

---

# Capstone Documentation

*Release 0.1*

**Kapil Chiravarambath**

**Jan 30, 2019**



# CONTENTS

<b>1</b>	<b>Getting started</b>	<b>3</b>
1.1	Download Source . . . . .	3
1.2	Change directory . . . . .	3
1.3	Create Environment . . . . .	3
1.4	Test Environment . . . . .	3
1.5	Download CoreNLP . . . . .	3
1.6	Start CoreNLP . . . . .	3
1.7	Start Prediction server . . . . .	3
1.8	View Results . . . . .	3
<b>2</b>	<b>Deep learning for Sentiment Analysis</b>	<b>5</b>
2.1	Motivation . . . . .	5
2.2	Problem Description . . . . .	5
2.3	Methodology . . . . .	7
2.4	Data Set Description . . . . .	7
2.5	References . . . . .	7
<b>3</b>	<b>Data Wrangling on Stanford Sentiment Treebank</b>	<b>9</b>
3.1	Data Description . . . . .	9
3.2	Parsing Sentiment Tree . . . . .	9
3.3	Environment Setup . . . . .	9
3.4	Parsing Example . . . . .	10
3.5	Caching the parsed trees . . . . .	11
<b>4</b>	<b>Data Insights on Stanford Sentiment Treebank</b>	<b>13</b>
4.1	Class label distribution in training data. . . . .	13
<b>5</b>	<b>Inferential Statistics</b>	<b>17</b>
5.1	N-gram length vs Sentiment Labels Relationship . . . . .	17
<b>6</b>	<b>Model Description</b>	<b>19</b>
6.1	Loss . . . . .	20
6.2	Back-progagation . . . . .	21
<b>7</b>	<b>Indices and tables</b>	<b>23</b>



Contents:



## GETTING STARTED

### 1.1 Download Source

> *git clone https://github.com/kc3/Springboard.git*

### 1.2 Change directory

> *cd capstone\_1*

### 1.3 Create Environment

> *conda env create -n 'capstone\_1' -f environment-cpu.yml*

### 1.4 Test Environment

> *tox*

### 1.5 Download CoreNLP

> Download and unzip corenlp from <http://nlp.stanford.edu/software/stanford-corenlp-full-2018-10-05.zip>

### 1.6 Start CoreNLP

> *java -mx4g -cp "" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9000 -timeout 15000\**

### 1.7 Start Prediction server

> *python -m src.webapp.webapp*

### 1.8 View Results

> Enter your movie review in the text box to view results!





---

## DEEP LEARNING FOR SENTIMENT ANALYSIS

### 2.1 Motivation

*Sentiment analysis* is a field of study that covers that analyzes people's opinions, appraisals, attitudes, and emotions towards entities such as products, services, organizations, individuals, issues, events, topics, and their attributes.

Although linguistics and natural language processing (NLP) have a long history, little research had been done about people's *opinions* and *sentiments*. Most current techniques (example, search engines work with facts (example, knowledge graph) rather than working with opinions. The advent of social media and availability of huge volumes of opinionated data in these media have caused a resurgence in research in this field.

*Opinions* are key influencers of human behaviors. Businesses and organizations always want to find consumer or public opinions about their products and services. Individual consumers also want to know the opinions of existing users of a product before purchasing it, and peer opinions about political candidates before making a voting decision in a political election. An example of an application would be tracking user opinions about companies to predict their stock prices or box office revenues for a movie.

### 2.2 Problem Description

This capstone project intends to explore recent advances in Natural Language Processing to improve accuracy of sentiment classification. The dataset used for the project would be the Rotten Tomatoes movie reviews dataset. The Rotten Tomatoes movie review dataset is a corpus of movie reviews used for sentiment analysis, originally collected by Pang and Lee [1]. In their work on sentiment treebanks, Socher et al. [2] used the Stanford parser [3] to create fine-grained labels for all parsed phrases in the corpus annotated by three human judges. The Sentiment Treebank along with the sentiment classification can be viewed at [5].

This is a **classification problem** where review phrases are labelled on a scale of five values: *negative*, *somewhat negative*, *neutral*, *somewhat positive*, *positive*. The goal of the project would be to accurately classify the sentiment of a any movie review sentence.

In general, sentiment analysis is investigated at different granularities, namely, *document*, *sentence* or *entity* levels. The review text usually comprises of a single sentence in this dataset, and the data does not contain which movie the review talked about. So, the scope of the project is only restricted to sentence level analysis without entity recognition as such. However, the text might be specific to some aspect of the movie such as its screenplay. For example,

*"The acting was mediocre, but the screenplay was top quality".*

In this case, the screenplay had very positive review (emphasized) while the acting had a very negative review (not emphasized). Aspects make sentiment analysis very domain specific. In general, every domain tends to have a specific vocabulary that cannot be used for other domains. The project scope is only restricted to movie review sentences.

