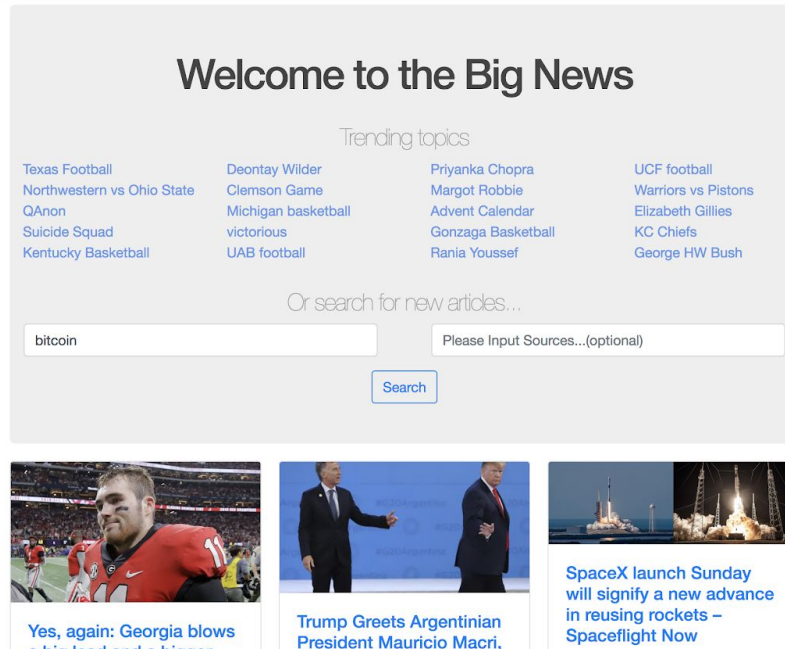


Second Iteration Demo

BIG NEWS



1 Demo

Date/Time: Tuesday, December 1, 2018, 3:00pm

Challenges

- As in the first iteration demo, we did not know how to deal with the infinite loop created by the logs from the static analysis in the pre-commit hook. Kiran suggested that we push these to a separate branch.
- In addition, for this iteration we need to show a history of logs and not just the most recent one. In other words, we need to append the log name with either a timestamp or some kind of increment.
- For the login page, we need to remove frontend elements for “forgot password” and “remember me” since this functionality does not actually exist.
- Test for modeling: collect the number of the correct classifications/total classifications
 - Assert overall accuracy > 70%
 - Instead of asserting individual articles

2 User Stories

Time Estimation

Story

As a student, I am interested in the topic about tech news. I want to spend 30 minutes every morning to find out the latest news in the industry.

Use case

Primary actor: registered user

Basic flow:

1. User logs in
2. System queries News API for top news stories
3. System uses Goose3 to extract headline, URL, text, and body of text.
4. System uses body of text to estimate words by words-per-minute
5. System displays article information, including calculated time estimate to User.

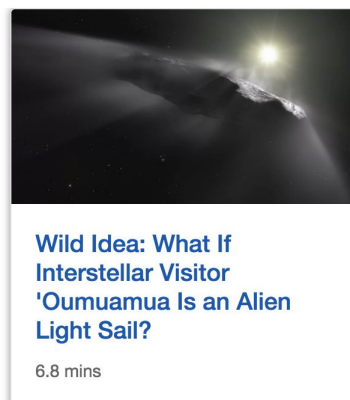
Alternative flow:

- 1a. User does not have account and needs to sign up.

Implementation

When we start the server, we use News API to produce the articles based on what are most trending via its [/top-headlines endpoint](#). From the URLs that we are given, we use the [goose3](#) python package in order to parse the contents of each article. As part of goose3, we are given the text contents of the article, and from that, we can use a set words per minute (200 wpm) to estimate how long the average user will take to read the article.

Calculation formula : Estimation time = words / 200 (words per minute)



This estimated time is then rounded to the first decimal place and displayed on the article card.

Conditions of Satisfaction

When I see a card that says it will take me 6.8 minutes to read, reading it at 200 words per minute should take me the listed amount of time. That is, the article is $6.8 * 200 = 1360$ words.

Testing

We test this in [test_article.py](#), which mainly focuses on testing the parsing functionality.

Equivalence Classes

Valid:

- **Input:** article of known length (about 500 words) via its URL
- **Output:** 2.5 minutes for someone who reads 200 words per minute.

