CS230

Automated Topic-Tagging for Software-Related Question-and-Answer Sites

Ankur Agrawal
Department of Computer Science
Stanford University
ankuragr@stanford.edu

Ryan A. Chi
Department of Computer Science
Stanford University
ryanchi@cs.stanford.edu

Varuni Gupta
Department of Computer Science
Stanford University
varuni@stanford.edu

Abstract

The software engineering domain produces a large amount of textual data, comprising a mixture of code, technical jargon, and natural language. Consequently, the correct classification and tagging of such data is an open problem in both community question answering (CQA) websites and massive open online Course (MOOC) forums in the domain. As such, an automated approach to the task would save both time and effort. In this work, several different models to automatically tag forum posts on Stack Overflow were developed, featuring convolutional neural networks (CNNs) with and without skip connections and long short-term memory networks (LSTMs). Our final model, a 10-hidden-layer convolutional neural network, achieved a top-K categorical accuracy of 0.89, an overall accuracy of 0.70, and a precision of 0.69, far outperforming our baseline model.

1 Introduction

Community question-and-answer sites are a fixture of numerous modern-day industries, including the computer science field. For easier browsing, such sites often request users to tag their posts with the topic they encompass, allowing those with strong expertise in the field to address the questions they are best suited to answer and those who are interested in a particular topic to quickly view the posts that are most relevant to them. Although tagging may be straightforward for experienced users, the task may be challenging for new users. They may inadvertently tag their posts incorrectly, selecting tags that are incorrect or less relevant and causing more difficulties for both the responder and original poster. Therefore, an automated alternative is desirable, even if only manifested as a set of suggested tags for the user to consider as they tag their post.

We propose an automated topic-tagging solution for computer science–related data, considering forum posts from Stack Overflow, a popular QAS website frequented by both professional and amateur programmers, as our motivating example. Given the title and body of a post (passed in a single string), our model predicts the probability that the post should be labeled with each of the top n Stack Overflow tags. For the purposes of this project, we assigned $n = 10$, but our model is viable for larger values of n as well.

CS230: Deep Learning, Autumn 2020, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Related work

Over the last decade, there have been numerous papers on the subject of automated topic tagging. As far as the authors are aware, the first to address the task of topic-labeling was published in 2007 by Mei et al. [1], which utilized a knowledge graph to generate candidate labels and two centrality measures (closeness centrality and betweenness centrality) to rank the labels in order of likeliness. More recently, a number of papers have focused on the task of automatically tagging Stack Overflow posts using a variety of approaches. Earlier work includes papers by Robinson and Guo at Google Cloud, using a Bag of Words approach [2], and by Adinarayanan at Amrita University, utilizing a hybrid Multinomial Bayes and Support Vector Machine (SVM) system [3]. Other approaches have included Random Forests (RF), [4], Naïve Bayes and Ridge Classifiers [5], and fine-tuning BERT [6]. Of these, it seems that training the classifier using BERT yields the best results (approximately 87+% accuracy), although it is worth noting that the model was not published in an academic journal, and thus it is difficult to designate it as state-of-the-art.

For this project, we designed several lightweight models that utilize word2vec embeddings trained on 15GB of Stack Overflow posts [7]. The embeddings were static, and as such, not fine-tunable. Drawing on the embeddings as an input layer, we experimented with a Bag of Words approach using logistic regression, convolutional structures (with and without skip connections, added when the number of convolutional layers was sufficiently high), and long short-term memory networks (LSTMs). Generally, we found that for this particular multilabel classification task, our highest-performing CNN architectures were able to outscore our highest-performing LSTM architectures in terms of topK categorical accuracy, overall accuracy, precision, and F1 score. However, one of the LSTMs was able to outperform the convolutional models in the metric of AUPRC. On average, skip connections did not significantly increase or decrease performance. Finally, for one of our models, we tried supplementing our CNN/LSTM model for natural language with a CNN model for programming language recognition based on GitHub's OctoLingua model; however, we were only able to accomplish this for posts that contained one of the eight OctoLingua languages.

3 Dataset and Features

3.1 Dataset

Dataset details: We used the Kaggle Stack Overflow dataset [8], taken from the Stack Exchange archive and updated on a quarterly basis. The dataset is a CSV with multiple columns (`Id, Tags, OwnerUserId, CreationDate, ClosedDate, Score, Title, Body`), from which we extracted the `Tags, Titles`, and `Post Questions` columns (using the Pandas CSV Reader). The posts were on average 592 words in length, but posts longer than 200 words were truncated to only contain the first 200 words to avoid making the embedding matrix excessively large.

Number of Examples: We tried a number of train-test splits and found that an 80% –20% split was optimal. As we trained on 53K examples, we had a total of approximately 42K train-set examples and approximately 11K test-set examples.

3.2 Preprocessing and Embeddings

For the majority of our CNN, LSTM, and logistic regression models, the title and the non-code portion of each post (concatenated into a single string) served as the models' only input. Specifically, we used a series of `regex` expressions to remove text enclosed within the `<code> ... </code>` tags (denoting the embedded code segments of each post) and extraneous characters such as line breaks. The remaining words were assigned an index using Keras' `TextVectorization` feature, and we constructed a matrix mapping from each word to its corresponding embedding using static Word2Vec embeddings, pre-trained on Stack Overflow posts [9]. As for the labels, we only considered the top 10 tags (sorted in order of decreasing frequency) as the possible set of labels for our model to predict. A list of the top 10 tags on our training dataset with their respective number of occurrences can be found in **A.1**.