Sentiment words themselves share a *common lexicon*. Example, would be positive words such as great, excellent or amazing and negative words such as bad, mediocre or terrible. Numerous efforts have been made to build such lexicons, which are not sufficient as the problem is more complex.

Some of the obstacles are: \* **Language ambiguity:** Example,

“The movie blows” vs “The movie blows away all expectations”.

The word “blows” has negative orientation in one and positive in other.

- **Contrapositive conjunction:** The review

“The acting sucked but the movie was entertaining”.

Here the review is of the form “X but Y”. The goal of the project would be to   
→ classify each phrase, X and Y accurately and then determine the overall sentiment   
→ for the movie, positive in this case.

- **Sentence negation:** There are two kinds of examples here, \*\* Negative positives:

“The movie was not very great”.

Here the phrase “very great” is positive but “not” changes the sentiment of the   
→ review.

\*\* Negative negatives:

“The movie was not that terrible”.

Here the phrase “terrible” is negative, but “not” does not make it positive.

- **Sarcasm:** Sarcastic comments are hardest to detect and deal with. Example,

“I was never this excited to leave the theater”.

This **is** a very negative comment, but very hard to classify.

- **Sentences with no opinions:** These are usually questions such as

“Has any seen this movie?”

**or** conditionals

“If the acting is good, I will go and see the movie.”

Both are neutral sentences. However **not** all questions **or** conditionals are neutral,   
→ example

“Has anyone else seen this terrible movie?” or

“If you looking for good acting, go and see this movie”.

- **Sentences with no sentiment words but with opinions:** Example,

“The movie solved my insomnia”.

This **is** a very negative review without a sentiment word such **as** good **or** bad.

This project will evaluate the models only on first three of the issues mentioned above. Language ambiguity is mitigated considerably by restricting scope only to movie reviews. The other two issues, contrapositive conjunction and sentence negation are mitigated by constructing parse trees of the text and using compositionality functions trained over known examples, which is what mostly the bulk of this project is about. The sarcastic comments and the sentences with no opinions will be not be specially handled.

## 2.3 Methodology

The goal of the project is to build a sentiment classification system using deep learning, namely Recursive Tensor Neural Network (RNTN). This method uses tensors to remove dependency on the vocabulary and captures different types of associations in the RNN.

The main components of the project would be: \* **Training Engine**: Train the RNTN model with the training data set. \* **Parser**: Parse the trees and extract sentiment labels. \* **Prediction App**: A Web Application to view the predictions of any movie review. \* **Stanford CoreNLP**: The project will reuse Stanford CoreNLP to do constituency parsing of the sentence for which prediction needs to be made.

## 2.4 Data Set Description

The project uses the data set from the original paper as contains fully parsed trees and sentiment labels. The train, test and dev data already split and parsed using the standard parser is exposed at [https://nlp.stanford.edu/sentiment/trainDevTestTrees\\_PTB.zip](https://nlp.stanford.edu/sentiment/trainDevTestTrees_PTB.zip)

In addition, the original data set that the paper [2] uses the following data: <http://nlp.stanford.edu/sentiment/stanfordSentimentTreebank.zip> <http://nlp.stanford.edu/sentiment/stanfordSentimentTreebankRaw.zip>

The data contains raw scores in range (1 to 25) which are mapped to (1 to 5) range for both complete sentences and parsed sub phrases.

## 2.5 References

- [1] Pang and L. Lee. 2005. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In ACL, pages 115–124.
- [2] Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank, Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Chris Manning, Andrew Ng and Chris Potts. Conference on Empirical Methods in Natural Language Processing (EMNLP 2013).
- [3] D. Klein and C. D. Manning. 2003. Accurate unlexicalized parsing. In ACL
- [4] <https://nlp.stanford.edu/sentiment/>
- [5] <https://nlp.stanford.edu/sentiment/treebank.html>
- [6] Sentiment Analysis and Opinion Mining, Bing Liu.



## DATA WRANGLING ON STANFORD SENTIMENT TREEBANK

Report on investigation of the Stanford Sentiment Treebank Dataset.

### 3.1 Data Description

The Stanford Sentiment Treebank Corpus (Socher, 2013) is a standardised dataset that is used in many benchmarks such as GLUE. As such we do not expect to find any data inconsistencies or incomplete or missing data in the datasets.

