Deep Learning-Based Sarcasm Detection

Rohan Bais (rbais@stanford.edu), Daniel Do (dktd@stanford.edu)
Stanford University
Department of Computer Science – CS230

***Abstract*—Natural Language Processing still has numerous fields to be explored and innovations to be found. We were interested in exploring machine learning for sarcasm detection in text, since sarcasm is a subtlety only noticeable in speech and barely noticeable in text. For this paper, we tried to perform this task with deep learning on a dataset of 1.3 million comments from Reddit, half of which are sarcastic and half of which are not. We built 3 models to detect sarcasm: standard neural networks, CNNs, and LSTM RNNs. After performing of hyperparameter search to some degree and training these networks, we found that LSTM-RNNs, as expected, have the strongest performance, followed by CNNs and lastly our standard neural network.**

I. Introduction

Machine Learning has evolved to an age where natural language processing is nowhere near uncommon, with sentiment classification and speech recognition being frequently employed in day to day life. With that said, it is not always easy. Language can be so subtle to the point where even humans can have trouble with classification tasks, even with a task like sentiment classification when sarcastic texts appear. Sarcasm is a facet of language that even humans can get wrong. Sarcasm is easy to spot with access to tone or way of speaking, which is why its never a problem to comprehend sarcasm in in-person conversations. It is when only the text is used that the task becomes significantly harder. With this difficult task in mind, we wanted to use machine learning to perform binary sarcasm classification on Reddit comments, to observe if such a task can even be possible. With general supervised machine learning methods, like SVM and Naive Bayes, it is can be difficult to handpick features for sarcasm detection. Thus, for our project, we leveraged neural networks to see if given a sequence of words whether or not neural networks can understand the features most important for sarcasm detection and generalize well to many sarcastic remarks. Our input to our algorithm is a sentence or comment and the output is a label on whether the comment is sarcastic or not. Using the input text, we create word embeddings and feed it directly into our 3 different algorithms, CNNs, RNNs and standard neural networks, to produce the binary label. With these varying types of neural networks, we hoped to see if the task is feasible across any network architecture.

II. Related Works

The body of work surrounding sarcasm detection clusters around classification using contextual features. Within this cluster, there exists a grouping of work around feature engineering, and work related to learning relevant features. A prominent example of feature engineering, Reyes and Rosso used a model integrating textual features and in particular the dimensions of representativeness and relevance, and their results proved to be "encouraging" for irony detection.[1] As for learning features, Bamman and Smith used non-linguistic features from Twitter such as author and audience information to gain a significant advantage over purely lexical analysis, with an ablation test proving that no single learned feature was crucial to the model.[2] Taking this approach further, Amir attempted to learn user embeddings through a model similar to training Word2Vec.[3] Their results proved to be better than the Bamman's state-of-the art model, which has test accuracy rate of 0.93 and F1-scores of 0.92, despite struggling with Twitters 1000 historical tweet trawling limit for users. All of these seemingly successful models have all used self-reported sarcasm on social media, scraping posts with either the "#irony" or "#sarcasm" hashtag. Although these datasets form a significant portion of tweets as a whole, they are not representative of sarcastic language on the internet as a whole. Kreuz defines the "principle of inferability" as the notion that speakers only use sarcasm when there exists a mutual understanding between the speaker and audience, which isnt the case when speakers have to self-identify sarcasm with social media tags.[4] Filatova alleviates this corpus representation problem by crowdsourcing sarcasm corpus generation through mTurk.[5]. These are all clever and they are certainly needed to build a proper sarcasm detection model; however, we are primarily interested in solely language-related features. Sarcasm does rely on context, but we want to explore how sarcasm detection works without any context of non-language-related features.

Although the focus in recent years have centered around contextual features, purely text-based features have proved also proved to be effective in distinguishing sarcastic comments from non-sarcastic comments. Joshi showed that various word embeddings such as GloVe and Word2Vec resulted in subtle F-score gains from previous word-based results, but ultimately concluded that word embeddings were not sufficient for optimal performance and that other features were necessary.[6] Our research is most similar to this project: our goal isnt to introduce novel methods, but to survey existing methods and perform an overarching analysis of effectiveness.