Using Keras' `MultiLabelBinarizer`, each post was labeled with a 10-dimensional multi-hot vector, with each index assigned a value of 1 or 0 depending whether the corresponding tag was present or not. Posts that had none of the top 10 tags were assigned a vector consisting of zeroes. For one of our

models (which ensembled the result from two CNNs, one trained on natural language and the other on code), we based the code-classifying CNN on GitHub's OctoLingua model and thus trained the model on the Rosetta Code repository, a dataset featuring 8 different GitHub programming languages. The languages of the training distribution is also included in the appendix.

To pre-process this dataset, we removed a percentage of file extensions from our training data at the training step to encourage the model to learn from the vocabulary of the files (and not overfit on the file extension feature, which is highly predictive). In addition, we converted the text into lowercase and removed all special characters.

3.3 Sample input & output

Input:

Title	Body
how can i handle safari system alert ...?	I wanted to open my app without safari system alert but I found out that is impossible ...

Output:

```
c#              0.078
java            0.035
.net            0.068
php             0.025
asp.net         0.102
javascript      0.048
c++             0.071
jquery          0.029
iphone          0.103
python          0.099

Most likely tag: iphone
```

Figure 1: Sample input & output

4 Methods

As earlier discussed, the majority of our models took only the natural language portion of each post into consideration, with the exception of one of our models, which comprised two CNNs, one for processing regular text and one (based on the OctoLingua model) for processing code.

4.1 Models

Overall, we experimented with eleven different models: one logistic regression baseline that used a Bag of Words approach to average the inputs, six CNNs without skip connections (featuring 1-6 convolutional layers, respectively), one six-layer CNN with a skip connection, one two-layer LSTM, one model consisting of two ensembled CNNs, and one five layer BERT model (which was not one of our final models). Diagrams and full details can be found in the appendix.

4.2 Loss Function

As this is was a multi-label classification problem, we used binary cross-entropy, which averages the log loss for each of the N labels.

$$L = -\frac{1}{N}\sum_{i=1}^{n} y_i log(\hat{y_i})$$

In addition, we did attempt to use Kullback–Leibler divergence, which minimizes the dissimilarity between two distributions (i.e., the fitted distribution and the actual distribution):

$$L = -\frac{1}{N}\sum_{i=1}^{n} p_i log(\frac{p_i}{q_i})$$

Unfortunately, Kullback–Leibler divergence caused the loss function not to decrease past several epochs; consequently, we did not include it in our final model.

4.3 Regularization

To combat the overfitting problem, we experimented with varying numbers of Dropout layers, eventually settling on two Dropout layers with $p = 0.5$. Additionally, we investigated adding both L1 and L2 regularization. As expected, both L1 regularization significantly reduced the overfitting problem–the train and test sets were essentially of equal performance. However, in both of these cases, the model was not able to properly converge and exhibited extremely small F1 and precision scores; therefore, we did not include either of these approaches in our final model.

Model	Batch	LR	BCE Loss	topK (k=2)	Accuracy	AUPRC	f1	Precision
CNN-6 (L1): TRAIN	64	1E-04	0.1444	0.6574	0.4486	0.5671	0.4016	0.6875
CNN-6 (L1): TEST	64	1E-04	0.1388	0.6627	0.4516	0.5939	0.4521	0.6989
CNN-6 (L2): TRAIN	64	1E-04	0.1361	0.7422	0.4349	0.7340	0.5820	0.5764
CNN-6 (L2): TEST	64	1E-04	0.1130	0.6570	0.4516	0.7244	0.6157	0.7530
CNN-6: TRAIN	64	1E-04	0.0316	0.9757	0.8525	0.9462	0.9047	0.9254
CNN-6: TEST	64	1E-04	0.2166	0.8943	0.6977	0.6515	0.6196	0.7088

Table 1: Overall performance (n in CNN-n/LSTM-n denotes the # of convolutional/ LSTM layers)

4.4 Optimizer

We trained using the RMSprop algorithm, which is designed to reduce the vanishing/exploding gradient problem via normalization/exponentially weighted averages and has been applied to text classification with reasonable success [11]. We also experimented with Adam (meant to combine the advantages of RMSprop & momentum) but received poor results and excluded it from the final model. RMSprop formula:

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$
$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db^2$$
$$W = W - \alpha \cdot \frac{v_{dw}}{\sqrt{v_{dw}} + \epsilon}$$
$$b = b - \alpha \cdot b - \alpha \frac{db}{\sqrt{v_{db}} + \epsilon}$$

5 Experiments/Results/Discussion

The results from our ten experiments (logistic regression, seven CNNs, an LSTM, and an OctoLingua-based model) are recorded in Table 2. In terms of overall trends, an increase in the number of convolutional layers correlated with an increase in topK accuracy, suggesting a more complex architecture allowed the models to better predict this metric. Interestingly, the increase in topK accuracy did not correlate with a lower overall binary cross-entropy loss. Perhaps this means that the models with large numbers of convolutional layers learned how to better predict the likeliest tags but did not learn to reduce the probability of the tags that were more unlikely. It's also interesting to note that while the CNNs vastly outperformed the LSTMs in most statistics, such as topK accuracy, accuracy, and precision, the best AUPRC performances came from LSTMs. Perhaps this suggests that the LSTMs were superior in recall and tended to err on the side of over-predicting tags.

