Recommender for Interactive Annotator Leaner System

Software Engineering Team
Task 11-791 Fall 2007

Team 3 Ñandú
Eric Davis, Steve Gardiner, Cari Sisson, Jike Yin
Table of Contents

1 Introduction
2 Workflow
3 Domain Modeling
4 Feature Exploration
5 Software Design
6 Experimentation
7 Team Process and Experience
8 Future Work
9 Deliverables

Appendix A: Supplementary Specification
1 Introduction

The goal of this project was to build a recommender algorithm for the Interactive Annotation Learner that recommended new training instances that would both reduce future effort of a human annotator and improve performance of the system. Our focus in this project was on the Elaboration and Construction phases of the Unified Process. We also dabbled in Extreme Programming and Scrum while developing our algorithm.

2 Workflow

The diagram below shows the three iterations of our project work. We began with the elaboration phase, as most of the inception phase work was defined by our assignment. According to the Unified Process, we first focused on visual modeling and use cases to help us understand the Interactive Annotation Learner domain. We used the white board to model the domain as a team and brainstormed a list of deliverables and assigned some of them to people to work on individually. We began coding in the elaboration phase, but the majority of our code was written in iterations two and three, during the construction phase.

Diagram 2.1: Iterations

This diagram shows how we actually worked, but it is very similar to our initial plan. In our first meeting we knew that we needed to begin by working to understand the domain. We used timeboxing to make sure that we had a simple working version of a recommender before we explored more complicated solutions to optimize our performance. This was important because of the short duration of the assignment and the challenges of group work.

In section 7 we will discuss the extreme programming practices we used in our construction phase.
3 Domain Modeling

In our first meeting, we worked on understanding the Interactive Annotator Learner (IAL) system and our task. We first identified conceptual classes and made a domain model.

Model 3.1: System Domain Model

We also made a simple model of our approach to the recommender itself.

Model 3.2: Recommender Domain Model
We had a lot of discussion about the number of separate data sets and whether to model the Active Set and the Training Set or just one set of documents. We ultimately decided to model the Active Set and Training Set as separate classes, since the documents in each set are distinct and different actions, such as removing and adding, are performed on each set. The sequence diagram below shows part of our discussion.

Model 3.3: Sequence Diagram

We decided that a system sequence diagram would not be useful to us because we are not working on a use case that includes any external user communicating with the system. The entire system has a user to correct and or verify the recommended documents, but our task was to model the recommender, and the "user" does not play a role at this time.

Our next step was to write a Use Case to better determine the possible flows of the system as a whole. We posited that understanding the system would give us a better feel of where the recommender fit and how to smoothly integrate it into the system.
Table 3.4: Use Case

<table>
<thead>
<tr>
<th>Use Case</th>
<th>RecommendDocuments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>The semi-supervised Document Annotation System</td>
</tr>
<tr>
<td>Level</td>
<td>subfunction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primary Actor</th>
<th>IAL Active Learning system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stakeholders and Interests</td>
<td>The human annotator wants RecommendDocuments to select documents which are not too difficult or time consuming for him/her to annotate. ACE(^1) want each group to practice software engineering methods championed in the class and create an algorithm(s) that smoothly integrates into the IAL system and performs better than the RandomRecommender baseline.</td>
</tr>
</tbody>
</table>

| Preconditions | The IAL system has learned a model \(m\) based on the data in \(D_{\text{train}}\). Active learner has annotated documents in \(D_{\text{act}}\) to make \(D_{\text{act}*}\). Model \(m\) has been applied to \(D_{\text{test}}\) and evaluated to get a score. There are un-annotated documents in \(D_{\text{act}}\). |

| Success Guarantee | \(N\) documents have been selected from \(D_{\text{act}*}\) and returned to the Active Learner. |

| Main Success Scenario | 1. IAL has learned model \(m\) based on the data in \(D_{\text{train}}\). 2. IAL applies the Recommender to select \(n\) documents from \(D_{\text{act}*}\). |

| Extensions | 1. There are no documents left in \(D_{\text{act}}\). a. The Recommender returns an empty list. |

| Special Requirements | The human annotator may have a limited amount of time available for annotation so the system should recommend an appropriate number of documents. The Active Learning system wants RecommendDocuments to select documents which are maximally distinct, and therefore most informative, from the documents which are already in \(D_{\text{train}}\). |

