# Estimating the Selectivity of *tf-idf* based Cosine Similarity Predicates

Sandeep Tata Jignesh M. Patel
Department of Electrical Engineering and Computer Science
University of Michigan
2260 Hayward Street, Ann Arbor, Michigan 48109
{tatas,jignesh}@eecs.umich.edu

## Abstract

An increasing number of database applications today require sophisticated approximate string matching capabilities. Examples of such application areas include data integration and data cleaning. Cosine similarity has proven to be a robust metric for scoring the similarity between two strings, and it is increasingly being used in complex queries. An immediate challenge faced by current database optimizers is to find accurate and efficient methods for estimating the selectivity of cosine similarity predicates. To the best of our knowledge, there are no known methods for this problem. In this paper, we present the first approach for estimating the selectivity of tf.idf based cosine similarity predicates. We evaluate our approach on three different real datasets and show that our method often produces estimates that are within 40% of the actual selectivity.

## 1 Introduction

A growing number of database applications require approximate string matching predicates on text attributes. For example, in data scrubbing [4] and data integration applications [5, 6], these predicates are valuable in dealing with spelling errors, typographical errors, and problems with non-uniform data representation. Address fields for instance can refer to the same location, but be written using different conventions ("1301 Beal Ave., Ann Arbor" vs. "1301 Beal Avenue, Ann Arbor"). Another example is the case of item descriptions which vary slightly from vendor to vendor. One might want to search on the description field to find similar items.

For many real world application, the authors in [3, 8] show that the cosine similarity metric can robustly handle spelling errors, rearrangement of words, and other differences in strings. They also demonstrate that cosine similarity searches and joins can be implemented completely in SQL without adding any code to the relational engine. While cosine similarity is a good metric for comparing strings, to the best of our knowledge, there are no known methods for estimating the selectivity of these predicates. As a result, optimizers may often produce inefficient plans for queries involving these predicates. With the increasing use of cosine similarity predicates, there is an urgent need to develop methods that can estimate the selectivity of these predicates.

In this paper, we discuss a technique for estimating the selectivity of tf.idf based cosine similarity predicates. We make use of a statistical summary of the distribution of different tokens in the database. We also make use of the distribution of the dot product of a typical query with a database row's tf.idf vector. We present two techniques that use the data in different ways and compare their performance on different datasets.

The rest of the paper is organized as follows: Section 2 describes related work and briefly reviews cosine similarity. Section 3 describes the summary structure we employ. Section 4 describes the algorithm used to compute the estimates. The experimental evaluation is presented in Section 5. Finally, we make concluding remarks and point to directions of future work in Section 6.

## 2 Review and Related Work

Cosine similarity is a vector-based measure of the similarity of two strings. The basic idea behind cosine similarity is to transform each string into a vector in some high dimensional space such that similar strings are close to each other. The cosine of the angle between two vectors is a measure of how "similar" they are, which in turn, is a measure of the similarity of these strings. If the vectors are of unit length, the cosine of the angle between them is simply the dot product of the vectors.

There are many ways of transforming a string in the database into a vector. The *tf.idf* vector is a popular choice for this representation. The *tf.idf* vector is composed of the product of a *term frequency* and the *inverse document frequency* for each token that appears in the string. The process of constructing the *tf.idf* vector is described below.

As a first step towards implementing the cosine similarity predicate, we construct a *tf.idf* vector for each row in the relation. If there are multiple string attributes of interest in each row, then we need to compute a vector for each string attribute. To keep the discussion simple, we will assume there is only one string attribute in the relation that is used in a cosine similarity operation.

The length of the *tf.idf* vector is equal to the total number of tokens. A token can be a q-gram or a word. If we are using q-grams, then the length of each vector is the total number of possible q-grams = $|A|^q$, where $|A|$ is the size of the alphabet. The vector stores the *tf.idf* value corresponding to each token for each string. The term *frequency* is the number of times the token appears in the string and is a measure of the importance of that token in the string. The inverse document frequency (inverse of the number of strings in which the token appears) serves to normalize the effect of tokens (like *"the"*) that appear commonly in many strings. The product of *tf* and *idf* is a measure of the importance of the token in the string and the database as a whole. Note that in most real datasets, the strings are very short when compared to the total number of possible tokens, and therefore these vectors tend to be very sparse.

