THE DBACL TEXT CLASSIFIER

L.A. BREYER

ABSTRACT. This is a summary of the mathematical aspects of the design of **dbacl**, a digramic Bayesian classifier for text documents. **dbacl** computes maximum (relative) entropy models for text corpora and can compute the Bayesian posterior distribution for a given document in terms of any number of previously computed models. Both learning and classifying is $O(n)$ in the number of lines of the input documents.

1. Introduction

dbacl is a digramic Bayesian classifier for text documents. It is a command line tool specifically designed to learn features of text corpora, by building a *maximum entropy*-type model in terms of these features. Feature selection depends on the currently selected locale, and can be specified using regular expressions if required. Computed models can be used to calculate Bayesian posterior probabilities for unknown text documents. The **dbacl** manual page, presented at the end of this document, contains a quick introduction and some examples of usage. The purpose of this document is to explain aspects of the mathematical design. To help the reader unfamiliar with Bayesian theory, we begin with a section outlining the main probabilistics concepts involved.

2. Bayesian text classification and decisions

A text document can be viewed as a very long string of characters. From the statistical perspective, the characters which form the words and sentences do not occur randomly, but obey probabilistic laws which govern the appearance of a character, given the sequence of characters immediately preceding it. Such models are called n-gram character models, where n refers to the number of relevant characters

Date: October, 2002; Revised November 2002, May, June 2004.

1991 *Mathematics Subject Classification.* Primary 60J, Secondary 60F.

Key words and phrases. SPAM, Bayes theorem, email, Maximum entropy.

in the model. **dbacl** uses a *digram* model, so that each character is predicted based on its immediate predecessor. A *trigram* model would depend on the previous two, but requires much more memory.

A text document can also be viewed as a very long string of words. The same statistical ideas apply here too. Shannon (1948) described several examples of such models for English text. Since each possible word is treated like a character in the character based models, this can be viewed as a character based model with a very large alphabet. Practically, these two types of models are very different, however, because individual characters occur so frequently that statistics are easy to estimate, while individual words are comparatively rare. More precisely, word distributions tend to follow a power law, and when classifying an unknown document, it is likely to contain words which have never occurred in the learning corpus.

Good text classification algorithms must therefore be able to predict new words at the same time as weigh existing words when comparing documents. Suppose that we can build a probabilistic model for the contents of an existing text corpus C. This will be represented by a probability measure $\mathbb{P}_C$ on strings, which is concentrated on strings similar to those found in C.

The maximum entropy formalism allows the construction of very general probability measures, by selecting a set of interesting *features* of the learning corpora as representative (Berger et al, 1996; Della Pietra et al, 1997).

Suppose we have constructed probability models $\mathbb{P}_{C_1}, \dots, \mathbb{P}_{C_k}$ for a set of corpora $C_1, \dots, C_k$. *A priori*, we assume that the new documents we wish to classify arise from category $1 \leq j \leq k$ with probability π_j. These probabilities must satisfy $\sum_{j=1}^k \pi_j = 1$. For equally likely categories (representing total ignorance a priori), we would take $\pi_j = 1/k$ for all j.

Let s be the string corresponding to an unknown text document we wish to classify. The *a posteriori* probability that s arises from the probability measure $\mathbb{P}_{C_j}$ (i.e. is most similar to the documents in C_j), is given by Bayes' theorem as

$$\mathbb{P}\big(C_j \,|\, s\big) = K^{-1}\mathbb{P}_{C_j}(s)\pi_j, \qquad K = \sum_{j=1}^{k} \mathbb{P}_{C_j}(s)\pi_j. \tag{1}$$

Note that this assumes that s must be compared only to the corpora $C_1, \dots, C_k$. The normalizing constant K does not allow for other choices, i.e. unclassifiable text.

Such a generalization is possible by modifying K, to account for other categories not directly modelled, but this will not be discussed here.

The simplest decision procedure for classification is the following: *say that s resembles category C_j if the conditional probability $\mathbb{P}(C_j \mid s)$ is greatest.* This is known as the MAP (Maximum A Posteriori) estimate.

More generally, we can introduce a loss function $\mathcal{L}[i, j]$ which represents the loss incurred when s is misclassified as resembling C_j, when in actual fact it ought to resemble C_i. Suppose we pick any decision rule w to do the classification, so that if s is an unknown text document, $w(s) = j$ means that s arises from category C_j under this rule. The best decision according to Bayesian Theory occurs when the expected posterior loss is minimized: *say that s resembles category C_j if $w_*(s) = j$, where w_* minimizes the expected posterior loss over all possible decision rules:*

$$w_* = \arg\min_w \sum_{i=1}^{k} \mathcal{L}[i, w(s)]\mathbb{P}(C_i \mid s).$$

The simplest example occurs when there are only two categories, and there is no loss incurred for correct classifications (i.e. $\mathcal{L}[1,1] = \mathcal{L}[2,2] = 0$). In this case, the optimal Bayesian classification procedure is given by:

$$\textit{Say that } s \textit{ resembles category } C_1 \textit{ if and only if } \quad \frac{\mathbb{P}_{C_1}(s)}{\mathbb{P}_{C_2}(s)} \geq \frac{\pi_2\mathcal{L}[2,1]}{\pi_1\mathcal{L}[1,2]}.$$

Note that **dbacl** calculates only the conditional probabilities $\mathbb{P}(C_j \mid s)$ for each learned category C_j, more general decisions with arbitrary *a priori* probabilities π_j and loss matrices can easily be performed by a wrapper script.

The equation (1) breaks down when there is a single category $C = C_1$ to be classified. Intuitively, **dbacl** should still measure how well the string s fits this single category.

One possible approach is to simply output the probability of s under the model, i.e. $\mathbb{P}_C(s)$. However, in case of strings consisting of more than a few characters only, this is very unlikely to be useful, since by the *concentration of measure* phenomenon, $\mathbb{P}_C$ is highly concentrated around the string C, leaving essentially zero probability mass for most other types of strings.

It is possible to factorize the output score as a product of the document complexity (i.e. number of recognized feature tokens in the document), and the *cross entropy* of the model, which measures the average number of bits per salient feature required, if we compress the string s with an optimal code based on the probability measure $\mathbb{P}_C$. The details are described in Section 8.

A refinement of this idea, described in Section 9.3, consists in outputting a confidence in the null hypothesis that the document belongs to the category, based on the observed value of the score. This approach works even when there is only a single category. **dbacl** can output a confidence percentage related to the empirical distribution of training documents. This percentage is high when the document score is close to the empirical model scores, and low for untypical scores, thus it can be viewed as a confidence for *accepting* the null hypothesis.

An important property of this confidence value is that it is uniformly distributed when the null hypothesis applies for the given document, so it can be viewed as a *p-value* for the purpose of Fisher's χ^2 test. This test allows an easy way of deciding whether a document belongs to some collection of categories: Let $p_1, \dots, p_n$ be the confidence in the computed score for categories $C_1, \dots, C_n$ respectively (converted to a real number in $[0, 1]$), and assume that the models for each category were constructed from independent data, then by Fisher's theorem

$$Y = -2 \sum_{i=1}^{n} \log p_i \sim \chi^2_{2n}, \tag{2}$$

and the resulting quantity $p(Y) = \mathbb{P}(\chi^2_{2n} \leq Y)$ measures the confidence in the validity of the compound hypothesis.

3. Digramic reference measure

For a given text corpus C, the core of the model for C constructed by **dbacl** is a digramic reference measure R_C on tokens. The measure R_C, as its name suggests, is defined in terms of digrams, i.e. pairs of consecutive *characters* appearing in C. This measure is used to assign a baseline probability to every valid token (character string), as follows: if t is a token with character length k, we write $t = t^1 t^2 \dots t^k$, where each t^j is a character. Then

$$R_C(t) = p_C(\diamond, t^1) \Big(\prod_{i=1}^{k} p_C(t^i, t^{i+1}) \Big) p_C(t^k, \diamond),$$

where $\diamond$ is a symbol never appearing in the alphabet.

The choice of appropriate conditional probabilities $p_C(\cdot, \cdot)$ is a problem in itself. Since version 1.8, **dbacl** offers several alternatives, among them

- *Dirichlet*: The conditional probabilities $p_C(\cdot,\cdot)$ are estimated from a hierarchical Dirichlet model (MacKay and Peto, 1995):
$$p_C(u,v) = \frac{\#(v|u) + \alpha\mu(v)}{\sum_r \#(r|u) + \alpha},$$
where μ is a probability measure and $\alpha > 0$, representing the maximum a posteriori estimate of the Dirichlet parameters, and $\#(r|u)$ is the number of times character r occurs in C immediately preceded by character v. The procedure for estimating α and μ follows the approach taken in the previously mentioned paper.
- *Maxent*: The conditionals are computed by maximizing entropy. This method is virtually identical to that discussed in later sections, replacing tokens with digrams, so won't be detailed here.
- *Uniform*: The conditionals are chosen uniformly, i.e.
$$p_C(u,v) = 1/\left|\text{alphabet size}\right|.$$
While the resulting reference measure doesn't depend on the data at all, it has some interesting properties in practice: Firstly, it doesn't bias *against* nonsensical tokens, which are often deliberately inserted in junk mail, and secondly, when used consistently for all models, it automatically *limits* classifications within the vocabulary appearing in the training corpora, because any unknown tokens obtain the same reference measure in all models.