Our final step to better understand the system we were working with was creating a Supplementary Specification. We did not get to practice designing this document in class, and we felt that if nothing else, the glossary would help clarify some of the terms in the domain. In particular we hoped to really clarify the difference between the Active Set, the Active*Set, the Training Set, and the Development Set. We found the glossary to be particularly helpful to do just that. It can be found below. The complete supplementary specification can be found in appendix A.

\(^1\)Andy, Carolyn, Eric
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>NER</td>
<td>Named Entity Recognition --&gt; identification of people, places, and organizations in text</td>
<td>a document/documents to find Persons, Places, and Organizations</td>
</tr>
<tr>
<td>Stanford NER</td>
<td>learning algorithm uses labeled training data to create &quot;model&quot; model applied to other data to produce new labels</td>
<td></td>
</tr>
<tr>
<td>feature</td>
<td>A feature is the specification of an attribute and its value. For example, length is an attribute. ``length is long&quot; is a feature of an example.</td>
<td></td>
</tr>
<tr>
<td>IAL</td>
<td>Interactive Annotation Learner: a system to interactively learn an annotation scheme by using a minimal amount of pre(human) labeled data</td>
<td>Recommender algorithm to recommend training examples for an user to look over to improve the overall performance of the system</td>
</tr>
<tr>
<td>active learning</td>
<td>There are situations in which unlabeled data is abundant but labeling data is expensive. In such a scenario the learning algorithm can actively query the user/teacher for examples. (wikipedia)</td>
<td>available human-labeled data to train</td>
</tr>
<tr>
<td>recommender</td>
<td>prioritizes data for manual inspection, uses variety of data to inform decision on what to present to user</td>
<td></td>
</tr>
<tr>
<td>label</td>
<td>annotation, either Person, Place, or Organization</td>
<td></td>
</tr>
<tr>
<td>Active Set</td>
<td>additional labeled data with gold-standard labels. used in active learning</td>
<td></td>
</tr>
<tr>
<td>Active Set*</td>
<td>elements from Active Set to which model has been applied. recommender recommends documents from this set to add to the training set</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>algorithm to map input to desired output (here: document --&gt; labels of Person, Place, and Organization)</td>
<td></td>
</tr>
<tr>
<td>Training Set</td>
<td>Labeled data used to train the model annotations (Person, Place, and Organization)</td>
<td></td>
</tr>
<tr>
<td>Test Set</td>
<td>3rd set of labeled data used to test performance of the system.</td>
<td></td>
</tr>
</tbody>
</table>
4 Feature Exploration

The following table shows the document features we implemented.