The Treebank consists of fully labeled parse trees that allows for a complete analysis of the compositional effects of sentiment in language. The corpus is based on the dataset introduced by (Pang and Lee, 2005) and consists of 11,855 single sentences extracted from movie reviews.

The sentences in the treebank were split into a train (8544), dev (1101) and test splits (2210) and these splits are made available with the data release [here](#).

### 3.2 Parsing Sentiment Tree

A typical training sample looks like this:

```
> (3 (2 But) (3 (2 he) (3 (2 somehow) (3 (3 (2 (2 pulls) (2 it)) (1 off)) (2 .))))))
```

One of the main checks on the first examination of data was to make sure that all trees could be parsed into properly formed trees. The tree nodes had to satisfy the following properties: \* Each node was either a leaf or an intermediate node with exactly two children. \* A Leaf Node must have a sentiment label and a word associated with it. \* Leaf Nodes have no children. \* An Intermediate Node must have exactly two children and a sentiment label associated with it. \* Intermediate Nodes do not have any word association.

Tests were written to verify that the entire training dataset satisfied the above properties [test\\_tree.py](#)

### 3.3 Environment Setup

```
from __future__ import print_function

import os
import sys

import numpy as np
import pandas as pd
```

```
PROJ_ROOT = os.pardir
```

```
# Add local python functions
sys.path.append(os.path.join(PROJ_ROOT, "src"))
```

## 3.4 Parsing Example

The following code parses the tree and rewrites it back as a text.

```
from features.tree import Tree
```

```
x = '(3 (2 But) (3 (2 he) (3 (2 somehow) (3 (3 (2 (2 pulls) (2 it)) (1 off)) (2 .))))'
→
t = Tree(x)
print(t.text())
```

```
But he somehow pulls it off .
```

In addition for aiding visualization in flask, a JSON conversion had to be defined.

```
print(json.dumps(t.to_json(), indent=4, sort_keys=True))
```

```
{
  "label": 3,
  "left": {
    "label": 2,
    "left": {},
    "probabilities": null,
    "right": {},
    "word": "But"
  },
  "probabilities": null,
  "right": {
    "label": 3,
    "left": {
      "label": 2,
      "left": {},
      "probabilities": null,
      "right": {},
      "word": "he"
    },
    "probabilities": null,
    "right": {
      "label": 3,
      "left": {
        "label": 2,
        "left": {},
        "probabilities": null,
        "right": {},
        "word": "somehow"
      },
      "probabilities": null,
      "right": {
        "label": 3,
        "left": {
          "label": 3,
          "left": {
```

(continues on next page)

(continued from previous page)

```

        "label": 2,
        "left": {
            "label": 2,
            "left": {},
            "probabilities": null,
            "right": {},
            "word": "pulls"
        },
        "probabilities": null,
        "right": {
            "label": 2,
            "left": {},
            "probabilities": null,
            "right": {},
            "word": "it"
        },
        "word": null
    },
    "probabilities": null,
    "right": {
        "label": 1,
        "left": {},
        "probabilities": null,
        "right": {},
        "word": "off"
    },
    "word": null
},
"probabilities": null,
"right": {
    "label": 2,
    "left": {},
    "probabilities": null,
    "right": {},
    "word": "."
},
"word": null
},
"word": null
},
"word": null
}

```

## 3.5 Caching the parsed trees

To save memory and cpu time on parsing trees a singleton object was defined [DataManager](#)

The parsed trees for all the three datasets (train, dev, test) were generated and the above conditions were checked for using asserts in the code.





## DATA INSIGHTS ON STANFORD SENTIMENT TREEBANK

The Stanford Sentiment Treebank Dataset contains fully labeled parse trees giving us sentiment for each word and well as phrases that can be obtained by performing constituency parsing of the trees.

One of the first goals is to examine how each class labels are distributed throughout the training data set.