There are two further points worth mentioning briefly:

- By design, **dbacl** ignores all digrams appearing *outside* any valid tokens (we define tokens later). Including counts for these extra digrams would skew classification, since the extra digrams represent essentially formatting data in the text corpus C. The special symbol $\diamond$ is used to mark the beginning and end of token, and has the effect of introducing a natural token length in R_C.
- In international locales, character sets can be essentially arbitrarily large. Consequently, the potential number of digrams is unbounded. **dbacl** copes with this by always calculating R_C in terms of *unibyte* digrams, rather than (strictly speaking) *character* digrams. Thus there are at most $(256)^2$ distinct digrams, and the conditional probabilities $p_C(\cdot,\cdot)$ may or may not be

interpretable as governing character transitions, depending on the chosen character set (in locales where characters require a single byte, the distinction is moot).

Wide character sets, which encode each possible character as a fixed width sequence of bytes, pose an information loss problem, since many characters are encoded with trailing NUL bytes. This makes the locale characters under the R_C measure essentially i.i.d., with a consequent loss of modelling information. The responsibility for using an "informative" character set is thus relegated to the user, who should use a corpus represented in a suitable multibyte character set.

It is easy to see that the measure R_C has unit mass on all tokens. Since $\diamond$ does not appear inside the string t, we can identify R_C with the law of an excursion from $\diamond$ for the markov chain with transition probabilities $p_C(x, y)$. Since the state space is finite, and there are no absorbing states, the length of each excursion is almost surely finite, which gives $\sum_t R_C(t) = 1$.

4. Document representation and features

When **dbacl** is given a text corpus C to learn, it treats the input as a long character string and builds the reference measure R_C. To obtain sophisticated corpus model probabilities however, this stream of characters must be converted into tokens. To cope with large text corpora, this tokenization should be compatible with a single pass through the data.

In the *default model*, non-alphabetic characters are ignored and what is left is a sequence of alphabetic word/tokens converted to lower case. Thus we can write formally $C = (t_1, \ldots, t_n)$, where each t_i is a word/token.

The default model has several limitations which arise out of the word/token representation of C. For example, word/tokens cannot overlap, which makes the construction of n-gram word models programmatically tedious. A better approach is through the use of features.

A feature is a function f which, given a string of characters s, returns a value $f(s) \in \{0, 1, 2, \ldots\}$. Typically, f will simply count the number of times a word/token t occurs in s, written $f_t(s)$. However, features can be more general, and **dbacl** can define features in terms of regular expressions. Most importantly, features can *overlap*.

To introduce the definition of a token associated with a feature, suppose given a finite set G of regular expressions. For our purposes, we shall be interested only in expressions which tag one or more subexpressions. Thus each regular expression $g \in G$ acts on an arbitrary string s as follows:

$$g(s) = \begin{cases} (s_1, \ldots, s_g) & \text{if } s \text{ matches } g \text{ and } s_1, \ldots, s_g \text{ were tagged} \\ \emptyset & \text{if } s \text{ does not match } g \end{cases}$$

In the definition below, we also use the string concatenation operator `cat`, i.e. we have $\texttt{cat}(s_1, s_2, \ldots, s_n) = s_1 \diamond s_2 \diamond \cdots \diamond s_n$.

Definition 1. *Let G be a finite set of regular expressions. A* token *t is a string of the form $t = \texttt{cat} \circ g(s)$, for some string s and some $g \in G$. The* feature *associated with t is the function $f = f_t$ defined by*

$$f_t(s) = \max\Big\{k : \textit{there exist distinct minimal substrings } u_1, \ldots, u_k \textit{ of } s, \\ \textit{such that } \texttt{cat} \circ g(u_j) = t \textit{ for all } j = 1, \ldots, k\Big\}.$$

A feature f_t thus counts the number of disjoint occurrences of the token t in the string s. The set of all features of s is written $\mathcal{F}(s)$. We also write $\mathcal{F}(G) = \cup\{\mathcal{F}(s) : s \text{ a string}\}$.

The *default model* of **dbacl** is obtained by taking

(3) $$G = \{\texttt{"[\^{}[:alpha:]]([[:alpha:]]+)"}, \texttt{"\^{}([[:alpha:]]+)"}\}$$

which gives tokens represented by alphabetic substrings only (modulo the fact that tokens are also converted to lowercase).

At this point, it makes sense to introduce the notion of *admissible string*, since classification will depend only on comparing features of ordinary strings. The set of admissible strings can effectively be enumerated by listing the values $\{f(s) : f \in \mathcal{F}(G)\}$.

Also, to compare different text documents, we shall consider them as very long admissible strings. However, it is difficult to compare strings of very different sizes. A training corpus might contain millions of characters, while the future text to be classified may only contain hundreds. Moreover, since the strings are admissible, they may contain sections of irrelevant characters, which do not appear in any token. Thus we also need a notion of complexity.

Definition 2. *Consider the equivalence relation $\sim$ on strings given by $s \sim s'$ if and only if $f(s) = f(s')$ for all $f \in \mathcal{F}(G)$. Each equivalence class is called a* G-admissible *string. The* G-complexity *of a G-admissible string s is defined as*

$$\chi_G(s) = \sum_{f \in F(G)} f(s).$$

When the set G is understood, we will drop the G-prefix. To perform a calculation later, we will need the following assumption on G:

Definition 3. *The set of regular expressions G is* simple *if there exist admissible strings which exhibit every combination of feature counts.*

It is easy to see that simple sets G exist. For example, if G matches single characters only, then a string of complexity k has exactly k characters, and by identifying the permutations, we obtain all admissible strings of complexity k. The *default model* is another example of a simple set G. However, n-gram models for $n > 1$ do not arise from a simple set G.

For admissible strings, the reference measure R_C induces a natural σ-finite measure $\mathbb{R}_C$ on the set of all admissible strings, as follows: if $f = f_t$ is the feature corresponding to a token t, we shall (ab)use notation and write [1] $R_C(f) = R_C(t)$. Then if s is admissible,

$$\mathbb{R}_C(s) = \chi(s)! \prod_{f \in \mathcal{F}(G)} \frac{1}{f(s)!} R_C(f)^{f(s)}. \tag{4}$$

This measure is not a probability. However, for a given complexity k and a *simple* set G, the set of all admissible strings s with $\chi(s) = k$ has unit $\mathbb{R}_C$-mass, due to $\sum_f R_C(f) = 1$. Thus it can be viewed as a family of multinomial probability distributions, indexed by complexity.

In the *default model* used by **dbacl**, the measure $\mathbb{R}_C$ has a straightforward interpretation. Since tokens are lower case alphabetic substrings of s, separated

[1] Provided we restrict ourselves only to features of the form f_t for some t, this notation essentially sets up a bijective correspondence between f_t and t, such that the measure $R_C(f)$ is the image of $R_C(t)$.

by non-alphabetic characters, $\mathbb{R}_C$ is the image, under the equivalence relation $\sim$, of the law of a Markov chain on the characters of a lower case string, where all non-alphabetic characters are amalgamated into a single artificial state $\diamond$. The complexity of s is the number of excursions from $\diamond$, and its initial value is $\diamond$.

With more general types of tokens, the measure $\mathbb{R}_C$ is not so easy to interpret.

5. Maximum Entropy: the simple case

Let G be a *simple* set of regular expressions. Given a corpus C, the digramic reference measure R_C is first computed, as described in an earlier section. This measure is important, since the input corpus C may not contain all possible tokens which **dbacl** will be exposed to in later use. However, for a given token t, the probability $R_C(t)$ can always be calculated. Thus by incorporating R_C, we obtain automatically a form of text *smoothing* (Zhai and Lafferty, 2001).

Viewing the corpus C as a single admissible string with complexity $\chi(C)$, **dbacl** computes the Maximum (Relative) Entropy probability $\mathbb{P}_C$, with respect to $\mathbb{R}_C$, subject to constraints on the mean values of interesting features. The solution $\mathbb{P}_C$ is thus a member of the convex set of probabilities (on admissible strings with complexity $\chi(C)$)

$$\Pi_C = \big\{\mathbb{Q} : \int f\, d\mathbb{Q} \leq a_f,\ f \in \mathcal{F}(C)\big\}. \tag{5}$$

Here the nonnegative real numbers a_f represent bounds on the means for the various features of C (the choice of a_f is discussed later). Provided Π_C is both nonempty and a finite Kullback-Leibler distance from $\mathbb{R}_C$, the probability of interest therefore satisfies

$$\mathbb{P}_C = \arg\min_{\mathbb{Q}\in\Pi_C} \int \log\big(d\mathbb{Q}/d\mathbb{R}_C\big) d\mathbb{Q} < \infty. \tag{6}$$

In that case, it is well known (Csiszar, 1984) that the functional form of $\mathbb{P}_C$ is

$$d\mathbb{P}_C(s) = Z_C^{-1} \exp\Big(\sum_{f\in\mathcal{F}(C)} \lambda_f f(s)\Big) d\mathbb{R}_C(s), \tag{7}$$

where the λ_f are real numbers, and Z_C is the appropriate normalizing constant.

Moreover, it can be shown that the λ_j are all nonnegative, and in fact $\lambda_j = 0$ if and only if $\int f d\mathbb{P}_C < a_f$.

