Using Naive Bayes to Detect Spammy Names in Social Networks

David Mandell Freeman
LinkedIn Corporation
2029 Stierlin Ct.
Mountain View, CA 94043 USA
dfreeman@linkedin.com

ABSTRACT

Many social networks are predicated on the assumption that a member's online information reflects his or her real identity. In such networks, members who fill their name fields with fictitious identities, company names, phone numbers, or just gibberish are violating the terms of service, polluting search results, and degrading the value of the site to real members. Finding and removing these accounts on the basis of their spammy names can both improve the site experience for real members and prevent further abusive activity.

In this paper we describe a set of features that can be used by a Naive Bayes classifier to find accounts whose names do not represent real people. The model can detect both automated and human abusers and can be used at registration time, before other signals such as social graph or clickstream history are present. We use member data from LinkedIn to train and validate our model and to choose parameters. Our best-scoring model achieves AUC 0.85 on a sequestered test set.

We ran the algorithm on live LinkedIn data for one month in parallel with our previous name scoring algorithm based on regular expressions. The false positive rate of our new algorithm (3.3%) was less than half that of the previous algorithm (7.0%).

When the algorithm is run on email usernames as well as user-entered first and last names, it provides an effective way to catch not only bad human actors but also bots that have poor name and email generation algorithms.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval — Spam; I.2.6 [**Artificial Intelligence**]: Learning

Keywords

Social networks, spam detection, Naive Bayes classifier

AISec'13, November 4, 2013, Berlin, Germany.

ACM 978-1-4503-2488-5/13/11 ...$15.00.
http://dx.doi.org/10.1145/2517312.2517314

1. INTRODUCTION

One of the fundamental characteristics of many social networks is that a user's online identity is assumed to be an accurate reflection of his or her real identity. For example, the terms of service of Facebook [13], LinkedIn [20], and Google+ [15] all require users to identify themselves by their real names. However, many users still sign up for these social networks with fake names. Often these users intend to engage in abusive activity, such as sending spam messages or posting malicious links. These users may use automation to create accounts, in which case fake names are generated from a predefined dictionary or even as random strings of characters.

In addition to the dedicated spammers, many users with no intention of abuse will fill in their personal information with something other than (or in addition to) their real name. Some users simply cannot be bothered to take the time to enter their personal information, and fill in name fields with strings such as "asdf," while others may choose to mask their identities out of privacy concerns. Still others may use the name fields to enter job titles, phone numbers, email addresses, or other non-name information, either out of hopes that this information will be surfaced more easily in searches or out of misunderstanding of the field labels. These accounts, even if not actively abusing the site, are still undesirable, as they may pollute search results and degrade the value of the site to members who abide by the terms of service.

Many previous authors have considered the problem of detecting spam and fake accounts in social networks, using signals such as clickstream patterns [30, 29], message-sending activity and content [3, 14], and properties of the social graph [8, 7, 6]. However, to our knowledge none have considered the problem of detecting fake accounts *from name information only.*

An approach using only name data has several appealing features:

- It can be used to detect both automated and human abusers and both malicious and non-malicous fakes.
- It can be used at registration time, when other signals such as clickstream history and connections graph are not present.
- It can detect fakes preemptively rather than waiting for the first round of abuse to occur.

Furthermore, finding fake names can be used as a complementary strategy to existing spam-detection methods and

can be used as part of a co-training process [5] to create a broad model for fake accounts.

1.1 Our contribution

We propose a robust, language-agnostic algorithm for spam detection in user-entered names and other very small texts. Specifically, we describe a set of features that can be extracted from such texts and used in a Naive Bayes classifier applied to these texts, as well as some algorithmic tweaks that improve the classifer's performance. We demonstrate the effectiveness of our classifier by running the algorithm on LinkedIn members' name data, including both a labeled validation set and unlabeled data from production, and computing several performance metrics. Implementing the algorithm on production data cut the false positive rate by more than half as compared with LinkedIn's previous method of name spam detection. We also show that running the algorithm on email usernames can supplement the name spam detection by correctly classifying some borderline cases.

1.2 Our Approach

We build our name-spam detection algorithm using a Naive Bayes classifier. This classifier has a long and successful history of dealing with the problem of email spam [25, 2, 16, 26, 27, 1]. However, all proposed uses of Naive Bayes for spam classification use individual words or groups of words as features. Using individual names as features would not be an effective means of detecting name spam. For example, if each name consisted of a single word, the Naive Bayes classifier would be equivalent to determining whether the percentage of fake accounts with a given name was above or below some predetermined threshold. Even more problematically, a whole-name-based classifier would be powerless to classify a name it had not seen before; all unique names would have to share the same classification, a significant problem on a global social network such as LinkedIn, where more than one third of all accounts have unique names [18].

Instead of using whole words, we use as our base feature set *n-grams of letters* that comprise these words. Specifically, for various small values of n we extract from an m-character name the $(m - n + 1)$ substrings of n consecutive characters. For example, with $n = 3$ the name *David* breaks into the three n-grams (Dav, avi, vid). We then augment the feature set by adding phantom "start-of-word" and "end-of-word" characters (represented by `\^` and `\$`, respectively) to each name and incorporating these into our n-grams; under this tranformation the 3-grams for *David* are $(\texttt{\^{}}Da, dav, avi, vid, id\texttt{\$})$.

1.3 Missing Features

It may occur that a name we wish to classify contains an n-gram that does not appear in our training set. There are two standard approaches to dealing with missing features in Naive Bayes: to ignore them, or to assign a value to a "missing feature" feature. We consider both options. To assign a value to the "missing feature" feature we borrow a technique from natural language processing:we randomly divide the training set in two and add up the category counts for features that appear in only one half of the set. Contrary to the results of Kohavi, Becker, and Sommerfield [17], we find that adding the "missing feature" feature significantly improves our results. More generally, we can apply the "missing feature" feature to any feature with fewer than c instances in the training set, with the optimal value of c determined experimentally on a validation set.

To improve further our treatment of missing features, we observe that for an n-gram that does not appear in the training set it may be the case the the two component $(n-1)$-grams do appear in the training set. In this case we can replace the missing n-gram feature with the two $(n-1)$-gram features; if one or both of these features are missing we can iterate recursively, up to the point where we are considering 1-grams. We find that this treatment gives a significant performance boost.

