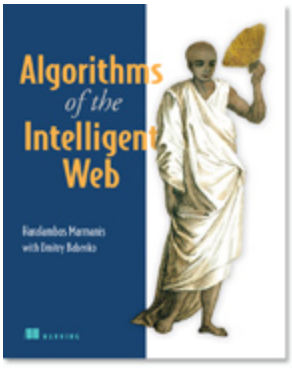
Improving Your Results Based on User Clicks

Excerpted from



[Algorithms of the Intelligent Web](#)
EARLY ACCESS EDITION

Author: Haralambos Marmanis with Dmitry Babenko
MEAP Release: April 2008
Softbound pint: February 2009 (est.); 325 pages
ISBN: 1933988665

This article is taken from the book Algorithms of the Intelligent Web. *This segment presents a probabilistic technique for conducting user click analysis.*

Link analysis allows us to take advantage of the structural aspects of the Internet. In this section, we will talk about a different way of leveraging the nature of the Internet, namely, user clicks. As you know, every time a user executes a query, he will either click on one of the results or he will click on the link that shows the next page of results, if applicable. In the first case, the user has identified something of interest and clicks on the link either because that is what he was looking for or because the result is interesting and he would like to explore further the related information, in order to decide if it is, indeed, what he was looking for. In the second case, the best results were not what the user wanted to see and he wants to have a look at the next page just in case the search engine is worth a dime!

Kidding aside, one of the reasons that evaluating relevance is a difficult task is that relevance is subjective. If you and I are looking results for the query “elections”, you may be interested in the US elections while I may be interested in the UK elections, or even in my own town’s elections. It is impossible for a search engine to know the intention, or the context, of your search without further information. So, the most relevant results for one person can be, and quite often are, different from the most relevant results for another person, even though the query terms may be identical!

User clicks allow us to take as input the interaction of each user with the search engine. Aristotle said: “We are what we repeatedly do” and that is the premise of user clicks analysis, i.e. your interaction with the search engine defines your own areas of interest and your own subjectivity. This is the first time that we describe an intelligent technique responsible for the *personalization* of a web application. Of course, a necessary condition for that is that the search engine can identify which queries come from a particular user. In other words, the user must be logged in your application or must have otherwise established a session with the application. It should be clear that our approach for user clicks analysis is applicable to every application that can record the user’s clicks, and it is not specific to search applications.

Now, let's assume that you have collected the clicks of the users as indicated in the file "user-clicks.csv", which you can find in the `data/ch02` directory together with the rest of the files that we have been using in this chapter. Our goal is to write an intelligent code that can help us leverage that information, much like the PageRank algorithm helped us to leverage the information about our network. That is, we want to use these data to *personalize* the results of the search, by appropriately modifying the ranking, depending on who submits the query. The comma separated file contains values in three fields:

1. A string that identifies the user
2. A string that represents the search query
3. A string that contains the URL that the user has selected in the past, after reviewing the results for that query

If you do not know the user (no login/no session of any kind), then you can use some default value such as "anonymous" – of course, you should ensure that "anonymous" is not actually a valid username in your application! If your data have some other format, it's OK. You should not have any problems adopting our code for your specific data. In order to personalize our results, we need to know: (1) the user, (2) their question, and (3) their past selections of links for that question. If you have that information available then you should be ready to get in action!

You may notice that, in our data, for the same user and the same query there are more than one entry. That is normal and you should notice it in your data as well. It is exactly the number of times that a click appears in that file that makes its URL a better candidate for our search results. Typically, the same user will click a number of different links for the same query because his interest at the time may be different or because he may be looking for additional information on a topic. Therefore, an interesting attribute that you should consider is a timestamp. Time related information can help you identify temporal structure in your data. Some user clicks follow periodic patterns; some are event driven; while some others are completely random. A timestamp can help you identify the patterns or the correlations with other events.

Before we dive into the details, let us see how we can obtain personalized results for our queries. Listing 2.11 shows our script which is similar to listing 2.9 but this time we load the information about the user clicks and we run the same query "google ads" twice, once for user "dmitry" and once for user "babis".