Substituting (4) into (7), we obtain

$$Z_C = \chi(C)! \sum_{\substack{s \text{ adm.} \\ \chi(s)=\chi(C)}} \prod_{f\in\mathcal{F}(G)} \frac{1}{f(s)!}\Big(\exp\big[\lambda_f 1_{\{f\in\mathcal{F}(C)\}} + \log R_C(f)\big]\Big)^{f(s)}$$
$$= \Big(\sum_{f\in\mathcal{F}(G)} \exp\big[\lambda_f 1_{\{f\in\mathcal{F}(C)\}} + \log R_C(f)\big]\Big)^{\chi(C)}$$
$$= \Big(1 - \sum_{f\in\mathcal{F}(C)} R_C(f) + \sum_{f\in\mathcal{F}(C)} e^{\lambda_f} R_C(f)\Big)^{\chi(C)},$$

which follows, assuming G is *simple*, by enumerating the admissible strings via their feature counts. Inserting (7) into (6) and using the assumption $\mathbb{P}_C \in \Pi_C$, we find that the parameters λ_f must be chosen to minimize

$$-\chi(C)\log\Big[1 - \sum_{f\in\mathcal{F}(C)} R_C(f) + \sum_{f\in\mathcal{F}(C)} e^{\lambda_f} R_C(f)\Big] + \sum_{f\in\mathcal{F}(C)} \lambda_f a_f. \tag{8}$$

This expression is convex as a function of each λ_f, and can be optimized one component at a time, but some care must be taken to cope with numerical precision errors when $\chi(C)$ is large. Differentiating (8) with respect to a single component λ_f and solving for a root gives the equation for an update $\lambda_f \to \lambda'_f$, $z_C \to z'_C$:

$$\lambda'_f = \log z_C + \log\frac{a_f}{\chi(C)} - \log R_C(f), \tag{9}$$
$$z'_C = 1 - \sum_{f\in\mathcal{F}(C)} R_C(f) + \sum_{f\in\mathcal{F}(C)} e^{\lambda_f} R_C(f).$$

With this particular form of update, all weights can be updated simultaneously before recomputing Z_C, which is much more expensive.

It is apparent from (7) that the solution measure $\mathbb{P}_C$ can be written

$$\mathbb{P}_C(s) = \chi(s)! \cdot \prod_{f\in\mathcal{F}(G)} \frac{1}{f(s)!}\widetilde{R}_C(f)^{f(s)}, \tag{10}$$

$$\text{where } \widetilde{R}_C(f) = \begin{cases} Z_C^{-1} e^{\lambda_f} R_C(f) & \text{if } f \in \mathcal{F}(C) \\ Z_C^{-1} R_C(f) & \text{otherwise.} \end{cases}$$

In (10), the measure $\widetilde{R}_C$ is a probability on $\mathcal{F}(G)$, and the same formula can be used for strings of all complexities, resulting in a consistent σ-finite measure on all admissible strings. Furthermore, for any $\chi > 0$, $\mathbb{P}_C$ is a probability measure when restricted to strings s with $\chi_G(s) = \chi$.

It remains to discuss the choice of constraints, and verify that a maximum entropy solution exists.

It is natural to impose the assumption that at least the empirical measure belongs to Π_C, or equivalently that $a_f \geq f(C)$. Provided $R_C(f) > 0$ for all $f \in \mathcal{F}(C)$, the requirement (6) follows easily, and therefore a maximum entropy solution exists.

However, the empirical measure in itself has limited predictive value, since it places all of its mass on the tokens observed in the training corpus C. To improve this deficiency, It makes sense to only consider the empirical measure as a good model of some putative $\mathbb{Q} \in \Pi_C$, *conditionally on observing only the tokens belonging to* $\mathcal{F}(C)$. This constraint is actually sufficient to derive suitable constants a_f, as follows:

According to this assumption, let $A = \{\omega : f(\omega) > 0 \text{ for all } f \in \mathcal{F}(C)\}$. The empirical measure has support on A. For any feature $f \in \mathcal{F}(C)$, we therefore write

$$\int f(\omega) d\mathbb{Q}(\omega) = \int f(\omega) d\mathbb{Q}(\omega|A) \cdot \mathbb{Q}(A) + 0 \cdot \mathbb{Q}(A^c) \approx (f(C)/\chi(C)) \cdot \mathbb{Q}(A).$$

Thus, if we take $a_f = f(C)$, then we obtain a satisfactory set Π_C, containing both the empirical measure and a range of possibilities which take unseen tokens into account (in those cases, $\mathbb{Q}(A) < 1$).

6. Maximum Entropy: the sequential case

In this section, we generalize the maximum entropy problem to handle the case of n-grams. Unfortunately, for $n \geq 2$, the set G is no longer *simple*, and working in terms of admissible strings is difficult.

Definition 4. *A set of regular expressions G is called* sequential *if, wenever a string of the form $x_1 \diamond \cdots \diamond x_n$ is matched (with each x_i not containing the symbol $\diamond$), then all subsets of the form $x_k \diamond \cdots \diamond x_{k+j}$, for $1 \leq k \leq n$ and $0 \leq j \leq n-k$, are also matched. We write $\mathcal{F}_j(G)$ for the subset of $\mathcal{F}(G)$ which is of the form $f_{a_1 \diamond \cdots \diamond a_j}$.*

For a fixed χ, we shall work with strings of the form $s = x_1 \diamond \cdots \diamond x_\chi$ only. We write

$$\mathbb{R}_C\big(x_1 \diamond \cdots \diamond x_\chi\big) = \prod_{i=1}^{\chi} R_C(x_i),$$

which is compatible with the earlier definition if G is *simple* (if we identify $\mathcal{F}_1(G)$-equivalent strings we obtain the earlier definition). Consider also the sets of probability measures

$$\Pi_k = \left\{\mathbb{Q} : \int f d\mathbb{Q} \le a_f, \quad f \in \mathcal{F}(C) \cap \left(\cup_{j=1}^k \mathcal{F}_j(G)\right)\right\}, \quad k = 1, 2, \ldots, \chi,$$

which are clearly decreasing (assuming the constraints a_f are consistent). By the Pythagorean theorem of Kullback-Leibler divergences, the maximum entropy solution $\mathbb{P}_{C,k} \in \Pi_k$ (relative to $\mathbb{R}_C$) has the recursive form

$$\begin{aligned} d\mathbb{P}_{C,m}(s) = d\mathbb{P}_C(s) &= Z_{C,m}^{-1} \prod_{k=1}^m \exp\left[\sum_{f \in \mathcal{F}_k(G) \cap \mathcal{F}(C)} \lambda_f f(s)\right] d\mathbb{R}_C(s), \\ &= \widetilde{Z}_{C,m}^{-1} \exp\left[\sum_{f \in \mathcal{F}_m(G) \cap \mathcal{F}(C)} \lambda_f f(s)\right] d\mathbb{P}_{C,m-1}(s), \end{aligned} \tag{11}$$

which implies that we can calculate the weights $\lambda_f, f \in \mathcal{F}(C) \cap \mathcal{F}_k(G)$ inductively, through optimization of

$$-\log \widetilde{Z}_{C,k} + \sum_{f \in \mathcal{F}(C) \cap \mathcal{F}_k(G)} \lambda_f a_f, \quad k = 1, \ldots m. \tag{12}$$

Due to the formula $\log Z_{C,k} = \log \widetilde{Z}_{C,k} + \log Z_{C,k-1}$, the optimal set of constants λ_f, at the k-th stage of optimization, remains the same if we replace $\log \widetilde{Z}_{C,k}$ with $\log Z_{C,k}$ in (12). To compute $Z_{C,k}$, we note first that for any $k < m \le \chi$,

$$\sum_{b_{k+1},\ldots b_m} f_{a_1 \diamond \cdots \diamond a_k \diamond b_{k+1} \diamond \cdots \diamond b_m}(s) \le f_{a_1 \diamond \cdots \diamond a_k}(s) \le \sum_{b_{k+1},\ldots b_m} f_{a_1 \diamond \cdots \diamond a_k \diamond b_{k+1} \diamond \cdots \diamond b_m}(s) + m - k,$$

and the first inequality is strict only if $a_1 \diamond \cdots \diamond a_k$ occurs within the suffix $x_{\chi-m+1} \diamond \cdots \diamond x_\chi$ of the string s. In particular, we have

$$\sum_{a_1,\ldots,a_k} \lambda_{a_1 \diamond \cdots \diamond a_k} f_{a_1 \diamond \cdots \diamond a_k}(s) \approx \sum_{b_1,\ldots,b_m} \lambda_{b_1 \diamond \cdots \diamond b_k} f_{b_1 \diamond \cdots \diamond b_m}(s), \tag{13}$$

with an error at most $\max_t |\lambda_t| (m-k)$. To ease the notation, when $f = f_{a_1 \diamond \cdots \diamond a_m}$, we write $\tau^j f = f_{a_1 \diamond \cdots \diamond a_{m-j}}$ for $j = 1, \ldots, m-1$. Also, $f_t \diamond y = f_{t \diamond y}$. A geometrical way of understanding this approximation is given in Figure 1.

$$
\begin{array}{ccccccc}
(k=3) & & & \lambda_{123} & \lambda_{234} & \lambda_{345} & \lambda_{456} \\
 & & & | & | & | & | \\
(k=2) & & \lambda_{12} & \lambda_{23} & \lambda_{34} & \lambda_{45} & \lambda_{56} \\
 & & & | & | & | & | \\
(k=1) & \lambda_1 & \lambda_2 & \lambda_3 & \lambda_4 & \lambda_5 & \lambda_6
\end{array}
$$

FIGURE 1. Approximation (13): For any string $s = a_1 \diamond \cdots \diamond a_6$, arrange the weights for all $1, 2, 3$-grams in an array as above. Then for each row k, the left hand side of (13) is the sum of weights on that row, while the right hand side of (13) is the sum of those weights which have a weight above them on row $m = 3$. The sum of the weights along each line has the form $\lambda_{xyz} + \lambda_{yz} + \lambda_z = \lambda_{xyz} + \lambda_{\tau^1(xyz)} + \lambda_{\tau^2(xyz)}$.