### 4.1 Class label distribution in training data.

One of the main goals of the experiment is to model the dataset based on

```
# Imports
import os
import sys
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Set path for models
PROJ_ROOT = os.pardir
sys.path.append(PROJ_ROOT)
from src.features.tree import Tree
from src.models.data_manager import DataManager
```

```
# Function to get class distribution in a node
label_size = 5
def get_num_labels(node):
    """Function to get number of labels of each type under a given tree structure."""
    r = np.zeros(label_size)
    r[node.label] = 1
    if node.isLeaf:
        return r
    else:
        return get_num_labels(node.left) + get_num_labels(node.right) + r

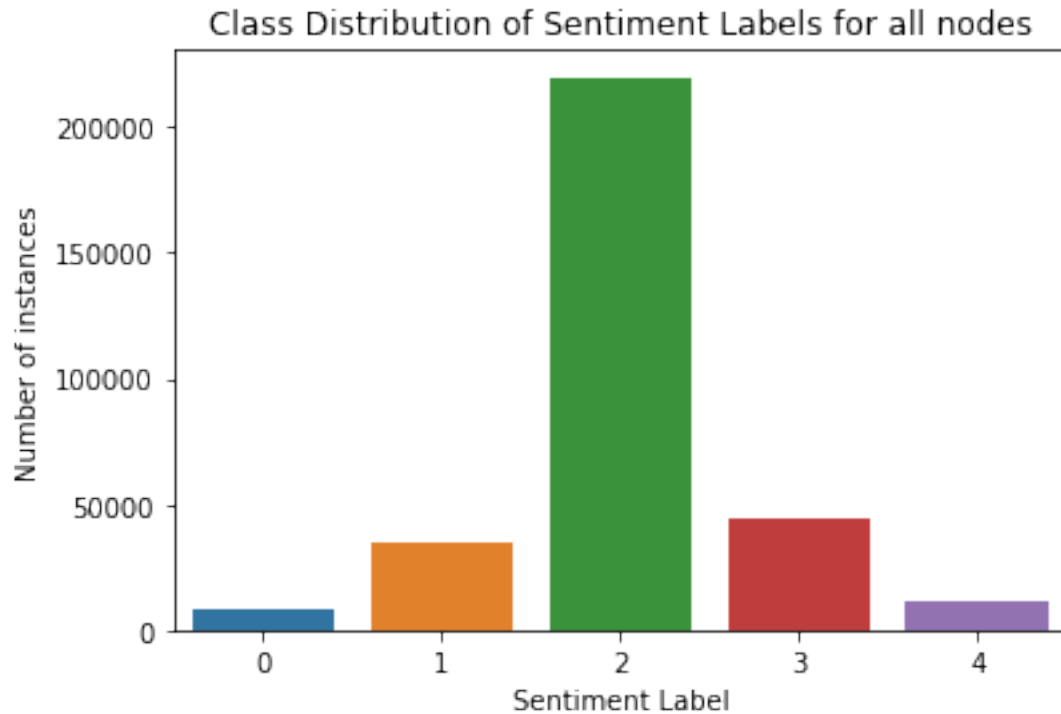
# Get parsed trees
trees_path = '../src/data/interim/trainDevTestTrees_PTB/trees/'
x_train = DataManager(trees_path).x_train

y = np.zeros(label_size)
for i in range(len(x_train)):
    y += get_num_labels(x_train[i].root)

print('Class Distribution of Sentiment Labels: {}'.format(y))
```

```
Class Distribution of Sentiment Labels: [ 8245.  34362. 219788.  44194.  11993.]
```

```
# Plot the distribution
_ = sns.barplot(list(range(5)), y)
_ = plt.xlabel('Sentiment Label')
_ = plt.ylabel('Number of instances')
_ = plt.title('Class Distribution of Sentiment Labels for all nodes')
```



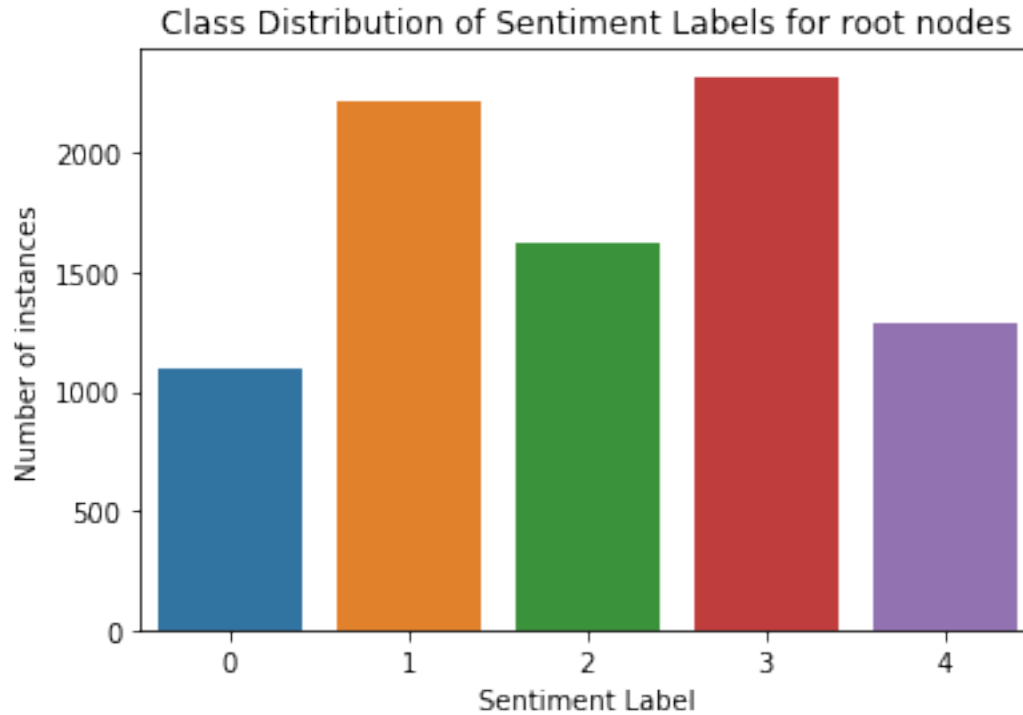
The class distribution is very heavily skewed towards neutral values. This class imbalance will influence the training of neural network and needs to be handled.

```
y_root = np.zeros(label_size)
for i in range(len(x_train)):
    r = np.zeros(label_size)
    r[x_train[i].root.label] = 1
    y_root += r

print('Class Distribution of Root Sentiment Labels: {}'.format(y_root))
```

```
Class Distribution of Root Sentiment Labels: [1092.  2218.  1624.  2322.  1288.]
```

```
# Plot the distribution of nodes for root nodes
_ = sns.barplot(list(range(5)), y_root)
_ = plt.xlabel('Sentiment Label')
_ = plt.ylabel('Number of instances')
_ = plt.title('Class Distribution of Sentiment Labels for root nodes')
```