Overall, although the LSTM-based model was predicted to exhibit better performance due to its ability to process sequential data, it seems that the CNN network was able to more effectively capture the low-level temporal relationships required for this task. Perhaps this was because LSTMs are more likely to suffer from the vanishing gradient problem than CNNs (although they certainly exhibit less of this problem than non-gated RNNs); indeed, we did observe that at multiple points during training, the LSTM network failed to improve its binary cross-entropy loss after only several epochs. That said, it is worth noting that we were able to perform a far greater number of experiments with convolutional

Model	Batch	LR	BCE Loss	topK (k=2)	Accuracy	AUPRC	f1	Precision
LOG. REG.	64	1E-3	0.1444	0.6574	0.4486	0.5671	0.4016	0.6875
	64	1E-4	0.1439	0.6475	0.4421	0.5696	0.4067	0.6848
	32	1E-3	0.1391	0.6553	0.4452	0.5925	0.4513	0.6990
	32	1E-4	0.1388	0.6627	0.4516	0.5939	0.4521	0.6989
CNN-1	64	1E-3	0.1182	0.7917	0.5035	0.7297	0.6346	0.7446
	64	1E-4	0.1086	0.7570	0.4975	0.7111	0.5682	0.7480
	32	1E-3	0.1315	0.8056	0.5311	0.7086	0.6188	0.7309
	32	1E-4	0.1044	0.7871	0.5223	0.7274	0.5933	0.7666
CNN-2	64	1E-3	0.1792	0.7912	0.5596	0.6928	0.5873	0.7718
	64	1E-4	0.1138	0.7573	0.4850	0.6961	0.5305	0.7423
	32	1E-3	0.1566	0.8496	0.5319	0.6818	0.5832	0.7112
	32	1E-4	0.1093	0.7663	0.5114	0.7144	0.5661	0.7500
CNN-3	64	1E-3	0.2290	0.8831	0.6176	0.6643	0.5871	0.7279
	64	1E-4	0.1150	0.6603	0.4302	0.7016	0.5626	0.7236
	32	1E-3	0.2100	0.8423	0.5699	0.6601	0.5582	0.7677
	32	1E-4	0.1130	0.7473	0.4702	0.7099	0.5724	0.7246
CNN-4	64	1E-3	0.1517	0.8455	0.6151	0.6973	0.6345	0.6901
	64	1E-4	0.1184	0.7287	0.4595	0.6986	0.5546	0.7529
	32	1E-3	0.1757	0.8791	0.5850	0.6759	0.6141	0.6738
	32	1E-4	0.1266	0.8008	0.5342	0.7067	0.5937	0.7313
CNN-5	64	1E-3	0.1603	0.8389	0.5508	0.6790	0.6061	0.6564
	64	1E-4	0.1307	0.7344	0.4208	0.6550	0.5254	0.7126
	32	1E-3	0.1774	0.8727	0.6407	0.6660	0.6224	0.6936
	32	1E-4	0.1315	0.8007	0.5490	0.6903	0.5940	0.7060
CNN-6	64	1E-3	0.1565	0.8622	0.5989	0.6550	0.6088	0.6875
	64	1E-4	0.2166	0.8943	0.6977	0.6515	0.6196	0.7088
	32	1E-3	0.1846	0.8879	0.6733	0.6568	0.6121	0.6971
	32	1E-4	0.2975	0.8801	0.6947	0.6217	0.6087	0.6759
CNN-6 + SKIP	64	1E-3	0.1726	0.8668	0.5861	0.6764	0.5820	0.5764
	64	1E-4	0.2176	0.8824	0.6764	0.6513	0.6157	0.7530
	32	1E-3	0.1739	0.8671	0.6669	0.6613	0.4152	0.6833
	32	1E-4	0.2857	0.8662	0.6810	0.6171	0.4118	0.6766
LSTM-2	64	1E-3	0.1361	0.7422	0.4349	0.7340	0.5820	0.5764
	64	1E-4	0.1130	0.6570	0.4516	0.7244	0.6157	0.7530
	32	1E-3	0.1389	0.7125	0.5371	0.5371	0.4152	0.6833
	32	1E-4	0.1576	0.5340	0.3394	0.4984	0.4118	0.6766

Table 2: Overall performance (n in CNN-n/LSTM-n denotes the # of convolutional/ LSTM layers)

architectures than those with LSTM architectures due to their relative speed of computation, and perhaps if we had performed more, we would have been able to produce a better result using an LSTM network. Another interesting trend to note is that the skip connections did not significantly change performance, providing neither a positive nor negative to most statistics. It seems that including a single skip connection was not sufficient to significantly impact the overfitting problem; perhaps more would have made a greater difference. Interesting, the two LSTMs with batch size 64 and the logistic regression baseline did succeed in the aspect of reducing overfitting, evidenced by the fact that their train and test scores were the closest (for brevity, the training set performance has been omitted from the above table). However, this ostensible bonus was undercut by the fact that neither achieved particularly good results with most statistics, with the exception of the two LSTMs with batch size 64 that were able to achieve the highest AUPRC scores. Finally, error anlysis was performed on a number of models; the confusion matrices are included in the appendix.

6 Conclusion/Future Work

Among all our models, the six–convolutional layer network achieved the best results, with a top-K categorical accuracy of 0.89, an overall accuracy of 0.70, and a precision of 0.69, far outperforming our baseline model.

Currently, our model predicts only predicts the top 10 tags and thus typically predicts tags on programming languages, APIs, operating systems, etc. However, to detect more specific tags (e.g., particular algorithms), we would like to integrate Code2Vec embeddings [12], a set of embeddings trained on 100M Java functions that returns a function's name given its function. Integrating these embeddings would likely augment the model's ability to predict tags outside of the top 50. If we had more time, we would have liked to experiment with a greater number of model architectures. An attention model would have likely increased our LSTM model's performance, and ResNets would have likely helped us reap the benefits of residual connections better than our single skip connections did.

7 Contributions

All three authors created the video and helped writing the report.

Ryan Chi: performed dataset preprocessing and cleaning, drafted the original model, tuned the hyperparameters, created the baseline, CNN, and LSTM models, created the LaTeX data tables, formatted the data visualizations.

