A Survey of Techniques for Sentiment Analysis in Movie Reviews and Deep Stochastic Recurrent Nets

Chase Lochmiller
Department of Computer Science
Stanford University
cjloch11@stanford.edu

Abstract

In this paper, we explore the task of sentiment analysis. We use the dataset provided by Socher et al (2013), which analyzes reviews on Rotten Tomatoes and includes 11,855 sentences and 215,154 unique phrases. We explore a number of different techniques in analyzing the dataset. As a baseline classifier, we implement Naive Bayes and will show that we can see substantial performance gains by encoding more information into our model. We explore both recursive networks that used the parsed tree structure, and build up sentiment predictions from that, as well as recurrent neural networks, which analyze each sentence and phrase as a varying length sequence with a single label. We explore a number of expansions on the vanilla techniques including using GRU's to encode longer term information into each recurrent cell. We also experiment with stochastic depth networks, which train an ensemble of shallow models at train time and use a deep network at test time.

1 Introduction

The task of sentiment analysis has obvious practical applications within industry, ranging from financial firms automatically interpreting news stories and economic data to make fast and automated trading and investment decisions, to social networks using sentiment analysis to extract valuable insight into the mood/opinion of the masses on a particular topic. In this paper, we build on the work of Pang and Lee (2005) and Socher et al (2013) by analyzing the Stanford Sentiment Treebank dataset, which contains movie reviews from Rotten Tomatoes along with labeled sentiment data for full sentences, along with labels for the sub phrases that came out of the parses of each individual sentence. This expanded dataset gives us a total of 215,154 unique labeled phrases in addition to the 11,855 full sentences. This finer grained breakdown of the data has allowed researchers to explore the data in more detail and pick up on features generated from the finer grained sentence constructions and put them to use with more complex modeling techniques. Simultaneously, research in the field of deep learning is massive in scope and has led to many interesting breakthroughs and innovative network structures. We study one such network structure that came from Deep Networks with Stochastic Depth by Huang et al (2016) in the computer vision community. In this paper we will attempt to reproduce the results of some of the canonical approaches that have had success in the past, as well as implementing some new results. We will establish a baseline approach with Naive Bayes to set the bar. We will then implement various neural network structures in the recurrent and the recursive neural network families. We then make some new and innovative modifications to the classical network structures, using the stochastic depth technique introduced by Huang et al (2016), which has only been applied to convolutional neural networks in the context of image classification. We attempt to apply it to both recurrent neural networks and recursive neural networks.

2 Related Work

When Pang and Lee originally introduced the Rotten Tomatoes movie review dataset in 2005, the most popular and effective techniques to capture sentiment were bag of words models. As the name indicates, these types of models analyze the problem by considering each sentence as a bunch of independent words, with no dependencies on word ordering or context. While they can be effective, especially in the presence of numerous superlatives, can miss out on very basic interactions because they do not encode any structure of the word ordering. For example if we see the word "great" in a review, it elicits a strong positive sentiment. However, if that word is preceded by "not", as in "not great", then likely paints a more grim picture than what we had originally thought. Effects like negation are just one of the shortcomings of bag-of-words type models. To overcome this, many took the route of trying to encode different one off rules with hand engineered features, however one quickly realizes many heuristically designed rules can exist for different scenarios. Language is complicated! To overcome this, researchers worked on developing techniques that had feature design embedded in their methodology. In order to make progress in this space, the idea of word vectors, or mapping the text of words to some high dimensional embedded vector space was developed so that non-linear learning techniques like neural networks could have some sort of vector inputs. Socher et al (2011) explored recursive auto-encoders for phrase and sentence representations to predict sentiment. The idea of vector representations of words has since been pushed even further with the development of word2vec (Miklov et al (2013)) and GloVe (Pennington et al (2014)) which are learned word vector representations based upon unsupervised learning techniques over very large corpora. Such representations can encode fundamental relationships between words into a vector space, so that you can have vector representations of words that when combined with standard vector operations like addition yield interesting results. For instance, "Italy - Rome + France = Paris", where we've in a sense encoded a fundamental feature of these words, namely the relationship of a Country and its capital city.

