Increasing the I.Q. of Your Smartphone: Reducing Email Keystrokes Using Personalized Word Prediction

Joseph A. Cooley
134 Final Project Milestone
jac@cs.dartmouth.edu

1 Introduction

In this work, we proposed to reduce the number of keystrokes required to write email on mobile devices such as smartphones. Existing devices can provide a frustrating user experience because they include small keyboards and context-insensitive typing corrections, which lead to errors and a poor email experience. We proposed to use our personal email as a source for both training and testing a model that predicts accurately the next typed word.

Our initial research suggested that we use a Markov model to compute $P(w_n|w_1 \cdots w_{n-1})$, where w_n is the nth word of an n-gram model and that the Good-Turing method [2] may provide a suitable means for smoothing probability mass among seen and unseen n-grams before predicting words.

In summary, we have completed the following tasks and appear to be slightly ahead of schedule.

- Collect, parse, and split email bodies into n-grams, and compute statistics for use in the prediction model.
- Create the Markov-based, word prediction model.
- Analyze model performance and productivity increase in writing emails.

In the rest of this document, we will describe the status of our research. In Section 2 we discuss the dataset, its origin, characteristics, and preprocessing. Then, in Section 3 we discuss the probability mass estimator used to smooth n-gram distributions and the technique used to predict words. In Section 4 we begin to analyze the performance of our approach, and in Section 5 we conclude by reviewing the schedule.

2 Dataset

We built our dataset using the sent mailbox from our gmail account. In total, the account contains nearly 1.4 GB of email total, approximately 500 MB of which corresponds to sent mail. The volume includes email headers, subject lines, message bodies, attachments, and any message content typed by others such as forwarded messages or reply-to text.

We fetched email from gmail using using a python script and the secure IMAP protocol[1]. Python includes modules for IMAP, SSL (used to secure IMAP), regular expression processing, and email processing, which we used extensively to build our dataset. Gmail transmitted each message in raw RFC822 format, which our script received and stored in a sqlite database. Then, we ran a separate python script to strip away all but message bodies from each message. A third script computed n-grams on each of the bodies, where $n \in \{1, 2, 3\}$, and Table 1 shows the vocabulary size and number of samples associated with each n-gram set.

Preprocessing

Preprocessing email was not straightforward. The goal was to capture typed text, no more, and no less. Unfortunately, many messages contained forwarded

[1] gmail supports secure IMAP access

n-gram Type	Vocabulary Size	# of Samples
unigram	59386	910812
bigram	268665	905937
trigram	422433	901314

Table 1: **Dataset statistics for** 4978 **emails from our gmail sent mailbox. The statistics are computed after preprocessing that includes stripping away message headers, attachments, and forwarded or reply-to message components. Preprocessing reduces signficantly the size of our dataset from** $\approx$ 500 **MB to** $\approx$ 6 **MB.**

```
(^From:)|(^Date:)|(^Subject:)|
(^To:)|(^>)|(wrote:\s*$)|
(Original Message)|
(^[0-9A-za-z+/]{30,}$)|
(^[0-9A-Za-z+/]+[=]*\$)
```

Figure 1: **Regular expression used to skip text. Many messages include the string "Original Message", headers, and lines prefixed with ">" because the message is a reply to another message. Others include base64 encoded text. These expressions allow the script to ignore such lines.**

```
([<>,"]+)|(\w@\w)|([*]{2,})|
(\[mailto:.*\])|([-_]{2,})
```

Figure 2: **Regular expression used to stip text. In some cases, messages include symbols or text that a user doesn't type, but the text exists in a line with user typed text. These expressions strip such non-typed characters.**

or reply-to components in special formats. For example, the character ">" at the beginning of a line typically refers to a reply-to text that should be ignored. In other special cases, constructions such as ":)" refer to valid typed text (i.e., emoticons). For these reasons, we preferred to forgo filtering punctuation and stop and stemming words—we want to account for frequently typed text and have not measured the impact of such filtering on the performance of our system.

