CSCI 467: Final Report - Sentiment Analysis with LSTM

Konstantinos Mylonas

1. Abstract

Sentiment Analysis is one of the early and well-known problems in Natural Langauge Processing (NLP). A big number of applications is interested in classifying text. Customer feedback applications for e-commerce and Social Media monitoring is only two out of many examples. A variety of methods that solve the problem efficiently have been proposed. In this project, the goal is to classify Tweets into one out of six emotions. The main methods utilised are Naive Bayes and an LSTM Neural Network. Finally, the LSTM's performance is compared with a fine tuned transformer. The LSTM outperforms Naive Bayes and surprisingly performs almost as good as the transformer on this task.

2. Introduction

This study addresses the challenge of sentiment analysis on English tweets, aiming to classify tweets into six distinct emotions: joy, sadness, fear, anger, surprise, and love. In this project, I explore the effectiveness of three distinct models: Naive Bayes, Long Short-Term Memory (LSTM) networks, and transformer-based models. Naive Bayes, a classical probabilistic model, serves as a baseline, providing insights into the performance of a simple yet interpretable approach. LSTMs, a special architecture of recurrent neural network (RNN), are known for their ability to capture long-term dependencies, making them suitable for tasks such as sequence classification. Finally, I delve into the cutting-edge transformer architecture, known for its attention mechanism and success in capturing contextual information across sequences. The machine learning framework that is used is PyTorch and the training is done efficiently on Google Colab GPU's. The necessary data and code are located at: Datasets Github Repo

3. Related Work

There have been several attempts of using LSTM and transformers for sentiment analysis. For example in (Wang et al., 2018), they compared the performance of LSTM with Naïve Bayes (NB) and Extreme Learning Machine (ELM). The results show that LSTM achieved better performance.
Another application of LSTM was to classify IMDB reviews to positive or negative (Murthy et al., 2020)

4. Dataset and Evaluation

4.1. Dataset

The dataset that I use is found on Huggingface datasets (Saravia et al., 2018). It consists of 417k English Tweets, each one with a label that is one of: anger, fear, joy, love, sadness, and surprise. Huggingface offers some very handy functions to load, split and preprocess a dataset. However, for educational purposes I chose to download the whole dataset and handle the data on my own. I generated three subsets by randomly sampling the original dataset: one for training (train set), one for validation (dev set) and one for evaluating the final results (test set). The size of the datasets is 291766 (70%), 83778 (20%), and 41265 (10%) tweets respectively. The same splits were used for each model.
The dataset is highly imbalanced. Focusing on the train set, the number of examples per class in decreasing order is the following:

- Joy: 99,048 (33%)
- Sadness: 84,588 (29%)
- Anger: 40,185 (14%)
- Fear: 33,413 (12%)
- Love: 24,109 (8%)
- Surprise: 10,423 (4%)

They same statistics hold for the dev and test sets as well. The result is summarized in in Figure 1.

4.2. Evaluation

The most famous evaluation metric for classification task is accuracy. However, using accuracy for an imbalanced dataset could be missleading. For instance, for this dataset, if we had a model that never classifies a Tweet to "Surprise", but is in general good in terms of the rest of the classes, it would achieve approximately 95% accuracy, leading us to the conclusion that it is a good model. A remedy for this problem would be to undersample the majority classes and create a new, balanced dataset. This would indeed work, and the final model would treat each class equally. But if the

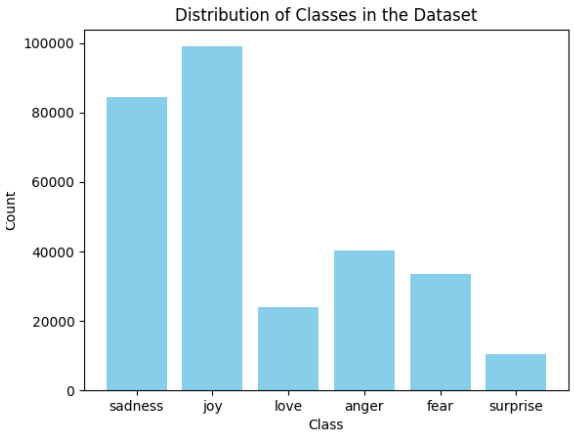

Figure 1. Number of examples per class in the train dataset

underlying task that we try to solve "is imbalanced", meaning that it is true that Surprise Tweets are less frequent, then we wouldn't want our model to assign the same importance to each class. For this reason I decided to continue with the imbalanced dataset and use a different evaluation metric than accuracy. In order to avoid the issues that raw accuracy can cause, I decided to use F1 score as an evaluation metric and choose the model that achieves that highest F1 score.

