Development Process for an Automated Tagger for Notmuch

Gifan Thadathil

1 Introduction

Notmuch is an excellent email system for efficient email storage and powerful querying. To provide better email organization, it makes use of ”tags.” Notmuch only provides the means to tag emails, but leaves the actual tagging to the user. Some users have manually created rule-based scripts to automate this tagging, however updating these rules can become tedious.

To avoid this tediousness, we have created a program that will automatically learn the tagging scheme used by the user from their existing Notmuch database of emails. After adding a call to the learned model in the Notmuch configuration files, any new emails will be automatically tagged.

This document is a detailed and somewhat informal account of the process taken to develop the tagger. It details each step of development as well as reasoning behind various development decisions. Note that this documentation does not reflect the actual linear progression of development. It is possible sections were made and then added onto much later. The goal is to provide a clear narrative of the process, not an accurate one. Roughly the development process can be broken down into the following steps:

1. Problem Formulation
2. Literature Review
3. Proposed Solution
4. Data Retrieval
5. Learning Implementation

2 Problem Formulation

Since tagging is assigning a label to an email, we have a multi-label classification problem. That is, if E is our set of emails and T is our set of tags, we want to correctly assign each email e a subset of tags in T. Furthermore, since email is typically text, this is an instance of text classification, which has a rich literature. Before diving into the literature, however, we need to note the real-world constraints and expectations of the user.

First we discuss some constraints brought by the email format and Notmuch interface. This should give us the background to flesh out a detailed usage scenario from the perspective of a user to get a sense of expectations. Then we will consider some questions to get a better idea of the behavior of the classifier.

2.1 Email Format

The object we are working with are emails, whose basic specification is given in RFC 5322. An example email message is given below.

```
From: Alice <alice@example.com>
To: Bob <bob@example.com>
Subject: Hello
Date: Mon, 1 Jan 2100 10:30:00 -0400
Reply-To: Alice <alice@example.com>

Hi Bob!
```

Note that the message is separated by a blank line into two section. The top section is called the “message header”, and the bottom section is called the “message body.” The header is comprised of fields that are essentially key-value pairs separated by a colon. A typical user simply can partially read the body of the message in addition to basic headers such as “From” and “Subject” and is able to tag the message appropriately. Therefore, we will make use of the message and the header.

However, we will not use the entire header. The header can contain some standard fields as well as custom ones. A list of standard headers is curated by the IANA. To keep things simple, we will only use the most commonly used standard header fields. Using custom fields and uncommon fields will be unhelpful for the classifier as there will most likely not be enough emails using such fields to help with classification.

2.2 Interfacing with Notmuch

Notmuch does not pull email from a server. Instead it already expects the user has downloaded their mail onto their device in the appropriate format. Once Notmuch is aware of this database of mail, the user can simply type `notmuch new` into a terminal and their mail will be indexed by Notmuch for quick searching. The user can then add tags as they wish. On receieving new email and pulling it locally to their device, the user must again execute `notmuch new` to index their new mail. Typically, users will run a script with some rules to do "initial tagging." Our tagger should work in a similar fashion. The idea of this program is to eliminate manual creation of the initial tagging script. The user should be able to run the tagger in the same way: as a post-new hook within the Notmuch configuration or within the configuration of their program to pull mail.

Notmuch makes use of a shared C library, but has a well documented API for Python, so we will create our program using Python. In addition, the python bindings allow us to read headers directly from the file rather than the Notmuch database. This is beneficial because Notmuch only indexes some of the headers. We may want more information.

2.3 Usage Scenario

After downloading the program, the user should simply place a call to it in one of their email system configuration files and be set. The only requirement is that the call to the program be executed after a call to `notmuch new`. The program should initialize itself on the first ever call and generate the classification model. From then on, it should automatically tag new emails. It is likely the model will incorrectly tag or fail to tag some emails. If the user fixes these, the model should note these differences and generate a better model. The idea is that the classifier work completely in the background without ever needing explicit work from the user.