Neural networks and deep learning are incredibly powerful tools, but they are not always straightforward in their application. Three fundamental issues that plague deep neural networks structures are: 1) their potentially long training times 2) exploding/vanishing gradients and 3) their tendency to overfit the training data. Technological advances such as the use of GPU's into the computation process have helped alleviate some of the stress of long training times, but it can still be a problem for very large models with lots of data. In this paper we explore the idea introduced by Huang et al (2016) of training an ensemble of models with stochastic depth. This technique allows us to use shorter models at training time, but still employ a deep model at testing time, thus leading to far shorter training times. The crux of idea is that for a given network, for each layer of our network i, we draw some Bernoulli random variable $d_i \sim Bernoulli(p_s)$, where p_s is the survival probability, a new hyper parameter of the network. If $d_i = 0$, then the activation function of layer i is replaced with the identity function and we feed forward the hidden state from the previous input layer as our output hidden state. So in the previous model, the output of hidden layer i can be expressed as:

$$\mathbf{h_i} = \text{ReLU}(f_i(\mathbf{h}_{i-1}) + \text{id}(h_{i-1}))$$

Training a model with stochastic depth, our hidden state now has an element of randomness depending 0n our our Bernoulli variable. It now becomes:

$$\mathbf{h_i} = \text{ReLU}(d_i f_i(\mathbf{h}_{i-1}) + \text{id}(h_{i-1}))$$

which when $d_i = 0$ just reduces to the identity function of h_{i-1}, and the original network structure when $d_i = 1$. While the original paper by Huang et al, describes this technique in convolutional neural networks in the application of image classification, the idea can be seen graphically in figure 1 for a recurrent neural network. More details on this implementation in a recurrent and a recursive architecture will be covered in the next section. This technique, while simple at the surface, helps with many issues that arise in our (potentially) deep networks. Similar to the popular "Dropout" method introduced by Srivstava and Hinton et al (2014), stochastic depth networks go one step further and instead of stochastically dropping out individual nodes, each Bernoulli random variable is tied to the presence of an entire hidden layer. By doing this, we effectively shorten our network and reduce the risk of vanishing gradients. Additionally, it adds an element of robustness to the model. For a given input, which in our case is a sequence of L different word vectors, there are 2^L different possible networks, of which we sample 1 of these networks and update it. By training with this

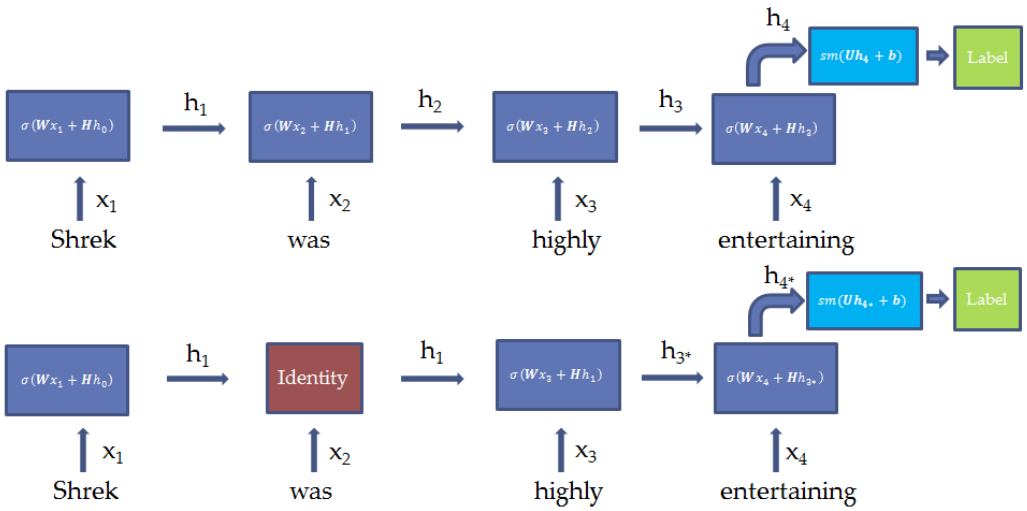

Figure 1: The top network is a vanilla recurrent neural network. Below we see a stochastic depth recurrent neural network, where the second recurrent cell, highlighted in red, has been replaced with the identity function, and it passes through the hidden state from the previous cell.

architecture, we are effectively training an ensemble of different models. Our network structures employed for the Stanford Sentiment treebank were a bit different and we will walk through the details of their implementation with stochastic depth in the next section. Thus the issue of overfitting can be alleviated by various regularization techniques such as adding an L2 penalty on your model weights, training with stochastic depth adds its own element of regularization that helps improve the models ability to generalize well in an out of sample context.