When a query comes in, the normalized *tf.idf* vector corresponding to the query is constructed. The *idf* of each term in the query is just 1. We compute the dot product of this vector with the vector for each row in the database: this is the cosine similarity. If the query and the string share more terms, the dot product is higher. In addition, if they share more "uncommon" terms, that contributes to the score more. The predicate is typically of the form *cosine_similarity (R.s,"Dr. Jekyll")* $> 0.5$, and is evaluated by selecting all those strings where the dot product exceeds the given threshold [3, 8].

To the best of our knowledge, there is no literature on techniques to estimate the selectivity of a cosine similarity predicate. The work closest to ours is [7] where the authors describe a selectivity estimation technique for a fuzzy string predicate. However, this fuzzy predicate is different from any of the well known predicates and has not been shown to perform like cosine similarity in real world tasks [3, 8].

In this paper, we focus on tf.idf based cosine similarity. Although there are other vector representations where cosine similarity can be used, tf.idf is a popular choice in many applications because of its simplicity and robustness. The techniques in this paper take advantage of some of the properties of tf.idf, and therefore will likely require adaptation to work with other vector representations.

Interestingly, the authors of [1] show that many metric distance measures follow a power law distribution for average number of neighbors with respect to distance. That is, number of neighbors within distance $s$ is proportional to $s^d$ where $d$ is some positive constant. As has been argued in [7], this property does not hold for similarity functions like the edit distance, and in our case, the cosine similarity function because of the large number of pairs of words within the same distance. Furthermore, this approach only estimates the average number of neighbors for a string in a dataset, and does not estimate the number of neighbors for a given query string which could be very different from the average.

## 3 Summary Structure

The summary structure we describe stores a concise representation of the distribution of the *tf.idf* values for each token. If we think of the *tf.idf* vectors for all tuples in the relation as a matrix, we observe that this matrix is very sparse. Table 1 shows a sketch of such a matrix. Most rows in this table are sparse because a given string

| rowID | string | Tok 1 | ..... | Tok N |
|---|---|---|---|---|
| 1 | $s_1$ | $w_1^1$ | ... | $w_N^1$ |
| . | . | . | . | . |
| . | . | . | . | . |
| R | $s_R$ | $w_1^R$ | ... | $w_N^R$ |
| | | $\mu_1, \sigma_1, C_1$ | ... | $\mu_N, \sigma_N, C_N$ |

Table 1: A Table and the tf.idf Vectors

is likely to contain only a small number of tokens. In addition, most columns in this table are also sparse, because very few tokens (like "the") are likely to appear in a large number of strings. We have observed empirically that the probability density function of the *tf.idf* weights for a column is characterized by a large mass of probability at zero (most tokens appear only in a few strings). The rest of the probability is distributed around a small positive value. The proposed summary structure (as shown in the last row of Table 1) captures this distribution by storing the following three values for each token:

1. Mean of X for X $\neq$ 0 ($\mu_i$),
2. Standard Deviation for X $\neq$ 0 ($\sigma_i$), and
3. Probability that a q-gram is non zero, 1-Prob(X=0) ($C_i$)

Assume that all the nonzero *tf.idf* values are stored in a table called *Vectors(token,row,value)*. That is, for each token in the original database, the table *Vectors* stores a record for each row in which this token appears with the *tf.idf* value for that token in that row. This is merely a compact way of storing the (sparse) *tf.idf* vector for each row of the database. The summary structure can be generated from *Vectors* by simply using the following SQL query:

```
SELECT token, avg(value),
stddev(value),count(row)/total_rows
FROM Vectors GROUP BY token;
```

Note that the size of this summary structure is bounded by the number of distinct tokens in the language from which the text is drawn. For example, [3, 8] show that for many real applications cosine similarity works well with a token size of three. Assuming an alphabet of size 50 (characters, numbers, punctuation, etc.) the maximum number of tokens (q-grams) is 125K. Also note that the size of the structure is largely independent of the database size, and for a large text database the summary structure is a very small proportion of the total size.

# 4 The Estimation Algorithm

The cosine similarity is the dot product of two *tf.idf* vectors representing the query and the database string. The key to estimating the selectivity of a cosine similarity predicate is to understand the distribution of the dot product. In other words, the problem at hand is to compute the cumulative distribution function of the dot product given a) the query vector, and b) a distribution characterizing the *tf.idf* vectors in the database. Once we compute this cumulative probability distribution function, calculating the probability that the cosine similarity exceeds a certain threshold becomes fairly simple.

We model the *tf.idf* vector in the database as a vector of random variables $(X_1 X_2 X_3 ... X_n)$ – one for each token. The dot product can now be modeled as:

$$Y = \Sigma_{i=i}^{n} u_i \times X_i \quad (1)$$

where $u$ is the query vector.

