TDT4215: Group Project Report

Ole Halvor Dahl, Alexander Gård, Maurice Kilgus, Atle Frenvik Sveen

7th April, 2017

1 Introduction

This report summarizes our work on the *Angry Planets* news recommender server. This recommender server was written as a group project for the course TDT4215, Web Intelligence at NTNU. The project is part of the CLEF NewsREEL Replay challenge [1], in which teams around the world compete against each other in creating the best performing news recommender engine based on real data gathered from several large-scale news publishers.

This report is structured as follows. In the background-section we cover the relevant theory in recommender systems and describe the domain of news recommendations, and the CLEF NewsREEL Replay challenge to some extent. The next section, Recommender Implementation, covers the issues and limitations placed upon our solution, the actual design of our solution, the libraries and frameworks we used, and examine the actual implementation of our recommender in detail. The evaluation section covers our experience working on the NewsREEL challenge and describes the results we obtained. The report concludes with a review of why our recommender performed the way it did, what we could have done to improve the recommender server, as well as the issues encountered during the work. We also point out what we learnt from this exercise, what worked and what did not.

2 Background

The aim of the CLEF NewsREEL Replay challenge is as follows: "whenever a visitor of an online news portal reads a news article on their side, the task is to recommend other news articles that the user might be interested in" [2].

The replay-aspect of the challenge means that the recorded, live, data fed to the recommender system is stored in a log-file and sent to the recommender server through the *Idomaar* [3] framework. Idomaar will also evaluate the performance of the recommender server, which gives us a way to assess the strengths and weaknesses of our recommender.

The main objective of a news recommender system is to suggest news articles to individual users, that have a high probability of being clicked on. An optimal recommender yields a high amount of clicks. This is often measured using a metric called *Clickthrough Rate* (CTR) [4]. These recommendations can be obtained using various means. The two main approaches we explored are *Content based filtering* and *Collaborative filtering.* Content based filtering is based on the principle that if most users who click on article 1 also click on article 2, then new users who click on article 1 will probably also click on article 2. Collaborative filtering is based on the fact that the interests of people tend to cluster. With this method, users with similar history of clicks are clustered together. These clusters are then used to recommend articles that most users in the cluster have clicked on, to users in the cluster who have not previously been showed the article.

The news recommendation domain has some particular characteristics. Several of these influence the design of a recommender designed to work on news items. The CLEF NewsREEL challenge also imposes some limitations on our recommender. While an optimal news recommender would balance all these requirements in perfect alignment, this is utopian in practice. What we have aimed for is a practical balance of all factors, so that the readers receive helpful recommendations.

One defining characteristic of news domains, is age. Old news are nearly never relevant, this is especially true in a recommender system. This means that the age of an article should probably influence its relevance. News articles are also subject to updates, a journalist may update some facts in an article, add new paragraphs, or correct spelling mistakes. This means that a news recommender system should be able to deal with updates to items, a fact that is also reflected in the NewsREEL challenge.

In contrast to many other domains where recommender systems are applied, the news domain usually does not have the advantage of explicit ratings from users. As a substitute, if the user visits an article this is counted as a positive rating. This method of gathering data is called *implicit feedback* [5].

Another constraint, imposed by the NewsREEL challenge, is that the recommender should work for several domains. That is, the same recommender server is used to recommend articles from several different news publishers. The system should therefore separate these domains, as a system that recommends an article from a competing news publisher is not desirable from a publishers point of view. Of course, if some of the publishers are collaborating, they may want to recommend articles from each other, but this possibility is not implemented.

There is no training phase in the NewsREEL challenge. The result of this is that the cold-start problem becomes even more important to handle correctly.

2.1 Envisioned use of the Recommender Server

The NewsREEL challenge omits most details on how the actual recommender server is supposed to be working on a real-life situation, but nevertheless we feel that at least a mental picture of this is helpful when implementing the server. In the following we describe our understanding of how a recommender server such as ours is supposed to work.

We envision the recommender server as a plug-in solution to existing news publishing systems (or Content Management Systems, CMSes, in general). The administrative part of a CMS, where journalists publish, edit, and delete articles are responsible for telling the recommender about changes to items through POST requests of the *item_update*-type. This may be deeply integrated into the CMS, or loosely linked using web hooks or similar integration points.