3 Approach: The Models

3.1 Baseline Approach: Naive Bayes

As a baseline classifier we use Naive Bayes. Naive Bayes operates under the conditional independence assumption, that given the class, each of our words are conditionally independent of one another, which we know is not true, as something as simple as the order of the words in a sentence can make a huge difference in the overall sentiment of a particular sentence of phrase. That said, this basic classifier using a bag of words model can achieve impressive results nonetheless, making it a strong baseline metric. Results are introduced in the next section. We are interested in learning $p(C^{(j)}|w_1^{(j)},...,w_n^{(j)})$, where $C^{(j)}$ is the class label of sentence/phrase j and $w_1^{(j)},...,w_n^{(j)}$ are the words that make up sentence/phrase j. By using the chain rule of probability, the definition of conditional probability and the conditional independence assumption, we are able to estimate this conditional probability as:

$$p(C^{(j)}|w_1^{(j)},...,w_n^{(j)}) = \frac{p(C^{(j)})p(w_1^{(j)},...,w_n^{(j)}|C^{(j)})}{p(w_1^{(j)},...,w_n^{(j)})} \quad (1)$$

$$\approx \frac{p(C^{(j)})\prod_{i=1}^{n} p(w_i^{(j)}|C^{(j)})}{p(w_1^{(j)},...,w_n^{(j)})} \quad (2)$$

The conditional independence assumption is what makes Naive Bayes tractable, as otherwise the search space is far too massive.

3.2 Recurrent Neural Networks and GRU's

A recurrent neural network architecture is a commonly used architecture for processing sequences of information. The vanilla recurrent neural network takes an input x_i at each time step i and produces

a hidden state for that time step h_i based upon x_i and h_i. That composition process is typically done through an affine composition that then undergoes some non-linear transformation, with the most popular activation functions being tanh, sigmoid and the rectified linear unit. Formally, for each time step, we have an input x_i and a hidden state that is defined as:

$$h_i = f(Wx_i + Hh_{i-1} + b_1)$$

where f is our activation function and W, H and b are the parameters of our model. Then depending on the task, we use the hidden state as the input to some affine transformation which is then input into a sigmoid function to output the probability distribution of the j^{th} label in our problem, $\hat{y}_j$. In our case, we would like to take the input of an entire phrase/sentence before we output an estimate of the sentiment label, thus each sentence is encoded as a variable length sequence and then a single output vector is generated at the end. Formally, this gives us an expression for $\hat{y}_j$:

$$\hat{y}_j = \text{softmax}(Uh_L + b_2)$$

where L is the length of our sequence. This structure can be seen in figure 1, as the top network in the model. When we actually want to predict the label of a particular sequence, we choose the label with the highest probability estimate in $\hat{y}_j$. This structure can become problematic for considering longer term relationships and managing vanishing gradients for longer sequences. In order to help with this, we also implemented GRU's with our recurrent cells.

A GRU (short for gated recurrent unit) adds to the recurrent neural network model by adding two gates at each time step that help control the flow of information within a sequence. First it introduces an update gate z_i:

$$z_i = \sigma(W^{(z)}x_i + H^{(z)}h_{(i-1)})$$

The second gate is known as the "reset gate" and is defined as:

$$r_i = \sigma(W^{(r)}x_i + H^{(r)}h_{(i-1)})$$

Using these gates, we define the hidden cell state, $\tilde{h}_i$ as:

$$\tilde{h}_i = \tanh(Wx_i + r_i \circ Hh_{(i-1)} + b_1)$$

where $\circ$ is the element-wise product or Hadamard product. However, this hidden state is only the information that is encoded into the cell and the state that actually gets passed to the following cell in the network, h_i is restricted by the update gate and is computed:

$$h_i = z_i \circ h_{(i-1)} + (1 - z_i) \circ \tilde{h}_i$$

This gives us a new cell definition within our network, but the flow of information between cells and in predicting labels remains the same.