Varuni Gupta: worked on implementing the LSTM model, Octolingua-based model, Ensembling model, coming up with the Top-K categorical metric, analyzing the predictions and accuracies of the models.

Ankur Agrawal: worked on implementing the BERT model, worked on setting up the AWS instance and programming environment on AWS, worked on fine tuning the CNN model's memory requirements to make it run on the entire stack overflow dataset.

7.1 Acknowledgements:

We are grateful to Jonathan Li for providing valuable insights on model architecture and deep learning strategy in general and to Shahab Mousavi for his feedback on our proposal. In addition, we would like to thank Ethan Chi and Shrey Gupta for their thoughtful advice throughout the course of our project. Finally, we would like to recognize Amazon AWS for sponsoring our project and Google Colaboratory for their free cloud services.

8 Code

Our model can be found at https://github.com/ryanachi/topic-tagging

References

[1] Mei, Qiaozhu, et al. "Automatic Labeling of Multinomial Topic Models." ACM, 12 Aug. 2007, sifaka.cs.uiuc.edu/czhai/pub/kdd07-label.pdf.

[2] Robinson, Sara, and Yufeng Guo. "Predicting Stack Overflow Tags with Google's Cloud AI." Stack Overflow Blog, 22 July 2019, stackoverflow.blog/2019/05/06/predicting-stack-overflow-tags-with-googles-cloud-ai/.

[3] Adinarayanan, Smrithi Rekha. "A Hybrid Auto-Tagging System for StackOverflow Forum Questions." International Conference on Interdisciplinary Advances in Applied Computing, 10 Oct. 2014.

[4] V. Jain and J. Lodhavia, "Automatic Question Tagging using k-Nearest Neighbors and Random Forest." *2020 International Conference on Intelligent Systems and Computer Vision (ISCV)*, Fez, Morocco, 2020, pp. 1-4, doi: 10.1109/ISCV49265.2020.9204309.

[5] Li, Susan. "Auto Tagging Stack Overflow Questions." *Auto Tagging Stack Overflow Questions*, 13 Mar. 2018.

[6] Kumar Anmol. "Stack Overflow EDA + BERT model Accuracy: 87.6". https://www.kaggle.com/anmolkumar/stack-overflow-eda-bert-model-accuracy-87-6.

[7] Efstathiou, V., Chatzilenas, C., Spinellis, D. 2018. "Word Embeddings for the Software Engineering Domain". In *Proceedings of the 15th International Conference on Mining Software Repositories.* ACM.

[8] Kaggle. "StackSample: 10% of Stack Overflow QA". https://www.kaggle.com/stackoverflow/stacksample.

[9] Ganesan Kavita. "C or Java? TypeScript or JavaScript? Machine learning based classification of programming languages". https://github.blog/2019-07-02-c-or-java-typescript-or-javascript-machine-learning-based-classification-of-programming-languages/

[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. "code2vec: Learning Distributed Representations of Code." In *Proc. ACM Program. Lang.* 3, POPL, Article 40 (January 2019), 29 page

[11] Libraries: Gensim, Keras, Matplotlib, Numpy, Pandas, Seaborn, Scikit-learn, Sklearn, Tensorflow, Tqdm, Transformers

A Appendix

A.1 Dataset Details

Tag	Occurrences
C#	6722
JAVA	3858
.NET	3598
PHP	3223
ASP.NET	3041
JAVASCRIPT	2852
C++	2509
JQUERY	2198
IPHONE	2111
PYTHON	2070

Table 3: Top 10 tags in the Stack Overflow dataset

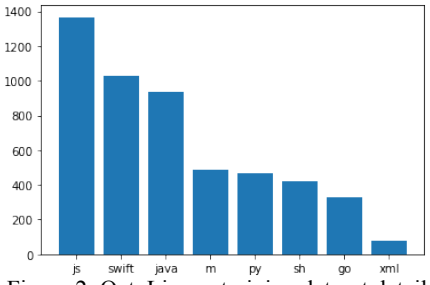


Figure 2: OctoLingua training dataset details

A.2 Model Architecture

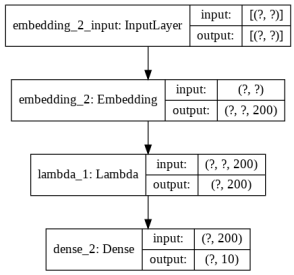


Figure 3: Baseline model

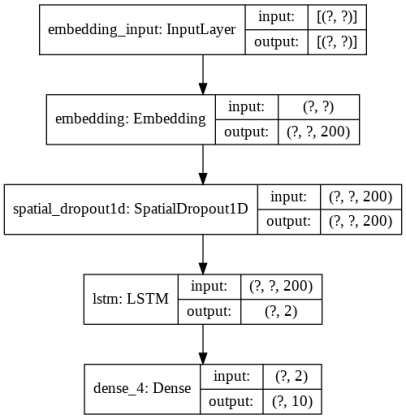


Figure 4: LSTM-2 model

```
Layer (type)                 Output Shape              Param #
=================================================================
input_word_ids (InputLayer)  [(None, 128)]             0
_________________________________________________________________
tf_bert_model (TFBertModel)  ((None, 128, 1024), (None 335141888
_________________________________________________________________
tf_op_layer_strided_slice (T [(None, 1024)]            0
_________________________________________________________________
dropout_75 (Dropout)         (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 15)                15375
_________________________________________________________________
activation_5 (Activation)    (None, 15)                0
=================================================================
Total params: 335,157,263
Trainable params: 335,157,263
Non-trainable params: 0
_________________________________________________________________
```