4.2.1. Weighted F1 score

In fact, I used a slightly modified F1 score than what is usually used, in order to be consistent with the following assumption for the specific task:

Assumption: Surprise Tweets are indeed less frequent, so the main focus of the model should be on the majority classes. Nevertheless, the model should not completely neglect the minority classes.

For this purpose the F1 score that I used, is a weighted average over the F1 score per class, with a slight tweak on the weights. The simple approach would have been to weight each class by its size, so the majority classes contribute to the average more than the minority classes. However, the weight of Surprise class would have been very small, as only 4% of the tweets are Surprise Tweets, and its contribution to the average would have been negligible. This could bring us to the previous problem that we had with accuracy. A model that does a terrible job at classifying suprise tweets, could be considered as a good model. On the other hand, if I treated each class equally, I would not have been consistent with the **Assumption**. Thus, I tried to find the sweet spot that respects the statistics of the problem, but smoothens them at the same time by giving a slight boost to the minority classes, and reduces the importance of majority classes by a small factor. The motivation comes from a technique that was used at Word2Vec (Mikolov et al., 2013) in order to increase the probability of sampling the less frequent words, while maintaining the probability of sampling the most frequent words relatively higher at the same time. Finally, the formula that I used for F1 score is:

Table 1. F1 weights

Class	$\alpha = 1$	$\alpha = 0.25$
sadness	0.28	0.20
joy	0.33	0.20
love	0.08	0.14
anger	0.13	0.16
fear	0.11	0.15
surprise	0.03	0.11

$$F1_{total} = \sum_{i=1}^{6} w_i \cdot F1_i$$

$$where\ w_i = \frac{count(i)^\alpha}{\sum_{j=1}^{6} count(j)^\alpha}$$

For the experiments I used

$$\alpha = 0.25$$

To be more precise, the weights with and without this technique are summarized in Table 1 and we can see that the desired outcome has been achieved.

5. Methods

5.1. Naive Bayes

Naive Bayes is used as a baseline. I reused much of the code from Homework 1 - Naive Bayes. As we know the two parameters of the model are

$$\pi, \tau$$

The probabilities are estimated with simple counting (plus some smoothing) on the training data. The Python's Counter is very useful for this purpose. The Laplacian smoothing value is

$$\lambda = 1$$

Naive Bayes F1 score is 0.802 and achieves an accuracy of 85% on the development set (similar results on the test set as well), which is not bad at all. However, as we discuss later, the neural network outperforms Naive Bayes in terms of both metrics.

5.2. LSTM

5.2.1. Architecture

The second model has been implemented from scratch, with the help of PyTorch and the training loop that we saw on Homework 2. It consists of a LSTM and two fully connected MLP layers. The LSTM is responsible to encode the sentence. The first of the linear layers takes as input the final hidden state of the LSTM. ReLU is used as a non-linearity and then a dropout with probability = 0.2 is applied. The activation is then passed to the second linear layer which generates the logits of the model. The loss function that is employeed is CrossEntropy.
There are many hyperparameters, like the size of the word embeddings, the dimensions of the hidden state, the number of layers (stacked on each other), the dropout probability and more. Another very important hyper parameter is the pretrained embeddings that are being used (GloVe in this case) and whether or not they are "*frozen*". Initially I tried to train my own embeddings. Then I used GloVe 6B - 50 dim (freeze) and things seem to work better. Next, I decided to take it one step further and use GloVe embeddings that were trained specifically on Twitter data and increase the dimensionality of word vectors (GloVe Twitter 27B 100 dim) but without any significant improvement. Last but not least, I fine tuned the 100 dim GloVe word vectors by training the model without freezing the embeddings so they were dynamically updated throughout the training process. This technique increased the accuracy from 0.91 to 0.93 and the F1 score from 0.90 to 0.92 (approximately). The final architecture that I used consists of a single LSTM layer, where the hidden state size is 32. The embedding size is 100 and the embeddings are not frozen. I also tried more complex architectures and the reasons why I chose this simple one is explained in the following sections.