III. Dataset and Features

The dataset used for the project was pulled from Kaggle dataset "Sarcasm From Reddit". These 1.3 million comments in the dataset were collected by randomly scraping popular subreddits and their most popular posts, with the efforts of

Ofer and Khodak.[13] Each of the 1.3 million data points consisted of the comment, the time posted, the author, the subreddit, the parent comment (the one that the current comment replied to), and the up-vote and down-vote numbers. The data varies over many particular subreddits, such as /r/politics, /r/atheism, /r/gaming, /r/technology, and /r/nba. The dataset is balanced in that approximately half of the comments made are sarcastic and half are not sarcastic. For our methods we only made use of the actual comments and the actual label, since we are explicitly interested in seeing if words themselves are enough to detect sarcasm.

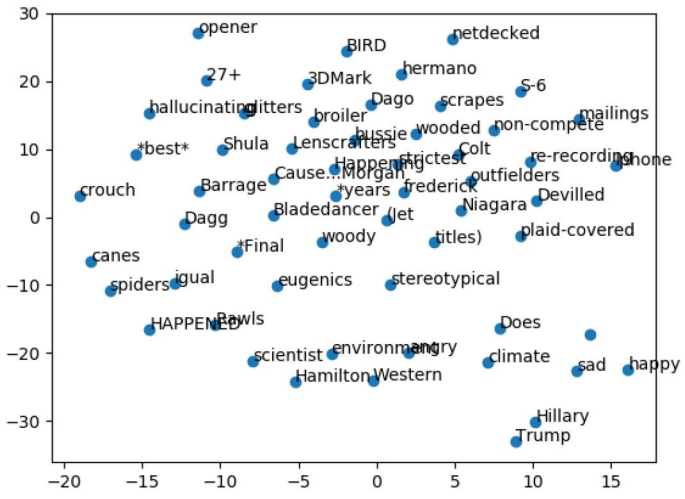

Fig. 1. Sample Word embeddings mapped in 2D. Words in similar contexts, like "Hillary" or "Trump", are close together

Having this vast amount of data, we used around 90 percent, 5 percent, and 5 percent of the datapoints for the training set, the dev set, and the test set respectively. Rather than use a one-hot vector of unigrams as our features; however, we opted to use the Word2Vec word-embedding library to properly capture the relationship between words in a high dimensional space. Word2Vec, as the name implies, converts words to vectors such that similarity measures are preserved in the space.[10] It also allowed us to capture the meaning of a sentence as a series of additions and subtractions on the produced individual word vectors. After preprocessing the sentences by removing the punctuation around some of the individual words, we fed the preprocessed sentences into gensim's Word2Vec library to directly train a Word2Vec model. The resulting Word2Vec model output a set of 100-dimensional word vectors, as visualized in figure (1).

For some of the methods, We were also interested in producing a TF-IDF weighting scheme to weigh important words more heavily in a sentence. TF-IDF is a numerical statistic that allows us to dictate the importance of a word in a document, or a sentence in our scenario. We thought using this statistic to amplify word vectors could better capture the meaning of a sentence through indicative and important words. It can be summarized with the following formula:

$$score_{w,d} = tf_{w,d} \times log\left(\frac{N}{df_w}\right)$$

Where $tf_{w,d}$ is the term frequency of word w in a document d, and N is the total number of documents and df_w is the total number of documents that contain a word w.

Term frequency ($tf_{w,d}$) measures the occurrence of the word in a document and inverse document frequency (idf or the logarithm term) measures the word's uniqueness to that particular document over all other documents. If a term is frequent in a single document but infrequent in other documents, the measure will be high since the word is deemed important as it occurs a lot in the document and it appears only in this or a few more documents. Common stop-words like "the" or "a" will have a high term frequency, but a low inverse document frequency due to the words being common to many documents. Using these particular combination of features, we then implemented various neural networks to classify sarcasm.

IV. Methods

For this task, we built 3 distinct types of models: standard neural networks, convolutional neural networks, and recurrent neural networks. For each of the methods we simply used the softmax cross-entropy loss:

$$L(\hat{y}, y) = \sum_i y_i log(\hat{y}_i)$$

While the loss function is similarly defined across these methods, the architecture and results of each of these three methods are quite different.