Figure 5: BERT model (not included as one of our final models due to not having enough GPU memory to train the model at a reasonable pace / large enough batch size)

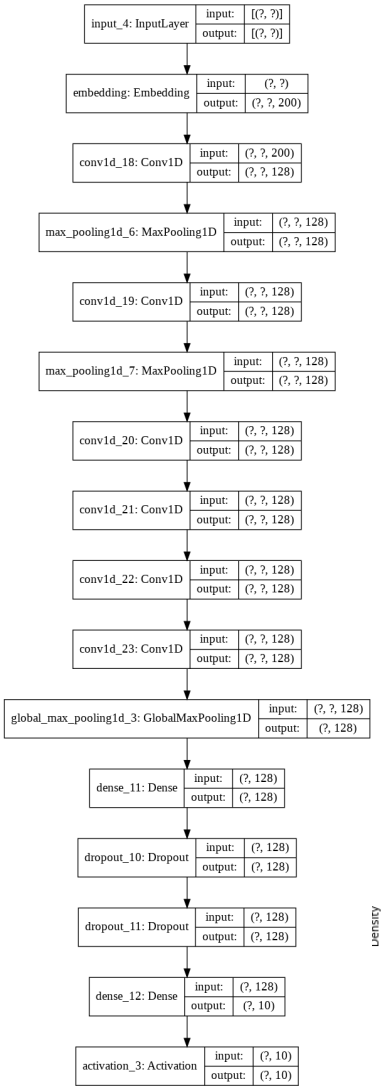


Figure 6: CNN-6 model (6 convolutional layers)