1.4 Experimental Results

We trained our classifier on a representative sample of 60 million LinkedIn accounts, with spam labels provided by the LinkedIn Security team. After optimizing parameters on a validation set of approximately 100,000 accounts, we measured the performance of two different versions of our classifier on a held-out test set of approximately 100,000 accounts. Both the validation and test sets were designed to consist of roughly half spam accounts.

We tested two versions of our algorithm: a "full" version using 5-grams and a "lightweight" version using 3-grams that uses less than 10% as much memory as the full version. After optimizing our algorithm's parameters, for the full algorithm we obtained an AUC of 0.85 and a max $F_{1/8}$-score of 0.92. For the lightweight version we obtained an AUC of 0.80 and a max $F_{1/8}$-score of 0.93. Further statistics can be found in Section 5. We consider $F_{1/8}$-score instead of the standard F_1-score because the cost of a false positive (blocking a good member) is significantly higher than the cost of a false negative (leaving a fake account alone).

We also used our algorithm to find several hundred false negatives (i.e., spam accounts that had not been caught) in our test set.

Finally, we ran our lightweight algorithm on the primary email addresses associated with accounts in our training and test sets, and found a noticeable improvement when the name and email features were combined.

2. MULTINOMIAL NAIVE BAYES

We begin by reviewing the Naive Bayes classifier [12, 23]. The task of this classifier is to approximate an unknown function $f\colon F \to L$, where $F = F_1 \times \cdots \times F_m$ is the feature space and L is the label space. In our application L is the binary set $\{0, 1\}$. If $X = (X_1, \ldots, X_m)$ is a random variable taking values in F and Y is a random variable taking values in L, then $f(x)$ can be estimated by computing $p_y = p(Y = y \mid X = \vec{x})$ and outputting the value of y for which p_y is maximized.

By Bayes' theorem and the law of total probability we have

$$p(Y = y \mid X = \vec{x}) = \frac{p(Y = y) \cdot p(X = \vec{x} \mid Y = y)}{\sum_y p(Y = y) \cdot p(X = \vec{x} \mid Y = y)}. \tag{2.1}$$

Now suppose we are trying to classify a text document that is represented as an ordered sequence of words $(x_1, \ldots, x_m)$; i.e., the feature F_i represents the word in the ith position. The *multinomial Naive Bayes* classifier makes the assumption that each word is generated independently from a multi-

nomial distribution.[1] Under this assumption, if we let θ_{wy} be the probability that the word w appears in class y, then we have

$$p(X = \vec{x} \mid Y = y) = \frac{(\sum f_w)!}{\prod_w f_w!} \prod_w (\theta_{wy})^{f_w}, \qquad (2.2)$$

where f_w denotes the number of instances of word w in the vector $(x_1, \ldots, x_m)$ and the products are taken over all words w in the vocabulary.

If the classification is binary and Y takes values 0 and 1, then we can divide the numerator and denominator of (2.1) by the terms with $y = 0$ to get

$$s(\vec{x}) := p(Y = 1 \mid X = \vec{x}) = \frac{\frac{p(Y=1)}{p(Y=0)} e^{R(\vec{x})}}{1 + \frac{p(Y=1)}{p(Y=0)} e^{R(\vec{x})}}, \qquad (2.3)$$

where

$$R(\vec{x}) = \log\left(\frac{p(X = \vec{x} \mid Y = 1)}{p(X = \vec{x} \mid Y = 0)}\right)$$

is the log of the conditional probability ratio for the feature vector $\vec{x}$. Substituting (2.2) into the formula for $R(\vec{x})$ and taking logarithms gives

$$R(\vec{x}) = \sum_w f_w \cdot \log\left(\frac{\theta_{w1}}{\theta_{w0}}\right). \qquad (2.4)$$

(Observe that in a short document almost all of the f_w will be equal to zero.)

Performing the classification task now comes down to computing estimates for the values $p(Y = y)$ (the *class priors*) and the parameter estimates θ_{wy}. We estimate θ_{wy} from training data as

$$\theta_{wy} = \frac{N_{w,y} + \alpha_{w,y}}{N_y + \sum_w \alpha_{w,y}}, \qquad (2.5)$$

where $N_{w,y}$ indicates the number of instances of word w occurring in class y in the training set and N_y is the sum of the $N_{w,y}$ over all features. The parameter $\alpha_{w,y}$ is a *smoothing parameter* and corresponds to adding $\alpha_{w,y}$ imagined instances of feature w in category y to the training data. Setting a nonzero smoothing parameter prevents a feature with examples in only one class from forcing the probability estimate $s(\vec{x})$ to be 0 or 1. Standard *Laplace smoothing* sets $\alpha_{w,y} = 1$ for all w and y; other techniques are to set $\alpha_{w,y}$ to be a small real number δ independent of w or to set $\alpha_{w,y} = \delta/N_{w,y}$. This last method is called *interpolated smoothing* by Chen and Goodman [9].

The Naive Bayes algorithm has demonstrated great success in classification problems (see e.g., [11, 24, 21]) even though the value $s(\vec{x})$ of (2.3) is often highly inaccurate as an estimate of probability [4, 31, 10]. However, if we interpret $s(\vec{x})$ as a score for the feature vector $\vec{x}$, we can then use a validation set to choose a cutoff point for classifying that depends on the relative cost of misclassifying in each class. (See Section 5.1 for further discussion.)

Implementation of the algorithm, once a suitable training set has been obtained, is straightforward. As long as the number of features in the training set is no more than a few million, the data required for classifying new instances can be kept in main memory on an ordinary PC or run on a distributed system such as Hadoop. Thus once the data is loaded, the algorithm can quickly classify many instances. We observe that computing $R(\vec{x})$ rather than $e^{R(\vec{x})}$ speeds up computation and eliminates underflow errors caused by multiplying together many small values.

[1]We choose this model over others such as *multivariate Bernoulli* and *multivariate Gauss* [22] because results from the literature [26, 21] suggest that this model typically performs best on text classification.