Capturing of user events and data is done trough embedded JavaScript code on the article pages. Whenever a user navigates to a new page, a request is sent to the recommender server with appropriate data.

In the same manner we envision the actual display of recommendations is implemented. An area of the page is reserved for displaying recommendations, and the content of this area is based on a request sent to the recommender server. The recommender server is sent information about the current article and user (if the user is logged in).

3 Recommender Implementation

As discussed above, we have identified some key characteristics of both the news domain in general and the NewsREEL Challenge in particular. Our recommender tries to deal with these constraints in order to provide the best possible recommendations. In the following sections, we discuss how our recommender is implemented in light of these constraints and provide a general overview of the recommender strategy chosen. The technical architecture of the recommender server will also be covered in detail.

3.1 Issues and Limitations

During our work on the recommender we have identified several aspects that influenced our design choices. In the following we list these and discuss how they influenced the final product.

3.1.1 Separation of Domains

The NewsREEL challenge dataset contains data from several news portals, identified by a unique domain id. In order to not mix recommendations across domains, we have chosen to use separate recommender processes for each domain. In practice this means that we extract the domain id from each incoming request, and dispatch the request to a handler specific to each domain. In this way we achieve separation of domains.

3.1.2 Cold Start Problems

For both collaborative filtering and content-based recommenders, cold start problems may be an issue. In the context of news recommenders as modeled by the NewsREEL Challenge this is especially important, as there is no training phase. As soon as the recommender is started it is expected to reply to recommendation requests.

We chose to handle this by utilizing *recency*. For the very first recommendation requests we simply return the most recent articles. In the NewsREEL data-set a frequent problem was that a recommendation request referred to an item that we had no knowledge of. This case was also handled using the recency approach.

3.1.3 Item Updates and Deletions

The news domain is a "living" domain, in that news are, by their nature, not static. New articles are added constantly, and older articles are updated or sometimes deleted. This is also reflected in the NewsREEL data-set. This fact has consequences for our recommender system and we need to keep track of changes to items, thus updating our matrices to keep everything synchronized.

3.1.4 Incomplete Texts and Language

In order to utilize content-based recommender techniques to the full, a good starting point is access to the full article text. However, the NewsREEL Challenge only provides article excerpts in their data-set. For simplicity we have chosen to treat the excerpts as the full texts. Another issue with the data-set is that it is based on German newspapers, and the texts are thus in German.

3.1.5 Time and Knowledge Restrictions

In the beginning of the project we had high hopes of implementing a performant and accurate recommender that would approach state-of-the art recommenders. However, we quickly learned why companies such as Netflix offer prizes in the million dollar range for small improvements to their recommendation algorithms. Recommender systems are complex, and with no prior knowledge in the project group, the setbacks caused by the Idomaar framework and data-set made us realize that we would not be able to implement a world class news recommender.

This meant that we adjusted our goals and focused on creating a recommender that could at least beat the "n-most-recent" sample implementation provided. In this way we felt that we would get some experience using existing frameworks and adapting them to our specific needs, much like what would be done in a real-life project constrained by both time and budget.

3.2 Recommender Design

For the reasons stated above we ended up with two different recommender implementations. We realised quite early that in addition to implementing an actual recommender system a large part of the task was to make sense of the incoming requests and be able to return sensible data, as well as to handle the separation of domains. In order to test the complete server we chose to re-implement the "n-most-recent" algorithm provided in the sample solution provided by CLEF [6]. This simple algorithm meant that we could make sure that we correctly parsed requests and replied in the correct format, and also provided a baseline to beat in terms of clickthrough rate.

However, just implementing a most-recent algorithm isn't exactly rocket science (nor particularly related to the curriculum), so we chose to implement a content-based recommender as our actual recommender once we made sure the rest of our code was working. We chose to implement a content-based recommender due to simplicity. We are aware that most successful news recommender systems use a combination of content-based and collaborative filtering algorithms. The provided data-set also contains much useful information for collaborative filtering. However, the user-related data in the data-set is provided using a complex structure referred to as *vectors*. While we could have spent more time trying to make sense of the data structure we prioritized to get going on an actual implementation.

