OF COURSE! Using Bayesian Inference to Build a More Dynamic Course Search

CS221 Final Project by Michael Dickens and Mihail Eric

December 13, 2013

Introduction

ExploreCourses is a search engine used regularly by the Stanford University community for browsing and finding courses. At the present moment, ExploreCourses can support basic query searches, including somewhat accurate retrieval given a course code and the exact title of a course. Almost any query search that does not fall into one of these two classes of searches will either return unrelated class results or more often no results at all. For this project, we sought to develop an improved course searching program that would be more robust in that it could support more diverse user input queries and would also be more dynamic in that the courses that were returned were more related given a universal metric that we define for assessing relatedness. The metric will be explained in a later section.

To achieve improved robustness, we implemented some basic natural language processing schemes for extracting useful and relevant information from a user input. The information that we were specifically looking for included course titles, course codes, department codes, and instructor names. To find more related courses, we extracted a variety of features that we considered relevant from all the data we could attain about a course and then we created a course-relatedness "graph" that assigns a relatedness score to each pair of classes, given their extracted features. To compute the relatedness score, we utilized a Bayesian inference scheme whereby we calculated the probabilities that two courses are related given that they have a pair of features in common. The Bayesian approach and the features used will be explained in later sections.

Feature Extraction

In order to determine an accurate label for relatedness between two courses, we had to extract a useful collection of features from the data for each course. This required some experimentation in order to find a good balance: too many features and the algorithm runs slowly; too few, and we cannot perform useful inference.

We used the following initial set of features:

- words in the title
- words in the description
- course code name
- course code ones digit
- course code tens digit
- instructors
- minimum units
- maximum units

Later, we took each of the course code features and combined them with each title, instructor, and description feature to create a set of binary features. This roughly quadruples the total number of features in the set. We also tried using word bigrams in the title and description, but this did not add substantial benefit.

A Bayesian Approach to Course-Relatedness

To determine course relatedness, we began simply by taking the total number of matching features between two courses. This approach proved too coarse: it matched courses with many common features that weren't actually all that related to each other.

A sudden insight came when we realized we could do much better by taking a Bayesian appraoch. Instead of simply counting the number of features in common, interpret each

feature in common as Bayesian evidence that the two courses are related and perform a Bayesian probability update. Similarly, if a feature does not occur in common between two courses, consider this evidence that they are not related.

Then, instead of considering each feature as equally strong evidence, weigh a feature against the prior probability of that feature occurring. We figure out the prior probability of a feature by counting how frequently it occurs in the database.

Thus, to find the probability that two courses are related, we update a prior probability estimate with the evidence given by each feature found in the two courses.

We had some difficulty in combining probabilities. We attempted to use the naive Bayes assumption to compute the combined probability, but this gave us unworkably-small probabilities. We updated the model to combine each of the feature probabiliteis by simply taking their sum. Although this operation is not meaningful in a probabilistic sense, it works as a useful heuristic for producing workable results.

Query Parsing

In order to satisfy a user's input queries, we need to extract useful information from a given query. Using the Python Natural Language Processing Toolkit, we employed the following natural language processing scheme: tokenize query, tag with parts of speech, chunk appropriately using regular expression grammars. Once a query is chunked, useful information can be derived through analysis of the corresponding parse tree. Using this scheme, we are able to support user query searches consisting of more complex phrases including ones of the form 'courses taught by Kannan Soundararajan.'

As an example of how our program processes a query, consider the example from above: 'courses taught by Kannan Soundararajan.' We first tokenize the query and assign part-of-speech tags to each token. Note that we used the built-in Python NLTK POS-tagger which is trained on the Penn Treebank corpus. This gives us the following output:

[('courses', 'NNS'), ('taught', 'VBD'), ('by', 'IN'), ('Kannan', 'NNP'), ('Soundarajan', 'NNP')]

where the second entry in each tuple represents the corresponding part-of-speech that the token has been identified as. Now, using a regular expression grammar of the form "PNOUN:{ < NNP>*}", we can chunk the text to get the following parse tree in flattened text form:

Tree('S', [('courses', 'NNS'), ('taught', 'VBD'), ('by', 'IN'), Tree('PNOUN', [('Kannan', 'NNP'), ('Soundarajan', 'NNP')])])

Here the grammar searches for strings of proper nouns, operating under the assumption that most strings of proper nouns in a user input will be associated with the name of an instructor. Thus, given this parse tree, we can deduce that the user wants to search for courses taught by 'Kannan Soundararajan.'