5.2.2. Preprocessing

The data are sequences of words of different lengths. We know that a neural network cannot work like that. So there is an important pre-processing step that needs to be taken before training the model. That is, map the words to integer indices and pad the sequences so that they have equal length. The length of the padded sequences is equal to the length of the longest sequence. There is definitely some margin for optimization here, since in many cases we don't need the whole sentence in order to understand the meaning. So some large sequences could be truncated and we could eventually end up with smaller sequence length. However up to this point, this way of dealing with the problem does not seem to add a significant overhead. Moreover, PyTorch methods like *pack_paddded_sequence* are utilized to efficiently deal with padded sequences and avoid processing padded time steps that do not contain meaningful data.

Table 2. Choose the size of hidden state

Model	Hidden	Layers	CE Loss	Dev F1
LSTM1	32	1	0.10994	0.92166
LSTM2	64	1	0.10497	0.91870
LSTM3	64	2	0.10351	0.91435
LSTM4	128	1	0.10837	0.92082
LSTM5	128	2	0.11032	0.92008

5.3. Transformer

As a third model I fine tuned the pre-trained DistilBERT for sequence classification (Sanh et al., 2019), which is a smaller (40% fewer parameters) and faster (60% faster), yet almost as efficient as the original BERT model according to the authors of the paper. The reason I chose this model is for training efficiency. The authors recommend to fine-tune the model for 4 epochs. The final setting that I used is: 4 epochs, 128 batch size, and 5e-5 learning rate. Due to limited time and resources, the transformer was not fine tuned on the entire dataset. Instead I used smaller subsets that are large enough to represent the dataset. More precisely, the train set contained 100k tweets, the dev set 20k tweets and the test set 10k tweets. **REVIEW** Interestingly, the transformer did not present a significant improvement. It was more vulnerable to overfitting, which makes sense because it has a very large number of parameters compared to the small LSTM, which didn't overfit in any of the cases as we will see later. Nevertheless, I was able to extract the best setting with some simple form of early stopping.
For this model I used the Hugginface API to load the pre-trained model and preprocess the data with an adequate BERT-specific tokenizer, while the training loop is done with PyTorch.

6. Experiments

6.1. Hyperparameters

Table 2 shows the F1 socre of different LSTMs on the development set for different hyperparameters. Batch size was fixed and equal to 2048. Each configuration was used to train a model for 100 epochs. Dropout and some simple form of early stopping are utilized in order to avoid overfitting. As we can see from the training curves none of the models overfit the data.
The experiments were conducted on Google Collab Pro. GPU's were used to make the training process faster. Due to resources limitations, I could not try very complex models but as *Table 2* suggests, there is no significant improvement after some point.
Apparently, increasing the complexity of the model does not improve further the performance for this task, but this is something that I investigate further later on. We see

that even the simplest model LSTM1 has a satisfying CE loss and F1 score. The same is true about accuracy, which I kept monitoring even though I'm not taking it into account for choosing hyperparameters. In contrast, in the midterm report we could, see that LSTM1's accuracy was 90%, whereas more complex models were above 92%, a fact that can be considered as an improvement. However, here there is no further improvement with more complex models. I believe that the 100dim, dynamically trained, GloVe vectors provide enough expressivity so even simple models can do the job efficiently. In other words, the number of parameters of LSTM1 is already large enough to capture most of the information. One could think then that if this is true, then the more complex models should overfit. This sounds actually like a valid argument. In Table 2 we can see some small increase in the loss for more complex models but I can't confidently say that this is due to some slight overfitting as there are other factors of randomness in the process and the results are not 100% reproducible. According to the curves, however, none of the models appear to overfit, and this could be attributed to the dropout and the large size of the dataset. In order to check the validity of my previous point, however, I tried an even simpler model, with hidden size=8, lets call it LSTM0.