The nodes for root sentiment labels are more evenly distributed, and the minor class imbalance will again be corrected for training.

```
### Top Positive and Negative words

from collections import defaultdict, Counter

vocab = defaultdict(list)
for i in range(len(x_train)):
    tree = x_train[i]
    stack = [tree.root]
    while stack:
        node = stack.pop()
        if node.isLeaf:
            vocab[node.word].append(node.label)
        else:
            stack.append(node.right)
            stack.append(node.left)

vocab_mean = defaultdict(float)
for k,v in vocab.items():
    vocab_mean[k] = np.mean(v)
```

```
positive_words = [x[0] for x in Counter(vocab_mean).most_common(50)]
print('Most positive words: ' + ','.join(positive_words))
```

```
Most positive words: charming,playful,astonishing,ingeniously,fun,pure,excellent,
↳award-winning,terrific,Freedom,love,Great,creative,humor,great,beautiful,pleasure,
↳better,sweet,perfect,smart,best,happy,funniest,glorious,delightful,honest,joy,
↳masterpiece,fresh,slam-dunk,encourage,entertaining,impressive,brilliantly,shines,
↳powerful,thoughtful,Oscar-worthy,nicest,pretty,remarkable,laughing,marvelous,worthy,
↳laughter,enthralling,captivating,goodies,Oscar-sweeping
```

(continues on next page)

(continued from previous page)

```
negative_words = [i[0] for i in sorted(vocab.items(), key=lambda x: x[1])]
print('Most negative words: ' + ','.join(negative_words[:20]))
```

```
Most negative words: Goddammit, Flawed, artless, bitchy, bruised, negativity, inferior,
↳ disinterest, disappoints, cringe, downer, grotesquely, horrendously, Snide, cold-fish,
↳ dehumanizing, pissed, trash-cinema, car-wreck, stalking
```

The sentiments for both kinds of words match expectations. One of the cross-checks would be to validate the generated word-embeddings against these sentiment values.

## INFERENCE STATISTICS

The Stanford Sentiment Treebank dataset only provides explanatory variables in the form of sentiments associated with words and phrases themselves and the sentence structure that results in the associated sentiment.

Here we examine the relationship between n-gram length and associated Sentiments.