3. THE BASIC ALGORITHM

3.1 N-grams of Letters

In typical email spam classifiers using Naive Bayes (e.g., [25, 26, 1]) the features w consist of words or phrases in a document, with separation indicated by whitespace characters or punctuation. For short texts such as names these features are clearly inadequate — most names will only have two features! Thus instead of entire words, we use *n-grams of letters* as our main features. Formally, we have the following:

Definition 3.1. *Let $W = c_1 c_2 \cdots c_m$ be an m-character string. We define n-grams(W) to be the $(m - n + 1)$ substrings of n consecutive characters in W:*

$$n\text{-grams}(W) = (c_1 c_2 \cdots c_n, c_2 c_3 \cdots c_{n+1}, \ldots, c_{m-n+1} c_{m-n+2} \cdots c_m).$$

For example, with $n = 3$ the name "David" breaks into the three n-grams (Dav, avi, vid).

Our choice of overlapping n-grams as features blatantly violates the Naive Bayes assumption that the features used in classification are independent. However, as observed by numerous previous authors (e.g. [10]), the classifier may still have good performance despite the dependencies among features; see Section 5.2 for further discussion.

Since names on LinkedIn consist of distinct first name and last name fields (neither of which is allowed to be empty), we have a choice in labeling our features: we can consider an n-gram coming from a first name to be either identical to or distinct from an n-gram coming from a last name. We considered both approaches in our initial trials and came to the conclusion that using distinct labels for first and last name n-grams performs best for all values of n. (See Figure 1.)

3.2 Dataset and Metrics

We trained our classifier on a sample of roughly 60 million LinkedIn accounts that were either (a) in good standing as of June 14, 2013 or (b) had been flagged by the LinkedIn Security team as fake and/or abusive at some point before that date. We labeled accounts in (a) as good and those in (b) as spam. We sampled accounts not in our training set to create a validation/test set of roughly 200,000 accounts. Since the incidence of spam accounts in our data set is very low, we biased our validation set to contain roughly equal numbers of spam and non-spam accounts. This bias allows us both to test the robustness of our algorithm to variations in the types of names on spam accounts and to speed up our testing process.

We began with of a preprocessing step that computed tables of n-gram frequencies from raw name data for $n = 1, \ldots, 5$. Our algorithms are fully Unicode-aware, which allows us to classify international names but leads to a large number of n-grams. To reduce memory usage, we ignored n-grams that had only a single instance in the training set. Table 1 shows the number of distinct n-grams for each value

	first/last distinct		first/last combined	
n	n-grams	memory	n-grams	memory
1	15,598	25 MB	8,235	24 MB
2	136,952	52 MB	86,224	45 MB
3	321,273	110 MB	252,626	108 MB
4	1,177,675	354 MB	799,985	335 MB
5	3,252,407	974 MB	2,289,191	803 MB

Table 1: Number of distinct n-grams in our training set for $n = 1, \ldots, 5$ and amount of memory required to store the tables in our Python implementation. Data on the left considers the sets of n-grams from first and last names independently, while data on the right considers the sets cumulatively.

of n and the corresponding amount of memory required to store the table of tuples $(w, \log(\theta_{w1}/\theta_{w0}))$ for all n-grams w.

3.3 Initial results

In our experiments we use the probability estimate $s(\vec{x})$ of (2.3) to assign to each test case a real-number score between 0 and 1. When comparing different models we use area under the ROC curve (AUC) as our primary metric to compare the outcomes of two trials. We favor this metric because it does not require us to make a decision on a threshold for spam labeling and/or the relative weights of false positives and false negatives. In addition, AUC is insensitive to bias in our validation set. Specifically, since our validation set is constructed as a sample of some fraction x of spam accounts and a different fraction y of non-spam accounts, changing the values of x and y should have no impact on the expected value of AUC.

In testing variants and parameters, we use a validation set of 97,611 accounts (48,461 spam and 49,150 not spam). Figure 1 shows results for our initial scoring, with values of n ranging from 1 to 5. We ran the classifier using two different feature sets: one where first name and last name n-gram data are combined, and one where they are distinct. We see that using distinct features is superior for all values of n. All subsequent tests were conducted using distinct features.

It appears from the data that increasing n improves the classification only up to a certain point — the results for $n = 5$ are worse than those for $n = 4$. We conjecture that the number of 5-grams that appear in the validation set but not in the training set offsets the improvement from increased granularity. See Section 4.2 for further discussion and ways to ameliorate the situation.

We also considered various smoothing parameters: Laplace smoothing $\alpha_{w,y} = \delta$ and interpolated smoothing $\alpha_{w,y} = \delta/N_{w,y}$, each for $\delta \in (0.01, 0.1, 1, 10, 100)$. To our surprise, and contrary to the results of Kohavi, Becker, and Sommerfield [17], we found that a constant $\alpha_{w,y} = 0.1$ performs best in our experiments, slightly outperforming interpolated smoothing with $\delta = 10$ (see Figure 2). We use this smoothing parameter for all subsequent experiments.

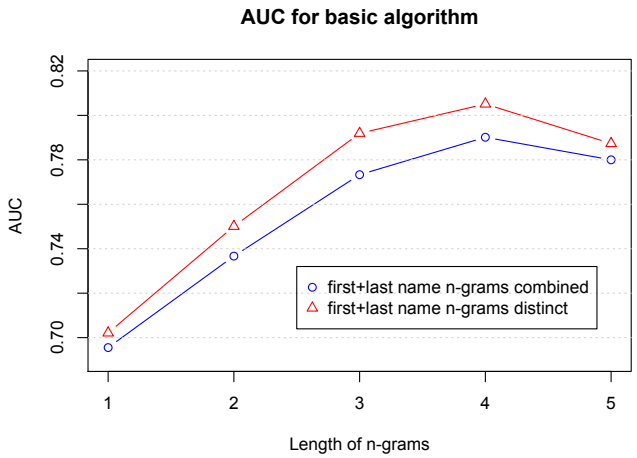

Figure 1: AUC for basic algorithm, with first and last name features distinct or combined.