An important aspect of our design is the emphasis put on two main concepts:

1. Separation of HTTP-protocol and JSON from implementation
2. Recommender interface implementation

By separating out HTTP-related parsing logic from the actual implementation we end up with a structured application that is usable outside of a web-environment, and one that is easy to test. While the ability to run the recommender using different protocols may not be relevant to this project, we found that the ability to write simple unit-tests proved useful, especially since running the Idomaar framework takes 5-10 minutes.

The use of an interface to describe the actual recommender implementation allowed us to easily swap recommendation algorithms. Although we ran short of time, and did not get a chance to test other algorithms, we did use this ability in order to test the rest of the server.

3.3 Tools, Libraries, and Frameworks

Given both time and knowledge constraints we have chosen to make use of existing libraries and frameworks where such tools exists and serve to simplify our task. We believe that this approach is beneficial, as this shifts the focus from technical specifics to the broader concept. This subsection will introduce the various tools used in our implementation, and explain how each of these tools interact with each other in the finished product.

3.3.1 Python

When it comes to creating web applications, there are quite a few languages to choose from. For our project, we decided on using *Python* [7] for both the web application, and for the recommender engine. Python comes with a large amount of machine learning libraries that can be used for recommendations, and several web application frameworks. While other languages such as Java would also be a good fit for a project like this, the decision ultimately fell on Python for for its rapid prototyping capabilities and clear language syntax.

3.3.2 Flask

Flask [8] is a small web framework written in Python. Flask's main focus is on being lightweight and is referred to as a micro-framework due to this fact. In addition to being lightweight, Flask also allows a great degree of flexibility when it comes to implementing web applications. There are other more well known web frameworks such as Django [9] and Pyramid [10], which comes with a large amount of different functionality included. However, due to the nature of this exercise, Flask was chosen. The reasoning behind this decision was that much of the functionality that was included in the larger web frameworks, would not be needed for our solution, and would unnecessarily make the things more complicated. The flexibility that Flask provides also makes development much easier, as we are not tied to any of the design decisions that comes with Django or Pyramid.

3.3.3 Pandas

Pandas [11] is a Python library that provides easy-to-use data structures and several tools for data manipulation and analysis. Pandas is a well-known library when it comes to managing data in Python. One of the reasons we chose Pandas for our project is because it simplifies data management greatly. However, the main reason for choosing Pandas was because of the library we used for our recommendations integrates really well with Pandas, which simplifies the data management even further.

3.3.4 Scikit-learn

Scikit-learn [12] is a Python library that contains efficient tools for data mining and analysis. Scikit-learn also provides several different kinds of tools for machine learning. Scikit-learn is a well known library when it comes to machine learning in Python, and contains all the tools that we needed for our recommender. There are other Python libraries as well, that are specifically made for recommendations. Some of those we looked into were: *Crab* [13], *pysuggest* [14] and *python-recsys* [15]. The reason for choosing Scikit-learn over these libraries was the flexibility that Scikit-learn provided, with a plethora of different algorithms that could be used for recommendations. Scikit-learn is also very well documented, which makes finding information during development significantly easier.

3.4 Recommender Implementation

The NewsREEL challenge and the Idomaar framework defines the outer boundaries of our recommender system. While the recommender system should be able to function in a real-life-setting that uses the same interface as the Idomaar framework, we have limited our focus to handling requests from the Idomaar framework. In some cases the resulting architecture of our recommender server may end up following non-standard web-practices.

In the following sections we describe the interface that Idomaar framework imposes and the consequences this has for our server. While these requirements are not explicitly stated in a single document, we found that a careful examination of the Idomaar framework itself, and the sample implementation of a recommender server and the ORP Protocol documentation[16] served as a good starting-point.