A. Baseline

For our baseline, we chose a standard 2-layer, 25 units each, neural network with around 0.2 probability of dropout per layer and a ReLu activation function per layer. For the feature vector for a single comment, we took all of the words and converted them to vectors using our embedding model and took the average across these word features as input to the model. Given the fact that Word2vec allows addition and subtraction across vectors to produce good analogies and better capture the meaning of the word, we hoped that an average of the vectors would be a good baseline for a standard neural network. Interested in seeing how effective TF-IDF is, we also created a weighted average such that we multiplied each word vector by its TF-IDF score to more heavily weigh important words. With these two feature vector generations, we passed them into the same model to see how they fare. For all i words in a sentence, the following two equations described our input vector to this model.

$$f_{avg} = \frac{1}{N}\sum_i w_i, \quad f_{weighted} = \frac{1}{N}\sum w_i \times tfidf(w_i)$$

Where f_{avg} represents the unweighted average word vector, $f_{weighted}$ represents the weighted average word vector, and $tfidf(w_i)$ represents the TF-IDF score of the word vector.

B. CNN

We were also interested in seeing how a convolutional neural network would perform in an NLP-related task such as this. Given that CNNs operate mostly on fixed images, we created a matrix input by stacking a comment's individual word embeddings horizontally as the rows of a matrix. This process is diagrammed in figure (2):

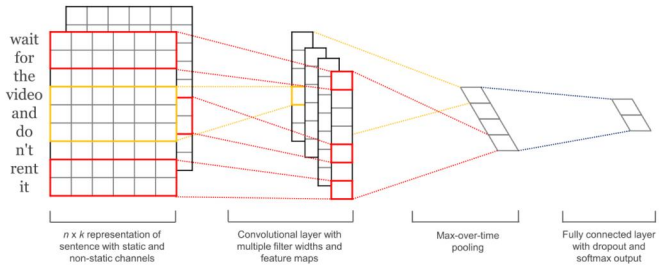

Fig. 2. Matrix of stacked word vectors is convoluted, concatenated, and fed into a fully connected layer for softmax output (courtesy of Peng.[11])

This figure maps out a 1-layer CNN that we used. Because the sentences can be of variable length, we padded the matrix so it is a 30 by 100 matrix, where 30 is the max number of words in a sentence, and 100 is the dimensionality of the word embeddings vectors, since they were stacked horizontally. With the image, we created 20 filters where 2 filters were of size n by 100 for $n = 1$ to 10. In other words, we created filters of the same column size to imitate a convolution of a window of n words for $n = 1$ to 10. After the convolutions we are outputted with variable size 1-dimensional vectors where we proceed to 1-max-pool each vector, concatenate the pooled results into a bigger vector of size 20, and connect this vector in a fully connected layer to output softmax probabilities.

C. LSTM-RNN

Another major interest was observing how an LSTM RNN network would perform on this language task. An RNN, recurrent neural network, is a model that utilizes internal memory to better handle sequential datasets. An LSTM model, long short-term memory model, is a specific type of RNN model designed to remember information in the model and utilize it over an unspecified time amount. Our LSTM model was straightforward, consisting of feeding in the matrix of sentence word embeddings through a default Keras LSTM consisting of 128 units, a dropout layer with probability .5, another default Keras LSTM layer consisting of 128 units, and then another dropout layer with probability 0.5. We passed the output of this dropout layer into a fully connected Keras Dense layer with two outputs, and then applied the softmax activation to these two outputs. An illustration of this model is shown in figure (3).

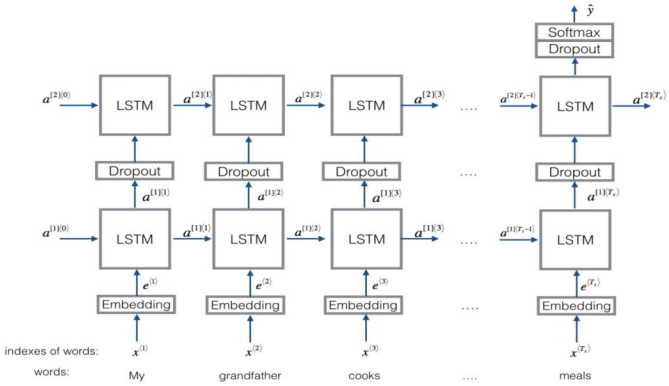

Fig. 3. RNN that takes in word embeddings and feeds it through two layers of 128 LSTM cell before softmax output (courtesy of Ng.[12])

