time of the presentation, neither pruning method was being used in the live system (it does, however, support the use of both pruning methods, and frequency based pruning was activated prior to project turn-in).

Once the classifier is active, our initial approach was to simply classify each document as relevant or irrelevant, and return only articles classified as relevant. However, we were concerned about the possibility that the classifier would return no articles if it classified everything as irrelevant. To guarantee the return of some articles while still attempting quality control, we made use of some of the details of the classifier.

Naïve Bayes assigns a score to each possible class when determining the class of a document (in our system there would be a score for the relevant class and a score for the irrelevant class), and the class with the highest score is the chosen class.

We decided that instead of only returning items for which \( S_R > S_{-R} \), we would use the following method:

1. Assign to each document the difference between the scores for the relevant and irrelevant classes: \( S_R - S_{-R} \)
2. Return a fixed number of items - those with the most positive differences

This approach allows for returning articles that would be classified as irrelevant (i.e. articles for which \( S_R < S_{-R} \)), but it favors returning articles that are ‘more relevant’.
Database

The live service runs on a PostgreSQL database. The entity representation diagram is given below.

The Channel table stores feed metadata, including the feed link and title. It also stores the HTTP cache validation tags 'etag' and 'last-modified' to avoid re-downloading article summaries if our stored version is up-to-date.

As defined in the RSS specifications, items are the individual article summaries in a feed. The Item table stores the data associated with each item, which includes (in addition to the feed it belongs to) at least one of a title or description, along with possibly a link to the full article and/or a publishing date (according to the most liberal RSS specification examined1). The Item table also includes an 'outdated' field, which keeps track of whether an item is 'old' - i.e. whether it is older than the 20th oldest article in its feed.

1 There are three major RSS specifications: .91, 1.0, and 2.0 (which is the most flexible in what is allowed).

.91: [http://www.rssboard.org/rss-0-9-1](http://www.rssboard.org/rss-0-9-1)
1.0: [http://web.resource.org/rss/1.0/](http://web.resource.org/rss/1.0/)
2.0: [http://cyber.law.harvard.edu/rss/rss.html](http://cyber.law.harvard.edu/rss/rss.html)
There are four junction tables: ‘Subscription’ keeps track of which feeds are associated with which categories. ‘Keep’ tracks which articles each user has queued. ‘Follows’ tracks what feeds each user likes (which may not yet have been added to a particular category). ‘Rates’ tracks the feedback users give on articles.

Django automatically creates the User table, along with a Session table for maintaining sessions (not shown here).

Typical Usage Scenarios

Scenario 1: Sign up, create a new category, find a feed, and assign the feed to a category

After a user signs up/logs in, they will be presented with the categories page, which is initially empty (fig 3.). The system detects when the user has no categories and provides guides to get started. The user can add categories by selecting the ‘+’ icon (fig. 3) and entering a name for the category (fig. 4). The users can then use the search bar on the right of the categories page to find feeds (fig. 5) and can associate them with categories by dragging the feed from the feed list to the desired category (fig. 6).
Figure 4: Adding a Category

Figure 5: Finding a feed
Scenario 2: Browse articles in a category. Flip through pages. Mark some articles as irrelevant. Read an article.

When a user selects a category to browse, he/she can navigate through pages by clicking the “prev”, “next”, or the paging dots (fig. 7). The user can mark articles as irrelevant by clicking the red “x” that appears when the mouse hovers over an article (fig. 8). The user may read the full article by clicking on the article’s display panel (fig. 9).
Scenario 3: Edit name and remove feeds in a category

The ‘edit’ option appears when the mouse hovers over a category (fig. 10). When the option is selected, an editing view for the category expands (fig. 11) which allows users to remove feeds and change the category name.
Scenario 4: View Queue

To view their queue, users can select the “queue” button on the top right.
Experiments and Evaluations

Classifier Testing

We wanted to evaluate whether feature pruning could be used to improve the F-measure of our classifier. To do this we made use of the RCV1 dataset\(^2\), which consists of articles that have been human-labeled using a set of subjects.

We chose a subset of subjects to represent a user with particular preferences, and treated the entire set of data as being part of feeds that were served to one category for the user. Under this scheme, articles having one of the chosen subjects would be considered relevant, and otherwise they would be irrelevant.