Listing 2.11 Accounting for user clicks in the search results

```
FetchAndProcessCrawler c = new FetchAndProcessCrawler("C:/iWeb2/data/ch02",5,200);

c.setUrls("biz");                                              Set up the URLs
c.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-01.html");
c.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-02.html");
c.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-03.html");
c.run();

LuceneIndexer lidx = new LuceneIndexer(c.getRootDir());        Lucene Indexing
lidx.run();
MySearcher oracle = new MySearcher(lidx.getLuceneDir());
```

```
PageRank pr = new PageRank(c.getCrawlData());                PageRank creation
pr.setAlpha(0.9);
pr.setEpsilon(0.00000001);
pr.build();

UserClick aux = new UserClick();
UserClick[] clicks = aux.load("C:/iWeb2/data/ch02/user-clicks.csv");   Loading user clicks

TrainingSet tSet = new TrainingSet(clicks);                  Create training set

NaiveBayes nb = new NaiveBayes("nb",tSet);                   Classifier definition

nb.trainOnAttribute("a-0");                                  Attribute selection
nb.trainOnAttribute("a-1");
nb.trainOnAttribute("a-2");

nb.train();                                                  Classifier training

oracle.setUserLearner(nb);                                   Assign the classifier to the searcher

UserQuery dQ = new UserQuery("dmitry","google ads");
oracle.search(dQ,5,pr);

UserQuery bQ = new UserQuery("babis","google ads");
oracle.search(bQ,5,pr);
```

You've seen the first part of this script in listing 2.9. First, we load the pages that we want to search. After that, we index them with Lucene and build the PageRank that corresponds to their structure. The part that involves new code comes with the class `UserClick`, which represents the click of a specific user on a particular URL. We also defined the class `TrainingSet`, which holds all the user clicks. Of course, you may wonder, what is wrong with the array of `UserClick`s? Why can't we just use these objects? The answer lies on the following fact. In order to determine the links that are more likely to be desirable for a particular user and query, we are going to load the user clicks onto a *classifier;* in particular, the `NaiveBayes` classifier.

2.4.1 Using the Naïve Bayes classifier

We will address classification extensively in chapters 5 and 10, however, we will describe some fundamentals here for clarity. The `NaiveBayes` class is a classifier and classifiers are agnostic to `UserClick`s; they are only concerned with `Concept`s, `Instance`s, and `Attribute`s. You can think of `Concept`s, `Instance`s, and `Attribute`s as the analogues of directories, files, and file attributes on your file system.

A classifier's job is to assign a `Concept` to an `Instance`, that's all a classifier does. In order to know what `Concept` should be assigned to a particular `Instance`, a classifier reads a `TrainingSet`, i.e. a set of `Instance`s that already have a `Concept` assigned to them. Upon loading those `Instance`s, the classifier *trains* itself or, in other words, *learns* how to map a `Concept` to an `Instance` based on the assignments in the `TrainingSet`. The way that each classifier trains depends on the classifier.

Our intention is to use the NaiveBayes classifier as a means of obtaining a relevance score for a particular URL based on the user and the submitted query. The good thing with the NaiveBayes classifier is that it provides us with something that is called the *conditional probability* of X given Y. That is, a probability that tells us how likely is it to observe event X provided that we already observed event Y. Thus, the NaiveBayes classifier can provide us with a measure of how likely it is that a user A wants to see URL X provided that he submitted query Q; in our case, Y=A+Q. In other words, we will not use the NaiveBayes classifier to classify anything. We will only use its capacity to produce a measure of relevance which exactly fits our purposes. Listing 2.12 shows the code from the class NaiveBayes.