V. Experiments/Results/Discussion

We performed hyperparameter search across the learning rate and the minibatch size for minibatch gradient descent across all of these methods, as these affected our models the most. After searching for the learning rate we chose $\eta = 0.0001$ across all methods, because anything bigger than this particular rate would diverge in reaching the minimum or oscillate frequently but never hit the optimum. Similarly, for our minibatches, we chose a minibatch of size 64 because at this batch size the cost curve produced far less noise than stochastic gradient descent's curve while maintaining a similar convergence time to stochastic gradient descent. Specific hyperparameters will be expounded upon for their respective methods.

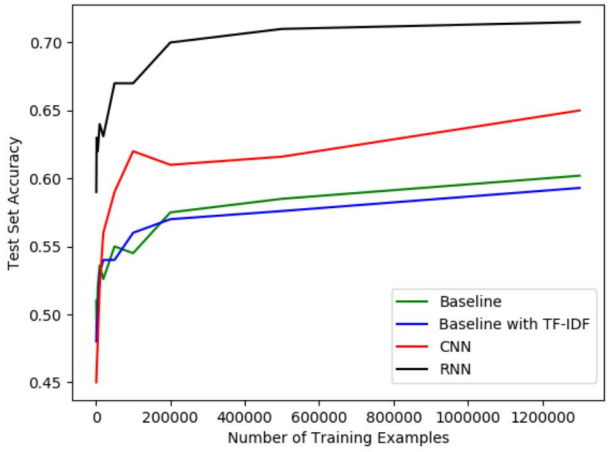

Fig. 4. Learning curves over number of examples for each method

Our primary metrics are precision, accuracy, recall, and F1-scores, which are shown in the table in figure (6). While accuracy is important, we are specifically interested in recall

Baseline Confusion Matrix

Actual \ Predicted	Sarcastic	Not Sarcastic
Sarcastic	0.587	0.413
Not Sarcastic	0.347	0.653

RNN Confusion Matrix

Actual \ Predicted	Sarcastic	Not Sarcastic
Sarcastic	0.695	0.305
Not Sarcastic	0.241	0.749

CNN Confusion Matrix

Actual \ Predicted	Sarcastic	Not Sarcastic
Sarcastic	0.576	0.424
Not Sarcastic	0.267	0.733

Fig. 5. Normalized confusion matrices for each neural network

since recall, in this context, would be defined as the rate at which we correctly classified sarcastic comments out of all sarcastic comments, which is what we wanted to see with our paper. Our experiments consisted of primarily two tests: trying out a good feature representation for the baseline, with TF-IDF inclusion, and testing these varied methods over an increasing number of training examples to see which performs the best.

The baseline performed as expected for a baseline: it had a low precision and recall rate. The accuracy seems higher only because the algorithm was good at classifying non-sarcastic comments. With a recall of 0.587, it does about as good as guessing for sarcastic comments. For this, we chose hyper-parameters of and 2-layers with 25 units each. Adding more layers severely increased the computational time, but without these layers we risked underfitting, so we compromised by creating a small amount of layers with more units. We then added ReLu functions in between the layers because ReLu functions have faster gradient descents than tanh or sigmoid functions. For our first experiment, with TF-IDF, we can see the results in figure (4). Unfortunately, adding the TF-IDF weighting scheme led to worst performance for the baseline. The baseline, without TF-IDF, performed well on sentences with sarcastic exaggerations like “Iron Man 3 was soooo fuuuun” and “Omg no waaaay” that were common to many sarcastically-labeled sentences. However, because they were common across many sarcastic comments the TF-IDF score added to weigh the average sentence vector was somewhat low, which would play less importance to this indicative word. In other words, sarcastic words are not necessarily the most important, so TF-IDF weighting ended up hurting our model.These two methods both failed on sarcastic comments that need context to understand sarcasm, for example “I love Mondays, work is so fun”. “Love” is a normal word usually, but the sarcasm itself is embedded within the word to indicate that it really means “hate”. These baseline methods don’t

Method	Train Acc.	Test Acc.	Precision	Recall	F-1
Standard NN	65.7%	60.2%	60.4%	58.7%	59.6%
NN w/ TFIDF	65.4%	59.3%	58.6%	59.1%	58.8%
CNN	66.7%	65.4%	68.2%	57.6%	62.5%
RNN	75.8%	71.9%	77.7%	69.5%	73.4%

Fig. 6. A list of desired metrics for each of the approaches

properly capture the embedded meaning within the word because averaging word vectors does not properly capture context of the word.