### 5.1 N-gram length vs Sentiment Labels Relationship

The original paper highlights this correlation, repeated here for emphasis. The N-grams length is directly related to the height of the node in the tree and is equal to  $\max(\text{left height}, \text{right height}) + 1$  where height of the leaves is assumed to be 1.

```
# Imports
import os
import sys
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Set path to model code
PROJ_ROOT = os.pardir
sys.path.append(PROJ_ROOT)
from src.features.tree import Tree
from src.models.data_manager import DataManager
```

```
# Function to get a tuple of (root_ngram, [(ngram, sentiment),...])
label_size = 5
def get_ngram_sentiment(node):
    if node.isLeaf:
        return (1, np.asarray([(1, node.label)]))
    else:
        left_h, left_arr = get_ngram_sentiment(node.left)
        right_h, right_arr = get_ngram_sentiment(node.right)
        curr_h = max(left_h, right_h) + 1
        curr_arr = np.concatenate([(curr_h, node.label)], left_arr, right_arr)
        return (curr_h, curr_arr)
```

```
# Get parsed trees
trees_path = '../src/data/interim/trainDevTestTrees_PTB/trees/'
x_train = DataManager(trees_path).x_train
```

```

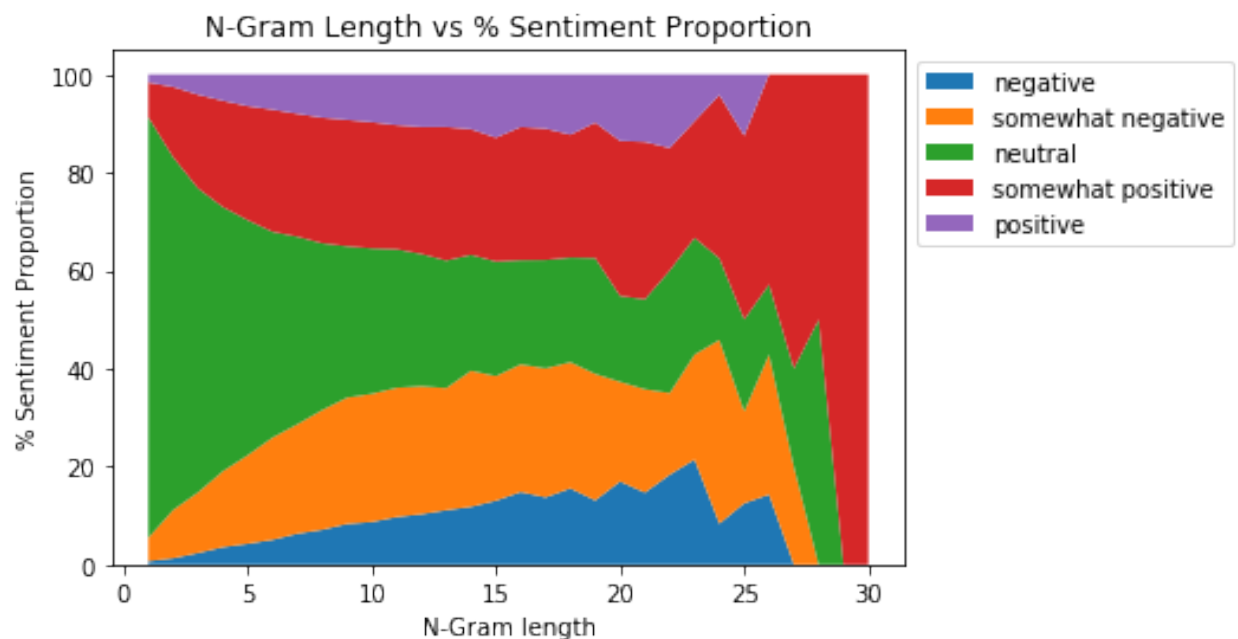
from collections import Counter, defaultdict
max_x = 0
ngrams = defaultdict(list)
for i in range(len(x_train)):
    h, arr = get_ngram_sentiment(x_train[i].root)
    max_x = max(max_x, h)
    for k, v in arr:
        ngrams[k].append(v)

ngram_counts = defaultdict(Counter)
for k, v in ngrams.items():
    ngram_counts[k] = Counter(v)

# Data
x = range(1, max_x+1)
y = list()
for i in range(max_x):
    a = np.zeros(label_size)
    for j in range(label_size):
        a[j] = ngram_counts[i+1][j]
    y.append(a * 100 / sum(a))

# Plot
_ = plt.stackplot(x, np.transpose(y), labels=['negative', 'somewhat negative',
↪ 'neutral', 'somewhat positive', 'positive'])
_ = plt.legend(bbox_to_anchor=(1, 1), loc='upper left', ncol=1)
_ = plt.xlabel('N-Gram length')
_ = plt.ylabel('% Sentiment Proportion')
_ = plt.title('N-Gram Length vs % Sentiment Proportion')
plt.show()