A straightforward approach to understanding the distribution of $Y$ is to model the distribution of each of the $X_i's$ and analytically compute the PDF of $Y$. However, this turns out to be extremely difficult for any non-trivial characterization of $X_i$. Alternately if we were to evaluate the PDF of $Y$ by sampling the PDF's of $X_i's$, it turns out that the number of samples required to accurately estimate the selectivity is prohibitively high. We therefore choose an alternate technique where we model the distribution of $Y$ and try to determine the parameters of the distribution.

In order to understand how the dot product is distributed, we generated a large set of sample queries by randomly picking strings in the database and introducing one or two errors in the string. We repeated this experiment for a variety of datasets. We observed that the distribution is as shown in Figure 1. The distribution is characterized by a mass of probability close to zero. The rest of the probability is distributed such that it peaks at a small positive value and a long tail tapering off to 0 at $Y$ = 1. After evaluating several well known distributions, we determined that this data was modeled accurately as an inverse normal distribution [9] with a mass of proba-

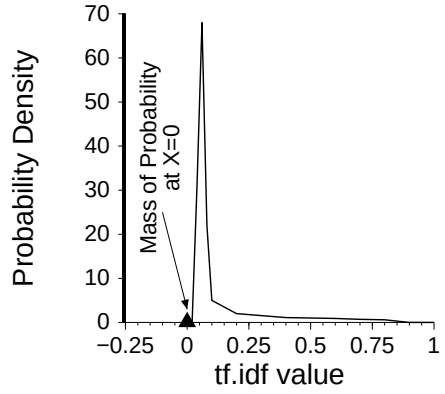


Figure 1: Typical Distribution of the tf.idf Dot Product

bility at 0. We show in the next section, that this simple empirical observation leads to surprisingly good results.

The inverse normal distribution can be completely characterized by its mean and standard deviation. We remind the reader that the probability distribution function of the inverse normal function is:

$$PDF = \sqrt{\frac{B}{2\pi y^3}} exp(-\frac{B}{2y}(\frac{y-A}{A})^2) \qquad (2)$$

where the mean is $A$, and the variance is $\frac{A^3}{B}$. The cumulative distribution function (CDF) is:

$$CDF = \Phi(\sqrt{\frac{B}{y}}\frac{y-A}{A}) + exp(\frac{2B}{A})\Phi(\sqrt{\frac{B}{y}}\frac{-y-A}{A}) \qquad (3)$$

where $\Phi(x)$ is the CDF of a standard Gaussian.

The problem now reduces to determining the parameters of the inverse normal distribution (mean and variance). We now present two empirical algorithms for estimating the mean and standard deviation of $Y$ using the data at hand. We then show through experiments in Section 5 that these techniques lead to good estimates.

## 4.1 Algorithm ES

A simple approach to estimate the mean of $Y$ is to use the weighted average of the means of $X_i's$ (in Equation 1) from each column of the *tf.idf* matrix. The mean of each $X_i$ is available in the summary structure. We compute

$$\mu^{ES} = \alpha \times \sum(C_i \times \mu_i \times u_i) \qquad (4)$$

where $C_i$ is the probability that token $i$ assumes a nonzero value. $\mu_i$ is the mean of the nonzero values of token $i$ as stored in the summary, and $u_i$ is the *tf.idf* weight of the token $i$ in the query vector.

The standard deviation is also computed similarly:

$$\sigma^{ES} = \beta \times \sum(C_i \times \mu_i \times u_i) \qquad (5)$$

We call this simple approach ES.

In the above equations, $\alpha$ and $\beta$ are empirically determined scaling constants for a given relation. They are present to accommodate for the fact that the weighted sum of means does not necessarily yield the actual mean of $Y$. In order to determine $\alpha$, we first assume $\alpha = 1$ . We determine the average value of the ratio $\frac{\mu^{actual}}{\mu^{ES}}$ for a training set of queries and set $\alpha$ to this value. $\beta$ is determined similarly. Using samples from a real workload for the training set will ensure that these values are more accurate.