Listing 2.12 The NaiveBayes classifier

```
/**
 * A basic implementation of the Naive Bayes algorithm.
 *
 * The emphasis is on teaching the algorithm, not optimizing its performance.
 *
 */
public class NaiveBayes implements Classifier {

        /**
         * You can use the NaiveBayes classifier in many occasions
         * So, let's give it a name to identify the instance of the Classifier.
         */
        private String name;

        /**
         * Every classifier needs a training set. Notice that both the name
         * of the classifier and its training set are intentionally set during
         * the Construction phase.
         *
         * Once you created an instance of the NaiveBayes classifier you cannot
         * set its TrainingSet but you can always get the reference to it and
         * add instances.
         */
        private TrainingSet tSet;

        /**
         * These are the probabilities for each concept
         */
        private HashMap<Concept,Double> conceptPriors;

        /**
         * This structure contains the fundamental calculation elements of
         * the Naive Bayes method, i.e. the conditional probabilities.
         */
        private HashMap<Concept,Hashtable<Attribute,HashSet<AttributeValue>>> p;

        /**
         * These are the attribute indices that we should consider for training
         */
        private ArrayList<String> attributeList;

        /**
         * The only constructor for this classifier takes a name and
         * a training set as arguments.
         *
         * @param name the name of the classifier
```

```
         * @param set the training set for this classifier
         */
        public NaiveBayes(String name, TrainingSet set) {

                this.name = name;
                tSet = set;

                conceptPriors = new HashMap<Concept,Double>(tSet.getNumberOfConcepts());
        }

        public Concept classify(Instance instance) {
                return null;
        }

        /**
         * Training simply sets the probability for each concept
         *
         */
        public boolean train() {

                boolean hasTrained = false;

                if ( attributeList == null || attributeList.size() == 0) {
                //ERROR
                System.out.print("Can't train the classifier without specifying the attributes for
training!");
                System.out.print("Use the method --> trainOnAttribute(Attribute a)");

                } else {

                        calculateConceptPriors();

                        calculateConditionalProbabilities();
                }

                return hasTrained;
        }

        public void trainOnAttribute(String aName) {

                if (attributeList ==null) {
                        attributeList = new ArrayList<String>();
                }

                attributeList.add(aName);
        }
        /**
         * Strictly speaking these are not the prior probabilities
         * but just the counts. However, we want to reuse these counts
         * and the priors can be obtained by a simple division.
         */
        private void calculateConceptPriors() {

                for (Concept c : tSet.getConceptSet()) {

                 //Calculate the priors for the concepts
                 int totalConceptCount=0;

                        for (Instance i : tSet.getInstances().values()) {
```

```
                        if (i.getConcept().equals(c)) {
                                totalConceptCount++;
                        }
                }

                conceptPriors.put(c, new Double(totalConceptCount));
        }
    }

private void calculateConditionalProbabilities() {

  p = new HashMap<Concept,Hashtable<Attribute,HashSet<AttributeValue>>>();

  for (Instance i : tSet.getInstances().values()) {

    for (Attribute a: i.getAtrributes()) {

        if (attributeList.contains(a.getName())) {

          if ( p.get(i.getConcept())== null ) {

            p.put(i.getConcept(),
                  new Hashtable<Attribute,HashSet<AttributeValue>>());

          }

          Hashtable<Attribute,HashSet<AttributeValue>> aMap = p.get(i.getConcept());

          if ( aMap.get(a) == null ) {

            HashSet<AttributeValue> avSet = new HashSet<AttributeValue>();
            avSet.add(new AttributeValue(a.getValue()));
             aMap.put(a, avSet);

          } else {

            for (AttributeValue aV : aMap.get(a)) {
                if ( aV.getValue().equals(a.getValue()) ) {
                  aV.count();
                  break;
                }
             }
           }
          }
        }
     }
  }



        public double getProbability(Concept c, Instance i) {

                double cP=0;

                if (tSet.getConceptSet().contains(c)) {

                        cP = (getProbability(i,c)*getProbability(c))/getProbability(i);

                } else {
```

```
                        // We have never seen this concept before
                        // assign to it a reasonable value
                        cP = 1/(tSet.getNumberOfConcepts()+1);
                }

                return cP;
        }

        public double getProbability(Instance i) {

                double cP=0;

                int j=0;

                for (Concept c : getTset().getConceptSet()) {

                        cP += getProbability(i,c)*getProbability(c);

                        j++;
                }

                return (cP == 0) ? (double)1/tSet.getSize() : cP;
        }

        public double getProbability(Concept c) {

                return conceptPriors.get(c)/tSet.getSize();
        }

public double getProbability(Instance i, Concept c) {

        double cP=1;

        for (Attribute a : i.getAtrributes()) {

                if ( attributeList.contains(a.getName()) ) {

                        Hashtable<Attribute,HashSet<AttributeValue>> aMap = p.get(c);

                        if ( aMap.get(a) == null) {
                         // one of the attribute values is not present for current concept.
                            cP *= 0.0 / conceptPriors.get(c);
                        } else {

                                for (AttributeValue aV : aMap.get(a)) {

                                        if (aV.getValue().equals(a.getValue())) {

                                        cP *= (double)(aV.getCount()/conceptPriors.get(c));

                                        break;
                                }
                         }
                }
        }
  }

  return (cP == 1) ? (double)1/tSet.getNumberOfConcepts() : cP;
}
```

```
  /**
   * @return the name
   */
  public String getName() {
    return name;
  }

  /**
   * @return the tSet
   */
  public TrainingSet getTset() {
    return tSet;
  }
}
```

If you recall the script in listing 2.11, we have created a training set, we have created an instance of the classifier with that training set, and before we assign the classifier to the `MySearcher` instance, we do the following two things:

- We tell the classifier what attributes should be taken into account for training purposes
- We tell the classifier to go ahead and train itself on the set of user clicks that we just loaded and for the attributes that we specified

The attribute with label "a-0" corresponds to the user. The attributes "a-1" and "a-2" correspond to the first two terms of the query, which we obtained by using Lucene's `StandardAnalyzer` class. As you can see, in listing 2.12, during training we are assigning probabilities based on the frequency of occurrence for each instance. The important method in our context is `getProbability(Concept c, Instance i)`, which we will use to obtain the relevance of a particular URL (`Concept`) when a specific user executes a specific query (`Instance`).

2.4.2 Combining Lucene indexing, PageRank, and user clicks

Armed with the probability of a user preferring a particular URL for a given query, we can proceed and combine all three techniques to obtain our enhanced search results. The relevant code is shown in listing 2.13.