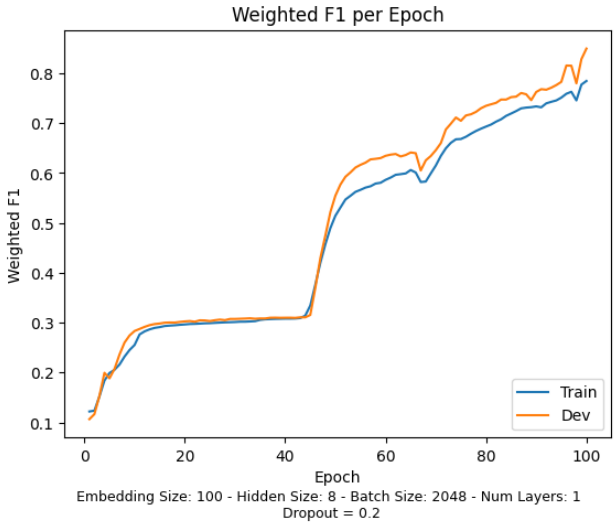

Figure 2. F1 Score for LSTM0

As we can see in Figure 2, the model is slowly learning and, suprisingly, gets pretty close to the more complex models. Moreover, the best F1 score on dev set is 0.84 and the loss 0.26. So its performance is indeed worse than the more complex models, but given its relative simplicity it performs pretty well. This result leads to the conclusion that given the 100 dim embeddings and their dynamic training, even simple models can perform well. Furthermore, the more complex models like LSTM4 and LSTM5 seem to be an overkill, whereas in the midterm report, where I used 50 dim frozen embeddings, it was more meaningful to use complex models because they demonstrated some improvement and LSTM1 was performing clearly worse. Having said that, I believe that for the specific task we don't need a model that is a lot more complex than LSTM1. For this reason I choose to continue with LSTM1 which achieves the highest F1 on dev score and has a satisfying loss, meaning that it is confident for its predictions, and is simple, yet good enough for our purposes. From now on whenever I refer to LSTM, I refer to LSTM1.

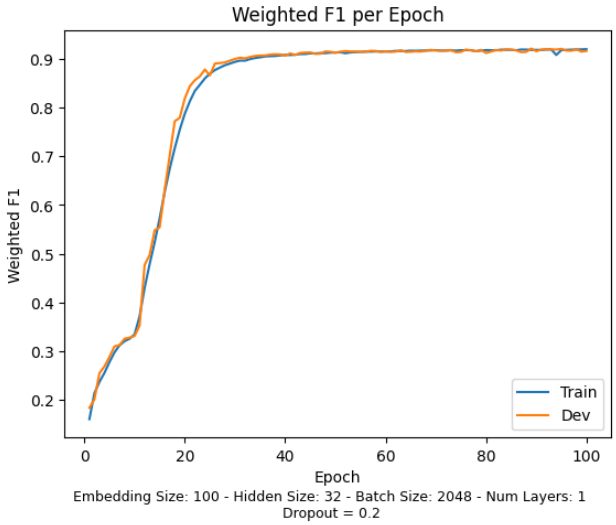

Figure 3. F1 Score for LSTM1

For completeness, Figure 3 and Figure 4 demonstrate the F1 Curve for LSTM1 and LSTM3 which is a relatively more complex model. As we can see, there is nothing that suggests that LSTM3 is better.
On some figures we see that Dev F1 is higher than Train F1 which could raise some concerns. This is due to the dropout, which makes the task "harder" during the training. Similar figures were generated for all of the models of Table 2, but they don't show anything interesting so I decided not to include them to keep space for further commenting.

6.2. LSTM vs. NB

Table 3 shows that LSTM outperforms Naive Bayes not only in terms of accuracy but F1 score as well. Especially in terms of F1, LSTM is significantly better which makes it clearly a better model. Naive Bayes is a simple and efficient model but it has its limitations. The Naive Bayes assumption assumes that the words of a sentence are independent from each other, given the label. We know that in reality this assumption does not hold and is an oversimplification of the