CNNs seemed to do better than our baseline method in this area. We only used 1-layer CNNs because when convoluting a feature map of the same width, it needs to be the same width as the dimension of the word vectors to capture the full embedding. As a result, convolutions will output a single 1-dimensional vector, and at that point there is nothing left to really convolute on without overfitting. With that said, 1 layer might have been too little, so we added feature maps of height 10 (to capture 10 nearby words within a word vector). Afterwords, we used a ReLu activation function before fully connecting the output layer into a network to output softmax probabilities. We chose ReLu specifically because sigmoid and tanh have slower gradient descents, and ReLu tends to work better for faster updates. Interestingly enough, CNNs improved somewhat. The accuracy is higher, but recall is around the same as the baseline, which means that the CNN was better at classifying non-sarcastic sentences rather than sarcastic ones. This was most likely due to the fact that CNNs, with a window of 10 words, used convolutions that were properly able to capture the embedded meanings of words better than what the baseline could do. The same example, “I love Mondays, it’s sooo fun”, was classified correctly, so it probably caught the meaning with its larger feature maps. The CNNs failed on sentences with long sentences and a short window of sarcasm, such as “After not being able to get out of school, which is incredible, I got bullied yet again”. In this sentence, the ”which is incredible” is sarcasm, but it is a small window of sarcasm in a huge amount of text. Using too many feature maps most likely paid less importance to a certain convolution and perhaps captured too many unnecessary convolutions for some sentences. Using a large window for feature maps, as shown from benefits and losses, can be a double-edged sword.

The LSTM RNNs outperformed both of these networks. The hyperparameters chosen for this were 128 cells and 2 layers with some dropout. With limited computing power, we could not try huge amounts of layers, but we risked underfitting again so we put in a large number of cells. Despite this, there still was some overfitting so we put in a dropout probability of 0.5 per layer. We expected this to perform the best though, due to the nature of LSTM cells and how they are able to capture word relationships from far away. We can see from the accuracy, precision and recall especially that RNNs were able to classify better. The recall is high, which means of all sarcastic comments, it was able to accurately identify the sarcastic comments with a rate of 70 percent. As a result,

we can see that LSTM-cell RNNs were able to avoid the same pitfalls of the baseline and the CNN. It was able to capture long term relationships to understand if sarcasm was embedded within a word and was able to avoid capturing too many details and context like CNNs did. The errors made with this model are closely intertwined with the overfitting issues.

These algorithms, as shown with the learning curve in figure (4) when performing the second experiment, did the same at a lower number of examples, gradually improved at different rates over larger numbers of examples. The baseline was outperformed by CNNs at large numbers of training examples, which was outperformed by RNNs. Despite having a dev set, these algorithms suffered from high bias and high variance. The high bias was more or less expected because of the limited number of layers we used and the difficulty of the problem, but the high variance was caused by something more subtle. The dev set was most likely from exactly the same distribution as the test set. Sarcasm can vary over the subreddits: one person in /r/politics may say "Bush is great, he only started a war that put us in massive debt" whereas /r/hiphopheads might say "Kendrick Lamar has no talent whatsoever, he only released grammy winning albums". These sentences and the contexts they were born in are quite different. There is a knowledge disconnect as many sarcasm in /r/politics will be more around politicians being inept and sarcasm in /r/hiphopheads will be more around a commonly known bad rapper being good. The knowledge difference across both of these domains is quite big, despite it being obvious to us, and this causes the overfitting. This difference is actually represented in the dev and tests sets. There are differences in compositions: the dev set has more /r/gaming sarcastic comments but the test set has more /r/politics comments. There are other differences in composition, but these main ones cause a data distribution mismatch that causes overfitting. Despite our efforts with dropout and lesser layers, overfitting still plagued the results.