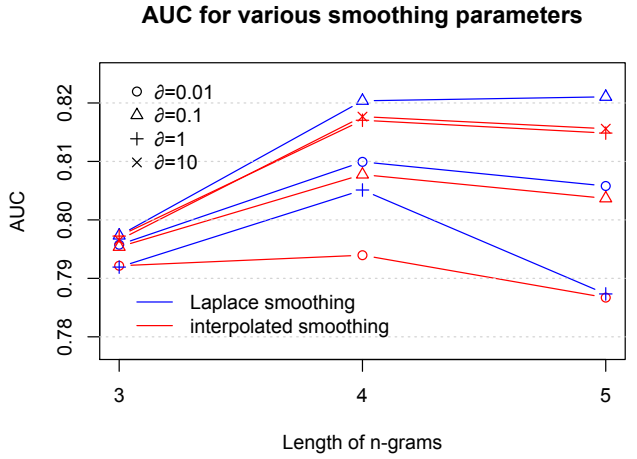

Figure 2: AUC for basic algorithm with various smoothing parameters. We omit the plots for $n = 1$ and $n = 2$ since all parameters perform approximately the same. We omit the plots for $\delta = 100$ since the results are significantly worse than those plotted.

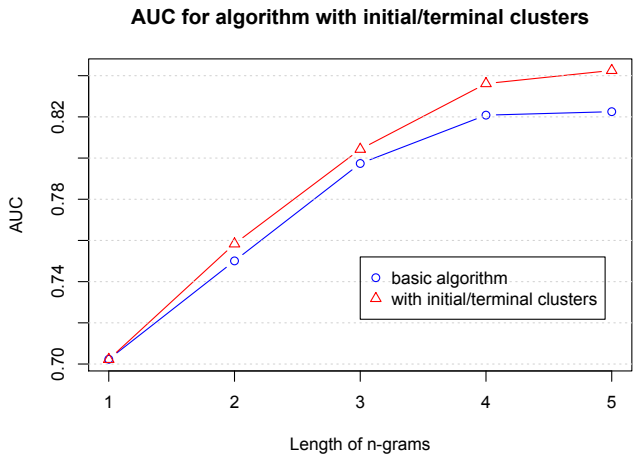

Figure 3: AUC for basic algorithm, with and without initial and terminal clusters.

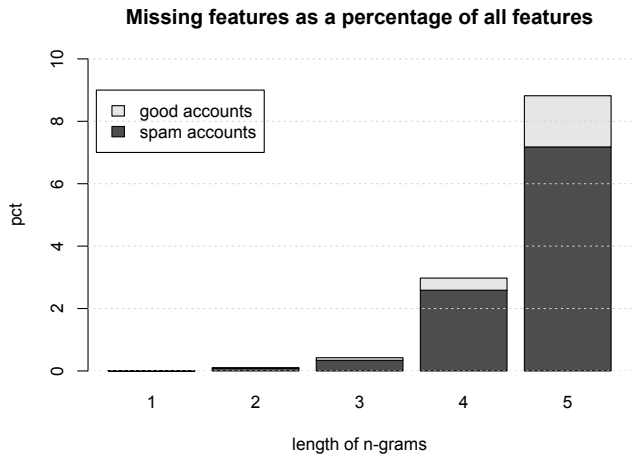

Figure 4: Proportion of missing features in validation set, broken down by spam label.

4. IMPROVING THE FEATURE SET

4.1 Initial and terminal clusters

To augment the feature set, we observe that certain n-grams may be more or less likely to correspond to fake accounts if they start or end a name. For example, in our training set the proportion if spam accounts with the 2-gram zz appearing at the start of a name is 13 times the proportion of spam accounts with zz appearing anywhere in the name. We thus augment the feature set by adding phantom "start-of-word" and "end-of-word" characters to each name before splitting it into n-grams.

Formally, we add two new symbols \^ and \$ to our dictionary of letters. For an m-character name $W = c_1c_2\cdots c_m$ we set $c_0 =$\^, $c_{m+1} =$\$, $W' = c_0c_1\cdots c_mc_{m+1}$, and compute n-grams(W') using the definition of (3.1). (Clearly this only adds information if $n > 1$.) With these phantom characters added, the name *David* breaks into 3-grams as (\^*Da*, *dav*, *avi*, *vid*, *id*\$).

We recomputed our training data with initial and terminal clusters; the results appear in Figure 3. The magnitude of the improvement increases with n., and for $n = 5$ the improvement is large enough to make $n = 5$ better than $n = 4$ (AUC 0.843 for $n = 5$ vs. 0.836 for $n = 4$).

4.2 Including missing features

It may happen that a name in our validation set contains an n-gram w that does not appear in our training set. There are two standard ways of dealing with this occurrence [17]: either ignore the feature w entirely (which is equivalent to assuming that $\theta_{w0} = \theta_{w1}$, i.e., the probabilities of w appearing in each class are identical), or estimate values for θ_{wy}. Until this point we have chosen the former; we now investigate the latter.

Intuitively, we expect an n-gram that is so rare as to not appear in our training set to be more likely to belong to a spammer than an n-gram selected at random. Indeed, this is what the data from the validation set show: the proportion of missing features that belong to spam accounts ranges between 76% and 87% as n varies. The data also supports our hypothesis that the reason $n = 5$ does not outperform $n = 4$ is that many more features are missing (8.8% vs. 3.0%), and thus were ignored in our initial analysis. The proportions of missing features for each n can be seen in Figure 4.

To estimate parameters for missing features, we take the approach of considering all missing features (for a fixed value of n) as a single n-gram $\tilde{w}$. We compute $\theta_{\tilde{w}y}$ by splitting the data in half and labeling n-grams that appear only in one half as "missing." More precisely, we do the following:

Algorithm 4.1. Input: a set χ of training samples W, labeled with classes y, and a fixed integer n. Output: parameter estimates for missing values.

1. Randomly assign each training sample $W \in \chi$ to one of two sets, $\mathcal{A}$ or $\mathcal{B}$, each with probability 1/2.

2. Define $g(\mathcal{A}) = \bigcup_{W\in\mathcal{A}} n\text{-grams}(W)$ and $g(\mathcal{B}) = \bigcup_{W\in\mathcal{B}} n\text{-grams}(W)$.

3. For each class y, define
$$N_{\tilde{w},y} = \sum_{\substack{w\in g(\mathcal{A})\\ w\notin g(\mathcal{B})}} N_{w,y} + \sum_{\substack{w\in g(\mathcal{B})\\ w\notin g(\mathcal{A})}} N_{w,y}$$

4. Compute $\theta_{\tilde{w}y}$ using (2.5).

When we include this "missing feature" feature, the AUC values for $n \le 4$ do not change noticeably, but the AUC for $n = 5$ increases from 0.843 to 0.849.