```



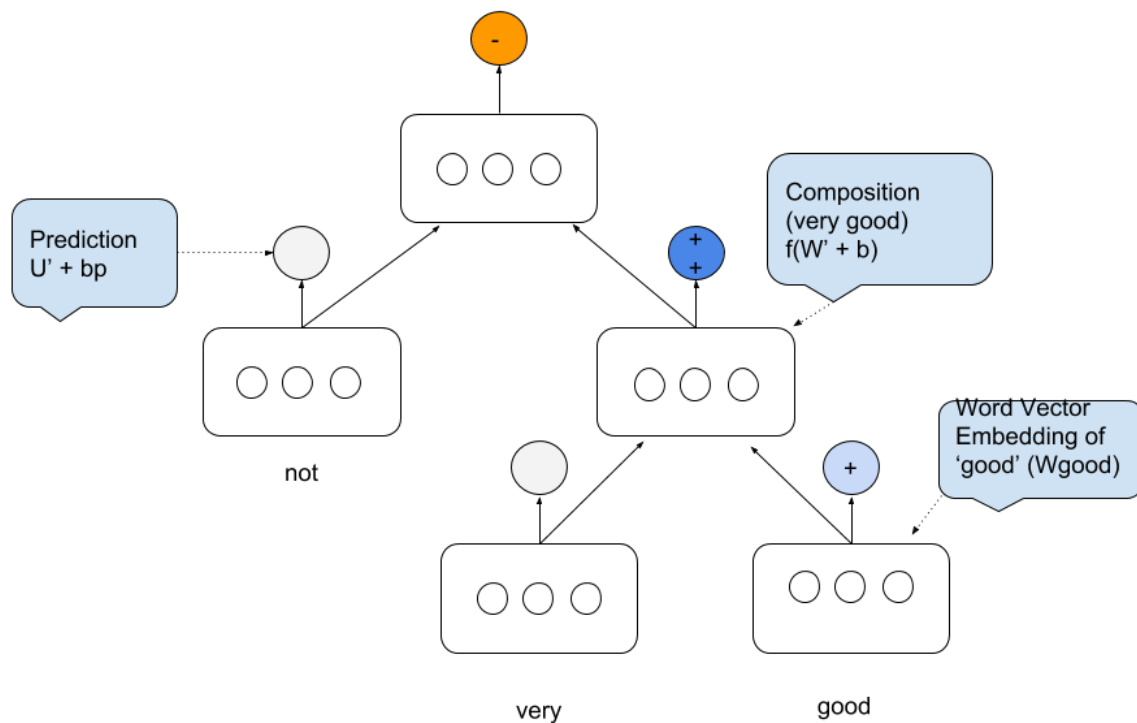
The graph reproduced from the original paper shows that the shorter phrases have mostly neutral sentiment and the longer the phrase, the more likely the sentence will have a positive or a negative sentiment associated with it.

## MODEL DESCRIPTION

During the data exploration phase, we observed that longer n-gram length is associated with a presence of a sentiment. However, bag of words kind of models fail to capture compositional effects associated with sentence structure such as sentence negation. The Recursive Neural Tensor Network Model (RNTN) is a recursive neural network model that captures these compositional effects by relying on constituency parsed representation of the trees.

The model computes word vector representations for each word in the vocabulary and generates similar word representations for intermediate nodes that are recursively used for predictions for the root node.

The following trigram example shows how the prediction occurs at each phase.



Training Example for a trigram (not very good)

Fig. 1: title

Each word vector is represented as a  $d$  dimensional word vector. All the word vectors are stacked in the word embedding matrix  $L$  of dimensions  $[d, V]$ , where  $V$  is the size of the vocabulary. The word vectors are initialized from a random uniform distribution in interval  $[-0.0001, 0.0001]$ , and the  $L$  matrix is seen as a parameter that is trained jointly with the compositionality models.

A word vector for every intermediate node in the tree, that is not a leaf, is a bi-gram and is computed recursively from its children.

The Composition step can be represented by the following equations:

$$zs = W * X + b \text{ (Standard term)}$$

$$zt = X^T * T * X \text{ (Neural Tensor term)}$$

$$a = f(zs + zt) \text{ (Composition function)}$$

where:  $*$   $W$ : Weights to be computed by the model of shape  $[d, 2 \times d]$  for Composition step.  $X$ : Word embeddings for input words stacked together.  $X$  is a column vector of shape  $[2 \times d, 1]$   $b$ : bias term for the node of shape  $[d, 1]$   $T$ : Tensor of dimension  $[2 \times d, 2 \times d, d]$ . Each  $T[:, :, i]$  slice generates a scalar, which is one component of the final word vector of dimension  $d$ .  $f$ : Non-linear function specifying the compositionality of the classifier.  $\tanh$  in this case.

The main benefit of this model is due to the tensor layer. This layer generates internal representations for the most common tree structures, and removes the need to maintain contextual representations for intermediate nodes.