We support course code and department code searches by tokenizing a query into unigram and bigram tokens and linearly searching for matches against a set of course/department codes. This scheme allows us to find multiple course codes in a user query. We also support title searches by searching for the input string in a set of all available course titles. In order to satisfy more complex query searches, we have a few parsing grammars in place for extracting information. A few of them include:

1. {<NNP>*}. For identifying strings of proper nouns.
2. {<VBD> <IN> <NNP>*}. For identifying phrases of the form: verb, proposition, proper noun.

As more grammars are added, the complexity of searches supported will increase.

Data

We used the ExploreCourses Java API to acquire information related to the 11,613 courses listed for enrollment for the 20132014 academic year. We populated a SQL datebase, using

the sqlite3 Python library, with the following information for each course: course title, course code, instructors teaching the course, minimum units of credit received for taking the class, maximum units units of credit received for taking the class, and course description.

However, for the purposes of the assessment, we used a database of a reduced subset of approximately 170 random classes taken from the CS and MATH departments. We had to utilize a reduced database because it was too computationally time-consuming to create a comprehensive relatedness graph for all 11,613 courses.

We use the query parser to find an exact match for the query. Then we use the course relatedness graph to find a set of the nearest courses. A query returns this set of courses in order of relatedness.

For example, if the user inputs 'CS221', we identify the course with the code 'CS221'. Then we find the courses most similar to CS221 and return those.

This approach has certain limitations: for example, a course that has CS221 as a pre-requisite might be more relevant to a search for 'CS221' but not be considered sufficiently closely related to CS221. In practice, this isn't much of a problem. Our current approach seems to work relatively well, but we may refine it in the future.

Metric for Assessment

We developed a simple point-based metric to objectively determine the quality of our course searcher as compared to ExploreCourses. To evaluate the performance of our searcher, we generated a random subset of 21 courses taken from our reduced database. We subdivided these 21 courses into five types of searches:

1. Specific course code such as CS109
2. Instructor name such as 'Mehran Sahami'
3. Course title or subset of course title such as 'Probability for Computer Scientists'
4. More complex queries

The more complex queries included two of each of the following:

1. 'courses taught by [instructor]'
2. '[course 1] and [course 2]'
3. 'courses in the [department] department'

We used the following point system to evaluate the accuracy of queries:

For each of the following, give the full points for the first hit, 3/5 of the points for the second hit, and 2/5 the points for the third hit. We only consider the top 3 hits.

- 100 points if exact match OR
- 10 points if same sequence as searched-for course (e.g. Math 51, 52, 53)
- 5 points if in the same department
- 10 points if has same instructor

Searches for instructors are judged simply by whether the result is taught by the instructor. An exact match is worth 100 points.

These numbers are somewhat arbitrary. To get a better idea of the reliability of our search algorithm, we should collect data on user satisfaction with real-world queries. Unfortunately, such a metric is not feasible at this time, so for now we will stick with the point-based system described above.

Results

Further Work

In the future, we can make two classes of improvements to our program: augmentations to the artificial intelligence schemes we employ to make our program 'smarter' or additions to improve the general robustness of the program's features.

First off, we will discuss some of the improvements to the artificial intelligence infrastructure we can implement. At the moment, our query parsing uses the built-in NLTK POS-tagger which has made specific mistakes on some queries in the past such as occasionally labeling 'CS109' as a proper noun. We may want to generate our own corpus of expected user queries along with parts-of-speech tagged, so that we can more appropriately train the POS-tagger that will be used for our program. We will also want to support a more diverse collection of chunking grammars that will allow for a greater complexity of query searches. Another thing we've also brainstormed is the possibility of supporting searches based on a user's previous history. Here we could employ some standard supervised machine learning classifying algorithms. We could use a similar Bayesian inference scheme whereby we find the probability that a user will want to find a certain course given that she has searched a number of other classes in the past.

As far as improving the overall robustness of the program, the code is definitely not implemented the most efficiently as the moment. We often do complete linear searches through huge databases to find matches which is something we can definitely improve. The computational slowness of the code is also the reason we had to develop a reduced databased for testing course-relatedness. In the future we would like to create a relatedness graph for the entire collection of 11,000 courses offered. We can also implement some user spell-correction algorithm, for fixing a query if it doesn't directly match a name or code available in a database. Something as basic as a memoized edit distance algorithm should work very well for our purposes. Given all these places for improvement, our searcher already outperforms "Explore Courses" so we would like to get our code live at some point in the future.

References

Bird, Steven, et al. (2009). Natural Language Toolkit. Retrieved from http://nltk.org/

Gabrilovich, E., & Markovitch, S. (2009). Wikipedia-based semantic interpretation for natural language processing. Journal of Artificial Intelligence Research, 34(2), 443.

Graham, Paul, "Probability." Retrieved from http://www.paulgraham.com/naivebayes.html.

*****TODO: FIX INDENTATION*****