2.4 Questions

2.4.1 Installation

How should the user install the program?
Using PyPi is probably the most "official" way to go about this. We should automatically look for the Notmuch configuration file to get the location of the database. If this is impossible, then it should fail gracefully on the first run. There is an issue here though. How will the user know it has failed? Perhaps it is better for the user to manually put the location of the database in the configuration file.

What about configuration?

Upon installation, the classifier should be able to work with sensible defaults. Any configuration should be accesible via a configuration file stored at `XDG_CONFIG_HOME`. The configuration file should be human editable, and it will most likely contain few configuration options, so a file with a simple list of key value pairs should be enough. The configuration file will most likely give the user access to tune various parameters of the classifier.

Any command line interface?
This program should work without a command line interface since it does a single thing. The configuration file should be all that is needed to run the program in the way the user wishes.

2.4.2 Parsing Email

How should we parse plain text email?
If the plain text email has no formatting, it should be enough to use the Python standard library to parse this type of text and get a list of words. From there, we can use an natural language processing library like NLTK to stem words and perform other preprocessing steps. The difficulties come when we need to parse email with different formatting. This does not refer to HTML formatting. We are referring to Markdown style formatting. Since any formatting rules can be applied, it is perhaps not possible to cover them all without treating each as a special case. A good heuristic to use would be to replace all non-alphabet characters, such as "*", with a space.

What about code?
Code is a more difficult problem since code contains a mix of formatting as well as plain text. However, it is possible that strings like variable names can provide information for the classifier. We believe removing formatting character should be enough. That way identifiers can be used by the classifier. Language keywords may be an issue though. They may not appear within many documents, and so they will not be removed by preprocessing steps. They may not be very useful for classification either since they are an artifact of the language rather than the context of the email.

How do we handle multiple languages?
Presumably, we will first support English and any other language that uses ASCII characters and has words separated by spaces. To support other languages, we will need to support Unicode. We are currently unsure how Notmuch handles Unicode email. Until we understand this and learn enough about the structures of other languages to separate words into a list.

What about formatting done for forwarded email and replies?
Typically plain text email that is a forward or a reply contain emails within them. It would seem inappropriate for the classifier to take quoted emails into account since it could bias the classifier to associating certain words to tags that belong to large threads. Fortunately, these emails within emails are typically marked, so when parsing the email we should be able to throw away this section. The problem, is the markers are not standardized. It is unclear

whether this will be a real issue. If an email does provide any markers, then nothing can be done.

What about mispelled words? This is a generally unfortunate situation, especially if a word important to the tag of the email is mispelled. We could use a spell checking library. Depending on the library, spell checking may not be available for languages besides English. In addition, it may be a huge performance hit to spell check every single word. It may be better to operate under the assumption that all emails are correctly spelled.

2.4.3 Classifying Email

Will the classifier tag spam or trashed mail?
It is possible that the classifier can tag spam, but it may not be the best option for the user. There are many powerful programs that already exist that focus solely on spam. It is best if the user uses one of those before using this classifier. There should be a configuration option that states what tags the classifier should completely ignore.

How will it know which email is new?
After calling `notmuch new`, mail that is new is given the tag "new". However, the user can use another tag if they want. There should be a configuration option that allows the user to set what tag refers to new mail that needs to be classified.

What about an undo option?
An undo option would be incredibly useful, but would require a command line interface. The goal is to avoid a command line interface, however. A good inbetween would be to add a "dry-run" entry to the configuration file, which will simulate the entire process but not actually change any tags within the Notmuch database. The simulation results could be outputted into a file.

What about threads?
Naturally, a user would tag all emails within a single thread with the same tags, but it possible a user may not. In the former case, the classifier can limit itself to tagging new emails that start their own threads. In the latter case, the classifier can ignore threads all together. But note there is some nuance to the former case. A user may want to assign additional tags depending on the content of the new emails in the thread. The question is whether to apply those new tags to all the messages in the thread or just the new ones. Perhaps there shoul be a configuration option that sets whether or not to assign every thread the same tags. This is a simplification, but should work for most cases since threads are a representation for conversations. A typical user would want to read any particular email in the context of the conversation.