4.3 Recursively iterating the classifier

The above method incorporates a "one-size-fits-all" approach to estimating parameters for missing features. We can do better by observing that even if a given n-gram is not in our training set, the two component $(n-1)$-grams may appear in the training set. If one or both of these $(n-1)$-grams are missing, we can go to $(n-2)$-grams, and so on. We thus propose the following recursive definition for estimating the parameter θ_{wy}:

Definition 4.2. *Let w be a character string of length n. Let $\theta_{\tilde{w}y}$ be a parameter estimate for missing values of 1-grams (as computed by Algorithm 4.1 wth $n = 1$), and let $\alpha_{w,y}$ be*

a smoothing parameter. We define θ_{wy} as follows:

$$\theta_{wy} = \begin{cases} \dfrac{N_{w,y} + \alpha_{w,y}}{N_y + \sum_w \alpha_{w,y}} & \text{if } N_{w,y} > 0, \\ \displaystyle\prod_{w' \in (n-1)\text{-grams}(w)} \theta_{w'y} & \text{if } N_{wy} = 0 \text{ and } n > 1, \\ \theta_{\tilde{w}y} & \text{if } N_{wy} = 0 \text{ and } n = 1, \end{cases}$$

We observe that under this formulation, if two consecutive n-grams are missing, then the overlapping $(n-1)$-gram appears twice in the product θ_{wy}. At the extreme, if a particular character does not appear in the training set, the "missing feature" parameter $\theta_{\tilde{w}y}$ is incorporated 2^{n-1} times in the product θ_{wy}.

When we use the formula of Definition 4.2 to compute the parameters θ_{wy} for our classifier, the results for $n \leq 4$ change negligibly, but the AUC for $n = 5$ improves from 0.849 to 0.854.

5. EVALUATING PERFORMANCE

5.1 Results on Test Set

We ran our algorithm on a sequestered test set of 97,965 accounts, of which 48,726 were labeled as spam and 49,239 were labeled as good. We tried two different sets of parameters: a "full" version using 5-grams that is informed by the above discussion, and a "lightweight" version using 3-grams that is designed to be faster and use less memory. In light of the above discussion we used the following parameters:

	Full Version	Lightweight Version
n	5	3
$\alpha_{w,y}$	0.1	0.1
initial/terminal	yes	yes
missing n-grams	$(n-1)$-grams (Definition 4.2)	fixed estimate (Algorithm 4.1)
AUC	**0.852**	**0.803**

Figure 5 gives histograms of scores for the two runs. As expected for the Naive Bayes classifier, scores cluster very close to 0 or 1. We see by comparing the histograms that scores computed by the lightweight algorithm are weighted more towards the high end than those computed by the full algorithm. For example, we have the following statistics for scores below 0.05 and above 0.95 (the leftmost and rightmost bars in the histograms, respectively):

Score+Label	Full	Lightweight
score < 0.05, spam	16191 (33%)	12525 (26%)
score < 0.05, good	928 (1.9%)	433 (0.9%)
score > 0.95, spam	19259 (40%)	25844 (53%)
score > 0.95, good	44966 (91%)	45753 (93%)

(The percentages indicate the proportion of all samples with that label appearing in the given histogram bucket.) As a result, the lightweight version finds fewer false positive spam names and more false negatives.

Precision and Recall.

In our setup, for a given score cutoff c, the precision function prec(c) denotes the fraction of correct labels out of all

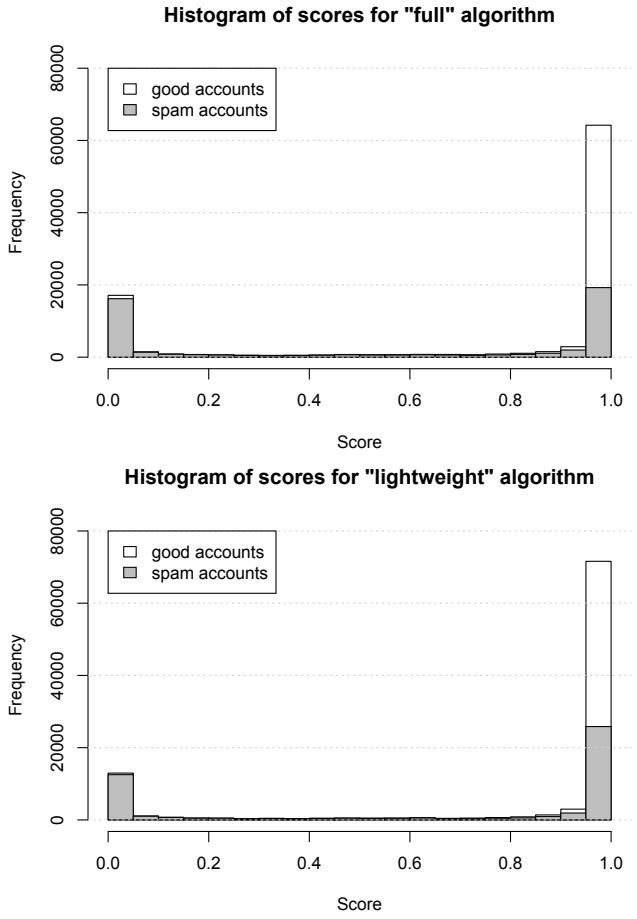

Figure 5: Histograms of scores for full and lightweight models, broken down by spam label.

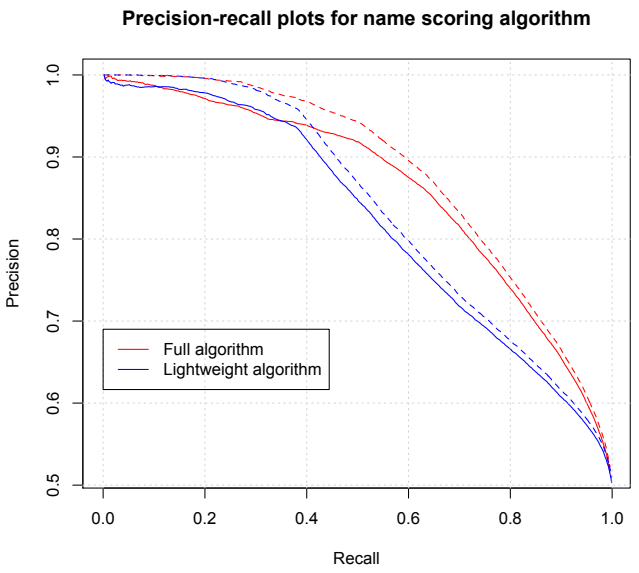

Figure 6: Precision–recall curves for full and lightweight models. Solid lines indicate original test set; dashed lines indicate test set corrected for mislabeled samples.

accounts labeled as spam by our classifier, while the recall function $\mathrm{rec}(c)$ denotes the fraction of correct labels out of all spam accounts. The solid lines in Figure 6 give precision–recall curves for the two versions of our algorithm when run against the sequestered test set. We observe that the algorithms perform similarly at the low end, where most samples are spam. The full algorithm differentiates itself in the middle of the range: as recall increases above 0.35 (corresponding to a score cutoff of about 0.1 for the full algorithm and 0.5 for the lightweight algorithm), the precision of the lightweight algorithm declines much more quickly than that of the full algorithm.