In addition, some email messages contain HTML from web-based email systems, and others contain base64 encoded objects such as pictures. Some messages contain signatures with long strings of "*", "-", or "=" characters. We did not attempt to handle perfectly all these special cases (and more). Rather, to address a majority of the issues, we used regular expressions such as the ones shown in Figures 1 and 2 to define which lines to skip and which characters to strip before computing n-grams. In some cases, we stripped away or ignored too much text and in others, too little.

Finally, we track each message recipient set, which includes names listed with the following address fields: "Cc", "Bcc", and "To". Each address can contain an alias and an email address as in

"Joseph Cooley" <jac@cs.dartmouth.edu>.

In some cases, the alias is a duplicate address of the one it precedes, as in the following:

"jac@cs.dartmouth.edu" <jac@cs.dartmouth.edu>.

We prevent cases like these from introducing duplicate addresses within our data structures by maintaining the recipient list in a python "set" structure. A python set functions as a mathematical set; it does not store duplicates.

Dataset Sanity Check

To sanity check the word distribution of the dataset, we plotted its n-gram frequencies and compared it to Zipf's Law. Standard linguistic texts should follow Zipf's Law for frequency distributions, which says that a few events occur with high frequency and a number of events occur with low frequency [9]. Figure 3 shows that our email text seems to follow Zipf's Law.

3 Estimation Algorithm

To predict words, we use a statistical estimate of the next word derived from a smoothed probability distribution over our n-gram sets. We have chosen to use a version of the Good-Turing [4] estimator called Simple Good-Turing (SGT) [2] to smooth probability mass among both seen and unseen n-gram values. SGT assumes the underlying distribution is binomial.