VI. Conclusion/Future Work

Natural Language Processing, while almost ubiquitous in real world machine learning applications, still remains a difficult topic. Language is not simply black and white: sentiment and tone can very and sentence structures can be convoluted and nuanced. With these subtleties, it makes machine learning applications in natural language processing much harder. With our project, in particular, we wanted to what degree can these subtleties be learned in sarcasm classification. We tried 3 different neural networks: standard neural networks, CNNs, and RNNs. With the standard neural networks, we tried to weigh word vectors with a TF-IDF scheme to see how that compares with the baseline as well. After our experiments, we concluded that, as expected, that LSTM RNNs performed the best, followed by CNNs and lastly our baseline. The TF-IDF weighting normalization did not help, which indicated that important words weren't necessarily indicative of sarcastic words. All of the algorithms suffered from high bias and somewhat high variance, due to the fact that sarcasm can differ across subreddits based on knowledge specific to that domain, like the fact that /r/politics would have a sarcastic comment pertaining to Trump and /r/hiphopheads saying that Kendrick Lamar is a "cheating and talentless hack". We expected LSTM RNNs to perform the best because LSTM RNNs are capable of remembering understanding long term dependencies that the other two methods would fail to notice in a sentence.Similarly, the CNNs outperformed the baseline mostly due to the fact that CNNs convolute across a consecutive sequence of word embeddings of variable length, which most likely captures the relationship of words and dependencies better than the baseline neural network would with a simple average-of-word-embeddings vector. While the results were promising, much could be done to improve this field. We tried some shallower networks, but attempted to broadly try out different types of networks to see what works best for such a task. For future work, deeper hyperparameter searching for each of these networks would most likely yield promising results, especially with trying out deeper networks with more layers and more units. Additionally, we were provided not just the actual comment, but the parent comments and subreddits, so in the future we could figure out how to incorporate these additional features and see if providing context to the sarcastic comment would help classification. Lastly, we could try attention models to develop a proper encoding of a sentence vector or use other sentence embedding libraries like Doc2Vec to properly capture the sentence meaning in a high dimensional space. With these additional steps, we could take one step closer to teaching machines the subtleties of language.

VII. Contributions

Rohan: Worked on the baseline and CNN methods and trained the Word2Vec features
Daniel: Parsed the data and worked on RNNs and the hyperparameter search on learning rate and minibatch gradient descent

References

[1] A. Reyes, P. Rosso, D. Buscaldi. "From humor recognition to irony detection: The figurative language of social media." Data & Knowledge Engineering 74 (2012): 1-12.

[2] D. Bamman, and N. Smith. "Contextualized Sarcasm Detection on Twitter." ICWSM. 2015.

[3] A. Silvio, et al. "Modelling context with user embeddings for sarcasm detection in social media." (2016)

[4] R. Kreuz, and G. Caucci. "Lexical influences on the perception of sarcasm." Proceedings of the Workshop on computational approaches to Figurative Language. Association for Computational Linguistics, 2007.

[5] E. Filatova "Irony and Sarcasm: Corpus Generation and Analysis Using Crowdsourcing." LREC. 2012.

[6] A. Joshi, et al. “Are Word Embedding-based Features Useful for Sarcasm Detection?.” LREC (2016).

[7] Google Brain Team. “Tensorflow Library” 2015.

[8] F. Chollet. “Keras Library” 2015.

[9] R. Rehurek “Gensim Word2vec library” 2015.

[10] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed Representations of Words and Phrases and their Compositionality, NIPS, 2013 p. 1-5

[11] C. Peng, M. Lakis, J. Pan, “Detecting Sarcasm in Text: An Obvious Solution to a Trivial Problem”, December 2015.

[12] A. Ng, “Emojify-V2”, CS230: Deep Learning, March 2018.

[13] D. Ofer, M. Khodak, N. Saunshi “A Large Self-Annotated Corpus for Sarcasm” https://www.kaggle.com/danofer/sarcasm, 2013.

VIII. GITHUB REPOSITORY LINK

Code link: https://github.com/aphex36/SarcasmClassifier