Now from (13), by summing over all strings s of the form $x_1 \diamond \cdots \diamond x_\chi$, we obtain for large values of χ,

$$
\begin{aligned}
Z_{C,k} &= Z_{C,k-1} \sum_s \exp\Bigg[\sum_{f \in \mathcal{F}_k(G) \cap \mathcal{F}(C)} \lambda_f f(s) \Bigg] d\mathbb{P}_{C,k-1}(s) \\
&\approx \sum_s \exp\Bigg[\sum_{f \in \mathcal{F}_k(G)} \Big(\lambda_f 1_{\{f \in \mathcal{F}(C)\}} + \sum_{j=1}^{k-1} \lambda_{\tau^j f} 1_{\{\tau^j f \in \mathcal{F}(C)\}} \Big) f(s) \Bigg] \mathbb{R}_C(s).
\end{aligned}
$$

As a notational simplification, below we shall write $\lambda_f^C = \lambda_f$ if $f \in \mathcal{F}(C)$, and $\lambda_f^C = 0$ otherwise.

Now for each s, the substrings $x_1, \ldots, x_\chi$ are independent under $\mathbb{R}_C$, and each $f \in \mathcal{F}_k(G)$ depends only on consecutive substrings $x_t, \ldots, x_{t+k}$. Therefore, by a simple modification of the standard Sanov theorem,

(14)

$$
\begin{aligned}
\lim_{\chi \to \infty} (\chi - k + 1)^{-1} \log Z_{C,k} &= \sup_\nu \left(\int_{\mathcal{F}_k(G)} \Big[\lambda_f^C + \sum_{j=1}^{k-1} \lambda_{\tau^j f}^C \Big] \nu(df) - \frac{1}{k} D\big(\nu \| R_C^{\otimes k}\big) \right) \\
&= \frac{1}{k} \log \Bigg[\sum_{f \in \mathcal{F}_k(G)} \exp\Big(k\lambda_f^C + \sum_{j=1}^{k-1} k\lambda_{\tau^j f}^C \Big) R_C^{\otimes k}(f) \Bigg],
\end{aligned}
$$

because the optimal ν for a problem of the form $\max_\nu \Big(\int h d\nu - D(\nu \| \mu) \Big)$ is always given by $d\nu = e^h d\mu / \int e^h d\mu$.

While the term in square brackets still must be evaluated over all possible features $f \in \mathcal{F}_k(G)$, a simplification is possible by noting that it equals $\Phi_{C,k,k}$, where

$$\Phi_{C,k,l} := \sum_{f\in\mathcal{F}_l(G)} \exp\Big(k\lambda_f^C + \sum_{j=1}^{l-1} k\lambda^C_{\tau^j f}\Big) R_C^{\otimes l}(f). \tag{15}$$

Since most of the constants λ_f^C are zero, we can expand this recursively, with the boundary case $\Phi_{C,k,0} = 1$:

$$\begin{aligned}
\Phi_{C,k,l} &= \sum_{\lambda_f^C\neq 0} \exp\Big(k\lambda_f^C + \sum_{j=1}^{l-1} k\lambda^C_{\tau^j f}\Big) R_C^{\otimes l}(f) + \sum_{\lambda_f^C=0} \exp\Big(k\lambda_f^C + k\sum_{j=1}^{l-1} \lambda^C_{\tau^j f}\Big) R_C^{\otimes l}(f) \\
&= \sum_{\lambda_f^C\neq 0} \exp\Big(k\lambda_f^C + \sum_{j=1}^{l-1} k\lambda^C_{\tau^j f}\Big) R_C^{\otimes l}(f) + \\
&\qquad \sum_y \sum_{\lambda^C_{\tau^1 f\diamond y}=0} \exp\Big(k\lambda^C_{\tau^1 f} + \sum_{j=1}^{l-2} k\lambda^C_{\tau^j\tau^1 f}\Big) R_C^{\otimes l-1}(\tau^1 f)\cdot R_C(y) \\
&= \sum_{\lambda_f^C\neq 0} \exp\Big(k\lambda_f^C + \sum_{j=1}^{l-1} k\lambda^C_{\tau^j f}\Big) R_C^{\otimes l}(f) + \\
&\qquad \sum_{\tau^1 f\in\mathcal{F}_k(G)} \exp\Big(k\lambda^C_{\tau^1 f} + \sum_{j=1}^{l-2} k\lambda^C_{\tau^j\tau^1 f}\Big) R_C^{\otimes l-1}(\tau^1 f)\Big(1 - \sum_{\tau^1 f\diamond y\in\mathcal{F}_l(C)} R_C(y)\Big) \\
&= \sum_{f\in\mathcal{F}_k(C)} \Big(e^{k\lambda_f} - 1\Big)\exp\Big(k\lambda^C_{\tau^1 f} + \cdots + k\lambda^C_{\tau^{l-1} f}\Big) R_C^{\otimes l}(f) + \Phi_{C,k,l-1}.
\end{aligned}$$

Summarizing the calculations so far, at the k-th stage, we can solve the following optimization problem, which is asymptotically equivalent to (12),

$$-\tfrac{(\chi-k+1)}{k}\log\Big[\sum_{f\in\mathcal{F}_k(C)}\Big(e^{k\lambda_f^C}-1\Big)e^{\sum_j k\lambda^C_{\tau^j f}} R_C^{\otimes k}(f) + \Phi_{C,k,k-1}\Big] + \sum_{f\in\mathcal{F}_k(C)} \lambda_f^C a_f. \tag{16}$$

the sums are only over features $f \in \mathcal{F}(C)$, and the number $\Phi_{C,k,k-1}$ is independent of $f \in \mathcal{F}_k(G)\backslash\mathcal{F}_{k-1}(G)$, hence can be treated essentially as a constant. Following (9), we thus solve (16) by calculating $\Phi_{C,k,k-1}$ and making updates $\lambda_f \to \lambda'_f$, $z_C \to z'_C$ of the form

$$\begin{aligned}
\lambda'_f &= \frac{1}{k}\log\Big[z_C\frac{a_f}{\chi-k+1}\Big] - \sum_{j=1}^{k-1}\lambda^C_{\tau^j f} - \frac{1}{k}\log R_C^{\otimes k}(f), \\
z'_C &= \Phi_{C,k,k-1} + \sum_{f\in\mathcal{F}_k(G)}\Big(e^{k\lambda_f^C}-1\Big)e^{\sum_j k\lambda^C_{\tau^j f}} R_C^{\otimes k}(f).
\end{aligned}$$

7. Constraints in the sequential models

We now come to discussing the natural form of the constraints a_f, and the probabilistic interpretation of sequential models. In the following, we shall denote the j-gram complexity of s by

$$\chi_j(s) = \sum_{f \in \mathcal{F}_k(G)} f(s).$$

Note that if $s = x_1 \diamond \cdots \diamond x_\chi$, then $\chi_j(s) = \chi - j + 1$.

7.1. **Bigram model.** We can build a pure word bigram[2] model by taking the set G of regular expressions as

$$G = \{\texttt{"[\^{}[:alpha:]]([[:alpha:]]+)[\^{}[:alpha:]]+([[:alpha:]]+)"}, \tag{17}$$
$$\texttt{"\^{}([[:alpha:]]+)[\^{}[:alpha:]]+([[:alpha:]]+)"}\}.$$

Every time a match (s_1, s_2) is found in the corpus C, it is converted into the token $t = s_1 \diamond s_2$. Let us call any alphabetic word such as s_1 or s_2 a *state*. Then from (11), we have for any string s with complexity $\chi_2(s)$,

$$\mathbb{P}_C(s) = \prod_{x,y} \widetilde{R}_C(x \diamond y)^{f_{x\diamond y}(s)}, \tag{18}$$

$$\text{where } \widetilde{R}_C(x \diamond y) \propto \begin{cases} e^{\lambda_{x\diamond y}} R_C(x) R_C(y) & \text{if } f_{x\diamond y} \in \mathcal{F}(C), \\ R_C(x) R_C(y) & \text{otherwise,} \end{cases}$$

and where the product extends over all possible pairs of states, and $f_{x\diamond y}$ is the feature associated with the token $x \diamond y$, which counts the number of occurrences of $x \diamond y$.

Now define

$$\nu(x) = \sum_y \widetilde{R}_C(x \diamond y), \quad P(x,y) = \widetilde{R}_C(x \diamond y)/\nu(x), \tag{19}$$

which gives $\sum_y P(x,y) = 1$ for all x, and note that $\sum_x \nu(x) = 1$ because $\widetilde{R}_C$ is a probability measure over all possible tokens, Inserting (19) into (18), and noticing that the second product below has only $\chi_2(C)$ terms (all other exponents are zero),

[2] We reserve the term digram for a character pair. A bigram can be consecutive words, or more generally any two strings tagged by a regular expression.

which we can enumerate as successive pairs in the sequence $x_0, x_1, \dots, x_{\chi_1(C)}$, we obtain

$$\begin{aligned}\mathbb{P}_C(s) &= \left(\prod_{x,y} \nu(x)^{f_{x\diamond y}(s)}\right) \prod_{x,y} P(x,y)^{f_{x\diamond y}(s)} \\ &= \pi(s) \prod_{t=0}^{\chi_2(C)} P\big(x_t, x_{t+1}\big).\end{aligned}$$

When several sequences are possible, we choose the minimal one under lexicographic ordering. From the measure $\pi(s)$, we define

$$\pi(x_0) = \pi\big(\{s : s = x_0 \diamond x_1 \diamond \cdots \diamond x_{\chi_1(C)}\}\big),$$

and using the facts that $\sum_s \mathbb{P}_C(s) = 1$ and $\sum_y P(x,y) = 1$ together, it follows that $\sum_x \pi(x) = 1$. Thus π can now be seen as the initial distribution of a finite Markov chain $X_0, \dots, X_{\chi_1(C)}$ with law $\mathbb{P}_C$,

$$\mathbb{P}\Big[X_0 = x_0, X_1 = x_1, \dots, X_{\chi(C)} = x_{\chi(C)}\Big] = \pi(x_0) \prod_{t=0}^{\chi_2(C)} P(x_t, x_{t+1}). \tag{20}$$

If we compare (20) with (18), we see that $f_{x\diamond y}(C)$ must be the empirical number of transitions $\{X_t = x, X_{t+1} = y\}$ in the corpus C, and if we proceed as before in the default model, we should take $a_f = f(C)$.

