Natural Language Models for Predicting Programming Comments

Dana Movshovitz-Attias
Computer Science Department
Carnegie Mellon University
dma@cs.cmu.edu

William W. Cohen
Computer Science Department
Carnegie Mellon University
wcohen@cs.cmu.edu

Abstract

Statistical language models have successfully been used to describe and analyze natural language documents. Recent work applying language models to programming languages is focused on the task of predicting code, while mainly ignoring the prediction of programmer comments. In this work, we predict comments from JAVA source files of open source projects, using topic models and n-grams, and we analyze the performance of the models given varying amounts of background data on the project being predicted. We evaluate models on their comment-completion capability in a setting similar to code-completion tools built into standard code editors, and show that using a comment completion tool can save up to 47% of the comment typing.

1 Introduction and Related Work

Statistical language models have traditionally been used to describe and analyze natural language documents. Recently, software engineering researchers have adopted the use of language models for modeling software code. Hindle et al. (2012) observe that, as code is created by humans it is likely to be repetitive and predictable, similar to natural language. NLP models have thus been used for a variety of software development tasks such as code token completion (Han et al., 2009; Jacob and Tairas, 2010), analysis of names in code (Lawrie et al., 2006; Binkley et al., 2011) and mining software repositories (Gabel and Su, 2008).

An important part of software programming and maintenance lies in documentation, which may come in the form of tutorials describing the code, or inline comments provided by the programmer. The documentation provides a high level description of the task performed by the code, and may include examples of use-cases for specific code segments or identifiers such as classes, methods and variables. Well documented code is easier to read and maintain in the long-run but writing comments is a laborious task that is often overlooked or at least postponed by many programmers.

Code commenting not only provides a summarization of the conceptual idea behind the code (Sridhara et al., 2010), but can also be viewed as a form of document expansion where the comment contains significant terms relevant to the described code. Accurately predicted comment words can therefore be used for a variety of linguistic uses including improved search over code bases using natural language queries, code categorization, and locating parts of the code that are relevant to a specific topic or idea (Tseng and Juang, 2003; Wan et al., 2007; Kumar and Carterette, 2013; Shepherd et al., 2007; Rastkar et al., 2011). A related and well studied NLP task is that of predicting natural language caption and commentary for images and videos (Blei and Jordan, 2003; Feng and Lapata, 2010; Feng and Lapata, 2013; Wu and Li, 2011).

In this work, our goal is to apply statistical language models for predicting class comments. We show that n-gram models are extremely successful in this task, and can lead to a saving of up to 47% in comment typing. This is expected as n-grams have been shown as a strong model for language and speech prediction that is hard to improve upon (Rosenfeld, 2000). In some cases however, for example in a document expansion task, we wish to extract important terms relevant to the code regardless of local syntactic dependencies. We hence also evaluate the use of LDA (Blei et al., 2003) and link-LDA (Erosheva et al., 2004) topic models, which are more relevant for the term extraction scenario. We find that the topic model performance can be improved by distinguishing *code* and *text* tokens in the code.

Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, pages 35–40,
Sofia, Bulgaria, August 4-9 2013.

2 Method

2.1 Models

We train n-gram models ($n = 1, 2, 3$) over source code documents containing sequences of combined code and text tokens from multiple training datasets (described below). We use the Berkeley Language Model package (Pauls and Klein, 2011) with absolute discounting (Kneser-Ney smoothing; (1995)) which includes a backoff strategy to lower-order n-grams. Next, we use LDA topic models (Blei et al., 2003) trained on the same data, with 1, 5, 10 and 20 topics. The joint distribution of a topic mixture θ, and a set of N topics z, for a single source code document with N observed word tokens, $d = \{w_i\}_{i=1}^{N}$, given the Dirichlet parameters α and β, is therefore

$$p(\theta, z, w|\alpha, \beta) = \\ p(\theta|\alpha) \prod_{w} p(z|\theta)p(w|z, \beta) \quad (1)$$

Under the models described so far, there is no distinction between text and code tokens.

Finally, we consider documents as having a mixed membership of two entity types, *code* and *text* tokens, $d = (\{w_i^{code}\}_{i=1}^{C_n}, \{w_i^{text}\}_{i=1}^{T_n})$, where the *text* words are tokens from comment and string literals, and the *code* words include the programming language syntax tokens (*e.g.*, `public`, `private`, `for`, etc') and all identifiers. In this case, we train link-LDA models (Erosheva et al., 2004) with 1, 5, 10 and 20 topics. Under the link-LDA model, the mixed-membership joint distribution of a topic mixture, words and topics is then

$$p(\theta, z, w|\alpha, \beta) = p(\theta|\alpha) \cdot \\ \prod_{w^{text}} p(z^{text}|\theta)p(w^{text}|z^{text}, \beta) \cdot \\ \prod_{w^{code}} p(z^{code}|\theta)p(w^{code}|z^{code}, \beta) \quad (2)$$

where θ is the joint topic distribution, w is the set of observed document words, z^{text} is a topic associated with a text word, and z^{code} a topic associated with a code word.