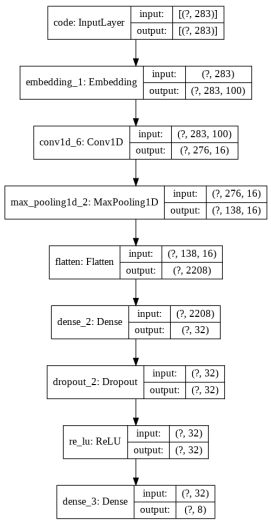


Figure 7: OctoLingua-based model

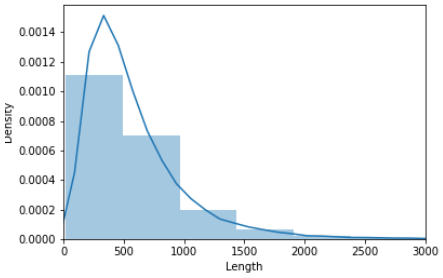


Figure 8: word count of Stack Overflow dataset posts

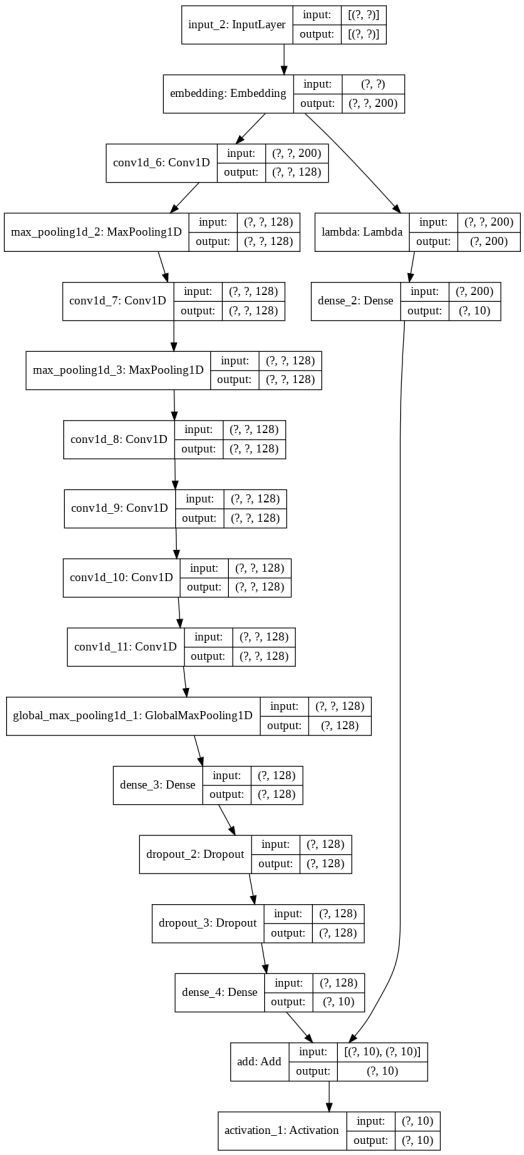


Figure 9: CNN-6 model
with a single skip connection

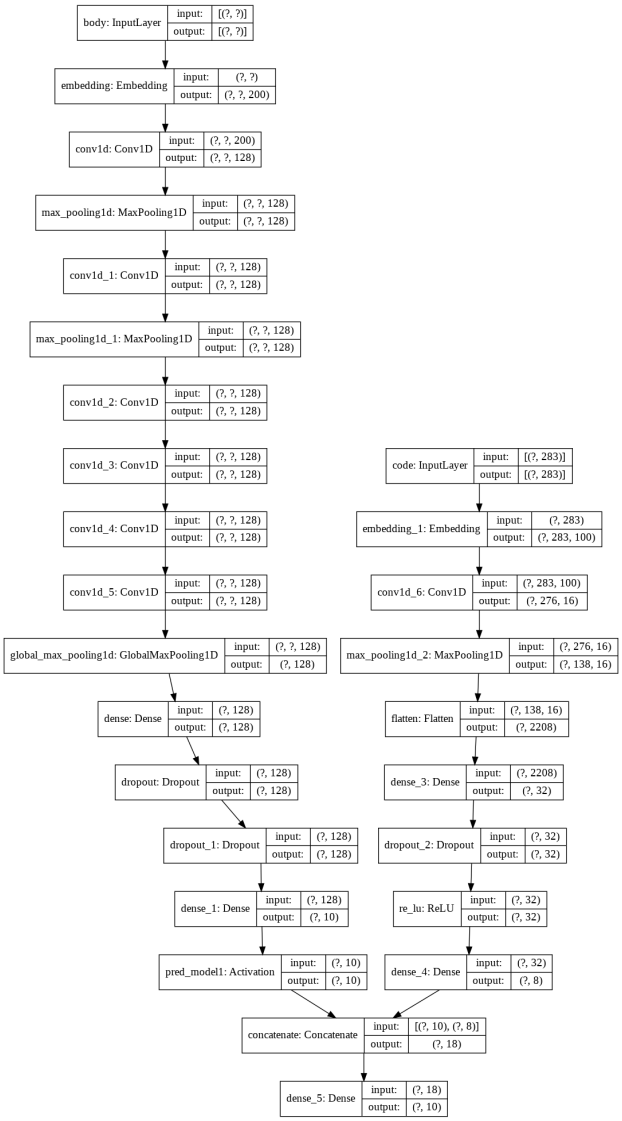


Figure 10: CNN model with
OctoLingua model

A.3 Experiments

```
Epoch 1/20
666/666 [==============================] - 488s 732ms/step - loss: 6.2356e-05 - accuracy: 0.1209 - top_k_categorical_accuracy: 0.6967 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 2/20
666/666 [==============================] - 491s 738ms/step - loss: 6.1887e-05 - accuracy: 0.0642 - top_k_categorical_accuracy: 0.6966 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 3/20
666/666 [==============================] - 475s 714ms/step - loss: 6.1600e-05 - accuracy: 0.0581 - top_k_categorical_accuracy: 0.7066 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 4/20
666/666 [==============================] - 462s 693ms/step - loss: 6.1254e-05 - accuracy: 0.0595 - top_k_categorical_accuracy: 0.6257 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 5/20
666/666 [==============================] - 464s 696ms/step - loss: 6.0955e-05 - accuracy: 0.0621 - top_k_categorical_accuracy: 0.6440 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 6/20
666/666 [==============================] - 460s 690ms/step - loss: 6.0942e-05 - accuracy: 0.0619 - top_k_categorical_accuracy: 0.6161 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 7/20
666/666 [==============================] - 457s 686ms/step - loss: 6.0775e-05 - accuracy: 0.0616 - top_k_categorical_accuracy: 0.6483 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 8/20
666/666 [==============================] - 459s 690ms/step - loss: 6.0680e-05 - accuracy: 0.0621 - top_k_categorical_accuracy: 0.6660 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 9/20
666/666 [==============================] - 462s 693ms/step - loss: 6.0947e-05 - accuracy: 0.0616 - top_k_categorical_accuracy: 0.6267 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 10/20
666/666 [==============================] - 463s 695ms/step - loss: 6.0876e-05 - accuracy: 0.0603 - top_k_categorical_accuracy: 0.6233 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 11/20
666/666 [==============================] - 470s 706ms/step - loss: 6.0713e-05 - accuracy: 0.0607 - top_k_categorical_accuracy: 0.6577 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 12/20
666/666 [==============================] - 486s 730ms/step - loss: 6.0700e-05 - accuracy: 0.0614 - top_k_categorical_accuracy: 0.6892 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 13/20
666/666 [==============================] - 486s 729ms/step - loss: 6.0594e-05 - accuracy: 0.0611 - top_k_categorical_accuracy: 0.6789 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 14/20
666/666 [==============================] - 490s 735ms/step - loss: 6.0477e-05 - accuracy: 0.0616 - top_k_categorical_accuracy: 0.7366 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 15/20
666/666 [==============================] - 479s 719ms/step - loss: 6.0456e-05 - accuracy: 0.0617 - top_k_categorical_accuracy: 0.7315 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 16/20
666/666 [==============================] - 487s 731ms/step - loss: 6.0168e-05 - accuracy: 0.0664 - top_k_categorical_accuracy: 0.7039 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 17/20
666/666 [==============================] - 480s 721ms/step - loss: 5.9834e-05 - accuracy: 0.0688 - top_k_categorical_accuracy: 0.7321 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 18/20
666/666 [==============================] - 466s 699ms/step - loss: 5.9665e-05 - accuracy: 0.0699 - top_k_categorical_accuracy: 0.7190 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 19/20
666/666 [==============================] - 464s 696ms/step - loss: 5.9518e-05 - accuracy: 0.0710 - top_k_categorical_accuracy: 0.7304 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
Epoch 20/20
666/666 [==============================] - 460s 690ms/step - loss: 5.9643e-05 - accuracy: 0.0701 - top_k_categorical_accuracy: 0.6843 - precision_m: 0.0000e+00 - f1_m: 0.0000e+00 - val_loss:
```