The following experiment was repeated 25 times to obtain average precision, recall, and F-measure after training with up to 150 documents in increments of 5.

Available at: http://www.ai.mit.edu/projects/jmlr/papers/volume5/lewis04a/lyrl2004_rcv1v2_README.htm
Document samples were taken from a pool of 23149 documents, with all articles labeled ‘government/social’ (6970) being considered relevant. Pruning was triggered whenever the number of features exceeded 1000, and reduced the number of features to 300 (150 per class):

1. Pick 150 documents for training the classifier and 3000 for testing from a set of 23149 documents.
2. Repeat:
   a. Add 5 training examples to the classifier
   b. Classify all testing documents. Record precision, recall, and F-measure.

![Image: F-Measure vs. Number of Training Examples for Classifier Variants]

Figure 13: F-measure vs. Number of Training Examples for Classifier Variants

It should be noted that documents in the obtained dataset are longer than the typical article summary: 120.9 words per instance (averaged over the 23149 documents) versus 31.4 words per article summary (averaged with 1394 articles from 20 feeds).

As visible from the graph, feature pruning does not enhance the performance of the system. Using two-sample unpooled t-tests with a critical region of .05, frequency based pruning was found to generally have similar recall and worse precision compared to no pruning, and mutual information based pruning was found to generally have worse recall and similar precision.

**Usability Testing**

We conducted usability tests with several computer science students, who are potential users of our RSS reader. The tests were organized into two rounds, the first round had three testers and the second round had two. For each tester, we assigned them tasks for all the scenarios, and let them perform the tasks while thinking aloud. We observed the tester's performance in the tasks and wrote
down anything noticeable. Every test revealed new bugs and usability issues. After the first round, we gathered all the notes and compiled them into to-do items, prioritized the bugs, and fixed them. After all the bugs were fixed, we conducted a second round of usability tests. Besides some minor issues, the testers’ experiences significantly improved thanks to the bug fixes.

Examples of bugs and usability issues:

- **Issue:** HTML not parsed correctly using our own regular expressions
  **Fix:** use a Python HTML parser, Beautiful Soup, to parse HTML pages
- **Issue:** No feedback when dragging a feed to a category
  **Fix:** Add highlight and text feedback to the category box when the feed is dropped
- **Issue:** Do not know what a feed is and how to add it
  **Fix:** Besides our guides (yellow post-it notes) after registration, we added a help page that guides users through the process of adding a feed
- **Issue:** When browsing articles, don’t know where the articles are from
  **Fix:** Add a source description under article title

What we learned/what we would have done differently

We’ve learned a great deal as well as realized many things that could be done better. By the end of the quarter, our main problems occur in UI, classifier, and performance. It would have been very beneficial if we have done them differently in the beginning.

Using a framework can be very beneficial. Django made the web development process manageable. It saved us time by providing library for database access, template frameworks and session management. All of which would of taken very long time to implement ourselves.

We learned that it is important to listen carefully to user feedback. We should have talked with people about our vision and goals, and developed paper prototypes before coding the user interface. Paper prototype allows us to make changes easily so we can have a well-tested UI before implementing the final version. This would have allowed us to create a friendlier user interface.

If we were starting over again, we also would have experimented with more diverse classification schemes. The Naïve Bayes classifier used in our system requires many training examples to learn the preferences of a user for a particular category. Users would likely become frustrated when they see articles similar to those they have provided negative feedback on. This can be especially difficult to avoid if an article is a conjunction of relevant and irrelevant concepts (for example, a user may find articles about international criminals interesting,
but does not want to hear about WikiLeaks). With these issues in mind, it would have been worthwhile to test a separate technique to filter content (like keyword based search) to get a more interesting comparison.

We could have improved the system performance-wise. Firstly, we implemented caching too early. The view is not updated when needed. This had a significant impact on our classifier’s performance. We should have got our system working properly, and then implemented caching later. Secondly, our current system updates feeds as we view a category. This does reduce the performance of the system, and have potential scalability issues if the number of users were to be increased. We would automatically update feeds more frequently and not updating when users choose category.