Invalid:

- **Input:** string that is not a URL (e.g. "nytimes article", "")
- **Output:** not a URL

- **Input:** empty input (e.g. "")
- **Output:** not a URL

Search by Keyword

Story

As an entrepreneur, I am interested in industry dynamics. I want to keep up-to-date on all the news about my industry competitors.

Use case

Primary Actor: registered user

Precondition: User has registered authentication.

Basic flow:

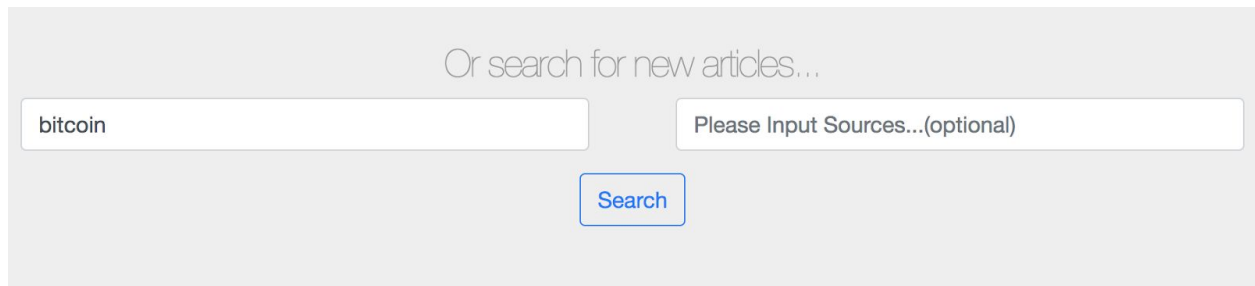
1. User inputs keyword(s) into the "keyword" field of the search form
2. System takes keyword and queries the NewsAPI with it
3. System takes articles returned by NewsAPI and extracts information about article
4. System presents this information to User on cards.
5. User clicks on article that they would like to read.

Alternative flow:

- 2a. NewsAPI cannot find any articles related to that keyword, so System informs User
- 5a. User cannot find an article they like and tries different keyword(s)

Implementation

In the search box, the user can input a word or words they would like to search for:



This is sent to the `/search` endpoint of our Flask application, which then uses the News API's [/everything endpoint](#), which searches for all articles using the input string as the `q` request parameter. This means that we hit the News API every time the user enters a new search; there is currently no persistent corpus of articles.

When we hit the News API `/everything` endpoint, we have the parameter `{'sortBy': 'publishedAt'}`, meaning the News API returns the most recent articles, but we can also change this to `{'sortBy': 'relevancy'}` to ensure that our results are more related to the user's search query.

Note: This story has changed from our revised proposal. We are not trying to determine the intent of the user from their search queries. Rather, we just show results based on an API.

Conditions of Satisfaction

Based on a list of company names that I give, I want the application to show me articles about those companies. For example, this means that if I give it the keywords “Apple, Google, Microsoft”, it will show me popular articles about these keywords.

Testing

The testing is in the [test_news_server_search.py](#) which tests both the searching by keywords and searching by sources functions. First we check whether the user has logged in to use our services. Second, we test valid and invalid equivalence classes including searching keywords and sources at the same time.

Equivalence Classes

Valid

- **Input:** “bitcoin” is our search query
- **Output:** a list of URLs — each of these URLs contains an article that contains “bitcoin” or “Bitcoin” in the headline or text.

- **Input:** “bitcoin bank” is our search query
- **Output:** a list of URLs — each of these URLs contains an article that contains at least one of [“bitcoin” or “Bitcoin”] or [“bank”] in the headline or text.

Invalid

- **Input:** empty
- **Output:** redirect to regular homepage with popular headlines overall
- **Input:** a string contains unrecognizable keywords
Output: no articles

Trending Topics

Story

As a busy person, I want to know what to focus on because I have too many news articles to read. I want the app to show me top keywords, so that I can pay attention to the most trending ones. This may take the form of, for example, a word cloud that highlights key words from a group of articles.

Use case

Primary Actor: registered user

Basic flow:

1. User logs in
2. System queries Google Trending Topics API for currently trending topics
3. System displays this to User

Alternative flow:

- 1a. User does not have account and needs to sign up.

Implementation

Trending topics			
Election Results	Idris Elba	Andrew Gillum	Joel Quenneville
Where Do I Go To Vote	Bob Hugin	I Voted Sticker	Democratic Party
Jamal Murray	Republican Party	Exit polls	California Propositions
Who Should I Vote For	voting location	Jason Garrett	Ind vs WI
Da Baby	my Polling Place	Issue 1 Ohio	Incumbent

This is currently implemented by static pulling from [Google Trending API](#) on server startup and storing them on Mongo DB. We serve the same trending words to any user that logs in, but in a future iteration, we would like to make sure these results are periodically refreshed.