7.2. **Pure n-gram models.** More generally, it also makes sense to tag n-grams via a suitable set of expressions G. Since regular expressions cannot straddle newline characters, there are diminishing returns with this idea in the current **dbacl** implementation. However, since regular expression matching is often polynomial in the size of the string, this limitation ensures acceptable training and classification speed.

When matching n-grams, we proceed as in the bigram example, and obtain a characterization of $\mathbb{P}_C$ as a random field. Tokens have the form $t = x_1 \diamond x_2 \diamond \cdots \diamond x_n$, and a string s with complexity $\chi_1(s)$ is interpreted as a *configuration* of $\chi_1(s)$ distinct sites. Then from (7), the measure $\mathbb{P}_C$ is a Gibbs random field where each site interacts with at most $n - 1$ others.

For a non-symmetric form similar to (19), we can also define

$$
\begin{aligned}
\widetilde{R}_C(x_1 \diamond \cdots \diamond x_n) &\propto \begin{cases} \exp\bigl(\lambda_{x_1\diamond\cdots\diamond x_n}\bigr) \prod_{k=1}^n R_C(x_k) & \text{if } f_{x_1\diamond\cdots\diamond x_n} \in \mathcal{F}(C) \\ \prod_{k=1}^n R_C(x_k) & \text{otherwise,} \end{cases} \\
\nu(x_1, \ldots, x_{n-1}) &= \sum_y \widetilde{R}_C(x_1 \diamond \cdots \diamond x_{n-1} \diamond y), \\
P(x_1, \ldots, x_{n-1}; y) &= \widetilde{R}_C(x_1 \diamond \cdots \diamond x_{n-1} \diamond y)/\nu(x_1, \ldots, x_{n-1}),
\end{aligned}
$$

and then we obtain the higher order Markovian form

$$
\mathbb{P}_C(s) = \pi(x_0, \ldots, x_{n-2}) \prod_{t=0}^{\chi_n(C)} P\bigl(x_t, \ldots, x_{t+n-2}; x_{t+n}\bigr).
$$

The measure π is a probability, as is apparent from using the relations $\sum_s \mathbb{P}_C(s) = 1$, and $\sum_y P(x_1, \ldots, x_{n-1}; y) = 1$. Comparing with (11), we find once again that we should take $a_f = f(C)$.

7.3. **General sequential models.** For a general sequential set G, the interpetation is essentially the same. Suppose that G matches k-grams for $k = 1, \ldots, n$. From (11), it is apparent that the Maximum Entropy model associated with G is an n-gram model based not on $\mathbb{R}_C$, but rather $\mathbb{P}_{C,n-1}$. The latter measure is a smoothed version of $\mathbb{R}_C$, since when an unknown feature $f \in \mathcal{F}_n(G)\backslash\mathcal{F}(C)$ is encountered, in the former case the associated token $x_1 \diamond \cdots \diamond x_n$ is predicted only from the digramic reference measure $\mathbb{R}_C$, while in the latter case, any recognized subset $x_1 \diamond \cdots \diamond x_k$ for $k < n$ is taken into account as well.

Since the nature of the model is nevertheless an n-gram model, it makes sense to take $a_f = f(C)$ for $f \in \mathcal{F}_n(G) \cap \mathcal{F}(C)$. Now the recursive interpretation of $\mathbb{P}_{C,n-1}$ means that it corresponds to a maximum entropy $(n-1)$-gram model, which suggests we take also $a_f = f(C)$ for $f \in \mathcal{F}_{n-1}(G) \cap \mathcal{F}(C)$, and so on, until we arrive at a complete specification of the constraints.

Remark. The smoothing just described doesn't take into account all the possible k-gram subsets of $x_1 \diamond \cdots \diamond x_k$, as one might expect. When the complexity $\chi_n(s)$ is suitably large, the approximation (13) is certainly good enough, and of course the large deviations take effect. For small values of $\chi_n(s)$, there is room for improvement, which can be handled by more general smoothing. A general form can be obtained with a probability measure μ_n:

$$f_{a_1\diamond\cdots\diamond a_k}(s) \approx \sum_{a_1\diamond\cdots\diamond a_k \subseteq t \in \mathcal{F}_n(G)} \mu_n(t) f_t(s),$$

where the sum is over those n-grams t which contain $a_1 \diamond \cdots \diamond a_k$ as a subset. This potential improvement is not implemented in **dbacl** due to the added complexity of calculation.

8. Text classification and mixed models

The equations for text classification are obtained by inserting (10) and (11) directly into (1).

8.1. **Simple case.** Major simplifications occur when comparing several corpora $C_1, \dots C_k$ whose model probabilities are based upon the same set of regular expressions G. If s is a string to be classified, then its complexity $\chi(s)$ is the same under each model, as are the feature counts $\{f(s) : f \in \mathcal{F}(G)\}$. Consequently, we have

$$\mathbb{P}(C_j \mid s) = K^{-1} \exp\Big(\sum_t f_t(s) \log \widetilde{R}_{C_j}(t)\Big), \quad K = \sum_{j=1}^k \exp\Big(\sum_t f_t(s) \log \widetilde{R}_{C_j}(t)\Big).$$

These equations are used by **dbacl** whenever the competing categories are based on the same set of regular expressions. This does not apply if the models are based on functionally equivalent expressions whose string representations are different. Evaluation of the sums for each category can be done simultaneously as the data is read, obtaining an $O(n)$ classification in the number n of lines. As each line is processed with one or more regular expressions, the cost involved is also polynomial in the average length of each line of s. Also, the memory cost for (fast) classification is clearly $O\big(\chi(C_1) + \cdots + \chi(C_k)\big)$ in this case.

The computations are more involved when the training corpora $C_1, \dots C_k$ depend on different sets of regular expressions $G_1, \dots, G_k$ say. The feature counts $\{f(s) : f \in G_k\}$ must be taken into account for each k separately, and the complexity of s varies, and must therefore be written $\chi_j(s)$. Using Stirling's formula $n! \sim \sqrt{2\pi n} \cdot n^n e^{-n}$, we have from (10) the approximation

$$\mathbb{P}(C_j \mid s) \sim \exp\Bigg[\sum_{f \in \mathcal{F}(G_j)} f(s) \log \widetilde{R}_{C_j}(f) + \frac{1}{2} \log\big[2\pi\chi_j(s)\big] + \chi_j(s) \log \chi_j(s)$$

$$- \sum_{\substack{f\in\mathcal{F}(G_j)\\ f(s)>0}} \Big(\frac{1}{2}\log\big[2\pi f(s)\big] + f(s)\log f(s)\Big)\Big],$$

$$= \exp\Big[-\chi_j(s) D(\ell_j(s)\|\widetilde{R}_{C_j}) + \frac{1}{2}\log\big[2\pi\chi_j(s)\big] - \sum_{\substack{f\in\mathcal{F}(G_j)\\ f(s)>0}} \frac{1}{2}\log\big[2\pi f(s)\big]\Big],$$

where $\ell_j(s)(f) = f(s)/\chi_j(s)$ is the empirical feature distribution of s (i.e. a measure on $\mathcal{F}(G)$), and $D(\cdot\|\cdot)$ is the Kullback-Leibler distance on $\mathcal{F}(G)$. This shows that the exponent is essentially proportional to the complexity of s as $\chi_j(s)$ gets large. Unfortunately, even if we drop the last two correction factors in the limit, calculating $D(\ell_j(s)\|\widetilde{R}_{C_j})$ still requires the storage in memory of all the final counts $f(s)$. Thus we see that for mixed multinomial models, calculating $\mathbb{P}(C_j \mid s)$ is unavoidably $O\big(\chi(s) + \chi(C_1) + \cdots + \chi(C_k)\big)$ in memory storage.