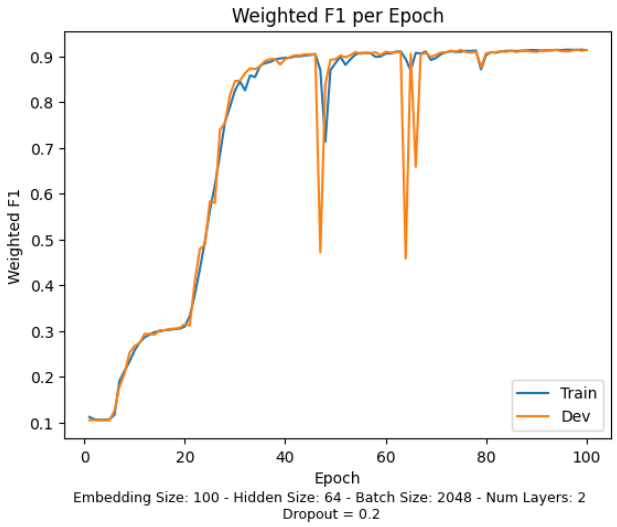

Figure 4. F1 Score for LSTM3

task. The meaning of a sentence is captured from it's words and the relationships between them.
On the other hand, LSTM is a special RNN architecture that allows the model to have some sort of memory of previous words. This feature effectively captures the context of a sentence and is the main reason why LSTM is a better choice for this task. Regarding NB, the confusion matrix (test set) is shown in Table 4 while the confusion matrix for LSTM is shown in Table 5. It is even more clear that NB suffers with the minority class Surprise, while LSTM does a great job. The tables are located at the very last page because they take a lot of space.

6.3. DistilBERT vs. LSTM

As mentioned earlier, the transformer does not significantly improve things either, as we would expect. Specifically, the transformer's best accuracy on the development set is 0.94261 and f1 is 0.92362. It is indeed slightly better, but not to the extent that we would expect. Regarding the performance on the test set the accuracy is 0.93920 and the F1 score is 0.92235. Showing the confusion matrix here would not be very insightful because as I mentioned earlier

Table 3. Compare Naive Bayes to the LSTM - Evaluation on test set

MODEL	ACCURACY	F1 SCORE
NB	85.520	0.80000
LSTM	93.515	0.91829
DISTILBERT	93.920	0.92235

the transformer is fine-tuned and evaluated on a smaller, yet representative dataset. Thus, it would not be meaningful to compare the exact numbers.
I was actually impressed to see that the transformer doesn't outperform the LSTM. There are definitely some more things to do in order to see the maximum performance of this transformer but in my opinion such a famous, pretrained state of the art model should be able to beat the traditional LSTM without much effort. For this reason, I decided to do some extra digging and fine-tune the transformer again, for another 4 epochs, starting from the previous checkpoint (thanks to Hugginface API saving and loading pretrained models is very easy). The train accuracy and F1 climbed to 98% while the performance on the development was slightly worse. This means that the transformer begun overfitting the data (finally, a model overfits this dataset :)!!!). After those results, I am now more confident about my initial point: the trained embeddings along with a relatively simple LSTM have enough expressivity power to tackle this problem efficiently. Finally, to support this argument even more, I measured the average length per tweet which is 20 words and the standard deviation which is around 11 words. This means that the sequences are relatively small. The main advantage that the transformers have over the LSTMs, is their ability to capture long term dependencies (due to their attention mechanisms). But for this scenario we don't really have very long sequences. With that in mind, seeing comparable performance between the two models sounds more realistic.

6.4. Freeze vs. Unfreeze Word Embeddings

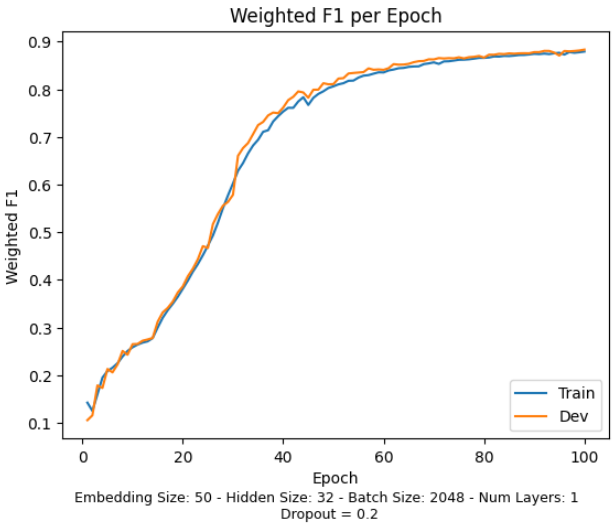

Figure 5. F1 Score for LSTM1 with frozen embeddings

In Figure 5 we see the performance of LSTM1 with frozen

50 dim word embeddings. The F1 score on the dev set was 0.88332 which is clearly worse than the dynamically trainned embeddings. Turns out that training the embeddings leads to great improvement