2.4.4 Updating the Classifier

How often should the classifier recreate itself to account for new data/tags? We need to ensure we update classifier as little as possible, as it may be an intensive operation. On the other hand, we need to ensure the classifier is as accurate as possible, which is better guaranteed by updating. If we only update when the accuracy on new mail falls below a threshold, then we only update it when needed truly needed, which advanced the former requirement. However, it is possible that the classifier stays above the threshold consistently, but could benefit from updating. In this case, it would be better to update if no updates have happened within some threshold of time. Furthemore, it is likely the user will run the classifier everytime they poll for new mail. Polling could happen from every couple seconds to every couple of minutes. This means we can only update after some threshold of time as well. Taking all of this into account, we should only update in two cases.

1. If x time has passed and accuracy is below y.
2. If time since last update is above z.

These parameters should all be set in the configuration file. There is also a question of whether accuracy should be used as the measure of classification success as opposed to precision or recall or f-measure. Perhaps this can be set in the configuation file as well.

3 Literature Review

The literature review will be composed of different sections that discuss literature on broad subproblems we may face. At this point, the problem is finding exactly what problems we may face. We know our overall problem is that of text classification, so to address this we look for surveys on text classification. Aggarwal and Zhai [1] wrote a book on text mining. In this book they have chapter dedicated to text classification. Reading this chapter, we can break down the problem as such:

1. Feature Generation - This refers to figuring out what features we can get from an email. That is, do we let each word be a feature? Do we look at the presence of a word or its frequency? If it is the latter case, what sort of frequency do we give it? Which headers should be features? Should we look at bi/tri/n-grams as well?
2. Feature Selection/Extraction - Assuming the words will in some way be features, there will be tons of features. How do we figure out which ones to keep and which ones to remove? If we have 4 forms of the same word, should we consider it all the same word? What about words that occur very often like "and"? Perhaps we want to create a new set of features if it helps with classification?

3. Classifier Type Selection - What type of classifier do we use? A decision tree? A neural network? A SVM?

3.1 Feature Generation

An email is made up of the headers and the message. It seems that Berry and Castellanos [2] found that using the extensive feature set of SpamAssassin, which includes information from the headers, results in better classification performance than using just the message. This is empirical evidence that fits the intuition we gave before, so we should also use headers.

There is really only one general way of generating features from the message. The n-grams of a message refer to each sequence of n numbers. So if our message is "Alice said hi to Bob", then the 3-grams are $(Alice, said, hi), (said, hi, to)$, and (hi, to, Bob). According to Aggarwal and Zhai [1], text is typically represented using 1-grams. This is also called the "bag-of-words" or "vector space model" representation of text. Since the bag-of-words model is standard and has been used extensively, we should use it as well. However, there are words like "data mining" that represent one concept, so perhaps we should include 2-grams, and 3-grams as well. This will add to the total number of features, but some preliminary feature selection should remove almost all the n-grams (for $n > 1$) this introduces.

Suppose we iterate over all email messages $M_1, M_2, \cdots, M_m$, representing each with a bag-of-words model, then we can derive an initial set of features $(w_1, w_2, \cdots w_n)$ where w_i represents a word that appears in any of the emails. Note that for our purposes, we'll just consider an n-gram (for $n > 1$) as a single word. Note our data set looks like the following

	w_1	w_2	$\cdots$	w_n
M_1				
M_2				
$\vdots$				
M_m				

Note that all the entries are missing values. Now that we have our features, we have to figure out exactly what values this features take. We want the values to represent how important a particular word is to that message. The intuition is that if certain is emails with a certain tag tend to have a certain set of words, then a new email, whose important words compromise much of that set, should have the same tag. There are multiple ways to define the importance of a word. This StackExchange post gives a good explanation, which we reproduce here.