Conclusions

Although the usefulness of the service is limited by speed and classification issues and many of the extra features we planned were not implemented, we have achieved our initial goals. The application displays articles in a newspaper layout, learns user's preference on the fly, and only shows stories that users really care about. In conclusion, we have done a successful project within the 10-week timeframe and with the resources we were given.

Ideas for future work:

- Add caching in a way that it will refresh when users expect to see changes
- Updating feeds should always be a crontab job that runs in the background. It shouldn't be triggered by user interactions, which take a long time.
- Refine the user interface to make it more robust and browser compatible.
- Try new classifiers and training data to improve our machine learning algorithms.
- Better method for retrieving articles or doing more to make it reliable.
- Implement “favorite” feature
- Supports for other languages
- More the public testing
Appendices

Who Did What

- **Sean Ren**: user interface design and implementation, Django view functions for most pages, javascript functions, usability testing
- **Peter Scheibel**: database schema, Django model definitions, Naive Bayes classifier
- **Kha Nguyen**: application deployment and maintenance, classifier utilities, repeat detection experiment, homepage, queue functionality for application, help page
- **Michael Mathews**: updating feeds, setting up crontabs, fetching/parsing of feeds and individual articles

External Code

*Python:*

[Django]
[http://www.djangoproject.com/]
Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

[Stemmer]
[http://pypi.python.org/pypi/stemming/1.0]
Stemming is done using Stemming 1.0 package for Python. Stemming was used with the intention to increase precision of the classifier.

[Feed Parser]
[http://www.feedparser.org/]
Universal feed parser is a python library for parsing feeds.

[HTML Parser]
[http://www.crummy.com/software/BeautifulSoup/]
Beautiful Soup is a Python HTML/XML parser designed for quick turnaround projects like screen-scraping.

*JavaScript:*

[jQuery]
[http://jquery.com/]
Query is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.

[Readability]
http://lab.arc90.com/experiments/readability/

Makes reading more enjoyable by removing clutter that surrounds what you are reading.

Readme for Read.me

Project URL

http://readme.cs.washington.edu

Browser Compatibility

The service is compatible with Firefox and Chrome (Internet Explorer has compatibility issues).

Deployment Instructions

The application was deployed on Cubist’s apache server with mod_wsgi.

1. Mod_WSGI: an apache module that provides for hosting Python Web applications. Depending on the platform, the installation process can be slightly different. For cubist server, we had to install the module for fedora. The module for fedora can be obtained at http://download.fedora.redhat.com/pub/epel/5/i386/repoview/mod_wsgi.html From this url, a rpm package can be downloaded and installed. (rpm -ivh package.rpm)

2. Apache configuration: in the case of cubist server, apache has been installed, and the configuration file is located at “/etc/httpd/conf/httpd.conf.” a. Once mod_wsgi was installed, Apache had to be told to connect to the module. This was done in Apache configuration file by “LoadModule wsgi_module modules/mod_wsgi.so.”

b. A virtual host is needed since cubist hosts multiple domains using the same IP address. This is done by <virtualhost 128.208.1.51:80> </virtualhost>. All other configurations must be encapsulated within these 2 tags.

c. WSGI Daemon: wsgi runs in daemon mode for the purpose of resetting the application on Apache without resetting Apache. Daemon mode also offers faster performance because we don’t have to tune the Apache MPM settings. Daemon mode is enabled by:

wsgirestrictstderr off
wsgidapemonprocess django
wsgiprocessgroup django
d. Direct Apache to the Readme application: we need to tell Apache to direct to our application using mod_wsgi. This is done by “wsgiscriptalias /www/htdocs/projects/10au/cse454/c/SmartRSS/apache/django.wsgi.” This alias will match the request to root directory to wsgi setting.
WSGI setting: is in django.wsgi file. The syntax of this file is Python syntax. What this file does is to add the paths of the application to Python search paths, and to point the Django settings module to our application settings module.

e. Media files: Django does not serve media files, so this is done separately by “alias /media/ /www/htdocs/projects/10au/cse454/c/SmartRSS/media/” and “alias/admin_media/ /usr/lib/python2.6/site-packages/django/contrib/admin/media/” to serve image contents.

f. Edit settings.py to configure the application settings

g. For a detailed configuration, please see the attached apache.txt file.