The LDA and link-LDA models use Gibbs sampling (Griffiths and Steyvers, 2004) for topic inference, based on the implementation of Balasubramanyan and Cohen (2011) with single or multiple entities per document, respectively.

2.2 Testing Methodology

Our goal is to predict the tokens of the JAVA class comment (the one preceding the class definition) in each of the test files. Each of the models described above assigns a probability to the next comment token. In the case of n-grams, the probability of a token word w_i is given by considering previous words $p(w_i|w_{i-1}, \ldots, w_0)$. This probability is estimated given the previous $n-1$ tokens as $p(w_i|w_{i-1}, \ldots, w_{i-(n-1)})$.

For the topic models, we separate the document tokens into the class definition and the comment we wish to predict. The set of tokens of the class comment w^c, are all considered as text tokens. The rest of the tokens in the document w^r, are considered to be the class definition, and they may contain both code and text tokens (from string literals and other comments in the source file). We then compute the posterior probability of document topics by solving the following inference problem conditioned on the w^r tokens

$$p(\theta, z^r|w^r, \alpha, \beta) = \frac{p(\theta, z^r, w^r|\alpha, \beta)}{p(w^r|\alpha, \beta)} \quad (3)$$

This gives us an estimate of the document distribution, θ, with which we infer the probability of the comment tokens as

$$p(w^c|\theta, \beta) = \sum_{z} p(w^c|z, \beta)p(z|\theta) \quad (4)$$

Following Blei et al. (2003), for the case of a single entity LDA, the inference problem from equation (3) can be solved by considering $p(\theta, z, w|\alpha, \beta)$, as in equation (1), and by taking the marginal distribution of the document tokens as a continuous mixture distribution for the set $w = w^r$, by integrating over θ and summing over the set of topics z

$$p(w|\alpha, \beta) = \int p(\theta|\alpha) \cdot \\ \left(\prod_{w} \sum_{z} p(z|\theta)p(w|z, \beta) \right) d\theta \quad (5)$$

For the case of link-LDA where the document is comprised of two entities, in our case *code* tokens and *text* tokens, we can consider the mixed-membership joint distribution θ, as in equation (2), and similarly the marginal distribution $p(w|\alpha, \beta)$ over both code and text tokens from w^r. Since comment words in w^c are all considered as text tokens they are sampled using *text* topics, namely z^{text}, in equation (4).

3 Experimental Settings

3.1 Data and Training Methodology

We use source code from nine open source JAVA projects: Ant, Cassandra, Log4j, Maven, Minor-Third, Batik, Lucene, Xalan and Xerces. For each project, we divide the source files into a training and testing dataset. Then, for each project in turn, we consider the following three main training scenarios, leading to using three training datasets.

To emulate a scenario in which we are predicting comments in the middle of project development, we can use data (documented code) from the same project. In this case, we use the in-project training dataset (*IN*). Alternatively, if we train a comment prediction model at the beginning of the development, we need to use source files from other, possibly related projects. To analyze this scenario, for each of the projects above we train models using an out-of-project dataset (*OUT*) containing data from the other eight projects.

Typically, source code files contain a greater amount of code versus comment text. Since we are interested in predicting comments, we consider a third training data source which contains more English text as well as some code segments. We use data from the popular Q&A website StackOverflow (*SO*) where users ask and answer technical questions about software development, tools, algorithms, etc'. We downloaded a dataset of all actions performed on the site since it was launched in August 2008 until August 2012. The data includes 3,453,742 questions and 6,858,133 answers posted by 1,295,620 users. We used only posts that are tagged as JAVA related questions and answers.

All the models for each project are then tested on the testing set of that project. We report results averaged over all projects in Table 1.

Source files were tokenized using the Eclipse JDT compiler tools, separating code tokens and identifiers. Identifier names (of classes, methods and variables), were further tokenized by camel case notation (e.g., 'minMargin' was converted to 'min margin'). Non alpha-numeric tokens (e.g., dot, semicolon) were discarded from the code, as well as numeric and single character literals. Text from comments or any string literals within the code were further tokenized with the Mallet statistical natural language processing package (McCallum, 2002). Posts from SO were parsed using the Apache Tika toolkit[1] and then tokenized with the Mallet package. We considered as raw code tokens anything labeled using a $<$*code*$>$ markup (as indicated by the SO users who wrote the post).