As a user interface upgrade, we would also like to make the sizes of the words change depending on the popularity of each words. This can be determined based on the Google Trending API, which also returns the number of searches (for example, `200,000+`).

Conditions of Satisfaction

I can get a rough idea of what is most popular in the news right now from a quick glance in real time (similar to Twitter trending topics maybe), so that I can dig deeper later.

Testing

We test this in the [test_news_server_trending.py](#), which mainly tests our requests to Google Trends API.

Equivalence Classes

Valid:

- **Input:** a request for Google Trends API
- **Output:** a dictionary of trending topics

Invalid:

- **Input:** no request
- **Output:** no trending topics

Personal Analytics - Time

Story

As someone who may look at a lot of different topics, I want to know what I am spending my own time on (which may not exactly be what is popular for the general population).

Use case

Primary Actor: registered user

Precondition: User has registered authentication.

Basic flow:

1. User clicks on article
2. System stores article's ID on MongoDB user history database
3. User goes to Analytics page
4. System goes to MongoDB user history database and collects all entries of articles read by User
5. System does basic analytics, e.g. sum of time of all articles read (clicked) on each day
6. System displays these numbers to the User

Alternative flows:

- 1a. User doesn't read any articles
- 3a. User reads multiple articles before going to Analytics page.

Implementation

When users click on articles in the main page or “Search” view, we will add the article to the database as seen by the user. Using the time estimation for each article, we display the date and time read for that day, the topic summary for articles in the reading history, and the details of the reading history including categorie, url and time estimation for each article:

Reading time...

12-01-18: 9.0 minutes

Topic...

business: 2

sports: 1

Reading History...

category: business

<http://techcrunch.com/2018/11/25/ohio-becomes-the-first-state-to-accept-bitcoin-for-tax-payments/>

1.4 minutes

category: sports

<https://www.ajc.com/blog/mark-bradley/yes-again-georgia-blows-big-lead-and-bigger-chance/OT4mG0wTRt1vYqEIM65O9L/>

5.3 minutes

category: business

<https://www.thisinsider.com/police-stopped-an-autopilot-driven-tesla-with-drunk-driver-asleep-2018-11>

2.3 minutes

In the final iteration, we plan to display personalized data visualizations using [D3.js](#) to show to total amount of time the user spends every day, supposing the time estimate is correct.

Conditions of Satisfaction

I want to be able to find how much time I am spending on my news consumption, e.g. how much time I spend reading daily, how many articles I’m reading daily, etc.

Testing

This is in the file [test_news_server_analytics.py](#), in which we tests equivalent classes of the functions including adding browsing history into the database, reading the browsing history from the database, personal analysis of time estimation for one day and topics that a user has read.

Equivalence Classes

Valid:

- Input: a reading history record for a user including article url, time estimation, category
- Output: the reading history has added to the database successfully
- Input: a test username (has read several articles)
- Output: the user’s reading history including the date, url, category, time estimation for each article the user read
- Input: a test user name (who has read several articles on one day), date

- Output: the user's total reading time for that day

Invalid:

- Input: a test username (read no article)
- Output: 0 mins for user's reading time, no reading topics, no reading history

Search by Sources

Story

As a journalist, I want to make sure I'm reading news from lots of different types of publications (e.g. CNN and Fox News).

Use case

Primary Actor: registered user

Precondition: User has registered authentication, and the User inputs a valid source ID

Basic flow:

1. User inputs source ID into the "source" field of the search form
2. System takes source ID and queries the NewsAPI with it
3. System takes articles returned by NewsAPI and extracts information about article
4. System presents this information to User on cards.
5. User clicks on article that they would like to read.

Alternative flow:

- 1a. User inputs both source ID as well as keyword in search form.
- 2a. NewsAPI cannot find any articles related to that source, so System informs User
- 5a. User cannot find an article they like and tries different source(s)

Implementation

Users can choose to search for specific source list besides searching articles by keywords in the search box, and the list will be sent to the `/search` endpoint of our Flask application, which then uses the [/everything endpoint](#), which searches for all articles using the input string as the `sources` request parameter. Currently, our user interface requires that the user inputs the source formatted as expected by the [News API](#); in a future iteration, we can have it such that the sources are given in a list to the user to pick from, so they don't have to worry about formatting.

Or search for new articles...