Table 4.1: Implemented Features

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
<th>Used in Final Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnnotationCapitalRatio</td>
<td>The ratio of annotations to the number of capitalized words, excluding sentence-initial capitalized words. The feature returns one minus the ratio.</td>
<td>X</td>
</tr>
<tr>
<td>CapitalizedSequence</td>
<td>This returns true if the document contains at least one string of three adjacent capitalized words.</td>
<td>X</td>
</tr>
<tr>
<td>CapWordRatio</td>
<td>The ratio of capitalized words, excluding sentence-initial capitalized words, to the total word count.</td>
<td></td>
</tr>
<tr>
<td>CommaCount</td>
<td>This returns the number of commas in the document.</td>
<td></td>
</tr>
<tr>
<td>ConfidenceofAnnotation</td>
<td>The confidence score assigned to the tagging.</td>
<td></td>
</tr>
<tr>
<td>ContentFunctionRatio</td>
<td>The ratio of content words to a manually defined set of function words.</td>
<td></td>
</tr>
<tr>
<td>NumberofAnnotations</td>
<td>The number of annotations assigned to the document.</td>
<td>X</td>
</tr>
<tr>
<td>PrepCountRatio</td>
<td>The ratio of prepositions to the total number of words.</td>
<td></td>
</tr>
<tr>
<td>PunctCount</td>
<td>The number of periods, commas etc. in the document</td>
<td></td>
</tr>
<tr>
<td>SentInitialCapTags</td>
<td>The number of sentence-initial words with tags.</td>
<td></td>
</tr>
<tr>
<td>WordCount</td>
<td>The word count. As part of our domain modeling, we determined the range of possible document lengths.</td>
<td>X</td>
</tr>
<tr>
<td>CommatoWordCountRatio</td>
<td>The ratio of commas to the word count.</td>
<td></td>
</tr>
</tbody>
</table>

To test the features we developed, we ran small experiments using single features and pairs of features. The graph below shows the f-measure of some individual features.

Graph 4.2: Comparison of individual features
As a result of our experimentation, we chose to use only normalized versions of AnnotationCapitalRatio, CapitalizedSequence, WordCount and AnnotationCount in our final recommender.

5 Software Design

Our initial domain modeling session involved modeling the IAL framework as a whole to make sure that we all had the same understanding of what we were trying to accomplish. This modeling turned out to be quite useful in exposing small misunderstandings, and also in refining the questions we needed to answer about the objects available to our recommender at runtime.

Next we turned to brainstorming approaches for recommending documents based on our understanding of the problem, linguistics and machine learning. A common thread which kept coming up was the idea of computing features, combining the features into a unified numeric score, and then recommending the top-scoring document. From these discussions, we formed a domain model consisting of FeatureSets with numeric scores (see Model 3.2). For simplicity and time, we decided to restrict our attention to features which took on numeric values.

Translating this domain into software objects, we discovered that neither of the new domain objects was actually necessary as a new software class. Scores were simply numbers (in our case, double-precision reals). A FeatureSet could be represented as a map from features to feature-values; we decided to let features be named by Strings and represent feature-values again as double. Thus a FeatureSet was just a Map from String to double. We assume that developers concerned with different features can come up with globally unique String names for each feature; if feature name collisions cause a problem, this can be addressed in a later iteration.

We created two software Fabrications for dealing with these concepts. First, an implementer of the FeatureSetExtractor interface has the responsibility to build a FeatureSet based on the information in a Document. In practice, many of our FeatureSetExtractors ended up dealing with a single feature, but the design allows related features to be computed together. Second, an implementer of the FeatureSetScorer has the responsibility to combine the feature-values in a FeatureSet into a single numeric score.

As a convenience, we defined a FeatureSetCollator, whose responsibility is to combine FeatureSets from various FeatureSetExtractors into a single larger FeatureSet. The FeatureSetCollator itself implements the FeatureSetExtractor interface (i.e. the Composite pattern), so that a client does not need to be aware of the difference between using a tree of FeatureSetExtractors and a single one. The FeatureSetCollator is especially vulnerable to the global uniqueness of feature names; a possible refinement would be to code the class to throw an exception if it encounters a duplicate name.
This fairly flexible design allowed us to split the work across several developers. As we came up with ideas for new features which might be useful, we could implement those as new FeatureSetExtractors without concern for how they would be used in our recommender. As a separate "development thread" we could be working towards integrating the FeatureSetExtractors and FeatureSetScorers into a Recommender. As mentioned earlier, this aspect of the design also contributed to the ability of the team to self-organize: based on their backgrounds, developers could naturally gravitate towards conceiving and implementing (1) clever features, (2) clever numeric functions to combine scores, or (3) integrating trivial implementations of the components into a working system. As long as someone was working on each piece, we could make progress and everyone could stake a claim on parts interesting to them individually.