Building on this model, we can take things one step further and introduce a stochastic element to our training procedure. At training time, for each node in our network we draw a random Bernoulli variable $d_i \sim \text{Bernoulli}(p_s)$, where p_s is the survival probability of a layer. Similar to the approach taken by Huang et al, we apply this technique to our individual hidden layers, and when d_i is 1, the layer remains unchanged as it was before. However when d_i is 0, we instead pass forward the hidden state of the previous node. In the case of GRU's, this gives us a new output:

$$h_i = d_i(z_i \circ h_{(i-1)} + (1 - z_i) \circ \tilde{h}_i) + (1 - d_i)h_{(i-1)}$$

We can see this clearly in figure 1, where the second network draws a random variable for each node in the network. The second node in our network has been dropped out and we can see that it simply feeds forward the hidden state of the first node. This in effect trains the model on the sentence "Shrek highly entertaining" instead of "Shrek was highly entertaining". On average, this greatly reduces the size of our networks, and leads to substantial speed ups at training time. At test time, we feed forward the entire input, but we must calibrate each input based on its probability of survival during the training process. Thus our test time hidden state output for the vanilla recurrent network is:

$$h_i^{\text{test}} = f(p_s Wx_i + Hh_{i-1} + b_1)$$

And for our GRU network, we get:

$$\tilde{h}_i^{\text{test}} = \tanh(p_s Wx_i + r_i \circ Hh_{(i-1)} + b_1)$$

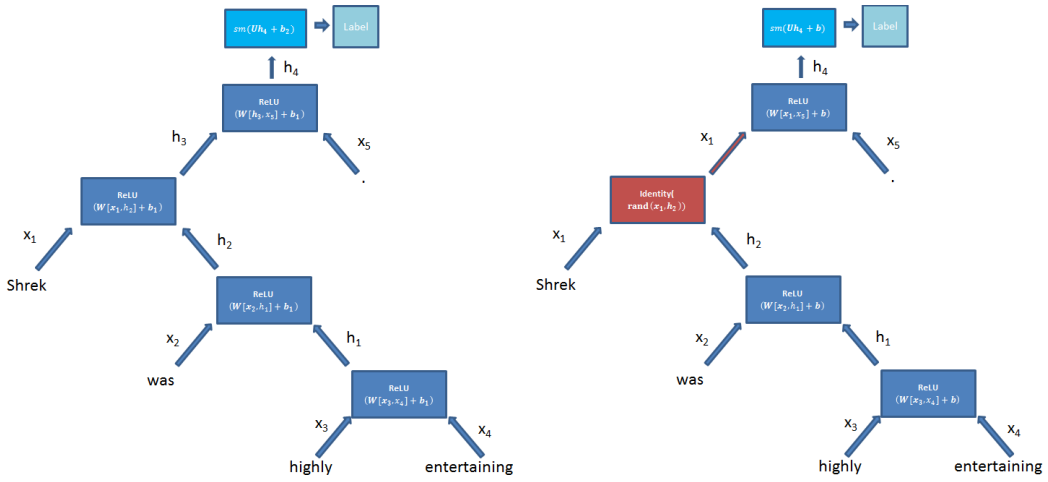

Figure 2: The network on the left is a vanilla recursive neural network. On the right we see a stochastic depth recursive neural network, where one of the nodes, highlighted in red, of our tree has been replaced by an identity function. We randomly decide which child state to pass through to the parent node (in the figure, we have chosen the left child, x_1). In this case, because we have selected x_1, it prunes the entire right sub-tree and greatly reduces the computations that are required.