F-score.

When trying to detect spammy names, it is typically the case that that the cost of a false positive in spam detection (blocking a good user) is significantly higher than the cost of a false negative (leaving a fake account alone). A good user who is blocked represents lost engagement with the site — and potentially lost revenue — while a bad user who is not blocked based on a spammy name may be later caught and blocked based on some other signal. Furthermore, false positives are magnified by the fact that the vast majority of users are *not* spammers.

Thus to determine a reasonable score cutoff for our classifier, we consider an F-score that weights precision more heavily than recall. Specifically, given a parameter β that indicates the ratio of importance of recall to importance of precision, we compute

$$\max F_\beta = \max_{c\in[0,1]} (1+\beta^2)\left(\frac{1}{\beta^2\,\mathrm{prec}(c)} + \frac{1}{\mathrm{rec}(c)}\right)^{-1}.$$

(See [28, Ch. 7].) We computed $\max F_\beta$ scores for $\beta = 1, 1/2, 1/4, \ldots, 1/64$ for the full and lightweight algorithms and obtained the following results:

Full algorithm				
β	max F_β-score	score cutoff	precision	recall
1	0.769	0.999	0.727	0.817
2^{-1}	0.803	0.966	0.862	0.629
2^{-2}	0.876	0.646	0.921	0.490
2^{-3}	0.921	0.127	0.943	0.370
2^{-4}	0.954	3.49e-05	0.966	0.230
2^{-5}	0.975	8.89e-11	0.980	0.155
2^{-6}	0.975	1.21e-08	0.976	0.179
Lightweight algorithm				
β	max F_β-score	score cutoff	precision	recall
1	0.729	0.999	0.643	0.842
2^{-1}	0.745	0.966	0.849	0.499
2^{-2}	0.862	0.670	0.936	0.380
2^{-3}	0.928	0.147	0.961	0.291
2^{-4}	0.964	1.19e-02	0.978	0.201
2^{-5}	0.978	5.75e-05	0.983	0.163
2^{-6}	0.981	6.34e-05	0.983	0.164

We see from these computations that if precision is weighted much higher than recall ($\beta \le 1/8$), the two algorithms perform roughly identically. As recall gains in weight, we see the full algorithm begin to outperform the lightweight version.

5.2 Relative Feature Weights

In general, all it takes is one spammy n-gram to make a name spammy; for example, we would consider the name "XXXDavid" to be spam even though the substring "David" looks good. Thus for our classifier to perform well, bad features should be weighted more heavily than good features.

Our analysis suggests that this property holds intrinsically for our data set, and we believe it results from the nature of our training data. Specifically, the "spam" labels in our training set consist of *all* accounts that have been flagged as spam, not just those that have spammy names. Thus it is far more likely for a "good" n-gram such as Dav to be found on a spam account than it is for a "spammy" n-gram such as XXX to be found on a good account. In terms of the parameters that comprise the score $s(\vec{x})$, the absolute value of $\log(\theta_{XXX1}/\theta_{XXX0})$ will tend to be much larger than the absolute value of $\log(\theta_{Dav1}/\theta_{Dav0})$. As a result, even one spammy n-gram will not be counterbalanced by any number of good n-grams.

To evaluate this concept quantitatively, we bucketed scores produced by our lightweight algorithm by $\lfloor R(\vec{x})\rfloor$ (see (2.4)) and computed the following quantities:

- percentage of samples in a bucket with a "very bad" feature, defined as an n-gram w for which $\log(\theta_{w1}/\theta_{w0}) < -1$,
- percentage of samples in a bucket with a "very good" feature, defined as an n-gram w for which $\log(\theta_{w1}/\theta_{w0}) > 1$.

The results appear in Figure 7. We see that as scores approach the middle from the extremes, the rate of very bad features stays high while the rate of very good features starts to decline sooner. For example, when $\lfloor R(\vec{x})\rfloor = 5$, 75% of samples have a "very good" feature with $\log(\theta_{w1}/\theta_{w0}) > 1$. On the other hand, when $\lfloor R(\vec{x})\rfloor = -5$, 94% of samples have a "very bad" feature with $\log(\theta_{w1}/\theta_{w0}) < -1$. We conclude that the "very bad" features contribute more to the ultimate score than the "very good" features.

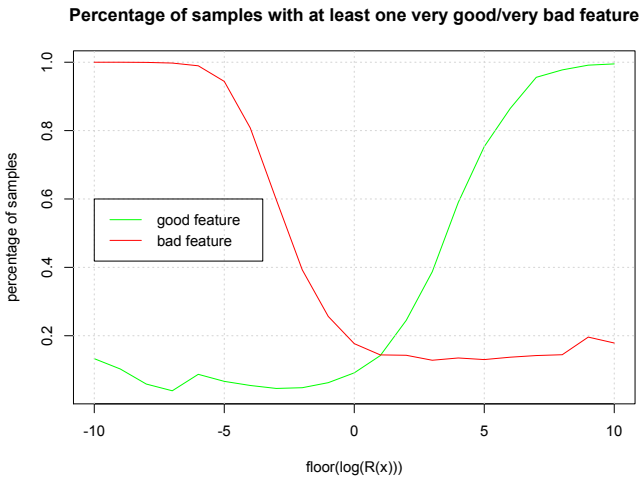

Figure 7: Percentage of samples with at least one very good or very bad feature, bucketed by $\lfloor \log R(\vec{x}) \rfloor$.