While implementing a FeatureSetScorer based on the weighted sum (with a default of equal weights) of the features, we came upon a weakness in our design. Features which naturally come from a domain with high numeric values (e.g. document length) were drastically overpowering features which naturally came from a domain with small numeric values (e.g. number of annotations under the current learned model). To address this, we wanted to standardize our feature values. The approach we chose was to re-calculate the feature values as z-scores, i.e. the number of standard deviations each feature value was from the mean of that feature value (i.e. we assume, probably incorrectly, that each feature's value is approximately normally distributed). This approach then required knowledge of the full distribution of each feature value, and our FeatureSetScorer had no mechanism for accepting distributions. We solved this in an ad hoc manner: our final Recommender uses a subclass of FeatureSetCollator which has an additional init method which allows it to compute the distribution statistics on the first call to recommend. Some sort of initialization with a large set of documents should probably be factored into the FeatureSetExtractor in a future iteration of the project.

During implementation of several FeatureSetExtractors we noticed a commonality that many extractors were counting occurrences of various phenomena in the text. Many of these phenomena could be concisely represented as regular expressions. We extracted the counting code into a separate class which used the java.util.regex package to count occurrences of the regular expression. Our initial idea was to implement the regular expression counting code as an abstract class and then force each extractor to extend the abstract class; we decided, however,
that this configuration was too restrictive for future work. Instead, we made the RegexOccurrenceCounter into a separate class which each extractor would instantiate for its own purposes.

The integration of the various components into a unified Recommender for use by the IAL framework was relatively straightforward. We just extended the FeatureSetCollator (with the ad hoc extension mentioned above), gave it all of our FeatureSetExtractor classes, and then used a WeightedSumScorer to combine the scores. Then we returned the N top-scoring documents as our recommendation. As an additional benefit, we were able to easily test (by adding and subtracting extractors from the set used) which features improved performance.

5.1 Patterns (GoF & GRASP)

It quickly became evident that some design patterns would come in handy as we were developing the recommender. As mentioned above, the behavior of the FeatureSetExtractors varied by type - one extractor was counting words, another counted commas, and so on. Since much of the code was similar in the FeatureSetExtractors, we used polymorphism and assigned the behavior of extracting features to an interface. The implementer then implemented this interface to extract a particular feature such as word count.

A pattern related to this use of Polymorphism is the Strategy pattern from Gang of Four. We wanted to design varying but related algorithms and have the ability to change these algorithms quickly and efficiently. The Strategy pattern enabled us to do so by defining the FeatureSetExtractor interface and having the implementers WordCount and others extract the relevant feature. This allowed a flexible design that demonstrated the FeatureExtractors were clearly related, but each FeatureExtractor could be changed quickly as needed.

As mentioned above, we also noticed that we needed to collect the extracted features and compute a score. To do this, we used the Composite pattern. We wanted to treat the features as a group of features instead of individual features. As such, each extracted feature implemented the FeatureExtractor Interface. Then, the FeatureSetCollator also implemented the same interface and enabled us to group together each individual feature score. In this way, the scorer does not know if it is dealing with one or more extracted features.

Finally, the classes we implemented exhibit Protected Variation. Due to the Polymorphism and Strategy patterns, it is quite easy to swap out and/or add a new FeatureSetExtractor without requiring too much of an effect on the overall algorithm (except for an increase in annotation accuracy, hopefully). It would also be quite easy to implement a new recommendation algorithm and support that algorithm with the extracted feature scores.