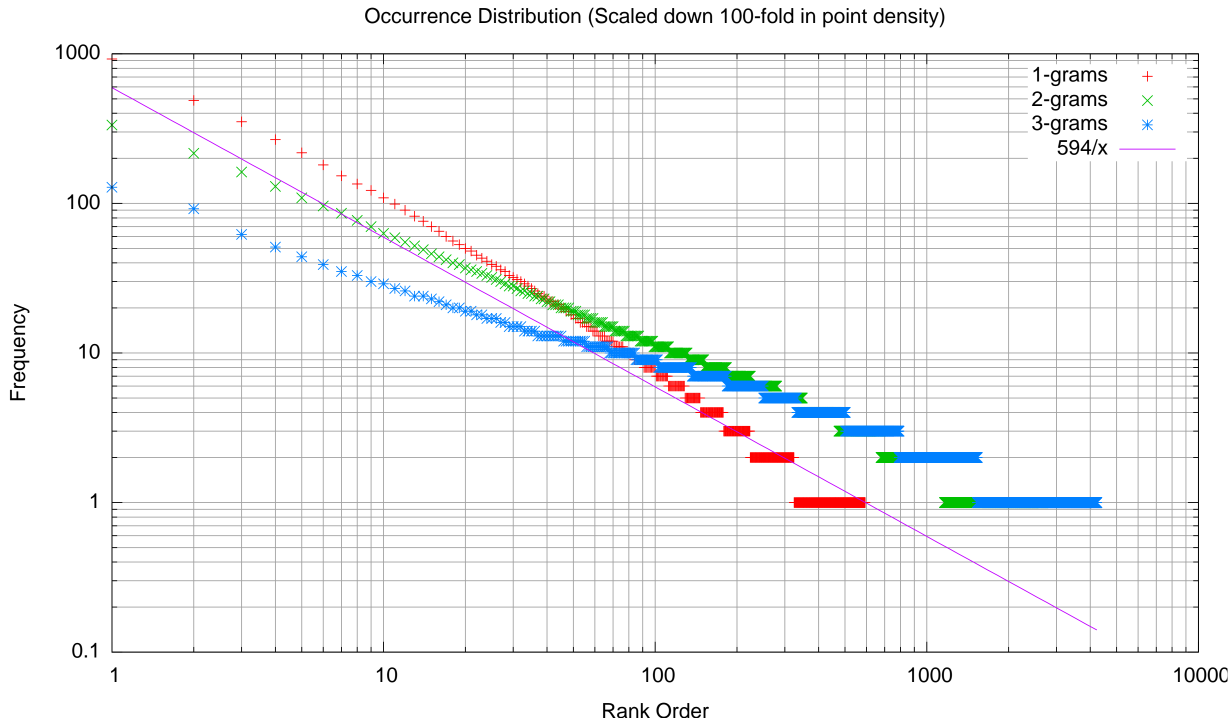

Figure 3: **Comparing n-gram frequencies to Zipf's Law. We sanity check our data to see if it follows the pattern we expect: Zipf's Law. The probability mass plot shows that n-gram frequencies of our dataset follow the Law. We plotted the Zipf line using an exponent of -1. To facilitate plotting, we filtered the vocaulary size down by two orders of magnitude to ≈ 600 elements for each n-gram set.**

In 1995, Gale and Sampson published the Simple Good-Turing model with an algorithm, sample data, and code for the estimator [2]. Our work relies on Sampon's C code [8] and a python script we wrote to manage n-grams and compute word predictions. The original Good-Turing estimator was developed by Alan Turing and an assistant I. J. Good while they worked at Beltchley Park to crack the German Enigma cipher during World War II [5].

The estimator deals with frequencies of frequencies of events and was designed to smooth a probability distribution in such a way that it accounts reasonably for events that have not occurred. A standard machine learning practice called Maximum Likelihood Estimation (MLE) does not work suitably for word prediction because it assigns probability mass solely to seen events. As we demonstrate next for unigrams, MLE neglects unseen events[2]:

$$P_{MLE}(w_i) = \frac{C(w_i)}{N},$$

where the probability of a word w_i is the count of the word $C(w_i)$ divided by the total number of words in the dataset N.

Add-one or Laplace smoothing as shown in equation 1 and seen in class, adds one to estimation components to account for unseen events. Unfortunately, it takes away too much probability mass from seen events and adds to much to unseen events.

$$\begin{aligned} P_{Laplace}(w_i) &= \frac{C(w_i)+1}{\sum_{j=1}^{V} C(w_j)+1} \\ &= \frac{C(w_i)+1}{N+V}, \end{aligned} \quad (1)$$

where j ranges across the entire vocabulatry V (unique words) in the dataset.

[2] A majority of the analytical discussion and notation in this section derives from a combination of three sources [3, 6, 7].

Frequency	Frequency of Frequencies
i	N_i
1	26972
10	693
100	13
1000	0
3288	2
31262	1

Table 2: **Sample of unigram frequencies where $N = 910812$. Good-Turing shifts probability mass from large N_i, which are better measurements, to unseen values. Note that even among the few samples shown, lower N_i values clearly become noisy. This behavior is common in linguistics data.**

Simple Good-Turing apportions probability mass to unseen events by using mass associated with events that occur 1 time. All events that occur n times are reassigned probability mass associated with events that occur $(n-1)$ times.

To make this notion concrete, consider Table 2 which contains a sample of unigram frequencies from our dataset. If we define the total number of words in the dataset as $N = \sum iN_i$ and use Good-Turning to compute the total probability of all unseen events as N_1/N, then the total probability of all unseen events in our unigram dataset is $26972/910812 = .0296$.