1. Binary Values - If word w_i is present in message M_j, then the value at entry (j, i) in the data set is 1. Otherwise it is 0. This is a good start since a word cannot be important if it doesn't appear, but it seems to remove too much information. In particular it is

possible a word can be present in two emails that get tagged with different tags, but the word appears more often in an email with a particular tag. For example "linear" may generally appear in emails tagged "math" but appear very often in emails with a tag "linearClassifierResearch". Term Frequency addresses this issue.

2. Term Frequency (tf) - If word w_i is present in message M_j, then the value at entry (j, i) in the data set is the number of times w_i appears in M_j. Some variations to this include dividing the count by the number of words in M_j. This is a measure of how often a word appears in a message. The intuition is that if a word appers more often, then it is more important to the document. This intuition fails with words, such as "the", which appear in many messages. tf-idf accounts for this.

3. Term Frequency-Inverse Document Frequency (tf-idf) - This value uses tf and a new value idf. The purpose of *idf* is to characterize how often a word appears in the entire set of emails. It is calculated as

   $$idf(w) = \log \frac{\text{Number of emails}}{\text{Number of emails containing } w}$$

   If a word is contained in a lot of emails, then idf is small. tf-idf is simply the product of tf and idf when tf is calculated using the number of words in an email. Note that a normalization step is typically done after calculating tf-idf values.

Out of these three, tf-idf values are clearly the best to assign the word features since they denote the importance of a word in the most robust manner.

3.2 Feature Selection/Extraction

Now that we have our word features, we have to figure out which ones to keep and which ones to discard. There are two ways of doing this. We can select from the original set of word features or we can extract a new set of features to classify by from the original set of word features. The former is a classical method of getting the final set of features and has the empirical evidence to back up its usefullnes. The latter is a more modern way which promises to do better but may not have as much evidence to back it up.

Before we talk about selection and extraction, we will talk about simple preprocessing steps that are technically selection, but seem to basic to go under that heading. According to Aggarwal and Zhai [1], these steps include

1. Removing punctuation.

2. Removing stop words, which are words very common to a language like "and" or "the", which are highly unlikely to discriminate between classes.

3. Stem words. This refers to removing prefixes and suffixes such that words like "relearn" and "unlearn" get coalesced into just "learn". This is useful because lexically they are all the same.

3.2.1 Feature Selection

We have been a bit misleading so far. The normal tf-idf is unsupervised, but Lan et al. [4] give a survey and comparison of other metrics similar to td-idf that work in a supervised manner. In these metrics rather than multiplying tf by idf, they multiply tf by something like information gain or chi square. They suggest a supervised metric that performs better than any other metric in text classification, but it only works marginally better than tf-idf. We can assign values based on these supervised metrics. And for feature selection we can remove features based on whether values for a word feature lie outside some threshold.

Aggarwal and Zhai [1] states people have used slightly different formulation of information gain, chi square, etc. to select features from the bag-of-words model before any explicit values have been assigned to them. With supervised metric, it seems we are assigning values that denote how important a word is to particular message in a supervised fashion. If a word is unimportant to all messages we can remove that word feature. In this different formulation, it seems we are assigning values that denote how important a word is to the entire set of messages, and removing word features based on this. It is unclear whether both approaches do the same thing in slightly different ways. We are hesitant to use supervised metrics since the study found it did not make much of a difference. In addition, the latter formulation is well documented.

Another method of feature selection is to use linear classifier weights. It does an initial classification of a subset of the data using the full feature set using a linear classifier like a linear SVM or neural network. After this is done, some weights will be given to the classifier. The idea is to use these weights to chooose the features. Small weights imply the feature is unimportant. Mladenić et al. [6] show that this sort of feature selection provides better results over prior ones. In addition it may simplify the implementation since the same classifier can be used for feature selection and for classification in the end. Hardin et al. [3] give some theoretical evidence stating weights generated by linear SVMs can assign zero weights to non-relevant features in the sample limit. However, they have also shown this can theoretically happen for relevant features as well.

3.2.2 Feature Extraction

According to Aggarwal and Zhai [1] there are a couples bodies of feature extraction algorithms for messages. They are all complicated so we do not discuss them in detail, but we will give a short overview.