**Algorithm ES(query,threshold,summary)**
1. Construct the *tf.idf* vector $u$ for the query.
2. Compute $\mu_{ES} = \alpha\Sigma_{i=1}^{N}(\mu_i \times C_i \times u_i)$
3. Compute $\sigma_{ES} = \beta\Sigma_{i=1}^{N}(\sigma_i \times C_i \times u_i)$
4. Compute over nonzero $u_i$ :
5. $nz_{ES} = 1 - (\Pi_{i=1}^{N}(1 - C_i))^{1/q}$
6. Compute Estimate =
$nz_{ES} \times inv\_normal\_cdf(threshold, \mu_{ES}, \sigma_{ES})$

Figure 2: Estimation using ES

In order to completely characterize $Y$, we also need to estimate the mass of probability at $Y = 0$. This is the probability that the dot product is zero. We use:

$$PZ^{ES} = (\Pi_{i=1}^{N}(1 - C_i))^{1/q} \qquad (6)$$

where $N$ is the number of nonzero *tf.idf* weights in the query vector, and $q$ is the length of the tokens used. In effect, we are computing the product of all the values

corresponding to the nonzero entries in the query vector. The exponentiation with $\frac{1}{q}$ is to correct for the fact that q-grams are usually not independent. For instance tokens like 'THA' and 'HAT' are more likely to co-occur because they constitute common words like 'THAT'. This simple approximation leads to some very good estimates.

Once we have $\mu^{ES}$, $\sigma^{ES}$, and $PZ^{ES}$, we calculate the selectivity $s$ of the query as:

$$s = (1 - PZ^{ES}) \times in\_cdf(threshold, \mu^{ES}, \sigma^{ES}) \quad (7)$$

where $in\_cdf$ is the CDF for the inverse normal distribution, and $threshold$ is the value obtained from the predicate of the form cosine_similarity(R.a,string) $\geq threshold$.

### 4.2 Algorithm EL

Although Algorithm ES gives us fairly good estimates, we found that instead of simply learning constants $\alpha$ and $\beta$ from a training workload, learning a simple function using linear regression can significantly improve the accuracy of the estimate.

Algorithm EL trains functions to estimate the actual mean and the actual standard deviation for the dot product from $\mu^{ES}$ and $\sigma^{ES}$ computed as in ES using $\alpha = 1$ and $\beta = 1$. In the training phase, we use the data from a set of sample queries that is representative of the workload. We train functions $f_\mu$ and $f_\sigma$ to estimate $\mu^{actual}$ and $\sigma^{actual}$ from $\mu^{ES}$ and $\sigma^{ES}$. We also train a function to better estimate $PZ^{actual}$ using $PZ^{ES}_{corrected}$. If there are changes to the query workload or the data itself, one can retrain these functions to increase their accuracy. (If such retraining is not feasible, then one can resort to the ES algorithm.) For the training function, we empirically tried and evaluated several families of function, including polynomials of various degrees, exponential functions, and combinations of polynomials and exponentials. We found that the following simple family of functions works best for training the estimators:

$$f(x) = c_1 + c_2 x + c_3 e^{-x^2} \quad (8)$$

## 5 Experimental Evaluation

In this section, we present an experimental evaluation of the estimates produced by the ES and EL algorithms on three datasets. The results are largely representative of many other datasets that we tried. The three dataset that we use are SCH, AUT, and HEAD as described below:

**EstimateEL(query,threshold,summary,$f_\mu, f_\sigma, f_{nz}$)**
1. Construct the *tf.idf* vector $u$ for the query.
2. Compute $\mu_{eq} = \Sigma_{i=1}^{N}(\mu_i \times C_i \times u_i)$
3. Compute $\sigma_{eq} = \Sigma_{i=1}^{N}(\sigma_i \times C_i \times u_i)$
4. Compute over nonzero $u_i$ :
5. $nz_{eq} = 1 - (\Pi_{i=1}^{N}(1 - C_i))^{1/k}$
6. $\mu_{EL} = f_\mu(\mu_{eq})$
7. $\sigma_{EL} = f_\sigma(\sigma_{eq})$
8. $nz_{EL} = f_{nz}(nz_{eq})$
9. Compute Estimate = $nz_{EL} \times inv\_normal\_cdf(threshold, \mu_{EL}, \sigma_{EL})$

Figure 3: Estimation using EL

1. SCH consists of 99,632 records with high school names and addresses in the USA. The total size of the dataset is 13MB. The school name field was used for cosine similarity.

2. AUT [2] is a set of 371,022 author names from DBLP totaling 8MB.

3. HEAD [10] contains 119,015 article headlines from the Wall Street Journal totaling 7.5MB.

For each dataset, we randomly chose a set of 50 strings from the database itself, and posed 5 queries with each string by varying the cosine similarity threshold from 0.2 to 0.6 in increments of 0.1. Another (different) set was similarly generated to first train ES and EL. Queries were roughly classified as having Low, Medium, or High selectivity based on whether they selected $> 10\%$, $1\% - 10\%$ or $< 1\%$ of the rows respectively.