The goal, then, in Good-Turing is to compute the probability for events seen i times as

$$P_{GT}(w_i) = \frac{i^*}{N}.$$

The trick is to compute i^* smoothly such thtat $p_0 \neq 0$ as it would be using a method such as MLE (i.e., applying MLE to unseen events yields $p_0 = i/N = 0/N = 0$). To do this, we rely on a theorem that states the following:

$$i^* = (i+1)\frac{\mathbb{E}[N_{i+1}]}{\mathbb{E}[N_i]} \text{ [1]}$$

When N_i is large (at lower frequencies) it represents a better measurement. In these cases, we replace $\mathbb{E}[N_i]$ with N_i and call i^* a *Turing* estimator [3]. Small values of N_i represent poor measurements with much noise, and so replacing $\mathbb{E}[N_i]$ with N_i is a poor choice. In these cases, we replace $\mathbb{E}[N_i]$ with a smoothed estimate $S(N_i)$ as suggested by Good [4] and call i^* according to the smoothing function used. Table 2 shows noise in our dataset as i increases: N_i oscillates at lower values.

At this point, the problem of smoothing boils down to choosing a good smoothing function and deciding when to switch between using N_i and the smoothing function. For the function, we use $\log(N_i) = a + b\log(i)$ defined by Gale and Sampson [2]. The value of b is learned using linear regression. Gale and Sampson call the associated Good-Turing estimate the Linear Good-Turing estimate (LGT) and a renormalized version of LGT in combination with the Turing-estimator, Simple Good-Turing (SGT). Note that once the C-code begins using the smoothing function, it continues to do so.

Word Prediction

So far, we have discussed smoothing a probability mass using Simple Good-Turing. After computing smooth mass values, we compute the following using them. Our word prediction is associated with the $P(w_1 \cdots w_i)$ that maximizes the following equation.

$$P(w_i | w_1 \cdots w_{i-1}) = \frac{P(w_1 \cdots w_i)}{P(w_1 \cdots w_{i-1})} \quad (2)$$

4 Analysis

Here, we apply the C-based SGT estimator to unigrams, bigrams, and trigrams from the entire dataset and subsequently compute word predictions using a python script.

Figure 4 depicts the probability mass associated with each these n-gram dissections of the dataset, Figure 5 shows word prediction performance using estimators derived from unigrams and bigrams according to equation (2), and Figure 6 shows keystroke reduction as a result of correct word prediction. We have not yet analyzed performance after splitting data into training and testing sets, nor have we analyzed the effects of trigrams on prediction performance. We plan to take these steps later.

To gain a sense of performance, we have smoothed the entire dataset and computed hits, near hits, and