The overall recommender is also highly cohesive. Each class focuses on a specific task, and no one class is over-bloated with dissimilar or unrelated methods. As we learned from GRASP and Gang of Four, this is generally a good idea when designing software. We adhered to this principle, and it really helped us to: divide up work, implement a modular design, and implement classes that made intuitive sense to us. Hence, the design patterns really helped us reason on and improve our overall design.
6 Experimentation

As discussed above, we experimented with individual features to determine the top scoring features based on f-measure. We then combined the top four features and tested our recommender against the deterministic recommender. We tested our combined features with the sigmoid squashing function and without. We also considered using hand-derived weights for each feature, but our preliminary results were roughly equivalent when we used hand-derived weights and when we used the sigmoid squashing function. As such, we focused on refining our features in terms of the sigmoid squashing function instead of developing our features in terms of weights. We leave determining optimal weights via hand or via an algorithm such as EM for future iterations. As can be seen from the graph below, our recommender steadily outperforms the deterministic recommender in f-measure; both with the sigmoid squashing function and without it. Since we decided to focus on f-measure in this iteration and performance using the sigmoid function outperformed performance without the sigmoid function, we decided to use the sigmoid function with the remaining tests (ENUA and f-measure versus ENUA).

Graph 6.1:

Examining our experimental results in terms of ENUA was a different story. Since we focused on f-measure at the expense of ENUA, our algorithm performed much worse than the deterministic recommender, as can be seen from the graph below. In future iterations, we would focus more on balancing f-measure and ENUA, so that our recommender would recommend documents that maximized score on documents but minimized user effort (in terms of new annotations and correcting existing annotations).
Not surprisingly, since our algorithm focused on maximizing f-measure at the expense of ENUA, we also performed poorly on f-measure versus ENUA. As can be seen from the graph below, our performance on f-measure versus ENUA is steadily below that of the deterministic recommender. In fact, the performance of the recommender without the sigmoid function is better than that of the performance of the recommender with the sigmoid function (at least with a low calculation of ENUA). As such, it may make sense to either combine the sigmoid function with learned weights in later iterations or merely focus on learned weights to both focus on maximizing our performance on f-measure and ENUA.
6.2 Reflection

Considering that we focused on f-measure at the expense of ENUA in these early iterations, our algorithm performed quite well. We steadily outperformed the deterministic algorithm in terms of f-measure, as was our goal for this stage. After talking to the client, we would focus on a more balanced algorithm that would maximize f-measure while minimizing ENUA (if the client required such an algorithm).

7 Team Process and Experience

As mentioned, we used several extreme programming practices. We refactored our code when we realized that many of our feature extractor classes had redundant code for tokenizing words and handling regular expressions. We created a utility class called RegexOccurrenceCounter, which counted the number of times a generic regular expression occurred in a document. We then implemented sub-classes for each feature that implemented the super class but specified a specific regular expression - a word or comma for example, and counted the occurrence of this regular expression per document. This preserved the function of the classes while improving the clarity of our code. It also facilitated the addition of new pattern matching feature extractors.

Several agile principles guided our approach. Without a client with whom we could discuss requirements, working software was our primary measure of progress and success. We set an internal deadline for feature development and then moved on to working a feature set scorer and recommender so that we could begin testing. Once we had this initial framework, it was easy to develop and incorporate new feature extractors. This workflow made it easier for us to divide work among team members.

Because of our limited time and busy schedules, it was necessary to do some work individually, but wherever possible we favored face-to-face meetings and an extreme programming team work environment. We worked in the same room together for approximately 25 hours. Whenever possible, we used a projector to facilitate discussion about code and the work environment.

Working in the same room was more productive for certain tasks, like writing feature extractors. It was helpful because the two more-experienced team members could give on-site assistance to the two lesser experienced developers. The experienced developers quickly answered questions and saved the inexperienced developers countless hours looking for documentation or finding the source of an error message. The shared work space also led to the use of refactoring, as discussed earlier. The face-to-face meetings also helped us maintain a consistent vision about what we could accomplish in the time that we had remaining. For example, we discussed several types of classifiers that might have been beneficial but turned out to be too time-consuming to implement in the current timebox.