The special case when a single category $C = C_1$ is provided needs a different approach. From the above computations, the probability of generating a string s from the probability $\mathbb{P}_C$ is exponentially small, with exponent $\chi(s)$. We assume that the features of s are stationary with marginal distribution $\mathbb{P}$ (a probability on $\mathcal{F}(G)$), and if we take the limit we obtain

$$\lim_{\chi(s)\to\infty} \chi(s)^{-1}\log\mathbb{P}_C(s) = -D(\mathbb{P}\|\widetilde{R}_C).$$

If the logarithm is taken to the base 2, then the resulting *cross entropy* , defined as $-\chi(s)^{-1}\log_2\mathbb{P}_C(s)$ and measured in bits, has the following alternative intepretation: if we compress the admissible string s with an arithmetic coder, and using the model $\mathbb{P}_C$, then the cross entropy gives the number of bits needed on average for each encoded feature. This quantity can be used when choosing the best (most parsimonious) model according to the *Minimum Description Length Principle.*

8.2. **Sequential case.** Due to the absence of factorial terms, the case when G is sequential is less cumbersome. From (11) and the Large Deviation approximation (Sanov), if m is the highest n-gram order in the model, then

$$\mathbb{P}(C_j \mid s) \approx K^{-1}\exp\Big(\sum_{f\in\mathcal{F}(C_j)} \lambda_t f(s) + \sum_{f\in\mathcal{F}_1(G)} \log R_{C_j}(f) - \chi_m(s)\log\hat{Z}_{C_j,m}\Big),$$

$$K = \sum_{j=1}^{k}\exp\Big(\sum_{f\in\mathcal{F}(C_j)} \lambda_t f(s) + \sum_{f\in\mathcal{F}_1(G)} \log R_{C_j}(f) - \chi_m(s)\log\hat{Z}_{C_j,m}\Big),$$

where $\hat{Z}_{C_j,m}$ was calculated as part of the entropy maximization (14), (15) as

$$\log \hat{Z}_{C_j,m}(s) = \frac{1}{m} \log\Big[\sum_{f\in\mathcal{F}_m(C_j)} \Big(e^{m\lambda_f + \sum_j m\lambda_{\tau^j f}} - 1\Big) R_{C_j}^{\otimes m}(f) + \Phi_{C_j,m,m-1}\Big].$$

However, when the competing categories are based on different sets of regular expressions, care must be taken to count each matched feature the correct number of times. This can be somewhat tricky with a single pass in case of complicated overlapping tokens, so **dbacl** makes several passes over each line of the string s, one for each regular expression in G. This method takes advantage of the fact that regular expressions do not straddle newlines, since one full line is kept in memory for regular expression matching anyway.

Every simple model also has a sequential interpretation, given by (10), when all factorial coefficients are omitted. Probabilistically, the simple model describes equivalence classes of strings, while the corresponding sequential model describes all actual permutations. Bayes' formula (1) cannot be directly applied with mixed string interpretations, so when comparing sequential and multinomial types of models, **dbacl** converts them all to sequential form.

For simple models, if the option *-M* is selected, **dbacl** will perform the full multinomial score calculation, e.g. for purposes of comparison with other software. Note that the cross entropy reported in the multinomial case is always lower than in the sequential case, since feature ordering, while irrelevant in both interpretations, is only ignored in the multinomial case. There is no difference in the classification results for sequential or multinomial unigram models, since the extra terms only depend on the empirical measure of the document being classified.

In the case of a single (sequential) category, the cross entropy is calculated in the usual way, namely $-\chi_m(s)^{-1}\log \mathbb{P}_{C,m}(s)$. Under suitable stationarity assumptions, the Shannon-McMillan-Breiman theorem guarantees that this quantity converges as $\chi_m(s)$ tends to infinity.

Remark. The user is responsible for making sure that G is in fact sequential, **dbacl** makes no attempt to verify consistency. The basic 1-grams are taken only from those regular expressions which tag exactly one susbstring, and each 1-gram is counted as many times as some regular expression in $\mathcal{F}_1(G)$ matches it.

9. Logarithmic scores

In this section, we further describe the factorizations of the logarithmic scores into meaningul information-theoretic quantities.

9.1. **Simple case.** As shown in Section 8, the log probability of belonging to the class C_j is approximately proportional to the complexity $\chi_j(s)$, via

$$-\log \mathbb{P}(C_j|s) \approx \chi_j(s) \cdot D(\ell_j(s) \| \widetilde{R}_{C_j}). \tag{21}$$

The Kullback-Leibler divergence measures the difference between the class model $\widetilde{R}_{C_j}$ and the empirical measure $\ell_j(s)$ of the document. Note that both these measures are measures on $\mathcal{F}(G)$, and that is the set where the divergence is computed.

9.2. **Sequential case.** For the sequential case, a similar factorization is possible. Again, we must work on $\mathcal{F}(G)$, but since the approximations depend on m (the highest n-gram order), it is natural to work only on the subset $\mathcal{F}_m(G)$. As in (10), define

$$\widetilde{R}_C(f) = \begin{cases} Z_{C,m}^{-1} \exp\Big[\lambda_f + \sum_{j=1}^{m-1} \lambda_{\tau^j f}\Big] \cdot R_C^{\otimes m}(f) & \text{if } f \in \mathcal{F}_m(C) \\ Z_{C,m}^{-1} R_C^{\otimes m}(f) & \text{if } f \in \mathcal{F}_m(G) \backslash \mathcal{F}_m(C), \end{cases} \tag{22}$$

where $Z_{C,m}$ was computed in (14). Let $\ell(s)(f) = f(s)/\chi_m(s)$, and write $H(\mu) = -\sum_f \mu(f) \log \mu(f)$ to represent the Shannon entropy of the measure μ on $\mathcal{F}_m(G)$. In terms of these quantities, we have

$$-\log \mathbb{P}(C|s) \approx \chi_m(s) \cdot \Big[D(\ell(s) \| \widetilde{R}_C) + H(\ell(s))\Big], \tag{23}$$

which can be verified by expanding the quantities on the left and comparing with the formula in Section 8. The extra Shannon entropy arises here because sequential model probabilities are defined on actual strings s, whereas in the simple case, the probabilities are defined on the equivalence classes of strings (i.e. *types*, Csiszar, 1998).

Given the score, the divergence can therefore be obtained by subtracting the Shannon entropy. Computationally, this requires keeping in memory all the relevant feature counts $f(s)$, as for simple models. A convenient way of calculating $H(\ell_j(s))$ for several categories simultaneously, is using Stirling's approximation $n \log n \approx$

$n - \log\sqrt{2\pi n} + \log n!$, and expanding into a telescoping sum:

$$H(\ell(s)) \approx \frac{1}{\chi(s)}\Big[\sum_{\{f:f(s)>0\}}\Big(1 - \frac{1}{2}\log(2\pi)\Big) + \sum_{k=2}^{f(s)}\Big(1 + \frac{1}{2}\log k(k+1)\Big)\Big] - \log\chi(s).$$

This has the advantage over the exact computation that the computed divergence is overestimated, reducing the incidence of sometimes negative values.

9.3. **Divergence asymptotics.** Let $\varphi(x) = 1 - x + x\log x$, then expanding this function into a Taylor series about $x = 1$, we have

$$D(\ell(s)\|\mathbb{P}_C) = \int \varphi\Big(\frac{d\ell(s)}{d\mathbb{P}_C}\Big)d\mathbb{P}_C \approx \frac{1}{2}\sum_f \left(\frac{f(s)}{\chi(s)\mathbb{P}_C(f)} - 1\right)^2 \mathbb{P}_c(f). \tag{24}$$

Standard asymptotic results suggest that $\chi(s)\cdot D(\ell(s)\|\mathbb{P}_C)$ has an approximate χ^2 distribution, when $\chi(s)$ is big enough, but in fact, the analysis is more delicate, because the number of features f considered is unbounded as $\chi(s)$ tends to infinity. While such an analysis is beyond the scope of this paper, practical experiments nevertheless suggest that $D(\ell(s)\|\mathbb{P}_C)$ has the more general Gamma distribution, $\Gamma(a_C, b_C)$. Making this assumption allows the computation of confidence estimates in the null hypothesis that the law of $\ell(s)$ is $\mathbb{P}_C$. More precisely, let $F_C(x) = \mathbb{P}(\Gamma(a_C, b_C) > x)$ be the tail probability, then the variable $p(s)$ defined by

$$p(s) = 2\cdot\Big[F_C\Big(D(\ell(s)\|\mathbb{P}_C)\Big) \wedge \Big(1 - F_C\Big(D(\ell(s)\|\mathbb{P}_C)\Big)\Big)\Big] \tag{25}$$

is uniformly distributed on $[0,1]$, and satisfies $p(s) \to 0$ if and only if $D(\ell(s)\|\mathbb{P}_C) \to \{0, +\infty\}$. Thus it can be viewed as a *p-value* for the purpose of Fisher's χ^2 test.

References

[1] Berger, A., Della Pietra, S.A., and Della Pietra, V.J. (1996) *A maximum entropy approach to natural language processing*. Computational Linguistics 22(1): 39–71.

[2] Csiszar, I. (1984) Sanov property, generalized I-projection and a conditional limit theorem. Ann. Probab. 12(3): 768–793.

[3] Csiszar, I. (1998) The Method of Types. IEEE Trans. Inform. Theory. 44(6): 2505–2523.

[4] Della Pietra, S.A., Della Pietra, V.J., and Lafferty, J. (1997) *Inducing features of random fields*. IEEE Trans. Pattern Analysis and Machine Intelligence 19(4): 380–393.

[5] MacKay, D. and Peto, L. (1995) *A Hierarchical Dirichlet Language Model*. Natural Language Engineering 1(3): 289–307.

[6] Shannon, C.E. (1948) *A mathematical theory of communication* The Bell System Technical Journal 27: 379–423.

[7] Zhai, C. and Lafferty, J. (2001) A study of smoothing methods for language models applied to ad hoc information retrieval. *SIGIR'01.*

E-mail address: `laird@lbreyer.com`

NAME

dbacl − a digramic Bayesian classifier for text recognition.

SYNOPSIS

dbacl [-0dvnirmMND] [-T *type*] -l *category* [-h *size*] [-H *gsize*] [-x *decim*] [-q *quality*] [-w *max_order*] [-e *deftok*] [-o *online*] [-L *measure*] [-g *regex*]... [FILE]...

dbacl [-vnimNRX] [-h *size*] [-T *type]* -c *category* [-c *category*]... [-f *keep*]... [FILE]...

dbacl -V

DESCRIPTION

dbacl is a Bayesian text document classifier, which uses a maximum entropy (minimum divergence) language model constructed with respect to a digramic reference measure (unknown words are predicted from digrams, i.e. pairs of letters).