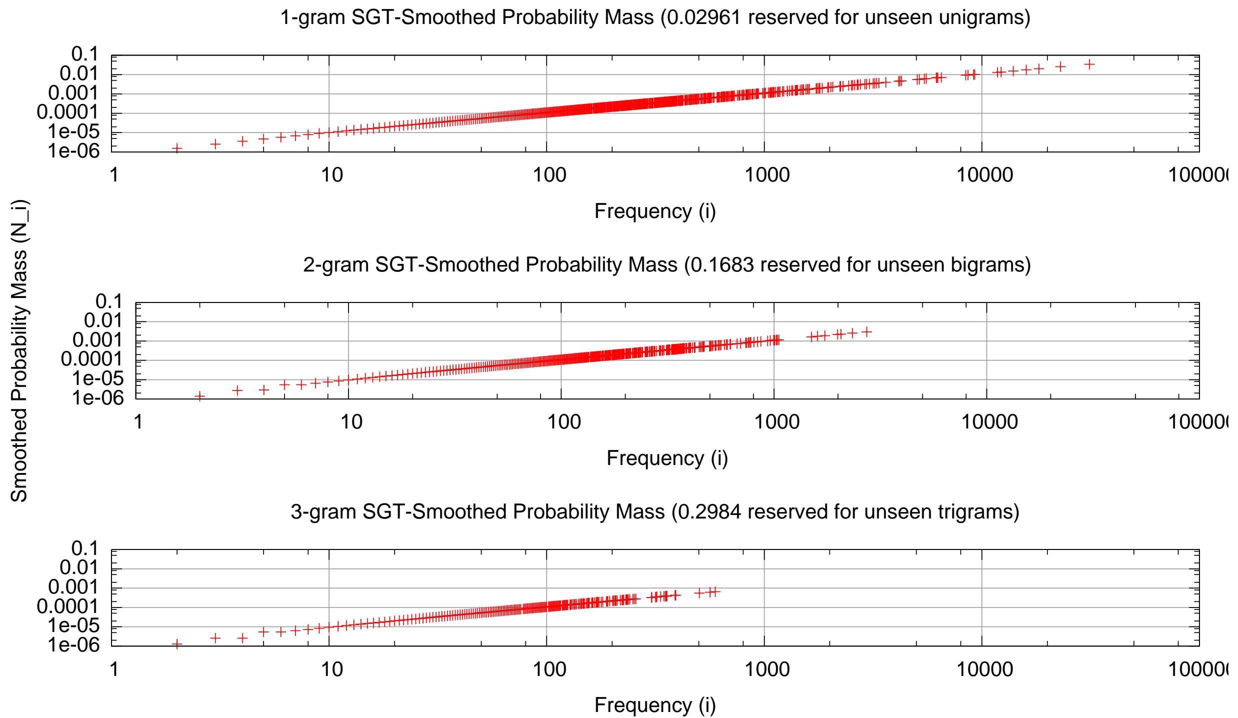

Figure 4: **Simple Good-Turing smoothed n-gram probability mass for {1,2,3}-grams. Notice how each log-log plot contains a smooth line, masses are concentrated in lower frequencies i as N_i increases, and reserved probability mass increases with the n-gram size. The last point suggests larger n-grams are very sparse. We do not have sufficient examples to see many new n-grams, and even if we had them, grammatical correctness would limit the number of new n-grams.**

misses for each message. A hit is the number of times a correct prediction is made; a near hit means that a bigram existed, but it wasn't chosen because it did't have the highest smoothed probability; and a miss is the number of incorrect predictions.

Overall, the mean of bigram performance is approximately 29% and the mean of the near-hit rate is closer to 71%. Assessing trigram performance, perhaps combining the two, and using context such as the next typed letter might lead closer to the upper-bound performance depicted in the plot. In the upper bound, we assume an ability tob convert all near hits to hits.

5 Schedule

We've completed the first two bullet points of the project and have reached our goal of choosing a word-prediction model by this milestone. Slightly ahead of schedule, we have also begun to analyze prediction performance.

- ~~Collect, parse, and split email bodies into n-grams, and compute cooccurrence statistics for use in the prediction model.~~
- ~~Create the Markov-based, word prediction model. [Completed at the milestone.]~~
- ~~Analyze model performance and productivity increase in writing emails.~~
- Complete analyzing bigram model performance.
- Analyze trigram model performance.
- Time permitting, investigate performance improvements associated with context such as subject, recipient set, and next-letter typed.