1. Latent Semantic Analsys (LSA) - A body of algorithms that take a co-occurrence matrix of words and documents, and produces an approximation to that matrix that better shows what words are relevant to different classes of documents. There are supervised and unsupervised versions of this. It has been shown there isn't much of a performance difference between unsupervised an supervised versions though.

2. Clustering - This a clever one. It essentially clusters the data with supervision, and chooses the word features that occur most commonly within each cluster. Some work has shown this provides an increase in classification performance.

3. Linear Discriminant Analysis (LDA) - TODO

4. Generalized Singular Value Decomposition - TODO

5. Word Embeddings - This refers to a recent work on algorithms which transforms a word into a representative vector. A prominent one includes word2vec. Unlike td-idf which only looks at presence and counts, word2vec takes semantics into account, which could help with classification. If we take a message and average up all the vectors for the words, then we can get a vector that represents the meaning of that message. The intuition would be that vectors that have the similar meanings will have small distance between them. The downside is that word2vec uses a neural network, so we will not a have a good idea of what is going on underneath and training times could be very long. Note that there is also something called doc2vec, which converts an entire message into a vector in a different way. Lau and Baldwin [5] show that doc2vec and word2vec perform very well at classification tasks against other document embedding algorithms. It may be possible to avoid training if we use a model already trained on external corpora. This would eliminate the need to update the tagger at all assuming the external corpora included every word that would show up in an email. This is unlikely so we would need retraining. This may not be an issue however since we can simply add the new words and retrain with the word vectors initialized to whatever we found before. However, the glaring issue is that files with pre-trained word vectors are giant: in the gigabytes. Additionally, perhaps the training corpora matters. We could train on the words from the emails, but [**?**] found that word2vec perform poorly when given a small set of words, which is likely to be the case with emails. They also showed LSA performed better with such small sets.

   It is difficult to choose between these since there doesn't seem to exist any paper that compares everything on a large set of documents in classification tasks. Perhaps it is best, if we implement one to begin and implement other and give the user the ability to switch. The promising results of word embedding suggest we should implement tha first, but the file sizes of pre-trained word vectors is a huge turn off as well as training times. We could simply keep the trained model elsewhere and provide and API to get the vectors, but this would be a privacy concern since the words would be sent to a third party. It would be better if all classification could be done offline.

References

[1] Charu C. Aggarwal and Cheng Xiang Zhai. *Mining Text Data*. Springer Publishing Company, Incorporated, 2012. ISBN 1461432227, 9781461432227.

[2] M.W. Berry and M. Castellanos. *Survey of Text Mining II: Clustering, Classification, and Retrieval*. SpringerLink: Springer e-Books. Springer London, 2007. ISBN 9781848000469. URL https://books.google.com/books?id=-nkPdyRtN9sC.

[3] Douglas Hardin, Ioannis Tsamardinos, and Constantin F. Aliferis. A theoretical characterization of linear svm-based feature selection. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 48–, New York, NY, USA, 2004. ACM. ISBN 1-58113-838-5. doi: 10.1145/1015330.1015421. URL http://doi.acm.org/10.1145/1015330.1015421.

[4] Man Lan, Chew Lim Tan, Jian Su, and Yue Lu. Supervised and traditional term weighting methods for automatic text categorization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(4):721–735, April 2009. ISSN 0162-8828. doi: 10.1109/TPAMI.2008.110. URL http://dx.doi.org/10.1109/TPAMI.2008.110.

[5] Jey Han Lau and Timothy Baldwin. An empirical evaluation of doc2vec with practical insights into document embedding generation. *CoRR*, abs/1607.05368, 2016. URL http://arxiv.org/abs/1607.05368.

[6] Dunja Mladenić, Janez Brank, Marko Grobelnik, and Natasa Milic-Frayling. Feature selection using linear classifier weights: Interaction with classification models. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, pages 234–241, New York, NY, USA, 2004. ACM. ISBN 1-58113-881-4. doi: 10.1145/1008992.1009034. URL http://doi.acm.org/10.1145/1008992.1009034.