3.4.1 HTTP Interface

The Idomaar framework sends all requests to the recommender server as a HTTP form-encoded POST request to a single HTTP-endpoint. The form-data sent contains two fields: *type* and *data*. The value of the *type* field is a string representing an enum with the following possible values:

- item_update
- event_notification
- recommendation_request
- error_notification

This value represents the type of request sent to the server. This type also determines the structure of the second field, *data*. This is a JSON-encoded string containing data relevant to the request type. In order to separate concerns we choose to split the logic of our server in a way that abstracts the format and encoding of input-data from the actual recommender system. Figure 1 shows this layout. The HTTP handler receives all requests to the endpoint and tries to find the type and data fields. The data is parsed as a Python dictionary. If this succeeds the data is passed to *handle_idomaar_message*, a function that dispatches the message to the correct handler based on the type, which in turn is responsible for returning an appropriate response.

The data-payload of the request can either be an *item* or a *vector*, depending on the request type. This data is parsed using a specialized a Python-object after checking the request type, and then passed further on.

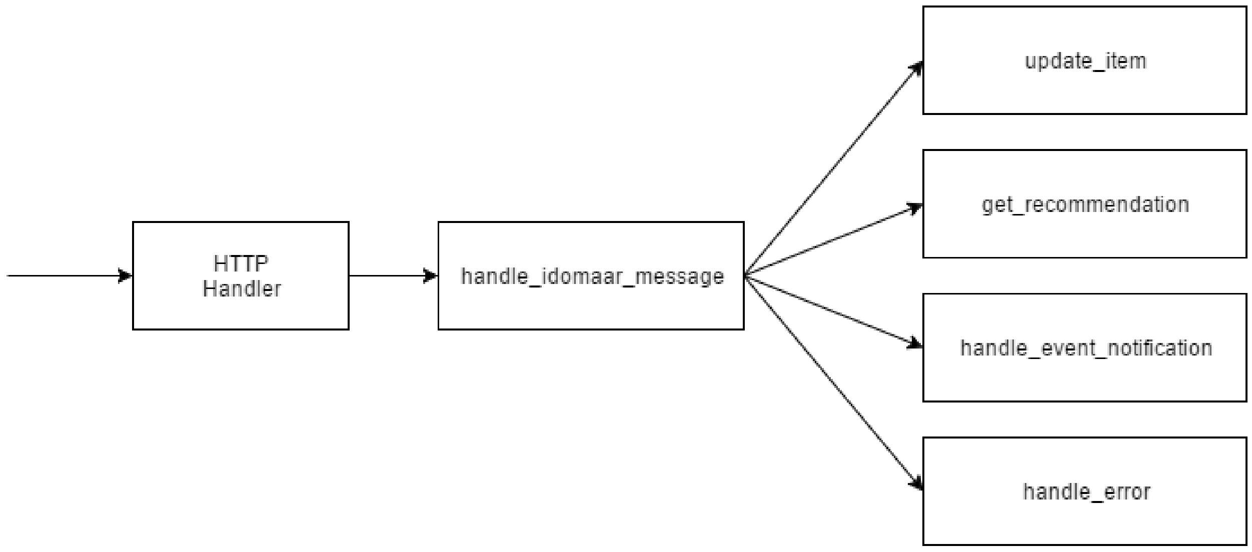

Figure 1: HTTP Request handling overview.

3.4.2 ItemTable Implementation

The actual handling of a request is then delegated to an *ItemTable*, which handles the different request types. We chose to implement this as as separate class in order to be able to separate items from different domains at an early stage. Each domain gets assigned its own ItemTable, which in turn handles the different request types. In practice the ItemTable delegates the handling of these requests to a Recommender object. Although Python does not have a specified way of describing an interface, our solution allows us to use the concept of an interface; should we want to use a completely different recommendation strategy all that is needed is to implement another recommender class with the required methods and swap out the recommender used in the ItemTable.