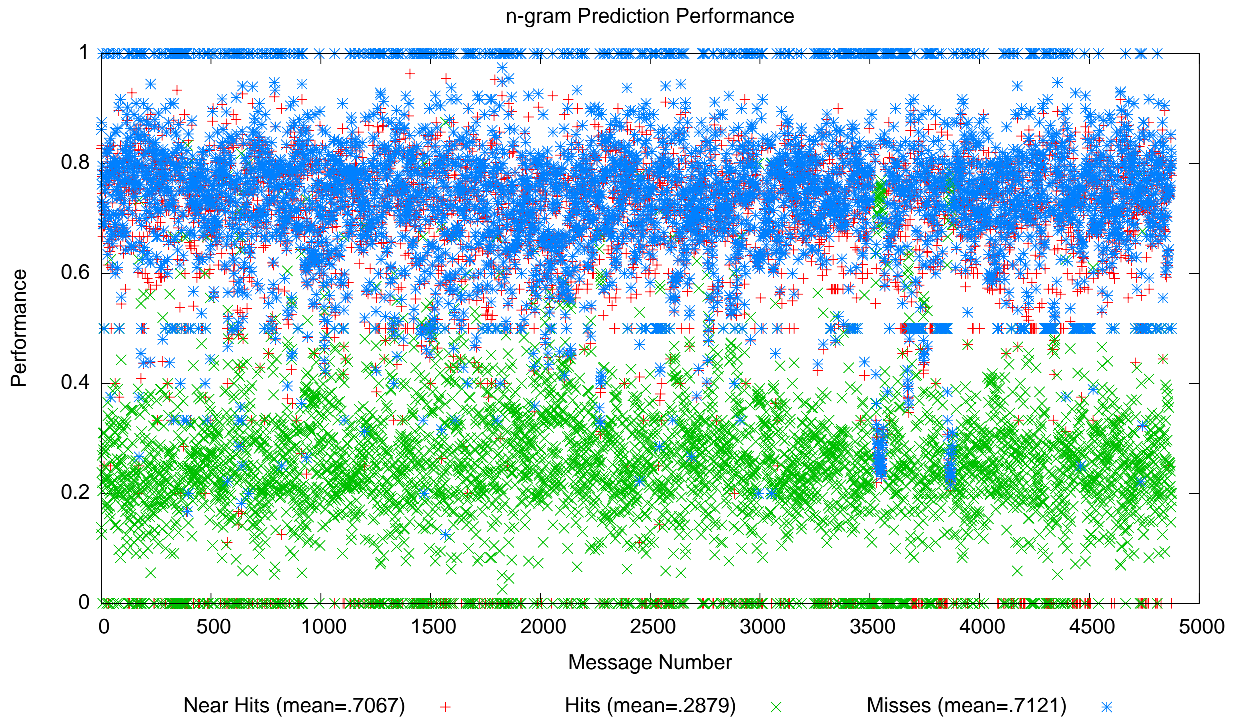

Figure 5: **Word prediction performance using unigrams and bigrams. We define "hit" to mean a correct word prediction, "miss" to mean an incorrect prediction, and "near hit" to mean that a matching bigram existed, but wasn't chosen because it didn't have the highest smoothed probability. The plot depicts the mean hit rate as $\approx 29\%$ and the mean near hit rate as $\approx 71\%$. We computed these values over our entire dataset after smoothing n-gram probability mass using SGT.**

- Time permitting, determine the reply-to address based on the message recipient set.

Next Steps

Next, we'd like to finish analyzing the bigram model, tweak the model to improve its performance, and begin analyzing the trigram model.

If time permits, we'd like to account for the message recipient set, message context, and typed letters to see if any of these attributes have a positive impact on prediction performance. In practice, both letter and message context seem important: certain topic-based words can be used multiple times within a message and typed letters narrow significantly the word prediction scope. Furthermore, using letter-typed context could convert a number of near hits and near saves to hits and saves. Exploiting letters might allow us to rely on posterior probability to reduce the set of predicted values.

Finally, we'd like to explore the use of the message recipient set to determine the *reply-to* email address. For example, I might use a jac@cs.dartmouth.edu reply-to address in messages to academic peers and a gmail reply-to address in messages to family members.

References

[1] K. Church, W. Gale, and J. Kruskal. Appendix A: The Good-Turing theorem. Computer Speech and Language, pages 19–54, 1991.

[2] W. Gale and G. Sampson. Good-Turing frequency estimation without tears. Journal of Quantitative Linguistics, 2(3):217–237, 1995.

[3] W. A. Gale. Good-Turing Smoothing Without Tears. http://citeseerx.ist.psu.edu/