3.3 Recursive Neural Networks

In one sense, a recursive neural network is a generalization for a recurrent neural network and is a great model for handling variable length sequences. Instead of a linear feed forward network as we did before, we instead encode our network into a binary tree, where the inputs of the sequence are fed into the leaves of our network and each composition layer combines the inputs with an affine transformation fed into a non-linearity.

$$h_i = \text{ReLU}(W[h_{\text{left}}; h_{\text{right}}] + b_1)$$

The matrix in our affine transformation, W, encodes the relationship between two vectors that are mapped to our word vector space and their composition. The hidden states are then propagated up the tree to the parent node and this same transformation is repeated all the way up to the root node. In our case, we use the parse of our sentence that is given to us by the Stanford Parser (Klein and Manning 2003). Every phrase in our sentence is a sub-tree of the full parse, and in the event we would like to make a prediction, we feed the hidden state of the node into an affine transformation and a softmax classifier to give us our probability distribution of labels $\hat{y}_i$

$$\hat{y}_i = \text{softmax}(Uh_i + b_2)$$

Figure 2 gives a graphical representation of our network. As in the recurrent case, we build upon this baseline model by adding in stochastic depth at training time. It is a slightly different representation though in this case. In effect, our implementation is equivalent to randomly pruning branches of the tree at training time. As before, for each node in our tree, we draw a random Bernoulli variable $d_i \sim \text{Bernoulli}(p_s)$, where p_s is the survival probability. When d_i is 1, the node remains unchanged in our tree structure. However when d_i is 0, we prune either the left or the right sub-tree and feed the hidden input state of the other child to the parent node. This is done by generating a second Bernoulli random variable $d_i^{\text{split}} \sim \text{Bernoulli}(0.5)$. More precisely, this gives us the output hidden state:

$$h_i = \text{ReLU}(d_i(W[h_{\text{left}}; h_{\text{right}}] + b_1) + (1 - d_i)(d_i^{\text{split}} h_{\text{left}} + (1 - d_i^{\text{split}})h_{\text{right}}))$$

This network representation is drawn in figure 2 on the right side. We see the node that is highlighted red has drawn a Bernoulli random variable of 0 and thus is pruned. In this case, the result is that our sentence is pruned down to just "Shrek". With this example we see that this technique has the possibility engage in very aggressive pruning of our sentences at training time and thus the survival probability must be adjusted accordingly. We discuss this more in the results section. As before, we

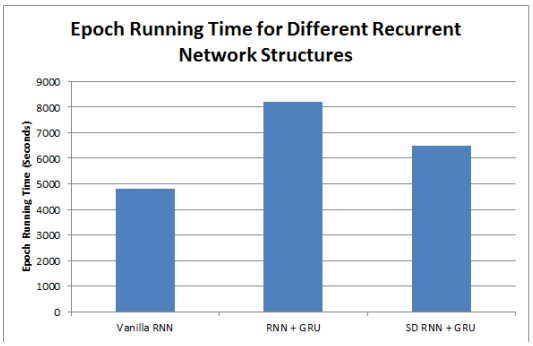

Figure 3: Plot of performance running time for various implementations of the recurrent neural network. We see by using stochastic depth with our GRU model, we are able to shave off about 20% of the running time

also need to make an adjustment to account for each nodes probability of survival during the training process when we feed forward our inputs. This gives us:

$$h_i^{\text{test}} = \text{ReLU}(p_s W[h_{\text{left}}; h_{\text{right}}] + b_1)$$

4 Experiments and Results

Throughout our experimentation, we used the Stanford Sentiment treebank as our dataset to analyze. We implemented all of our techniques using the TensorFlow libraries in python, using a combination of local resources and AWS. To analyze the data, we looked purely at the task of binary classification, as opposed to the fine grained analysis. In doing so, we filtered out all data points with a label of 2 (neutral) in our train, dev and test sets. Beyond exploring the dataset and implementing some of the historical models used to analyze it, we had two primary goals with new innovations in implementing the stochastic depth model into this project. The first was to analyze the running time of the model with and without stochastic depth and to see if we could improve upon it. The second, was to see if we could create a better performing model in test accuracy, by creating a model that is more robust against variations in the training data.
We start with running time. On the surface, this would appear to be the easy objective to knock out. However, due to some complications with tensor flow, our stochastic depth recursive neural network actually ran much slower than the baseline model (about 2-3x depending on the size of training set). We were not able to properly debug this issue, but we think it has something to do with the way tensor flow is managing the computation graph. However, in the case of recurrent neural networks, we were able to achieve much faster training times using the stochastic depth model as opposed to the standard model. The range of the speedup depended on how we set the survival probability, which one would naturally guess. We can see a chart showing the running time of the various models in figure 3, where we see a running time savings of about 20% by using the stochastic depth model on the GRU model compared to the base case GRU. This was with a survival probability set to 50%. We saw about a 50% savings on run time when we cut the survival probability down to 25%.