5.3 False Positives

We manually reviewed all 928 names in our test set that were labeled "good" but were given a score of 0.05 or less by our full algorithm. We found that a majority of these samples (552, or 59%) were in fact spammy names that had not been labeled as such. Labeling these alleged false postives samples correctly significantly improves precision at the low end of the score spectrum. The dashed lines in Figure 6 show precision–recall plots for the test set with the labels corrected.

Of the manually-reviewed false positives that we determed not to be spammy names, we observed a few patterns that repeated:

- Mixed-language names. For example, a Chinese person may enter his or her name in Chinese characters and English transliteration. While LinkedIn does support names in multiple languages [19], users may not be aware of this feature or may want both languages to show simultaneously.
- Users interchanging the first and last name fields. Since our algorithm uses different features for first and last names, a name that looks good when entered in the correct fields may come up as spam when the fields are interchanged.
- Users entering readable names with extra accents and/or unusual characters. For example, a user might enter "David" as "∂åν1δ." We speculate that these users are attempting to maintain a legitimate presence on the site while making it more difficult to search for their names.
- Users entering non-name information in the name fields, such as degrees, professional credentials, or military ranks.

In Section 7 we discuss some possible approaches to filtering out false positives in these categories.

5.4 False Negatives

Recall that the "spam" accounts in our training and test sets consist of accounts that had been flagged as fraudulent and/or abusive for *any* reason, not just having a spammy name. Many of the spam accounts on LinkedIn have names that do not appear spammy, so we would not expect our classifier to detect these accounts. The histograms of Figure 5 support this assertion: while more than 90% of good accounts have scores greater than 0.95, either 40% (full algorithm) or 53% (lightweight algorithm) of the spam accounts in our test set have scores above that point and would thus be labeled as false negatives.

To further support this assertion, we manually reviewed a sample of 100 accounts in our test set that were labeled as spam but were given scores above 0.646 by the full algorithm and above 0.670 by the lightweight algorithm. (These cutoffs were chosen to maximize $F_{1/4}$-score as discussed in Section 5.1.) Only seven of these accounts had names that were spammy in our judgment.

If we extrapolate this 93% false negative rate in our labeling of high-scoring accounts to the entire validation set above the cutoffs, then at these cutoffs we obtain recall 0.93 for the full algorithm and 0.90 for the lightweight algorithm (compared with 0.49 and 0.38, respectively, using the initial labels).

5.5 Running on Live Data

We implemented the lightweight version of our algorithm in Python and trained it on a subset of the dataset discussed in Section 3.2. Using Hadoop streaming, we ran the algorithm daily on new LinkedIn registrations throughout the month of May 2013. The lowest-scoring accounts were flagged as spam and sent to the Trust and Safety team for appropriate action. We ran our new algorithm in parallel with LinkedIn's previous name-spam detection procedure, which was based upon regular expressions.

As of early July, 3.3% of the accounts labeled as spam by our new algorithm had had this label removed, while 7.0% of the accounts labeled as spam by the old algorithm had had their spam labels removed. LinkedIn deemed the new algorithm so effective that the old algorithm was retired by mid-June.

6. SCORING EMAIL ADDRESSES

Our algorithm can be applied to detect spam not only in names but also in other short strings. In particular, email usernames are particularly relevant for this kind of scoring. While there is nothing about an email address that violates terms of service or degrades the value of a social network, there are still email addresses that are "spammier" than others. For example, a real user's email address will often look like a name or a word in some language, while a user who just wants to get past a registration screen may sign up with "asdf@asdf.com," and a bot generating a series of accounts may sign up with email addresses consisting of random strings of characters at some email provider.

We experimented with email data by incorporating it in our algorithm in two ways: running the algorithm on email usernames alone, and running the algorithm on email usernames along with first and last names. (That is, we added n-grams for email username to the feature set that included n-grams for first and last names.) For the latter input data,

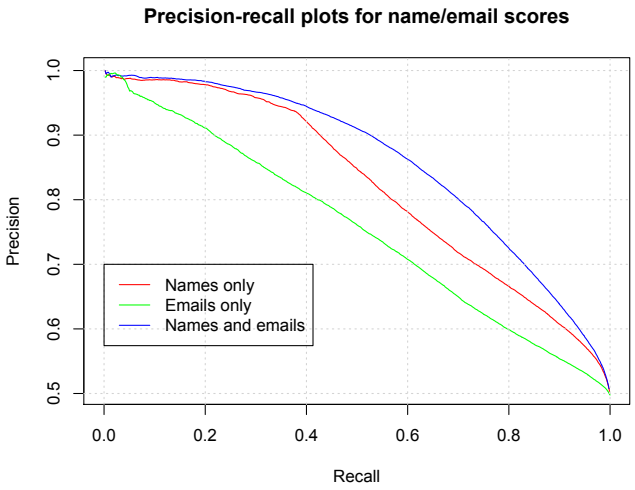

Figure 8: Precision–recall curves for algorithm on email usernames, first/last name, and all three feature sets.

the max $F_{1/8}$-score is 0.936 (precision 0.966, recall 0.312), at score cutoff 0.013. Precision–recall curves appear in Figure 8.

By itself, our algorithm applied to email usernames is not too helpful in finding spammers. Precision drops off very quickly, and even at the very low end we have worse results than our name scoring algorithm: of the 6,149 accounts with email scores less than 0.05, we find 5,788 spam accounts and 361 good accounts. Correcting for the mislabeled accounts we found in Section 5.3 shifts only 28 labels from good to spam.

However, we find that our results improve when we use email scoring to supplement name scoring; the email features provide a sort of "tiebreaker" for scores that are otherwise borderline. In particular, adding email data improves results in the middle of the precision–recall range. Further work is needed to assess the magnitude of this effect and to investigate other ways of combining email scoring with name scoring.

7. CONCLUSIONS AND FURTHER WORK

We have presented a Naive Bayes classifier that detects spammy names based on the n-grams (substrings of n characters) appearing in that name, and discussed techniques for improving and optimizing the algorithm. We tested "full" and "lightweight" versions of the algorithm on data from LinkedIn and showed that both were effective at catching spam names at high precision levels. The lightweight version becomes less effective more quickly than the full version as the score cutoff increases.