On order to check that the recommender worked as expected without diving into recommender implementation we first developed a *MostRecentRecommender*, which just returns the n most recent items. This was later replaced with a *ContentBasedRecommender*, described in the following subsection. This flexible architecture allowed us to change one part of our system without changing the other parts. We've only chosen to define the methods deemed necessary for our implementation, but in principle this interface should cover all possible methods

3.4.3 ContentBasedRecommender Implementation

The ContentBasedRecommender is the class that implements the recommender logic of our system. In order to fulfill the Recommender interface that ItemTable specifies it implements the following public methods:

- **add_item**: Add a new item to the table
- **update_item**: Update an existing item
- **remove_item**: Remove an item from the table
- **has_item**: Check if there is an item with the given id in the table
- **get_recomended_items**: Get recommendations from the table

These methods provide enough functionality to implement a content-based recommender. Each of the methods that operate on items manipulates a *Pandas* data frame, which for simplicity contains two fields: *id* and *text*. The id corresponds to the item id we receive in the request and the text field is made up by combining the item attributes *title* and *text*. For this project we chose to let the Pandas data frame containing our data reside in memory. This is not a good solution for a production-ready system, as it would not allow for splitting the load on several servers, nor will it provide persistence in case of system crashes.

The *get_recommended_items* method receives a vector, from which we extract the item id the user is currently looking at, and the *limit* of recommendations requested. If the item id is not stored in the data frame we return the n most recent items, otherwise we proceed to use the content-based recommender.

The algorithm used is rather easy; first we create a TF-Idf matrix using the text-field. The *Scikit-learn* library provides a *TfidfVectorizer*, which handles stop-word removal, word analysis, and building of the TF-Idf matrix. The library provides an English-language stop-word list, but since the data we are dealing with is in German, we had to supply our own list. This list contains words that are frequently occurring in all texts without adding any "meaning" to the text. The Tf-Idf (Term Frequency - Inverse Document Frequency) matrix is a calculation of the frequency of the words in a specific text (a specific news article) compared to the frequency of the same word across all documents. From the Tf-Idf matrix we compute a matrix of cosine similarities, and extract the ids of the *limit* most similar items.

4 Evaluation

In this section we will discuss the results we obtained and provide an overview of our experience working on this task

4.1 Working with the NewsREEL challenge

Clef NewsREEL Replay is a framework with a competitional objective to evaluate recommender solutions against huge data-sets. People from all over the world can participate and compete in order to find the best recommender. The evaluation results will be compared using the so-called *prediction accuracy*, which is the "number of recommended items predicted by an algorithm that corresponds to items that were clicked by a user, normalised by the total number of requests for recommendations that were sent to that system" [1].

However, setting up the Idomaar framework and implementing our recommender did not go as smoothly as expected. We would like to discuss the problems we encountered during this project. At first, the tutorial provided by NewsREEL to set up everything didn't work with the provided example data-set, even if you followed the guide step-by-step. This was very frustrating at the beginning of the project and was also very time costly. After spending a lot of time on debugging the orchestration of a dozen frameworks managed using Puppet and shell-scripts in Vagrant a new tutorial was released that clarified this. We felt punished for starting early.

The tutorials both referenced an old sample data-set, which did not match the format of the dataset supplied for this challenge, and the Idomaar framework did not work on the new data-set. Luckily, a fellow student described how to patch the Idomaar framework to make it work with the new data. However, we where never able to use the complete challenge data-set, and had to rely on the nr100000.log-file, an excerpt with 100.000 events.

Another frustration was that the actual format and structure of the data sent to our recommender server was not documented in detail. The provided ORP Protocol documentation [16] did provide some understanding. Especially the data structure and the meaning of vectors should have been explained more thoroughly. The baseline recommender [6] provided by NewsREEL was referenced in the tutorial. This code was poorly structured and did not help our understanding.

The last major frustration was related to the Idomaar framework as well. The process of running the 100.000 events data-set on our machines took about 5-10 minutes. In addition, we found that subsequent runs of the evaluator produced different results. This problem was avoided by starting and stopping the virtual machine between runs. In combination this meant that the feedback-loop from tweaking our recommender to verifying the results took around 15 minutes. This does not encourage tweaking of parameters.

4.2 Recommender Evaluation Results

In the end, we managed to get the evaluation running, based on the implementation described in section 3. The results of the evaluation can be seen in table 1.