On the performance side of things, we had the opposite problem. We struggled to get anything to perform well with the recurrent neural network structure, and were faintly able to get anything to train well at all. We hypothesize that this issue was due to some complication with tensor flow and variable sequence inputs. We used the scan() method to process our inputs, and we were able to achieve reasonable training set error ($\sim 80\%$), but we were unable to get our model to make any reasonable progress towards good generalization error on the dev set. Its possible this error arose from our inability to search a wide enough hyper-parameter space, but we think its more likely to be a problem with our implementation somehow which could not be resolved. The recursive neural network models on the other hand we were able to get working well, despite the slow running time of the stochastic depth based models. We tried to cast a wide net to search over in our hyper parameters, but training time was very costly in this model, especially when trained over the entire dataset. Everyone can agree that AWS was probably the big winner here. That said, we were able to yield some interesting results. Scanning over a number of different survival probabilities, we saw

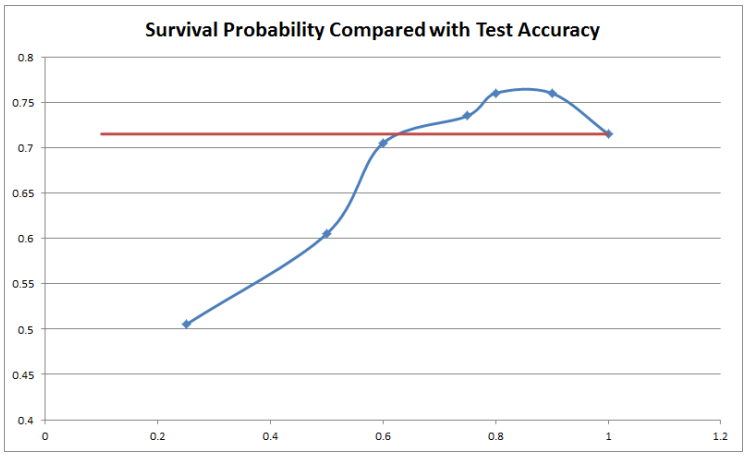

Figure 4: Plot of performance compared with survival probability. This was training our model with a fixed set of hyper parameters apart from survival probability. We see we are able to outperform the baseline, which does not have stochastic depth for a narrow range of probabilities

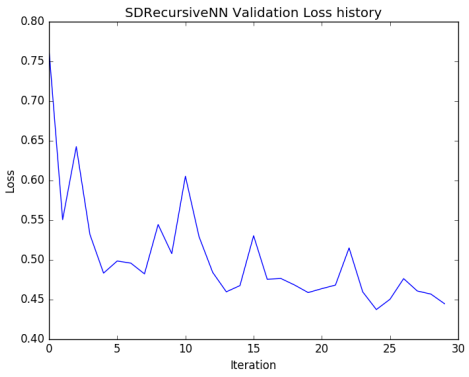

Figure 5: Example plot where we see see training loss continue to trend downward (thought a bit of a volatile trend) for the stochastic depth recursive neural model

that the performance did indeed drop off very dramatically once the survival probability got too low. We conjecture that this is due to the aggressive pruning of the parse tree. Yet we did see that we were able to add value to the performance within a certain range of probabilities, where we were able to outperform the baseline model. We can see that in figure 4, where we plot the performance of the model agains the survival probability.

An unexpected phenomenon we noticed, but were later able to rationalize was in our training of the stochastic depth recursive models, we often saw them converge much slower than the non-stochastic counterparts. Many iterations in, we would still see the dev set loss creeping downward, while for the same iteration on the non-stochastic model the dev set loss was on the steady rise. We suspect that due to the stochastic nature of the model and the way it often crops substantial pieces of our parse trees, that we are left with a very large search space in our ensemble models, and each pass though the data can be very very different. Running for more iterations almost becomes a necessity, which to a certain extent counteracts any gains we had saved on our running time due to dealing with shorter networks. We cannot conclude this for certain, but it was an observation during all of our training. We can see this trend in figure 5.