3.2 Evaluation

Since our models are trained using various data sources the vocabularies used by each of them are different, making the comment likelihood given by each model incomparable due to different sets of out-of-vocabulary tokens. We thus evaluate models using a character saving metric which aims at quantifying the percentage of characters that can be saved by using the model in a word-completion settings, similar to standard code completion tools built into code editors. For a comment word with n characters, $w = w_1, \ldots, w_n$, we predict the two most likely words given each model filtered by the first $0, \ldots, n$ characters of w. Let k be the minimal k_i for which w is in the top two predicted word tokens where tokens are filtered by the first k_i characters. Then, the number of saved characters for w is $n - k$. In Table 1 we report the average percentage of saved characters per comment using each of the above models. The final results are also averaged over the nine input projects. As an example, in the predicted comment shown in Table 2, taken from the project *Minor-Third*, the token *entity* is the most likely token according to the model *SO trigram*, out of tokens starting with the prefix 'en'. The saved characters in this case are 'tity'.

4 Results

Table 1 displays the average percentage of characters saved per class comment using each of the models. Models trained on in-project data (*IN*) perform significantly better than those trained on another data source, regardless of the model type, with an average saving of 47.1% characters using a trigram model. This is expected, as files from the same project are likely to contain similar comments, and identifier names that appear in the comment of one class may appear in the code of another class in the same project. Clearly, in-project data should be used when available as it improves comment prediction leading to an average increase of between 6% for the worst model (26.6 for *OUT* unigram versus 33.05 for *IN*) and 14% for the best (32.96 for *OUT* trigram versus 47.1 for *IN*).

[1] http://tika.apache.org/

Model	n-gram			LDA				Link-LDA			
n / topics	1	2	3	20	10	5	1	20	10	5	1
IN	33.05 (3.62)	43.27 (5.79)	47.1 (6.87)	34.20 (3.63)	33.93 (3.67)	33.63 (3.67)	33.05 (3.62)	35.76 (3.95)	35.81 (4.12)	35.37 (3.98)	34.59 (3.92)
OUT	26.6 (3.37)	31.52 (4.17)	32.96 (4.33)	26.79 (3.26)	26.8 (3.36)	26.86 (3.44)	26.6 (3.37)	28.03 (3.60)	28 (3.56)	28 (3.67)	27.82 (3.62)
SO	27.8 (3.51)	33.29 (4.40)	34.56 (4.78)	27.25 (3.67)	27.22 (3.44)	27.34 (3.55)	27.8 (3.51)	28.08 (3.48)	28.12 (3.58)	27.94 (3.56)	27.9 (3.45)

Table 1: Average percentage of characters saved per comment using n-gram, LDA and link-LDA models trained on three training sets: *IN*, *OUT*, and *SO*. The results are averaged over nine JAVA projects (with standard deviations in parenthesis).

Model	Predicted Comment
IN trigram	"Train a named-entity extractor"
IN link-LDA	"Train a named-entity extractor"
OUT trigram	"Train a named-entity extractor"
SO trigram	"Train a named-entity extractor"

Table 2: Sample comment from the *Minor-Third* project predicted using *IN*, *OUT* and *SO* based models. Saved characters are underlined.

Dataset	n-gram	link-LDA
IN	2778.35	574.34
OUT	1865.67	670.34
SO	1898.43	638.55

Table 3: Average words per project for which each tested model completes the word better than the other. This indicates that each of the models is better at predicting a different set of comment words.

Of the out-of-project data sources, models using a greater amount of text (*SO*) mostly outperformed models based on more code (*OUT*). This increase in performance, however, comes at a cost of greater run-time due to the larger word dictionary associated with the *SO* data. Note that in the scope of this work we did not investigate the contribution of each of the background projects used in *OUT*, and how their relevance to the target prediction project effects their performance.

The trigram model shows the best performance across all training data sources (47% for *IN*, 32% for *OUT* and 34% for *SO*). Amongst the tested topic models, link-LDA models which distinguish *code* and *text* tokens perform consistently better than simple LDA models in which all tokens are considered as text. We did not however find a correlation between the number of latent topics learned by a topic model and its performance. In fact, for each of the data sources, a different number of topics gave the optimal character saving results.