The size of the summary structure was less than 3% and took less than 3 minutes to construct in each case. Both ES and EL are efficient and take less than 1 millisecond per query to compute an estimate.

We report the average percentage error in Figures 4, 5, and 6. That is, we report $\frac{|estimate - actual|}{actual} \times 100$. The

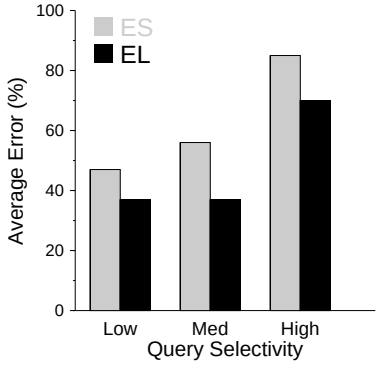


Figure 4: SCH Dataset

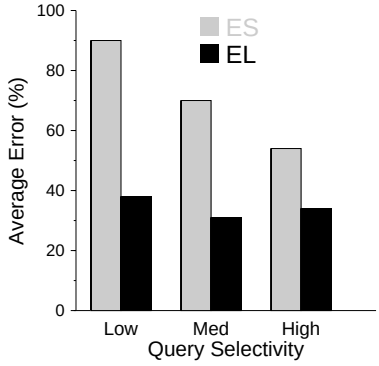


Figure 5: AUT Dataset

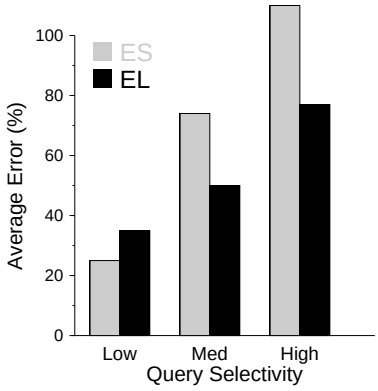


Figure 6: HEAD Dataset

figures show that in each case EL is more accurate than ES by 10 to 50 percentage points. For instance, in Figure 5, in the case of low selectivity queries, ES incurs a 90% error while EL has less than 40% error. Although ES is fairly accurate in many cases, it occasionally has a very large error (eg. high selectivity queries in SCH and HEAD). In all low and medium selectivity cases, the estimates provided by EL have less than 40% error. The error is usually higher in the case of highly selective queries as can be expected. The benefits of using the more complex learning model in EL are evident as they pay off in terms of more accurate estimates.

# 6 Conclusions and Future Work

In this paper, we have presented the problem of estimating the selectivity of cosine similarity predicates. To our knowledge, this is the first paper to address this problem. We discussed why estimating the selectivity of cosine similarity predicates is a very difficult problem, and proposed a solution based on careful empirical observations about the distribution of the dot product of typical queries. We showed that the approach is space efficient (summaries are small in size) and time efficient (estimation time is also small). We also showed that this technique has reasonably good accuracy in practice.

Directions for future work include exploring analytical modeling for the tf.idf dot product, and alternative approaches that might lead to more accurate estimates.

# References

[1] Caetano Traina and Agma J. M. Traina and Christos Faloutsos. Distance Exponent: A New Concept for Selectivity Estimation in Metric Trees. In *ICDE*, pages 195–195, 2000.

[2] Digital Bibliography and Library Project (DBLP), `http://dblp.uni-trier.de/`.

[3] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-grams in a DBMS for approximate string processing. *IEEE Data Engineering Bulletin*, 24(4):28–34, 2001.

[4] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text Joins for Data Cleansing and Integration in an RDBMS. In *ICDE*, pages 729–731, 2003.

[5] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text Joins in an RDBMS for Web Data Integration. In *WWW*, pages 90–101, 2003.

[6] Y. Huang and G. Madey. Web Data Integration Using Approximate String Join. In *WWW*, pages 364–365.

[7] L. Jin and C. Li. Selectivity Estimation for Fuzzy String Predicates in Large Data Sets. In *VLDB*, pages 397–408, 2005.

[8] N. Koudas, A. Marathe, and D. Srivastava. Flexible String Matching Against Large Databases in Practice. In *VLDB*, pages 1078–1086, 2004.

[9] A. Papoulis. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill.

[10] The LDC Corpus Catalog, `http://wave.ldc.upenn.edu/Catalog/`.