7. Discussion

7.1. Analyze Errors

The previous results don't suggest a good reason to change our focus from the LSTM, which was the initial proposal for this project. Instead they increase the curiosity to investigate even further how this simple LSTM is almost as good as a pretrained transformer. Next I do some manual error analysis by printing 10 Tweets where the LSTM was wrong. Some of them are the following:

7.1.1. Example 1

Text:Text: i suppose should make me happy but somehow i feel agitated and nervous trapped
True: anger
Pred: fear

7.1.2. Example 2

Text: Text: i still don t feel like i ve accepted it
True: joy
Pred: love

7.1.3. Example 3

Text: i am in good physical condition being able to keep up with hailey and having a solid energy base to get me through the day feels amazing
True: surprise
Pred: joy

7.1.4. Example 4

Text: i said feeling somewhat helpless what am i going to tell them
True: sadness
Pred: fear

I think this result is highly interpretable and clears any remaining "confusion". Most of the examples that the model fails to classify correctly are in fact very vague. One could say that they are bad examples. I believe that if we asked from 10 different humans to label those examples, we would get a variety of different assignments and a lot of them would actually do "the same mistakes" as the model. For example, in Example 1, I can't blame the model for predicting fear. It is a quite reasonable choice. The same is true for the rest of the examples. Why is Example 2 joy? It is not clear at all.

8. Conclusion

In conclusion, I believe that I now have have some solid understanding of why things work the way they work for this problem specifically. I started with Naive Bayes as a baseline and imporived it with LSTM, especially in terms of F1. In my opinion, the way that I measured F1 is a good choice and is consistent with the nature of the task, in the real world. Generating a balanced dataset would have made the problem very simple.
Training the word embeddings has been proven to be very helpful. In fact this change (from midterm report) empowers simple models to achieve great performance on the task (like LSTM1), while more complex models are unable to improve things anymore. The same simple models perform clearly worse with frozen word embeddings.
To delve deeper into the details of the problem, I decided to employee and fine tune a famous pretrained transformer, because I was really confused about the fact that the more complex models failed to further improve the results. Surprisingly, even the transformer failed to outperform the simple LSTM1. With some further investigation, this was attributed to two reasons: 1. The sequences of our dataset are relatively small and 2. it looks like what the models get wrong is actually vague and subjective because even humans would disagree with some of the true labels of the examples that the model gets wrong. So maybe our models are already close to "optimal". They learned almost everything of what could be learned from this dataset. Finally, sentiment analysis is, especially for simple sequences such as brief tweets, a problem that is easy to solve with modern technology. Nevertheless, I found this project really educational and tried to make it as challenging as possible. I learned the internal details and challenges of training NLP models, such as padding sequences, pre-processing and mapping words to indices, packing, creating a vocabulary etc. Moreover, I had the chance to see in practice the benefits of training on GPUs, and lastly gained some experience with Hugginface and pre-trained models.

References

Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space, 2013.

Murthy, G. S. N., Allu, S. R., Andhavarapu, B., Bagadiand, M., and Belusonti, M. Text based sentiment analysis using lstm. *International Journal of Engineering Research and Technology (IJERT)*, 2020.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019.

Saravia, E., Liu, H.-C. T., Huang, Y.-H., Wu, J., and Chen, Y.-S. CARER: Contextualized affect represen-

tations for emotion recognition. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3687–3697, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1404. URL https://www.aclweb.org/anthology/D18-1404.

Wang, J.-H., Liu, T.-W., and Xiong Luo, L. W. An lstm approach to short text sentiment classification with word embeddings. In *The 2018 Conference on Computational Linguistics and Speech Processing*, 2018.

Table 4. NB - Confusion Matrix

ACTUAL/PREDICTED	SAD	JOY	LOVE	ANGER	FEAR	SURPRISE
SAD	11410	360	51	193	157	22
JOY	388	12732	392	92	111	40
LOVE	210	1192	2142	32	25	5
ANGER	460	216	37	4755	126	8
FEAR	412	219	11	200	3702	50
SURPRISE	186	383	16	37	344	549

Table 5. LSTM - Confusion Matrix

MODEL	SAD	JOY	LOVE	ANGER	FEAR	SURPRISE
SAD	11669	31	7	306	177	3
JOY	19	12644	966	46	11	69
LOVE	2	75	3523	2	1	3
ANGER	60	31	1	5371	139	0
FEAR	90	17	0	150	4074	263
SURPRISE	4	57	3	2	141	1308