Figure 11: LSTM model training performance

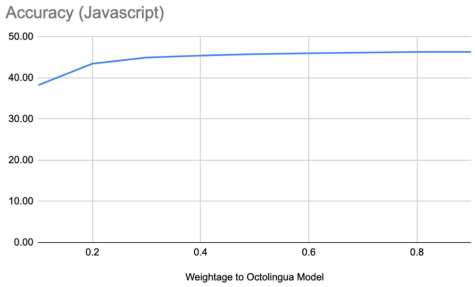


Figure 12: CNN + OctoLingua model performance on Javascript-tagged posts

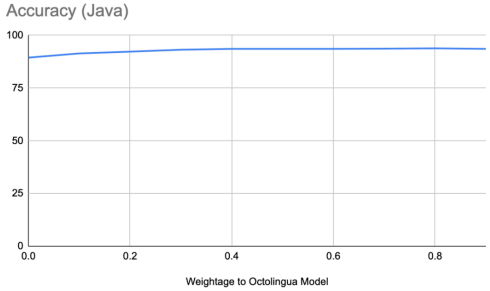


Figure 13: CNN + OctoLingua model performance on Java-tagged posts

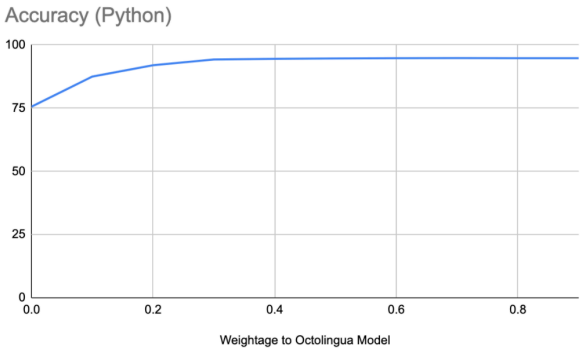
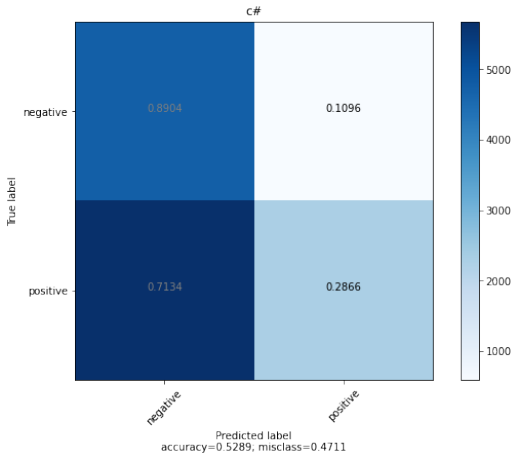
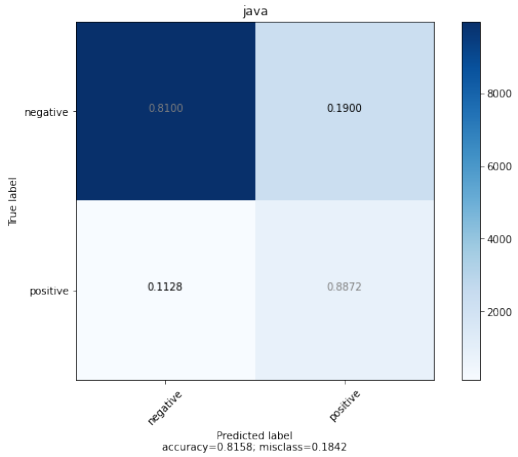


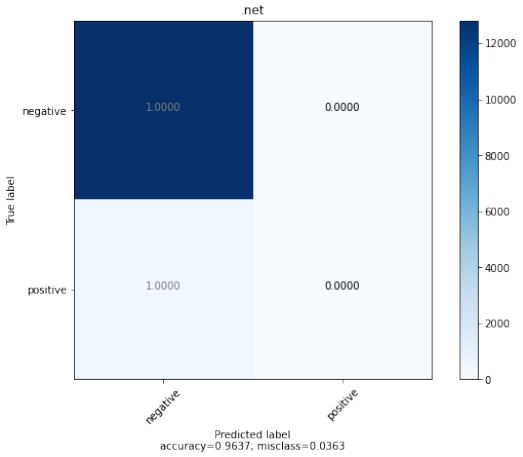
Figure 14: CNN + OctoLingua model performance on Python-tagged posts

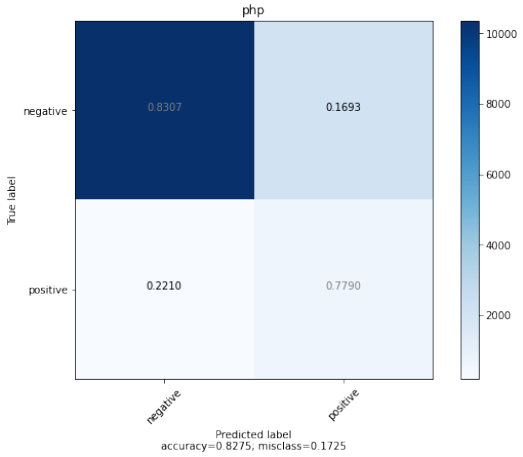
A.4 Error Analysis: Confusion Matrices

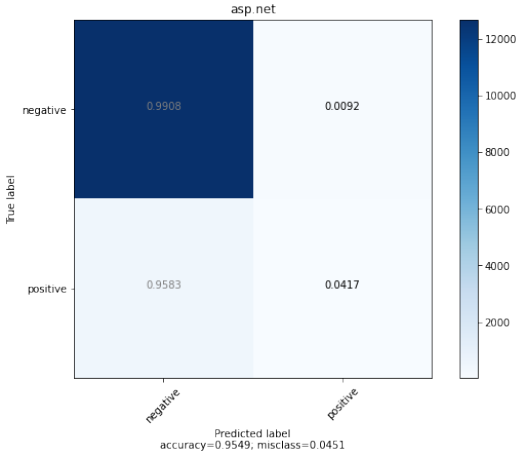
In figures 15-24, the CNN-6 model's performance on each of the top 10 tags. It seems that generally, the CNN model tended err on the side of underpredicting (rather than overpredicting) tags, as evidenced by the model's high rate of false negatives and slightly lower rate of true positives. In terms of specific tags, it seems that the model did surprisingly poorly on the C# tags—perhaps this is because during the pre-processing step, the punctuation removal caused the words "C#" and "C++" to become identical. To avoid this issue, we would consider using a more conservative tokenizer.