The generated word vectors are used as parameters to optimize and as feature inputs to a softmax classifier to project weights to sentiment probabilities. For classification into five classes, we compute the posterior probability over labels given the word vector via:

$$y = U^T * a + bs$$

where,  $U$ : Sentiment Classification Matrix Weights to be computed by model for Projection Step of shape  $[d, \text{label\_size}]$  where  $\text{label\_size}$  is the number of classes.  $bs$ : Bias term for Projection Step of shape  $[\text{label\_size}, 1]$   $a$ : Output of the composition layer

For the above example, let

$b$  = Word vector embedding of 'very'

$c$  = Word vector embedding of 'good'

$p1$  = Composed Word vector for phrase 'very good'

The vector is composed as:

$$p1 = f \left( \begin{bmatrix} b \\ c \end{bmatrix}^T T^{[1:d]} \begin{bmatrix} b \\ c \end{bmatrix} + W \begin{bmatrix} b \\ c \end{bmatrix} \right)$$

Similarly, the final sentiment of the phrase 'not very good' is computed recursively using word embedding for 'not' and  $p1$ . Let,

$a$  = Word vector embedding of 'not'

$p2$  = Composed Word vector for phrase 'not very good'

$$p2 = f \left( \begin{bmatrix} a \\ p1 \end{bmatrix}^T T^{[1:d]} \begin{bmatrix} a \\ p1 \end{bmatrix} + W \begin{bmatrix} a \\ p1 \end{bmatrix} \right)$$

## 6.1 Loss

The goal of the optimizer is to maximize the prediction or minimize the cross-entropy error between the predicted distribution  $y_i$  and the target distribution  $t_i$ . This is equivalent (up to a constant) to minimizing the KL-divergence



between the two distributions. The error ( $E$ ) as a function of the RNTN parameters,

$$\theta = (L, W, U, T)$$

$$E(\theta) = \sum_i \sum_j t_j^i \log y_j^i + \lambda ||\theta||^2$$

where

$i$  = index of every node in the tree.

$j$  = index of every class in the label.

We use L2 regularizer here as it is more computationally efficient and we do not need feature selection.

The optimization is done using AdaGrad optimizer as it adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features.

## 6.2 Back-progagation

For understanding back-propagation, we start by looking at the derivative of the loss function at Projection and Composition steps for all nodes with respect to parameters ( $U, W, T$ ).

The derivative of the loss function at with respect to  $U$  is the softmax cross-entropy error, which is simple the sum of each node error, that is,

$$\delta_{i,s} = \sum_j U_{ij} (y_j - t_j) f'(x_i)$$

where,

$\delta_{i,s}$  is the softmax error at Projection Layer.

$y_i$  is the ground truth label.

$t_i$  is the predicted softmax probability.

$x_i$  is the vector from the Composition layer.

$f'$  is the derivative of tanh and is given by  $f'(x) = 1 - f(x)^2$ .

$\otimes$  indicates a *Hadamard* product.

Next we look at how error changes with respect to Composition Layer weights  $W$  and  $T$ .

The error due to Composition Layer changes depending on which node we are looking at. For the root node, this value is the softmax error from the Projection Layer. For other nodes, this error can only be computed in a top-down fashion from root node to the leaves.

Let  $\delta_{i,com}$  be the incoming error vector at node  $i$ . For the root node  $\delta_{p2,com} = \delta_{p2,s}$ . This can be used to compute the standard derivative with respect to  $W$  as  $W^T \delta_{p2,com}$ .

Similarly, the derivative with respect to  $T$  can be obtained by looking at each tensor slice for  $k = 1, \dots, d$  as, >

$$\frac{\partial E^{p2}}{\partial V^{[k]}} = \delta_{p2,com} \begin{bmatrix} a \\ p1 \end{bmatrix} \begin{bmatrix} a \\ p1 \end{bmatrix}^T$$

The total derivative for the error with respect to  $W$  and  $T$  at node  $p2$  becomes, >  $\delta_{p2,out} = (W^T \delta_{p2,com} + S) \otimes f' \left( \begin{bmatrix} a \\ p1 \end{bmatrix} \right)$

where,

$$S = \sum_{k=1}^d \delta_{p2,com}^k (V^{[k]} + (V^{[k]})^T) \begin{bmatrix} a \\ p1 \end{bmatrix}$$

The children of  $p2$ , will then each take half of this vector and add their own softmax error message for the complete  $\delta$ . In particular, we have for  $p1$ ,

$$\delta_{p1,com} = \delta(p1, s) + \delta_{p2,out} [d + 1 : 2d],$$

where,

$[d + 1 : 2d]$  represents the vector corresponding to the right child.

The full derivative is the sum of derivatives at all nodes, or

$$\frac{\partial E}{\partial V[k]} = \sum_i \sum_{k=1} d\delta_{i,com}^k.$$

The derivative of  $W$  can be computed in exactly similar way.

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`