Sentence Level Binary Classification		
Classifier	Train	Test
Naive Bayes	74.1%	72.9%
Recursive NN	87.2%	76.6%
Recursive NN + SD	90.8%	77.2%
RNN - Vanilla	80.1%	49.1%
RNN - GRU	79.4%	50.9%
RNN - GRU + SD	81.3%	49.8%

Our results can be seen in the table below. While we would have liked to achieve more competitive results, there is a good deal of fine-tuning of the models that needs to be done in order to achieve the best performance for a particular model class. With more time and computation power, we think that would be achievable.

4.1 Qualitative Analysis

While reviewing the performance of the model versus the baseline model, naive bayes, we were very impressed with the ability of naive babes to perform well. It is even able to pick up on negation to a small extent, which we found to be a bit surprising. For instance, the sentence ”You might not buy the ideas.” was correctly classified as negative, when intuitively we would have thought it to be classified as positive in a bag of words model because of the presence of the word ”ideas” (ideas are usually positive, right? I guess maybe not bad ones :). However, it seems the presence of the word ”not” is at least slightly negative and negative enough to overcome the slight positivity of the word ”ideas”, thus it correctly labels the sentence negative. However, we do see cases where our model is able to capture negation properly and the baseline naive bayes model falls short. For example the sentence ”Not a bad journey at all.” Here we have the presence of a clearly very negative sentiment word in ”bad”, however it is negated with the word ”not”. Our model correctly predicts this sentiment as positive, whereas the baseline naive bayes model incorrectly predicts negative. Similarly, ”frenetic but not really funny .” has the word ”funny” in there which again has a positive sentiment as a stand alone word. In this context however it is being negated and the sentence contains a negative sentiment, which our more complex model is able to capture and predict.

4.2 Conclusions

There are a ton of improvements to be made with this model, but it provides a solid foundation. Working with a large dataset made brought the cost of experimentation into perspective, as a single run through the full dataset can be quite costly and time consuming. This made it challenging to get to all of the experiments and iterations on the model that the author would have liked to try. There is a lot of hyper parameter tuning to be done in all of the recursive models as well as refinement of the implementations to get them to run faster and more efficiently. A number of alternative techniques can be tried, expanding on the current baseline. One iteration that we would have liked to try would have been to drop out only leaf layers in our recursive stochastic depth network, as this may have made training with the survival probability a bit more stable. After spending a substantial amount of time analyzing sentiment of reviews, this xkcd comic is the final thought I will leave the reader with. Thanks for a great term!

THE PROBLEM WITH AVERAGING STAR RATINGS

References

[1] Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts (2013) *Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank* Stanford University, Stanford, CA 94305, USA

[2] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, Kilian Weinberger (2016) *Deep Networks with Stochastic Depth* Cornell University, Tsinghua University

[3] B. Pang and L. Lee (2005) *Seeing Stars: Exploiting Relationships for Sentiment Categorization with Respect to Rating Scales*

[4] Richard Socher Jeffrey Pennington Eric H. Huang Andrew Y. Ng Christopher D. Manning (2011) Semi-Supervised Recursive Autoencoders for Predicting Sentiment Distributions Stanford University, Stanford, CA 94305, USA

[5] Tomas Mikolov Ilya Sutskever Kai Chen Greg Corrado Jeffrey Dean (2013) Distributed Representations of Words and Phrases and their Compositionality [6] Jeffrey Pennington, Richard Socher, Christopher Manning (2014)GloVe: Global Vectors for Word Representation

[7] KyungHyun Cho Bart van Merrienboer Dzmitry Bahdanau Fethi Bougares Holger Schwenk Yoshua Bengio (2014) Learning Phrase Representations using RNN Encoder-Decoer for Statistical Machine Translation

[8] Nitish Srivastava Geoffrey Hinton Alex Krizhevsky Ilya Suskever Ruslan Salakhutdinov Dropout: A simple Way to Prevent Neural Networks from Overfitting

[9] Dan Klein and Christopher D. Manning 2003 Accurate Unlexicalized Parsing