Note that in this work, all topic models are based on unigram tokens, therefore their results are most comparable with that of the unigram in Table 1, which does not benefit from the backoff strategy used by the bigram and trigram models. By this comparison, the link-LDA topic model proves more successful in the comment prediction task than the simpler models which do not distinguish *code* and *text* tokens. Using n-grams without backoff leads to results significantly worse than any of the presented models (not shown).

Table 2 shows a sample comment segment for which words were predicted using trigram models from all training sources and an in-project link-LDA. The comment is taken from the *TrainExtractor* class in the *Minor-Third* project, a machine learning library for annotating and categorizing text. Both *IN* models show a clear advantage in completing the project-specific word *Train*, compared to models based on out-of-project data (*OUT* and *SO*). Interestingly, in this example the trigram is better at completing the term *named-entity* given the prefix *named*. However, the topic model is better at completing the word *extractor* which refers to the target class. This example indicates that each model type may be more successful in predicting different comment words, and that combining multiple models may be advantageous.

This can also be seen by the analysis in Table 3 where we compare the average number of words completed better by either the best n-gram or topic model given each training dataset. Again, while n-grams generally complete more words better, a considerable portion of the words is better completed using a topic model, further motivating a hybrid solution.

5 Conclusions

We analyze the use of language models for predicting class comments for source file documents containing a mixture of *code* and *text* tokens. Our experiments demonstrate the effectiveness of using language models for comment completion, showing a saving of up to 47% of the comment characters. When available, using in-project training data proves significantly more successful than using out-of-project data. However, we find that when using out-of-project data, a dataset based on more words than code performs consistently better. The results also show that different models are better at predicting different comment words, which motivates a hybrid solution combining the advantages of multiple models.

Acknowledgments

This research was supported by the NSF under grant CCF-1247088.

References

Ramnath Balasubramanyan and William W Cohen. 2011. Block-lda: Jointly modeling entity-annotated text and entity-entity links. In *Proceedings of the 7th SIAM International Conference on Data Mining*.

Dave Binkley, Matthew Hearn, and Dawn Lawrie. 2011. Improving identifier informativeness using part of speech information. In *Proc. of the Working Conference on Mining Software Repositories*. ACM.

David M Blei and Michael I Jordan. 2003. Modeling annotated data. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*. ACM.

David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of Machine Learning Research*.

Elena Erosheva, Stephen Fienberg, and John Lafferty. 2004. Mixed-membership models of scientific publications. *Proceedings of the National Academy of Sciences of the United States of America*.

Yansong Feng and Mirella Lapata. 2010. How many words is a picture worth? automatic caption generation for news images. In *Proc. of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Yansong Feng and Mirella Lapata. 2013. Automatic caption generation for news images. *IEEE transactions on pattern analysis and machine intelligence*.

Mark Gabel and Zhendong Su. 2008. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 339–349. ACM.

Thomas L Griffiths and Mark Steyvers. 2004. Finding scientific topics. *Proc. of the National Academy of Sciences of the United States of America*.

Sangmok Han, David R Wallace, and Robert C Miller. 2009. Code completion from abbreviated input. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 332–343. IEEE.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE.

Ferosh Jacob and Robert Tairas. 2010. Code template inference using language models. In *Proceedings of the 48th Annual Southeast Regional Conference*. ACM.

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95.*, volume 1, pages 181–184. IEEE.

Naveen Kumar and Benjamin Carterette. 2013. Time based feedback and query expansion for twitter search. In *Advances in Information Retrieval*, pages 734–737. Springer.

Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. Whats in a name? a study of identifiers. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 3–12. IEEE.

Andrew Kachites McCallum. 2002. Mallet: A machine learning for language toolkit.

Adam Pauls and Dan Klein. 2011. Faster and smaller n-gram language models. In *Proceedings of the 49th annual meeting of the Association for Computational Linguistics: Human Language Technologies*, volume 1, pages 258–267.

Sarah Rastkar, Gail C Murphy, and Alexander WJ Bradley. 2011. Generating natural language summaries for crosscutting source code concerns. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 103–112. IEEE.

Ronald Rosenfeld. 2000. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278.

David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2007. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM.

Yuen-Hsien Tseng and Da-Wei Juang. 2003. Document-self expansion for text categorization. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 399–400. ACM.

Xiaojun Wan, Jianwu Yang, and Jianguo Xiao. 2007. Single document summarization with document expansion. In *Proc. of the National Conference on Artificial Intelligence*. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Roung-Shiunn Wu and Po-Chun Li. 2011. Video annotation using hierarchical dirichlet process mixture model. *Expert Systems with Applications*, 38(4):3040–3048.