Publisher Domain	Clicked Recommendations	Total Recommendations	Errors	CTR
13554	0	1398	0	0
694	0	474	0	0
1677	3	2277	0	1
35774	163	17261	0	9
418	0	1332	0	0
all	166	22742	0	7

Table 1: Idomaar results of running the nr100000.log file against our recommender server

As can be seen in table 1, there is a significant contrast in the results from the different publisher domains. The domain with id 35774, has a CTR of 9, which is quite a lot, compared to the other domains which barely produced any results at all.

One reason for these results might be that this domain has a lot of data tied to it, and therefore has a good foundation to provide more accurate recommendations than the other domains. There cannot be drawn a conclusion whether or not there is a positive correlation between the amount of data collected and CTR, with just one sample, but it might be worth looking into in the future.

While we can see that there is a large contrast between the 35774 domain and the other domains, we are still unsure of how good a CTR of 7 is. From this image [17], we can see that the best results from CLEF NewsReel 2016 was a CTR of 1.17%. Compared to our CTR of $7/1000 = 0.7\%$, it is not all that bad, even though there is still a considerable difference.

A good way of evaluating our recommender would be A/B testing. This would allow us to directly observe how well our recommender performs compared to other recommenders. The benfit of using this method is that we can classify the results of our recommender right after testing, which would give us immediate feedback on how well the recommender performed. However implementing A/B testing would take a lot of time and resources, as A/B testing would require its own infrastructure to perform the evaluations.

Another method of evaluation would be to see how well the recommender performs on a live data stream. Since live data is what the recommender is ultimately designed for, it would give more accurate results on how well it performes in a real-life scenario. However, it would make it more difficult to test how well the recommenders performs amongst each other, as the data is continuously changing.

5 Discussion and Conclusion

Our implementation of a news recommendation server based on an ongoing, online challenge using real-life data from actual news sites has been an interesting project. Most of all it has served as a rather overwhelming introduction to the domain of recommender systems in general, and news recommender systems in particular.

In addition it has provided an insight in a non-ideal world with delayed responses, wrong or missing information, and minor issues that delays progress for long time spans. As our team did not know each other before the project started, and we all follow different study plans and schedules, the task of coordination was also an obstacle. Digital communication and collaboration tools such as Slack and GitHub has been important tools in this regard.

We are still somewhat in the dark as to whether the results we obtained can be considered "good", but we are content with managing to produce some results at all; some weeks ago we did not think this was within our reach.

5.1 Learning Points

Before we could start implementing our recommender we needed a proper understanding of the entire system. One of the main sources for this information was the codebase of the Idomaar framework and the baseline recommender. Learning to read and understand source code written by others is a valuable asset, not only for this project, but also for future projects.

While implementing our recommender we tried out a few different recommender algorithms with different results. It was a quite interesting learning experience to see how the different algorithms that we knew how worked theoretically, performed in a real-life scenario. A key takeaway is that recommender systems are complex. In most systems one can use unit tests to ensure that parts of the system works as expected and then be quite certain that the complete system works as expected. A recommender system is more about tweaking parameters and using clever tricks. To assess the impact of such tweaks, a complete evaluation has to be run. The fact that results may not be consistent across test-runs also means that we have to rely on statistics and probability.

On the organizational side we learned a bit about priorities. We probably spent a bit too much time on technical issues and ideas, and not enough time on understanding the system. Had we been more thorough in the early stages, and focused on obtaining information and discussing recommender strategies, we would probably had more time to spend on actual recommender implementation and tweaking. As always, proper background knowledge makes it easier to come up with a good solution.

5.2 Future Work

If given more time, there are a few aspects of this project that could be further improved upon. While we do have a working recommender algorithm, it would be interesting to see how the algorithm we implemented compares to other recommender algorithms and how data points could be added to enhance the current algorithm. Since our overall architecture allows for easy switching of algorithm implementation this would be possible to test without having to refactor the complete codebase.

One interesting aspect to consider would be the location of the user in relation to places mentioned in the news stories. Using Natural Language Processing we could have extracted place-names from the stories and performed a georeferencing query to obtain coordinates for the place. The sample dataset vectors does contain a value for geo-location, which presumably is the user location. However, this location was not self-explanatory, and including a query to a georeferencing service would have made look-ups slow. In addition, there is a lot of ambiguity in place names, so this look-up could possibly have made a negative impact.