The agile principle of allowing self-organizing teams came naturally to us. Team members came to the project with varying expertise an assumed the role that fit their skills best. To keep things moving in our limited timeframe, we let team member choose tasks that interested them to work on before the next meeting. This divide-and-conquer strategy worked well, as many of the tasks could be handled individually. We found that brainstorming in group meetings and then using the group ideas to help complete our individual tasks worked well. Tasks like experimentation and documentation were easy to divide and assign to individuals to work on outside of group meeting times. This helped us meet our deadline and kept everyone interested, as group members got to focus on things that interested them.
8 Future Work

We know the project is finished because we have an absolute deadline for the course. If we were building a real system for a real client, we might have a couple more iterations before we could consider ourselves done.

First and foremost, we have a fuzzy goal from the client: increase F-measure without incurring too much user effort. We have increased F-measure but have also incurred a certain amount of user effort. Whether the quantities observed in our experiments are acceptable or not is a subjective decision, and ultimately lies with the client. We would have to ask them whether we need to work harder on bringing user effort down (or F-measure up). This will also allow us to better estimate how many more iterations we need to do.

Additionally, we focused on using developer time efficiently but did not necessarily optimize the performance of our algorithms. Significant performance gains could be achieved by caching feature values; this could be done simply by implementing a caching Proxy FeatureSetExtractor.

8.1 Google hits to verify model annotations

Another idea we had to improve our recommender algorithm was to use Google hits. We would extract every annotation from the model in a given document, search for each annotation in Google and determine the number of hits. We would then calculate the mean and standard deviation for all the annotations. If an annotation’s number of hits were to be much lower or higher than the mean, (3 times of the standard deviation, for example), we might think that this annotation is special and pay extra attention to it. We thought about using the Google SOAP searching API, but it is limited to 1,000 searches per day. To get around this limit we could simulate an http client visiting Google with an HTTP protocol. This resolves the search limit, but it introduces a speed issue. Using this method leads to the low access speed of only 5 searches every second. This is rather slow and would cause a large reduction in speed. In the future, we would attempt to get more authorization from Google to resolve the speed issue and make this a viable strategy.

8.2 Improved hand-tuning of weights

Currently all the features in our model have equal weights. We want to change it so that we can tune the weights arbitrarily. This change not only can improve the current model, but also can get ready for another subsequent improvement: automated learning of the weights. After implementing this feature, we can only adjust the weights by our external experience. But in future, we will employ some technique to determine the weight, for example: neural networks, a genetic algorithm or active learning.

By employing the genetic algorithm, we can also tackle another problem in our model: how to combine these features. So far, we only add all the features’ scores up. There must be some other more complicated and suitable ways to combine them. It is obviously not easy to find them out, but we can take advantage of genetic algorithms.

For a genetic algorithm, we can also construct different models for different documents according to their characteristics. For example, we could generate special algorithms for
documents with high percentages of nouns, documents about sports, or documents about the GRE.

However, after considering the workload for the server (each generation would submit thousands of times); we just left this as a possible enhancement for the future.

8.3 Diversification

As discussed in the Software Design section, we believe the more diverse the documents we recommend, the greater the improvement we can expect in the model, so another enhancement is to increase the diversification. Diversification involves not only the content between documents, but also the content within a document. We can calculate the similarity between any two documents though comparing some of their statistics. For example, we could calculate and refer to their Longest Common Subsequence (LCS). When analyzing one document’s diversification, we can use some compression method, for example LZW, and calculate the compression rate. Of course, a higher the rate means less diversification. With these figure, we can try to recommend as diverse a set of documents as possible.

9 Deliverables

jar file

AnnotationCapitalRatio.java
CapWordRatio.java
CommaCount.java
CommaToWordCountRatio.java
ConfidenceofAnnotation.java
ContentFunctionRatio.java
FeatureSetCollator.java
FeatureSetExtractor.java
FeatureSetRecommender.java
FeatureSetScorer.java
NanduRecommender.java
NounCountRatio.java
NumberOfAnnotations.java
PrepCountRatio.java
PunctCount.java
RegexOccurrenceCounter.java
WeightedSumScorer.java
WordCount.java

javadocs
Nandú Final Report