One important direction for further work is to reduce the algorithm's false positive rate. In Section 5.3 we discussed some categories of false positives returned by the algorithm; here we present some possible approaches to detecting and eliminating them.

- Mixed-language names: if multiple languages are detected, one could try to parse each name field into language blocks and score each block separately. If the name field contains a real name in each language then both scores will be high.
- Switching name fields: one could try to score the first name as a last name and vice versa. If the name is actually spam then both permutations should score badly, while if the name is legitimate but the fields are switched the scores of the permuted names should be high. The relative weighting of the permuted scores vs. the original scores would have to be determined experimentally; we don't want to negate the advantage gained by scoring first and last names independently (cf. Figure 1).
- Unusual characters: one could try scoring a "reduced" name obtained by mapping Unicode characters to ASCII strings. Again we would want to ensure that this score was appropriately weighted so as to not negate the advantage of scoring on the entire Unicode alphabet.
- Non-name information: one could try to detect appropriate non-name information such as degrees or qualifications (as opposed to irrelevant information such as company names or job titles) by matching to a list. A better approach would be to have a separate field for such information, so the name field could be reserved for the name only.

Another direction for further work is to strengthen the adversarial model. Specifically, we have trained our classifier under the assumption that the distribution of spammy names is constant over time, while in practice it is likely that malicious spammers will change their name patterns when they find themselves getting blocked due to our classification. We expect that if this adaptation occurs at a significant scale, then the offenders will be caught LinkedIn's other spam-detection systems and the name-spam model can then be retrained using the most recent data. To investigate these effects we plan to retrain the model at regular intervals and track the success rate of each iteration.

Acknowledgments.

The author thanks John DeNero for fruitful discussions and Sam Shah, Vicente Silveira, and the anonymous referees for helpful feedback on initial versions of this paper.

8. REFERENCES

[1] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, and C. D. Spyropoulos. An experimental comparison of Naive Bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '00, pages 160–167, New York, NY, USA, 2000. ACM.

[2] I. Androutsopoulos, G. Paliouras, V. Karkaletsis, G. Sakkis, C. D. Spyropoulos, and P. Stamatopoulos. Learning to filter spam e-mail: A comparison of a naive Bayesian and a memory-based approach. In *Proceedings of the Workshop "Machine Learning and Textual Information Access", European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 1–13, 2000.

[3] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida. Detecting spammers on Twitter. In *CEAS 2010 - Seventh annual Collaboration, Electronic messaging, AntiAbuse and Spam Conference*, 2010.

[4] P. N. Bennett. Assessing the calibration of Naive Bayes posterior estimates. Technical report, DTIC Document, 2000. Available at `http://www.dtic.mil/dtic/tr/fulltext/u2/a385120.pdf`.

[5] A. Blum and T. M. Mitchell. Combining labeled and unlabeled data with co-training. In *COLT*, pages 92–100, 1998.

[6] G. Brown, T. Howe, M. Ihbe, A. Prakash, and K. Borders. Social networks and context-aware spam. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, CSCW '08, pages 403–412, New York, NY, USA, 2008. ACM.

[7] Q. Cao, M. Sirivianos, X. Yang, and T. Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *NDSI*, 2012.

[8] Q. Cao and X. Yang. Sybilfence: Improving social-graph-based sybil defenses with user negative feedback. Duke CS Technical Report: CS-TR-2012-05, available at `http://arxiv.org/abs/1304.3819`, 2013.

[9] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, ACL '96, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.

[10] P. Domingos and M. Pazzani. Beyond independence: Conditions for the optimality of the simple Bayesian classifier. In *Machine Learning*, pages 105–112. Morgan Kaufmann, 1996.

[11] P. Domingos and M. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine learning*, 29(2-3):103–130, 1997.

[12] R. Duda and P. E. Hart. *Pattern classification and scene analysis*. Wiley and Sons, Inc., 1973.

[13] Facebook Inc. Statement of rights and responsibilities. `http://www.facebook.com/legal/terms`, accessed 17 Jul 2013.

[14] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 35–47, New York, NY, USA, 2010. ACM.

[15] Google Inc. User content and conduct policy. `http://www.google.com/intl/en/+/policy/content.html`, accessed 17 Jul 2013.

[16] J. Hovold. Naive Bayes spam filtering using word-position-based attributes. In *CEAS*, 2005.

[17] R. Kohavi, B. Becker, and D. Sommerfield. Improving simple Bayes. In *Proceedings of the European Conference on Machine Learning*, 1997.

[18] LinkedIn Corporation. Internal data, accurate as of 1 Jun 2013.

[19] LinkedIn Corporation. Creating or deleting a profile in another language. `http://help.linkedin.com/app/answers/detail/a_id/1717`, accessed 22 July 2013.

[20] LinkedIn Corporation. User agreement. `http://www.linkedin.com/legal/user-agreement`, accessed 17 Jul 2013.

[21] A. McCallum and K. Nigam. A comparison of event models for Naive Bayes text classification. In *Proceedings of AAAI*, 1998.

[22] V. Metsis, I. Androutsopoulos, and G. Paliouras. Spam filtering with Naive Bayes — which Naive Bayes? In *CEAS*, 2006.

[23] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[24] I. Rish. An empirical study of the Naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[25] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A Bayesian approach to filtering junk e-mail. In *Learning for Text Categorization: Papers from the AAAI Workshop, Madison Wisconsin*, volume Technical Report WS-98-05, pages 55–62. AAAI Press, 1998.

[26] K.-M. Schneider. A comparison of event models for Naive Bayes anti-spam e-mail filtering. In *EACL*, pages 307–314, 2003.

[27] A. K. Seewald. An evaluation of Naive Bayes variants in content-based learning for spam filtering. *Intelligent Data Analysis*, 11(5):497–524, 2007.

[28] C. J. van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 2nd edition, 1979.

[29] G. Wang, T. Konolige, C. Wilson, X. Wang, H. Zheng, and B. Y. Zhao. You are how you click: Clickstream analysis for sybil detection. In *Proceedings of The 22nd USENIX Security Symposium*, pages 241–256, 2013.

[30] C. M. Zhang and V. Paxson. Detecting and analyzing automated activity on Twitter. In *PAM*, pages 102–111, 2011.

[31] H. Zhang. The optimality of Naive Bayes. In *FLAIRS Conference*, pages 562–567, 2004.