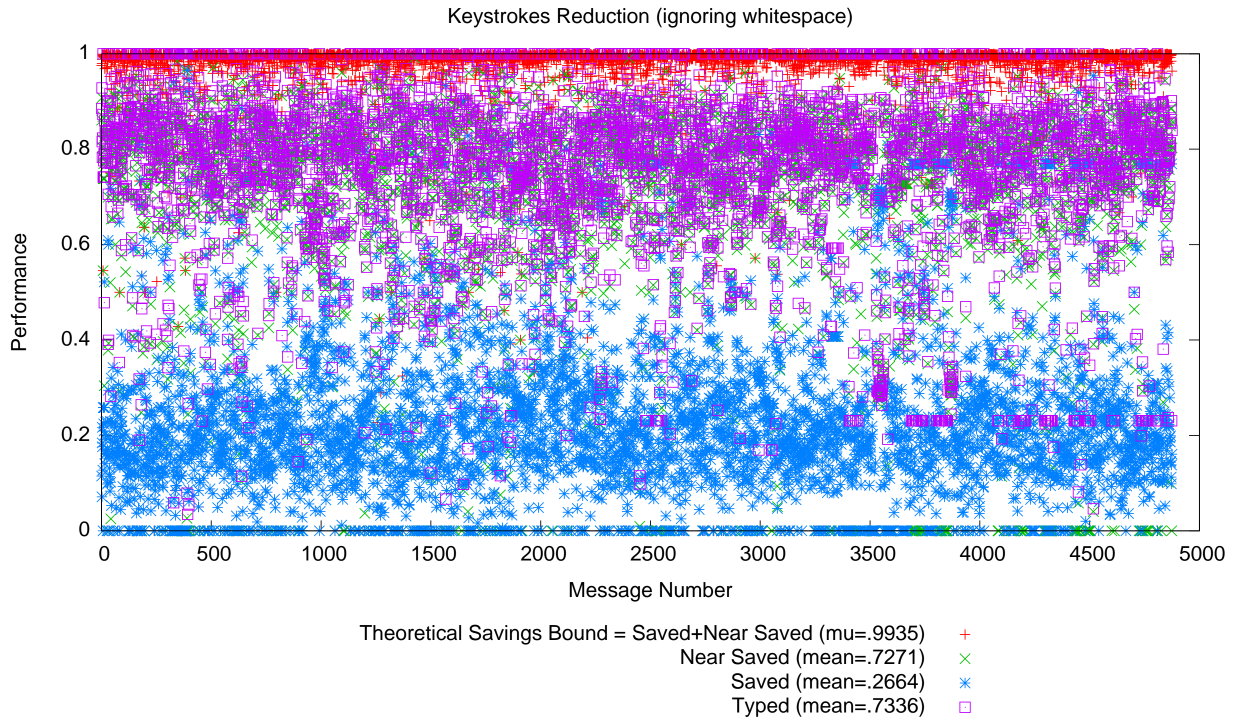

Figure 6: **Keystroke reduction using unigrams and bigrams, not counting whitespace. We define "saved" to mean a correct word prediction that eliminates typing; "typed" to mean characters typed by the user because of an incorrect prediction; and "near saved" to mean that a matching bigram existed, so typing could have been avoided, but the system didn't choose the bigram because it didn't have the highest smoothed probability. The plot depicts the mean savings rate as $\approx 27\%$ and the mean near save rate as $\approx 73\%$. We computed these values over our entire dataset after smoothing n-gram probability mass using SGT. We may find a lower "near save" when we split the dataset into training and testing portions.**

viewdoc/download?doi=10.1.1.110.8518&rep=rep1&type=pdf, 1995.

[4] I. J. Good. The population frequencies of species and the estimation of population parameters. Biometrika, 40(3-4):237, 1953.

[5] Good-turing frequency estimation. http://en.wikipedia.org/wiki/Good-Turing_frequency_estimation, 2010.

[6] J. Hockenmaier. Lecture 4: Smoothing. http://www.cs.uiuc.edu/class/fa08/cs498jh/Slides/Lecture4HO.pdf, 2008.

[7] C. Manning and H. Schütze. Foundations of Statistical Natural Nanguage Processing. MIT Press, 2000.

[8] G. Sampson. Simple good-turing frequency estimator. http://www.grsampson.net/D_SGT.c, 2010.

[9] Zipf's law. http://en.wikipedia.org/wiki/Zipf's_law, 2010.