If using the **-l** command form, **dbacl** learns a category when given one or more FILE names, which should contain readable ASCII text. If no FILE is given, **dbacl** learns from STDIN. The result is saved in the binary file named *category*.

If using the **-c** command form, **dbacl** attempts to classify the text found in FILE, or STDIN if no FILE is given. Each possible *category* must be given separately, and should be the file name of a previously learned text corpus. When several (more than one) categories are specified, the text classification is performed by computing the Bayesian posterior probability for each model, given the input text, and with a uniform prior distribution on categories.

When only a single category is specified, **dbacl** calculates a score which is the product of the cross entropy and the complexity of the input text (needs the **-n** and **-v** options). A low score (close to zero) indicates a good fit of the *category* model. The cross entropy measures the average compression rate which is achievable, under the given *category*, for the features of the input text.

The **-X** switch can be used when classifying, to ouput a confidence level in the computed scores. This is a percentage which estimates how likely the score is, when assuming that the given category is correct. This is not a substitute for the actual score, and should not be used on its own to decide the best category. A low confidence (e.g. 10% or less) can be interpreted as an "unsure" flag. Not all category models support confidence scores, but "email" categories do.

By default, **dbacl** will classify the input text as a whole. However, when using the **-f** option, **dbacl** can be used to filter each input line separately, printing only those lines which match one or more models identified by *keep*.

Learning and classifying cannot be mixed on the same command invocation. By default, in text mode (**-T** text) non-alphabetic characters in the input are always ignored, as is word case, unless one or more regular expressions are supplied. In email mode (**-T** email) the default tokens are more complicated (currently "adp", see **-e** option).

If the **-w** option is given, the default features consist of all n-grams up to *max_order*, where each first order component is tokenized according to the applicable **-e** option. You can always visualize the tokens by using the **-D** option while learning. If **-w** is not given, **dbacl** assumes *max_order* equals 1. The default feature selection is completely overriden however if the **-g** switch is encountered.

When supplying one or more regular expressions with the **-g** option, only those features of the input text or texts which match *regex* are analysed (with possible overlap between matches), and only those subexpressions of *regex* which are explicitly tagged with parentheses are used as features in the model (if several subexpressions are tagged simultaneously, the resulting feature is the concatenated string of these tagged expressions in order of tag appearance).

For regular exression syntax, see **regex**(7).**For**best**results,**keep**the**number**of**concatenated**expression-s**strictly**below**8. When learning with regular expressions, the **-D** option will print all matched text features, which is very useful to see what is going on.

When saving category files, **dbacl** first writes out a temporary file in the same location, and renames it afterwards. If a problem or crash occurs during learning, the old category file is therefore left untouched.

This ensures that categories can never be corrupted, no matter how many processes try to simultaneously learn or classify, and means that valid categories are available for classification at any time. Category files are created with user and group read permissions only.

Learning roughly works by tweaking probabilities until the training data is least surprising. The amount of tweaking can be selected by the **-q** option. More tweaking means longer learning and better accuracy, but usually the fastest setting is close enough.

When relearning the same category several times, the previously learned probabilities are reused if possible, which can speed up tweaking dramatically (you can use the **-0** option to force relearning from scratch).

Note that classification accuracy depends foremost on the amount and quality of the training samples, and then only on amount of tweaking.

EXIT STATUS

When using the **-l** command form, **dbacl** returns zero. When using the **-c** form, **dbacl** returns a positive integer corresponding to the *category* with the highest posterior probability. In case of a tie, the first most probable category is chosen. If an error occurs, **dbacl** returns zero.

OPTIONS

-0 When learning, prevents weight preloading. Normally, **dbacl** checks if the category file already exists, and if so, tries to use the existing weights as a starting point. This can dramatically speed up learning. If the **-0** (zero) switch is set, then **dbacl** behaves as if no category file already exists. This is mainly useful for testing.

-a Append scores. Every input line is written to STDOUT and the dbacl scores are appended. This is useful for postprocessing with **bayesol**(1). For ease of processing, every original input line is indented by a single space (to distinguish them from the appended scores), and the line with the scores (if **-n** is used) is prefixed with the string "scores ". If a second copy of **dbacl** needs to read this output later, it should be invoked with the **-A** switch.

-d Dump the model parameters to STDOUT. In conjunction with the **-l** option, this produces a human-readable summary of the maximum entropy model. In conjunction with the **-c** option, displays the contribution of each token to the final score. Suppresses all other normal output.

-e Select character class for default (non regex-based) tokenization. By default, tokens are alphabetic strings only. This corresponds to the case when *deftok* is "alpha". Possible values for *deftok* are "alpha", "alnum", "graph", "cef" and "adp". The last two are custom tokenizers intended for email messages. See also **isalpha**(3).

-f Filter each line of input separately, keeping the *category* identified as *keep*. This option should be used repeatedly for each *category* which must be kept. *keep* can be either the *category* file name, or a positive integer representing the required *category* in the same order it appears on the command line.

Output lines are flushed as soon as they are written. If the input file is a pipe or character device, then an attempt is made to use line buffering mode, otherwise the more efficient block buffering is used.

-g Learn only features described by the extended regular expression *regex*. This overrides the default feature selection method (see **-w** option) and learns, for each line of input, only tokens constructed from the concatenation of strings which match the tagged subexpressions within the supplied *regex*. All substrings which match *regex* within a suffix of each input line are treated as features, even if they overlap on the input line.

As an optional convenience, *regex* can include the suffix ||*xyz* which indicates which parenthesized subexpressions should be tagged. In this case, *xyz* should consist exclusively of digits 1 to 9, numbering exactly those subexpressions which should be tagged.

-h Set the size of the hash table to 2^*size* elements. When using the **-l** option, this refers to the total number of features allowed in the maximum entropy model being learned. When using the **-c** option toghether with the **-M** switch and multinomial type categories, this refers to the maximum number of features taken into account during classification. Without the **-M** switch, this option has no effect.

-i Fully internationalized mode. Forces the use of wide characters internally, which is necessary in some locales. This incurs a noticeable performance penalty.

-j Make features case sensitive. Normally, all features are converted to lower case during processing, which reduces storage requirements and improves statistical estimates for small datasets. With this option, the original capitalization is used for each feature. This can improve classification accuracy.

-m Aggressively maps categories into memory and locks them into RAM to prevent swapping, if possible. This is useful when speed is paramount and memory is plentiful, for example when testing the classifier on large datasets. Locking may require relaxing user limits with **ulimit**(1).**Ask**youroursystemadministrator.

-n Print scores for each *category*. Each score is the product of two numbers, the cross entropy and the complexity of the input text under each model. Multiplied together, they represent the log probability that the input resembles the model. To see these numbers separately, use also the **-v** option. In conjunction with the **-f** option, stops filtering but prints each input line prepended with a list of scores for that line.

-q Select *quality* of learning, where *quality* can be 1,2,3,4. Higher values take longer to learn, and should be slightly more accurate. The default *quality* is 1 if the category file doesn't exist or weights cannot be preloaded, and 2 otherwise.

-o When learning, reads/writes partial token counts so they can be reused. Normally, category files are learned from exactly the input data given, and don't contain extraneous information. When this option is in effect, some extra information is saved in the file *online*, after all input was read. This information can be reread the next time learning occurs, to continue where the previous dataset left off. If *online* doesn't exist, it is created. If *online* exists, it is read before learning, and updated afterwards. The file is approximately 3 times bigger (at least) than the learned *category*.

-r Learn the digramic reference model only. Skips the learning of extra features in the text corpus.

-v Verbose mode. When learning, print out details of the computation, when classifying, print out the name of the most probable *category*. In conjunction with the **-n** option, prints the scores as an explicit product of the cross entropy and the complexity.

-w Select default features to be n-grams up to *max_order*. This is incompatible with the **-g** option, which always takes precedence. If no **-w** or **-g** options are given, **dbacl** assumes **-w** 1. Note that n-grams for n greater than 1 can straddle line breaks. If you're learning single lines of text, it's more accurate to use the **-g** option as appropriate.

-x Set decimation probability to 1 - 2ˆ(-*decim*). To reduce memory requirements when learning, some inputs are randomly skipped, and only a few are added to the model. Exact behaviour depends on the applicable **-T** option (default is **-T** "text"). When the type is not "email" (eg "text"), then individual input features are added with probability 2ˆ(-*decim*). When the type is "email", then full input messages are added with probability 2ˆ(-*decim*). Within each such message, all features are used.

-A Expect indented input and scores. With this switch, **dbacl** expects input lines to be indented by a single space character (which is then skipped). Lines starting with any other character are ignored. This is the counterpart to the **-a** switch above. When used together with the **-a** switch, **dbacl** outputs the skipped lines as they are, and reinserts the space at the front of each processed input line.

-D Print debug output. Do not use normally, but can be very useful for displaying the list features picked up while learning.

-H Allow hash table to grow up to a maximum of 2ˆ*gsize* elements during learning. Initial size is given by **-h** option.

-L Select the digramic reference measure for character transitions. The *measure* can be one of "uniform", "dirichlet" or "maxent". Default is "uniform".

-M Force multinomial calculations. When learning, forces the model features to be treated multinomially. When classifying, corrects entropy scores to reflect multinomial probabilities (only applicable to multinomial type models, if present). Scores will always be lower, because the ordering of features is lost.

-N Print posterior probabilities for each *category*. This assumes the supplied categories form an exhaustive list of possibilities. In conjunction with the **-f** option, stops filtering but prints each input line prepended with a summary of the posterior distribution for that line.

-R Include an extra category for purely random text. The category is called "random". Only makes sense when using the **-c** option.