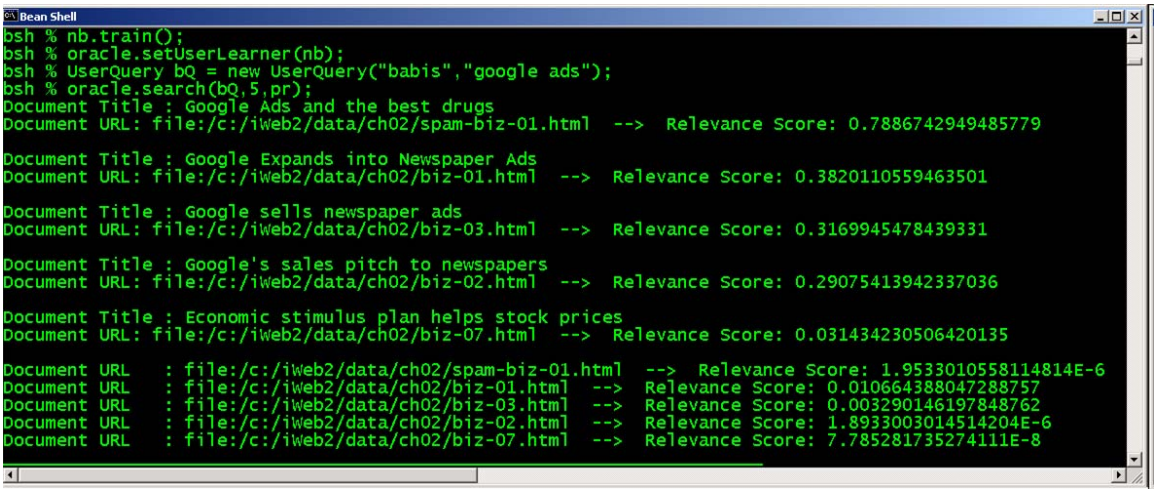
Listing 2.13 Putting them all together: Lucene indexing, PageRank values, and user click probabilities

```
/**
 * A method that combines the score of an index based search
 * and the score of the PageRank algorithm to achieve better
 * relevance results, while personalizing the result set based on
 * past user clicks on the same or similar queries.
 *
 * @param userID identifies the person who issues the query
 * @param query is the whole query
 * @param numberOfMatches defines the maximim number of desired matches
 * @param pR the PageRank vector
 * @return the result set
 */
public SearchResult[] search(UserQuery uQuery, int numberOfMatches, Rank pR) {

  SearchResult[] docResults = search(uQuery.getQuery(), numberOfMatches);
  String url;

  StringBuilder strB = new StringBuilder();
```

```
  int docN = docResults.length;

  if (docN > 0) {

    int loop = (docN<numberOfMatches) ? docN : numberOfMatches;

        for (int i = 0; i < loop; i++) {

          url = docResults[i].getUrl();

          UserClick uClick = new UserClick(uQuery,url);

/**
 * TODO: Book Section 2.4 -- Weighing the scores to meet your needs
 *
 */
          double indexScore = docResults[i].getScore();

          double pageRankScore  = pR.getPageRank(url);

          BaseConcept bC = new BaseConcept(url);

          double userClickScore = learner.getProbability(bC, uClick);

          // Create the final score
          double hScore;

          if (userClickScore == 0) {

            hScore = indexScore * pageRankScore * EPSILON;

          } else {

             hScore = indexScore * pageRankScore * userClickScore;
           }

          // Update the score of the results
           docResults[i].setScore(hScore);

        strB.append("Document URL   : ").append(docResults[i].getUrl()).append("  -->  ");
        strB.append("Relevance Score: ").append(docResults[i].getScore()).append("\n");
    }
  }

  strB.append(PRETTY_LINE);
  System.out.println(strB.toString());

  return docResults;

}
```

Figure 2.8 shows the results for user “dmitry”. As you can see, due to the fact that “dmitry” clicked several times on the page “biz-03.html” in the past, the relevance score for that page is the highest. The second best hit is page “biz-01.html”, which is also in the user clicks file. The spam page appears third but that is a side effect of the small number of pages; we intentionally did not include our scaling `m` factor to demonstrate its impact on the results.

Figure 2.8 Combining Lucene, PageRank, and user clicks together for personalized, high-relevance, search results



Similarly, in figure 2.9, we execute the same query, i.e. “google ads”, but this time we do it as user “babis”. We have reversed the clicks of the user “dmitry” to create the clicks for the user “babis”. Indeed, the results show that the first hit is page “biz-01.html”, whereas page “biz-03.html” is second. Everything else is the same. The only difference in the result set comes from the fact that the query was executed by different users and that difference reflects exactly what the application *learned* from the file “user-clicks.csv”.

Figure 2.9 Combining Lucene, PageRank, and user clicks together for personalized, high-relevance, search results