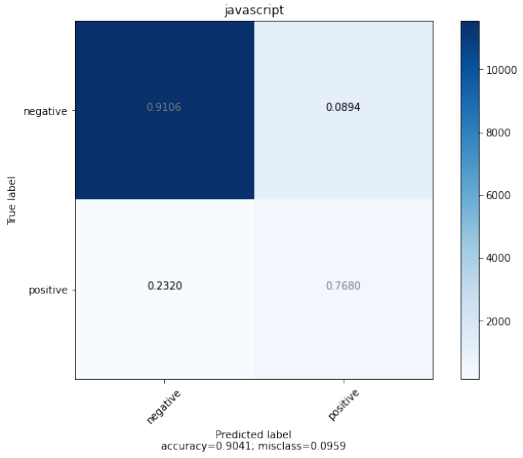


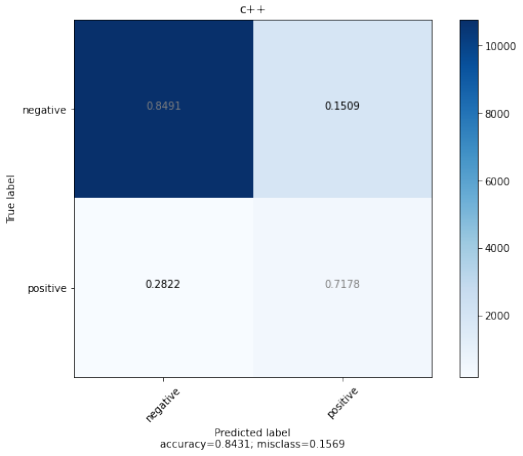


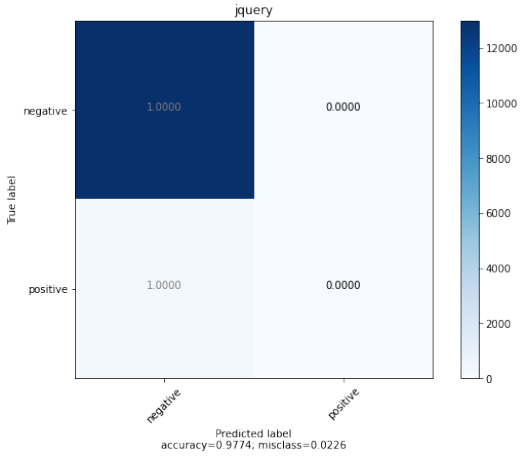

.net
negative
positive
True label
1.0000
0.0000
1.0000
0.0000
negative
positive
Predicted label
accuracy=0.9637; misclass=0.0363

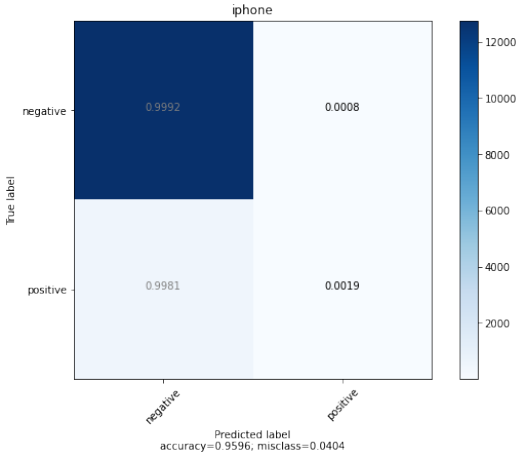

php
negative
positive
True label
0.8307
0.1693
0.2210
0.7790
negative
positive
Predicted label
accuracy=0.8275; misclass=0.1725

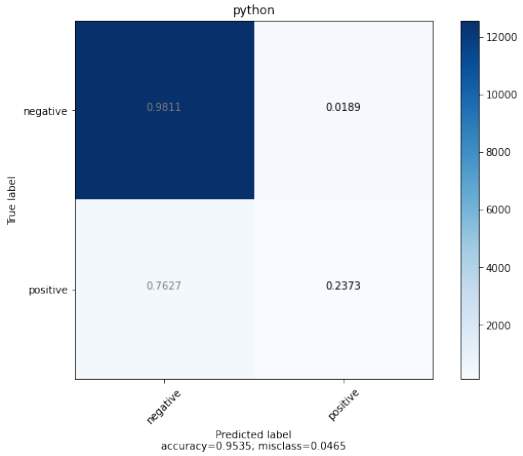

asp.net
negative
positive
True label
0.9908
0.0092
0.9583
0.0417
negative
positive
Predicted label
accuracy=0.9549; misclass=0.0451


javascript
negative
positive
True label
0.9106
0.0894
0.2320
0.7680
negative
positive
Predicted label
accuracy=0.9041; misclass=0.0959


c++
negative
positive
True label
0.8491
0.1509
0.2822
0.7178
negative
positive
Predicted label
accuracy=0.8431; misclass=0.1569


jquery
negative
positive
True label
1.0000
0.0000
1.0000
0.0000
negative
positive
Predicted label
accuracy=0.9774; misclass=0.0226


iphone
negative
positive
True label
0.9992
0.0008
0.9981
0.0019
negative
positive
Predicted label
accuracy=0.9596; misclass=0.0404


python
negative
positive
True label
0.9811
0.0189
0.7627
0.2373
negative
positive
Predicted label
accuracy=0.9535; misclass=0.0465