-T Specify nonstandard text format. By default, **dbacl** assumes that the input text is a purely ASCII text file. This corresponds to the case when *type* is "text".

There are several types and subtypes which can be used to clean the input text of extraneous tokens before actual learning or classifying takes place. Each (sub)type you wish to use must be indicated with a separate **-T** option on the command line, and automatically implies the corresponding type.

The "text" type is for unstructured plain text. No cleanup is performed. This is the default if no types are given on the command line.

The "email" type is for mbox format input files or single RFC822 emails. Headers are recognized and most are skipped. To include extra RFC822 standard headers (except for tracking headers), use the "email:headers" subtype. To name all headers in the email, use the "email:xheaders" subtype. To skip all headers, except the subject, use "email:noheaders". To scan binary attachments for strings, use the "email:atts" subtype.

When the "email" type is in effect, HTML markup is automatically removed from text attachments except text/plain attachments. To also remove HTML markup from plain text attachments, use "email:noplain". To prevent HTML markup removal in all text attachments, use "email:plain".

The "html" type is for removing HTML markup (between <html> and </html> tags) and surrounding text. Note that if the "email" type is enabled, then "html" is automatically enabled for compatible message attachments only.

The "xml" type is like "html", but doesn't honour <html> and </html>, and doesn't interpret tags (so this should be more properly called "angle markup" removal, and has nothing to do with actual XML semantics).

When "html" is enabled, most markup attributes are lost (for values of 'most' close to 'all'). The "html:links" subtype forces link urls (network location only) to be parsed and learned, which would otherwise be ignored. The "html:alt" subtype forces parsing of alternative text in ALT attributes and various other tags. The "html:scripts" subtype forces parsing of scripts, "html:styles" forces parsing of styles, "html:forms" forces parsing of form values, while "html:comments" forces parsing of HTML comments.

-V Print the program version number and exit.

-X Print the confidence in the score calculated for each *category*, when used together with the **-n**or**N**switch. The confidence is an estimate of the typicality of the score, assuming the null hypothesis that the given category is correct. When used with the **-v** switch alone, factorizes the score as the empirical divergence plus the shannon entropy, multiplied by complexity, in that order. The **-X** switch is not supported in all possible models, and displays a percentage of "101" if it can't be calculated.

USAGE

To create two category files in the current directory from two ASCII text files named Mark_Twain.txt and William_Shakespeare.txt respectively, type:

```
% dbacl -l twain Mark_Twain.txt
% dbacl -l shake William_Shakespeare.txt
```

Now you can classify input text, for example:

```
% echo "howdy" | dbacl -v -c twain -c shake
twain
% echo "to be or not to be" | dbacl -v -c twain -c shake
shake
```

Note that the **-v** option is necessary, otherwise **dbacl** does not print anything. The return value is 1 in the first case, 2 in the second.

Suppose a file document.txt contains English text lines interspersed with noise lines. To filter out the noise lines from the English lines, assuming you have an existing category shake say, type:

```
% dbacl -c shake -f shake -R document.txt > document.txt_eng
% dbacl -c shake -f random -R document.txt > document.txt_rnd
```

Note that the quality of the results will vary depending on how well the categories shake and random represent each input line. It is sometimes useful to see the posterior probabilities for each line without filtering:

```
% dbacl -c shake -f shake -RN document.txt > document.txt_probs
```

You can now postprocess the posterior probabilities for each line of text with another script, to replicate an arbitrary Bayesian decision rule of your choice.

In the special case of exactly two categories, the optimal Bayesian decision procedure can be implemented for documents as follows: let *p1* be the prior probability that the input text is classified as *category1*. Consequently, the prior probability of classifying as *category2* is 1 - *p1*. Let *u12* be the cost of misclassifying a *category1* input text as belonging to *category2* and vice versa for *u21*. We assume there is no cost for classifying correctly. Then the following command implements the optimal Bayesian decision:

```
% dbacl -n -c category1 -c category2 | awk ’{ if($2 * p1 * u12 > $4 * (1 - p1) * u21) { print $1; } else {
        print $3; } }’
```

dbacl can also be used in conjunction with **procmail**(1) to implement a simple Bayesian email classification system. Assume that incoming mail should be automatically delivered to one of three mail folders located in $MAILDIR and named *work*, *personal*, and *spam*. Initially, these must be created and filled with appropriate sample emails. A **crontab**(1) file can be used to learn the three categories once a day, e.g.

```
CATS=$HOME/.dbacl
5  0 * * * dbacl -T email -l $CATS/work $MAILDIR/work
10 0 * * * dbacl -T email -l $CATS/personal $MAILDIR/personal
15 0 * * * dbacl -T email -l $CATS/spam $MAILDIR/spam
```

To automatically deliver each incoming email into the appropriate folder, the following **procmailrc**(5) recipe fragment could be used:

```
CATS=$HOME/.dbacl

# run the spam classifier
:0 c
YAY=| dbacl -vT email -c $CATS/work -c $CATS/personal -c $CATS/spam

# send to the appropriate mailbox
:0:
* ? test -n "$YAY"
$MAILDIR/$YAY

:0:
$DEFAULT
```

Sometimes, **dbacl** will send the email to the wrong mailbox. In that case, the misclassified message should be removed from its wrong destination and placed in the correct mailbox. If it is left in the wrong category,

dbacl will learn the wrong corpus statistics.

The default text features (tokens) read by **dbacl** are purely alphabetic strings, which minimizes memory requirements but can be unrealistic in some cases. To construct models based on alphanumeric tokens, use the **-e** switch. The example below also uses the optional **-D** switch, which prints a list of actual tokens found in the document:

% dbacl -e alnum -D -l twain Mark_Twain.txt | less

It is also possible to override the default feature selection method used to learn the category model by means of regular expressions. For example, the following duplicates the default feature selection method in the C locale, while being much slower:

% dbacl -l twain -g '^([[:alpha:]]+)' -g '[^[:alpha:]]([[:alpha:]]+)' Mark_Twain.txt

The category twain which is obtained depends only on single alphabetic words in the text file Mark_Twain.txt (and computed digram statistics for prediction). For a second example, the following command builds a smoothed Markovian (word bigram) model which depends on pairs of consecutive words within each line (but pairs cannot straddle a line break):

% dbacl -l twain2 -g '(^|[^[:alpha:]])([[:alpha:]]+)||2' -g
'(^|[^[:alpha:]])([[:alpha:]]+)[^[:alpha:]]+([[:alpha:]]+)||23' Mark_Twain.txt

More general, line based, n-gram models of all orders can be built in a similar way. To construct paragraph based models, you should reformat the input corpora with **awk**(1) or **sed**(1) to obtain one paragraph per line. Line size is limited by available memory, but note that regex performance will degrade quickly for long lines.

Handy tip: Create the hidden directory $HOME/.dbacl, and set DBACL_PATH=$HOME/.dbacl in your shell initialization script. Then you can refer to categories without writing full paths.

ENVIRONMENT

DBACL_PATH
When this variable is set, its value is prepended to every *category* filename which doesn't start with a '/' or a '.'.

SIGNALS

INT If this signal is caught, **dbacl** simply exits without doing any cleanup or other operations. This signal can often be sent by pressing Ctrl-C on the keyboard. See **stty**(1).

HUP, QUIT, TERM
If one of these signals is caught, **dbacl** stops reading input and continues its operation as if no more input was available. This is a way of quitting gracefully, but note that in learning mode, a category file will be written based on the incomplete input. The QUIT signal can often be sent by pressing Ctrl- on the keyboard. See **stty**(1).

USR1 If this signal is caught, **dbacl** reloads the current categories at the earliest feasible opportunity. This is not normally useful at all, but might be in special cases, such as if the **-f** switch is invoked together with input from a long running pipe.

NOTES

dbacl generated category files are in binary format, and may or may not be portable to systems using a different byte order architecture (this depends on how **dbacl** was compiled). The **-V** switch prints out whether categories are portable, or else you can just experiment.

dbacl does not recognize functionally equivalent regular expressions, and in this case duplicate features will be counted several times.

With every learned category, the command line options that were used are saved. When classifying, make sure that every relevant category was learned with the same set of options (regexes are allowed to differ), otherwise behaviour is undefined.

Specific documentation about the design of **dbacl** and the statistical models that it uses can be found in /home/laird/local/share/dbacl/doc/dbacl.ps . For a basic overview of text classification using **dbacl**, see

/home/laird/local/share/dbacl/doc/tutorial.html . A companion tutorial geared towards email filtering is /home/laird/local/share/dbacl/doc/email.html .

If you get many digitization warnings, then you are trying to learn too much data at once, or your model is too complex. **dbacl** is compiled to save memory by digitizing final weights, but you can disable digitization by editing dbacl.h and recompiling.

dbacl offers several built-in tokenizers (see **-e** switch) with more to come in future versions, as the author invents them. While the default tokenizer may evolve, no tokenizer should ever be removed, so that you can always simulate previous **dbacl** behaviour subject to bug fixes and architectural changes.

The confidence estimates obtained through the **-X** switch are underestimates, ie are more conservative than they should be.

BUGS

"Ya know, some day scientists are gonna invent something that will outsmart a rabbit." (Robot Rabbit, 1953)

SOURCE

The source code for the latest version of this program is available at the following locations:

http://www.lbreyer.com/gpl.html
http://dbacl.sourceforge.net

AUTHOR

Laird A. Breyer <laird@lbreyer.com>

SEE ALSO

awk(1), **bayesol**(1), **crontab**(1), **less**(1), **mailcross**(1), **mailfoot**(1), **mailinspect**(1), **mailtoe**(1), **procmailex**(5), **regex**(7), **stty**(1), **sed**(1)