While we chose to base our recommender on a content-based approach, the provided dataset contains a lot of user information that would possibly serve as good input for a collaborative filtering approach, and also perhaps facilitate a hybrid solution that considers both the content of the articles and the preferences of the user.

Another thing we could look into is storing the recommender in a more persistent state than in volatile memory. This would allow us to, among other things, share the recommender after it has been trained, and it could also be used to run the recommender in parallel in a large-scale production environment.

It would also be worth looking in to how the results from our recommender would be affected when other data-sets are used. This might give insight in how to further improve upon our recommendations.

5.3 Concluding Remarks

This concludes our report on the news recommender system. While not everything went as smooth as we expected, and we faced a lot of challenges that where not directly related to recommender systems, we still feel that we learned something from this project. Making an attempt to implement a recommender system by using existing libraries gives a good understanding of the context such systems work in, a fact that is easy to miss when reading about theory.

References

[1] C. NewsREEL, *Tasks*, 2016. [Online]. Available: `http://www.clef-newsreel.org/tasks/` (visited on 04/04/2017).

[2] ——, *Overview*, 2016. [Online]. Available: `http://www.clef-newsreel.org` (visited on 04/04/2017).

[3] crowdrec.eu, *Idomaar, crowdrec reference framework*, 2017. [Online]. Available: `http://rf.crowdrec.eu/` (visited on 04/04/2017).

[4] Google, *Clickthrough rate*, 2017. [Online]. Available: `https://support.google.com/adwords/answer/2615875?hl=en&from=6305&rd=2&visit_id=1-636268918882964419-2840082593` (visited on 04/04/2017).

[5] D. W. Oard and J. Kim, *Implicit feedback for recommender systems*, Aug. 1998. [Online]. Available: `http://www.aaai.org/Papers/Workshops/1998/WS-98-08/WS98-08-021.pdf` (visited on 04/04/2017).

[6] A. Dai, *Andreas-dai/newsreel-template: Example code and evaluator for clef newsreel*, 2015. [Online]. Available: `https://github.com/andreas-dai/NewsREEL-Template` (visited on 04/04/2017).

[7] P. S. Foundation, *Welcome to python.org*, 2017. [Online]. Available: `https://www.python.org/` (visited on 04/04/2017).

[8] A. Ronacher, *Welcome — flask (a python microframework)*, 2017. [Online]. Available: `http://flask.pocoo.org/` (visited on 04/04/2017).

[9] D. S. Foundation, *The web framework for perfectionists with deadlines — django*, 2017. [Online]. Available: `https://www.djangoproject.com/` (visited on 04/04/2017).

[10] A. Consulting, *Welcome to pyramid, a python web framework*, 2017. [Online]. Available: `https://trypyramid.com/` (visited on 04/04/2017).

[11] *Python data analysis library - pandas: Python data analysis library*, 2017. [Online]. Available: `http://pandas.pydata.org/` (visited on 04/04/2017).

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[13] W. y Limonada, *Recommender systems framework in python - scikit-recommender v0.1 documentation*, 2017. [Online]. Available: `http://muricoca.github.io/crab/` (visited on 04/04/2017).

[14] R. N. Cabral, *Google code archive - long-term storage for google code project hosting*, 2017. [Online]. Available: `https://code.google.com/archive/p/pysuggest/` (visited on 04/04/2017).

[15] O. Celma, *Python-recsys/python-recsys: A python library for implementing a recommender system*, 2017. [Online]. Available: `https://github.com/python-recsys/python-recsys` (visited on 04/04/2017).

[16] T. Brodt, T. Heintz, A. Bucko, and A. Palamarchuk, "ORP Protocol," plista GmbH, Tech. Rep., 2016. [Online]. Available: `https://orp.plista.com/documentation/download` (visited on 04/04/2017).

[17] *Clef results 2016*, 2016. [Online]. Available: `https://image.slidesharecdn.com/newsreel-results-v04-160907155116/95/clef-newsreel-2016-results-12-638.jpg?cb=1473263590` (visited on 04/04/2017).