Conditions of Satisfaction

I can give a list of sources that are interesting to me, and the app will give me articles from all those sources.

Testing

The testing is in the [test_news_server_search.py](#) which tests both the searching by keywords and searching by sources functions. First we check whether the user has logged in to use our services. Second, we test valid and invalid equivalence classes including searching keywords and sources at the same time.

Equivalence classes

Valid:

- **Input:** "cnn, abc-news"
- **Output:** a list of URLs — each of these URLs contains an article that either from CNN or from ABC News
- **Input:** "cnn"
- **Output:** a list of URLs from CNN
- **Input:** "Trump" as keyword and "abc-news" for sources
- **Output:** a list of URLs -- each of these URLs contains an article that contains "Trump" in the headline or text, and are from ABC News.

Invalid:

- **Input:** empty input
- **Output:** no article

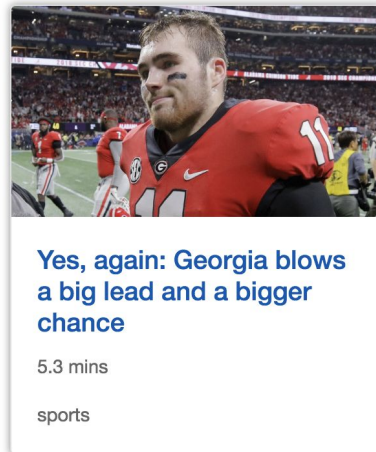
Personal Analytics - Topic Modeling

Story

As a consumer of various news articles, I want to know what topics I tend to read about, so I can adjust my behavior accordingly. For example, if I read a lot about science, I want to know this, so maybe I can also pay more attention to articles about politics.

Implementation

For NLP, we plan on composing a corpus by scraping the NewsAPI based on different “category” parameters (so that it is labeled training data). From this corpus, we will build a topic model based on words (likely naive bayes algorithm). Then we can label new articles from each day based on the category produced from this topic model.



Conditions of Satisfaction

As a user, I want to make sure that most of the articles in my “science” category are related to science. I would be satisfied if the actual accuracy rate of the topic is up to 90%.

Testing

The testing is in the file -- [test_model.py](#) to test our NLP algorithm for article category.

Valid:

- **Input:** We test using some number, say 10, of articles from our testing data
- **Output:** We make sure that our NLP algorithm properly grouped articles of the same category together with some error threshold, say 20%.

Periodic Refreshing of Database (for final iteration)

Story

As a user who checks for news daily, I want to make sure I see new articles every day.

Implementation

Our MongoDB server will periodically refresh the articles on the main page every day.

Conditions of Satisfaction

At least within 24 hours, there are always different articles displayed on the main page. We may later adjust this to have shorter periods (for example, refreshing every 30 minutes).

Testing

- **Input:** Current timestamp
- **Output:** Boolean variable — whether or not to do a fresh query of the News API to repopulate the main page

Continuous Integration

Pre-commit

As in the [example](#) posted by Professor Kaiser, we set up our pre-commit hook in [/bin/git-hooks](#). As shown in the demo, we write to time stamped files kept in the [/logs](#) folder of our logs branch. We test using the [pytest](#) framework in the [tests](#) folder of our repository. Details on how to push to code or logs are found in our [README](#).

Coverage Report

In order to produce the test coverage report, we use the [pytest-cov](#) package. This is already part of the pre-commit hook and the Travis CI automated testing.

From [#72.1](#)

```
----- coverage: platform linux, python 3.4.6-final-0 -----
Name                               Stmts  Miss  Cover
-----
Article.py                         93      2   98%
models.py                         121     88   27%
news_server.py                    138     32   77%
parsers.py                        16      0  100%
tests/test_article.py             42      0  100%
tests/test_model.py               60      6   90%
tests/test_news_server.py         34      0  100%
tests/test_news_server_analytics.py 83      0  100%
tests/test_news_server_search.py  59      0  100%
tests/test_news_server_trending.py 23      0  100%
trainingSetCollector.py           19     19    0%
-----
TOTAL                             688    147   79%

===== 29 passed in 985.99 seconds =====
The command "py.test --cov=. -v tests/" exited with 0.
```

Post-commit

We used [Travis CI](#) for continuous integration. As in the pre-commit, it tests using the [pytest](#) framework (with the [pytest-cov](#) package) in the [tests](#) folder of our repository. After running on the cloud at [travis-ci.com](#), the reports are pushed to [GitHub](#).

GitHub Repository

URL: <https://github.com/XJBCoding/NewsServer>

This is a public repository.